Arm® Development Studio Morello Edition

Version 2020.1M0

User Guide

arm

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
102234_2020.1M0_00_en

Arm® Development Studio Morello Edition

User Guide

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
2020.1M0-00	29 October 2020	Non-Confidential	First release for Arm Development Studio Morello Edition

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 click through or signed written agreement covering this document with Arm, then the click through or 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.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US 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/company/policies/trademarks*.

Copyright © 2020 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)

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

developer.arm.com

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

Arm® Development Studio Morello Edition User Guide

Preface

About this book 14

Chapter 1 Debugging Embedded Systems

1.1 About endianness 1-17
1.2 About accessing AHB, APB, and AXI buses 1-18
1.3 About virtual and physical memory 1-19
1.4 About address spaces 1-20
1.5 About debugging hypervisors 1-21
1.6 About debugging big.LITTLE™ systems 1-22
1.7 About debugging bare-metal symmetric multiprocessing systems 1-23
1.8 About debugging multi-threaded applications 1-25
1.9 About debugging shared libraries 1-26
1.10 About debugging TrustZone enabled targets 1-28
1.11 About debugging a Unified Extensible Firmware Interface (UEFI) 1-30
1.12 About debugging MMUs 1-31
1.13 About debugging caches 1-33
1.14 About Arm® Debugger support for overlays 1-36
1.15 Debugging a loadable kernel module 1-37
1.16 Useful commands for debugging a kernel module 1-41

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 2 Controlling Target Execution

2.1 Overview: Breakpoints and Watchpoints 2-43
2.2 Running, stopping, and stepping through an application 2-45
2.3 Working with breakpoints 2-47
2.4 Working with watchpoints 2-48
2.5 Importing and exporting breakpoints and watchpoints 2-50
2.6 Viewing the properties of a breakpoint or a watchpoint 2-51
2.7 Associating debug scripts to breakpoints 2-53
2.8 Conditional breakpoints 2-54
2.9 Assigning conditions to an existing breakpoint 2-55
2.10 Conditional watchpoints 2-57
2.11 Assigning conditions to an existing watchpoint 2-58
2.12 Pending breakpoints and watchpoints 2-59
2.13 Setting a tracepoint 2-61
2.14 Handling UNIX signals 2-62
2.15 Handling processor exceptions 2-64
2.16 Using semihosting to access resources on the host computer 2-66
2.17 Working with semihosting 2-68
2.18 Configuring the debugger path substitution rules 2-70

Chapter 3 Examining the Target

3.1 Examining the target execution environment 3-74
3.2 Examining the call stack 3-75
3.3 About trace support 3-76
3.4 About post-mortem debugging of trace data 3-79

Chapter 4 Running Arm® Debugger from the operating system command-line or from a script

4.1 Overview: Running Arm® Debugger from the command-line or from a script 4-81
4.2 Command-line debugger options 4-82
4.3 Run Arm® Development Studio IDE from the command-line to clean and build your projects 4-89
4.4 Running a debug session from a script 4-91
4.5 Specifying a custom configuration database using the command-line 4-93
4.6 Capturing trace data using the command-line debugger 4-95
4.7 Working with the debug server 4-97
4.8 Arm® Debugger command-line console keyboard shortcuts 4-99

Chapter 5 Debugging with Scripts

5.1 Exporting Arm® Debugger commands generated during a debug session 5-101
5.2 Creating an Arm® Debugger script 5-102
5.3 Creating a CMM-style script 5-103
5.4 Support for importing and translating CMM scripts 5-104
5.5 About Jython scripts 5-106
5.6 Jython script concepts and interfaces 5-108
5.7 Creating Jython projects in Arm® Development Studio 5-110
5.8 Creating a Jython script 5-114
5.9 Running a script 5-116
5.10 Use case scripts 5-118
5.11 Metadata for use case scripts 5-119

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.12 Definition block for use case scripts 5-120
5.13 Defining the Run method for use case scripts 5-122
5.14 Defining the options for use case scripts 5-123
5.15 Defining the validation method for use case scripts 5-126
5.16 Example use case script definition 5-127
5.17 Multiple use cases in a single script 5-128
5.18 usecase list command 5-129
5.19 usecase help command 5-130
5.20 usecase run command 5-131

Chapter 6 Perspectives and Views

6.1 App Console view 6-135
6.2 Arm Asm Info view 6-137
6.3 Arm assembler editor 6-138
6.4 Breakpoints view 6-140
6.5 C/C++ editor 6-144
6.6 Commands view 6-148
6.7 Debug Control view 6-151
6.8 Stack view 6-155
6.9 Disassembly view 6-158
6.10 Events view 6-162
6.11 Event Viewer Settings dialog box 6-165
6.12 Expressions view 6-169
6.13 Expression Inspector 6-173
6.14 Functions view 6-174
6.15 History view 6-177
6.16 Memory view 6-179
6.17 MMU/MPU view 6-188
6.18 Modules view 6-193
6.19 Registers view 6-195
6.20 NVIC Registers view 6-202
6.21 OS Data view 6-205
6.22 Overlays view 6-207
6.23 Cache Data view 6-208
6.24 Screen view 6-210
6.25 Scripts view 6-213
6.26 Target Console view 6-217
6.27 Target view 6-218
6.28 Trace view 6-220
6.29 Trace Control view 6-225
6.30 Variables view 6-228
6.31 Timed Auto-Refresh Properties dialog box 6-234
6.32 Memory Exporter dialog box 6-235
6.33 Memory Importer dialog box 6-236
6.34 Fill Memory dialog box 6-237
6.35 Export Trace Report dialog box 6-238
6.36 Trace Dump dialog box 6-240
6.37 Breakpoint Properties dialog box 6-242
6.38 Watchpoint Properties dialog box 6-245
6.39 Tracepoint Properties dialog box 6-247

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.40 Manage Signals dialog box 6-248
6.41 Functions Filter dialog box 6-249
6.42 Script Parameters dialog box 6-250
6.43 Debug Configurations - Connection tab 6-251
6.44 Debug Configurations - Files tab 6-254
6.45 Debug Configurations - Debugger tab 6-258
6.46 Debug Configurations - OS Awareness tab 6-261
6.47 Debug Configurations - Arguments tab 6-262
6.48 Debug Configurations - Environment tab 6-264
6.49 Debug Configurations - Export tab 6-266
6.50 DTSL Configuration Editor dialog box 6-267
6.51 Probe Configuration dialog box 6-269
6.52 About the Remote System Explorer 6-270
6.53 Remote Systems view 6-271
6.54 Remote System Details view 6-272
6.55 Target management terminal for serial and SSH connections 6-273
6.56 Remote Scratchpad view 6-274
6.57 Remote Systems terminal for SSH connections 6-275
6.58 Terminal Settings dialog box 6-276
6.59 Debug Hardware Configure IP view 6-280
6.60 Debug Hardware Firmware Installer view 6-282
6.61 Connection Browser dialog box 6-285
6.62 Preferences dialog box 6-286
6.63 Properties dialog box 6-288

Chapter 7 Reference

7.1 About loading an image on to the target 7-291
7.2 About loading debug information into the debugger 7-292
7.3 About passing arguments to main() 7-294
7.4 Updating multiple debug hardware units 7-295
7.5 Standards compliance in Arm® Debugger 7-296

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

List of Figures
Arm® Development Studio Morello Edition User Guide

Figure 1-1 Threading call stacks in the Debug Control view 1-25
Figure 1-2 Adding individual shared library files 1-26
Figure 1-3 Modifying the shared library search paths 1-27
Figure 1-4 Cache Data view (showing L1 TLB cache) 1-33
Figure 1-5 DTSL Configuration Editor (Shown with cache read option enabled) 1-34
Figure 1-6 Typical connection settings for a Linux kernel/Device Driver Debug 1-38
Figure 1-7 Typical Files settings for a Linux kernel/Device Driver Debug 1-39
Figure 2-1 Debug Control view 2-45
Figure 2-2 Viewing breakpoints 2-47
Figure 2-3 Setting a watchpoint on a data symbol 2-48
Figure 2-4 Import and export breakpoints and watchpoints 2-50
Figure 2-5 Viewing the properties of a breakpoint 2-51
Figure 2-6 Watchpoint Properties 2-52
Figure 2-7 Breakpoint Properties dialog box 2-55
Figure 2-8 Watchpoint Properties dialog box 2-58
Figure 2-9 Manage signals dialog box (UNIX signals) 2-62
Figure 2-10 Manage Signals dialog box 2-64
Figure 2-11 Typical layout between top of memory, stack, and heap 2-66
Figure 2-12 Set Path Substitution 2-70
Figure 2-13 Path Substitution dialog box 2-71
Figure 2-14 Edit Substitute Path dialog box 2-71
Figure 3-1 Target execution environment 3-74

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure 3-2 Stack view showing information for a selectedcore ... 3-75
Figure 4-1 Enable trace in the DTSL options ... 4-95
Figure 4-2 Command-line debugger connection with DTSL options enabled. ... 4-96
Figure 5-1 Commands generated during a debug session ... 5-101
Figure 5-2 PyDev project wizard ... 5-110
Figure 5-3 PyDev project settings ... 5-112
Figure 5-4 Jython auto-completion and help ... 5-114
Figure 5-5 Scripts view ... 5-116
Figure 6-1 App Console view ... 6-135
Figure 6-2 Arm Asm Info pop-up ... 6-137
Figure 6-3 Arm Asm Info view ... 6-137
Figure 6-4 Assembler editor ... 6-138
Figure 6-5 Breakpoints view showing breakpoints and sub-breakpoints ... 6-140
Figure 6-6 C/C++ editor ... 6-144
Figure 6-7 Show disassembly for selected source line ... 6-146
Figure 6-8 Inspect the value of the highlighted variable ... 6-147
Figure 6-9 Commands view ... 6-148
Figure 6-10 Debug Control view ... 6-151
Figure 6-11 Show in Stack ... 6-155
Figure 6-12 Stack view showing information for a selected core ... 6-155
Figure 6-13 Stack view locked to a selected context ... 6-156
Figure 6-14 Disassembly view ... 6-158
Figure 6-15 Events view (Shown with all ports enabled for an ETB:ITM trace source) ... 6-162
Figure 6-16 Event Viewer Settings (Shown with all Masters and Channels enabled for an ETR:STM trace source) ... 6-166
Figure 6-17 Events view - Live Decode tab ... 6-167
Figure 6-18 Expressions view ... 6-169
Figure 6-19 Inspect the value of the highlighted variable ... 6-173
Figure 6-20 Functions view ... 6-174
Figure 6-21 History view ... 6-177
Figure 6-22 Memory view ... 6-179
Figure 6-23 Memory view with details panel ... 6-181
Figure 6-24 Memory view with Show Cache option enabled ... 6-182
Figure 6-25 Memory View preferences ... 6-184
Figure 6-26 Show Characters column in the Memory view ... 6-186
Figure 6-27 Select Characters from context menu to show the ASCII characters ... 6-186
Figure 6-28 MMU/MPU Translation tab view ... 6-188
Figure 6-29 MMU/MPU Tables tab view ... 6-189
Figure 6-30 MMU/MPU Memory Map tab view ... 6-190
Figure 6-31 MMU settings ... 6-191
Figure 6-32 Modules view showing shared libraries ... 6-193
Figure 6-33 Registers view (with all columns displayed) ... 6-195
Figure 6-34 Registers - CP ... 6-196
Figure 6-35 Registers access rights ... 6-198
Figure 6-36 NVIC Registers view ... 6-203
Figure 6-37 OS Data view (showing Keil CMSIS-RTOS RTX Tasks) ... 6-205
Figure 6-38 Overlays view ... 6-207
Figure 6-39 Cache Data view (showing L1 TLB cache) ... 6-208
Figure 6-40 Screen buffer parameters ... 6-210
Figure 6-41 Screen view ... 6-211

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure 6-42 Scripts view 6-213
Figure 6-43 Recent scripts 6-215
Figure 6-44 Add to favorites 6-215
Figure 6-45 Remove from favorites 6-215
Figure 6-46 Running scripts from the Commands view 6-216
Figure 6-47 Target view 6-218
Figure 6-48 Trace view with a scale of 100:1 6-220
Figure 6-49 Trace view for Cortex-M3 Thumb instructions 6-224
Figure 6-50 Trace Control view 6-225
Figure 6-51 Variables view 6-228
Figure 6-52 How to add variables to the view 6-228
Figure 6-53 Add Global Variables dialog box 6-229
Figure 6-54 Timed Auto-Refresh properties dialog box 6-234
Figure 6-55 Memory Exporter dialog box 6-235
Figure 6-56 Memory Importer dialog box 6-236
Figure 6-57 Fill Memory dialog box 6-237
Figure 6-58 Export Trace Report dialog box 6-239
Figure 6-59 Trace Dump dialog box 6-241
Figure 6-60 Breakpoint properties dialog box 6-242
Figure 6-61 Watchpoint Properties dialog box 6-245
Figure 6-62 Tracepoint Properties dialog box 6-247
Figure 6-63 Manage Signals dialog box 6-248
Figure 6-64 Function filter dialog box 6-249
Figure 6-65 Script Parameters dialog box 6-250
Figure 6-66 Connection tab 6-251
Figure 6-67 Files tab (Shown with file system configuration for an application on a Fixed Virtual Platform) 6-254
Figure 6-68 Debugger tab (Shown with settings for application starting point and search paths) 6-258
Figure 6-69 OS Awareness tab 6-261
Figure 6-70 Arguments tab 6-262
Figure 6-71 Environment tab (Shown with environment variables configured for a Fixed Virtual Platform) 6-264
Figure 6-72 New Environment Variable dialog box 6-265
Figure 6-73 Export tab 6-266
Figure 6-74 Configuration Editor (Shown with Trace capture method set to DSTREAM) 6-268
Figure 6-75 Probe Configuration dialog box 6-269
Figure 6-76 Remote Systems view 6-271
Figure 6-77 Remote System Details view 6-272
Figure 6-78 Terminal view 6-273
Figure 6-79 Remote Scratchpad 6-274
Figure 6-80 Remote Systems terminal 6-275
Figure 6-81 Terminal Settings (Telnet) dialog box 6-276
Figure 6-82 Terminal Settings (SSH) dialog box 6-277
Figure 6-83 Terminal Settings (Serial) dialog box 6-278
Figure 6-84 Debug Hardware Configure IP view 6-280
Figure 6-85 Debug Hardware Firmware Installer view 6-282
Figure 6-86 Connection Browser (Showing a USB connected DSTREAM) 6-285
Figure 6-87 Window preferences dialog box 6-286
Figure 6-88 Project properties dialog box 6-288
Figure 7-1 Load File dialog box 7-291

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure 7-2 Load additional debug information dialog box 7-293

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

List of Tables
Arm® Development Studio Morello Edition User Guide

Table 4-1 *Arm DS IDE arguments* *4-89*
Table 6-1 *Function icons* *6-175*
Table 6-2 *Files tab options available for each Debug operation* *6-255*
Table 6-3 *Debug Hardware Configure IP view contents* *6-281*

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

This preface introduces the *Arm® Development Studio Morello Edition User Guide*.

It contains the following:

- *About this book* on page 14.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this book

This book describes how to use the debugger to debug bare-metal and Linux platforms.

Using this book

This book is organized into the following chapters:

Chapter 1 Debugging Embedded Systems
Gives an introduction to debugging embedded systems.

Chapter 2 Controlling Target Execution
Describes how to control the target when certain events occur or when certain conditions are met.

Chapter 3 Examining the Target
This chapter describes how to examine registers, variables, memory, and the call stack.

Chapter 4 Running Arm® Debugger from the operating system command-line or from a script
This chapter describes how to use Arm Debugger from the operating system command-line or from a script.

Chapter 5 Debugging with Scripts
Describes how to use scripts containing debugger commands to enable you to automate debugging operations.

Chapter 6 Perspectives and Views
Describes the perspective and related views in the Integrated Development Environment (IDE).

Chapter 7 Reference
Lists other information that might be useful when working with Arm Debugger.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`<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 *Arm Development Studio Morello Edition User Guide*.
- The number 102234_2020.1M0_00_en.
- If applicable, 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® Developer*.
- *Arm® Documentation*.
- *Arm Morello Program*.
- *Arm Morello Program Community*.
- *Arm® Glossary*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 1
Debugging Embedded Systems

Gives an introduction to debugging embedded systems.

It contains the following sections:

- *1.1 About endianness* on page 1-17.
- *1.2 About accessing AHB, APB, and AXI buses* on page 1-18.
- *1.3 About virtual and physical memory* on page 1-19.
- *1.4 About address spaces* on page 1-20.
- *1.5 About debugging hypervisors* on page 1-21.
- *1.6 About debugging big.LITTLE™ systems* on page 1-22.
- *1.7 About debugging bare-metal symmetric multiprocessing systems* on page 1-23.
- *1.8 About debugging multi-threaded applications* on page 1-25.
- *1.9 About debugging shared libraries* on page 1-26.
- *1.10 About debugging TrustZone enabled targets* on page 1-28.
- *1.11 About debugging a Unified Extensible Firmware Interface (UEFI)* on page 1-30.
- *1.12 About debugging MMUs* on page 1-31.
- *1.13 About debugging caches* on page 1-33.
- *1.14 About Arm® Debugger support for overlays* on page 1-36.
- *1.15 Debugging a loadable kernel module* on page 1-37.
- *1.16 Useful commands for debugging a kernel module* on page 1-41.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.1 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

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.2 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 Arm 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 irrespective of whether the processor is running or halted.

Within a debug session in Arm 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
```

When using address prefixes in expressions, you can also use address space parameters to specify additional behavior. See *Address space prefixes* for information on how to do this.

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 concepts

1.4 About address spaces on page 1-20

Related references

6.6 Commands view on page 6-148

Related information

Arm Debugger commands

Address space prefixes

info memory command

info memory-parameters command

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.3 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 also 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

- Processors that are compliant with Arm Architectures prior to Armv6 do not support physical addressing in this manner. This includes the Arm7™ and Arm9™ family of processors.
- Physical address access is not enabled for all operations. For example, the Arm hardware does not support 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

6.6 Commands view on page 6-148

Related information

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.4 About address spaces

An address space is a region of memory that is defined by specific attributes. For example, a memory region can be Secure or Non-Secure.

You can refer to different address spaces in Arm Debugger using address space prefixes. These can be used for various debugging activities, such as:

- Setting a breakpoint in a specific memory space.
- Reading or writing memory.
- Loading symbols associated with a specific memory space.

Note

See *Address space prefixes* for information on how to use an address space prefix with the debug commands.

Arm Debugger also uses these prefixes when reporting the current memory space where the execution stopped, for example:

- For address spaces in Armv7-based processors:

  `Execution stopped in SVC mode at S:0x80000000`

  `Execution stopped in SYS mode at breakpoint 1: S:0x80000BA8`.
- For address spaces in Armv8-based processors:

  `Execution stopped in EL3h mode at: EL3:0x0000000080001500`

  `Execution stopped in EL1h mode at breakpoint 2.2: EL1N:0x0000000080000F6C`

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

1.2 About accessing AHB, APB, and AXI buses on page 1-18
1.3 About virtual and physical memory on page 1-19
1.5 About debugging hypervisors on page 1-21

Related information

Address space prefixes

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.5 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.

Arm 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

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 Arm Development Studio scripts using the `$vmid` debugger variable.

Related references

6.6 Commands view on page 6-148

Related information

Arm Debugger Commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.6 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 Arm 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 Arm 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

1.7 About debugging bare-metal symmetric multiprocessing systems on page 1-23

Related references

6.6 Commands view on page 6-148

Related information

Arm Debugger Commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.7 About debugging bare-metal symmetric multiprocessing systems

Arm 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 box. 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.

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

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

1.6 About debugging big.LITTLE™ systems on page 1-22
7.1 About loading an image on to the target on page 7-291
7.2 About loading debug information into the debugger on page 7-292

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
2.2 Running, stopping, and stepping through an application on page 2-45
6.4 Breakpoints view on page 6-140
6.6 Commands view on page 6-148
6.9 Disassembly view on page 6-158
6.16 Memory view on page 6-179
6.18 Modules view on page 6-193
6.19 Registers view on page 6-195
6.30 Variables view on page 6-228

Related information

Arm Debugger Commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.8 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), for example:

```
os_idle_demon #3 stopped (USR) (ID 255)
```

where `#3` is the unique ID used by the debugger, `(USR)` indicates the user mode, and `ID 255` 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 **Development Studio** perspective are associated with the selected stack frame and are updated when you select another frame.

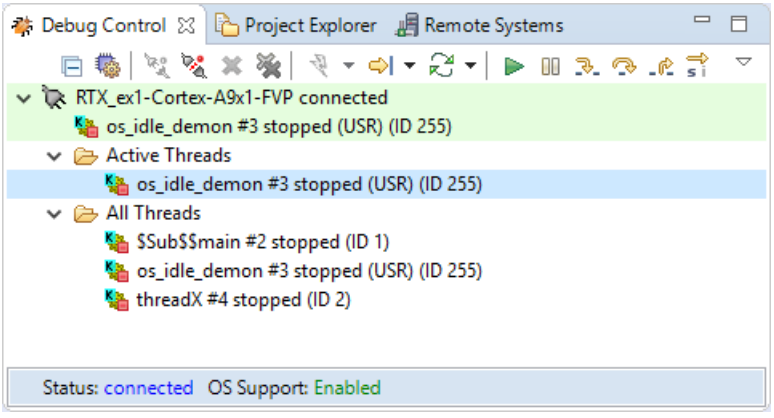


Figure 1-1 Threading call stacks in the Debug Control view

Related references

6.4 Breakpoints view on page 6-140
6.6 Commands view on page 6-148
6.9 Disassembly view on page 6-158
6.16 Memory view on page 6-179
6.18 Modules view on page 6-193
6.19 Registers view on page 6-195
6.30 Variables view on page 6-228

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.9 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.

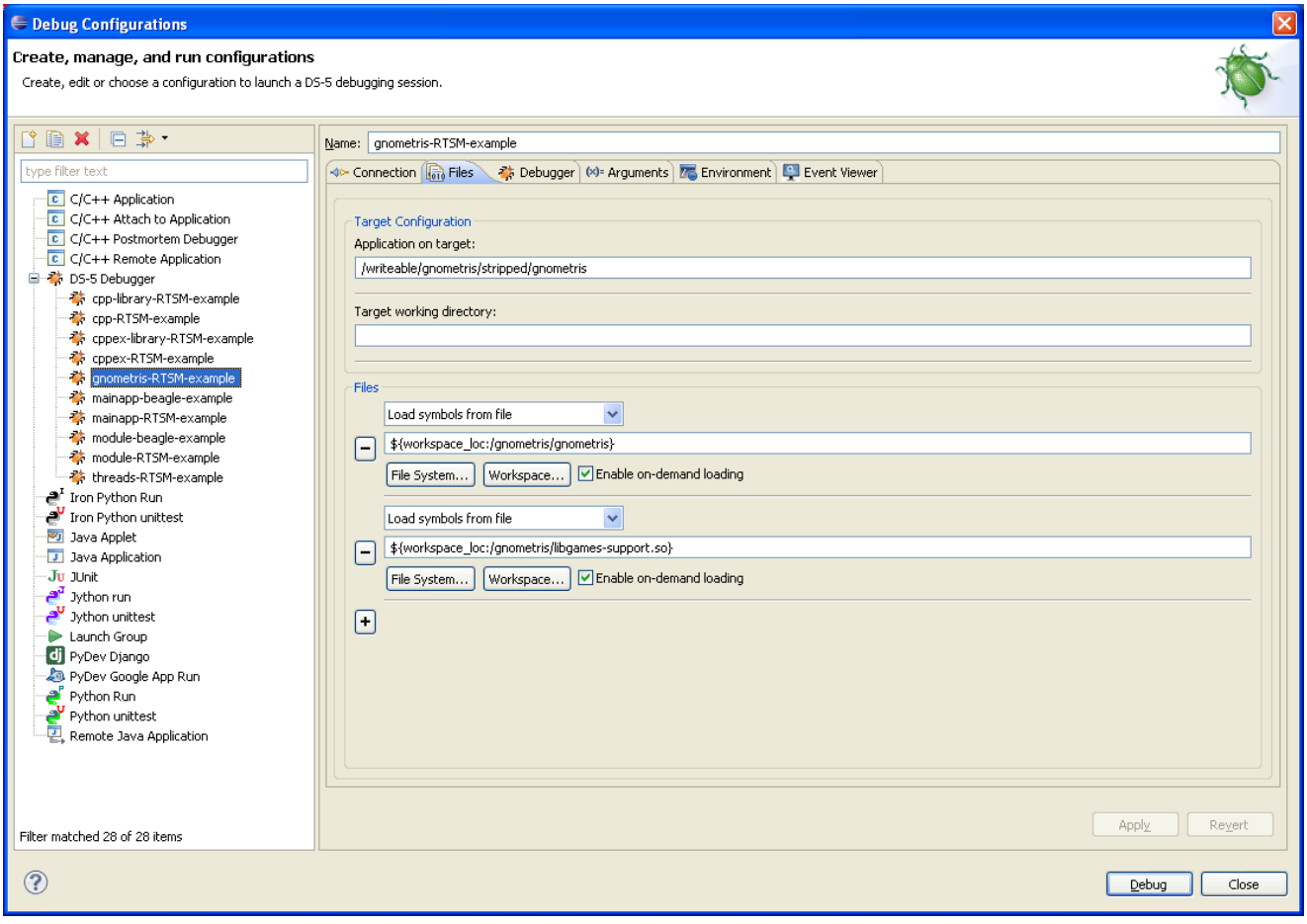


Figure 1-2 Adding individual shared library files

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

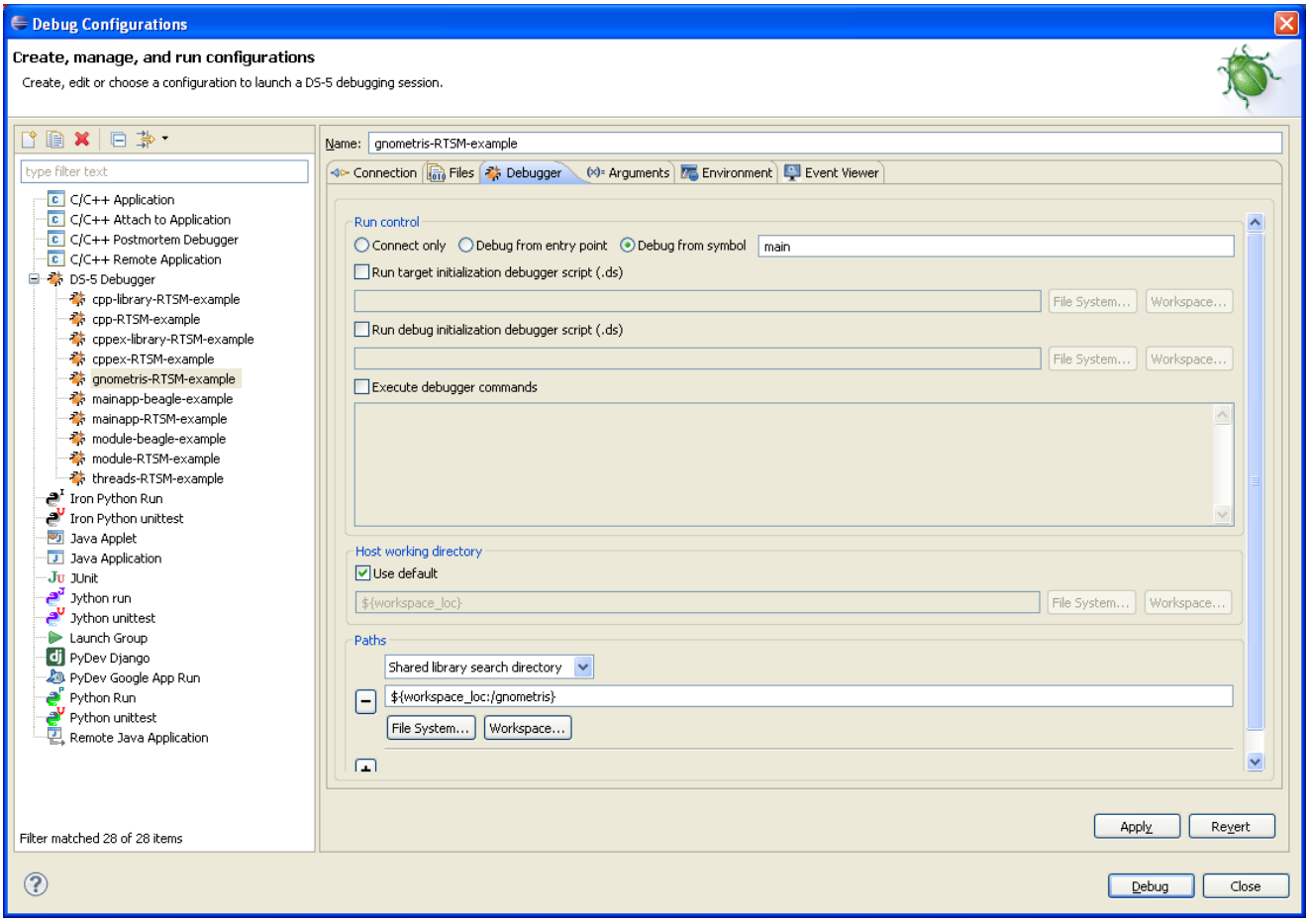


Figure 1-3 Modifying the shared library search paths

For more information on the options in the Debug Configurations dialog box, use the dynamic help.

Related references

2.2 Running, stopping, and stepping through an application on page 2-45
3.1 Examining the target execution environment on page 3-74
3.2 Examining the call stack on page 3-75
2.14 Handling UNIX signals on page 2-62
2.15 Handling processor exceptions on page 2-64
6.4 Breakpoints view on page 6-140
6.6 Commands view on page 6-148
6.9 Disassembly view on page 6-158
6.16 Memory view on page 6-179
6.18 Modules view on page 6-193
6.19 Registers view on page 6-195
6.30 Variables view on page 6-228

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.10 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

6.4 Breakpoints view on page 6-140
6.6 Commands view on page 6-148
6.9 Disassembly view on page 6-158
6.16 Memory view on page 6-179
6.18 Modules view on page 6-193
6.19 Registers view on page 6-195
6.30 Variables view on page 6-228

Related information

Arm Debugger Commands
Arm Security Technology Building a Secure System using TrustZone Technology

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Technical Reference Manual
Architecture Reference Manual

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.11 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.

Arm Development Studio provides a complete UEFI development environment which enables you to:

- Fetch the UEFI source code via the Eclipse Git plug-in.
- Build the source code using Arm Compiler.
- Download the executables to a software model (a Cortex-A9x4 FVP is provided with Development Studio) or to a hardware target (available separately).
- Run/debug the code using Arm Debugger.
- Debug dynamically loaded modules at source-level using Jython scripts.

For more information, see this blog: *UEFI Debug Made Easy*

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.12 About debugging MMUs

Arm 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 Armv7 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*.

Arm Debugger provides visibility of MMU translation tables for some versions of the Arm architecture. To help you debug MMU related issues, Arm 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 Arm® Debugger

In some specific circumstances, Arm 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, Arm 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 Arm 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:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
cache flush
```

Note

Flushing large caches might take a long time.

Related concepts

1.13 About debugging caches on page 1-33

Related references

6.17 MMU/MPU view on page 6-188

Related information

mmu command

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.13 About debugging caches

Arm 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.

Cache Data: L1 Instruction TLB

Linked: smp_primes-A15MPCore-versatile

L1 Instruction TLB | CPU Caches: Cortex-A15_0

Virtual Address	Physical Address	Valid	OS	IS	nG	M	NS	H	VMID	ASID	MAIR	Domain
0x80000000	0x80000000	1	1	1	0	1	0	0	0	0	0x4F	0
0x80001000	0x80001000	1	1	1	0	1	0	0	0	0	0x4F	0
0x0	0x0	0	1	1	0	1	0	0	0	0	0x1	0
0x50670000	0x56F10F3000	0	0	1	1	0	1	1	1	192	0x2	0
0x70C81000	0xEC1BC0000	0	0	1	1	0	1	0	42	44	0x22	0
0x2BA10000	0x7A2D633000	0	0	1	0	0	0	0	90	74	0x14	0
0x50670000	0x56F10F3000	0	0	1	1	1	0	0	10	72	0x2	0
0x93393000	0x720B012000	0	0	0	0	0	0	0	186	44	0xA2	8
0x38679000	0x7AA422D000	0	0	0	1	0	0	1	2	175	0x6B	8
0x8A600000	0xCB779AA000	0	0	0	0	0	1	0	52	94	0x40	1
0xA8772000	0xFC40F83000	0	0	1	1	1	1	0	16	232	0x13	2
0xE0A00000	0xAAB1950000	0	0	1	1	0	0	0	65	156	0x3	2
0x39731000	0xD8013B6000	0	0	1	1	1	0	0	53	24	0x30	12
0x19048000	0x95E1162000	0	0	1	1	0	0	0	137	1	0x3	1
0x4B466000	0x12F80F8000	0	0	1	1	0	1	0	169	0	0xF	0
0x70074000	0x3491043000	0	0	1	1	0	0	1	134	110	0xB	9
0x52314000	0x9BAF49C000	0	0	1	0	0	1	0	35	105	0x82	2
0xA0A51000	0x7E992F4000	0	1	0	0	0	1	0	11	225	0x43	0
0x19E25000	0x42F4B40000	0	1	0	1	0	1	0	152	184	0x42	8
0x78635000	0xDE98353000	0	1	0	0	0	0	0	67	96	0x9A	1
0x50670000	0x56F10F3000	0	0	1	1	0	0	1	33	40	0x2	0
0x86D72000	0x14C13420000	0	0	1	1	0	1	1	8	230	0x46	0

Figure 1-4 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 box 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 box. When this option is not enabled, the information presented might be less accurate due to debugger interactions with the target.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

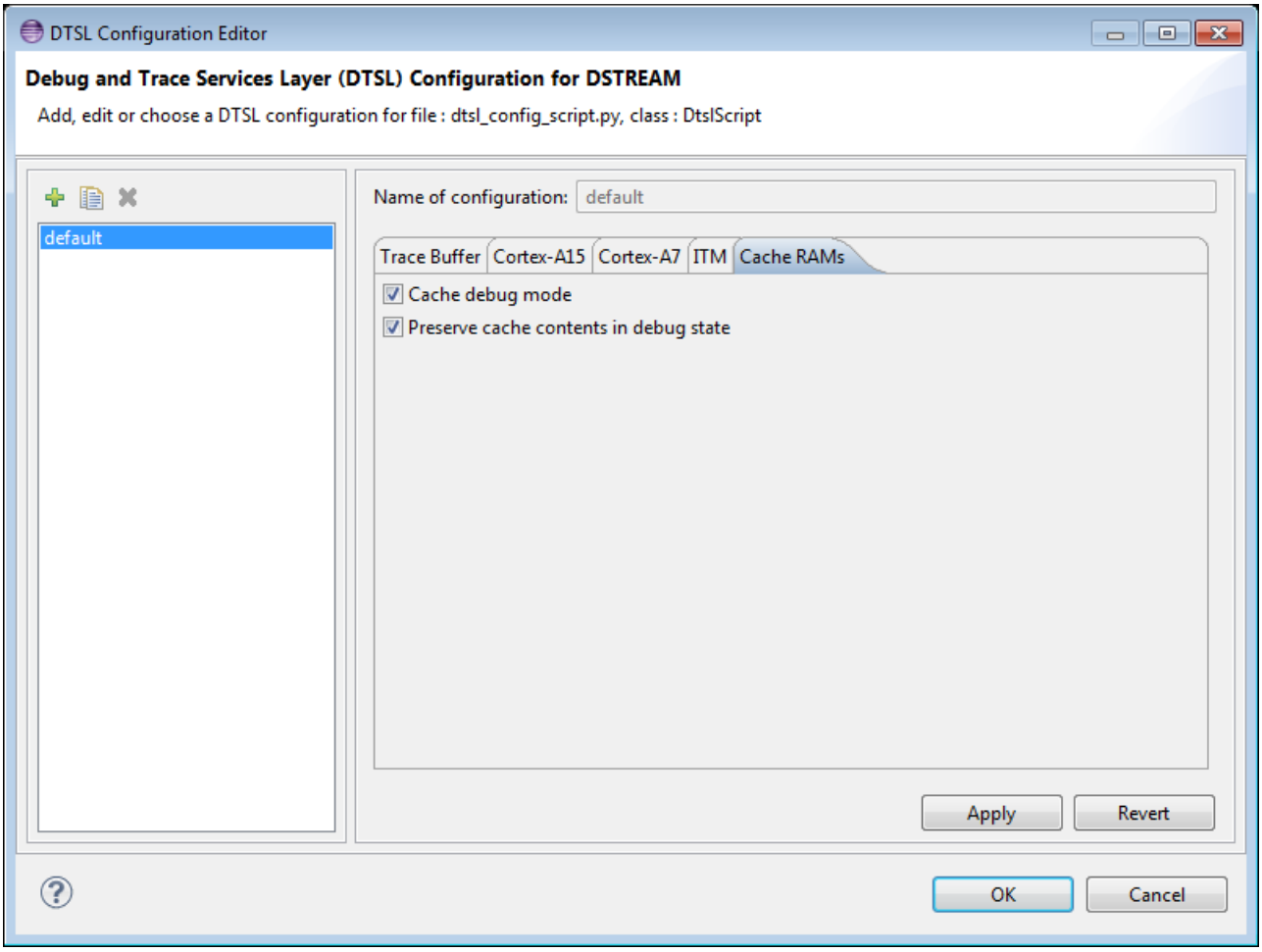


Figure 1-5 DTSL Configuration Editor (Shown with cache read option enabled)

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 box, 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 that 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`

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

6.23 Cache Data view on page 6-208

6.16 Memory view on page 6-179

6.50 DTSL Configuration Editor dialog box on page 6-267

Related information

cache command

memory command

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.14 About Arm® Debugger support for overlays

Overlaying is a programming method that allows applications to share execution regions of memory between different pieces of code at runtime. A piece of code can be transferred to the overlay region to be executed when needed. This piece of code is replaced with another when needed.

Code does not need to be stored in memory and could reside in other storage such as off-chip flash memory. Embedded systems, especially systems without support for virtual memory addressing and systems with limited memory, sometimes use overlays.

An overlaid application consists of:

- Several overlays - These are blocks of code that are not resident in memory all the time.
- The overlay manager (must be part of the non-overlaid code) - The overlay manager takes care of loading and unloading different overlays as they are needed.
- Several overlay regions - These are regions in memory that overlays are loaded into as needed. Each overlay region can contain different overlays at different times.
- Some non-overlaid code - This is code that is permanently resident in memory.
- Data (both RO and RW) - Data is never overlaid.

As a developer, you must decide which functions from your application are used in overlays. You must also decide which functions are used in the same overlay.

You can specify the functions to overlay by annotating the function declarations in your source code. The annotation indicates which overlay each function must be in.

During compilation, the linker assigns overlays to a particular region. When overlays are loaded, it can only be loaded into that region. The linker also detects direct calls between overlays, and between overlays and non-overlaid code. The linker then redirects the calls through veneers, which call the overlay manager to automatically load the target overlay.

Arm Debugger automatically enables overlay support on the presence of linker generated tables and functions as described by the following symbols:

- `Region$$Count$$AutoOverlay`
- `LoadAddr$$Table$$AutoOverlay`
- `CurrLoad$$Table$$AutoOverlay`
- `__ARM_notify_overlay_loaded`

After loading your overlay-enabled application in Arm Development Studio, you can work with overlays from both the command-line console and from the user interface:

- To see detailed information about the currently loaded overlays and the functions within each overlay, use the *Overlays* on page 6-207 view. You can also use the *info overlays* command to view information about the currently loaded overlays and functions within each overlay.
- To enable or disable overlay support, use the *set overlays enabled* command with the on, off, or auto options. The default setting is auto.

See the `overlay_manager` example that is provided with Arm Development Studio for a reference implementation of overlays.

Related references

6.22 Overlays view on page 6-207

Related information

Arm Compiler Software Development Guide: Overlay support

info overlays command

set overlays enabled command

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.15 Debugging a loadable kernel module

You can use Arm Development Studio to develop and debug a loadable kernel module. Loadable modules can be dynamically inserted and removed from a running kernel during development without the need to frequently recompile the kernel.

This tutorial uses a simple character device driver `modex.c` which is part of the Armv7 Linux application examples available in Arm Development Studio.

You can use `modex.c` to compile, run, and debug against your target. The `readme.html` in the `<installation_directory>/examples/docs/kernel_module` contains information about customizing this for your target.

Note

If you are working with your own module, before you can debug it, you must ensure that you:

- Unpack kernel source code and compile the kernel against exactly the same kernel version as your target.
- Compile the loadable module against exactly the same kernel version as your target.
- Ensure that you compile both images with debug information. The debugger requires run-time information from both images when debugging the module.

Procedure

1. Create a new **Debug Configuration**.
   a. From the main Arm Development Studio menu, select **Run** > **Debug Configurations**.
   b. In the **Debug Configurations** dialog box, create a **New Launch Configuration** and give it a name. For example, `my_board`.
   c. In the **Connection** tab, select the target and platform and set up your target connection.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

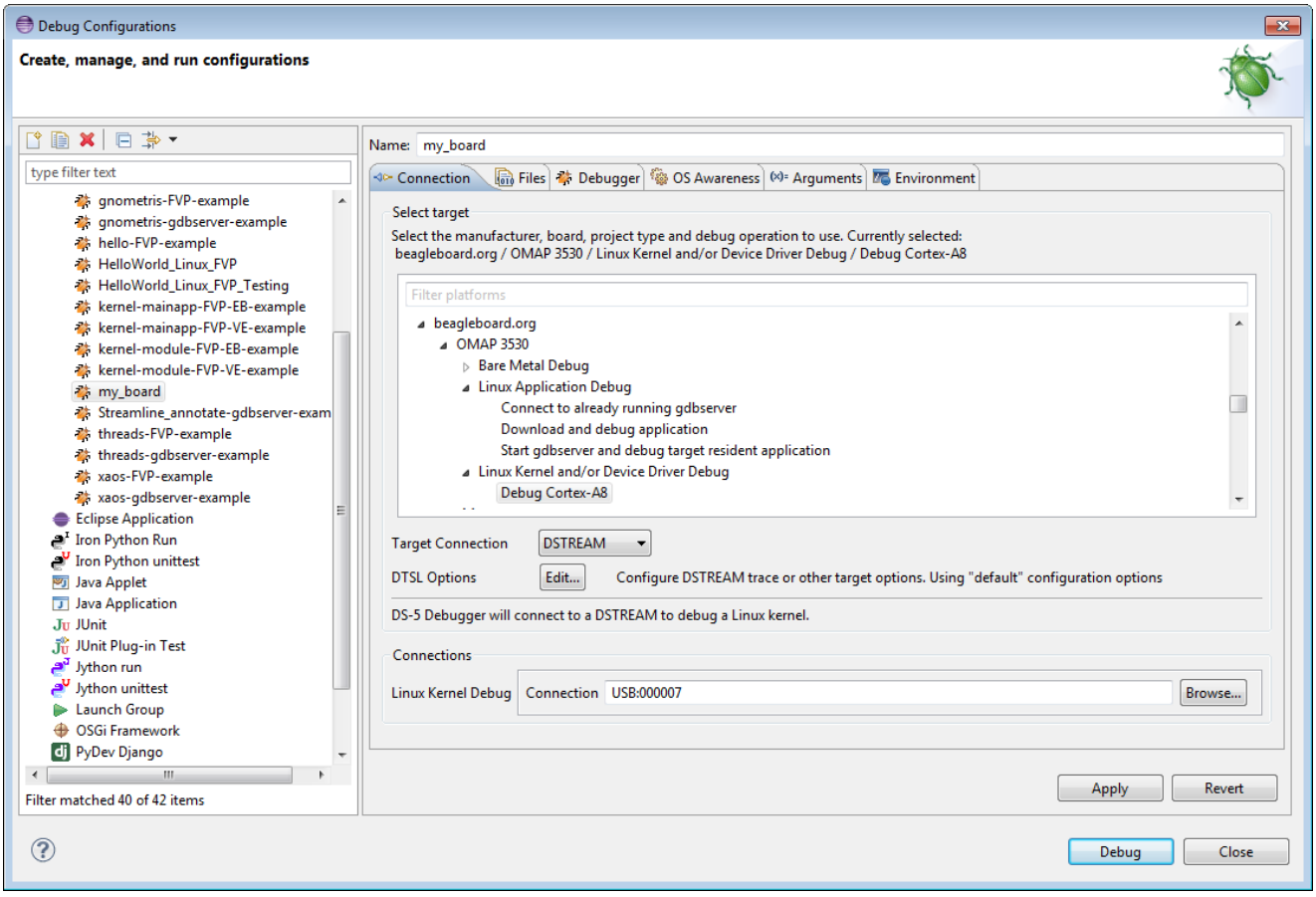


Figure 1-6 Typical connection settings for a Linux kernel/Device Driver Debug

d. In the **Files** tab, set up the debugger settings to load debug information for the Linux kernel and the module.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

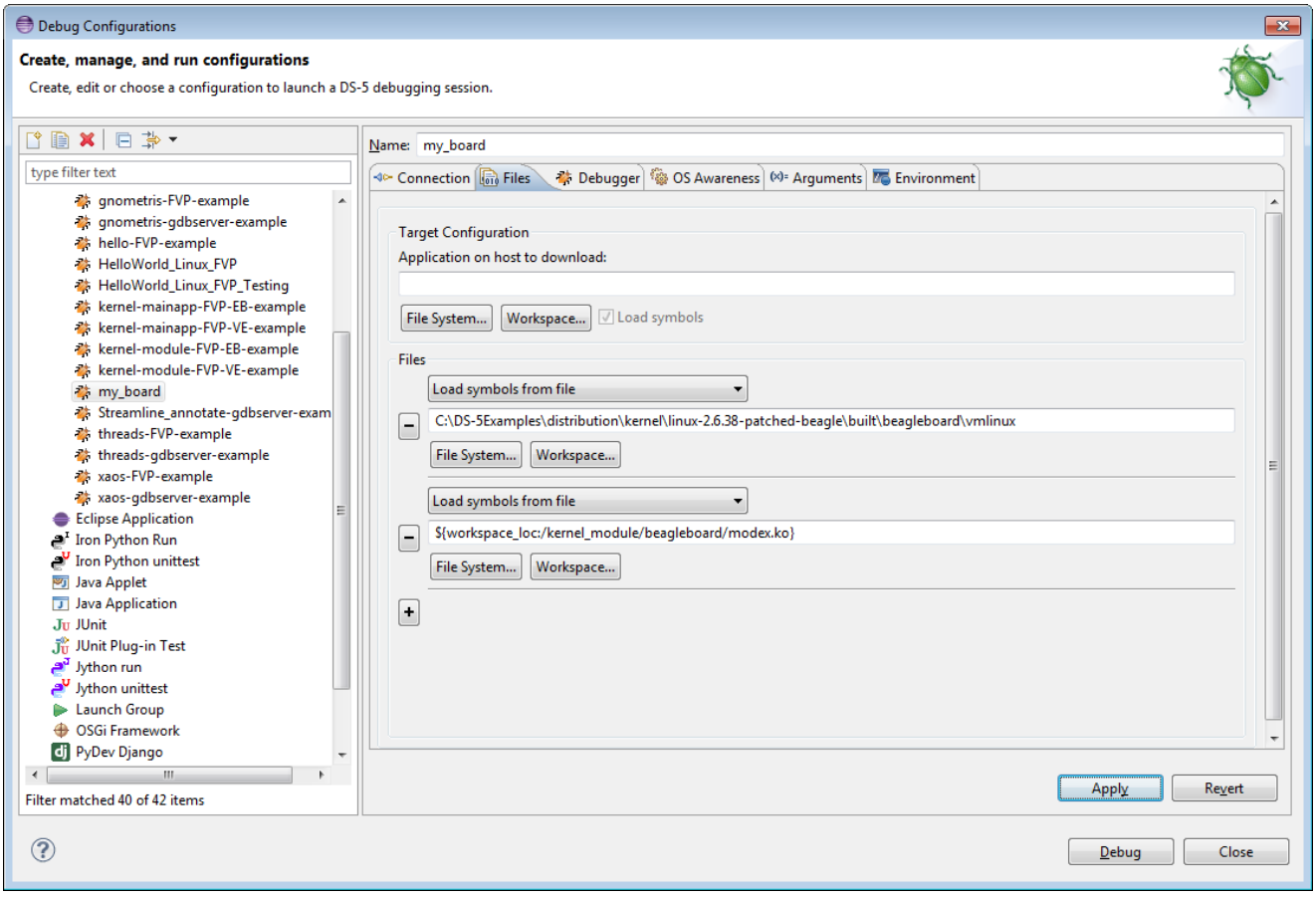


Figure 1-7 Typical Files settings for a Linux kernel/Device Driver Debug

   e. In the **Debugger** tab, select **Connect only** in the **Run control** panel.
   f. Click **Debug** to connect the debugger to the target.

2. Configure and connect a terminal shell to the target. You can use the Remote System Explorer (RSE) provided with Arm Development Studio.
3. Using RSE, copy the compiled module to the target:
   a. On the host workstation, navigate to `.../linux_system/kernel_module/stripped/modex.ko` file.
   b. Drag and drop the module to a writeable directory on the target.
4. Using the terminal shell, insert the `modex.ko` kernel module.
   a. Navigate to the location of the kernel module.
   b. Execute the following command: `insmodmodex.ko`

   The **Modules** view updates to display details of the loaded module.
5. To debug the module, set breakpoints, run, and step as required.
6. To modify the module source code:
   a. Remove the module using commands as required in the terminal shell. For example: `rmmod modex`
   b. Recompile the module.
   c. Repeat steps 3 to 5 as required.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OS modules loaded after connection are displayed in the Modules view.

Note

When you insert and remove a module, the debugger stops the target and automatically resolves memory locations for debug information and existing breakpoints. This means that you do not have to stop the debugger and reconnect when you recompile the source code.

Related references

1.16 Useful commands for debugging a kernel module on page 1-41

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.16 Useful commands for debugging a kernel module

A list of useful commands that you might want to use when debugging a loadable kernel module.

Useful terminal shell commands

lsmod

Displays information about all the loaded modules.

insmod

Inserts a loadable module.

rmmod

Removes a module.

Useful Arm Debugger commands

info os-modules

Displays a list of OS modules loaded after connection.

info os-log

Displays the contents of the OS log buffer.

info os-version

Displays the version of the OS.

info processes

Displays a list of processes showing ID, current state and related stack frame information.

set os-log-capture

Controls the capturing and printing of Operating System (OS) logging messages to the console.

Related tasks

1.15 Debugging a loadable kernel module on page 1-37

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 2
Controlling Target Execution

Describes how to control the target when certain events occur or when certain conditions are met.

It contains the following sections:

- *2.1 Overview: Breakpoints and Watchpoints* on page 2-43.
- *2.2 Running, stopping, and stepping through an application* on page 2-45.
- *2.3 Working with breakpoints* on page 2-47.
- *2.4 Working with watchpoints* on page 2-48.
- *2.5 Importing and exporting breakpoints and watchpoints* on page 2-50.
- *2.6 Viewing the properties of a breakpoint or a watchpoint* on page 2-51.
- *2.7 Associating debug scripts to breakpoints* on page 2-53.
- *2.8 Conditional breakpoints* on page 2-54.
- *2.9 Assigning conditions to an existing breakpoint* on page 2-55.
- *2.10 Conditional watchpoints* on page 2-57.
- *2.11 Assigning conditions to an existing watchpoint* on page 2-58.
- *2.12 Pending breakpoints and watchpoints* on page 2-59.
- *2.13 Setting a tracepoint* on page 2-61.
- *2.14 Handling UNIX signals* on page 2-62.
- *2.15 Handling processor exceptions* on page 2-64.
- *2.16 Using semihosting to access resources on the host computer* on page 2-66.
- *2.17 Working with semihosting* on page 2-68.
- *2.18 Configuring the debugger path substitution rules* on page 2-70.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.1 Overview: 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 can specify other actions to take before resuming execution.

Breakpoints

A breakpoint enables you to interrupt your application when execution reaches a specific address. When execution reaches the breakpoint, normal execution stops before any instruction stored there is executed.

Types of breakpoints:

- Software breakpoints stop your program when execution reaches a specific address.

  Software breakpoints are implemented by the debugger replacing the instruction at the breakpoint address with a special instruction. Software breakpoints can only be set in RAM.
- Hardware breakpoints use special processor hardware to interrupt application execution. Hardware breakpoints are a limited resource.

You can configure breakpoint properties to make them:

- Conditional

  Conditional breakpoints trigger when an expression evaluates to true or when an ignore counter is reached. See *Conditional breakpoints* on page 2-54 for more information.
- Temporary

  Temporary breakpoints can be hit only once and are automatically deleted afterwards.
- Scripted

  A script file is assigned to a specific breakpoint. When the breakpoint is triggered, then the script assigned to it is executed.

Note

- 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.

The **Target** view shows the breakpoint capabilities of the target.

Considerations when setting breakpoints

Be aware of the following when setting breakpoints:

- The number of hardware breakpoints available depends on your processor. Also, there is a dependency between the number of hardware breakpoints and watchpoints because they use the same processor hardware.
- If an image is compiled with a high optimization level or 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 in 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.
- 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* on page 7-292 are found.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Watchpoints

A watchpoint is similar to a breakpoint, but it is the address of a data access that is monitored rather than an instruction being executed. You specify a global variable or a memory address to monitor. 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 read, write, or read/write watchpoints.

Considerations when setting watchpoints

Be aware of the following when setting watchpoints:

- Depending on the target, it is possible that a few additional instructions, after the instruction that accessed the variable, might also be executed. This is because of pipelining effects in the processor. This means that the address that your program stops at might not exactly correspond with the instruction that caused the watchpoint to trigger.
- Watchpoints are only supported on scalar values.
- Watchpoints are only supported on global or static data symbols because they are always in scope and at the same address. Local variables are no longer available when you step out of a particular function.
- 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.

Related references

2.3 Working with breakpoints on page 2-47
2.4 Working with watchpoints on page 2-48
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.2 Running, stopping, and stepping through an application

Arm Debugger enables you to control the execution of your application by sequentially running, stopping, and stepping at the source or instruction level.

Once you have connected to your target, you can use the options on the stepping toolbar **Stepping Toolbar** in the **Debug Control** view to run, interrupt, and step through the application. See *Debug Control* on page 6-151 for more information.

You can also use the **Commands** view to enter the *execution control* group of commands to control application execution.

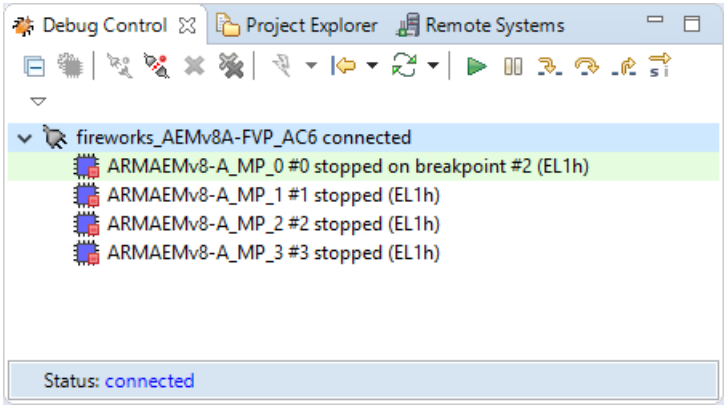


Figure 2-1 Debug Control view

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 this default setting.
- 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 the available hardware breakpoint resources are not enough, then the debugger displays an error message.
- Stepping on multicore targets are dependent on SMP/AMP and debugger settings.

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

- Source level or instruction level.

  In source level debugging, you step through one line or expression in your source code. For instruction level debugging, you step through one machine instruction. Use the **Toggle stepping mode** button on the toolbar to switch between source and instruction level debugging modes.
- Into, over, or out of all function calls.

  If your source is compiled with debug information, using the *execution control* group of commands, you can step into, step through, or step out of functions.
- Through multiple statements in a single line of source code, for example a `for` loop.

Toolbar options

Continue running the application - Click to start or resume execution.

Interrupt running the application - Click to pause execution.

Step through - Click to step through the code.

Step over - Click to step over code.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Step out - Click to continue running to the next line of code after the selected stack frame finishes.

Toggle stepping mode - Click to change the stepping mode between source line and instruction.

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
```

See *Commands view* on page 6-148 for more information.

Related concepts

1.9 About debugging shared libraries on page 1-26

Related references

3.1 Examining the target execution environment on page 3-74

3.2 Examining the call stack on page 3-75

2.14 Handling UNIX signals on page 2-62

2.15 Handling processor exceptions on page 2-64

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.3 Working with breakpoints

The debugger allows you to set software or hardware breakpoints depending on the type of memory available on your target.

To set a 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. See *Disassembly view* on page 6-158 for more information.

To temporarily disable a breakpoint, in the **Breakpoints** view, select the breakpoint you want to disable, and either clear the check-box or right-click and select **Disable breakpoints**. To enable the breakpoint, either select the check-box or right-click and select **Enable breakpoints**. See *Breakpoints view* on page 6-140 for more information.

To delete a breakpoint, double-click on the breakpoint marker or right-click on the breakpoint and select **Toggle Breakpoint**.The following figure shows how breakpoints are displayed in the C/C++ editor, the **Disassembly** view, and the **Breakpoints** view.

Additionally, you can view all breakpoints in your application in the **Breakpoints** view.

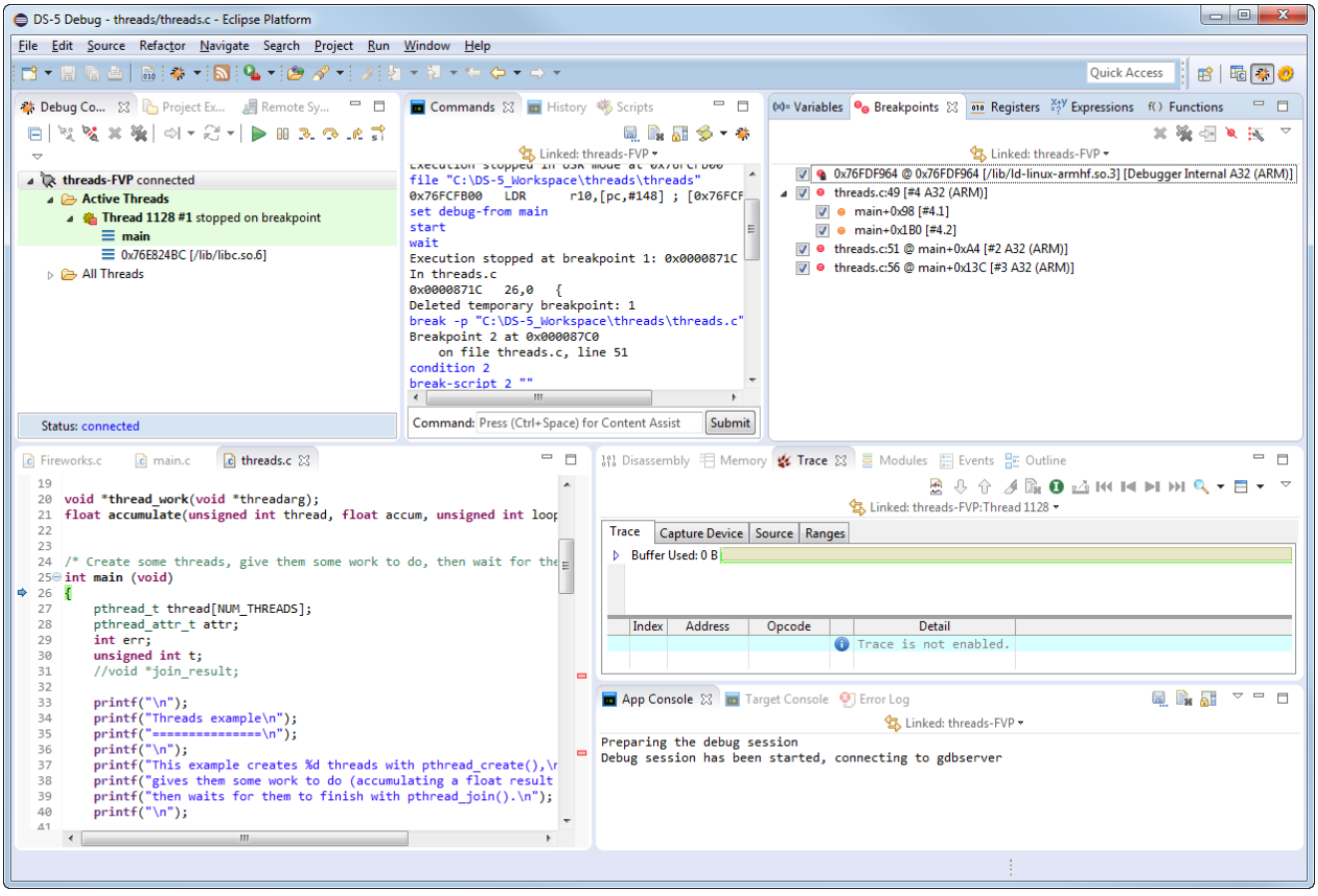


Figure 2-2 Viewing breakpoints

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.4 Working with watchpoints

Watchpoints can be used to stop your target when a specific memory address is accessed by your program.

- If monitoring a global variable, in the `Variables` view, right-click on a data symbol and select **Toggle Watchpoint** to display the``Add Watchpoint`` dialog box.
- If monitoring a memory address, in the `Disassembly` view, right-click on a memory address and select **Toggle Watchpoint** to display the `Add Watchpoint` dialog box.

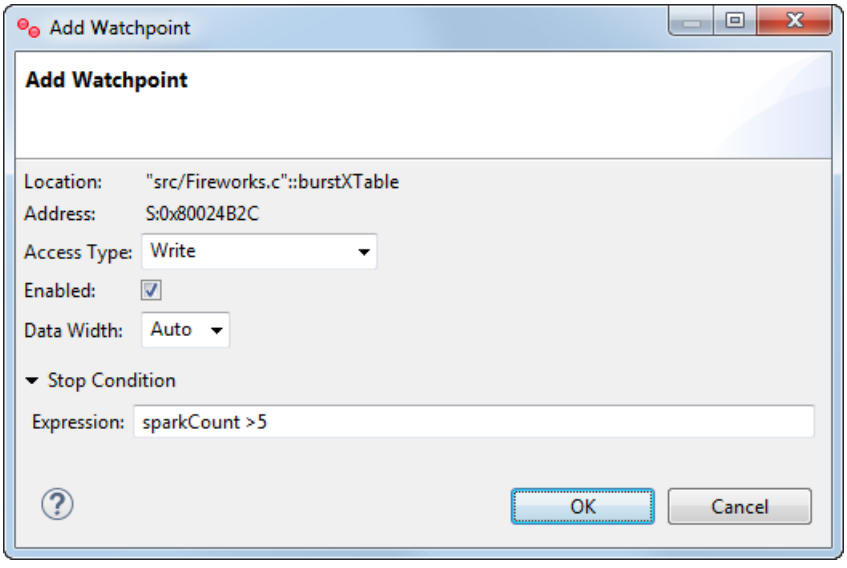


Figure 2-3 Setting a watchpoint on a data symbol

Setting a watchpoint

1. Select the required **Access Type**. You can choose:
   - **Read Read access watchpoint** - To stop the target when a read access occurs.
   - **Write Write access watchpoint** - To stop the target when a write access occurs.
   - **Access Read or Write access watchpoint** - To stop the target when either a read or write access occurs.
2. If you want to enable the watchpoint when it is created, select **Enable**.

   Note

   The default is enabled, but if a conditional watchpoint exists, the watchpoint is created disabled. Only one watchpoint can be enabled if a conditional watchpoint exists.

3. Specify the width to watch at the given address, in bits. Accepted values are: 8, 16, 32, and 64 if supported by the target.

   This parameter is optional. The width defaults to:
   - 32 bits for an address.
   - The width corresponding to the type of the symbol or expression, if entered.
4. Expand **Stop Condition** and in the **Expression** field, enter a C-style expression. For example, if your application code has a variable `x`, then you can specify: `x == 10`. If no expression is specified, then the breakpoint or watchpoint condition is deleted.
5. Click **OK** to apply your selection.

If you created a watchpoint to monitor a global variable, you can view it in the **Variables** view. If you created a watchpoint to monitor a memory address, you can view it in the **Memory** view.

Also, you can view all watchpoints and breakpoints in your application in the **Breakpoints** view.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Deleting a watchpoint

To delete a watchpoint, right-click a watchpoint and either select **Remove Watchpoint** or select **Toggle Watchpoint**.

Disabling a watchpoint

To disable a watchpoint, right-click a watchpoint and select **Disable Watchpoint** to temporarily disable it. To re-enable it, select **Enable Watchpoint**.

Related tasks

2.11 Assigning conditions to an existing watchpoint on page 2-58

Related references

6.38 Watchpoint Properties dialog box on page 6-245

Related information

awatch command

rwatch command

watch command

watch set property command

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.5 Importing and exporting breakpoints and watchpoints

You can import and export Arm Development Studio breakpoints and watchpoints from within the **Breakpoints** view. This makes it possible to reuse your current breakpoints and watchpoints in a different workspace.

To import or export breakpoints and watchpoints to a settings file, use the options in the **Breakpoints** view menu.

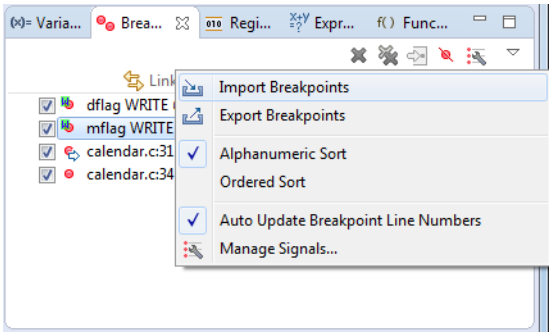


Figure 2-4 Import and export breakpoints and watchpoints

Note

- All breakpoints and watchpoints shown in the **Breakpoints** view are saved.
- Existing breakpoints and watchpoints settings for the current connection are deleted and replaced by the settings from the imported file.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.6 Viewing the properties of a breakpoint or a watchpoint

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

Viewing the properties of a breakpoint

There are several ways to view the properties of a breakpoint. You can:

- In the **Breakpoints** view, right-click a breakpoint and select **Properties…**.
- In the **Disassembly** view, right-click a breakpoint and select **Breakpoint Properties**.
- In the code view, right-click a breakpoint and select **Arm DS Breakpoints** > **Breakpoint Properties**.

This displays the **Breakpoint Properties** dialog box.

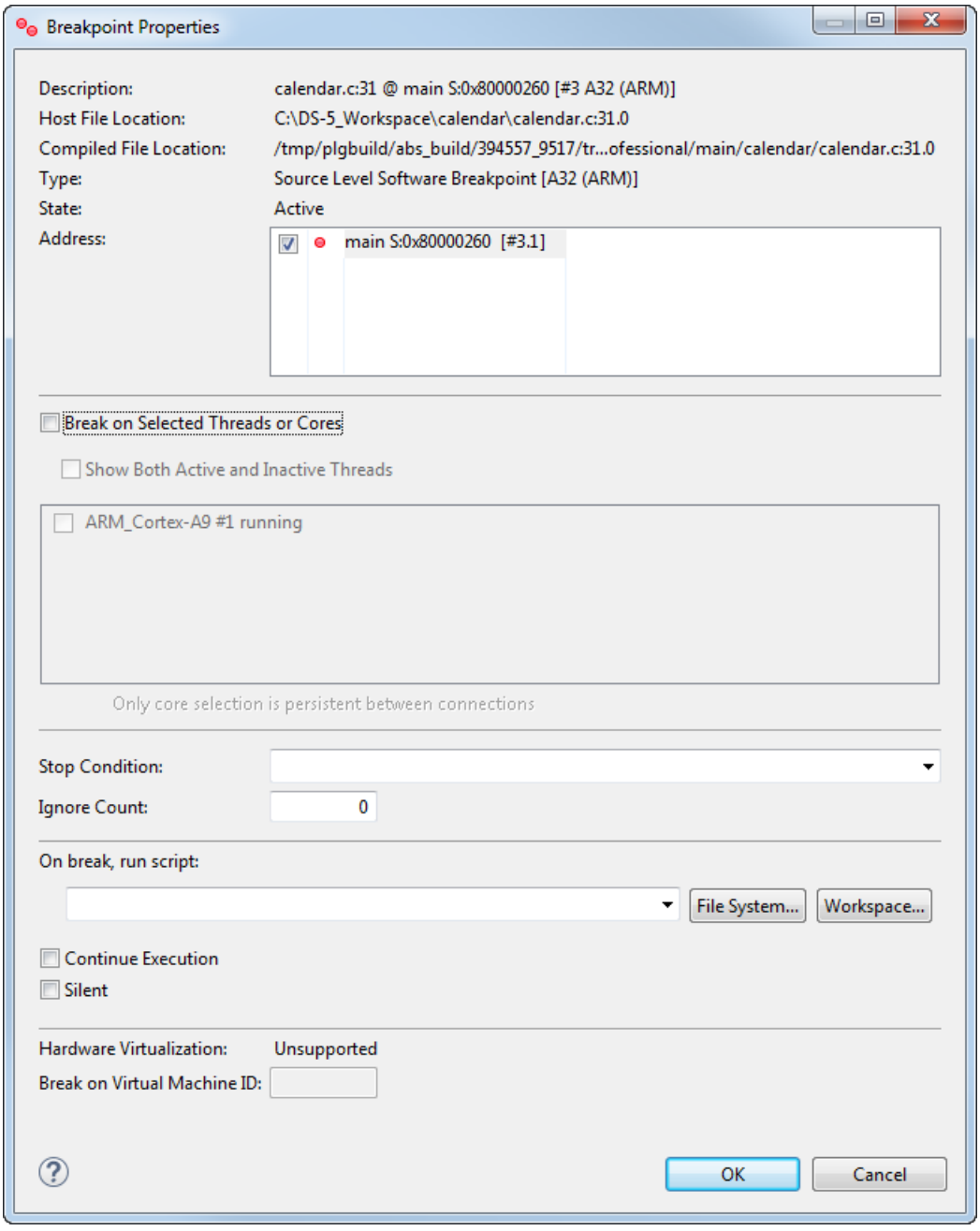


Figure 2-5 Viewing the properties of a breakpoint

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Viewing the properties of a watchpoint

There are several ways to view the properties of a watchpoint. You can:

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

This displays the **Watchpoint Properties** dialog box:

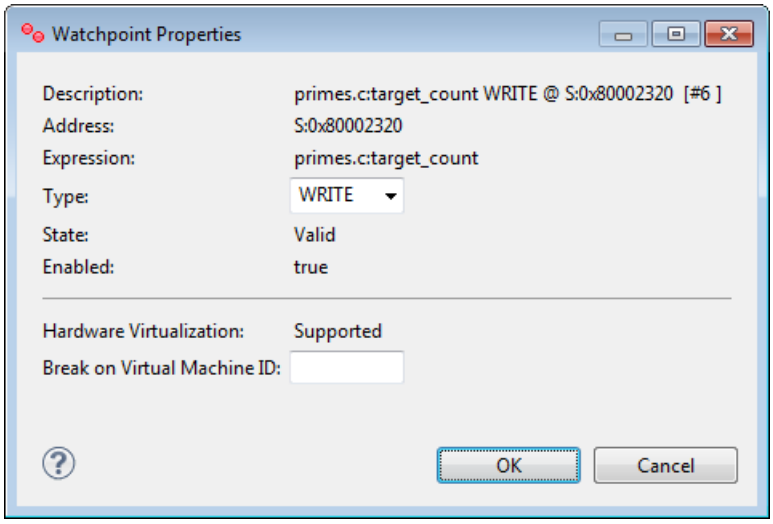


Figure 2-6 Watchpoint Properties

- Use the options available in the **Type** options to change the watchpoint type.
- If your target supports virtualization, enter a virtual machine ID in **Break on Virtual Machine ID**. This allows the watchpoint to stop only at the virtual machine ID you specify.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.7 Associating debug scripts to breakpoints

Using conditional breakpoints, you can run a script each time the selected breakpoint is triggered. You assign a script file to a specific breakpoint, and when the breakpoint is hit, the script executes.

If using the user interface, use the **Breakpoint Properties** dialog box to specify your script. See *Breakpoint Properties* on page 6-242 for more information.

If using the command-line, use the *break-script* command to specify your script.

Be aware of the following when using scripts with breakpoints:

- If you assign a script to a breakpoint that has sub-breakpoints, the debugger attempts to execute the script for each sub-breakpoint. If this happens, an error message is displayed. For an example of sub-breakpoints, see *Breakpoints view* on page 6-140.
- Take care with 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.
- If you put the *continue* command at the end of a script, this has the same effect as setting the **Continue Execution** option on the **Breakpoint Properties** dialog box.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.8 Conditional breakpoints

Conditional breakpoints have properties assigned to test for conditions that must be satisfied to trigger the breakpoint. When the underlying breakpoint is hit, the specified condition is checked and if it evaluates to true, then the target remains in the stopped state, otherwise execution resumes.

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.

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.

See *Assigning conditions to an existing breakpoint* on page 2-55 for an example. Also, see the details of the *break* command to see how it is used to specify conditional breakpoints.

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.
- If you assign a script to a breakpoint that has sub-breakpoints, the debugger attempts to execute the script for each sub-breakpoint. If this happens, an error message is displayed. For an example of sub-breakpoints, see *Breakpoints view* on page 6-140.

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.

Related references

6.3 Arm assembler editor on page 6-138
6.4 Breakpoints view on page 6-140
6.5 C/C++ editor on page 6-144
6.6 Commands view on page 6-148
6.9 Disassembly view on page 6-158
6.12 Expressions view on page 6-169
6.16 Memory view on page 6-179
6.19 Registers view on page 6-195
6.30 Variables view on page 6-228

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.9 Assigning conditions to an existing breakpoint

Using the options available on the **Breakpoint Properties** dialog box, 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.

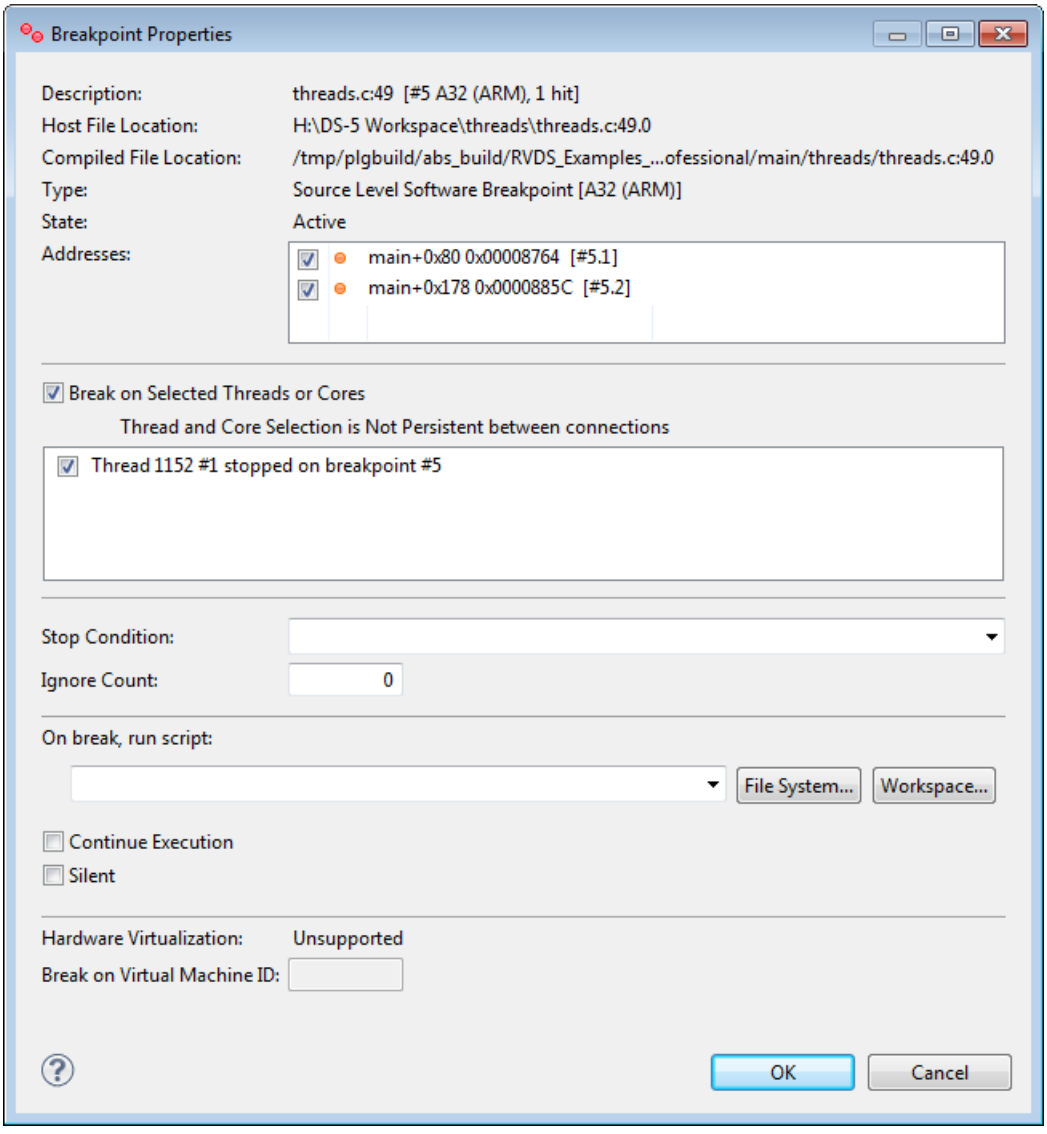


Figure 2-7 Breakpoint Properties dialog box

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 code has a variable `x`, then you can specify: `x == 10`.

      Note

      See the *break* command to see how it is used to specify conditional breakpoints.

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, runscript** 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.

7. 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.
8. Select **Silent** if you want to hide breakpoint information in the **Commands** view.
9. If required, specify a Virtual Machine ID (VMID).

   Note

   You can only specify a Virtual Machine ID (VMID) if hardware virtualization is supported by your target.

10. Once you have selected the required options, click **OK** to save your changes.

Related references

6.3 Arm assembler editor on page 6-138
6.4 Breakpoints view on page 6-140
6.5 C/C++ editor on page 6-144
6.6 Commands view on page 6-148
6.9 Disassembly view on page 6-158
6.12 Expressions view on page 6-169
6.16 Memory view on page 6-179
6.19 Registers view on page 6-195
6.30 Variables view on page 6-228

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.10 Conditional watchpoints

Conditional watchpoints have properties that are assigned to test for conditions that must be satisfied to trigger the watchpoint. When the conditional watchpoint is hit, the specified condition is checked and if it evaluates to true, the watchpoint is triggered, and the target stops.

For example, using conditional watchpoints you can:

- Set a watchpoint to stop when accessing data from a memory region, but only when a variable evaluates to a given value.
- On processors that implement virtualization extensions, you can set a watchpoint to trigger only when running within a specific virtual machine (determined through its Virtual Machine ID (VMID)).
- On all Armv7 and Armv8 processors, you can set a watchpoint to trigger only when a specific process is running, as defined by the current Context ID.

Note

- Conditional watchpoints are not supported on gdbserver connections currently.
- You can create several conditional watchpoints, but when a conditional watchpoint is enabled, no other watchpoints (regardless of whether they are conditional) can be enabled.
- Conditional watchpoints can be intrusive and lower performance if they are hit frequently since the debugger stops the target every time the watchpoint triggers.

See *Working with watchpoints* on page 2-48 for details about assigning a condition to watchpoint when creating it. See *Assigning conditions to an existing watchpoint* on page 2-58 for details about assigning conditions to an existing watchpoint.

You can also use the *awatch*, *rwatch*, and *watch* commands to assign conditions to a watchpoint.

Considerations when creating conditional watchpoints

- If the instruction causing the trap occurred synchronously, then to evaluate a condition after any state has changed (for example, a store to an address), the debugger steps the instruction that caused the trap. The debugger then proceeds to evaluate the condition to see whether to stop on the watchpoint. This is required so that a specific address can be watched and trap immediately after a specific value is written to that address.
- Ignore count or core/thread specific watchpoints are not supported.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.11 Assigning conditions to an existing watchpoint

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

To specify the condition which must evaluate to true at the time the watchpoint is triggered for the target to stop, use the **Stop Condition** field.

Note

You can create several conditional watchpoints, but when a conditional watchpoint is enabled, no other watchpoints (regardless of whether they are conditional) can be enabled.

Procedure

1. In the **Breakpoints** view, select the watchpoint that you want to modify and right-click to display the context menu.
2. Select **Properties…** to display the **Watchpoint Properties** dialog box.
3. If not selected, select **Enabled**.
4. Specify the width to watch at the given address, in bits. Accepted values are: 8, 16, 32, and 64 if supported by the target.

   This parameter is optional. The width defaults to:
   - 32 bits for an address.
   - The width corresponding to the type of the symbol or expression, if entered.
5. Expand **Stop Condition** and in the **Expression** field, enter a C-style expression. For example, if your application code has a variable `x`, then you can specify: `x == 10`.

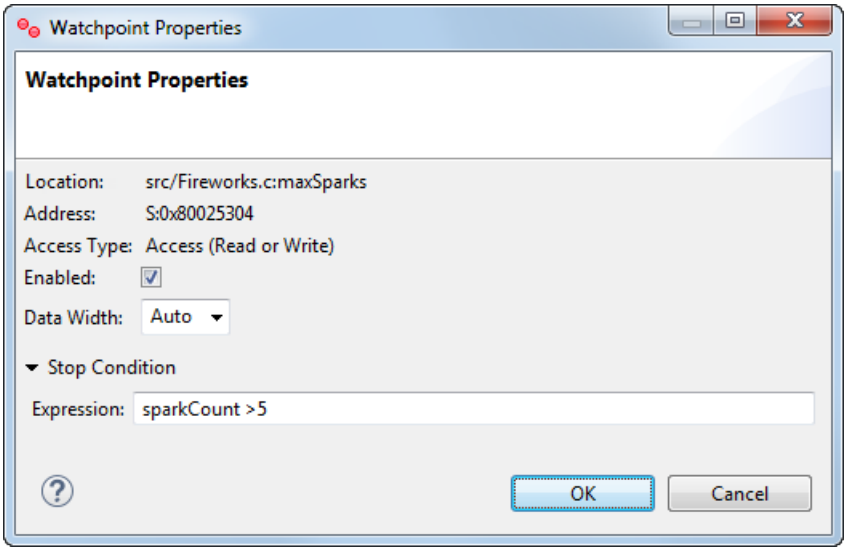


Figure 2-8 Watchpoint Properties dialog box

6. Click **OK**, to apply the condition to the watchpoint.

Related references

2.4 Working with watchpoints on page 2-48
6.38 Watchpoint Properties dialog box on page 6-245

Related information

awatch command
rwatch command
watch command
watch set property command

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.12 Pending breakpoints and watchpoints

A pending breakpoint or watchpoint is one that exists in the debugger but is not active on the target until some precondition is met, such as a shared library being loaded.

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.

When a new shared library is loaded, the debugger re-evaluates all pending breakpoints and watchpoints. Breakpoints or watchpoints with addresses that can be resolved are set as standard execution breakpoints or watchpoints and those with unresolved addresses remain pending. The debugger automatically changes any breakpoints or watchpoints in a shared library to a pending one when the library is unloaded by your application.

Manually setting a pending breakpoint or watchpoint

To manually set a pending breakpoint or watchpoint, you can use the `-p` option with any of these commands:

advance

break

hbreak

tbreak

thbreak

watch

awatch

rwatch

Note

You can enter debugger commands in the **Commands** view. See *Commands view* on page 6-148 for more information.

Examples

```
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
```

Resolving a pending breakpoint or watchpoint

You can force the resolution of a pending breakpoint or watchpoint. This might be useful, for example, if you have manually modified the shared library search paths.

To resolve a pending breakpoint or watchpoint:

- If using the user interface, right-click on the pending breakpoint or watchpoint that you want to resolve, and select **Resolve**.
- If using the command-line, use the *resolve* command.

Related references

6.3 Arm assembler editor on page 6-138

6.4 Breakpoints view on page 6-140

6.5 C/C++ editor on page 6-144

6.6 Commands view on page 6-148

6.9 Disassembly view on page 6-158

6.12 Expressions view on page 6-169

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.16 Memory view on page 6-179
6.19 Registers view on page 6-195
6.30 Variables view on page 6-228

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.13 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 tracepoint 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.
- The instruction execution history panel in 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. See *About trace support* on page 3-76 for more information about Trace Start, Stop, and Trigger Point.

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

6.3 Arm assembler editor on page 6-138
6.4 Breakpoints view on page 6-140
6.5 C/C++ editor on page 6-144
6.6 Commands view on page 6-148
6.9 Disassembly view on page 6-158
6.12 Expressions view on page 6-169
6.16 Memory view on page 6-179
6.19 Registers view on page 6-195
6.30 Variables view on page 6-228

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.14 Handling UNIX signals

When debugging a Linux application you can configure the debugger to stop or report when a UNIX signal is raised.

To manage UNIX signals in the debugger, either:

- Select **Manage Signals** from the **Breakpoints** toolbar or the 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.

Note

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

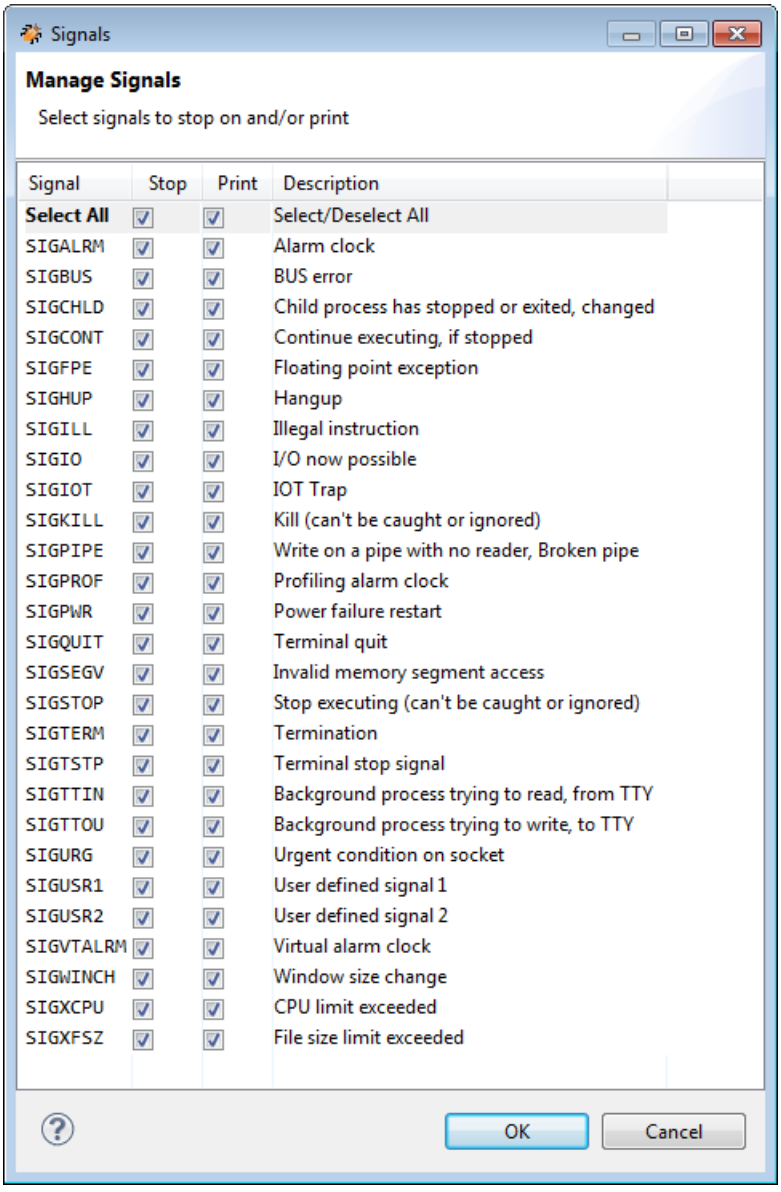


Figure 2-9 Manage signals dialog box (UNIX signals)

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

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

Examples

If you want the application to ignore a signal, but log the event when it is triggered, then you must enable stopping on a signal.

Ignoring a SIGHUP 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.

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

Debugging a 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.

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

Related concepts

1.9 About debugging shared libraries on page 1-26

Related references

2.2 Running, stopping, and stepping through an application on page 2-45
3.1 Examining the target execution environment on page 3-74
3.2 Examining the call stack on page 3-75
2.15 Handling processor exceptions on page 2-64
6.4 Breakpoints view on page 6-140
6.6 Commands view on page 6-148
6.40 Manage Signals dialog box on page 6-248

Related information

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.15 Handling processor exceptions

Arm processors handle exceptions by jumping to one of a set of fixed addresses known as exception vectors.

Except for a Supervisor Call (SVC) or SecureMonitor Call (SMC), these events are not part of normal program flow. The events can happen unexpectedly, perhaps 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 handling.

When vector catch is enabled, the effect is similar to placing a breakpoint on the selected vector table entry. But in this case, vector catches use dedicated hardware in the processor and do not use up valuable breakpoint resources.

Note

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

To manage vector catch in the debugger, either:

- Select **Manage Signals** from the **Breakpoints** toolbar or the view menu to display the **Manage Signals** dialog box.

  For each individual signal that you want information, select either the **Stop** or **Print** option. The **Stop** option stops the execution and prints a message. The **Print** option prints a message, but continues execution. You can view these messages in the **Commands** view.

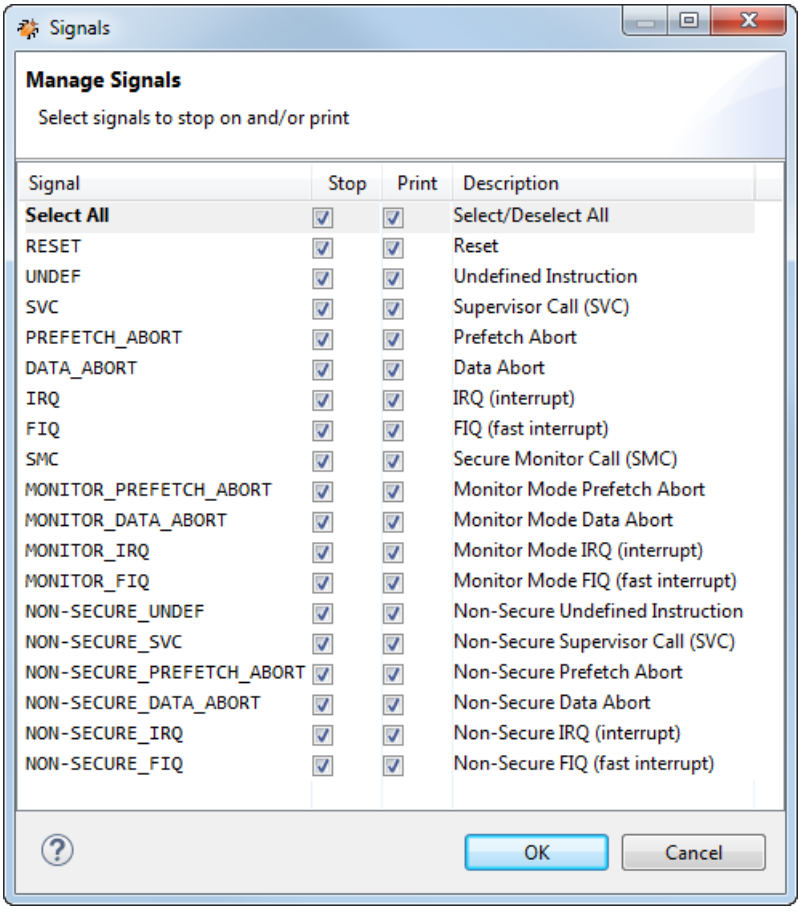


Figure 2-10 Manage Signals dialog box

- Use the *handle* command and view the results in the **Commands** view.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

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

Examples

Debugging an exception handler

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 in the **Commands** view. You can then step or run to the handler, if present.

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

Ignoring an exception

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

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

Related concepts

1.9 About debugging shared libraries on page 1-26

Related references

2.2 Running, stopping, and stepping through an application on page 2-45
3.1 Examining the target execution environment on page 3-74
3.2 Examining the call stack on page 3-75
2.14 Handling UNIX signals on page 2-62
6.4 Breakpoints view on page 6-140
6.6 Commands view on page 6-148
6.40 Manage Signals dialog box on page 6-248

Related information

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.16 Using semihosting to access resources on the host computer

Semihosting is a mechanism that enables code running on an Arm target or emulator to communicate with and use the Input/Output facilities on a host computer. The host must be running the emulator, or a debugger that is attached to the Arm target.

Examples of these facilities include keyboard input, screen output, and disk I/O. For example, you can use this mechanism to enable functions in the C library, such as `printf()` and `scanf()`, to use the screen and keyboard of the host instead of having a screen and keyboard on the target system.

This is useful because development hardware often does not have all the input and output facilities of the final system. Semihosting enables the host computer to provide these facilities.

Semihosting is implemented by a set of defined software instructions, for example, SVCs, that generate exceptions from program control. The application invokes the appropriate semihosting call and the debug agent then handles the exception. The debug agent provides the required communication with the host.

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.

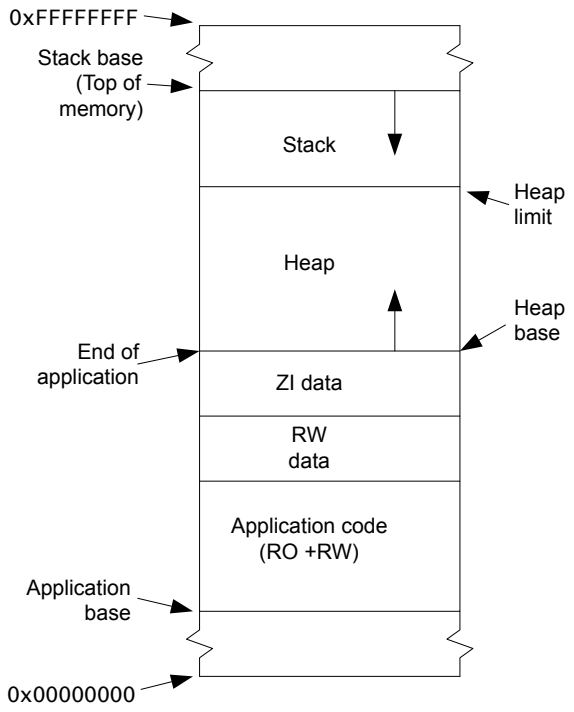


Figure 2-11 Typical layout between top of memory, stack, and heap

Semihosting support in Arm® Development Studio

The suite of tools in Arm Development Studio supports the latest semihosting specification for both AArch64 and AArch32 states on both software models and real target hardware.

The *Semihosting for AArch32 and AArch64 Version 2.0 specification* introduces support for semihosting in mixed AArch64 and AArch32 systems by using HLT trap instructions in the A64, A32, and T32 instruction sets.

- In Arm Compiler v6.6 and later, to build a project using HLT-based semihosting, import the symbol `__use_hlt_semihosting`. HLT-based semihosting libraries are then selected automatically at link-time.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See *Using the C and C++ libraries with an application in a semihosting environment* section in the Arm Compiler Arm C and C++ Libraries and Floating-Point Support User Guide for more information.

- In Fast Models and FVPs, semihosting is enabled when the `semihosting-enable=true` option is set. See *Configuring the model* in the Fixed Virtual Platform (FVP) Reference Guide for more information.
- In Arm Debugger, semihosting is enabled automatically when an image is loaded that contains the special symbols `__auto_semihosting` or `__semihosting_library_function`, or if you explicitly enable semihosting using the `set semihosting enabled on` command. See the *set semihosting* command documentation for more information.

Related references

7.3 About passing arguments to main() on page 7-294

2.17 Working with semihosting on page 2-68

6.47 Debug Configurations - Arguments tab on page 6-262

6.1 App Console view on page 6-135

Related information

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.17 Working with semihosting

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

Enabling semihosting support

By default, semihosting support is disabled in the debugger. However, Arm Debugger enables semihosting automatically if either `__auto_semihosting` or `__semihosting_library_function` ELF symbols are present in an image. Also, if the image is compiled with Arm Compiler 5.0 and later, the linker automatically adds `__semihosting_library_function` to an image if it uses functions that require semihosting.

In C code, you can 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.

Examples

```
#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;
}
```

Using semihosting from the command-line console

The input/output requests from application code to a host workstation running the debugger are called semihosting messages. 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.

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 options:

`--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.

Note

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.

See *Command-line debugger options* on page 4-82 for more information.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Using semihosting from the user interface

The **App Console** view in the **DS Debug** perspective controls all the semihosting input/output requests (`stdin`, `stdout`, and `stderr`) between the application code and the debugger.

Related references

7.3 About passing arguments to main() on page 7-294

2.16 Using semihosting to access resources on the host computer on page 2-66

6.47 Debug Configurations - Arguments tab on page 6-262

6.1 App Console view on page 6-135

Related information

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.18 Configuring the debugger path substitution rules

During the debugging process, the debugger attempts to open the corresponding source file when execution stops at an address in the image or shared object.

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

- The path that is 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.

Procedure

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

   Note

   You must be connected to your target to access the **Path Substitution** menu option.

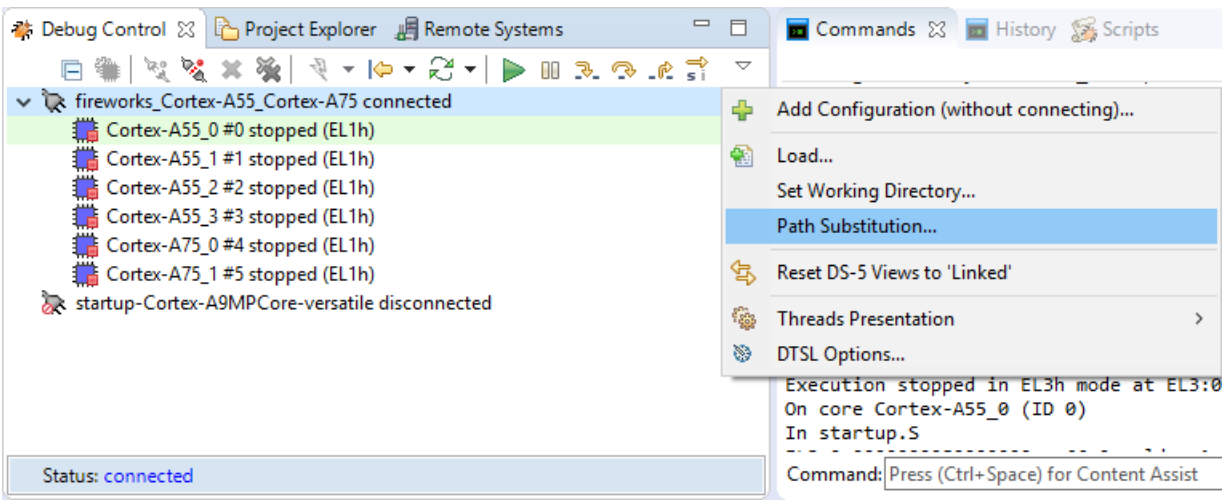


Figure 2-12 Set Path Substitution

3. Click on the required toolbar icons in the **Path Substitution** dialog box:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

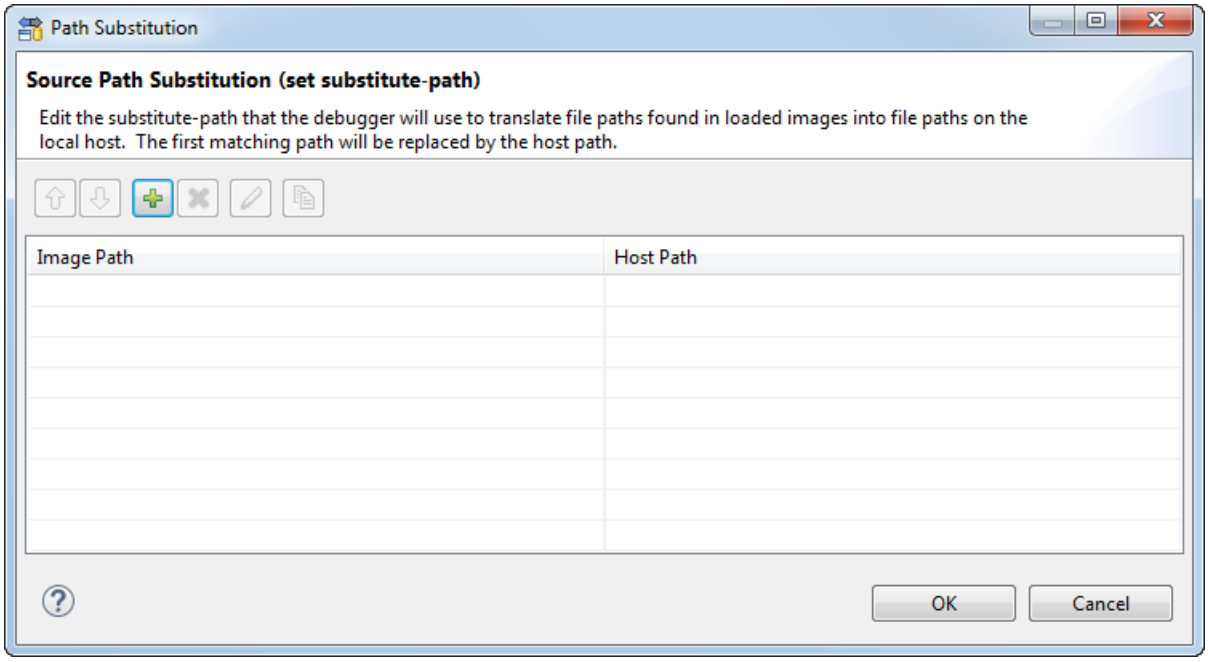


Figure 2-13 Path Substitution dialog box

a. Add a path using the **Edit Substitute Path** dialog box.

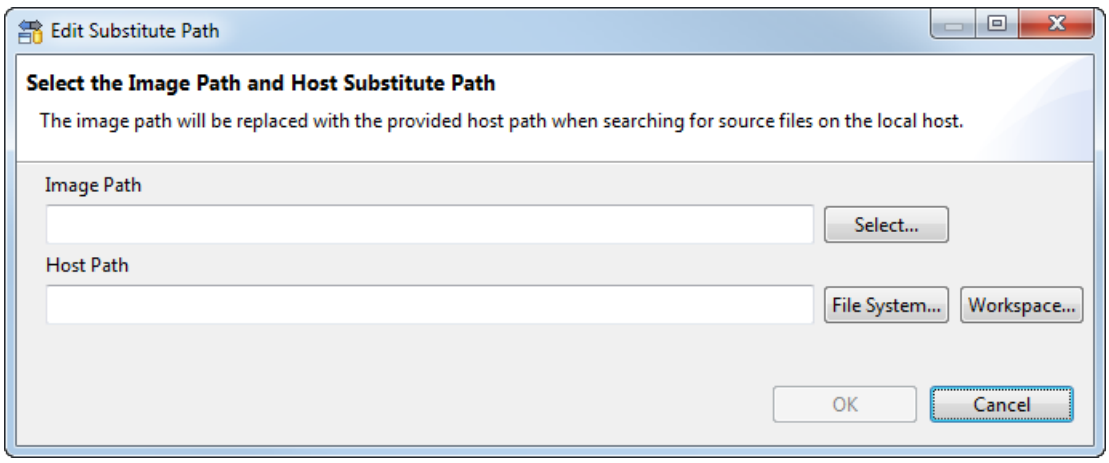


Figure 2-14 Edit Substitute Path dialog box

b. **Image Path** - Enter the original path for the source files or **Select…** a compilation path.
c. **Host Path** - Enter the current location of the sources. Click **File System…** to locate the source files in an external folder or click **Workspace…** to locate the source files in a workspace project.
d. Click **OK** to accept the changes and close the dialog box.

4. Delete an existing path.
   a. Select the path that you want to delete in the **Path Substitution** dialog box.
   b. Click **Delete Path** to delete the selected path.
   c. Click **OK** to accept the changes and close the dialog box.
5. Edit an existing path.
   a. Select the path that you want to edit in the **Path Substitution** dialog box.
   b. Click **Edit Path** to edit the path in the **Edit Substitute Path** dialog box.
   c. Make your changes and click **OK** to accept the changes and close the dialog box.
6. Duplicate substitution rules.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

a. Select the path that you want to duplicate in the **Path Substitution** dialog box.
b. Click **Duplicate substitution rules** to display the **Edit Substitute Path** dialog box.
c. Make your changes and click **OK** to accept the changes and close the dialog box.
d. If required, you can change the order of the substitution rules.
e. Click **OK** to pass the substitution rules to the debugger and close the **Path Substitution** dialog box.

Related concepts

7.2 About loading debug information into the debugger on page 7-292

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 3
Examining the Target

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

It contains the following sections:

- *3.1 Examining the target execution environment* on page 3-74.
- *3.2 Examining the call stack* on page 3-75.
- *3.3 About trace support* on page 3-76.
- *3.4 About post-mortem debugging of trace data* on page 3-79.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.1 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 **Development Studio** 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.

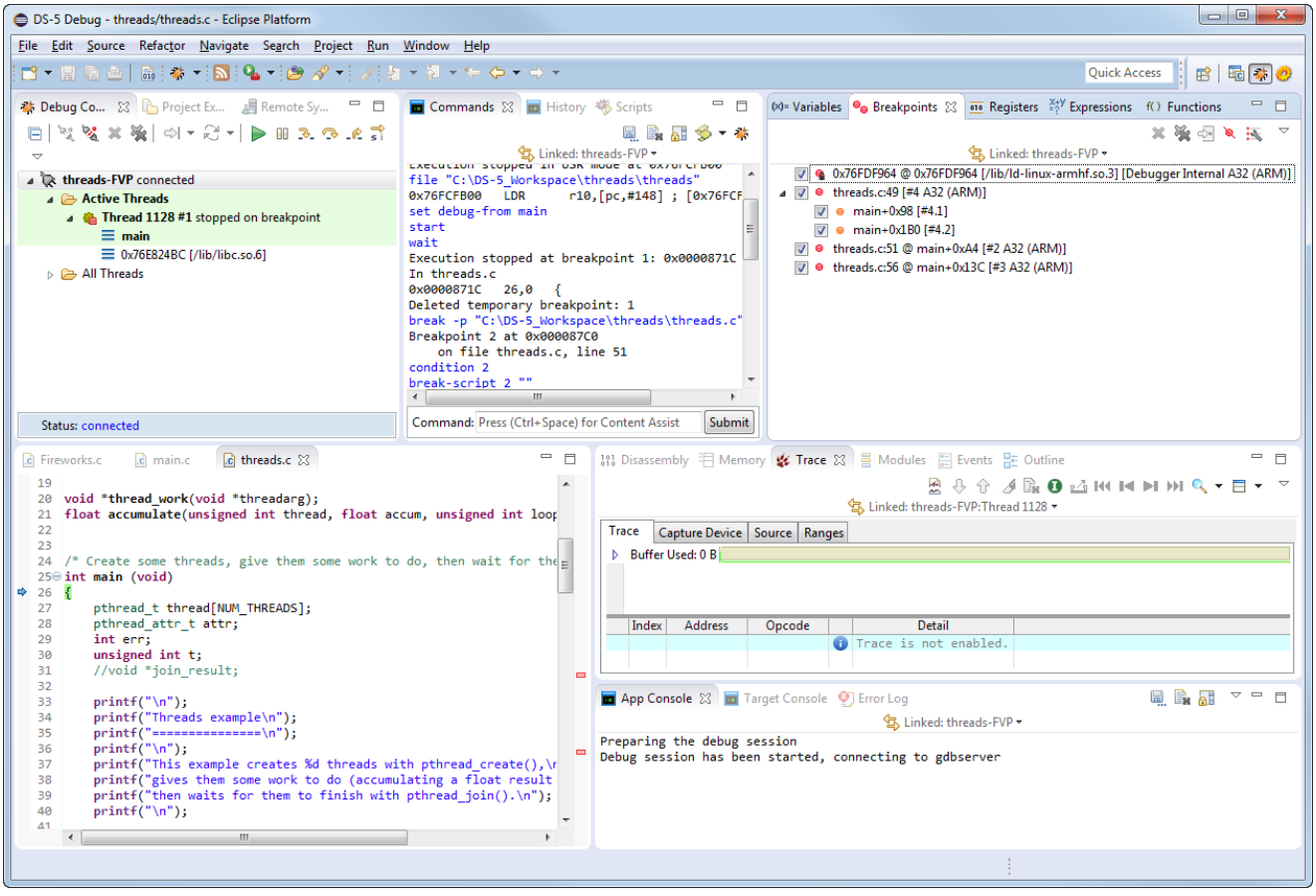


Figure 3-1 Target execution environment

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

1.9 About debugging shared libraries on page 1-26

Related references

2.2 Running, stopping, and stepping through an application on page 2-45
3.2 Examining the call stack on page 3-75
2.14 Handling UNIX signals on page 2-62
2.15 Handling processor exceptions on page 2-64

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2 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.

Use the **Stack** view to display stack information for the currently active connection in the **Debug Control** view. All the views in the **Development Studio** perspective are associated with the current stack frame and are updated when you select another frame. See *Stack view* on page 6-155 for more information.

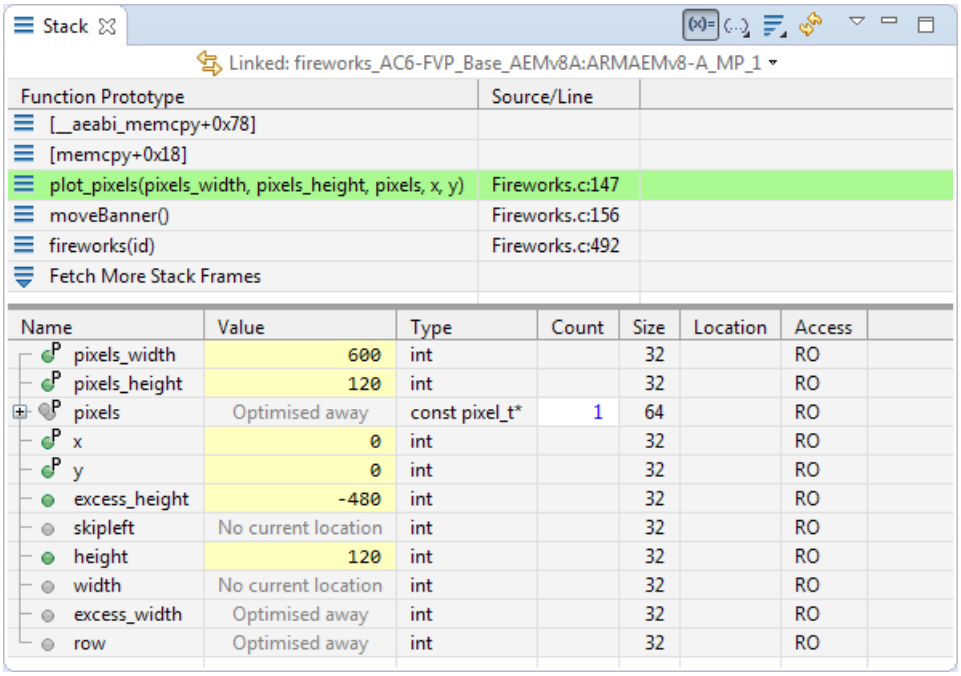


Figure 3-2 Stack view showing information for a selectedcore

Related concepts

1.9 About debugging shared libraries on page 1-26

Related references

2.2 Running, stopping, and stepping through an application on page 2-45

3.1 Examining the target execution environment on page 3-74

2.14 Handling UNIX signals on page 2-62

2.15 Handling processor exceptions on page 2-64

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.3 About trace support

Arm Development Studio enables you to perform tracing on your application or system. Tracing enables you to non-invasively capture, in real-time, the instructions and data accesses that were executed. 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.

When a program fails, and the trace buffer is enabled, you can see the program history associated with the captured trace. 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.

Before the debugger can trace your platform, you must ensure that:

- You have a debug hardware agent, such as an Arm DSTREAM unit with a connection to a trace stream.
- The debugger is connected to the debug hardware agent.

Trace hardware

Processor 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. Unless using start/stop debug to observe the trace data, 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 problem is that data trace requires very high bandwidth as every load or store operation generates trace information. This is a problem because 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 several thousand lines of program trace.

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.

When no trace ranges are set, trace data for all virtual addresses is captured. When any trace ranges are set, trace capture is disabled by default, and is only enabled when within the defined ranges.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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. To set trace points in the source view, right-click in the margin and select the required option from the **Arm DS Breakpoints** context menu. To set trace points in the Disassembly view, right-click on an instruction and select the required option from the **Arm DS Breakpoints** context menu. Trace points are listed in the Breakpoints view. The following types of trace points are available:

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 DSTREAM, 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

3.4 About post-mortem debugging of trace data on page 3-79

4.1 Overview: Running Arm® Debugger from the command-line or from a script on page 4-81

Related tasks

4.5 Specifying a custom configuration database using the command-line on page 4-93

Related references

4.2 Command-line debugger options on page 4-82

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.4 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 Arm Development Studio command prompt, enter `debugger --help` to view the list of available options.

Related concepts

3.3 About trace support on page 3-76

4.1 Overview: Running Arm® Debugger from the command-line or from a script on page 4-81

Related tasks

4.5 Specifying a custom configuration database using the command-line on page 4-93

Related references

4.2 Command-line debugger options on page 4-82

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 4
Running Arm® Debugger from the operating system command-line or from a script

This chapter describes how to use Arm Debugger from the operating system command-line or from a script.

It contains the following sections:

- *4.1 Overview: Running Arm® Debugger from the command-line or from a script* on page 4-81.
- *4.2 Command-line debugger options* on page 4-82.
- *4.3 Run Arm® Development Studio IDE from the command-line to clean and build your projects* on page 4-89.
- *4.4 Running a debug session from a script* on page 4-91.
- *4.5 Specifying a custom configuration database using the command-line* on page 4-93.
- *4.6 Capturing trace data using the command-line debugger* on page 4-95.
- *4.7 Working with the debug server* on page 4-97.
- *4.8 Arm® Debugger command-line console keyboard shortcuts* on page 4-99.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.1 Overview: Running Arm® Debugger from the command-line or from a script

Arm Debugger can operate in a command-line only mode, with no graphical user interface.

This is very useful when automating debug and trace activities. By automating a debug session, you can save significant time and avoid repetitive tasks such as stepping through code at source level. This becomes particularly useful when you need to run multiple tests as part of regression testing.

This mode has the advantage of being extremely lightweight and therefore faster. However, by extension, it also lacks the enhancements that a GUI brings when connecting to a target device such as being able to see synchronization between your source code, disassembly, variables, registers, and memory as you step through execution.

If you want, you can drive the operation of Arm Debugger with individual commands in an interactive way. However, Arm Debugger when used from the command-line, is typically driven from scripts. With Arm Debugger, you might first carry out the required debug tasks in the graphical debugger. This generates a record of each debug task, which can then be exported from the **History** view as a (`.ds`) script.

You can edit scripts using the **Scripts** view. Alternatively, a debug script can be written manually using the `Arm Debugger commands` for reference.

Arm Development Studio also supports Jython (.py) scripts which provide more capability than the native Arm DS scripting language. These can be loaded into Arm Debugger to automate the debugger to carry out more complex tasks.

See *Command-line debugger options* on page 4-82 for syntax and instructions on how to launch Arm Debugger from the command line.

Related tasks

4.5 Specifying a custom configuration database using the command-line on page 4-93

Related references

4.2 Command-line debugger options on page 4-82

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.2 Command-line debugger options

You can use the `armdbg` command to run Arm Debugger from the command-line without a graphical user interface.

If you are using Windows, use the Arm Development Studio **Command Prompt**. On Linux, set the required environment variables, and use the UNIX shell.

Launch the command-line debugger using the following syntax:

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

Where:

`armdbg`

Invokes the Development Studio command-line debugger.

`--option <arg>`

The debugger option and its arguments. This can either be to configure the command-line debugger, or to connect to a target.

`...`

Additional options, if you need to specify any.

Note

When connected to your target, use any of the *Arm 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`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

--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. You can specify partial entries such as `"Arm Development Boards"` or `"Arm Development Boards::Versatile Express A9x4"` and press **Enter** to view the next possible entries.

For example, to connect to an Arm Versatile Express A9x4 target using DSTREAM and a USB connection, first use `--cdb-list` to identify the entries in the configuration database within `Arm Development Boards`, use:

```
armdbg --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"
```

--cdb-entry-param <arg>

Specifies connection parameters for the debugger:

- Use `arg` to specify the parameters and their values.
- Use `--cdb-list` to identify the parameters the debugger needs. Parameters that the debugger might need are:

  Connection

  Specifies the TCP address or the USB port number of the debug adapter to connect to.

  Address

  Specifies the address for a gdbserver connection.

  Port

  Specifies the port for a gdbserver connection.

  dtsl_options_file

  Specifies a file containing the DTSL options.

  model_params

  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:

```
--cdb-entry-param "Connection=TestTarget" --cdb-entry-param
"dtsl_options_file=my_dtsl_settings.dtslprops"
```

Group `model_params` together in one `--cdb-entry-param` parameter. Use spaces to separate the pairs of parameters and values, for example:

```
--cdb-entry-param model_params="-C cluster.cpu0.semihosting-enable=1 -C
cluster.cpu0.semihosting-cmd_line='hello world!'"
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`--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.
- Connection type refers to DSTREAM, so there is no `Connection type` when connecting to a model.

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

```
armdbg --cdb-list
```

For example, to list all the configuration database entries for the manufacturer NXP, use:

```
armdbg --cdb-list="NXP"
```

`--cdb-root <arg>`

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

Note

- To specify more than one configuration database, you must separate the directory paths using a colon (`:`) for Linux systems or a semicolon (`;`) for Windows systems.
- 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.
- 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.

`--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`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`--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.

`--launch-config <path>`

Start a debug connection using the command-line launch configuration file specified in `<path>`.

For example:

```
armdbg --launch-config "path/to/debugconfiguration.cli"
```

You can create a command-line launch configuration using the *Export tab* on page 6-266 in the **Debug Configurations** dialog box.

`--launch-config-connect-only true | false`

Specifies that the debugger only connect to the target when using the `--launch-config` option. The default is `--launch-config-connect-only false`.

You can use `--image` and `--stop_on_connect` option in combination with the `--launch-config-connect-only` option.

For example:

- To connect to the target without performing any other actions (such as loading images or running initialization scripts) as specified in the debug configuration file:

  ```
  armdbg --launch-config "path/to/debugconfiguration.cli" --launch-config-connect-only true
  ```

- To connect to the target without performing any other actions (such as loading images or running initialization scripts) as specified in the debug configuration file, and load an image:

  ```
  armdbg --launch-config "path/to/debugconfiguration.cli" --launch-config-connect-only true --image "path/to/image.axf"
  ```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`--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` or `--semihosting-error`

Specifies a file to write semihosting `stderr`.

`-i` or `--semihosting-input`

Specifies a file to read semihosting `stdin`.

`-o` or `--semihosting-output`

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`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`--server`

Specifies whether to start the debugger as a server to connect remotely.

Note

You cannot use the debugger interactively (using the `--interactive` option) when using it as a server.

When started as a server, by default, the debugger allocates a free port and listens on all available addresses.

You can specify an address or port by specifying them in the `[addr:]port` format.

For example:

- If you want to make the debugger listen on `port 1234` on all available addresses, you can specify them using the following forms of the command:

  ```
  armdbg --cdb-entry ... --server 1234
  ```

  ```
  armdbg --cdb-entry ... --server:1234
  ```

  ```
  armdbg --cdb-entry ... --server *:1234.
  ```
- If you want to bind to a particular address, you can specify an address or host name using the following forms of the command:

  ```
  armdbg --cdb-entry ... --server localhost:1234
  ```

  ```
  armdbg --cdb-entry... --server 10.2.2.2:1234
  ```

See *Working with the debug server* on page 4-97 for more information.

`--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 you can debug with Arm Debugger.

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, for Linux application debug or when using DSTREAM, 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

To connect to the Arm FVP Rainier model and specify an image to load, use:

```
armdbg --cdb-entry "Arm FVP::Morello::Bare Metal Debug::Bare Metal Debug::Rainierx4 Multi-
Cluster SMP" --image "path/to/image.axf"
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

To exit the connection, enter the `quit` debugger command.

Related references

4.8 Arm® Debugger command-line console keyboard shortcuts on page 4-99
5.1 Exporting Arm® Debugger commands generated during a debug session on page 5-101
2.16 Using semihosting to access resources on the host computer on page 2-66

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.3 Run Arm® Development Studio IDE from the command-line to clean and build your projects

You can run Arm Development Studio IDE from the command-line to clean and build your projects. This might be useful when you want to create scripts to automate build procedures.

Before you begin, make sure that the Arm Development Studio IDE session is closed.

Use the Arm Development Studio command-line console to load the Arm Development Studio IDE, `make`, and other utilities on your `PATH` environment variable. To launch the console:

- On Windows, select **Start** > **All Programs** > **Arm Development Studio** > **Arm DS Command Prompt**.
- On Linux, run `<install_directory>/bin/suite_exec <shell>` to open a shell.

Run `armds_idec.exe` (on Windows) or `armds_ide` (on Linux) with the following Arm Development Studio IDE arguments as required.

Table 4-1 Arm DS IDE arguments

Argument	Description
`-nosplash`	Disables the Arm Development Studio IDE splash screen.
`--launcher.suppressErrors`	Causes errors to be printed to the console instead of being reported in a graphical dialog box.
`-application com.arm.cmsis.pack.project.headlessbuild`	Mandatory argument telling Arm Development Studio IDE to run the headless builder.
`-data <workspaceDir>`	Specify the location of your workspace.
`-import <projectDir>`	Import the project from the specified directory into your workspace. Use this option multiple times to import multiple projects.
`-build <projectName>[/<configName>] | all`	Build the project with the specified name, or all projects in your workspace. By default, this argument builds all the configurations within each project. You can limit this action to a single configuration, such as `Debug` or `Release`, by specifying the configuration name immediately after your project name, separated with '`/`'. Use this option multiple times to build multiple projects.
`-cleanBuild <projectName>[/<configName>] | all`	Clean and build the project with the specified name, or all projects in your workspace. By default, this argument cleans and builds all the configurations within each project. You can limit this action to a single configuration, such as `Debug` or `Release`, by specifying the configuration name immediately after your project name, separated with '`/`'. Use this option multiple times to clean and build multiple projects.
`-cmsisRoot <path>`	Set the path to the CMSIS Packs root directory

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

On Windows, you must run `armds_idec.exe` from either the **Arm DS Command Prompt**, or directly from the `<install_directory>/bin` directory. Do not run the `armds_idec.exe` executable that is in the `<install_directory>/sw/eclipse` directory.

The `armds_idec.exe` executable in `<install_directory>/bin` acts as a wrapper for `armds_idec.exe` in `<install_directory>/sw/eclipse`. Running the executable from the `<install_directory>/bin` directory sets up the Arm Development Studio environment (paths, environment variables, and other similar items) in the same way as the **Arm DS Command Prompt**.

For example:

```
"C:\Program Files\Arm\Development Studio <version>\bin\armds_idec.exe" -nosplash -
application com.arm.cmsis.pack.project.headlessbuild -data "C:\path\to\your
\workspace" -cleanBuild startup_Cortex-R8
```

Examples

On Windows, to list and view the full set of available options, use the command:

```
armds_idec.exe -nosplash -consoleLog --launcher.suppressErrors -application
com.arm.cmsis.pack.project.headlessbuild -help
```

On Windows, to clean and build *all* the projects in a specific workspace, use the command:

```
armds_idec.exe -nosplash -application com.arm.cmsis.pack.project.headlessbuild -data
C:<\path\to\workspace> -cleanBuild all
```

On Linux, to build the Release configuration of project **MyProject** in a specific workspace, use the command:

```
armds_ide -nosplash -application com.arm.cmsis.pack.project.headlessbuild -data </
path/to/workspace> -build MyProject/Release
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.4 Running a debug session from a script

To automate a debug session from a script, create a text file with a `.ds` file extension and list, line-by-line, the debugger commands that you want to execute. Then, use the `debugger` command to run the script.

Debugger script format

The script is a text file with a `.ds` file extension. The debugger commands are listed one after the other in the file.

Things to remember when you create a `.ds` script file.

- The script file must contain only one command on each line.
- If required, you can add comments using `#`.
- Commands are not case-sensitive.
- The `.ds` file extension must be used for an Arm Debugger script.

Note

If you are using the Arm Development Studio graphical user interface (GUI) to perform your debugging, a full list of all the Arm Debugger commands generated during a debug session is recorded in the **History** view. You can select the commands that you want in your script file and right-click and select **Save selected lines as a script…** to save them to a file.

Examples

A simple sample script file is shown below:

```
# Filename: myScript.ds
# Initialization commands
load file "struct_array.axf"  # Load image and 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)
delete 1                      # Delete breakpoint number 1
delete 2                      # Delete breakpoint number 2
```

Running an Arm® Development Studio script

After creating the script, use the **debugger** command to run the script using the command-line interface.

On Windows, use the **Arm Development Studio Command Prompt**. On Linux, set the required environment variables, and use the UNIX shell.

There are two scenarios where you might run scripts:

- You have set up your target and it is connected to Development Studio.

  In this case, use the `source` command to load your script.
- You are yet to configure the target and connect to it.

  In this case, you have to use the appropriate debugger options and arguments to configure and connect to your target. Along with the configuration options, use the **--script=filename** option to run your script.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For example:

```
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=TCP:10.5.20.64" --cdb-entry-param "dtsl_options_file=C:\
\Arm_DS_Workspace\\my_dtsl_settings.dtslprops" --script= C:\\Arm_DS_Workspace\
\my_script.txt
```

See *Specifying a custom configuration database using the command-line* on page 4-93 and *Capturing trace data using the command-line debugger* on page 4-95 for examples of how debugger options and arguments are used.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.5 Specifying a custom configuration database using the command-line

Some targets might not be available in the default Arm Development Studio configuration database. For example, a custom target which is available only to you. In this case, you can specify a custom configuration database which contains the details for your target.

Use the Arm Debugger command-line options to view the entries in your custom configuration database and determine the connection names. Then, use additional commands and options to specify the details of your configuration database and connect to your target.

Procedure

1. Launch an Arm Development Studio command-line console.
   a. Add the `<install_directory>/bin` directory to your `PATH` environment variable. If it is already configured, then you can skip this step.
   b. Open a terminal.
2. List the entries in the user-specified configuration databases. Use the following syntax:

   `debugger --cdb-list --cdb-root path_to_cdb1[:path_to_cdb2]`. For example, `debugger --cdb-list --cdb-root \\Arm_DS_Workspace\\MyConfigDB1:Arm_DS_Workspace\\MyConfigDB2`.

   Where:

   `debugger`

   Is the command to invoke Arm Debugger.

   `--cdb-list`

   Is the option to list the entries in the configuration database.

   `--cdb-root`

   Is the option to specify the path to one or more configuration databases.

   `path_to_cdb1 and path_to_cdb2`

   Are the directory paths to the configuration databases.

   Note

   Arm Debugger processes configuration databases from left to right. The information from processed databases are replaced with information from 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 end of the list on the command-line.

3. When you have determined the details of your target, use the following command-line syntax to connect to the target in your custom configuration database:

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

   For example:

   ```
   debugger --cdb-entry "Arm FVP::Morello::Bare Metal Debug::Bare Metal
   Debug::Rainierx4 Multi-Cluster SMP" --cdb-root /path/to/custom/database
   ```

   Where:

   `debugger`

   Is the command to invoke Arm Debugger.

   `--cdb-entry`

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Specifies the target to connect to.

`Manufacturer::Platform::Project type::Execution environment::Activity::Connection type`

Correspond to the entries in your custom configuration database. The entries have a tree data structure, where each entry has entries within it.

`--cdb-root`

Is the command to specify your custom configuration database.

`path_to_cdb1`

Is the directory path to your 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.
- 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

4.1 Overview: Running Arm® Debugger from the command-line or from a script on page 4-81

Related references

4.2 Command-line debugger options on page 4-82

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.6 Capturing trace data using the command-line debugger

To capture trace data using the command-line debugger, you must enable the relevant trace options in the Debug and Trace Services Layer (DTSL) configuration settings in Arm Development Studio.

For this task, it is useful to setup the DTSL options using the graphical interface of debugger.

When you have setup the DTSL options, the debugger creates a file that contains the DTSL settings. You can then use this file when invoking the command-line debugger to perform trace data capture tasks, for example, run a script which contain commands to `start` and `stop` trace capture.

An example script file might contain the following commands:

```
loadfile C:\Arm_DS_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
```

Procedure

1. Using the graphical interface of Arm Debugger, open the **Debug Configurations** dialog box for your trace-capable target.
2. In the **Connections** tab, under **DTSL options**, click **Edit** to open the **DTSL Configuration Editor** dialog box.
   a. Select a **Trace capture method** in the **Trace Buffer** tab.
   b. If required, select the relevant tab for your processor, and then select **Enable core trace**.
   c. Click **Apply** to save the settings.

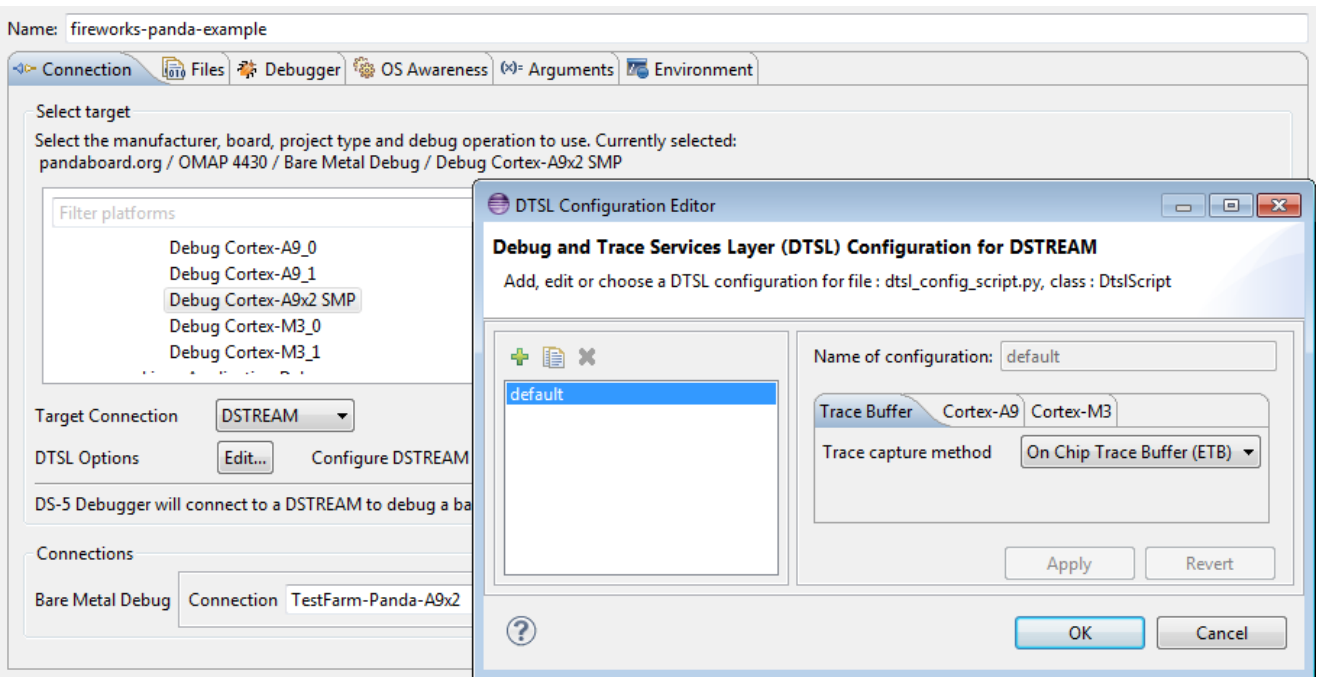


Figure 4-1 Enable trace in the DTSL options

These settings are stored in a `*.dtslprops` file in your workspace. The filename is shown in the **Name of configuration** field in the dialog box. In this example, the settings are stored in the `default.dtslprops` file.

3. Copy the DTSL settings file, for example `default.dtslprops` , from your workspace to a different directory and change its name, for example to `my_dtsl_settings.dtslprops`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4. Open the **Arm DS Command Prompt** and:
   a. Use the `--cdb-list` command-line argument to identify your target name and configuration.
   b. Invoke the command-line debugger with your target name and configuration and 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:\\Arm_DS_Workspace\
\my_dtsl_settings.dtslprops"
```

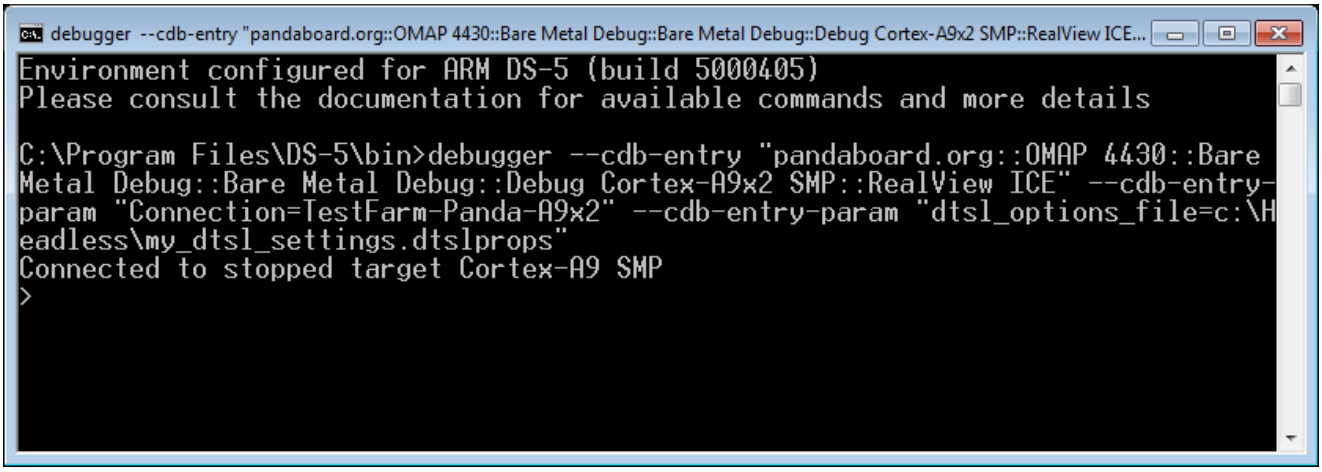


Figure 4-2 Command-line debugger connection with DTSL options enabled.

The debugger connects to your target. You can now issue commands to `load` and `run` an image, and also to `start` and `stop` trace capture.

Tip

If you create a script file containing trace capture commands, you can specify this script file when invoking the command-line debugger, for example:

```
debugger --cdb-entry "pandaboard.org::OMAP 4430::Bare Metal Debug::Bare MetalDebug::Debug
Cortex-A9x2 SMP::RealView ICE" --cdb-entry-param "Connection=TestFarm-Panda-A9x2" --cdb-
entry-param "dtsl_options_file=C:\\Arm_DS_Workspace\\my_dtsl_settings.dtslprops" --script=C:\
\Arm_DS_Workspace\\my_script.txt.
```

Related concepts

4.1 Overview: Running Arm® Debugger from the command-line or from a script on page 4-81

Related references

4.2 Command-line debugger options on page 4-82
6.43 Debug Configurations - Connection tab on page 6-251
6.50 DTSL Configuration Editor dialog box on page 6-267

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.7 Working with the debug server

You can run Arm Debugger as a server to connect remotely. You can then connect to the debugger from a different host or from a different process on the same host using the Telnet protocol.

Starting the debug server

Use the `--server` debugger option to start the debug server.

For example, to start the debug server, connect to the Arm FVP Rainier model and specify an image to load:

```
debugger --cdb-entry "Arm FVP::Morello::Bare Metal Debug::Bare Metal Debug::Rainierx4 Multi-
Cluster SMP" --image "/path/to/image.axf" --server
```

By default, Arm Debugger assigns a free port and listens on all available addresses.

If necessary, you can bind a port and address by passing the `[addr:]port` parameters to the `--server` command option.

For example:

- To listen on port 1234, you can use any of the following options:

```
debugger --cdb-entry "..." --server 1234
debugger --cdb-entry "..." --server :1234
debugger --cdb-entry "..." --server *:1234
```

- To bind to a particular address, you can specify an address or host name along with the port name:

```
debugger --cdb-entry "..." --server localhost:1234
debugger --cdb-entry "..." --server 10.2.2.2:1234
```

Connecting a debug client

You can connect your client to an Arm Debugger server session using the Telnet protocol. To connect to a debug server session using Telnet, use the Telnet `open` command.

For example:

- If you want to connect to a debug server session on the host 10.2.2.2 at port 1234, at the Telnet prompt, enter:

```
open 10.2.2.2 1234
```

- If you want to connect to a debug server session on your local host at port 1234, at the Telnet prompt, enter:

```
open localhost 1234
```

For details of Telnet commands, check the Telnet command documentation.

After connecting to the debug server, you can use the *debugger commands* to debug your application.

Note

You cannot use the debugger interactively (using the `--interactive` option) when using it as a server.

Disconnecting a debug client

You can disconnect a debug client by closing the Telnet session. Closing the Telnet session only disconnects the client, but does not stop the debug session.

You can reconnect to a debug session at later time by reopening a connection to the debug server. When reconnected, the debug session resumes from where it was left off.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Stopping the debug server

To stop the debug server session:

- On the client use the `quit` command.
- On the debug server host, use the `CTRL+C` keys on your keyboard.

Related references

4.2 Command-line debugger options on page 4-82

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.8 Arm® Debugger command-line console keyboard shortcuts

Arm Debugger provides editing features, a command history, and common keyboard shortcuts to use when debugging from the command-line.

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 references

4.2 Command-line debugger options on page 4-82

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 5
Debugging with Scripts

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

It contains the following sections:

- *5.1 Exporting Arm® Debugger commands generated during a debug session* on page 5-101.
- *5.2 Creating an Arm® Debugger script* on page 5-102.
- *5.3 Creating a CMM-style script* on page 5-103.
- *5.4 Support for importing and translating CMM scripts* on page 5-104.
- *5.5 About Jython scripts* on page 5-106.
- *5.6 Jython script concepts and interfaces* on page 5-108.
- *5.7 Creating Jython projects in Arm® Development Studio* on page 5-110.
- *5.8 Creating a Jython script* on page 5-114.
- *5.9 Running a script* on page 5-116.
- *5.10 Use case scripts* on page 5-118.
- *5.11 Metadata for use case scripts* on page 5-119.
- *5.12 Definition block for use case scripts* on page 5-120.
- *5.13 Defining the Run method for use case scripts* on page 5-122.
- *5.14 Defining the options for use case scripts* on page 5-123.
- *5.15 Defining the validation method for use case scripts* on page 5-126.
- *5.16 Example use case script definition* on page 5-127.
- *5.17 Multiple use cases in a single script* on page 5-128.
- *5.18 usecase list command* on page 5-129.
- *5.19 usecase help command* on page 5-130.
- *5.20 usecase run command* on page 5-131.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.1 Exporting Arm® Debugger commands generated during a debug session

A full list of all the Arm Debugger commands generated during the current debug session is recorded in the **History** view. Before closing Eclipse, you can select the commands that you want in your script file and click on **Export the selected lines as a script file** to save them to a file.

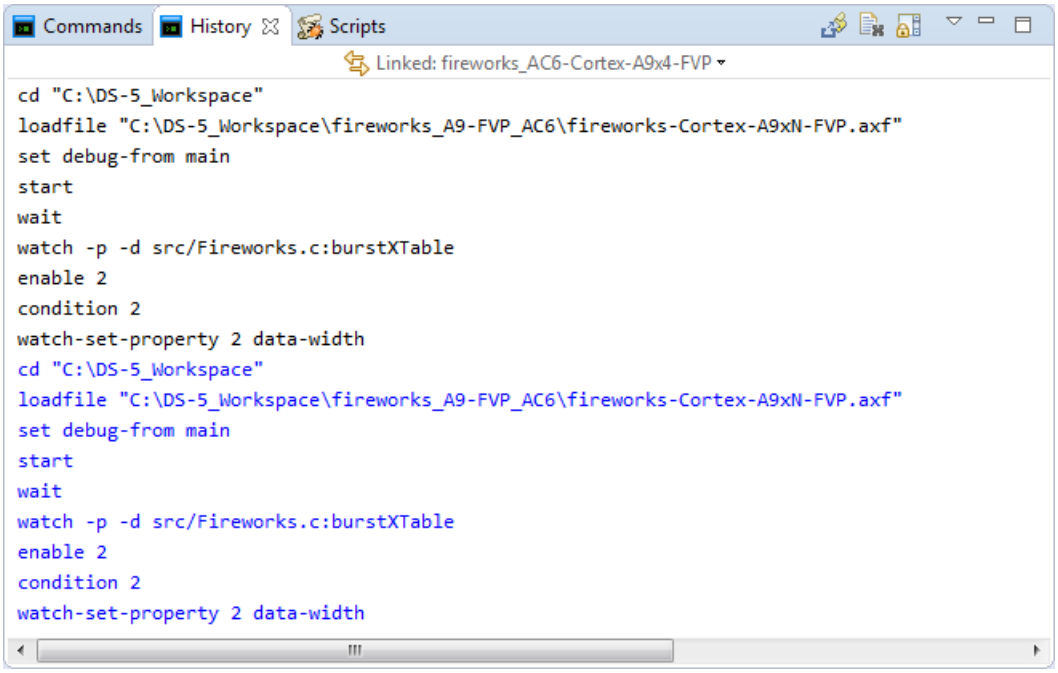


Figure 5-1 Commands generated during a debug session

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.2 Creating an Arm® Debugger script

Shows a typical example of an Arm 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
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.3 Creating a CMM-style script

Arm Development Studio provides a small subset of CMM-style commands which you can use to create a CMM-style script.

The debugger script file must conform to the following standards:

- The script file must contain only one command on each line. If necessary, you can add comments using the `//` tags.
- The `.cmm` or `.t32` file extension must be used to identify a CMM-style script.

After creating your script, you must use the Arm Debugger *source* command to load and run the script.

The example below shows a typical CMM-style script.

Examples

```
// 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
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.4 Support for importing and translating CMM scripts

You can use the import and translate CMM script feature in Arm Development Studio to reuse existing CMM scripts for your platform. After the translation is complete, the CMM script is converted into a Jython script which you can then run in Development Studio.

During the *CMM file import process* on page 5-104 in Development Studio:

- Translates and maps the CMM commands to their equivalent Jython commands or calls to the Arm Debugger API.

  If a CMM command translation is not supported, it is marked up as a stub function in the resultant Jython script. Using the information contained in the stub function, you can implement your own functionality for unsupported CMM commands.

  See *Supported CMM commands for translations* on page 5-105 for the list of commands that are currently translated.
- Translates flow control statements in CMM such as `IF`, `ELSE IF`, `ELSE`, `WHILE`, and `RePeaT` to their Jython equivalents.
- Also imports and process complex nested commands and functions, CMM variables, and variable assignments. CMM subroutines that are specified with labels, ENTRY commands for subroutine parameters, and RETURN statements to return values from subroutines are converted directly to valid Jython subroutine syntax.

Logging and error handling

For every translated CMM script, the translation process also creates a log file and places it in the same folder as the translated script file. The log contains a summary, the detailed breakdown of the translated elements, and details of any errors, if any.

To view the log, right-click on the imported CMM script and select **Show log for the translation of** `<YourFilename.cmm>`.

This section contains the following subsections:

- *5.4.1 Importing and translating a CMM script* on page 5-104.
- *5.4.2 Supported CMM commands for translations* on page 5-105.

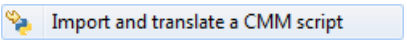
5.4.1 Importing and translating a CMM script

If you have existing CMM scripts for your platform, you can use the script import feature in Arm Development Studio to import and translate them.

Prerequisites

- CMM scripts must have the .cmm extension to use the Development Studio script import feature.
- Scripts are associated with a debug configuration. When importing or editing scripts, first select the associated debug configuration for your target in the **Debug Control** view.

Procedure

1. To import a CMM script into Development Studio, in the **Scripts** view, click **Import a script or directory** and select 

.

   Note

   You can also drag and drop the script to be imported into the script view, either from the **Project Explorer** view or your operating system file explorer.

2. Browse and select the CMM script that you want to import, and click **Open**.
3. Choose a location for the translated CMM script and click **OK**. To view the imported CMM script, in the **Scripts** view, expand the **CMM** node and locate your script.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Related references

5.4.2 Supported CMM commands for translations on page 5-105
6.25 Scripts view on page 6-213

5.4.2 Supported CMM commands for translations

The following CMM commands are supported by the translation process.

- `Break.Set` - Set a breakpoint or watchpoint.
- `Break.Delete` - Delete an address or symbol breakpoint or watchpoint.
- `pBreak.Delete` - Delete a source level breakpoint or watchpoint.
- `Core` or `Core.Select` - Switch to the numbered core in the connection.
- `Data.Load` - Load an image.
- `Data.Load.Elf` - Load an ELF format file.
- `Data.Load.Bin` - Load a binary format file.
- `Data.Load.auto` - Load a file after automatically detecting the file format.
- `Data.Long` - Read a 32-bit value from memory.
- `Data.Set` - Write byte-wise to memory.
- `Go.Direct` or `Go` - Continue to run the core (additionally set temporary breakpoints before running).
- `Print` - Print the output.
- `Register` - Read a register.
- `Register.Set` - Write a register.
- `System.ResetTarget` or `Sys.ResetTarget` - Reset the target on the current connection.
- `Wait` - Wait until a specified condition is true or for a set time period.

For detailed descriptions and formats for these individual commands, check the documentation provided with your debugger.

Related tasks

5.4.1 Importing and translating a CMM script on page 5-104

Related references

5.4 Support for importing and translating CMM scripts on page 5-104
6.25 Scripts view on page 6-213

Related tasks

5.4.1 Importing and translating a CMM script on page 5-104

Related references

6.25 Scripts view on page 6-213
5.4.2 Supported CMM commands for translations on page 5-105

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.5 About Jython scripts

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.getIdentifier(), registers)
# Output:
#         Identifier: 1, Registers: Core::R0 = 0x00000010 (16), Core::R1 = 0x00000000 (0),
Core::R2 = 0x0000A4A4 (42148)
#         ...
```

Related tasks

5.7.1 Creating a new Jython project in Arm® Development Studio on page 5-110

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.7.2 Configuring an existing project to use the Arm® Development Studio Jython interpreter on page 5-113
5.8 Creating a Jython script on page 5-114
5.9 Running a script on page 5-116

Related references

5.6 Jython script concepts and interfaces on page 5-108
6.42 Script Parameters dialog box on page 6-250

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.6 Jython script concepts and interfaces

Summary of important Arm Debugger Jython interfaces and concepts.

Imports

The debugger module provides a Debugger class for initial access to Arm 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 Arm 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 Arm Development Studio 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 Arm Development Studio command 'print $ENTRYPOINT' and print the result
print ec.executeDSCommand('print $ENTRYPOINT')
```

Error Handling

The methods on the debugger Jython interfaces throw DebugException whenever an error occurs. You can catch exceptions to handle errors in order to provide more information. Here is an example:

```
# Catch a DebugException and print the error message
try:
ec.getRegisterService().getValue('ThisRegisterDoesNotExist')
except DebugException, de:
print "Caught DebugException: %s" % (de.getMessage())
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information on Arm Debugger Jython API documentation select **Help Contents** from the **Help** menu.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.7 Creating Jython projects in Arm® Development Studio

To work with Jython scripts in Arm Development Studio, the project must use Arm DS Jython as the interpreter. You can either create a new Jython project in Development Studio with Arm DS Jython set as the interpreter or configure an existing project to use Arm DS Jython as the interpreter.

This section contains the following subsections:

- *5.7.1 Creating a new Jython project in Arm® Development Studio* on page 5-110.
- *5.7.2 Configuring an existing project to use the Arm® Development Studio Jython interpreter* on page 5-113.

5.7.1 Creating a new Jython project in Arm® Development Studio

Use these instructions to create a new Jython project and select Arm DS Jython as the interpreter.

Procedure

1. Select **File** > **New** > **Project...** from the main menu.
2. Expand the **PyDev** group.
3. Select **PyDev Project**.

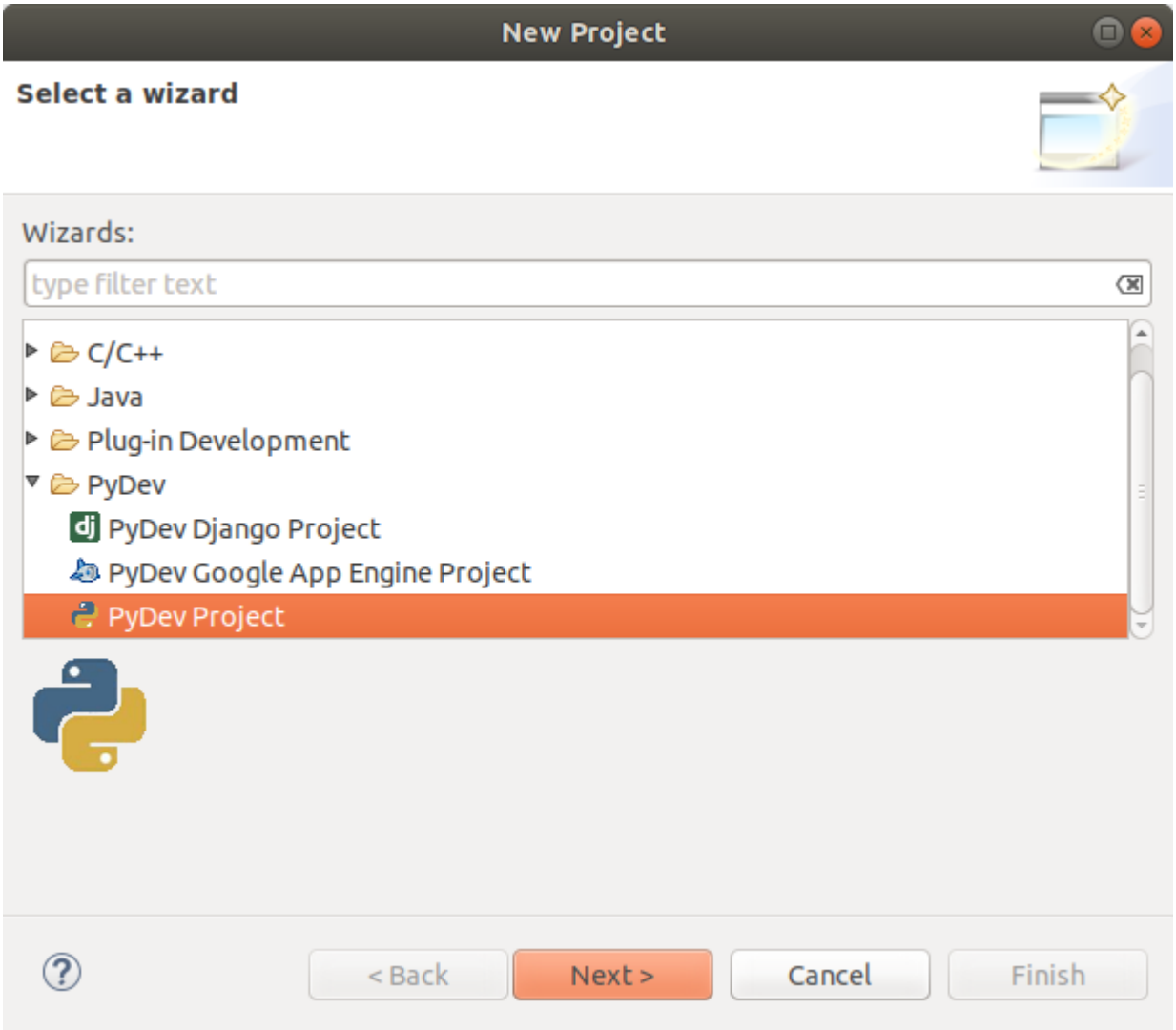


Figure 5-2 PyDev project wizard

4. Click **Next**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 **Arm DS Jython**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

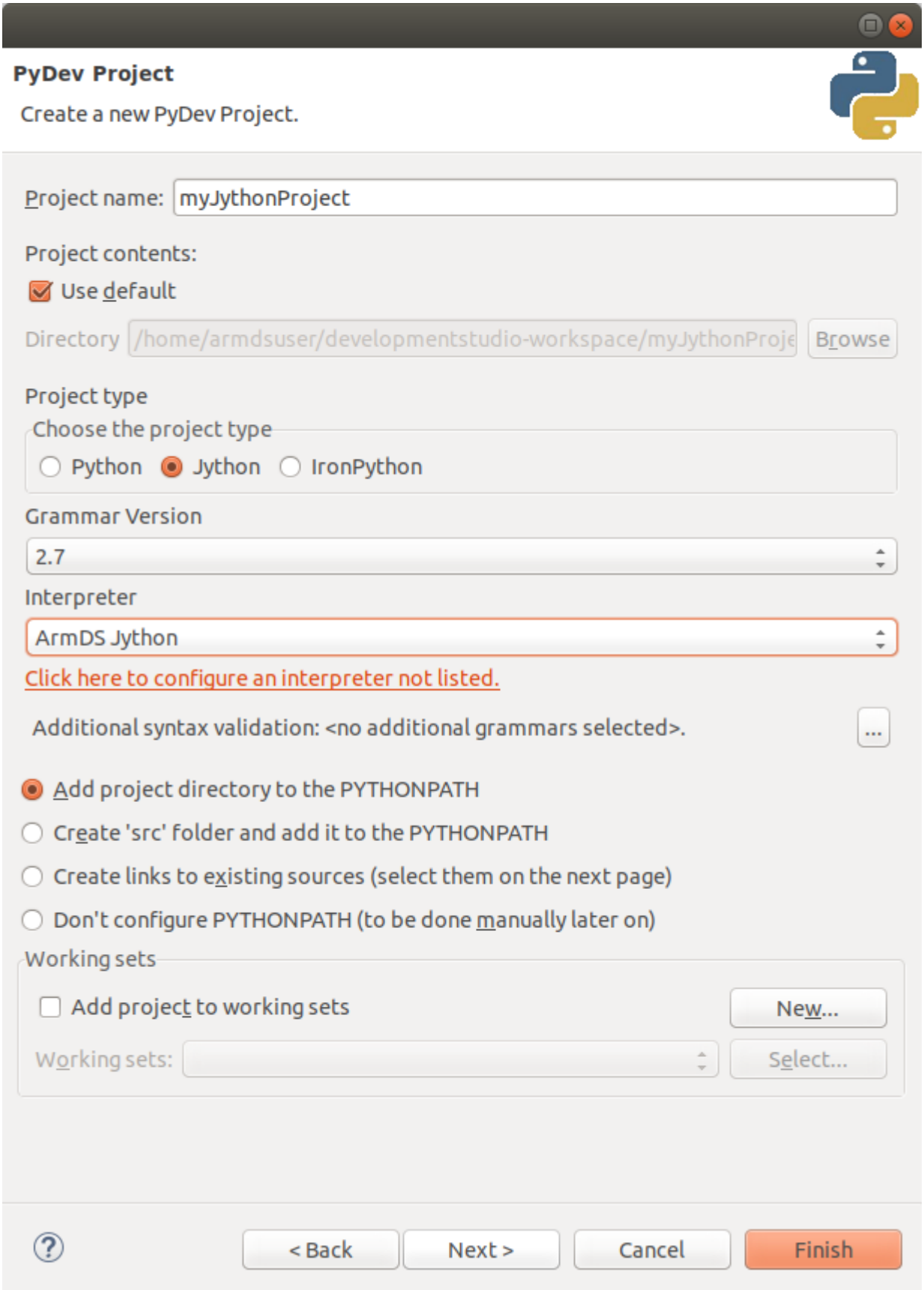


Figure 5-3 PyDev project settings

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

Related concepts

5.5 About Jython scripts on page 5-106

Related tasks

5.7.2 Configuring an existing project to use the Arm® Development Studio Jython interpreter on page 5-113
5.8 Creating a Jython script on page 5-114
5.9 Running a script on page 5-116

Related references

5.6 Jython script concepts and interfaces on page 5-108
6.42 Script Parameters dialog box on page 6-250

5.7.2 Configuring an existing project to use the Arm® Development Studio Jython interpreter

Use these instructions to configure an existing project to use Arm DS 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 **Arm DS 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

5.5 About Jython scripts on page 5-106

Related tasks

5.7.1 Creating a new Jython project in Arm® Development Studio on page 5-110
5.8 Creating a Jython script on page 5-114
5.9 Running a script on page 5-116

Related references

5.6 Jython script concepts and interfaces on page 5-108
6.42 Script Parameters dialog box on page 6-250

Related tasks

5.7.1 Creating a new Jython project in Arm® Development Studio on page 5-110
5.7.2 Configuring an existing project to use the Arm® Development Studio Jython interpreter on page 5-113

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.8 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.

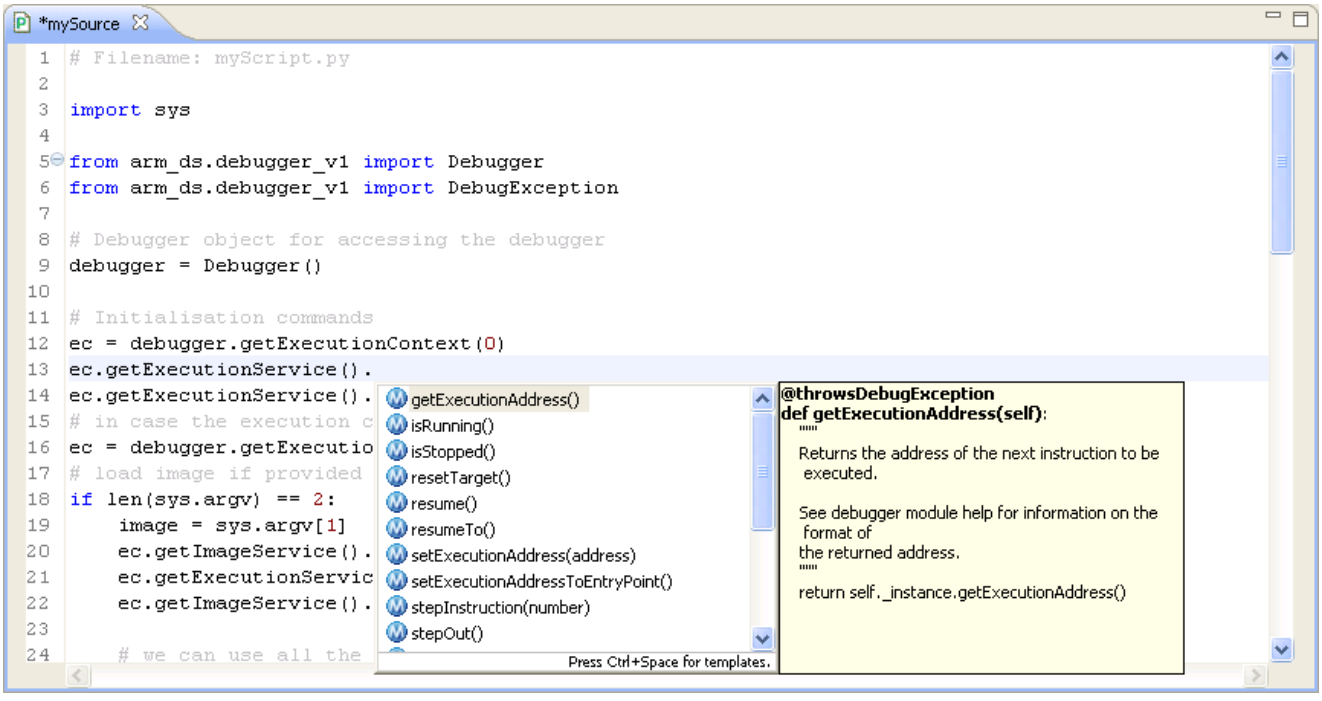


Figure 5-4 Jython auto-completion and help

4. Edit the file to contain the desired scripting commands.
5. Run the script in the debugger.

Next Steps

You can now view an entire Jython interface in the debugger. To open the source code that implements a debugger object or interface, **Ctrl+Click** on the object or interface of interest.

Related concepts

5.5 About Jython scripts on page 5-106

Related tasks

5.7.1 Creating a new Jython project in Arm® Development Studio on page 5-110
5.7.2 Configuring an existing project to use the Arm® Development Studio Jython interpreter on page 5-113
5.9 Running a script on page 5-116

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Related references

5.6 Jython script concepts and interfaces on page 5-108
6.42 Script Parameters dialog box on page 6-250

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.9 Running a script

Use the **Scripts** view to run a script in Arm Development Studio. You can run a script file immediately after the debugger connects to the target.

Procedure

1. To run a script from the Arm Development Studio IDE:
   a. Launch Arm Development Studio IDE.
   b. Configure a connection to the target.

   Note

   Arm Debugger configurations can include the option to run a script file immediately after the debugger connects to the target. To do this, in the **Debugger** tab of the Development Studio **Debug Configuration** dialog box, select the appropriate script file option. See *Debug Configurations -Debugger tab* on page 6-258 for more information.

   c. Connect to the target.
2. After your target is up and running, select the scripts that you want to execute and click the **Execute Selected Scripts** button.

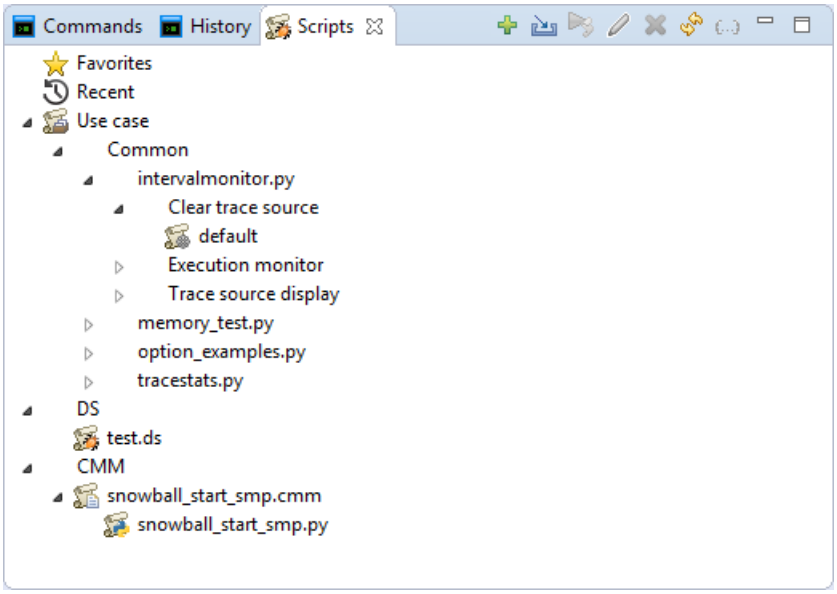


Figure 5-5 Scripts view

Note

Debugger views are not updated when commands issued in a script are executed.

Related concepts

5.5 About Jython scripts on page 5-106

Related tasks

5.7.1 Creating a new Jython project in Arm® Development Studio on page 5-110

5.7.2 Configuring an existing project to use the Arm® Development Studio Jython interpreter on page 5-113

5.8 Creating a Jython script on page 5-114

Related references

5.1 Exporting Arm® Debugger commands generated during a debug session on page 5-101

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.2 Creating an Arm® Debugger script on page 5-102
5.3 Creating a CMM-style script on page 5-103
6.6 Commands view on page 6-148
6.15 History view on page 6-177
5.6 Jython script concepts and interfaces on page 5-108
6.42 Script Parameters dialog box on page 6-250

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.10 Use case scripts

Use case scripts provide an extension to existing scripts that can be used in Arm Development Studio.

Development Studio provides many pre-defined platform and model configurations in the configuration database for connecting to and debugging a range of targets. The configuration database can be extended using the Platform Configuration Editor or Model Configuration editor.

When a connection to a configuration is established, you can then run a use case script to invoke custom behavior. You can use Jython and access the Arm Debug and Trace Services Layer (DTSL) libraries and debugger APIs. You can use use case scripts to provide a complex trace configuration, trace pin multiplexing or configure multiple CoreSight components within a single script.

Use case scripts provide a simple but highly configurable way to implement user-defined functionality without having to write any boilerplate code for setting up, configuring and maintaining a script. Use case scripts still have access to the full scripting APIs.

The benefits of use case scripts include:

Use case scripts work with Arm Development Studio

There are built-in functions within Development Studio to query, list, and run use case scripts including those in configuration databases and platform specific scripts. Use case scripts are analyzed and validated before they are run. A clear error message is reported if there are errors in the construction of the script.

Built-in functionality

Use case scripts take a set of options to configure the values given to the script on the command line and have a built-in mechanism to save and load sets of options. A trace configuration for a particular target or a complex set of options to recreate a bug can be saved and loaded when the script is run.

Flexibility

Use case scripts provide an easy way of changing the number of arguments, options, and names of methods as the script develops. You can define positional arguments to use case scripts. It is easy to add, remove or modify positional arguments. When running the script from the command line, you must specify values for the positional arguments. Multiple use cases are supported within a single use case script to allow sharing of common code, where each use case provides a single piece of functionality.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.11 Metadata for use case scripts

A use case definition block is a comment block that is usually at the start of the script.

The definition block can define various metadata:

- The title of the use case.
- A brief description of the use case.
- Where the options must be retrieved from for the use case.
- The method to validate the use case.
- A multi-line help text which can provide usage text and a more verbose description of the use case.
- The entry point to the use case.

Only the entry point is required to define a use case. Other metadata definitions are optional.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.12 Definition block for use case scripts

Each use case definition block must begin with a `USECASE` header line to define the start of a use case. Without the header, Arm Development Studio does not recognize the script as a use case script.

Tags describe the content of a use case. Each tag is surrounded by dollar (`$`) signs to distinguish them from any other text. All tags are defined `$<tag>$ <value>` with the exception of the `$Help$` tag. If a tag is defined in the use case definition block it can only be present once. Duplicate names of tags are not accepted in a single use case.

Only the `$Run$` tag which describes the entry point or main method to the use case needs to be defined for a valid use case. To report meaningful help when searching for use case scripts on the Arm Development Studio command-line, it is recommended that you also define the `$Title$` and `$Description$` tags in each use case.

Run

The `$Run$` tag specifies the name of the entry point to a single use case. When you run a use case on the command line, it calls the method with the `$Run$` tag. For details of how to define the entry point, and how to supply additional arguments to the method, see *Defining the Run method for use case scripts* on page 5-122.

Example

```
...
$Run$ mainMethod
...
```

Title

The `$Title$` tag specifies the title in a use case definition block. This is a single line string which is the title of this use case script and is displayed when searching for use case scripts on the command-line.

Example

```
...
$Title$ Usecase Title
...
```

Description

The `$Description$` tag specifies the description in a use case definition block. This is a single line string which is the description of this use case and is displayed when searching for use case scripts on the command-line.

Example

```
...
$Description$ A brief description of this use case
...
```

Options

The `$Options$` tag specifies a method within the use case script, which can be called to retrieve a list of options. For a description of how to define the options function, and how to construct a list of options, see *Defining the options for use case scripts* on page 5-123.

Example

```
...
$Options$ myOptions
...
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Validation

The `$Validation$` tag specifies the validation method in the use case definition block. You must specify this to validate the options provided when the script is run. For a description of how to define the validation function, see *Defining the validation method for use case scripts* on page 5-126.

Example

```
...
$Validation$ myValidation
...
```

Help

The use of `$Help$` tags is slightly different from the use of other tags. The `$Help$` tag enables writing a multi-line block of text, which appears in the output when a user requests help about the use case script. This can be used to provide usage description, parameters for the use case scripts, or a more verbose help text.

To define a block of help text, enclose the block in `$Help$` tags. Formatting, such as new lines and spaces are preserved in the `$Help$` block. Everything, including the definition of other tags, are consumed within a `$Help$` block. The `$Help$` block must be completed before other tags or code is defined.

Example

```
$Help$
This is part of the help text
...
$Help$
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.13 Defining the Run method for use case scripts

The method named in the `$Run$` tag must have at least one parameter, called `options`, which provides access to the script options.

Examples

```
...
$Run$ mainMethod
...
```

```
def mainMethod(options):
   print "Running the main method"
```

It is possible to define an entry point to the use case that requires more than one parameter. The definition of the `$Run$` tag only defines the function name. The definition of the function within the script defines how many parameters are needed.

Examples

```
...
$Run$ main
...
```

```
# main method with positional arguments
def main(options, filename, value):
   print "Running the main method"
   # Using the positional arguments supplied to the script.
   print "Using file: " + filename + " and value: " + value
```

In this example `main` requires two parameters, `filename` and `value` which must be supplied on the command-line when the use case script is run.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.14 Defining the options for use case scripts

Each use case has a list of options which specify a set of values which can be supplied on the command-line to a particular use case to configure what values are supplied when use case is run.

When you define options, you can group or nest them to organize and provide a more descriptive set of options.

The method which provides the options is defined with the `$Options$` tag in a use case definition block. The `$Options$` tag provides the name of a method in the script which returns a single list of options.

Examples

```
...
$Options$ myOptions
...
```

```
def myOptions():
   return [list_of_options]
```

It is important that the function that supplies the options takes no arguments and returns a list of options in the same format as described below. If the `$Options$` tag is defined, and the function named in this tag is not present in the script or the function used to supply the options takes arguments, then an error occurs when running the script.

There are five option types which can be used to define options to a use case script. These are:

- UseCaseScript.booleanOption
- UseCaseScript.enumOption
- UseCaseScript.radioEnumOption
- UseCaseScript.integerOption
- UseCaseScript.stringOption

All of these options require:

- A variable name, that is used to refer to the option in a use case script
- A display name
- A default value that must be of the same type as the defined option

UseCaseScript.booleanOption

Describes a boolean. Possible values are `True` or `False`.

UseCaseScript.integerOption

Describes a whole integer value such as: 4, 99 or 100001. In addition to the fields required by all options, a minimum and a maximum value can be supplied which restricts the range of integers this option allows. Additionally, an output format of the integer option can be defined, which can be either decimal (base 10) or hexadecimal (base 16).

UseCaseScript.stringOption

Describes a string such as `traceCapture` or `etmv4` which can be used, for example, to refer to the CoreSight components in the current system by name.

UseCaseScript.enumOption, UseCaseScript.radioEnumOption

Both these describe enumerations. These can be used to describe a list of values that are allowed for this option. Enumeration values can be strings or integers.

Examples

```
UseCaseScript.booleanOption(name="timestamps", displayName="Enable global
timestamping", defaultValue=True)
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Defines a boolean option ‘timestamps’ which is true by default.

```
UseCaseScript.stringOption(name="traceBuffer", "Trace Capture Method", defaultValue="none")
```

Defines a string option ‘traceBuffer’ with a default value ‘none’.

```
UseCaseScript.integerOption(name="cores", displayName="Number of cores", defaultValue=1,
min=1, max=16, format=UseCaseScript.DEC)
```

Defines an integer option ‘cores’ with a default value of 1, minimum value of 1 and maximum value of 16. The output format is in decimal (base 10).

```
UseCaseScript.integerOption(name="retries", displayName="Attempts to retry", defaultValue=5,
min=1, format=UseCaseScript.HEX)
```

Defines an integer option ‘retries’ with a default value of 5, minimum value of 1 and no maximum value. The output will be in hexadecimal (base 16).

```
UseCaseScript.enumOption(name="contextid", displayName="Context ID Size", values = [("8", "8
bits"), ("16", "16 bits"), ("32", "32 bits")] , defaultValue="32")
```

Defines an enumeration ‘contextid’ with default value of ‘32’, the values are restricted to members of the enumeration: “8”, “16” or “32”.

Nesting Options

Options can be organized or grouped using the following:

- UseCaseScript.tabSet
- UseCaseScript.tabPage
- UseCaseScript.infoOption
- UseCaseScript.optionGroup

The groups are not associated with a value, and do not require a default. You can use the groups with the option names to construct a complete set of options.

To specify the nesting of options, each option can include an optional `childOptions` keyword which is a list of other options which are nested within this option. An example set of nested options can be seen below.

Examples

The following example shows an options method, complete with a set of options.

```
def myOptions():
      return [
          UseCaseScript.optionGroup(
          name="options",
          displayName="Usecase Options",
          childOptions=[
              UseCaseScript.optionGroup(
                  name="connection",
                  displayName="Connection Options",
                  childOptions=[
                      UseCaseScript.integerOption(
                         name="cores",
                         displayName="Number of cores",
                         defaultValue=1,
                         min=1,
                         max=16),
                      UseCaseScript.stringOption(
                         name="traceBuffer",
                         displayName="Trace Capture Method",
                         defaultValue="none"),
                  ]
              ),
              UseCaseScript.enumOption(
              name="contextID",
              displayName="Context ID Size",
              values = [("8", "8 bits"), ("16", "16 bits"), ("32", "32 bits")] ,
              defaultValue="32"),
              ]
          ),
      ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Options are accessed according to how they are nested within the list of options. In this example there is a group called `options` with two child options:

options.contextID

This is an enumeration within the options group, and stores the value of the contextID in this example.

options.connection

This is another group for nesting options. It does not store any value. It contains the options:

options.connection.cores

This is an integer that accesses `cores` variable.

options.connection.traceBuffer

This is a string that accesses `traceBuffer` variables in the list of options.

Using options in the script

The entry point and validation functions both take an `options` object, as the required first parameter, which can be used to get and set the option values. The functions that are provided to work with defined options are `getOptionValue(<name>)` and `setOptionValue(<name>, <value>)`.

For the `name` of the option, use the full name, for example `group.subgroup.variable`. The `value` must be of the correct type for the named variable, for example string, integer or a member of the enumeration.

To find out the full name of the option, either see the definition of options in the use case script or issue a `usecase help <script_name.py>` on the command-line.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.15 Defining the validation method for use case scripts

The method defined using the `$Validation$` tag names a method in the use case script which provides validation for the use case.

The validation method takes a single parameter, called `options` , which can be used to access the options supplied to the script.

The example shows a validation method called `myValidation` in the use case definition block. The validation method is used to validate the set of options supplied to the use case script.

Examples

```
...
$Validation$ myValidation
...
```

```
def myValidation(options):
   # Get the options which define the start and end of a trace range
   # These options have been defined in the function defined in the $Options$ tag
   start = options.getOptionValue("options.traceRange.start")
   end = options.getOptionValue("options.traceRange.end")
   # Conditional check for validation
   if(start >= end):
        # Report a specific error in the use case script if the validation check fails
        UseCaseScript.error("The trace range start must be before the end")
```

It is important that the function that supplies the validation takes a single parameter, which is the use case script object, used to access the options defined for use in the script.

If the `$Validation$` tag is defined, and the method referred to in this tag is not present in the script or the validation function takes the wrong number of arguments, an error occurs when running the script.

Error reporting

This validation example throws an error specific to use case scripts. If validation is not successful, for example start >= end, in our script, the Arm Development Studio command-line displays:

```
UseCaseError: The trace range start must be before theend
```

It is possible not to use the built-in use case error reporting and throw a standard Jython or Java error such as:

```
raise RuntimeError("Validation wasunsuccessful")
```

This displays an error in the Arm Development Studio command-line:

```
RuntimeError: Validation was unsuccessful
```

However, it is recommended to use the built-in use case script error reporting so that a clear user-defined error is raised that originates from the use case script.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.16 Example use case script definition

This is an example of a complete use case script.

After a use case script is defined, you can use the command-line options in Arm Development Studio to query and execute the script when connected to a target.

Examples

```
"""
USECASE
$Title$ Display Title
$Description$ A brief description of this use case
# Refers to a function called myOptions which returns a list of options
$Options$ myOptions
# Refers to a function called myValidation which validates the options in the script
$Validation$ myValidation
$Help$
usage: usecase.py [options]
A longer description of this use case
And additional usage, descriptions of parameters and extra information.
...
...
$Help$
# Function called mainMethod which defines the entry point to this usecase
$Run$ mainMethod
"""
```

```
def myOptions():
      return [
          UseCaseScript.optionGroup(
              name="options",
              displayName="Usecase Options",
              childOptions=[
                  UseCaseScript.optionGroup(
                     name="connection",
                     displayName="Connection Options",
                     childOptions=[
                         UseCaseScript.integerOption(
                             name="cores",
                             displayName="Number of cores",
                             defaultValue=1,
                             min=1,
                             max=16),
                         UseCaseScript.stringOption(
                             name="traceBuffer",
                             displayName="Trace Capture Method",
                             defaultValue="none"),
                     ]
                  ),
                  UseCaseScript.enumOption(
                  name="contextID",
                  displayName="Context ID Size",
                  values = [("8", "8 bits"), ("16", "16 bits"), ("32", "32 bits")],
                  defaultValue="32"),
              ]
          ),
      ]
def myValidation(options):
    print "Performing validation..."
    if(options.getOptionValue('options.connection.cores') > 8):
        UseCaseScript.error('Having more than 8 cores is not allowed for this usecase')
def mainMethod(options, address):
    print "Running the main method"
    print "The address supplied %s" % address
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.17 Multiple use cases in a single script

You can define multiple use cases within a single script. This is useful to allow use cases to share common code, and each use case can provide a single piece of functionality.

Each use case requires its own use case definition block which begins with a `USECASE` header. When defining use cases in the same script, they can share options and validation functions but the entry point to each use case must be unique.

Multiple use case blocks can be defined as a single multi-line comment at the top of the script:

Examples

```
USECASE
...
...
$Run$ mainMethod
USECASE
...
...
$Run$ entry2
```

```
...
def mainMethod(options):
   print "Running the first main method"
...
def entry2(script, param1):
   print "Running the second main method"
...
```

Multiple use case blocks can be defined as separate blocks dispersed throughout the script:

Examples

```
USECASE
...
...
$Run$ mainMethod
```

```
...
def mainMethod(options):
   print "Running the first main method"
...
```

```
USECASE
...
...
$Run$ entry2
```

```
...
def entry2(script, param1):
   print "Running the second main method"
...
```

There is no limit to how many use cases you can define in a single script.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.18 usecase list command

The `usecase list` command allows the user to search for use case scripts in various locations.

The output reports the location searched for use cases and, if any use case scripts are found, prints a table listing:

- The script name.
- Entry points to the script. Each entry point defines a single use case.
- The title of each use case.
- The description of each use case, if defined.

If a `usecase list` command is issued without any parameters on the command line, Arm Development Studio reports all the use case scripts it finds in the current directory.

A `usecase list path/to/directory/` command lists any use case scripts it finds in the directory specified. The `usecase list` command accepts relative paths to locations as well as tilde (~) as the user home directory on Unix based systems.

Several use cases are shipped with Arm Development Studio and are found in the default configuration database under `/Scripts/usecase/`. When creating a use case it can be added to `/Scripts/usecase/` in the default database, or a custom user database. These scripts are also listed by the `usecaselist -s` command.

Use case scripts might be platform specific, and use cases can be defined as only visible for the current target. A use case script created in the configuration database in `/Boards/<Manufacturer>/<Platform>/` is only visible when a connection is made to the `<Manufacturer>/<Platform>` configuration.

`usecase list -p` lists all use case scripts associated with the current platform. For example, if a connection was made to the configuration in `/Arm FVP (Installed with Arm DS)/VE_AEMv8x1/` and the `usecase list-p` was run, only use case scripts in this directory for the current configuration are listed.

Issuing `usecase list -a` lists all scripts in the current working directory, in `/Scripts/usecase/` in the configuration database, and those for the current platform.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.19 usecase help command

The `usecase help` command is available to print help on how to use the use case scripts and information about the options available.

The syntax for running `usecase help` is:

```
usecase help [<flag>] <script_name> [<entry_point>]`
```

Where:

`<flag>`

Is one of:

`-p`

To run a script for the current platform.

`-s`

To run a script in the `/Scripts/usecase/` directory in the configuration database.

`<entry_point>`

Is the name of the entry point or main method defined in the use case. If a use case script defines more than one entry point, then you must specify the `<entry_point>` parameter.

`<script_name>`

Is the name of the use case script.

Running use case help on a valid script prints:

- Text defined in a `$Help$` block in the use case script.
- A set of built-in options that are defined in every use case script.
- A list of information about the options defined in this use case.

For each option the help text lists:
- The display name of the option.
- The unique option name for getting or setting options.
- The type of option, which is integer, string, boolean, or enumeration.
- The default value of the option.

For enumeration options, a list of the enumerated values are listed. For integer options, the maximum and minimum values are displayed, if they have been specified.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.20 usecase run command

The `usecase run` command runs the script from the specified entry point.

The basic syntax for running a use case script from the command-line is:

```
usecase run <script_name> [<options>]
```

The options that you can use are defined in the method with the `$Options$` tag of the current use case within the script. You can get more information about them using the `usecase help` command.

The syntax for setting options on the command line is `--options.name=value` or `--options.namevalue`. If you do not specify a value for an option explicitly, the option takes the default value defined in the script.

Note

The examples use `options` as the name of the top level group of options. However, you can give a different name to the top level group of options in the use case script.

When issuing the `usecase run` command, rather than specifying an absolute path to a script, specify a `-p` or `-s` flag before the script name. For example, issue `usecase run -p <script_name> [<options>]` to run a use case script for the current platform.

If there is more than one use case defined in a single script, the entry point or main method to use when running the script must be defined. For scripts with multiple use cases the syntax is `usecase run <script_name> <entry_point> [<options>]`.

If the entry point to the use case accepts positional arguments, you must specify them on the command-line when the script is run. For example, if the main method in the use case script `positional.py` in the current working directory is defined as follows:

```
...
$Run$ main
...
def main(script, filename, value):
       print("Running the main method")
```

The syntax to run the script is:

```
usecase run positional.py [<options>] <filename> <value>
```

Examples

```
usecase run myscript.py --options.enableETM=True --options.enableETM.timestamping=True
       --options.traceCapture "DSTREAM"
```

Runs a use case script called `myscript.py` in the current working directory, setting the options defined for this use case.

```
usecase run multipleEntry.py mainOne --options.traceCapture "ETR"
```

Runs a use case script called `multipleEntry.py` in the current working directory. The entry point to this use case script is `mainOne`. A single option is specified after the entry point.

```
usecase run -s multipleScript.py mainTwo filename.txt 100
```

Runs a use case script in the `/Scripts/usecase/` directory in the configuration database called `multipleScript.py` in the current working directory. The entry point to this use case script is `mainTwo` which defines two positional arguments. No options are supplied to the script.

Saving options

On the command-line, providing a long list of options might be tedious to type in every time the script is run over different connections. A solution to this is to use the built-in functionality `--save-options`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For example, you can run the script `usecase run <script_name> --<option1=...>, --<option2=...> --save-options=</path/to/options.txt>` where `options.txt` is the file in which to save the options. This saves the options to this use case script, at runtime, to `options.txt`.

If you do not specify an option on the command-line, its default value is saved to the specified file.

Loading options

After saving options to a file, there is a similar mechanism for loading them back in. Issuing `usecase run <script_name> --load-options=<path/to/options.txt>` loads the options in from `options.txt` and, if successful, runs the script.

You can combine options by providing options on the command-line and loading them from a file. Options from the command-line override those from a file.

Example:

The options file `options.txt` for `myscript.py` contains two option values:

- `options.a=10`.
- `options.b=20`.

Running `usecase run myscript.py--load-options=options.txt` results in `options.a` having the value `10` and `options.b` having the value `20`, loaded from the specified file. If an option is set on the command-line, for example `usecase run--options.b=99 --load-options=options.txt`, it overrides those options retrieved from the file. `options.a` takes the value 10, but `options.b` takes the new value `99` provided on the command-line and not the one stored in the file. This is useful for storing a standard set of options for a single use case and modifying only those necessary at runtime.

Showing options

When running a script, the user might want to see what options are being used, especially if the options are loaded from an external file. The built-in option `--show-options` displays the name and value of all options being used in the script when the script is run.

Examples

`usecase run <script_name> --show-options`

Prints out a list of the default options for this script.

`usecase run <script_name> --option1=x, --option2=y --show-options`

Prints out a list of options for the script, with updated values for option1 and option2.

`usecase run <script_name> --load-options=<file> --show-options`

Prints out a list of options taking their values from the supplied file. If an option is not defined in the loaded file, its default value is printed.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 6
Perspectives and Views

Describes the perspective and related views in the Integrated Development Environment (IDE).

It contains the following sections:

- *6.1 App Console view* on page 6-135.
- *6.2 Arm Asm Info view* on page 6-137.
- *6.3 Arm assembler editor* on page 6-138.
- *6.4 Breakpoints view* on page 6-140.
- *6.5 C/C++ editor* on page 6-144.
- *6.6 Commands view* on page 6-148.
- *6.7 Debug Control view* on page 6-151.
- *6.8 Stack view* on page 6-155.
- *6.9 Disassembly view* on page 6-158.
- *6.10 Events view* on page 6-162.
- *6.11 Event Viewer Settings dialog box* on page 6-165.
- *6.12 Expressions view* on page 6-169.
- *6.13 Expression Inspector* on page 6-173.
- *6.14 Functions view* on page 6-174.
- *6.15 History view* on page 6-177.
- *6.16 Memory view* on page 6-179.
- *6.17 MMU/MPU view* on page 6-188.
- *6.18 Modules view* on page 6-193.
- *6.19 Registers view* on page 6-195.
- *6.20 NVIC Registers view* on page 6-202.
- *6.21 OS Data view* on page 6-205.
- *6.22 Overlays view* on page 6-207.
- *6.23 Cache Data view* on page 6-208.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- *6.24 Screen view* on page 6-210.
- *6.25 Scripts view* on page 6-213.
- *6.26 Target Console view* on page 6-217.
- *6.27 Target view* on page 6-218.
- *6.28 Trace view* on page 6-220.
- *6.29 Trace Control view* on page 6-225.
- *6.30 Variables view* on page 6-228.
- *6.31 Timed Auto-Refresh Properties dialog box* on page 6-234.
- *6.32 Memory Exporter dialog box* on page 6-235.
- *6.33 Memory Importer dialog box* on page 6-236.
- *6.34 Fill Memory dialog box* on page 6-237.
- *6.35 Export Trace Report dialog box* on page 6-238.
- *6.36 Trace Dump dialog box* on page 6-240.
- *6.37 Breakpoint Properties dialog box* on page 6-242.
- *6.38 Watchpoint Properties dialog box* on page 6-245.
- *6.39 Tracepoint Properties dialog box* on page 6-247.
- *6.40 Manage Signals dialog box* on page 6-248.
- *6.41 Functions Filter dialog box* on page 6-249.
- *6.42 Script Parameters dialog box* on page 6-250.
- *6.43 Debug Configurations - Connection tab* on page 6-251.
- *6.44 Debug Configurations - Files tab* on page 6-254.
- *6.45 Debug Configurations - Debugger tab* on page 6-258.
- *6.46 Debug Configurations - OS Awareness tab* on page 6-261.
- *6.47 Debug Configurations - Arguments tab* on page 6-262.
- *6.48 Debug Configurations - Environment tab* on page 6-264.
- *6.49 Debug Configurations - Export tab* on page 6-266.
- *6.50 DTSL Configuration Editor dialog box* on page 6-267.
- *6.51 Probe Configuration dialog box* on page 6-269.
- *6.52 About the Remote System Explorer* on page 6-270.
- *6.53 Remote Systems view* on page 6-271.
- *6.54 Remote System Details view* on page 6-272.
- *6.55 Target management terminal for serial and SSH connections* on page 6-273.
- *6.56 Remote Scratchpad view* on page 6-274.
- *6.57 Remote Systems terminal for SSH connections* on page 6-275.
- *6.58 Terminal Settings dialog box* on page 6-276.
- *6.59 Debug Hardware Configure IP view* on page 6-280.
- *6.60 Debug Hardware Firmware Installer view* on page 6-282.
- *6.61 Connection Browser dialog box* on page 6-285.
- *6.62 Preferences dialog box* on page 6-286.
- *6.63 Properties dialog box* on page 6-288.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.1 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, you must enable semihosting support in the debugger.

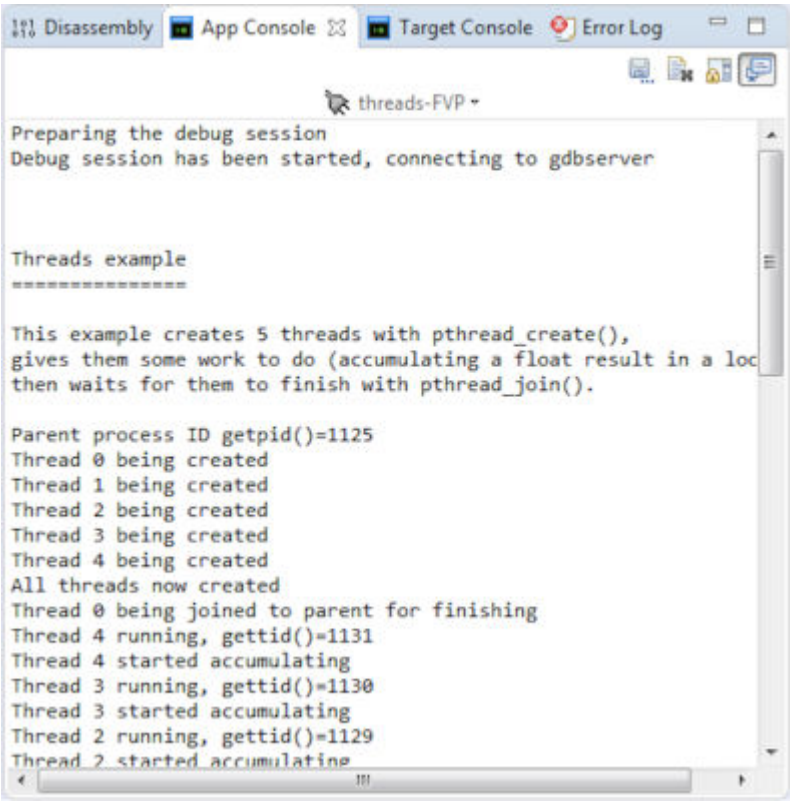


Figure 6-1 App Console view

Note

Default settings for this view, for example the maximum number of lines to display, are controlled by the Arm 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Toggles Scroll Lock

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

Bring to front when semihosting output is detected

Enabled by default. The debugger automatically changes the focus to this view when semihosting output is detected.

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

2.16 Using semihosting to access resources on the host computer on page 2-66
2.17 Working with semihosting on page 2-68
Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.2 Arm Asm Info view

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

When editing assembly language source files using the Arm **Assembly** or **Disassembly** editor, hover your mouse over an instruction to access more information.

The related documentation appears in a pop-up window.

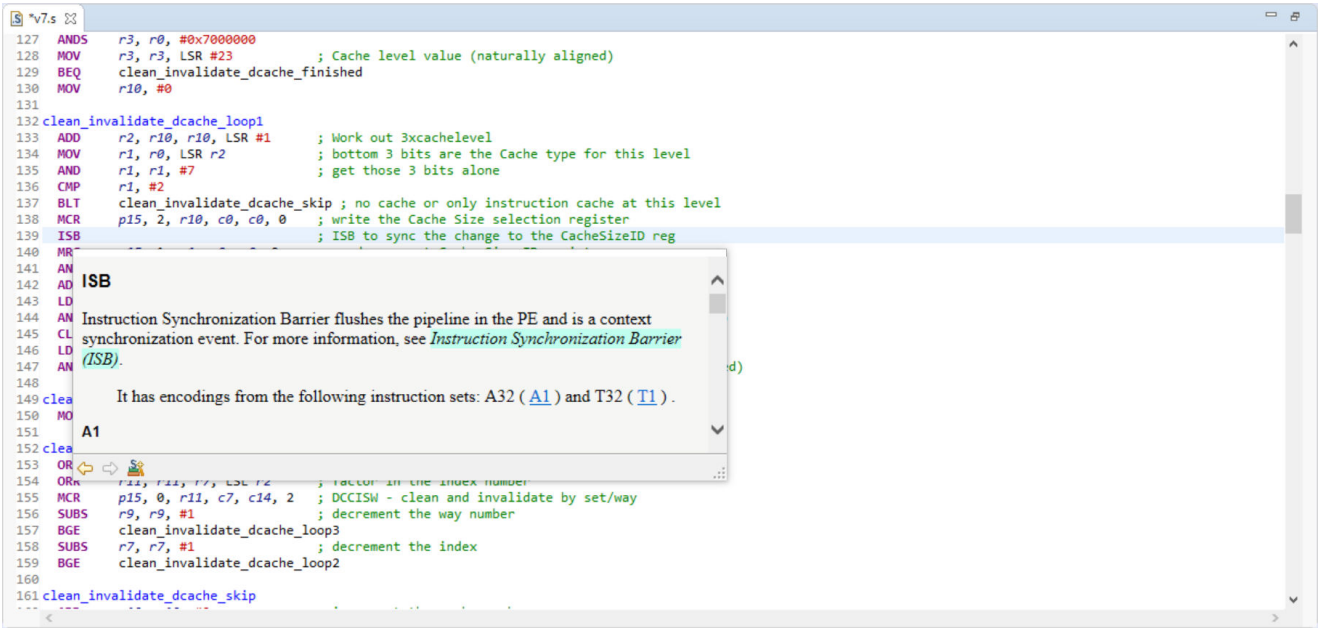


Figure 6-2 Arm Asm Info pop-up

To open this information in the **Arm Asm Info** view, click .

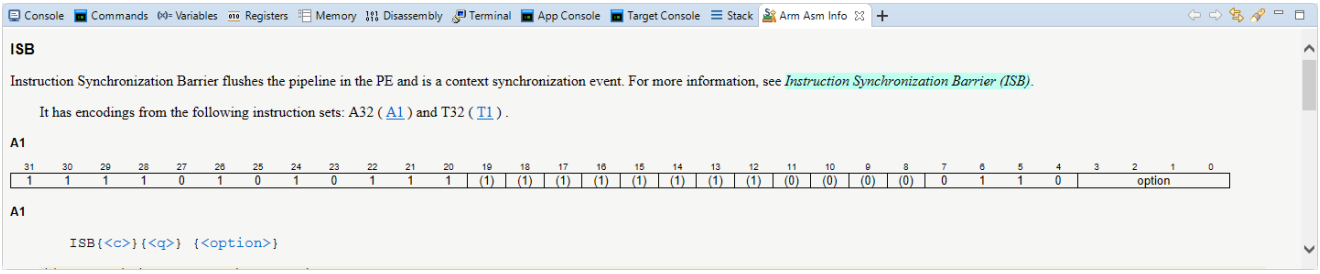


Figure 6-3 Arm Asm Info view

To manually show this view:

1. Ensure that you are in the **Development Studio** perspective.
2. Select **Window** > **Show View** > **Other...** to open the **Show View** dialog box.
3. Select the **Arm Asm Info** view from the **Arm Debugger** group.

Related references

Chapter 6 Perspectives and Views on page 6-133

6.3 Arm assembler editor on page 6-138

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.3 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.

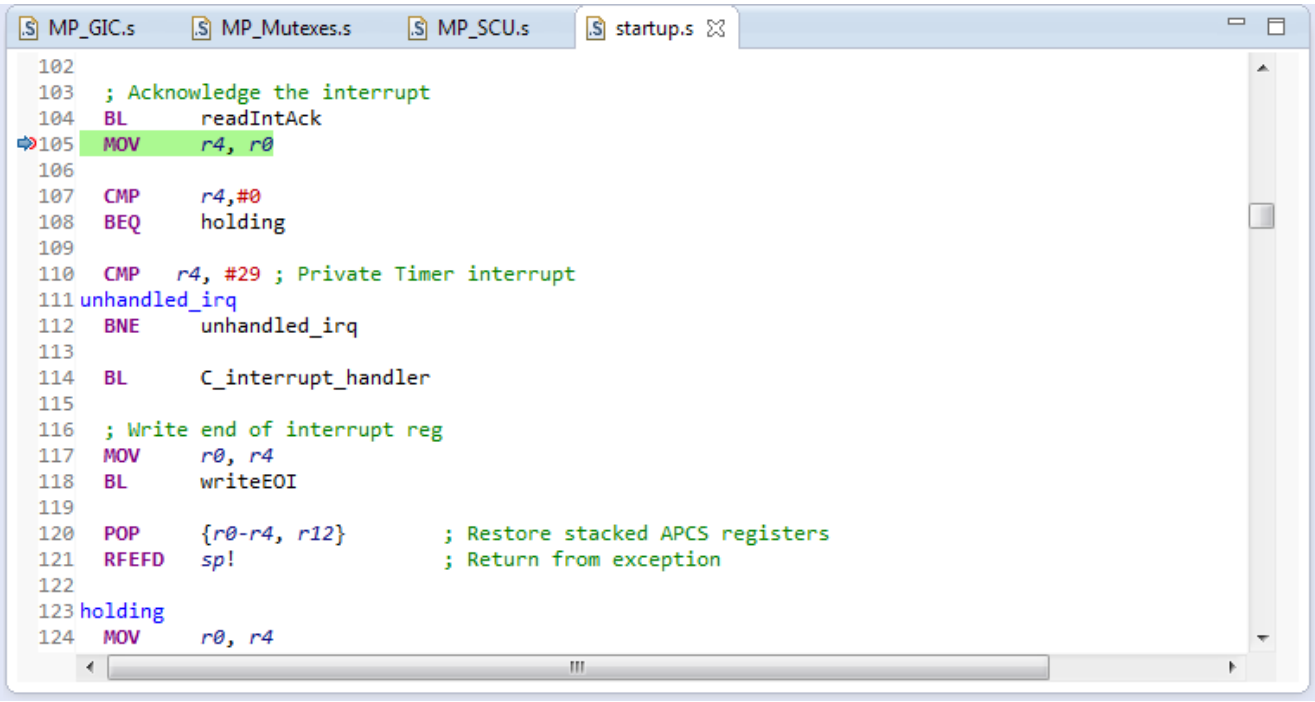


Figure 6-4 Assembler editor

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:

Arm DS 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

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:

Arm DS C/C++ Breakpoint

Select to use the Development Studio perspective breakpoint scheme. This is the default for the Development Studio 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 Development Studio Breakpoints. This menu is also available from the **Run** menu in the main menu bar at the top of the C/C++, Debug, and Development Studio perspectives.

The menu options under **Arm DS Breakpoints** do not retain this setting and always refer to Development Studio Breakpoints.

Show Line Numbers

Show or hide line numbers.

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

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
Chapter 6 Perspectives and Views on page 6-133
3.1 Examining the target execution environment on page 3-74
3.2 Examining the call stack on page 3-75
6.2 Arm Asm Info view on page 6-137

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.4 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.

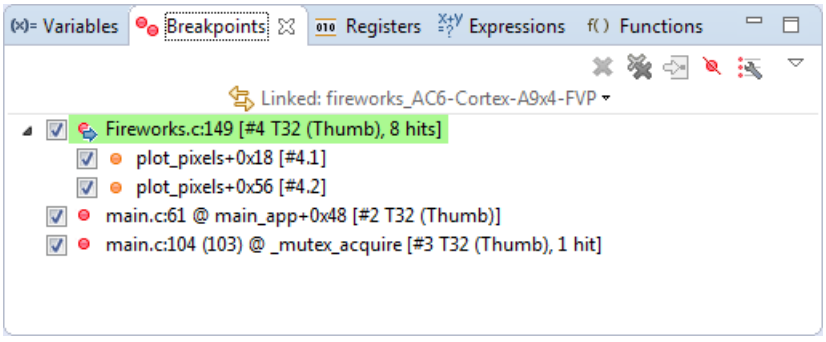


Figure 6-5 Breakpoints view showing breakpoints and sub-breakpoints

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>]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

<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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 currently set breakpoints or watchpoints. 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

1.8 About debugging multi-threaded applications on page 1-25
1.9 About debugging shared libraries on page 1-26
1.10 About debugging TrustZone enabled targets on page 1-28

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
Chapter 6 Perspectives and Views on page 6-133
3.1 Examining the target execution environment on page 3-74
3.2 Examining the call stack on page 3-75

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.5 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.

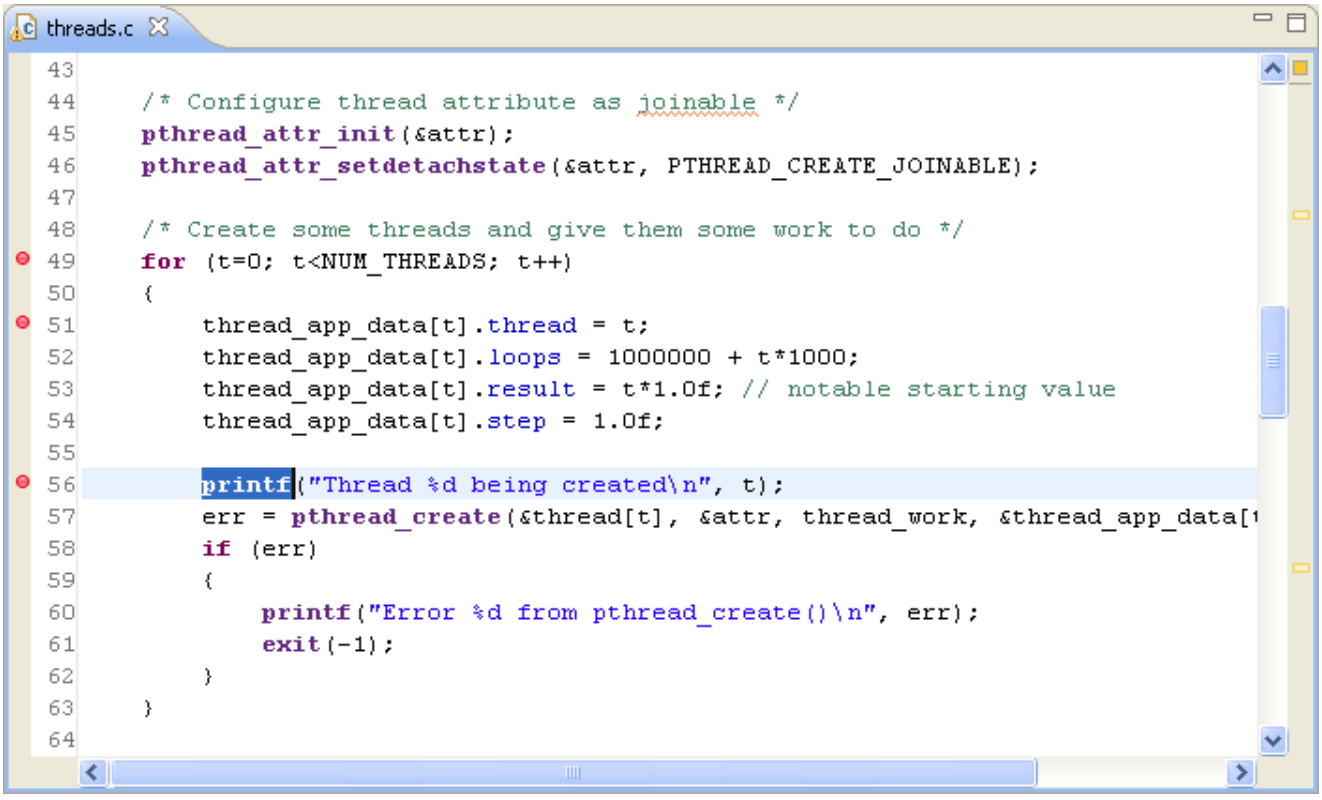


Figure 6-6 C/C++ editor

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:

Arm DS Breakpoints menu

The following breakpoint options are available:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

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 Development Studio Breakpoints. When using Arm Debugger you must select **Arm DS C/C++Breakpoint**. Development Studio 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

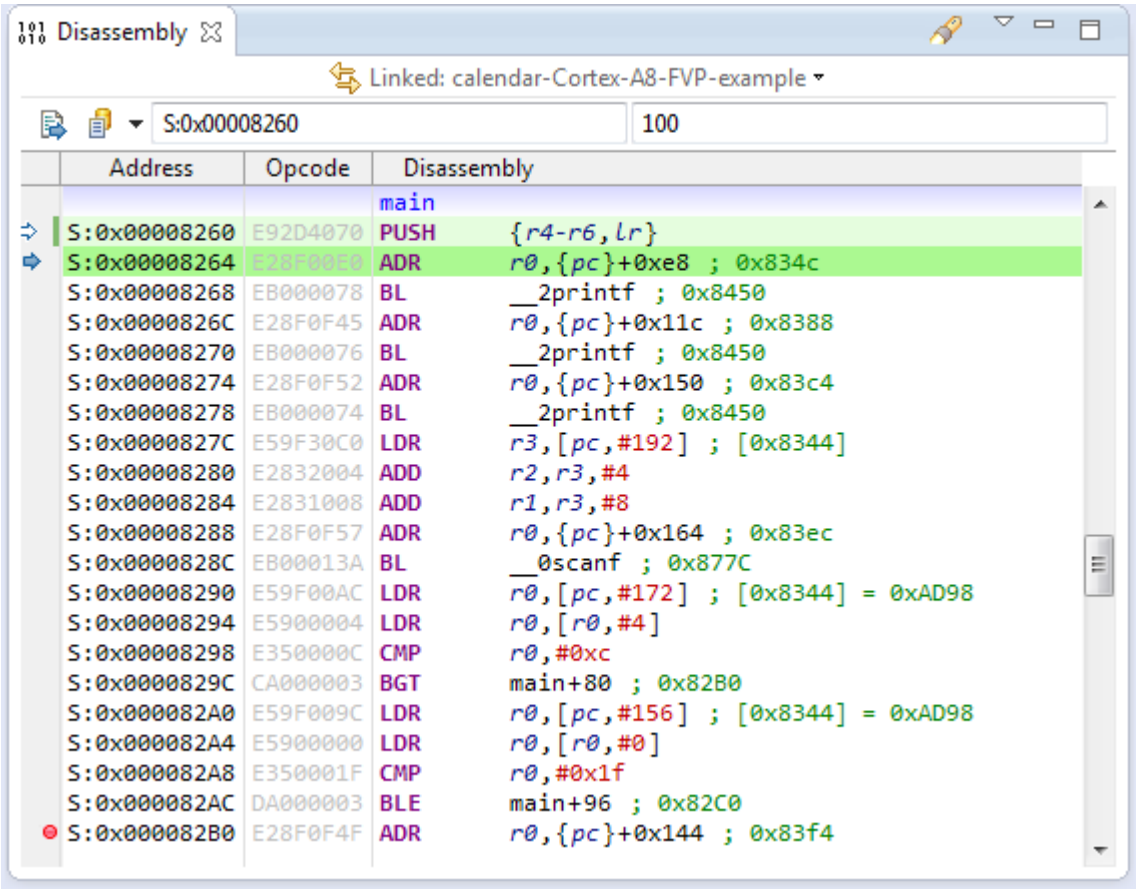


Figure 6-7 Show disassembly for selected source line

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Inspect

Selecting this option opens the **Expression Inspector** window, and allows you to monitor the values of the highlighted variables or expressions, and manually enter variables or expressions to monitor.

Note

You must highlight the variable you want to inspect before invoking the menu, otherwise the Inspect option is not available.

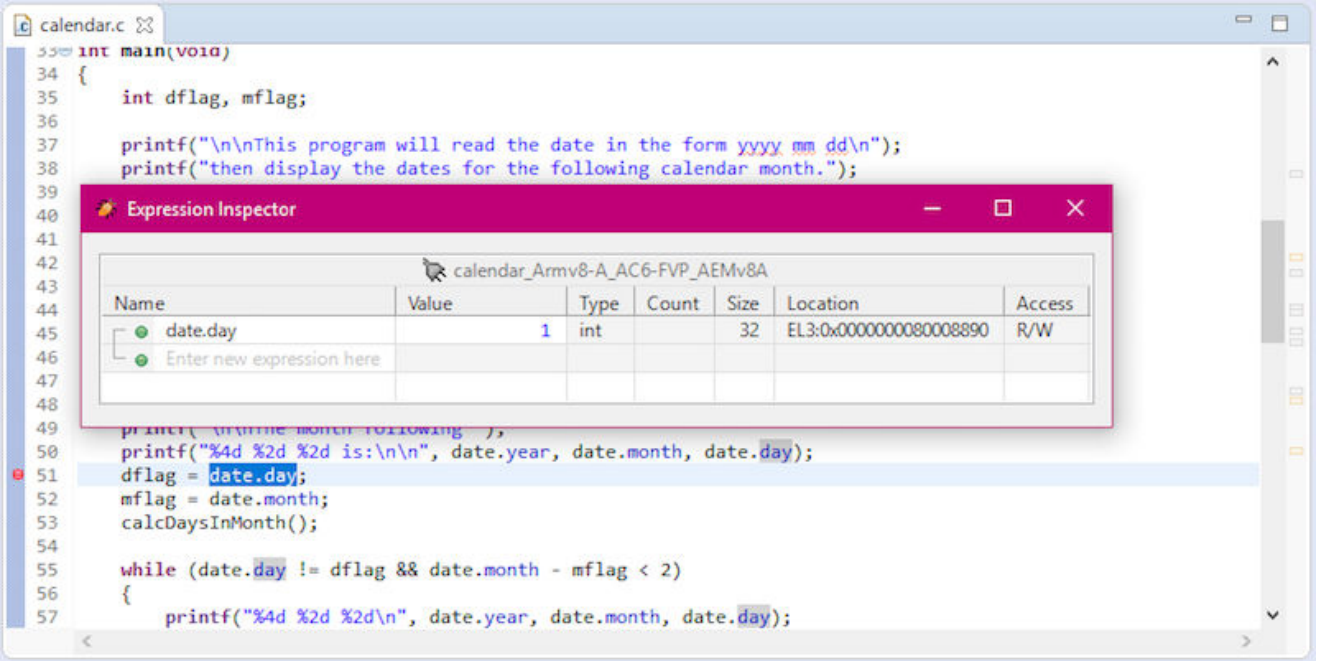


Figure 6-8 Inspect the value of the highlighted variable

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

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
6.12 Expressions view on page 6-169
Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.6 Commands view

Use the **Commands** view to display Arm 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.

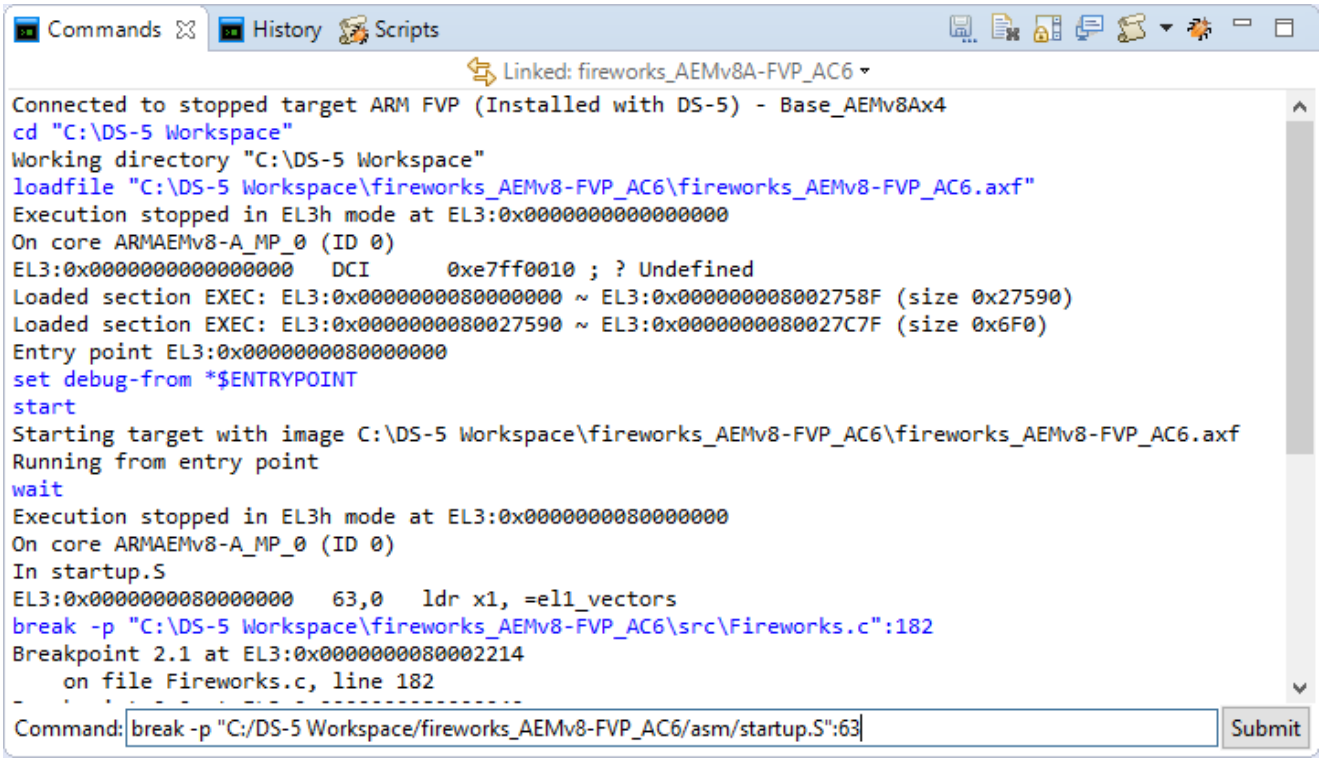


Figure 6-9 Commands view

You can execute Arm Debugger commands by entering the command in the field provided, then click **Submit**.

Note

You must connect to a target to use this feature.

You can also use content assist keyboard combinations, provided by Eclipse, to display a list of Arm 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 Development Studio perspective keyboard shortcuts in the Related reference section for details about specific content assist keyboard combinations available in Arm Debugger.

Note

To access the default settings for this view, select **Window** then **Preferences**. From here, you can specify settings such as the default location for script files, or the maximum number of lines to display.

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Linked: context

Links this view to the selected connection in the **Debug Control** view. This is the default setting. 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.

Bring to front when a command is executed

Disabled by default. When enabled, the debugger automatically changes the focus to this view when a command is executed.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

New Commands View

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

Related concepts

1.8 About debugging multi-threaded applications on page 1-25
1.9 About debugging shared libraries on page 1-26
1.10 About debugging TrustZone enabled targets on page 1-28

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
Chapter 6 Perspectives and Views on page 6-133

Related information

Arm Debugger commands listed in alphabetical order
Development Studio perspective keyboard shortcuts

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.7 Debug Control view

Use the **Debug Control** view to display target connections with a hierarchical layout of running cores, threads, or user-space processes.

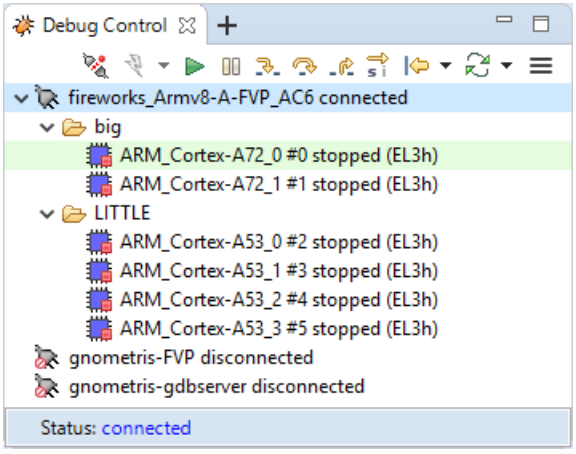


Figure 6-10 Debug Control view

This view enables you to:

- Connect to and disconnect from a target.
- View a list of running cores, threads, or user-space processes as applicable.
- Load an application image onto the target.
- Load debug information when required by the debugger.
- Look up stack information.
- 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.
- Reset the target.

Some of the views in the Development Studio perspective are associated with currently selected execution context. Each associated view is synchronized depending on your selection.

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.

Connection execution states are identified with different icons and background highlighting and are also displayed in the status bar.

When working with threads, note that the current active thread is always highlighted, as shown in the following figure:

Toolbar and context menu options

The following options are available from the toolbar or context menu:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Collapse All

Collapses all expanded items.

Display Cores/Display Threads

Click to toggle between viewing cores or threads. This option is only active for bare-metal connections with OS awareness enabled.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

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.

Show in Stack

Opens the **Stack** view, and displays the stack information for the selected execution context.

DTSL options

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

Reset Development Studio Views to 'Linked'

Resets Arm Development Studio 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 **DebugControl** 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Threads Presentation

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

Related concepts

1.8 About debugging multi-threaded applications on page 1-25
1.9 About debugging shared libraries on page 1-26

Related references

6.8 Stack view on page 6-155

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.8 Stack view

Use the **Stack** view to display stack information for the currently active connection in the **Debug Control** view. You can view stack information for cores, threads, or processes depending on the selected execution context.

To view stack information:

1. In the **Debug Control** view, right-click the core, thread, or process that you want stack information for, and select **Show in Stack**. This displays the stack information for the selected execution context.

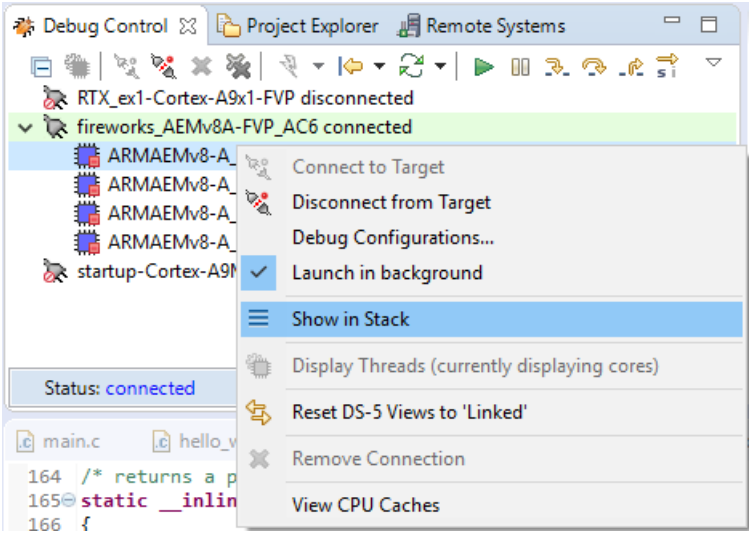


Figure 6-11 Show in Stack

2. Stack information is gathered when the system is stopped.

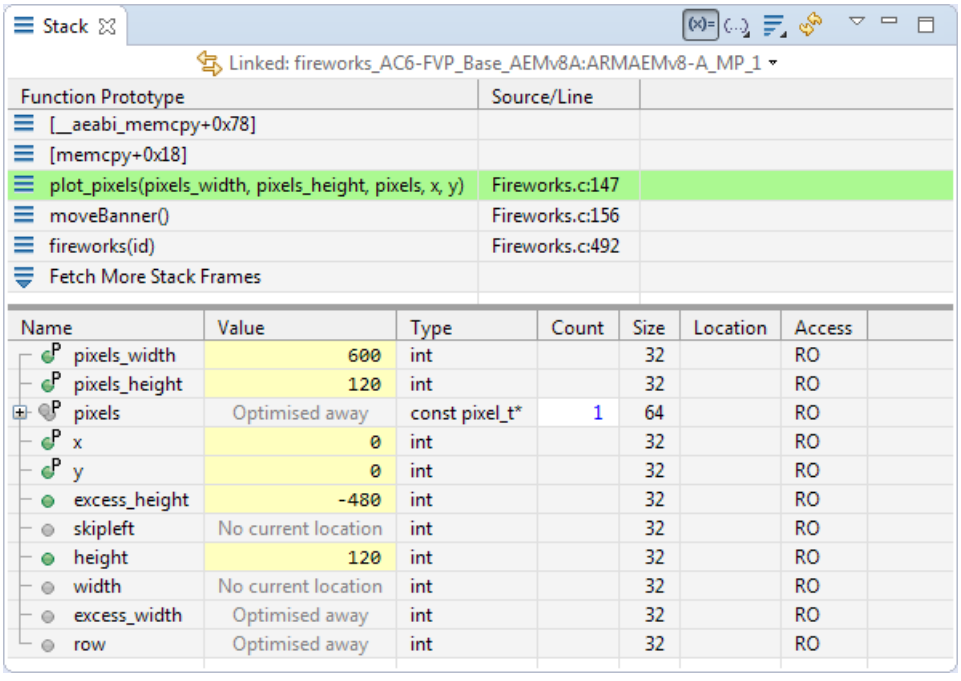


Figure 6-12 Stack view showing information for a selected core

Some of the views in the **Development Studio** perspective are associated with the currently selected stack frame. Each associated view is synchronized accordingly.

You can also:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Lock Stack view information display to a specific execution context

You can restrict **Stack** view information display to a specific execution context in your current active connection. In the **Stack** view, click Linked: <context> and select the execution context to lock. For example, in the below figure, **Stack** view is locked to the selected thread.

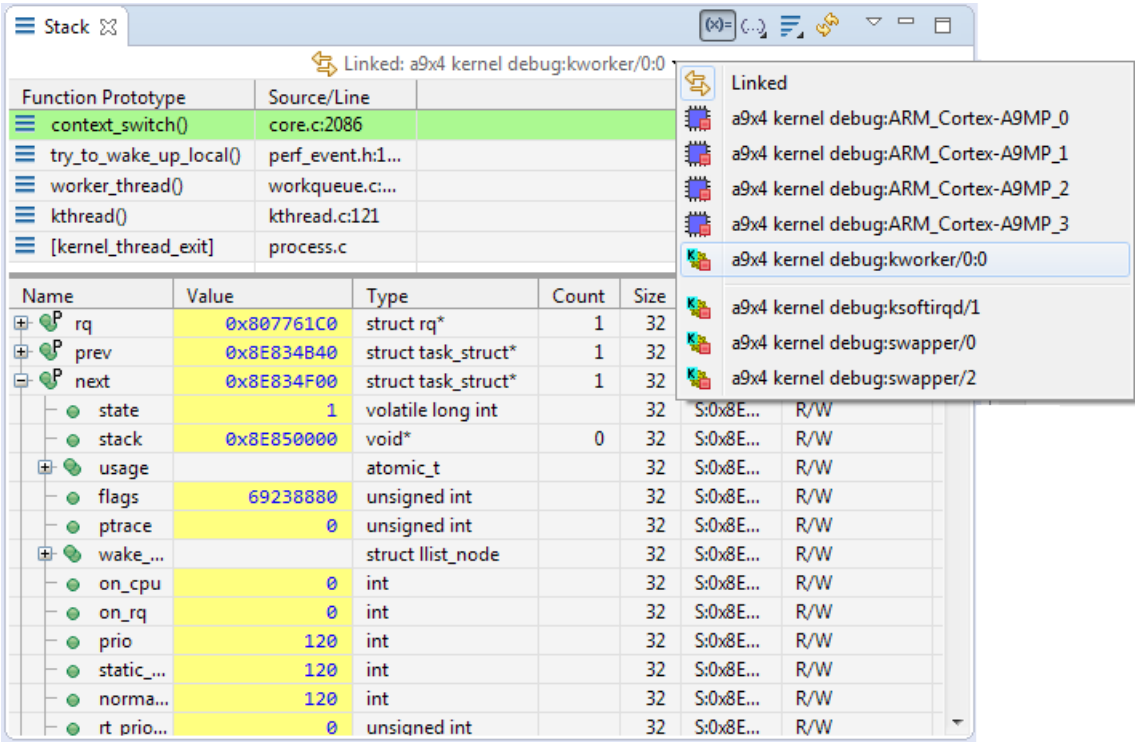


Figure 6-13 Stack view locked to a selected context

Show or hide the Local Variables panel

Click (x)= to show or hide the **LocalVariables** panel. You can interact with local variables as you would in the **Variables** view. See *Variables view* on page 6-228 for more information about working with variables.

Set function prototype display options

Click (...) to set the function prototype display options. You can choose to show or hide function parameter types or values.

Note

Displaying a large number of function parameter values might slow the debugger performance.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

View more stack frames

To see more stack frames, click **Fetch More Stack Frames** to view the next set of stack frames.

By default, the **Stack** view displays five stack frames, and each additional fetch displays the next five available frames.

To increase the default depth of the stack frames to view, on the **Stack** view menu, click and select the required stack depth. If you need more depth than the listed options, click **Other** and enter the depth you require.

Note

Increasing the number of displayed stack frames might slow the debugger performance.

Refresh the view

To refresh or update the values in the view, click .

Show in Disassembly

Right-click a stack frame and select **Show in Disassembly** to open the **Disassembly** view and locate the current instruction for that stack frame.

Show in Memory

Right-click a stack frame and select **Show in Memory** to open the **Memory** view and display the memory location used to store that stack frame.

Step Out to This Frame

Right-click a stack frame and select **Step Out to This Frame** to run to the current instruction at the selected stack frame.

Toolbar options

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

New Stack View

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

Freeze Data

Toggles the freezing of data in the currently selected execution context. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

Update View When Hidden

Updates the view when it is hidden behind other views. By default, this view does not update when hidden.

Related references

6.7 Debug Control view on page 6-151

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.9 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.

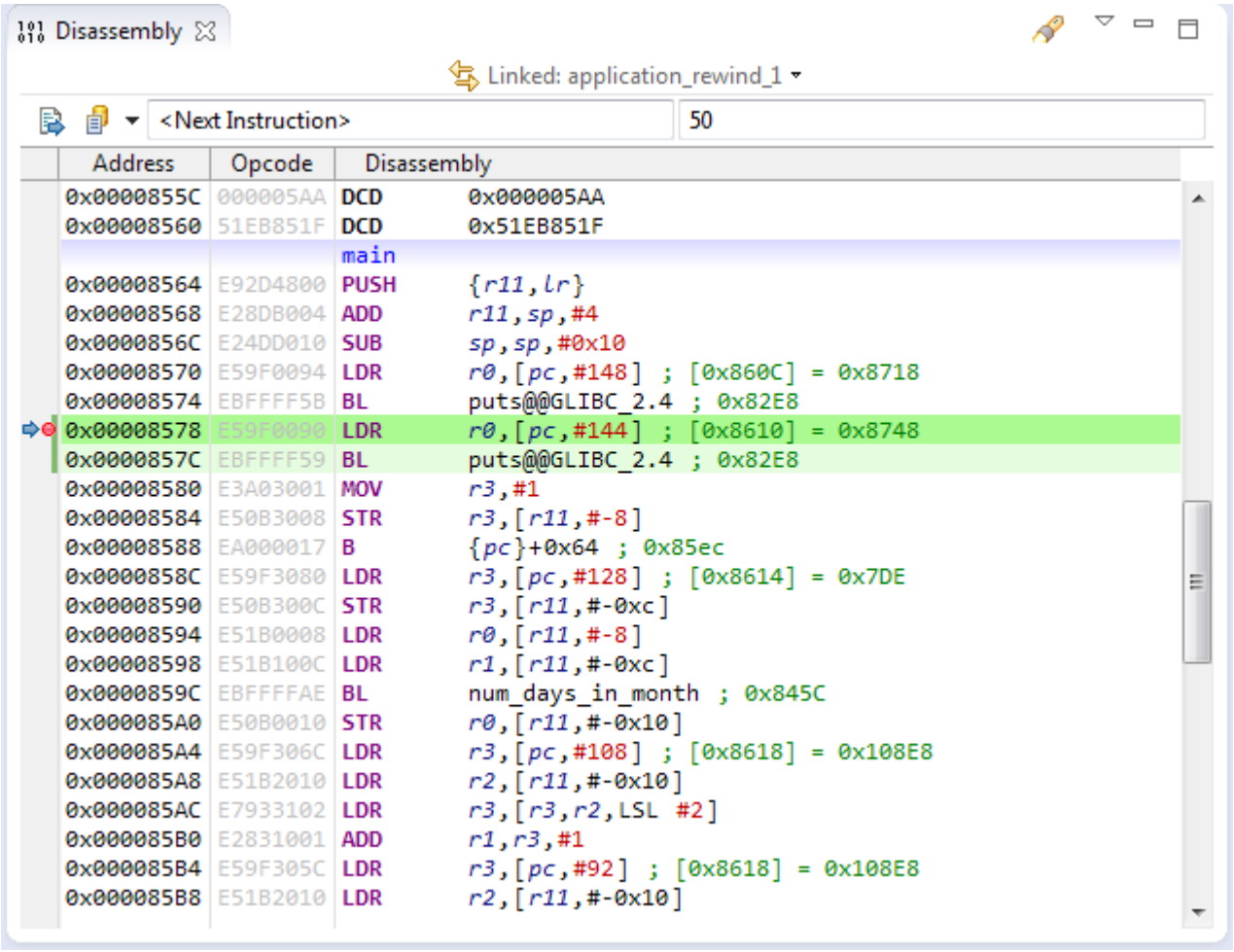


Figure 6-14 Disassembly view

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 `$PC+256`, `$lr`, 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 option also disables and enables the **Size** and **Type** fields. You can still use the **Refresh** option to manually refresh the view.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

1.8 About debugging multi-threaded applications on page 1-25
1.9 About debugging shared libraries on page 1-26
1.10 About debugging TrustZone enabled targets on page 1-28

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.10 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.

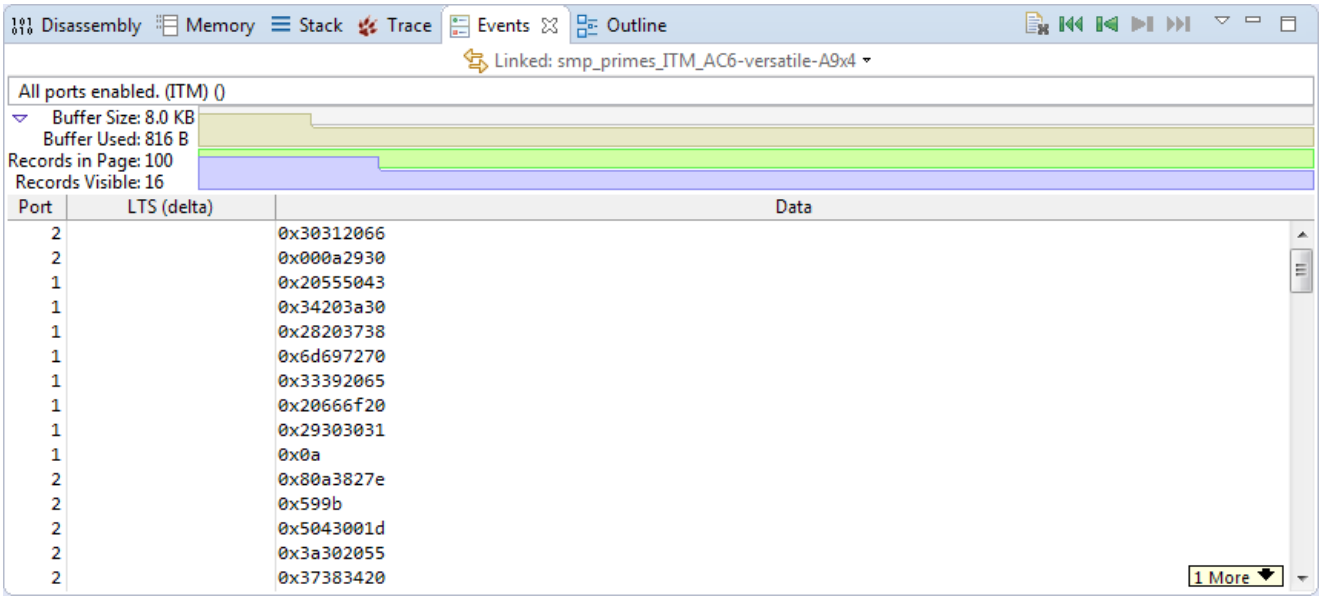


Figure 6-15 Events view (Shown with all ports enabled for an ETB:ITM trace source)

Note

- Use the *Event Viewer Settings dialog box* on page 6-165 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.
- To view the **Live Decode** tab and streaming trace data, you must enable the **Show live decode console** option in the **Event Viewer Settings** dialog box.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 Global Timestamp…

Displays the **Search by Timestamp** dialog box 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. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

Events Settings…

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

Open Trace Control View

Opens the **Trace Control** view.

DTSL Options…

Opens the *DTSL Configuration Editor* on page 6-267 dialog box.

Events context menu

Advance / Go back local timestamp amount

Displays the **Advance / Go back local timestamp** dialog box which allows you to move forward or backward from the current record by a local timestamp amount. A negative number indicates a movement backward. The search moves forward or backward from the current record until the sum of intervening local timestamps is equal to or exceeds the specified value.

Global 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 6 Perspectives and Views on page 6-133
6.11 Event Viewer Settings dialog box on page 6-165

Related information

CoreSight Components Technical Reference Manual
CoreSight System Trace Macrocell Technical Reference Manual
Armv7-M Architecture Reference Manual Documentation

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.11 Event Viewer Settings dialog box

Use the **Event Viewer Settings** dialog box 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

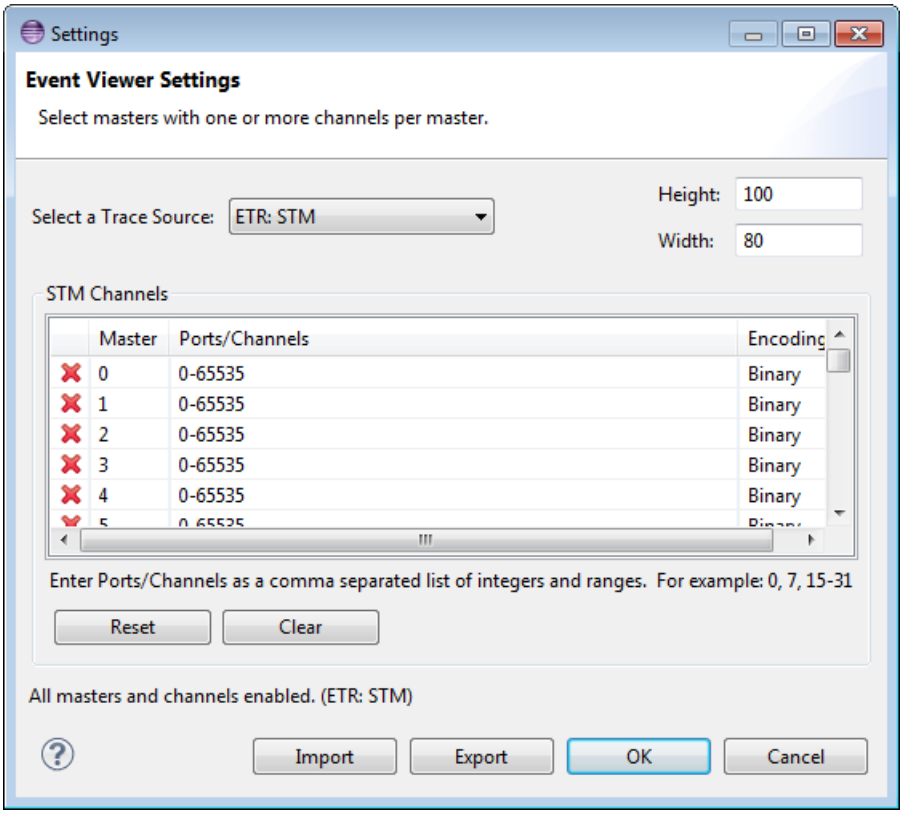


Figure 6-16 Event Viewer Settings (Shown with all Masters and Channels enabled for an ETR:STM trace source)

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**.

Show local timestamps

Select to display local timestamps. Local timestamps are expressed as an interval since the last local timestamp. Local timestamps are available for CoreSight ITM trace and M-profile ITM trace.

Show global timestamps (M-profile only)

Select to display global timestamps. Global timestamps are expressed as a 48-bit or 64-bit counter value.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Show event counter wrapping (M-profile only)

Select to display event counter wrapping packets from the M-profile Data Watchpoint and Trace (DWT) unit.

Show exception tracing (M-profile only)

Select to display exception trace packets from the M-profile DWT unit.

Show PC sampling (M-profile only)

Select to display PC sampling packets from the M-profile DWT unit.

Show data tracing (M-profile only)

Select to display data trace packets from the M-profile DWT unit.

Show live decode console

Select to display the **Live Decode** tab in the **Events** view. The **Live Decode** tab displays streaming trace data from your target.

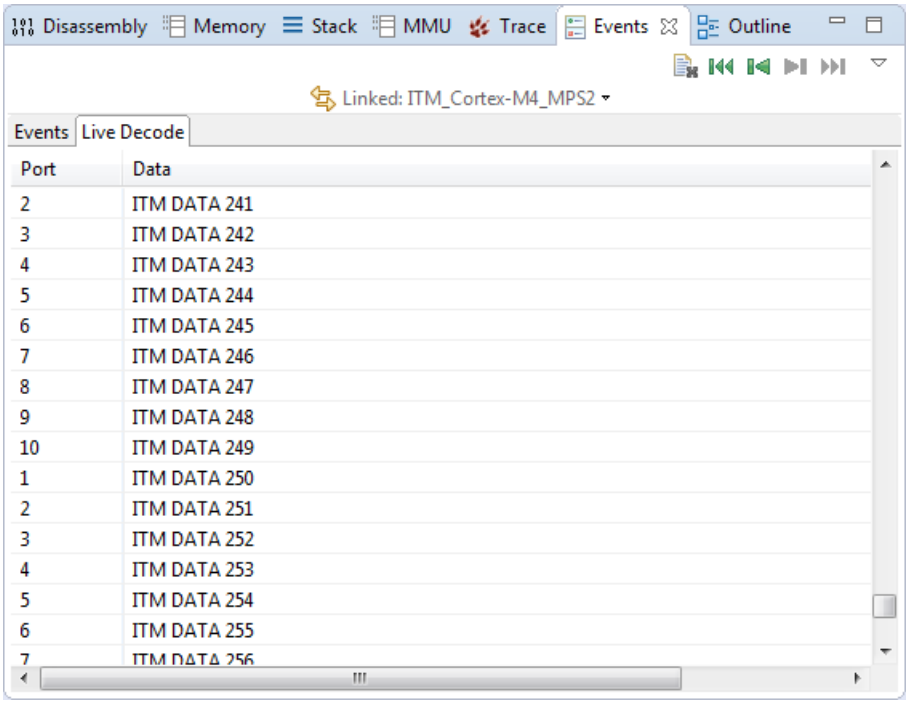


Figure 6-17 Events view - Live Decode tab

Note

- To use the **Live Decode** tab, your target must support ITM or STM trace and must be connected to a DSTREAM-ST unit.
- Live streaming data is read-only.
- If generating a large amount of ITM or STM trace, Arm recommends:
  - Turning off **Core Trace** in the **DSTL Options** dialog box. This is to avoid data overflow issues when a large amount of data is generated.
  - Disabling some channels or ports to reduce data overload issues in the **Live Decode** tab.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

6.10 Events view on page 6-162

Related information

CoreSight Components Technical Reference Manual

CoreSight System Trace Macrocell Technical Reference Manual

Armv7-M Architecture Reference Manual Documentation

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.12 Expressions view

Use the **Expressions** view to create and work with expressions.

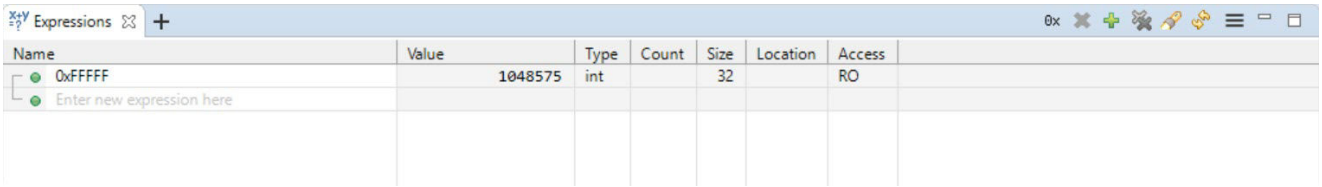


Figure 6-18 Expressions view

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.

Toolbar options

The following options are available from the toolbar menu:

Show all numerical values in hexadecimal

Click the button to change all numeric values to hexadecimal values. This works as a toggle and your preference is saved across sessions.

Delete

In the **Expressions** view, select the expression you want to remove from view, and click to remove the selected expression.

Add New Expression

Click the button to add a new expression.

Remove All Expressions

Click the button to remove all expressions.

Search

Click the button to search through all expressions.

Refresh Expressions View

Click the button to refresh or update the values in the view.

View Menu

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

Link with

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.

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 option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

Context menu options

The following options are available from the context menu:

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.

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. These formats are **Binary**, **Boolean**, **Hexadecimal**, **Octal**, **Signed Decimal**, and **Unsigned decimal**.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 box 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.

Column headers

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Show All Columns

Displays all columns.

Reset Columns

Resets the columns displayed and their widths to the default.

All columns are displayed by default.

Examples

When debugging the Linux kernel, to view its internal thread structure,use these expressions:

For Armv8 in SVC mode, with 8K stack size:

```
(struct thread_info*)($SP_SVC &amp;~0x1FFF)
```

For Armv8 AArch64 in EL1, with 16K stack size:

```
(struct thread_info*)($SP_EL1 &amp;~0x3FFF)
```

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

6.13 Expression Inspector on page 6-173
2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
6.5 C/C++ editor on page 6-144
Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.13 Expression Inspector

The **Expression Inspector** is a light-weight version of the **Expressions** View. It allows you to monitor the values of the highlighted variables or expressions, and manually enter variables or expressions to monitor.

Instead of presenting the information in a view, the **Expression Inspector** is a floating window. You can have multiple **Expression Inspector** windows open at one time.

To invoke the **Expression Inspector**, you need to highlight a variable or expression of interest in the C/C ++ editor view, then right-click anywhere in the view and select the **Inspect** option. If you have not highlighted anything, the **Inspect** option is unavailable.

You can also manually enter variable names or expressions in the empty field at the bottom of the list.

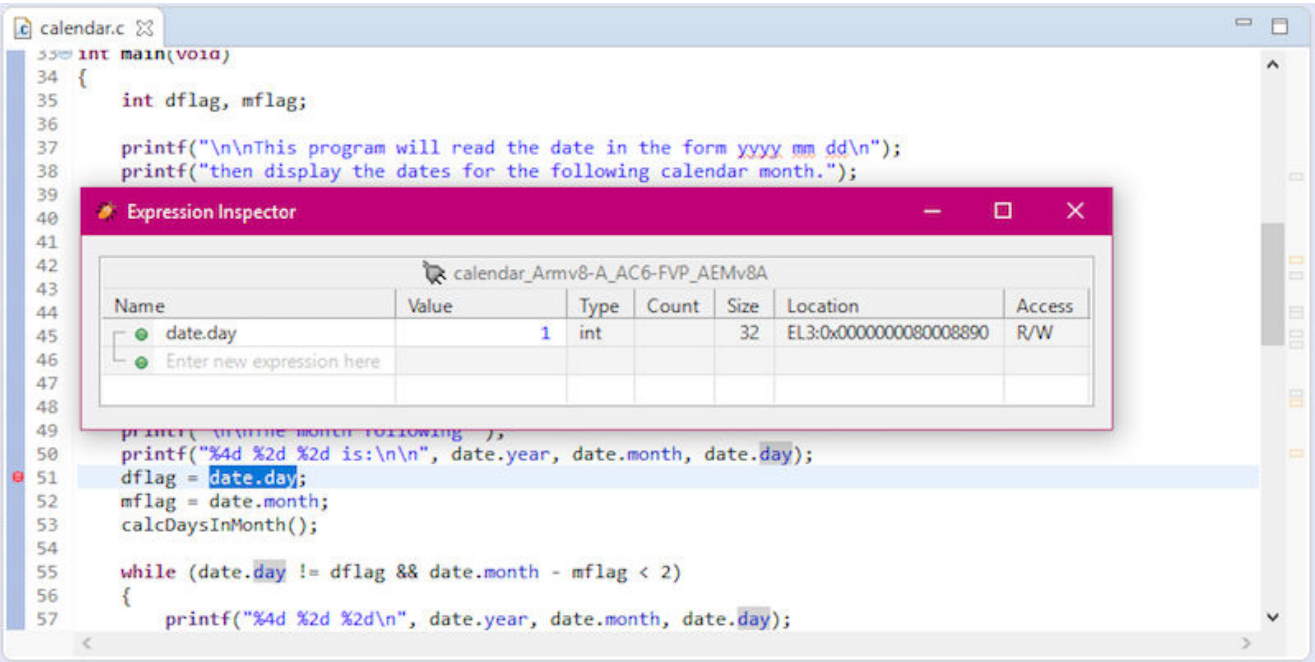
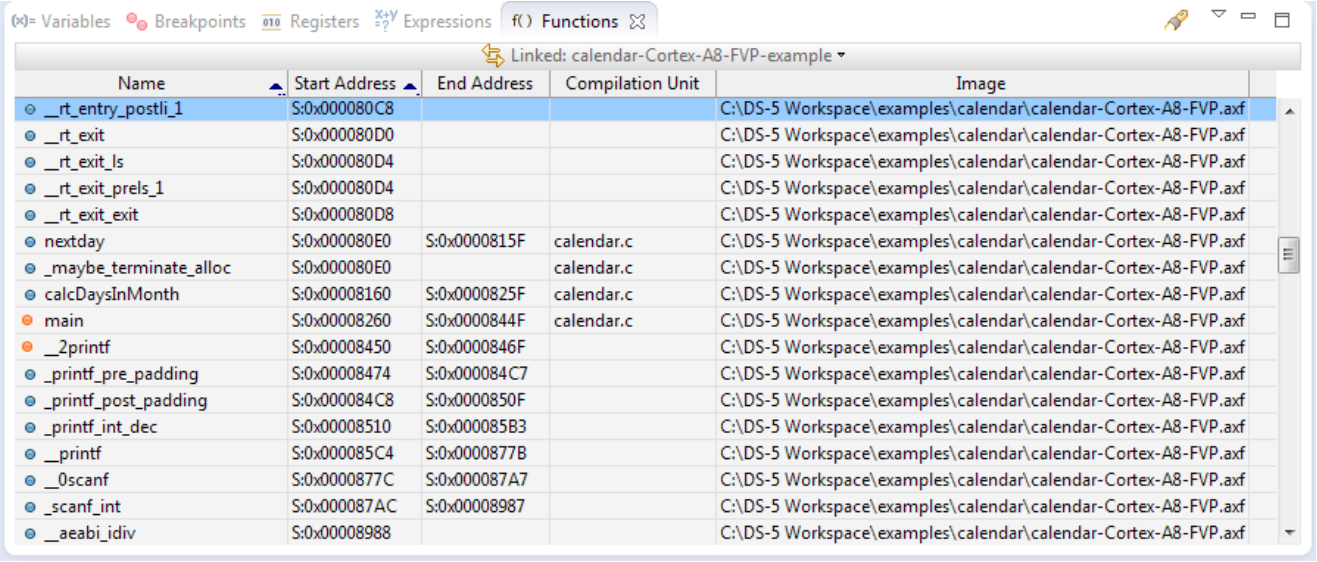


Figure 6-19 Inspect the value of the highlighted variable

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.14 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 6-20 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reset Columns

Resets the columns displayed and their widths to the default. The `Name`, `StartAddress`, `End Address`, `Compilation Unit`, and `Image` columns are displayed by default.

In the **Functions** view, the functions are represented as:

Table 6-1 Function icons

Icon	Description	Icon	Description
	Function		Function with a breakpoint set
	Static function		Static function with a breakpoint 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.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

Filters…

Displays the Functions Filter dialog box. This enables you to filter the functions displayed in the view.

Related references

Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.15 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.

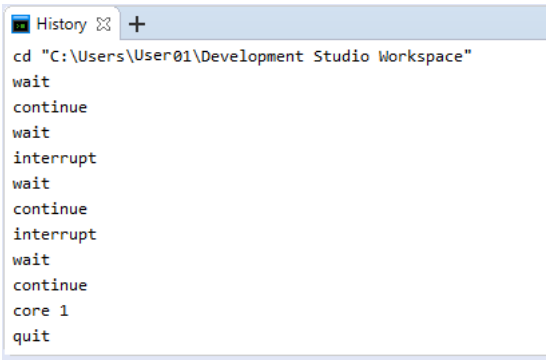


Figure 6-21 History view

Note

Default settings for this view are controlled by a Arm 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.16 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.
- 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 4 bytes.
- Display the ASCII character equivalent of the memory values.
- Freeze the view to prevent it from being updated by a running target.

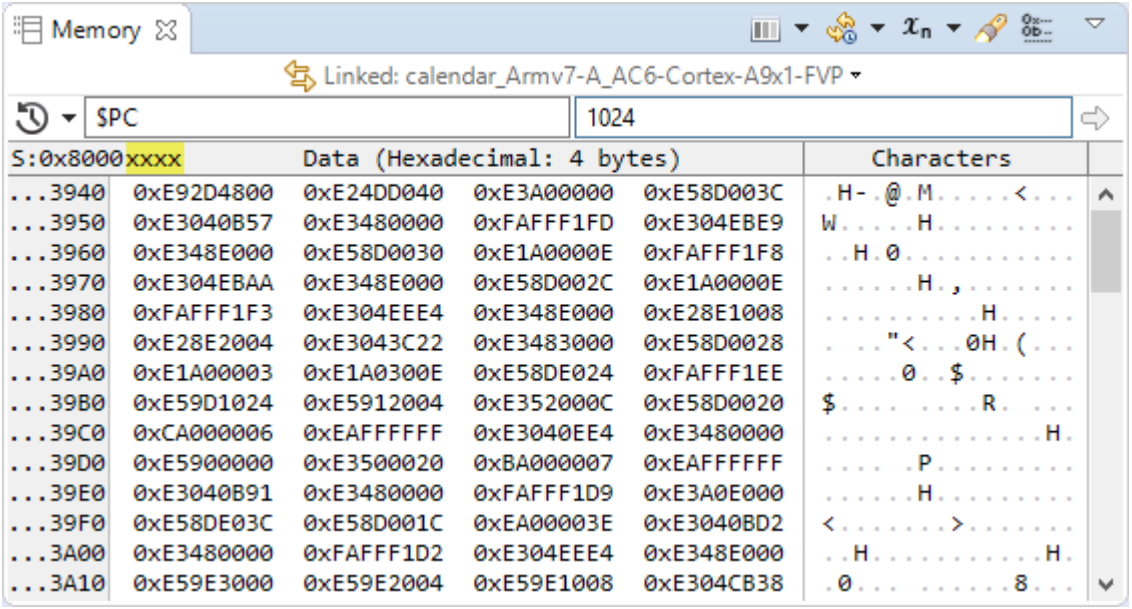


Figure 6-22 Memory view

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 button

Addresses and expressions you specify in the **Address** field are added to the list, and persist until you clear the history list. If you want to keep an expression for later use, add it to the **Expressions** view.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Number of columns

The options enable you to resize the number of columns shown in the **Memory** view.

Fit to view

Select to resize the number of columns automatically.

2, 4, 8, 16

Select the number of columns to display in the **Memory** view in a 2, 4, 8, or 16-column layout.

Custom

Select and specify the custom column layout you require.

To specify these values as default, set them under **Window** > **Preferences** > **Arm DS** > **Debugger** > **Memory View** .

Timed auto refresh

Use the **Timed Auto-Refresh** option to specify an auto-refresh of the values in the view.

Start

Starts the timed auto-refresh.

Stop

Stops the timed auto-refresh.

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 processor core, so the memory content returned might be different to the value that the core reads.

Properties

Displays the *Timed Auto-Refresh Properties* on page 6-234 dialog box where you can specify these options as default.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Use the **Hide Base Prefix** option to hide base prefixes where applicable.

For example:

- For hexadecimal values, this option hides the preceding `0x`. The value `0xEA000016` is shown as `EA000016`.
- For binary values, this option hides the preceding `0b`. The value `0b00010110` is shown as `00010110`.
- For octal values, this option hides the preceding `0`. The value `035200000026` is shown as `35200000026`.

Search

Searches through debug information for symbols.

Details panel

Show or hide the details panel, which displays the value of the selected memory cell in different formats. **Memory view with details panel**

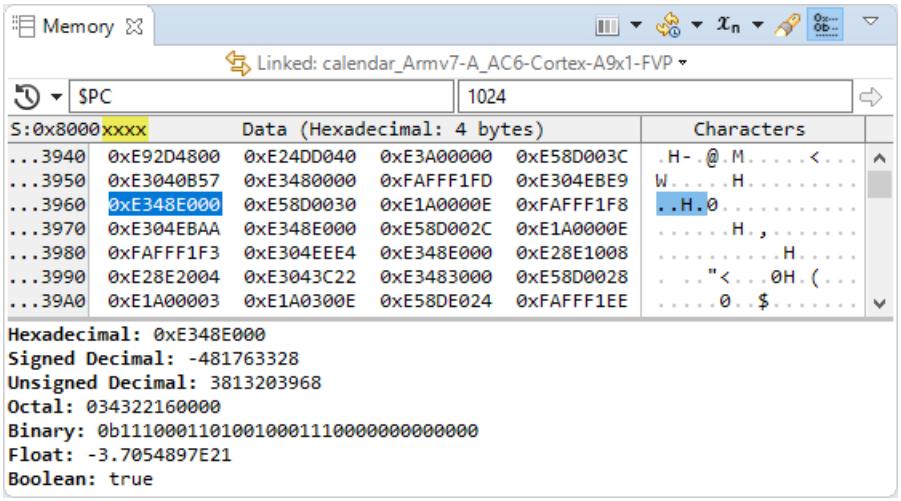


Figure 6-23 Memory view with details panel

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 clear the view history. If you want to keep an expression for later use, add it to the **Expressions** view.

Size field

The number of bytes to display.

View Menu

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

New Memory View

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

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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. If populated, the cache columns display the state of each cache using the Modified/Owned/Exclusive/Shared/Invalid(MOESI) cache coherency protocol notation.

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.

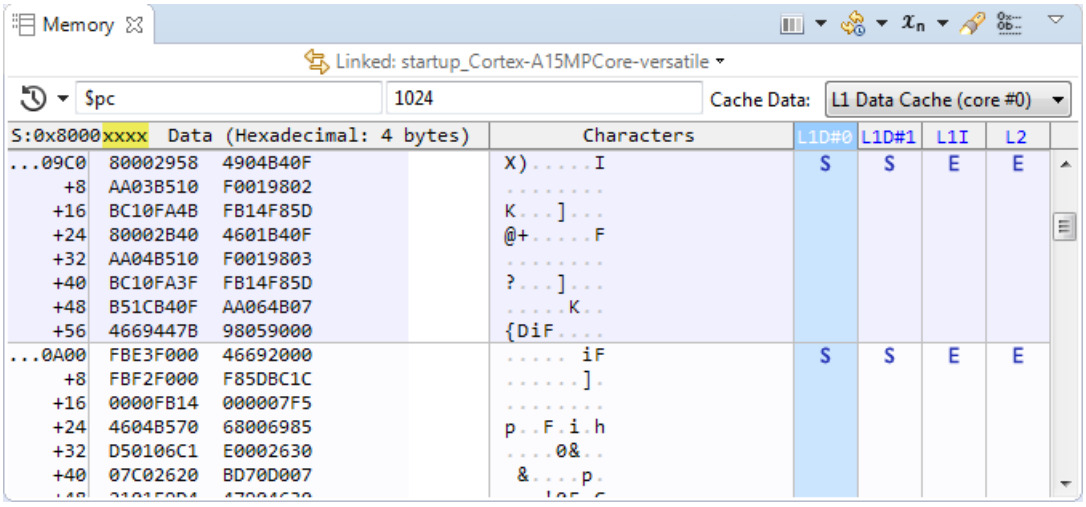


Figure 6-24 Memory view with Show Cache option enabled

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.

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 list.

Import Memory

Reads data from a file and writes it to memory.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Export Memory

Reads data from memory and writes it to a file.

Fill Memory

Writes a specific pattern of bytes to memory

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 option also disables or enables the **Address** and **Size** fields. You can still use the **Refresh** option to manually refresh the view.

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.

Characters

Displays the **Characters** column.

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. After a watchpoint is set, you can:

Enable Watchpoint

If disabled, enables the watchpoint at the selected address.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Disable Watchpoint

Disables the watchpoint at the selected address.

Remove Watchpoint

Removes the watchpoint at the selected address.

Watchpoint Properties

Displays and lets you change the watchpoint properties.

Translate Address <address>

Displays the **MMU** view and translates the address of the selected value in memory.

Memory view default preferences

You can specify default preferences to apply to the **Memory** view. Specifying default options ensures that your preferences are applied across all debug connections.

To specify default options for the **Memory** view, set them under **Window** > **Preferences** > **Arm DS** > **Debugger** > **Memory View**.

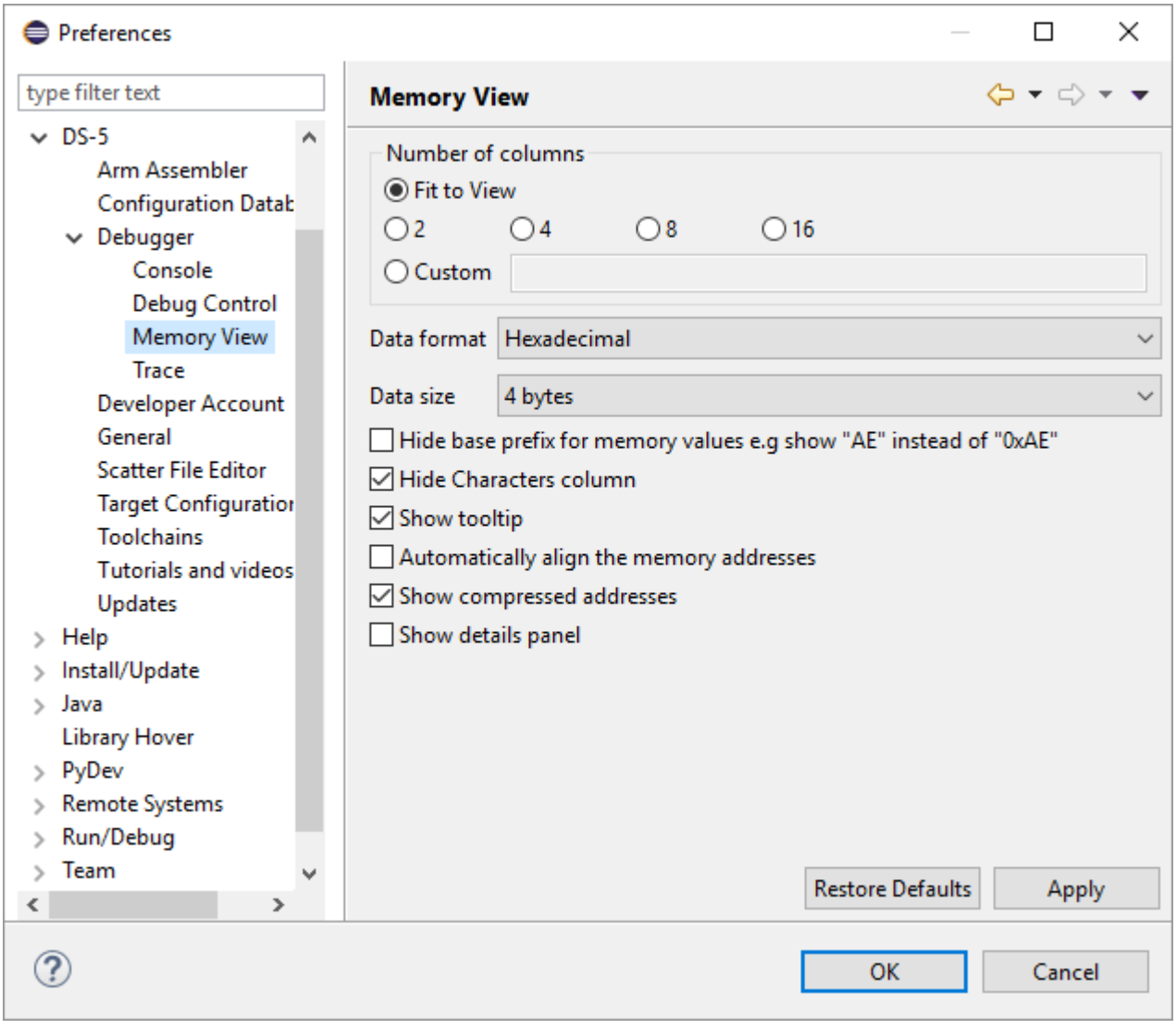


Figure 6-25 Memory View preferences

Number of columns

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The options enable you to resize the number of columns shown in the **Memory** view.

Fit to view

Select to resize the number of columns automatically.

2, 4, 8, 16

Select the number of columns to display in the **Memory** view in a 2, 4, 8, or 16-column layout.

Custom

Select and specify the custom column layout you require.

Data format

Specify the format of the memory cell values. The default is hexadecimal.

Data size

Set the width of the memory cells in the **Memory** view. The default is 4 bytes.

Hide base prefix for memory values

Select to hide the base prefix where applicable.

For example:

- For hexadecimal values, this option hides the preceding **0x**. So the value **0xEA000016** is shown as **EA000016**.
- For binary values, this option hides the preceding **0b**. So the value **0b00010110** is shown as **00010110**.
- For octal values, this option hides the preceding **0**. So the value **035200000026** is shown as **35200000026**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Hide Characters column

Select to the hide the **Characters** column in the **Memory** view.

When this option is unselected, ASCII characters are shown in the **Memory** view: **Show Characters column in the Memory view**

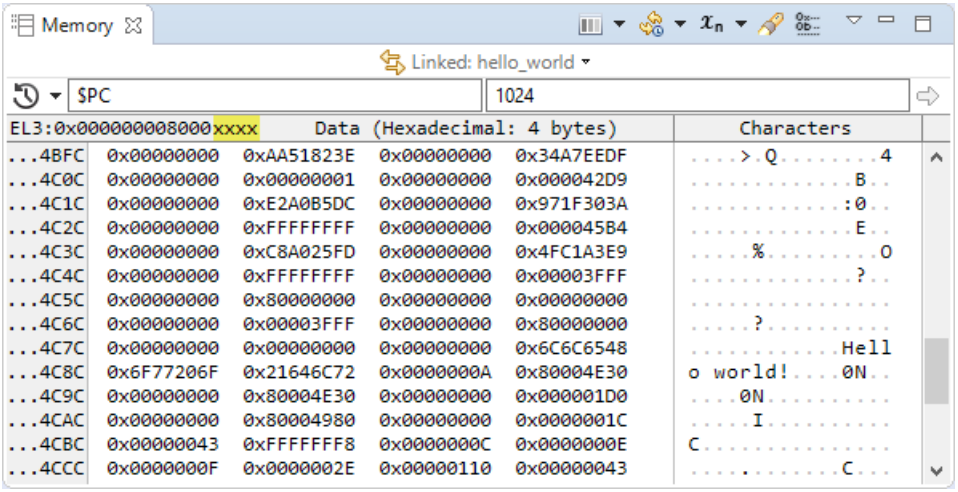


Figure 6-26 Show Characters column in the Memory view

Another way to toggle the visibility of the **Characters** column, is to right-click the context menu and select or unselect the Characters option: **Select Characters from context menu to show theASCII characters**

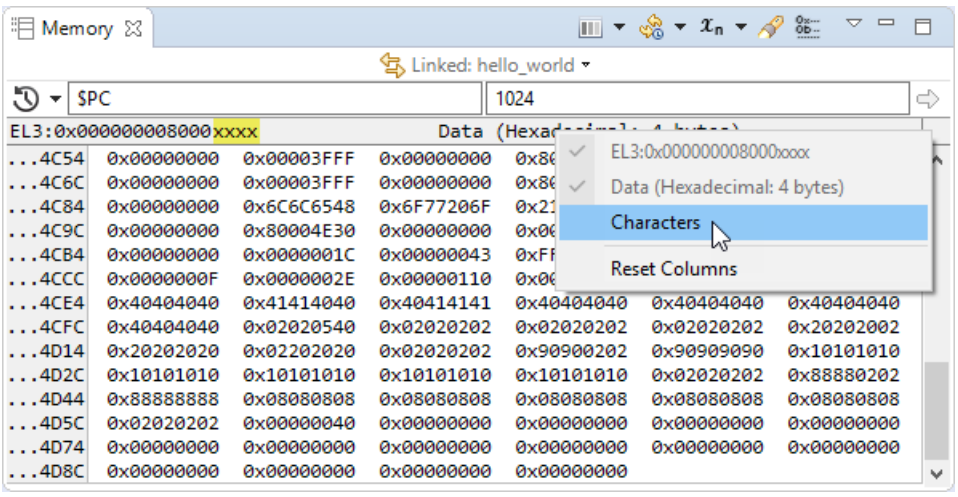


Figure 6-27 Select Characters from context menu to show the ASCII characters

Show tooltip

Select to control the display of tooltips on memory cell values.

Automatically align the memory addresses

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 details panel

Shows the details panel, which displays the value of the selected memory cell in different formats.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Related concepts

1.8 About debugging multi-threaded applications on page 1-25
1.9 About debugging shared libraries on page 1-26
1.10 About debugging TrustZone enabled targets on page 1-28

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.17 MMU/MPU view

Use the **MMU/MPU** 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.
- Inspect and traverse MMU and MPU tables.
- See an overview of the virtual memory map.
- Freeze the view to prevent it from being updated by a running target.

Note

MMU awareness is only supported on Armv7-A and Armv8-A architectures. MPU awareness is only supported on Armv8-M architecture.

MMU/MPU Translation tab

For targets that support address translations, the **Translation** tab enables you to translate:

- Virtual address to physical address.
- Physical address to one or more virtual addresses.

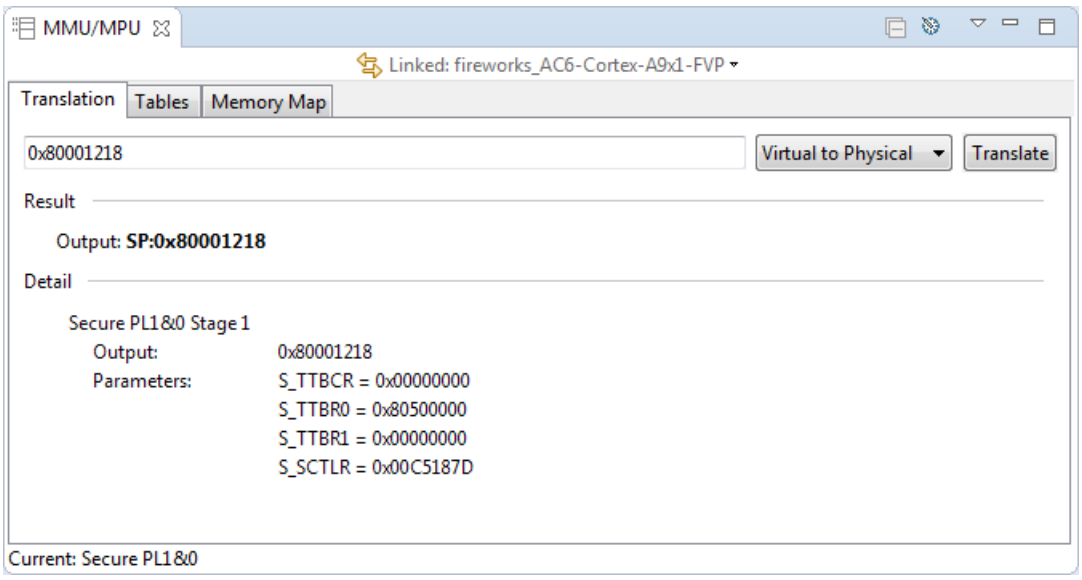


Figure 6-28 MMU/MPU Translation tab view

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 box.

MMU/MPU Tables tab

The **Tables** tab displays the content of tables used by the MMU or MPU. For targets with multiple translation regimes, you can change the translation regime using the **MMU Settings** dialog box.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

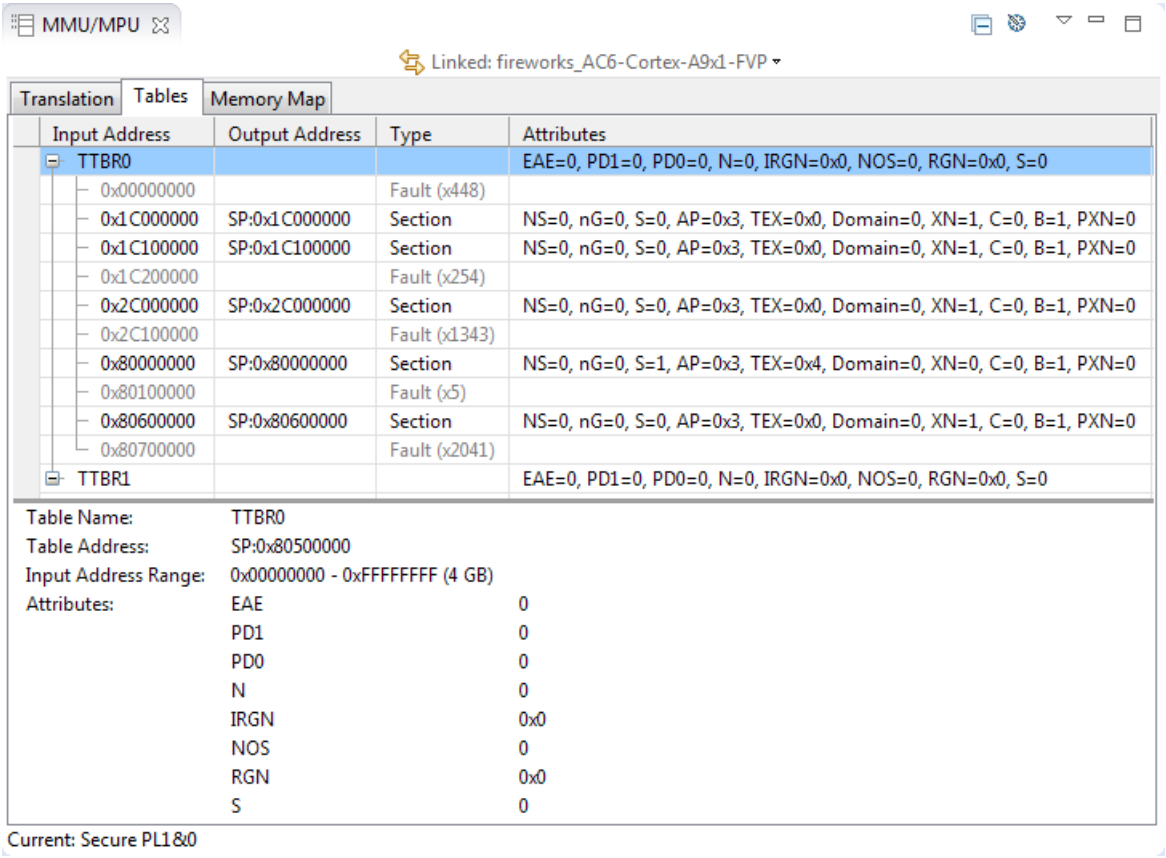


Figure 6-29 MMU/MPU Tables tab view

For targets which support address translation, 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 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Next-level Table Address

Specifies the **Descriptor Address** for the next level of lookup in the translation table.

For targets which do not support address translation, the **Tables** tab contains the following columns:

Base

Specifies the base address of the region.

Note

For Armv8-M targets which have an Implementation Defined Attribution Unit (IDAU) region, you must define the IDAU region of your target before the **Tables** tab can display region information. See **setidau-region** command documentation for details.

Limit

Specifies the last address of the region.

Type

Specifies the region type.

Attributes

Specifies the memory attributes of the IDAU region.

MMU/MPU Memory Map tab view

The **Memory Map** tab provides a view of the virtual memory layout by combining the MMU or MPU table entries that map contiguous regions of memory with a common memory type, for example, cacheability, shareability, and access attributes.

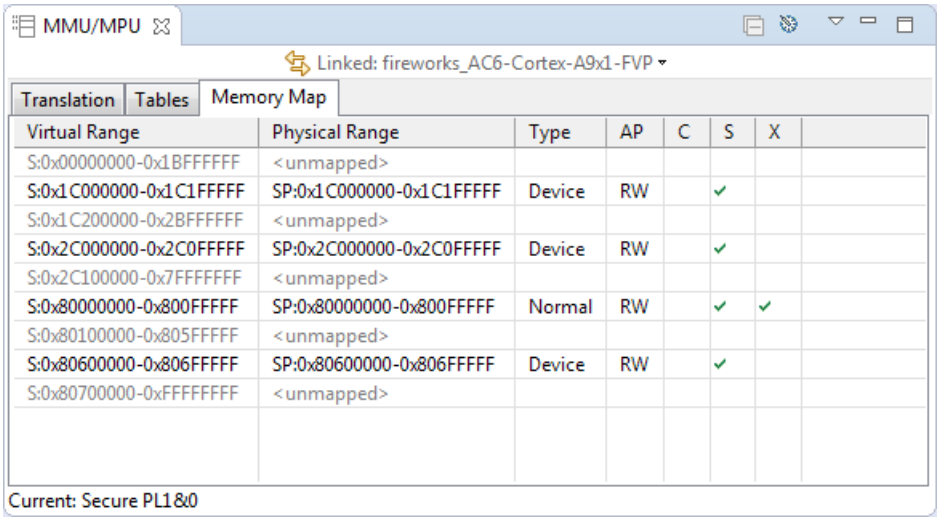


Figure 6-30 MMU/MPU Memory Map tab view

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 **DebugControl** view.

MMU settings

This enables you to change the translation regime and input parameters. It contains:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

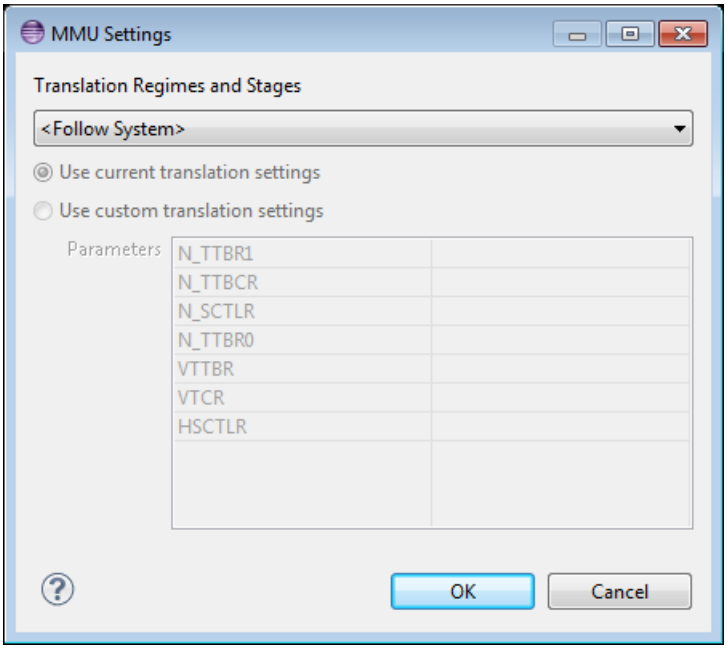


Figure 6-31 MMU settings

The **MMU Settings** dialog box 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 Arm 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/MPU View

Displays a new instance of the **MMU/MPU** 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Freeze Data

Toggles the freezing of data in the current view. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

Coalesce Invalid Entries

Condenses the contiguous rows of faulty or invalid input addresses into a single row in the **Tables** tab.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.18 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.

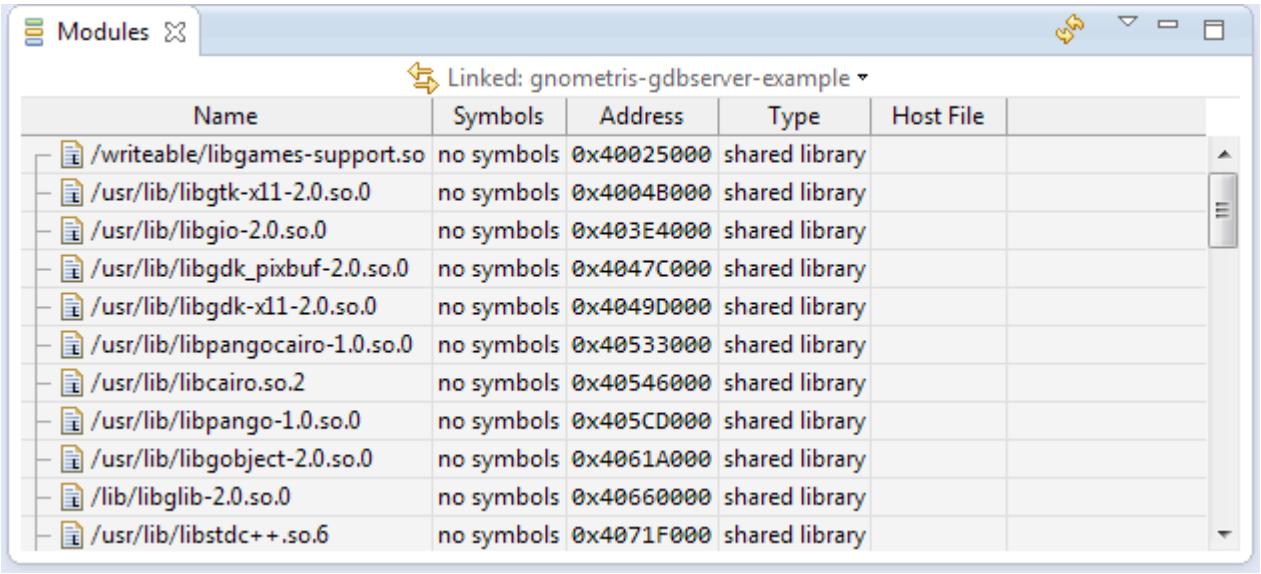


Figure 6-32 Modules view showing shared libraries

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

1.8 About debugging multi-threaded applications on page 1-25
1.9 About debugging shared libraries on page 1-26
1.10 About debugging TrustZone enabled targets on page 1-28

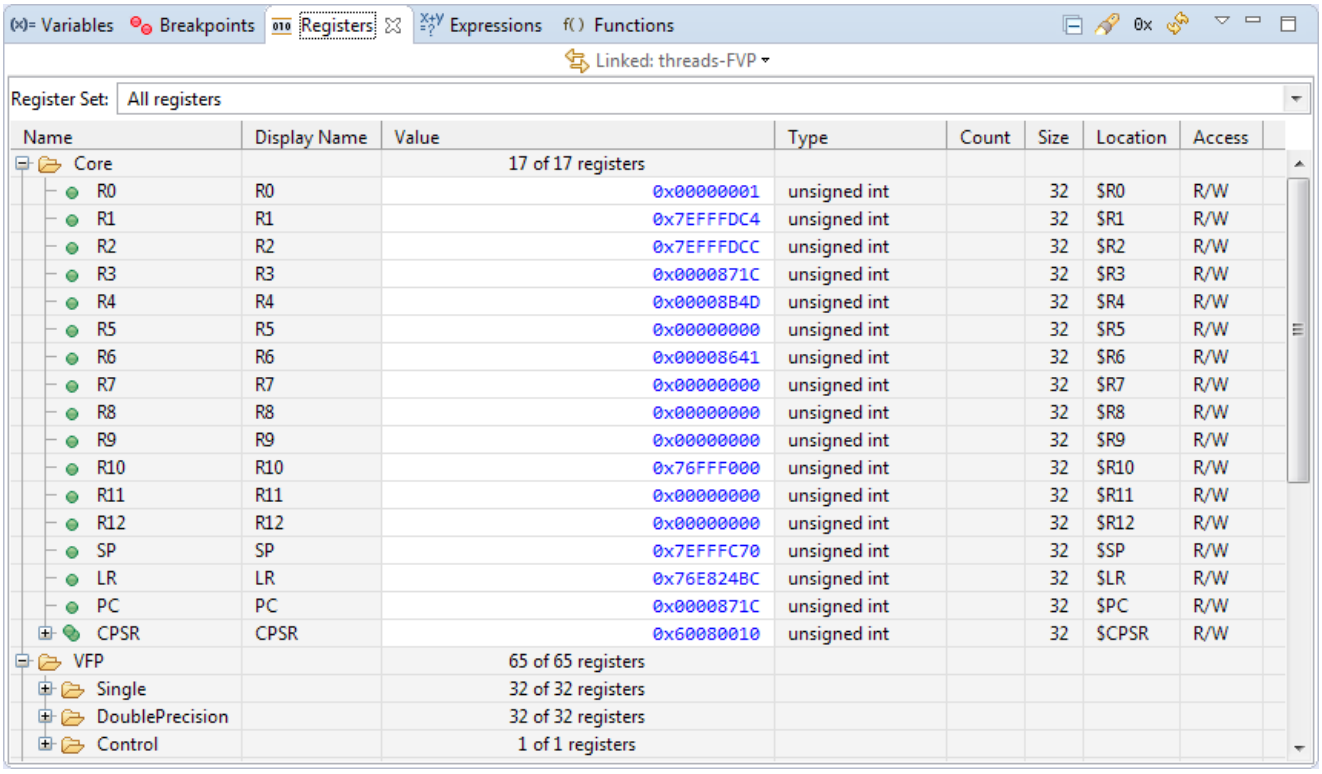
Related references

Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.19 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	0x7EFFFDCC	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	0x7EFFFC70	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
VFP		65 of 65 registers					
Single		32 of 32 registers					
DoublePrecision		32 of 32 registers					
Control		1 of 1 registers					

Figure 6-33 Registers view (with all columns displayed)

You can:

Browse registers available on your target

The **Registers** view displays all available processor registers on your target. Click and expand individual register groups to view specific registers.

Click to collapse the registers tree.

If you want to refresh the **Registers** view, from the view menu click .

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Search for a specific register

You can use the search feature in the **Registers** view to search for a specific register or group.

If you know the name of the specific register or group you want to view, click to display the search bar. Then, enter the name of the register or group you are looking for in the search bar. 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.

Press **Enter** on your keyboard, or double-click the register or group in the search results to select it in the **Register** view.

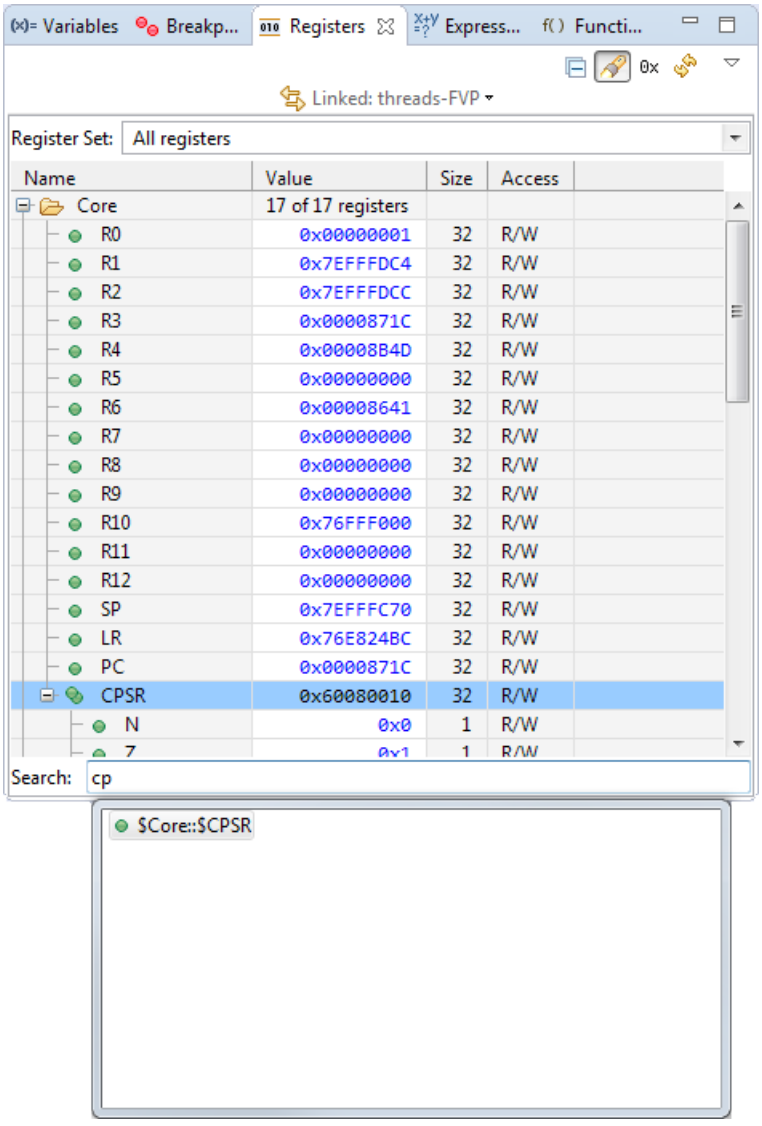


Figure 6-34 Registers - CP

Tip

You can also use **CTRL+F** on your keyboard to enable the search bar. You can use the **ESC** key on your keyboard to close the search bar.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Toggle between numerical and hexadecimal values

Click the 0x button to change all numeric values to hexadecimal values. This works as a toggle and your preference is saved across sessions.

Create and manage register sets

You can use register sets to collect individual registers into specific custom groups.

To create a register set:

1. In the **Registers** view, under **Register Set**, click **All Registers** and select Create.
2. In the **Create or Modify Register Set** dialog box:
   - Give the register set a name in **Set Name**, for example **Core registers**. You can create multiple register groups if needed.
   - Select the registers you need in **All registers**, and click **Add**. Your selected registers appear under **Chosen registers**.
   - Click **OK**, to confirm your selection and close the dialog box.
3. The **Registers** view displays the specific register group you selected.
4. To switch between various register groups, click **All Registers** and select the group you want.

To manage a register set:

1. In the **Registers** view, under **Register Set**, click **All Registers** and select Manage.
2. In the **Manage Register Sets** dialog box:
   - If you want to create a new register set, click **New** and create a new register set.
   - If you want to edit an existing register set, select a register set, and click **Edit**.
   - If you want to delete an existing register set, select a register set and click **Remove**.
3. Click **OK** to confirm your changes.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

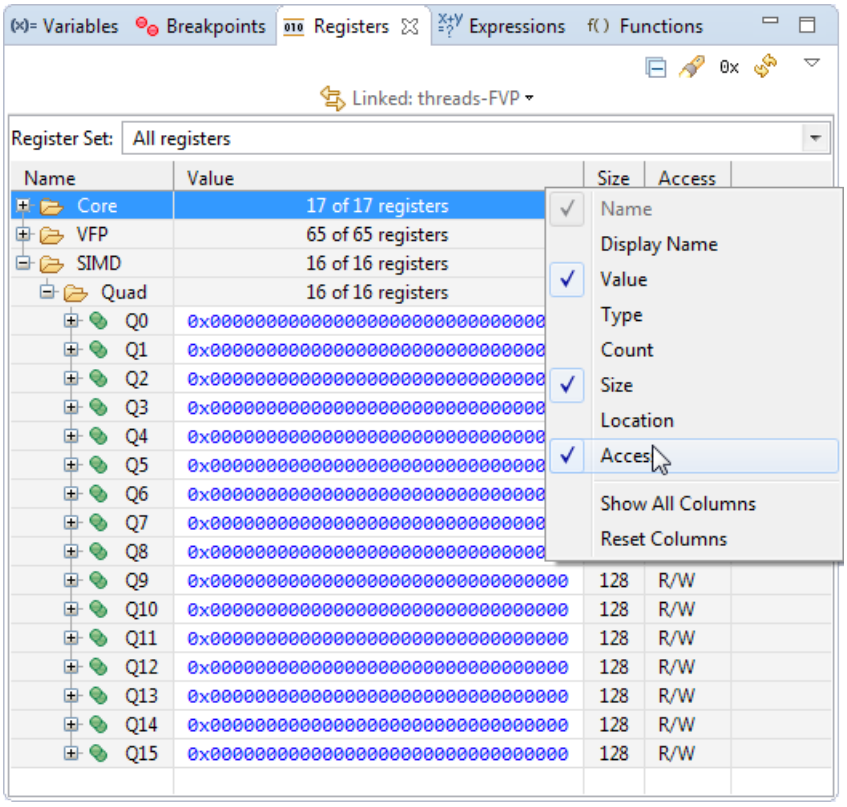


Figure 6-35 Registers access rights

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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. These formats are **Binary**, **Boolean**, **Hexadecimal**, **Octal**, **Signed Decimal**, and **Unsigned decimal**.

View Menu

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

New Registers View

Creates a new instance of the **Registers** view.

Freeze Data

Toggles the freezing of data in the current view. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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

1.8 About debugging multi-threaded applications on page 1-25
1.9 About debugging shared libraries on page 1-26
1.10 About debugging TrustZone enabled targets on page 1-28

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.20 NVIC Registers view

Use the **NVIC Registers** view to see an alternative view of the registers involved in the NVIC exception/interrupt system.

Note

- The **NVIC Registers** view is only enabled for Armv6-M and Armv7-M architectures.
- You can also use the **Registers** view to view register information.

The **NVIC Registers** view updates when registers are changed by the debugger or are manually changed through the command prompt or register view.

Each exception is in one of the following states:

Inactive

The exception is not active and not pending.

Active

An exception that is being serviced by the processor but has not completed. An exception handler can interrupt the execution of another exception handler. In this case, both exceptions are in the active state.

Pending

The exception is waiting to be serviced by the processor. An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.

Active and pending

The exception is being serviced by the processor and there is a pending exception from the same source.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

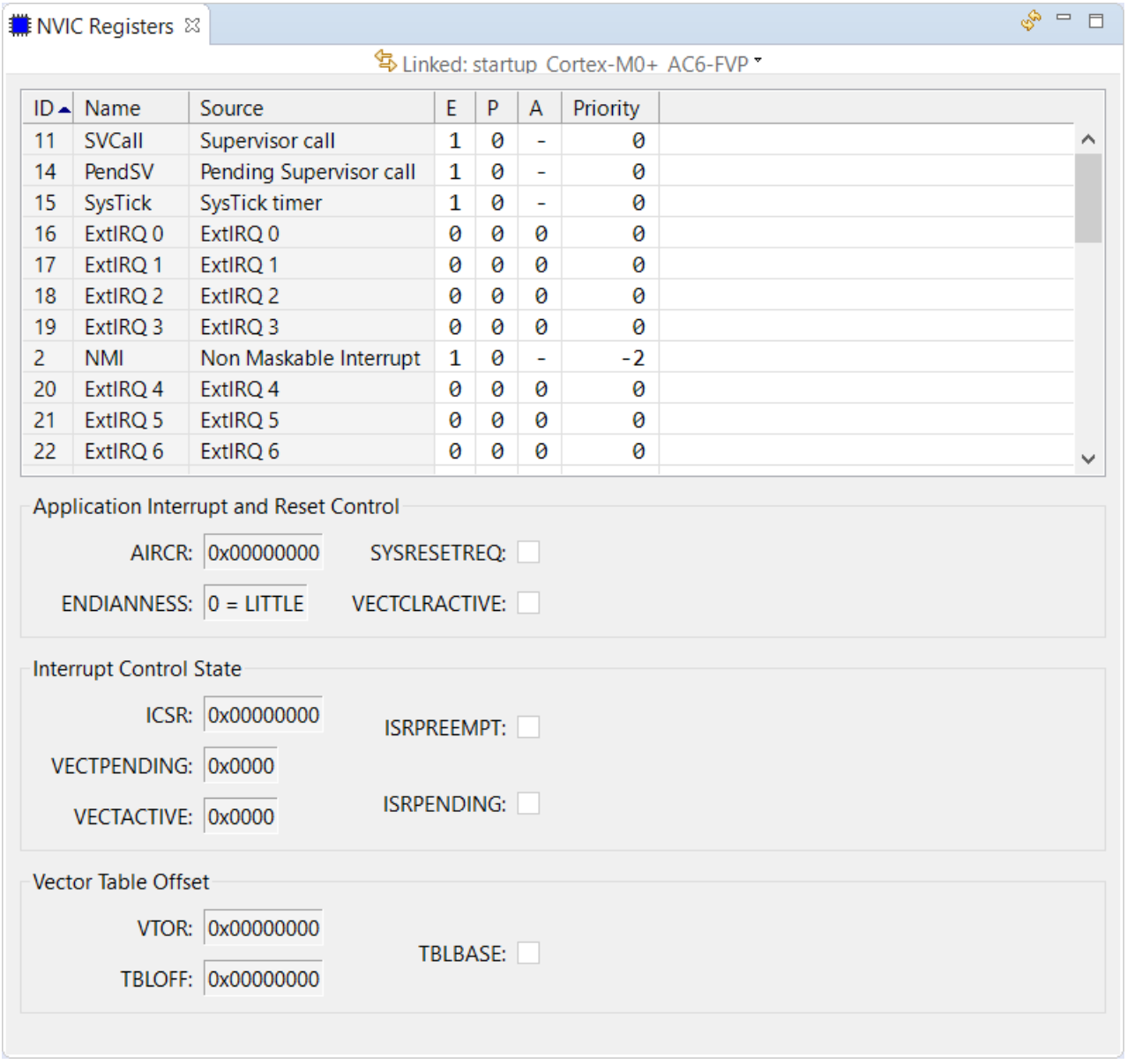


Figure 6-36 NVIC Registers view

You can view:

Current exceptions and interrupts

A table showing the name and status of current exceptions and interrupts.

ID

ID of the exception or interrupt. Entries with an ID of up to, and including, 16 are the system exceptions. The remaining entries are external interrupts extracted from the NVIC_* group of system registers.

Note

This exact number will vary between platforms.

Name

Name of the exception or interrupt.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Source

Source of the exception or interrupt.

E

Enabled state of the exception or interrupt. The value 1 indicates True, 0 indicates False, and - indicates not applicable.

P

Pending state of the exception or interrupt. The value 1 indicates True, 0 indicates False, and - indicates not applicable.

A

Active state of the exception or interrupt. The value 1 indicates True, 0 indicates False, and - indicates not applicable.

Priority

The priority of the exception or interrupt.

Application Interrupt and Reset Control

Displays the Application Interrupt and Register Control Register (AIRCR) information.

Interrupt Control State

Displays the Interrupt Control and State Register (ICSR) information.

Vector Table Offset

Displays the Vector Table Offset Register (VTOR) information.

Note

The VTOR information is only available for Armv7-M architectures.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

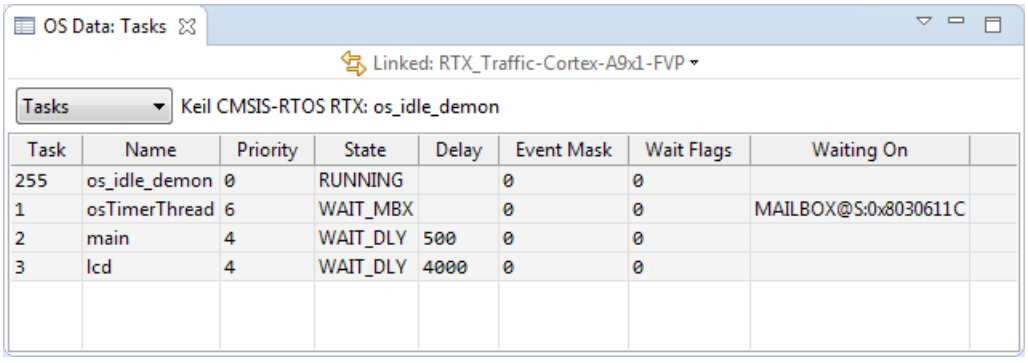
6.21 OS Data view

Use the **OS Data** view to display OS awareness information for RTOSs. You can view details about tasks, semaphores, mutexes, and mailboxes.

Note

The **OS Data** view is not used when debugging Linux applications. To view the running threads information for Linux applications, use the **Debug Control** view.

To view information, select a table from the list.



Task	Name	Priority	State	Delay	Event Mask	Wait Flags	Waiting On
255	os_idle_demon	0	RUNNING		0	0	
1	osTimerThread	6	WAIT_MBX		0	0	MAILBOX@S:0x8030611C
2	main	4	WAIT_DLY	500	0	0	
3	lcd	4	WAIT_DLY	4000	0	0	

Figure 6-37 OS Data view (showing Keil CMSIS-RTOS RTX Tasks)

Note

Data in the **OS Data** view depends 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 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Freeze Data

Toggles the freezing of data in the current view. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

Note

If the data is frozen, the value of a variable cannot be changed.

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.

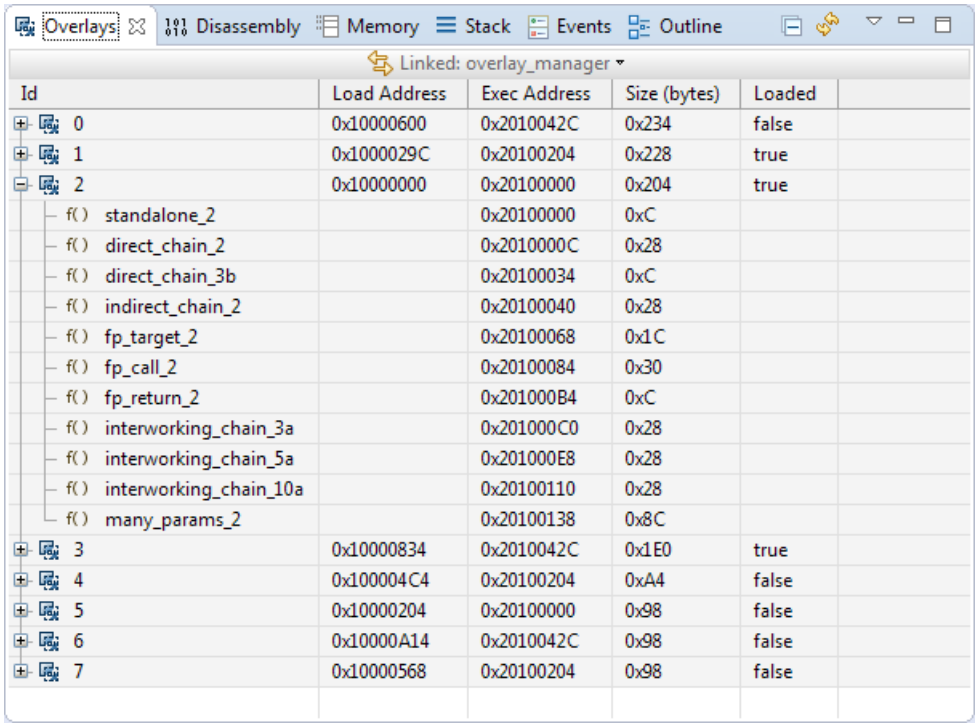
Select All

Selects all text.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.22 Overlays view

Use the **Overlays** view to interact with the currently loaded overlaid application.



Id	Load Address	Exec Address	Size (bytes)	Loaded
0	0x10000600	0x2010042C	0x234	false
1	0x1000029C	0x20100204	0x228	true
2	0x10000000	0x20100000	0x204	true
f() standalone_2		0x20100000	0xC	
f() direct_chain_2		0x2010000C	0x28	
f() direct_chain_3b		0x20100034	0xC	
f() indirect_chain_2		0x20100040	0x28	
f() fp_target_2		0x20100068	0x1C	
f() fp_call_2		0x20100084	0x30	
f() fp_return_2		0x201000B4	0xC	
f() interworking_chain_3a		0x201000C0	0x28	
f() interworking_chain_5a		0x201000E8	0x28	
f() interworking_chain_10a		0x20100110	0x28	
f() many_params_2		0x20100138	0x8C	
3	0x10000834	0x2010042C	0x1E0	true
4	0x100004C4	0x20100204	0xA4	false
5	0x10000204	0x20100000	0x98	false
6	0x10000A14	0x2010042C	0x98	false
7	0x10000568	0x20100204	0x98	false

Figure 6-38 Overlays view

Using the **Overlays** view, you can:

View information about overlays

The **Overlays** view shows the **ID** and functions, the **Load Address**, **Exec Address**, and **Size** for each overlay. It also shows if the overlay is **Loaded** or not.

Locate the function in the source

To locate the function in the source code and move to the **Code Editor** view, double-click a function.

Toolbar options

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

New Overlays View

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

Freeze Data

Toggles the freezing of data in the currently selected execution context. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

Related concepts

1.14 About Arm® Debugger support for overlays on page 1-36

Related information

info overlays command

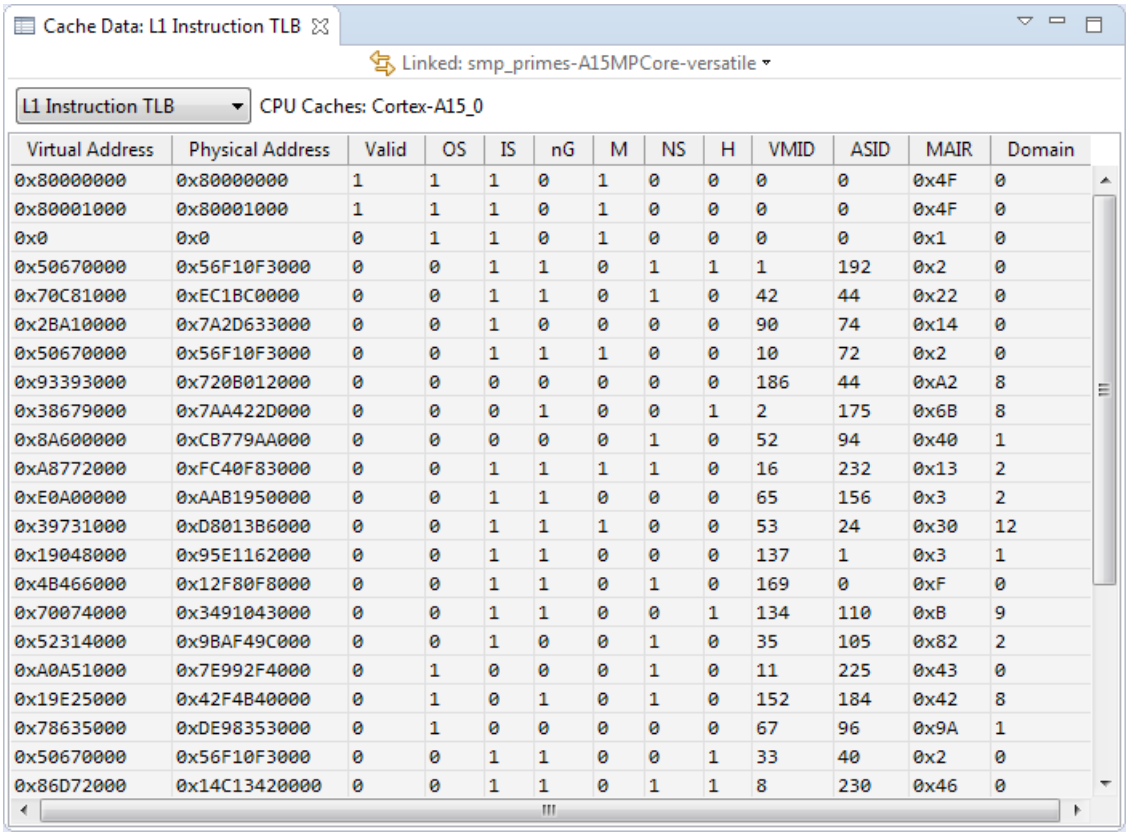
set overlays enabled command

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.23 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 box.

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
0x80000000	0x80000000	1	1	1	0	1	0	0	0	0	0x4F	0
0x80001000	0x80001000	1	1	1	0	1	0	0	0	0	0x4F	0
0x0	0x0	0	1	1	0	1	0	0	0	0	0x1	0
0x50670000	0x56F10F3000	0	0	1	1	0	1	1	1	192	0x2	0
0x70C81000	0xEC1BC0000	0	0	1	1	0	1	0	42	44	0x22	0
0x2BA10000	0x7A2D633000	0	0	1	0	0	0	0	90	74	0x14	0
0x50670000	0x56F10F3000	0	0	1	1	1	0	0	10	72	0x2	0
0x93393000	0x720B012000	0	0	0	0	0	0	0	186	44	0xA2	8
0x38679000	0x7AA422D000	0	0	0	1	0	0	1	2	175	0x6B	8
0x8A600000	0xCB779AA000	0	0	0	0	0	1	0	52	94	0x40	1
0xA8772000	0xFC40F83000	0	0	1	1	1	1	0	16	232	0x13	2
0xE0A00000	0xAAB1950000	0	0	1	1	0	0	0	65	156	0x3	2
0x39731000	0xD8013B6000	0	0	1	1	1	0	0	53	24	0x30	12
0x19048000	0x95E1162000	0	0	1	1	0	0	0	137	1	0x3	1
0x4B466000	0x12F80F8000	0	0	1	1	0	1	0	169	0	0xF	0
0x70074000	0x3491043000	0	0	1	1	0	0	1	134	110	0xB	9
0x52314000	0x9BAF49C000	0	0	1	0	0	1	0	35	105	0x82	2
0xA0A51000	0x7E992F4000	0	1	0	0	0	1	0	11	225	0x43	0
0x19E25000	0x42F4B40000	0	1	0	1	0	1	0	152	184	0x42	8
0x78635000	0xDE98353000	0	1	0	0	0	0	0	67	96	0x9A	1
0x50670000	0x56F10F3000	0	0	1	1	0	0	1	33	40	0x2	0
0x86D72000	0x14C13420000	0	0	1	1	0	1	1	8	230	0x46	0

Figure 6-39 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:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

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

1.13 About debugging caches on page 1-33

Related references

6.50 DTSL Configuration Editor dialog box on page 6-267
6.16 Memory view on page 6-179

Related information

Debug command: cache

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.24 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:

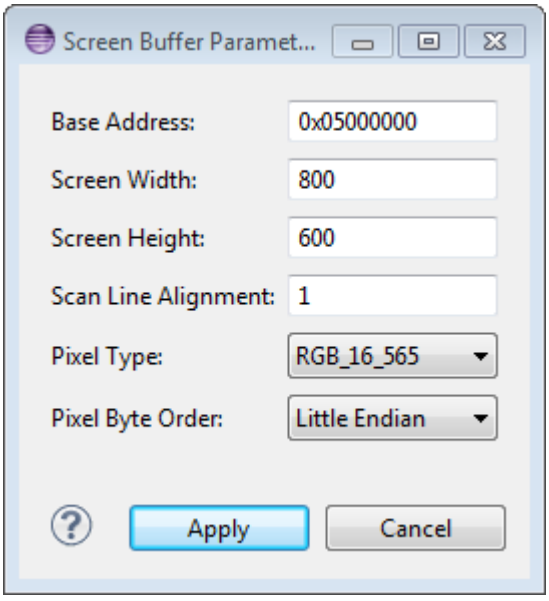


Figure 6-40 Screen buffer parameters

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

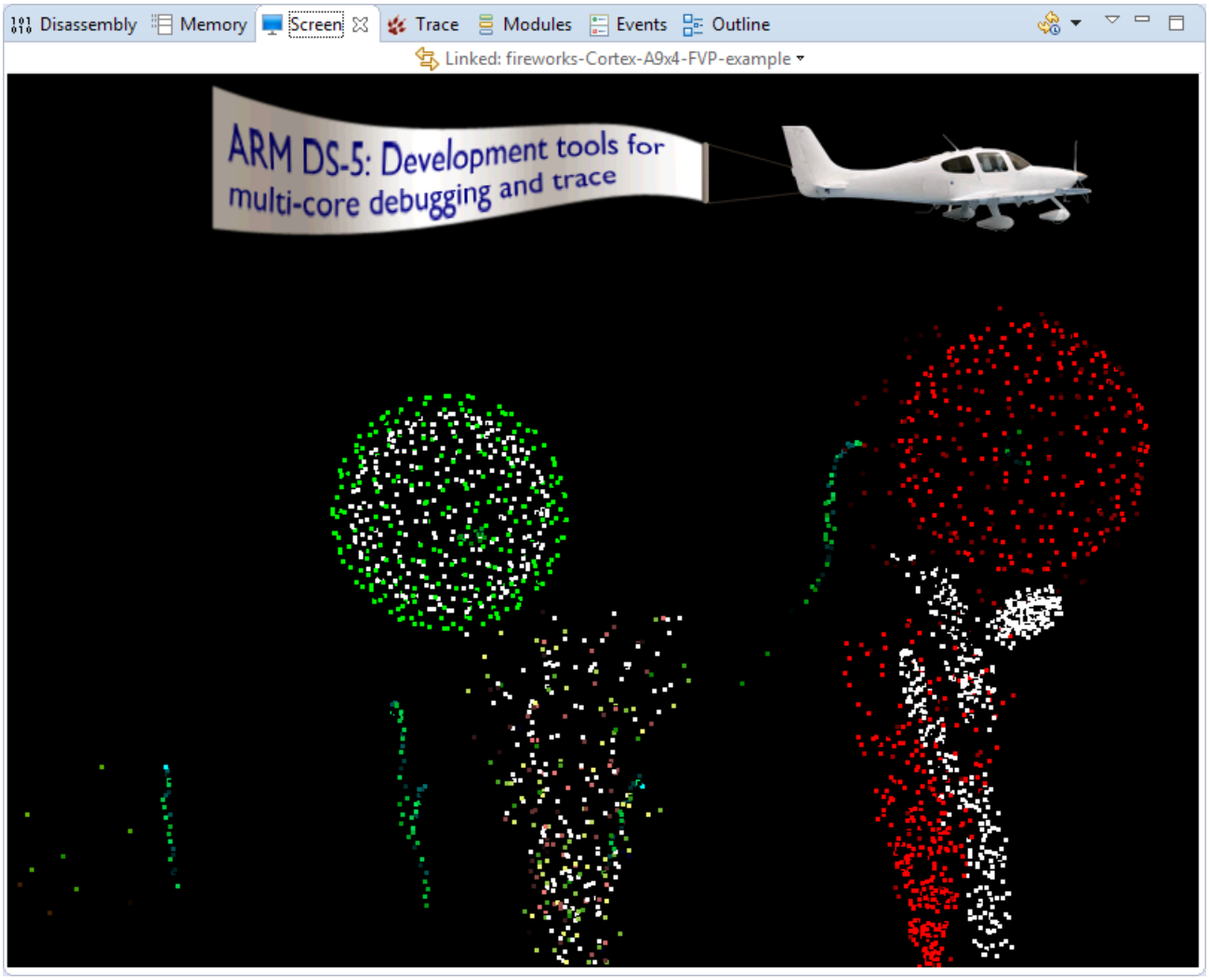


Figure 6-41 Screen view

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

Operation is as follows:

- 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view.

The Screen view is not visible by default. To add this view:

1. Ensure that you are in the Development Studio perspective.
2. Select **Window** > **Show View** to open the Show View dialog box.
3. Select **Screen** view.

Related references

Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.25 Scripts view

Use the **Scripts** view to work with scripts in Arm Development Studio. You can create, run, edit, delete, and configure parameters for the various types of scripts supported by Development Studio.

Note

- Scripts are dependent on a debug connection. To work with scripts, first create a debug configuration for your target and select it in the **Debug Control** view.
- Debugger views are not updated when commands issued in a script are executed.

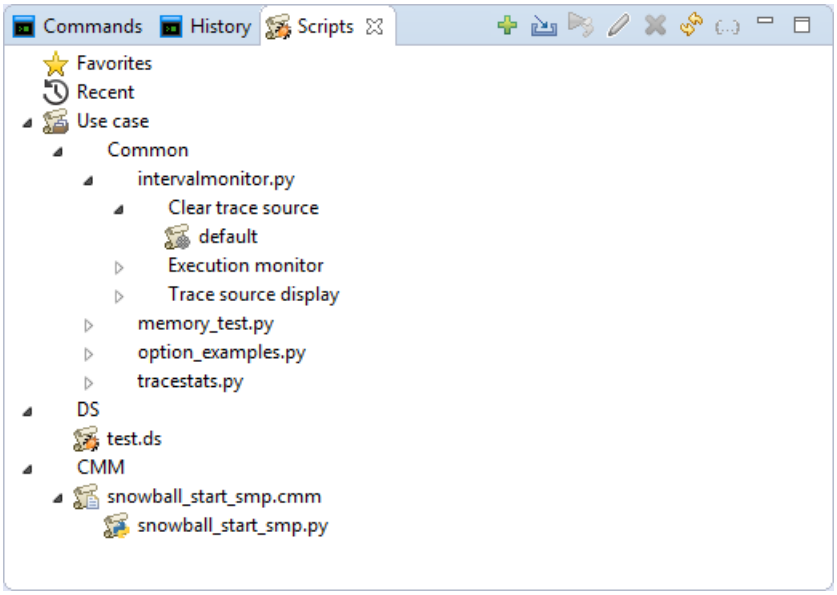


Figure 6-42 Scripts view

Using the **Scripts** view, you can:

Create a script

To create a script, click to display the **Save As** dialog box. Give your script a name and select the type of script you want. You can choose any of the following types:

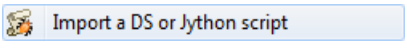
- Debugger Script (`*.ds`)
- Jython script (`*.py`)
- Text file (`*.txt`)

The script opens in the editor when you save it.

Import scripts

Click to view options for importing scripts.

Import an Arm DS or Jython script

To import a Development Studio or Jython script, click 

and browse and select your file.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Import and translate a CMM script

To *import and translate* on page 5-104 a CMM script:

1. Click Import and translate a CMM script and browse and select your file.
2. Click **Open**.
3. In the **Translation Save Location** dialog box, choose a location to store the translated file.
4. Click **OK** to translate the file and save it at the selected location.

Add additional use case script directories

To add more use case script directories, click Add use case script directory and either enter the folder location or click **Browse** to the script folder location.

Note

To change the location for other scripts that are supported in Arm Debugger, from the main menu, select **Window** > **Preferences** > **Arm DS** > **Debugger** > **Console** . Then, select the **Use a default script folder** option, and either enter the folder location or click **Browse** to the script folder location.

Execute your script

After connecting to your target, select your script and click to execute your script. You can also double-click a script to run it.

Note

When working with use case scripts, you must select the use case configuration to run your script.

Edit your script

Select your script and click to open the script in the editor.

Delete your script

Select the script that you want to delete and click . Confirm if you want to delete the script from the view, or if you want to delete from the disk, and click **OK**.

Refresh the script view

Click to refresh the view.

Configure script options

Select a script, and click to configure script options.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Recents and Favorites

After you run a script, you can easily access it again from the **Recent** list. The **Scripts** view stores the five most recently run scripts. When you run a new script it appears at the top of the list.

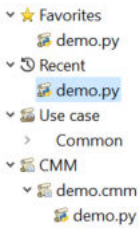


Figure 6-43 Recent scripts

Add to Favorites

You can add your frequently accessed scripts to a list of favorites to easily access them later. To add a script to the list of favorites, right-click the script, and select **Add to favorites**.

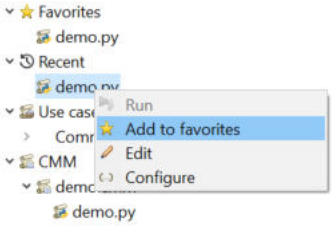


Figure 6-44 Add to favorites

Remove from favorites

To remove your scripts from the **Favorites** list, right-click a script and select **Remove from favorites**.

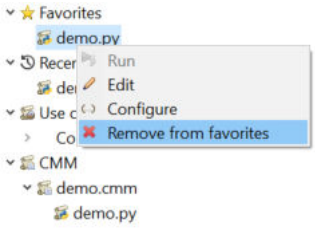


Figure 6-45 Remove from favorites

To delete multiple scripts, select them with Ctrl+click and press the **Delete** key.

Tip

You can also access your favorites and recently accessed scripts from the *Commands* on page 6-148 view.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

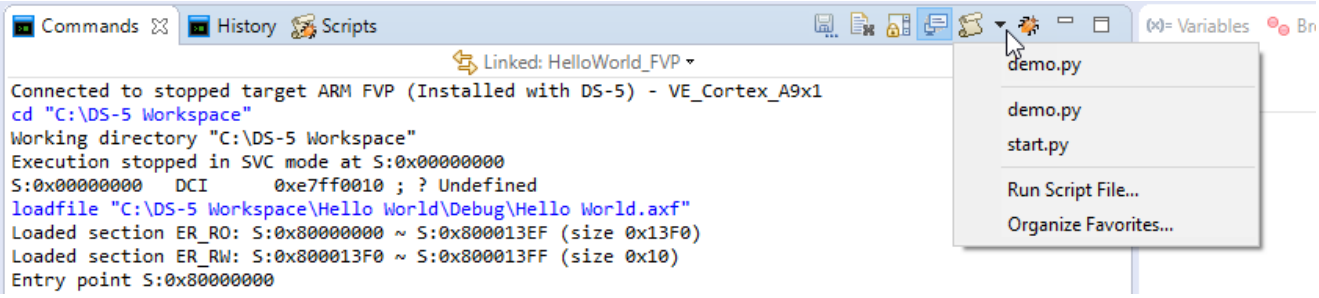


Figure 6-46 Running scripts from the Commands view

Related concepts

5.10 Use case scripts on page 5-118

Related references

Chapter 5 Debugging with Scripts on page 5-100

Chapter 4 Running Arm® Debugger from the operating system command-line or from a script on page 4-80

5.4 Support for importing and translating CMM scripts on page 5-104

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.26 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 Arm 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.

Bring to Front when target output is detected

Enabled by default. The debugger automatically changes the focus to this view when target output is detected.

Copy

Copies the selected text.

Paste

Pastes text that you have previously copied.

Select All

Selects all text.

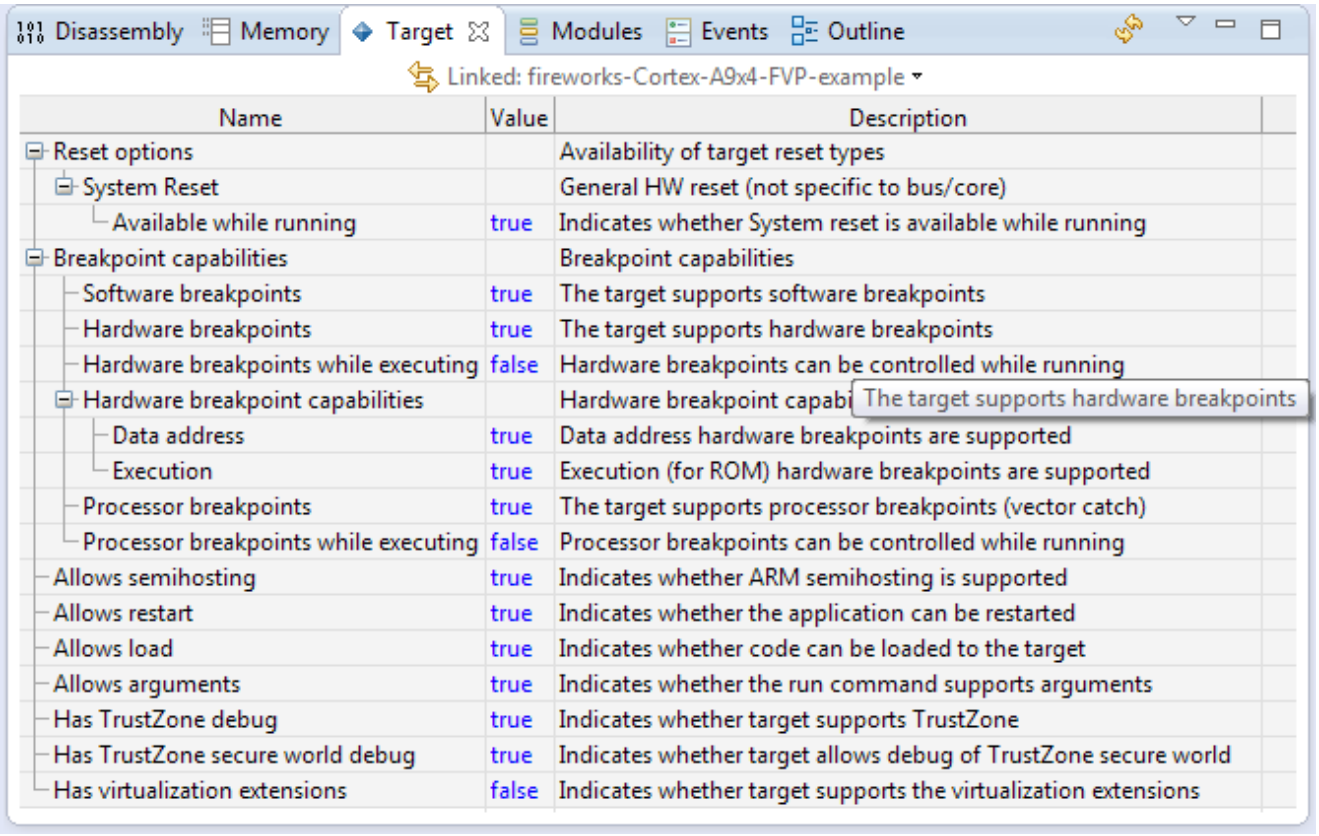
Related references

Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.27 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 capabi
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 6-47 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The **Target** view is not visible by default. To add this view:

1. Ensure that you are in the **Development Studio** 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 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.28 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.

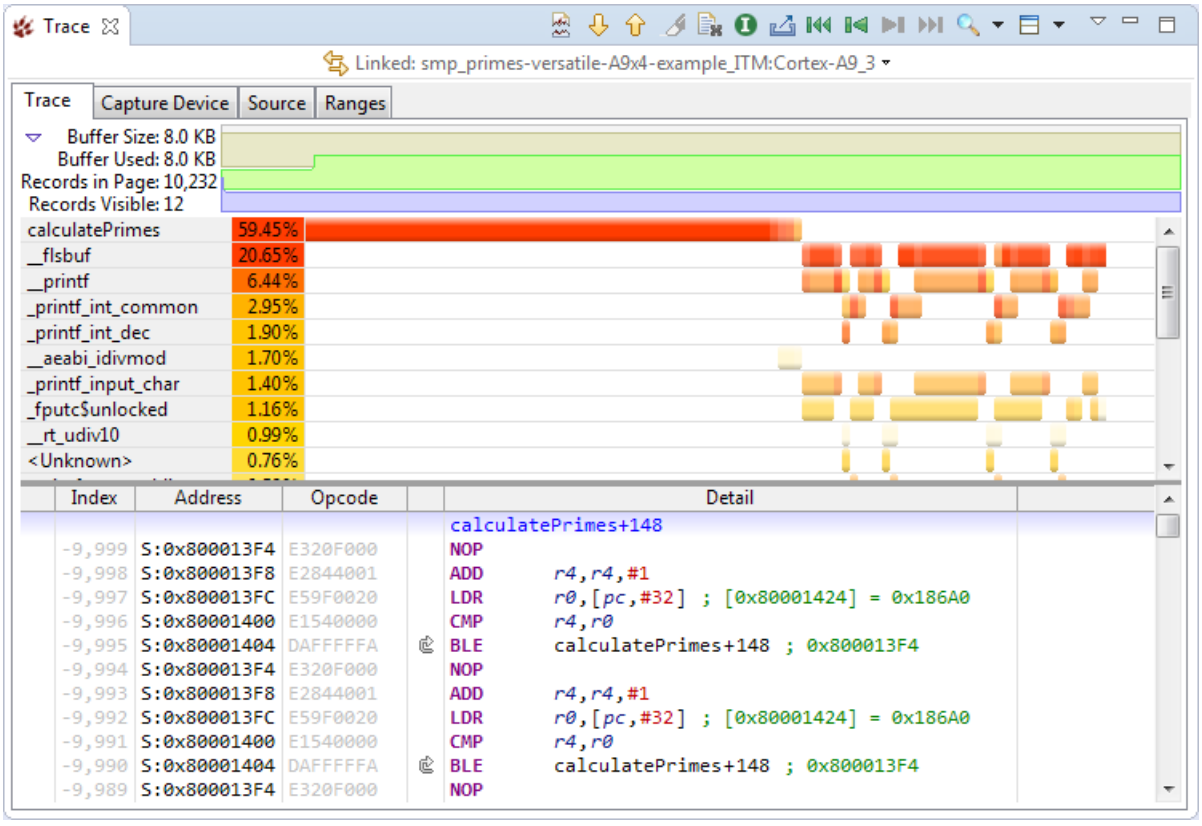


Figure 6-48 Trace view with a scale of 100:1

The **Trace** view might not be visible by default. To add this view:

1. Ensure that you are in the **Development Studio** 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the 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.

When your code hits a Trace stop point, the **Trace** tab shows:

Index

The number of instructions before the Trace stop point. The index at the Trace stop point is 0. The index for the instruction immediately before that is -1, and so on.

Address

The address in memory of the instruction.

Opcode

The opcode for the instruction expressed in hexadecimal.

Detail

The disassembly of the instruction, trace events, and errors.

The unlabeled column (to the right of the Opcode column) displays symbols that give additional information. For example, an exception, a backward branch, an instruction that was canceled before completion, or an instruction that was not executed. Hover your mouse pointer over one of these symbols to display context-sensitive help.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

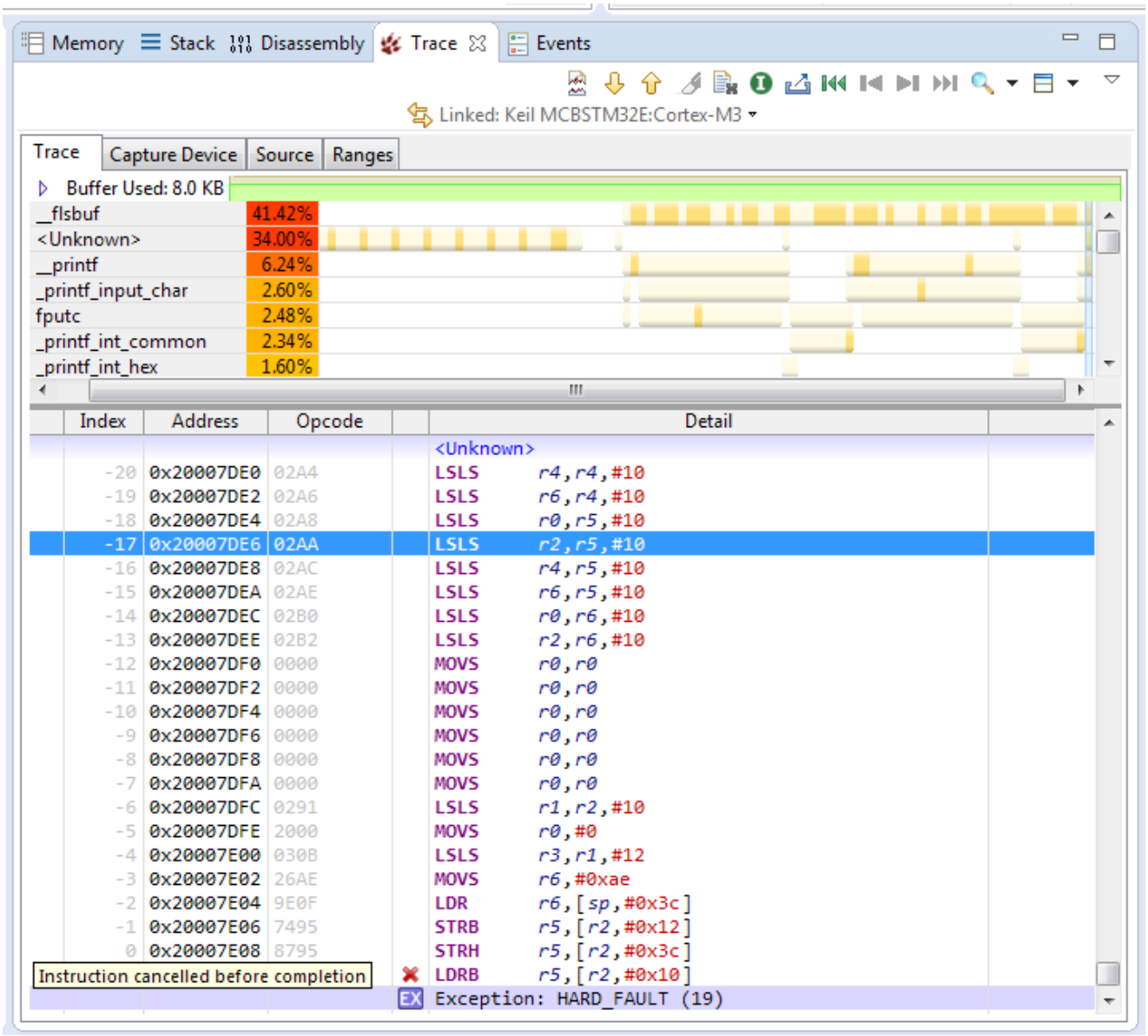


Figure 6-49 Trace view for Cortex-M3 Thumb instructions

Related tasks

4.6 Capturing trace data using the command-line debugger on page 4-95

Related references

Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.29 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.

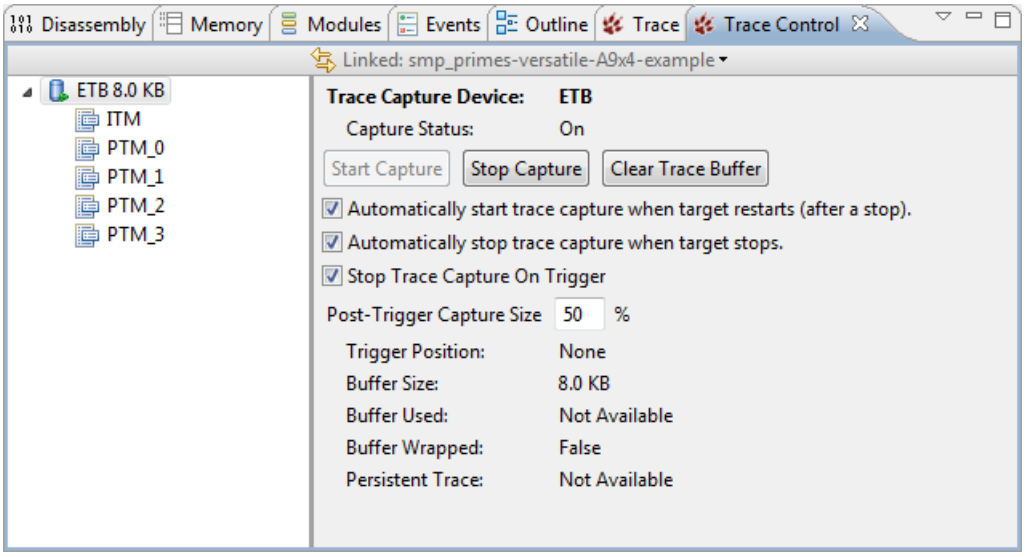


Figure 6-50 Trace Control view

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.

Note

This information is only available for hardware targets.

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:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

New Trace Control View

Select this option to open a new **Trace Control** view.

Refresh

Select this option to refresh the current **Trace Control** view.

DTSL Options…

Select this option to open the Debug and Trace Services Layer (DTSL) **Configuration** dialog box. You can use this dialog box to configure additional debug and trace settings, such as, adding, editing or choosing a DTSL connection configuration.

Trace Dump…

Select this option to open the **Trace Dump** dialog box. In this dialog box, you can configure and export the raw trace data from the buffer and write it to a file.

Related references

Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.30 Variables view

Use the **Variables** view to work with the contents of local, file static, and global variables in your program.

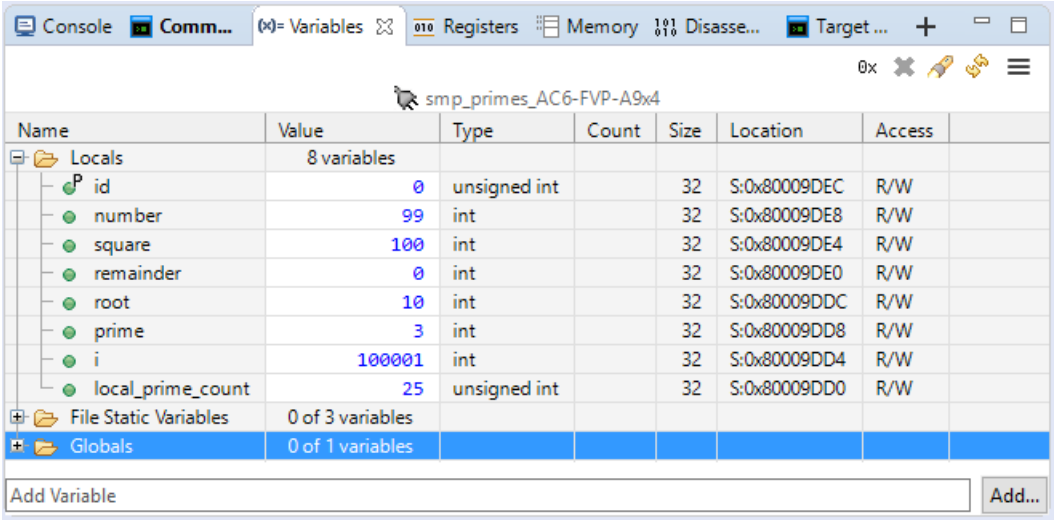


Figure 6-51 Variables view

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 and remove variables from the view. Keep the set of variables in the view to a minimum to maintain good debug performance.

Add a specific variable to the Variables view

If you know the name of the 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 nu to view variables with nu in their name. Double-click the variable to add it to the view.

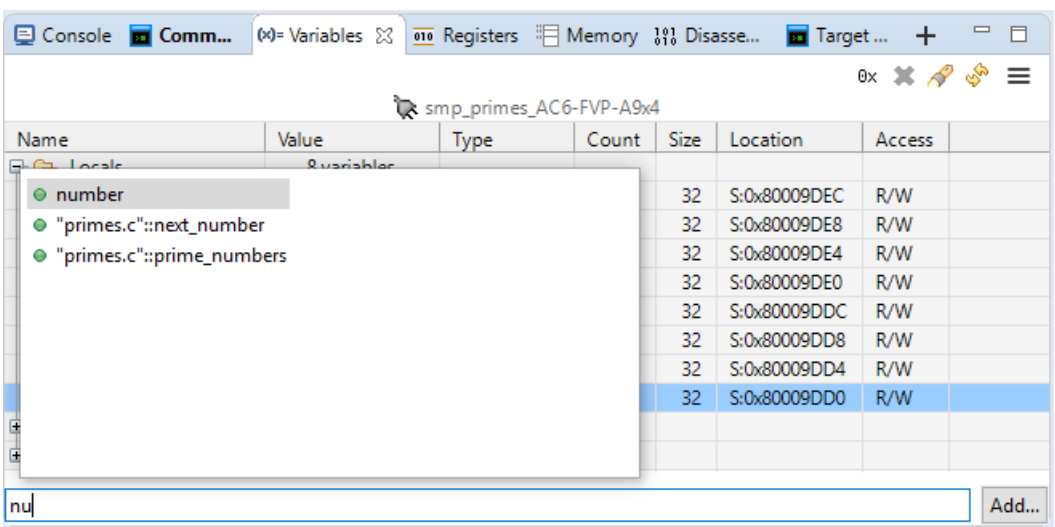


Figure 6-52 How to add variables to the view

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Add one or more variables

If you want to view all the available variables in your code, click **Add** to display the **Add Variable** dialog box. 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.

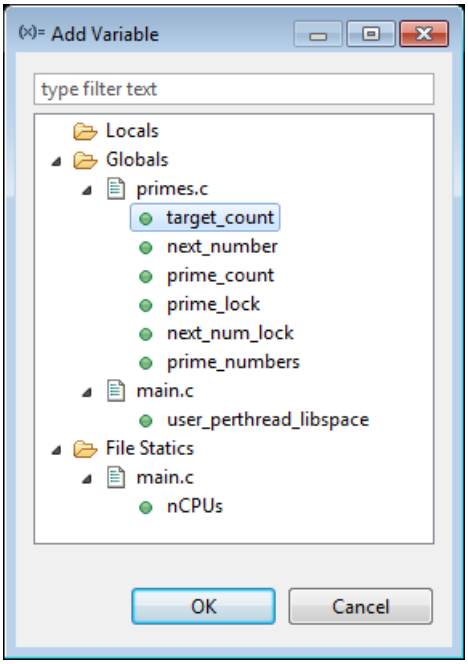


Figure 6-53 Add Global Variables dialog box

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 , or press **Delete** on your keyboard, to remove the selected variables. If you want to reset the view to display the default variables again, then from the view menu, select .

Search for a specific variable

You can use the search feature in the **Variables** view to search for a specific variable in view.

If you know the name of the specific variable, click to display the **Search Variables** dialog box. Either enter the name of the variable you want or select it from the list.

Press **Enter** on your keyboard, or double-click the variable to select and view it in the **Variables** view.

Note

You can also use **CTRL+F** on your keyboard to display the **Search Variables** dialog box.

Refresh view

To refresh or update the values in the view, click .

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Toggle between numerical and hexadecimal values

Click 0x to change all numeric values to hexadecimal values. This works as a toggle and your preference is saved across sessions.

Modify the value of variables

You can modify the values of variables that have 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:`<connection>`

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 is 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 box 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. These formats are **Binary**, **Boolean**, **Hexadecimal**, **Octal**, **Signed Decimal**, and **Unsigned decimal**.

View Menu

The following **ViewMenu** 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. This option prevents automatic updating of the view. You can still use the **Refresh** option to manually refresh the view. You cannot modify the value of a variable if the data is frozen.

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

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 gray 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reset Columns

Resets the columns displayed and their widths to the default.

Related concepts

1.8 About debugging multi-threaded applications on page 1-25
1.9 About debugging shared libraries on page 1-26
1.10 About debugging TrustZone enabled targets on page 1-28

Related tasks

2.9 Assigning conditions to an existing breakpoint on page 2-55

Related references

2.13 Setting a tracepoint on page 2-61
2.8 Conditional breakpoints on page 2-54
2.12 Pending breakpoints and watchpoints on page 2-59
Chapter 6 Perspectives and Views on page 6-133

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.31 Timed Auto-Refresh Properties dialog box

Use the **Timed Auto-RefreshProperties** 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.

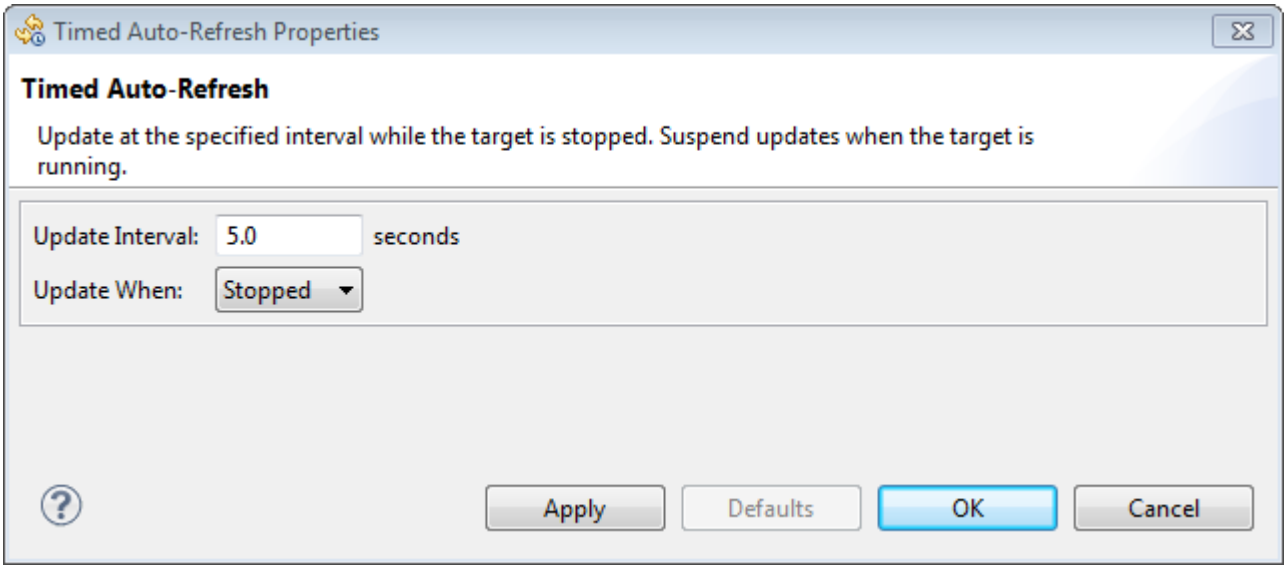


Figure 6-54 Timed Auto-Refresh properties dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.32 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.

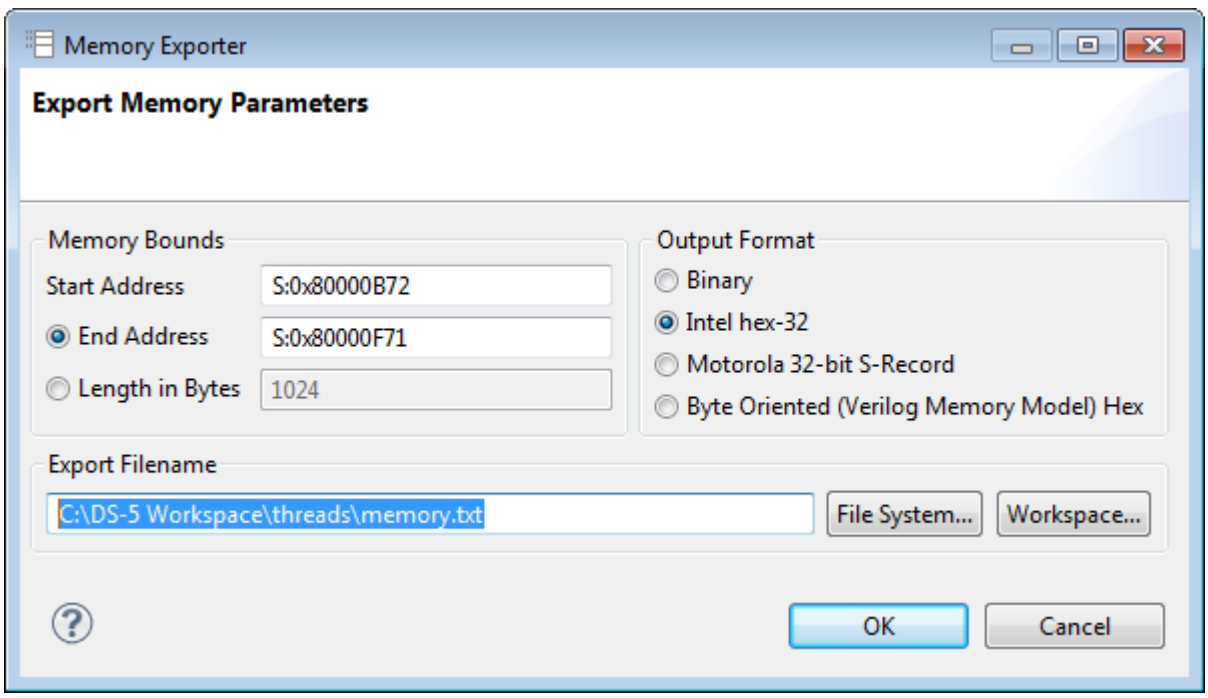


Figure 6-55 Memory Exporter dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.33 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.

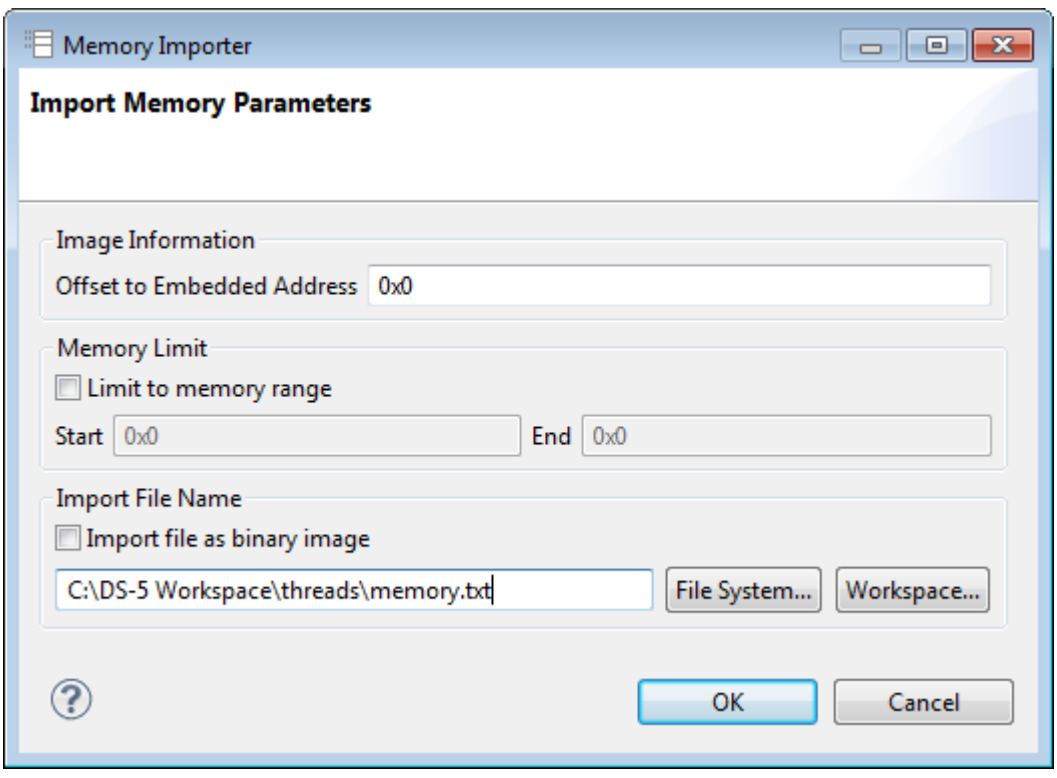


Figure 6-56 Memory Importer dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.34 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.

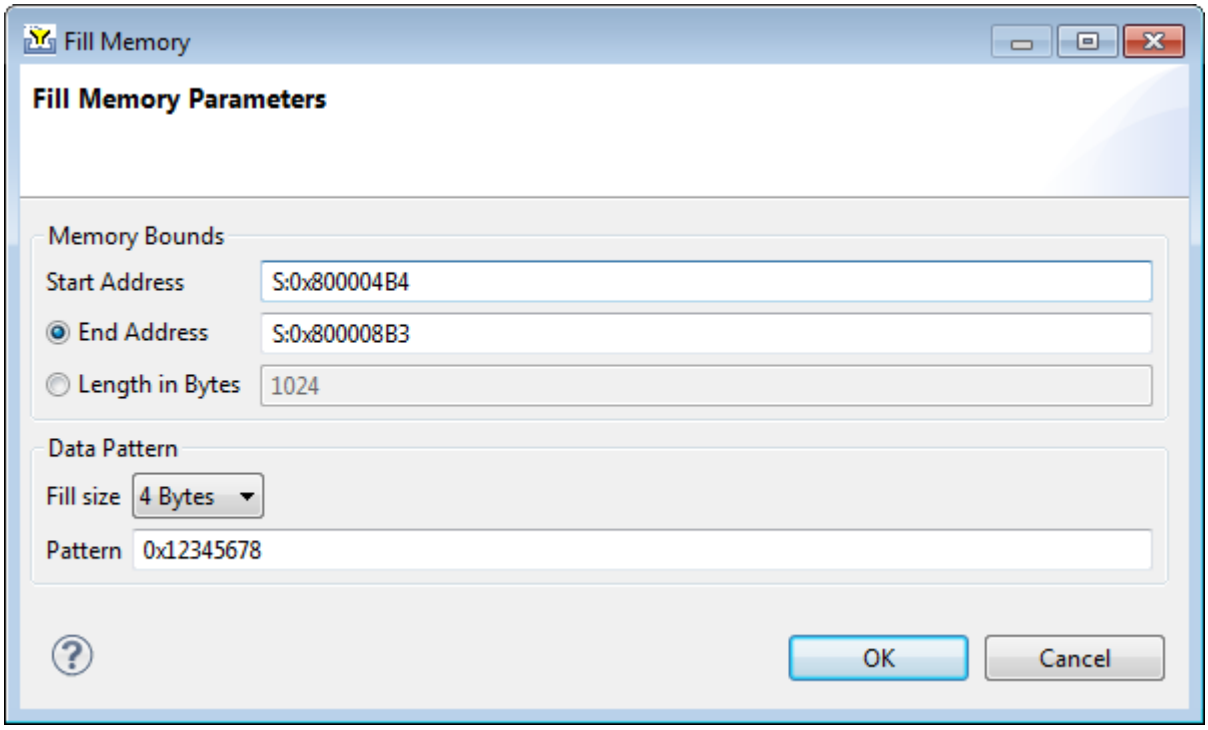


Figure 6-57 Fill Memory dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.35 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Uncheck All

Enables you to unselect all the trace filters.

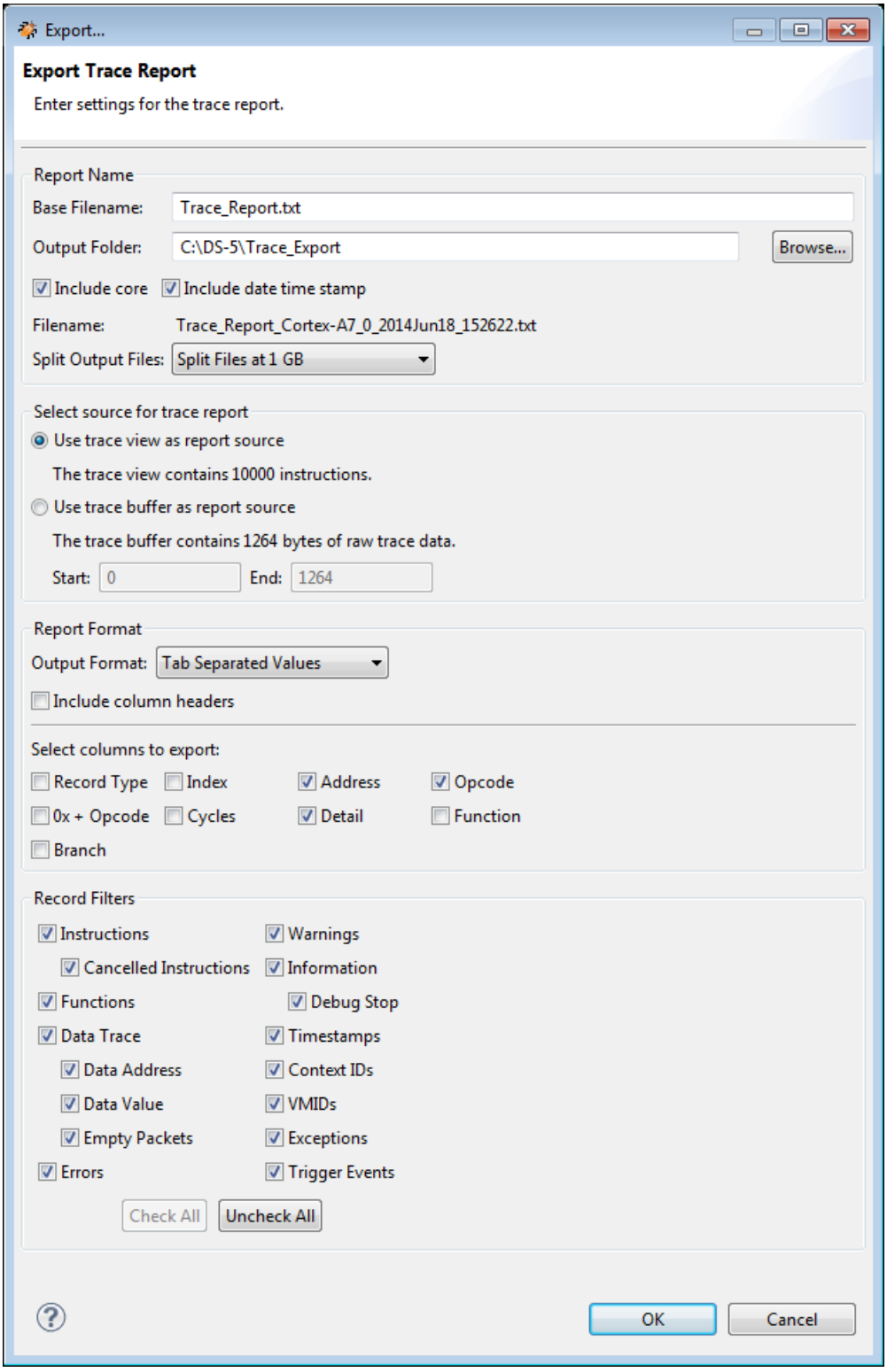


Figure 6-58 Export Trace Report dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.36 Trace Dump dialog box

Use the **Trace Dump** dialog box to generate an output file containing the encoded trace data from the buffer. You can also specify for metadata files to be created, which describe the trace sources that produced that data.

Select Trace Capture Device

Select the trace capture device which should have its data dumped.

Select Trace Source

Select the trace source which should have its data dumped. You can select multiple sources.

Raw Dump for Devices

If selected, dump the data retrieved from RDDI, including any device-specific formatting.

If unselected, provides the target data captured in a format suitable for export. Typically, this is 16-byte CoreSight frames with full frame syncs removed.

Dump Trace and Metadata

Dump both the trace data and the metadata.

Dump Trace Only

Dump the trace data only.

Dump Metadata Only

Dump the metadata only.

Base Directory

Full path to the base directory to write the output to.

Dump Directory

New directory to create within the **Base Directory**. The dump output is printed within this directory.

Split Output Files

Maximum output file size before the output is split and written to multiple files.

Available options:

- Split Files at 1MB.
- Split Files at 32MB (FAT limit).
- Split Files at 512MB.
- Split Files at 1GB (default).
- Split Files at 2GB (FAT16 limit).
- Split Files at 4GB (FAT32 limit).

Help

Open the **Show Contextual Help** dialog box.

OK

Accept the current configure options and create the **Trace Dump** file.

Cancel

Exit the **Trace Dump** dialog box.

Reset defaults

Reset all configuration options to their default values.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

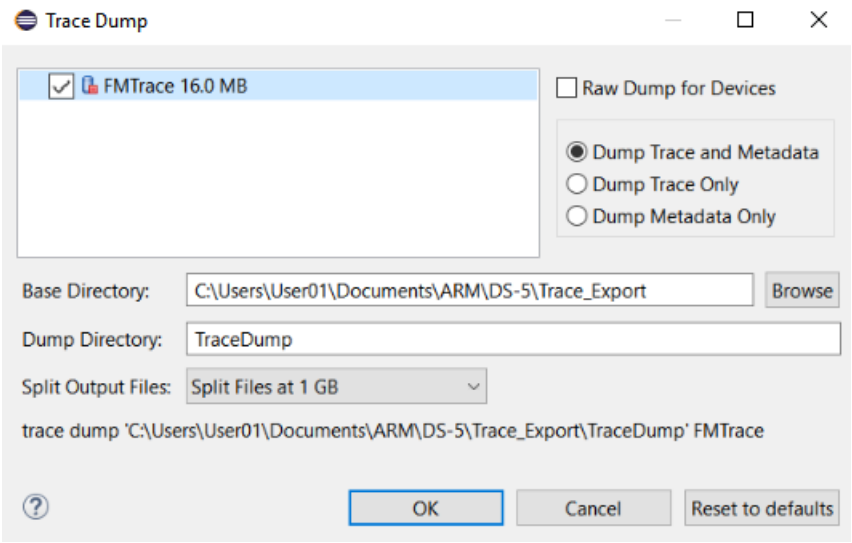


Figure 6-59 Trace Dump dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.37 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.

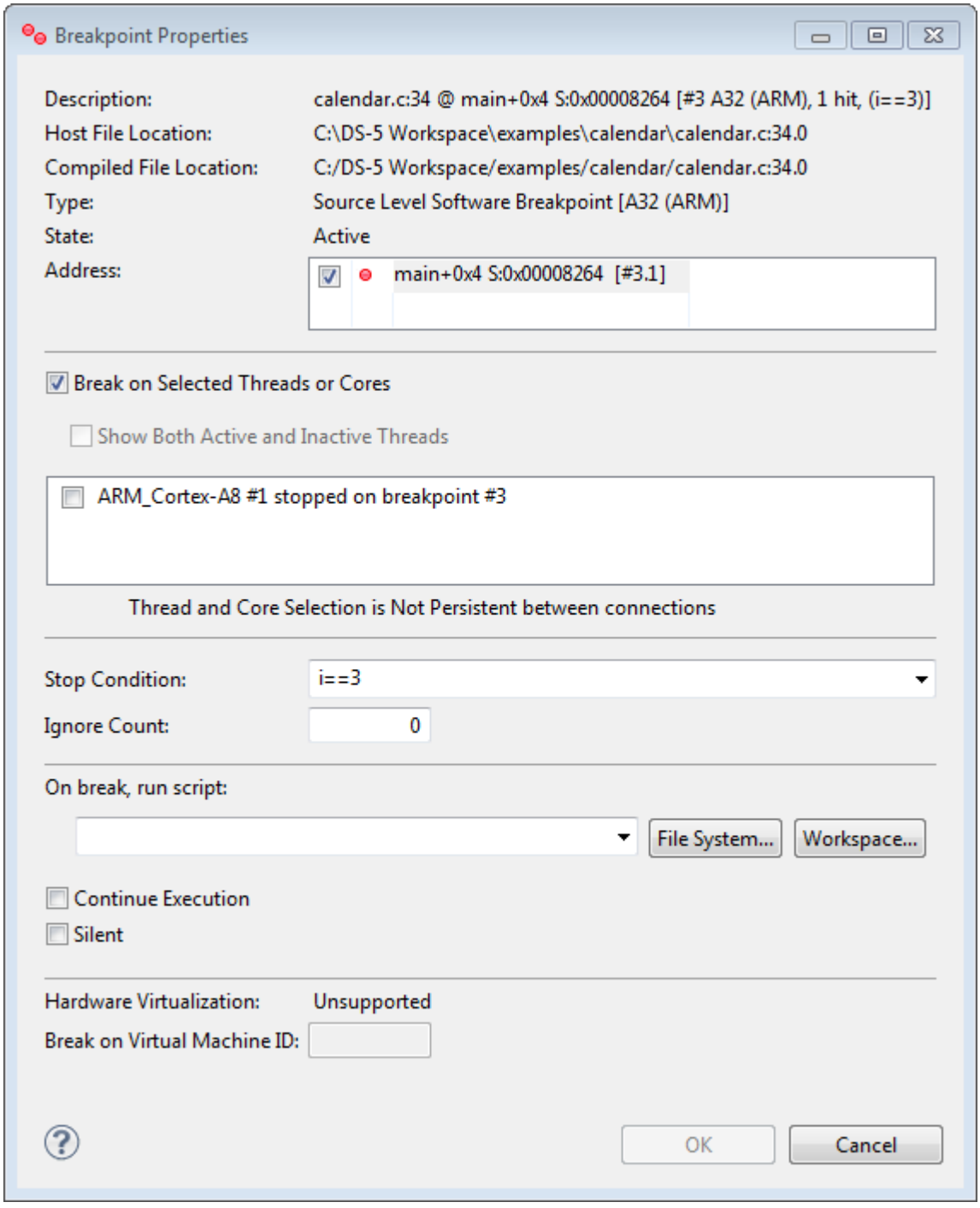


Figure 6-60 Breakpoint properties dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Breakpoint information

The breakpoint information shows the basic properties of a breakpoint. It comprises:

Description

This shows:

- 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 are 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 VirtualMachine ID (VMID) of the guest operating system to which the breakpoint applies.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.38 Watchpoint Properties dialog box

Use the **Watchpoint Properties** dialog box to display the properties of a watchpoint.

You can:

- View the **Location** at which the watchpoint is set.
- View the memory **Address** at which the watchpoint is set.
- View the **Access Type** of the watchpoint.
- Enable or disable the watchpoint.
- Set the **Data Width**.
- Specify a **Stop Condition**.

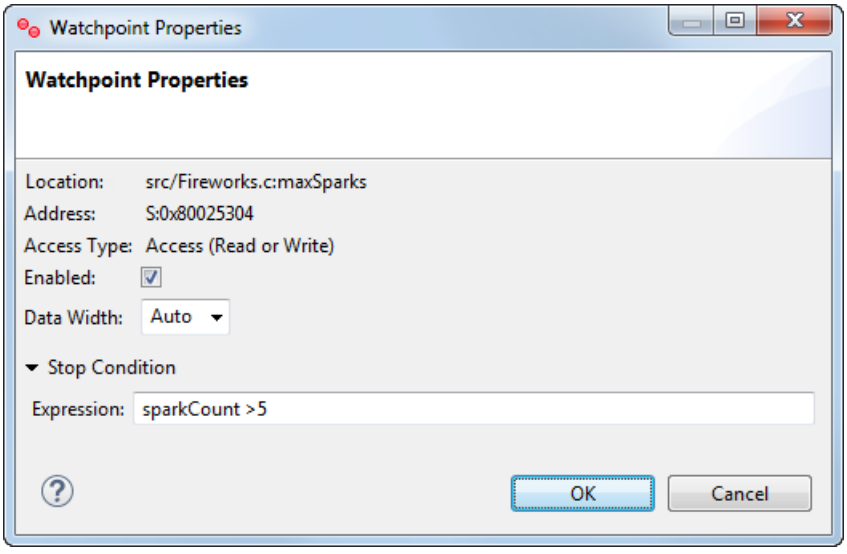


Figure 6-61 Watchpoint Properties dialog box

Location

The data location at which the watchpoint is set.

Address

The memory address at which the watchpoint is set.

Access Type

The type of access specified for the watchpoint.

Enabled

Select to enable watchpoint, deselect to disable watchpoint.

Data Width

Specify the width to watch at the given address, in bits. Accepted values are: 8, 16, 32, and 64 if supported by the target. This parameter is optional.

The width defaults to:

- 32 bits for an address.
- The width corresponding to the type of the symbol or expression, if entered.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Stop Condition

Specify the condition which must evaluate to true at the time the watchpoint is triggered for the target to stop. Enter a C-style expression. For example, if your application code has a variable *x*, then you can specify: **x == 10**.

Note

You can create several conditional watchpoints, but when a conditional watchpoint is enabled, no other watchpoints (regardless of whether they are conditional) can be enabled.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.39 Tracepoint Properties dialog box

Use the **Tracepoint Properties** dialog box to display the properties of a tracepoint.

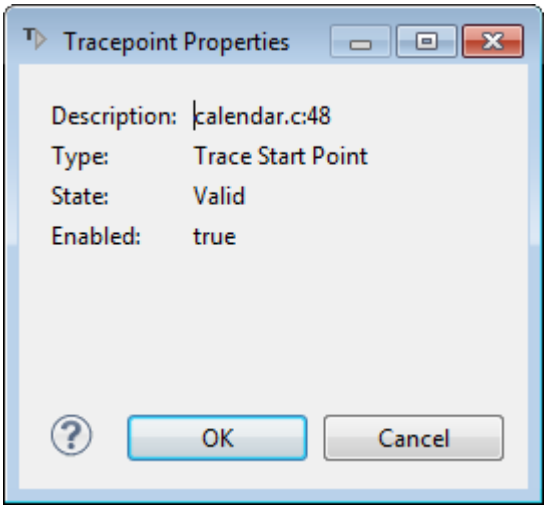


Figure 6-62 Tracepoint Properties dialog box

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.
- The start and stop points for trace must always exist as a pair. Whenever you set a start or stop point, also set its partnering stop or start point.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.40 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.

To view the **Manage Signals** dialog box:

1. Select **Manage Signals** from the **Breakpoints** toolbar or the view menu.
2. Select the individual **Signal** you want to **Stop** or **Print** information, and click **OK**.

View the results in the **Command** view.

Note

You can also use the *infosignals* command to display the current signal handler settings.

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.

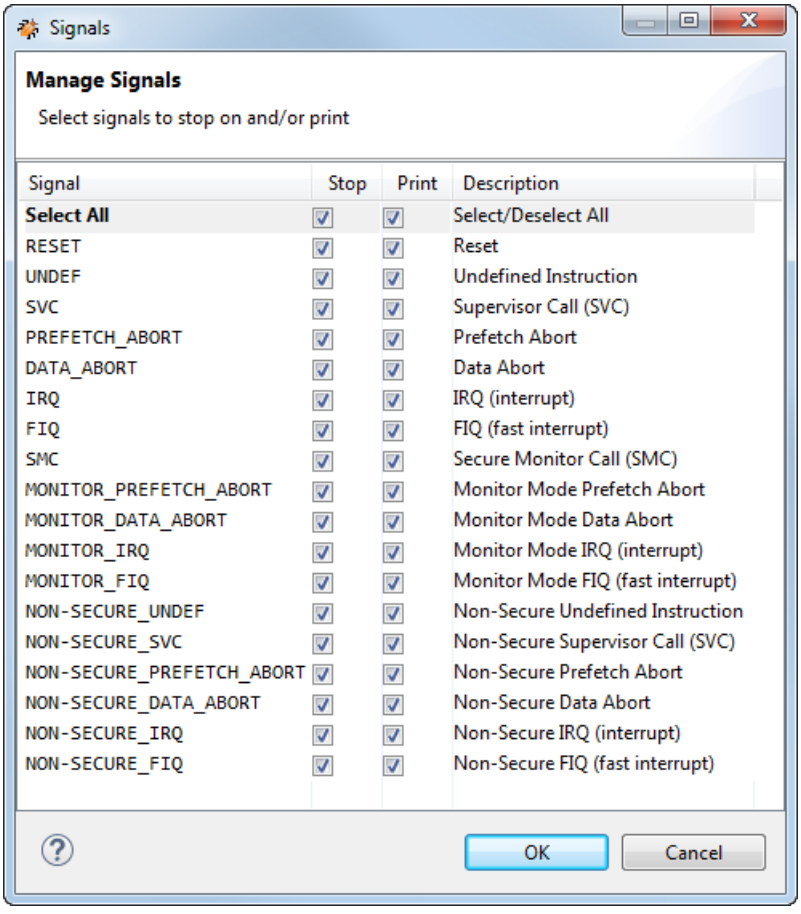


Figure 6-63 Manage Signals dialog box

Related references

2.14 Handling UNIX signals on page 2-62
2.15 Handling processor exceptions on page 2-64
6.4 Breakpoints view on page 6-140

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.41 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.

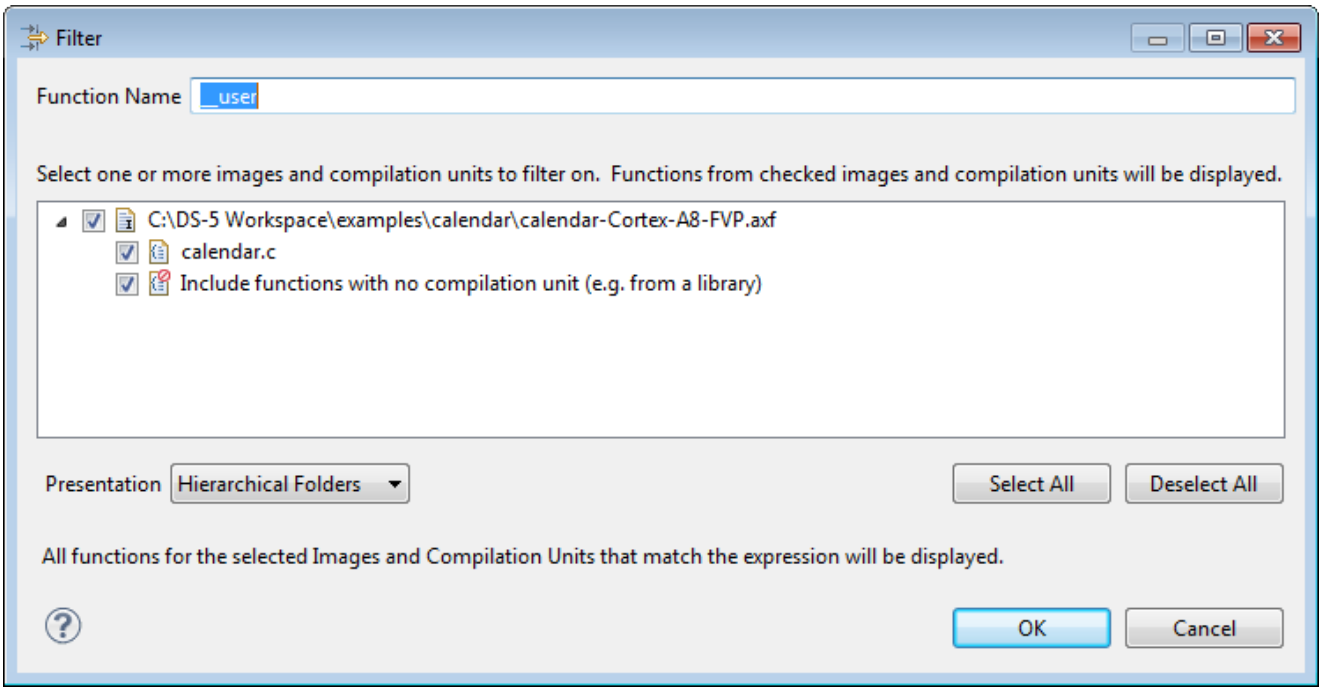


Figure 6-64 Function filter dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.42 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.

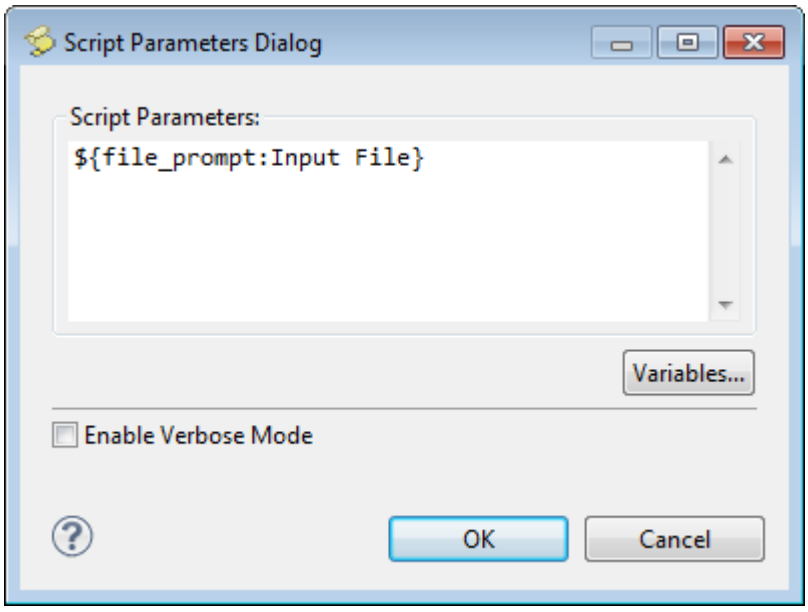


Figure 6-65 Script Parameters dialog box

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.43 Debug Configurations - Connection tab

Use the **Connection** tab in the **Debug Configurations** dialog box to configure Arm Debugger target connections. Each configuration that you create is associated with a single target processor or a cluster of architecturally similar processors (For example, a Symmetric Multi-Processing (SMP) configuration).

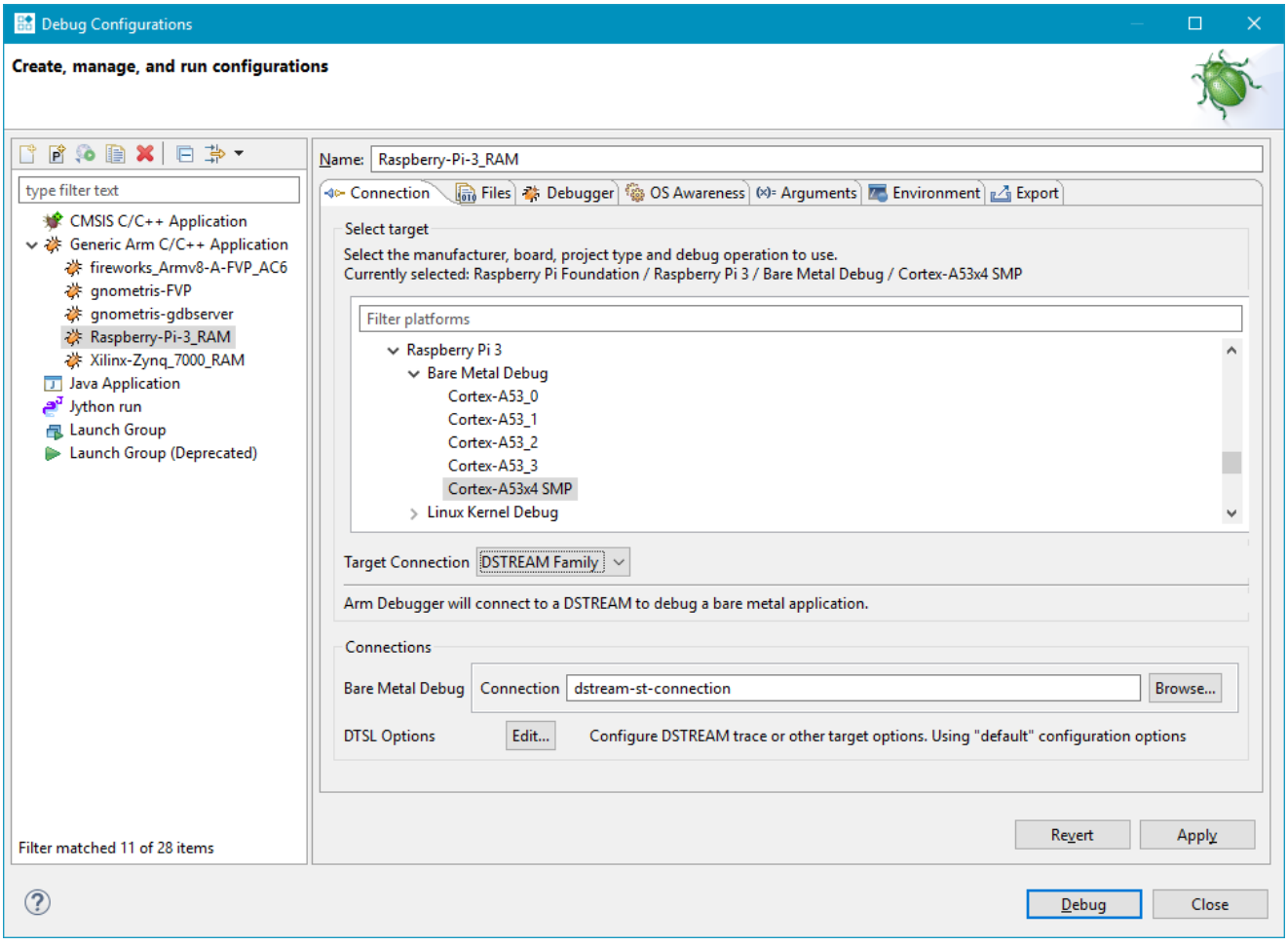


Figure 6-66 Connection tab

If the development platform has multiple processors that are architecturally different (For example, a Cortex-A and Cortex-M), then you must create a separate configuration for each processor.

Note

- Options in the **Connection** tab are dependent on the type of platform that you select.
- When connecting to multiple targets, you cannot perform synchronization or cross-triggering operations.

Select target

These options enable you to select the target manufacturer, board, project type, and debug operation.

Target Connection

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 from the list.

Note

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**.

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.

  Note

  Only available when the selected target is **Connect to already running application**.

- 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** option 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

Select your debug adapter from the list. In **Connection**, specify the host name, IP address, or the fully qualified domain name (FQDN) of your debug hardware adapter. You can also click **Browse…** to display all the available debug hardware adapters on your local subnet or USB connections.

Model parameters

Specify the parameters for launching your model.

The options available depend on the interface of your model. For Component Architecture Debug Interface (CADI) models, you can specify **Model parameters**. For Iris models, you can either select **Launch a new model** and specify the **Model parameters**, or select **Connect to an already running model** and specify the **Connection address** of the model.

See the *Fast Models documentation* for model parameters to use.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DTSL Options

Select **Edit…** to configure additional debug and trace settings.

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.44 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. If required, you can also specify the target file system folder to which files can be transferred.

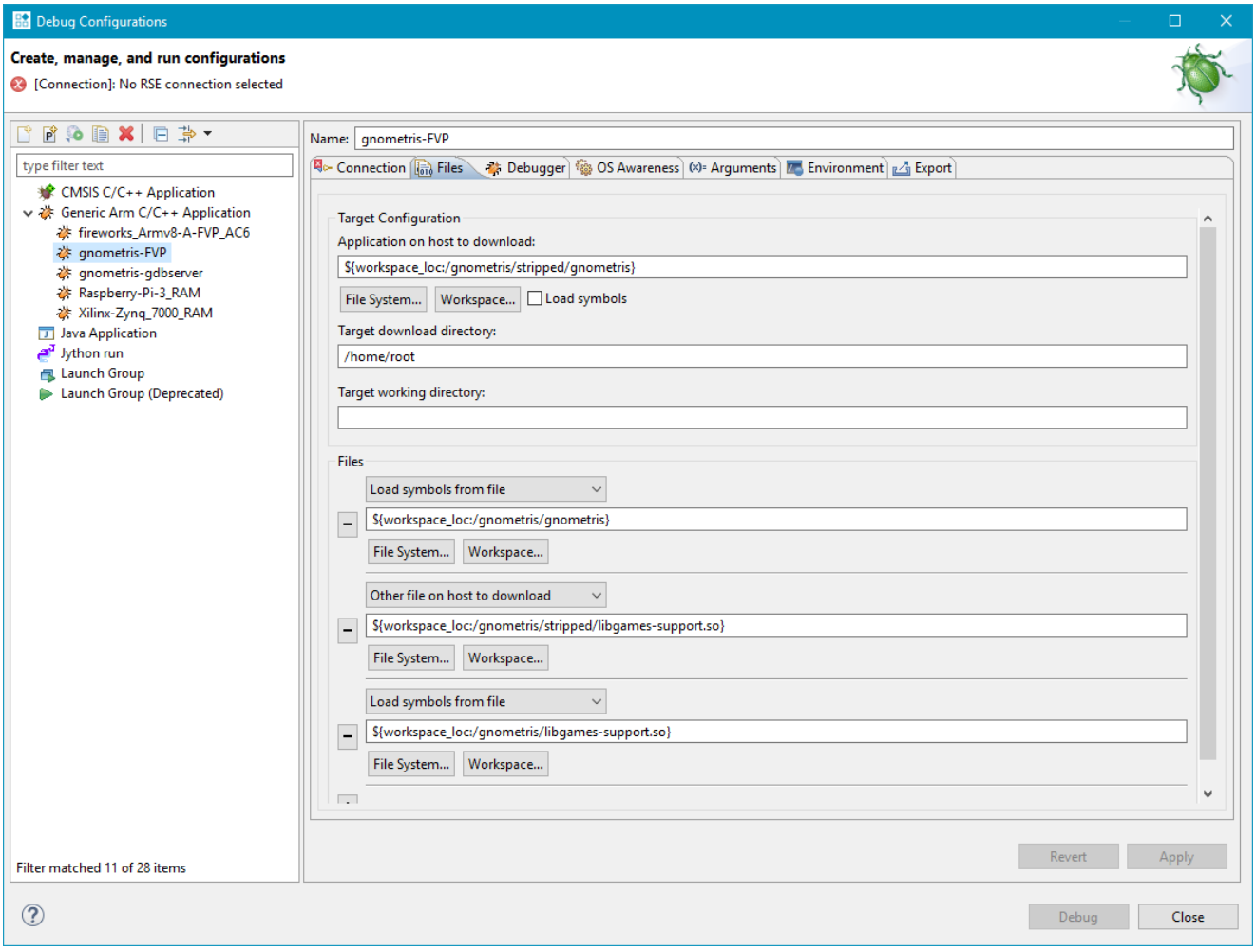


Figure 6-67 Files tab (Shown with file system configuration for an application on a Fixed Virtual Platform)

Note

Options in the **Files** tab depend on the type of platform and debug operation that you select.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 6-2 Files tab options available for each Debug operation

	Download and debug application	Debug target resident application	Connect to already running gdbserver	Debug using DSTREAM	Debug and ETB Trace using DSTREAM
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.

Files tab options summary

The options available on the **Files** tab 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 **File System…** to locate the file in an external directory from the Development Studio workspace.
- Click **Workspace…** to locate the file in a project directory or sub-directory within the Development Studio workspace.

  For example, to download the stripped (no debug) Gnometris application image, select the `gnometris/stripped/gnometris` file.

Select **Load symbols** to load the debug symbols from the specified image.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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) Gnometris 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/gnometris/stripped/gnometris`

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 **File System…** to locate the file in an external directory from the workspace.
- Click **Workspace…** to locate the file in a project directory or sub-directory within the workspace.

For example, to load the debug version of Gnometris you must select the `gnometris` application image that is available in the `gnometris` 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. You can:

- Enter the host location and file name in the field provided.
- Click **File System…** to locate the file in an external directory from the workspace.
- Click **Workspace…** to locate the file in a project directory or sub-directory within the workspace.

For example, to download the stripped (no debug) Gnometris shared library to the target you can select the `gnometris/stripped/libgames-support.so` file.

Path to target system root directory

Specifies the system root directory to search for shared library symbols.

The debugger uses this directory to search for a copy of the debug versions of target shared libraries. The system root on the host workstation must contain an exact representation of the libraries on the target root file system.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.45 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.

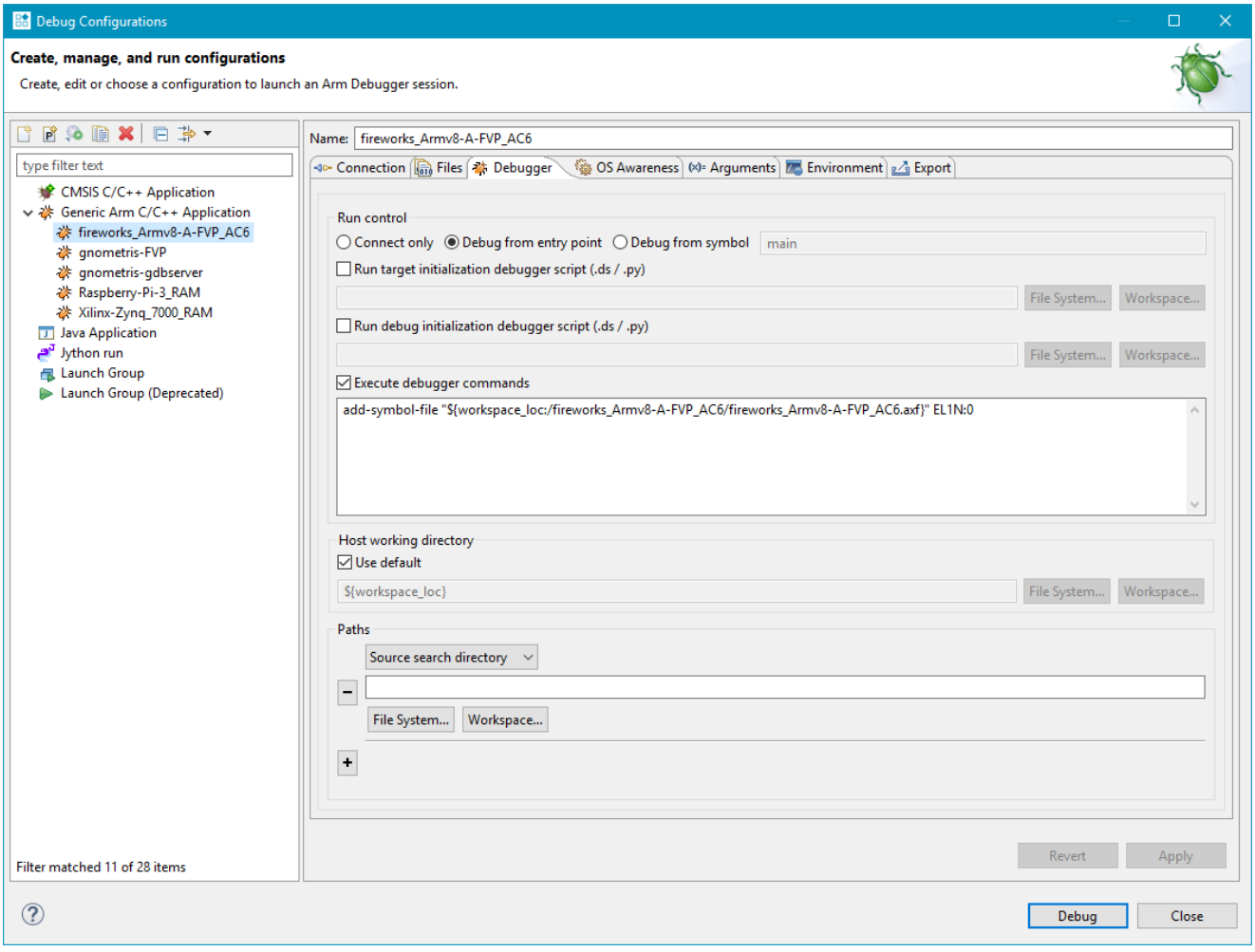


Figure 6-68 Debugger tab (Shown with settings for application starting point and search paths)

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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 **File System…** to locate the file in an external directory from the workspace.
- Click **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 **File System…** to locate the external directory.
- Click **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 **File System…** to locate the directory in an external location from the workspace.
- Click **Workspace…** to locate the directory within the workspace.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shared library search directory

Specify a directory to search for shared libraries:

- 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 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.46 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.

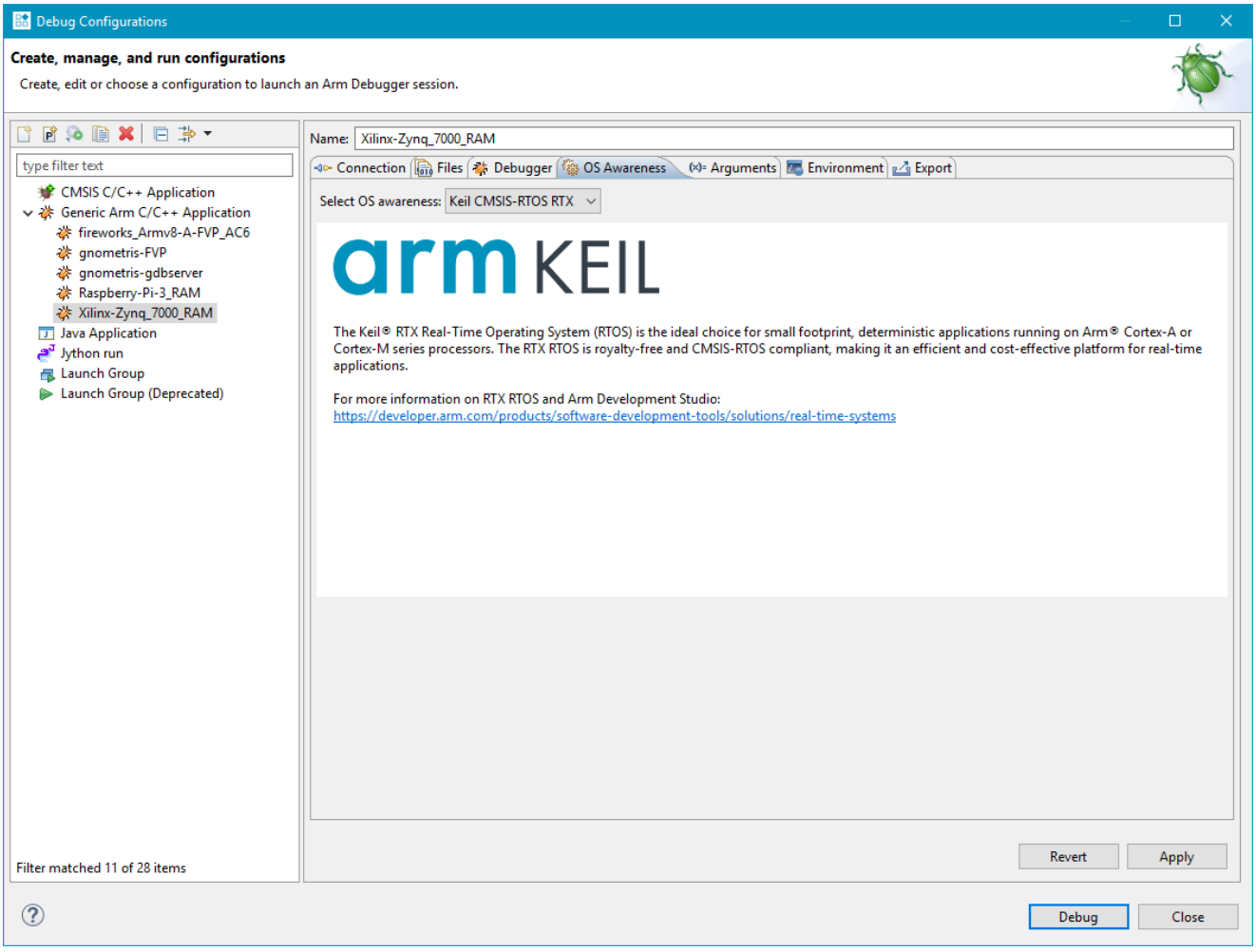


Figure 6-69 OS Awareness tab

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.47 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.

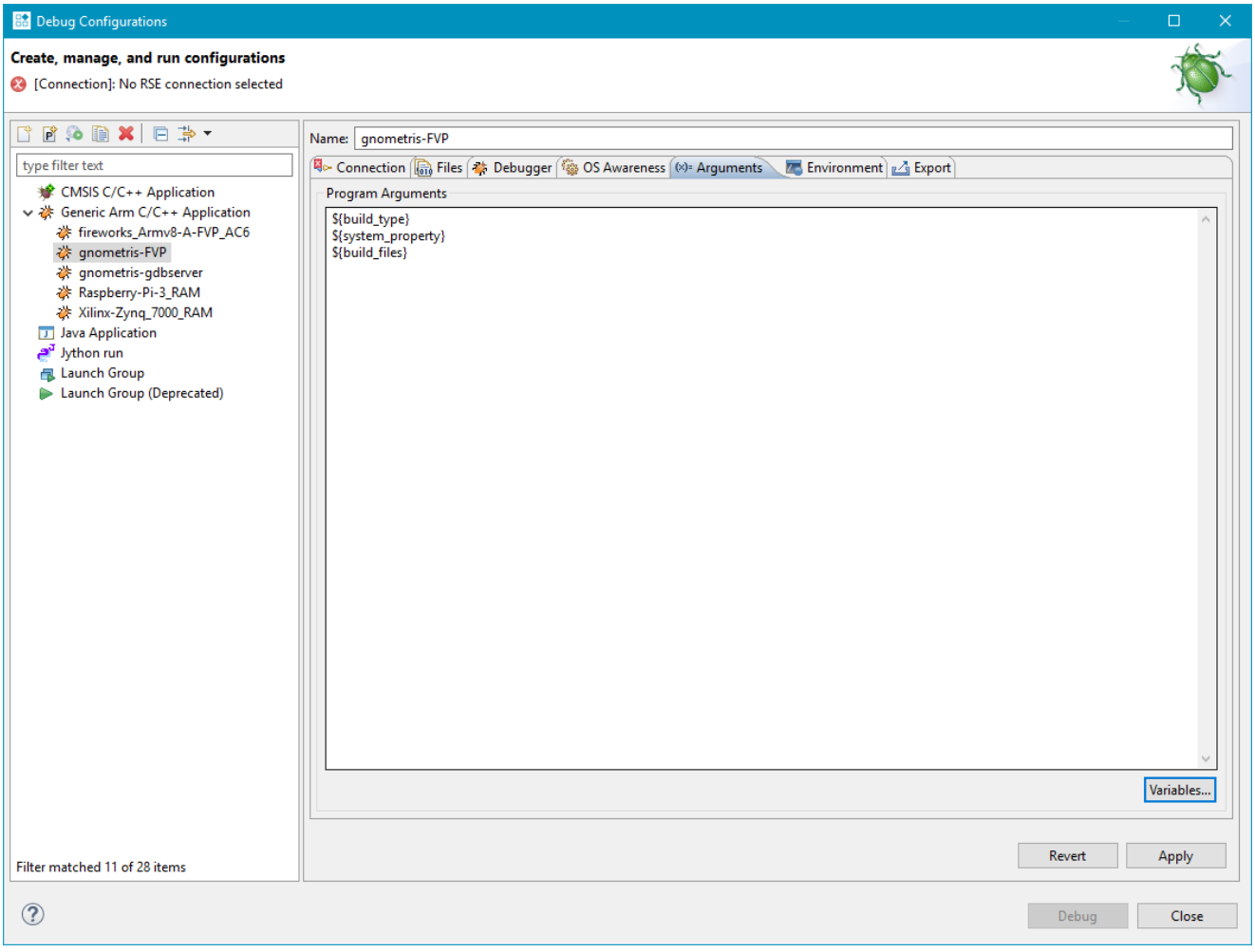


Figure 6-70 Arguments tab

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

Related references

2.16 Using semihosting to access resources on the host computer on page 2-66
2.17 Working with semihosting on page 2-68

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.48 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.

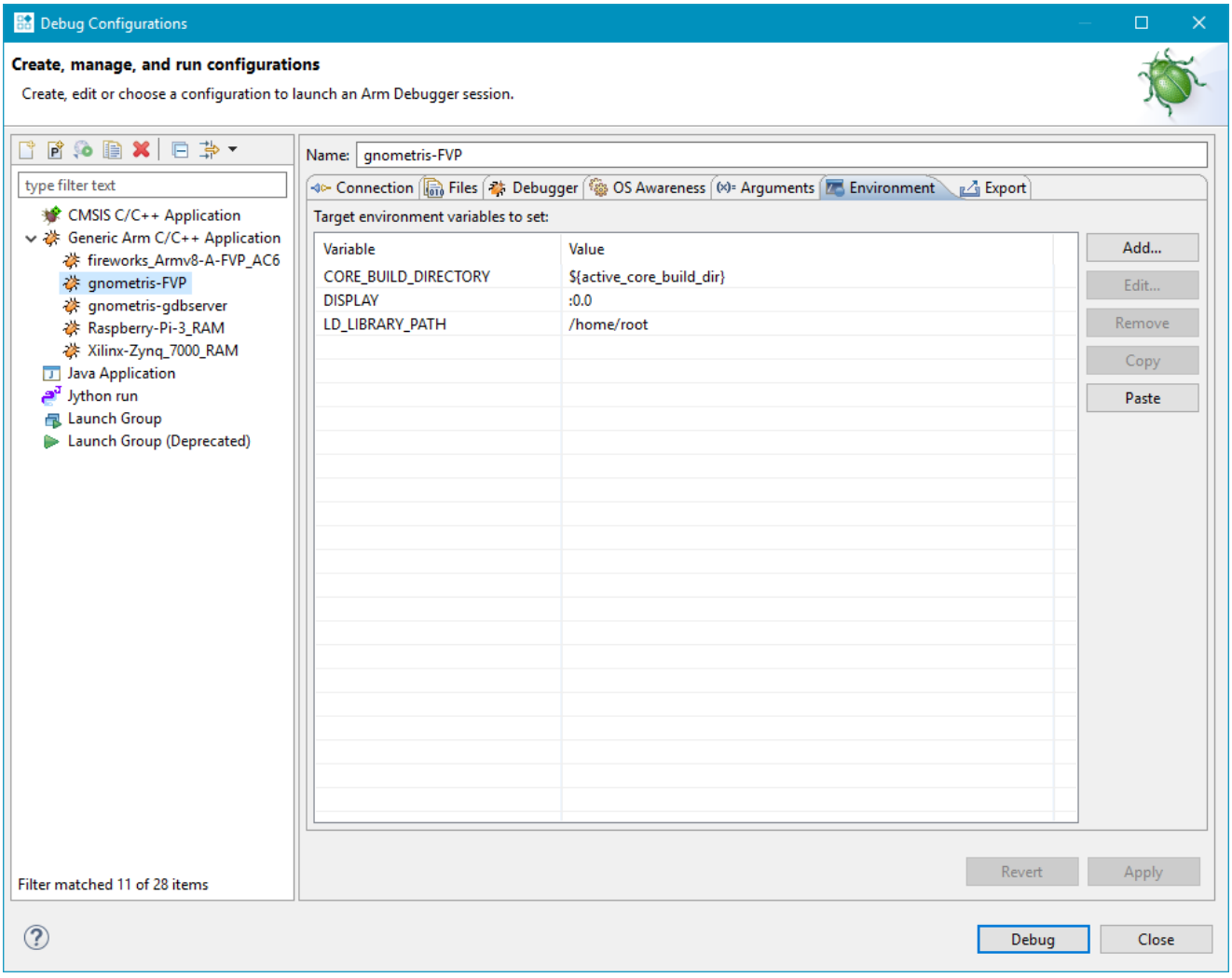


Figure 6-71 Environment tab (Shown with environment variables configured for a Fixed Virtual Platform)

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

New…

Opens the **New Environment Variable** dialog box where you can create a new target environment variable.

For example, the Gnometris application is provided as a packaged example in Arm Development Studio. To debug the Gnometris application on a model, you must create a target environment variable for the `DISPLAY` setting.

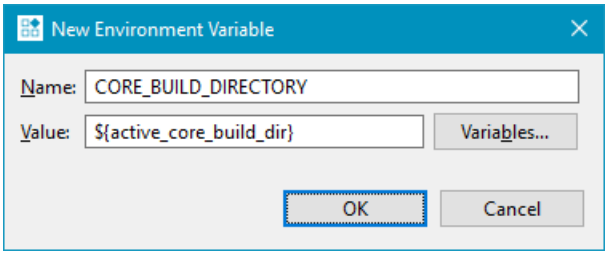


Figure 6-72 New Environment Variable dialog box

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.49 Debug Configurations - Export tab

An Arm Development Studio debug launch configuration typically describes the target to connect to, the communication protocol or probe to use, the application to load on the target, and debug information to load in the debugger. Use the **Export** tab in the **Debug Configurations** dialog box to export the current launch configuration to a format that can be used from the command-line debugger.

On exporting, all Eclipse variables are replaced with their actual values so that the resulting file can be used across multiple machines or workspaces. Path variables are resolved to their absolute path values.

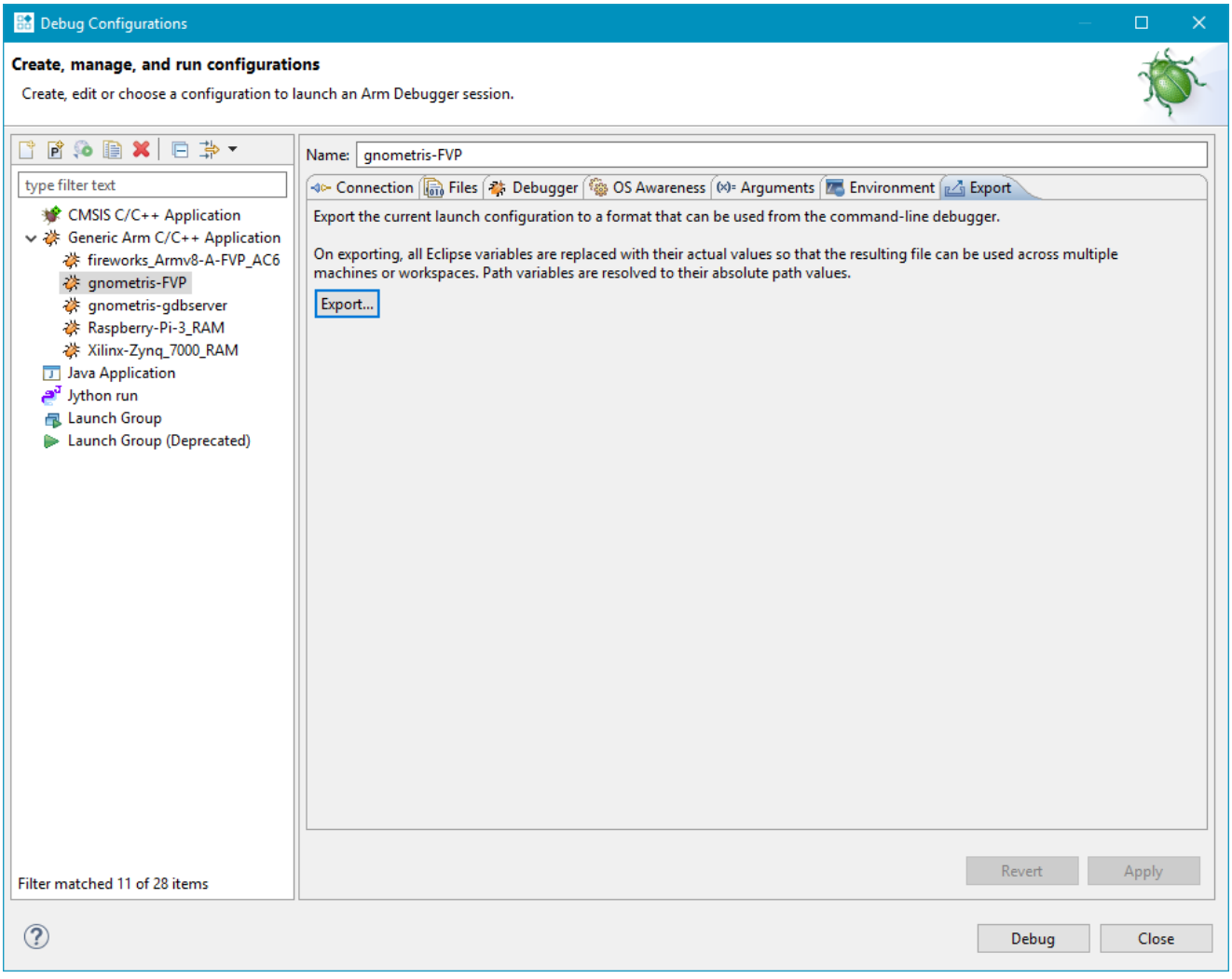


Figure 6-73 Export tab

Export a launch configuration to use at the command-line

1. Create a launch configuration with the required target, application, and image.
2. In the **Export** tab, of the **Debug Configurations** dialog box, click **Export**, and save the `.cli` file.

After exporting the file, use the `--launch-config` *command-line debugger command* on page 4-82 to load the launch configuration from the command-line debugger.

For example, on Windows platforms: `debugger --launch-config "C:\Workspace\debugconfiguration.cli"`

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.50 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

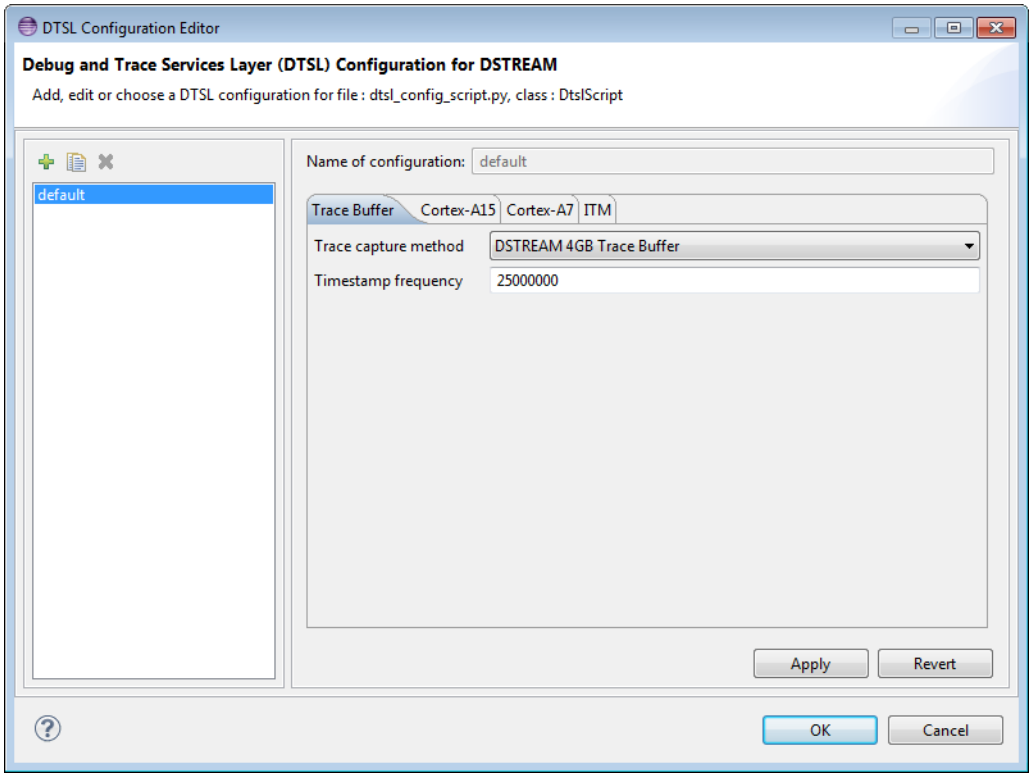


Figure 6-74 Configuration Editor (Shown with Trace capture method set to DSTREAM)

Related concepts

1.13 About debugging caches on page 1-33

Related references

6.23 Cache Data view on page 6-208

Related information

cache commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.51 Probe Configuration dialog box

Edit and save configuration options for your probe using the **Probe Configuration** dialog box.

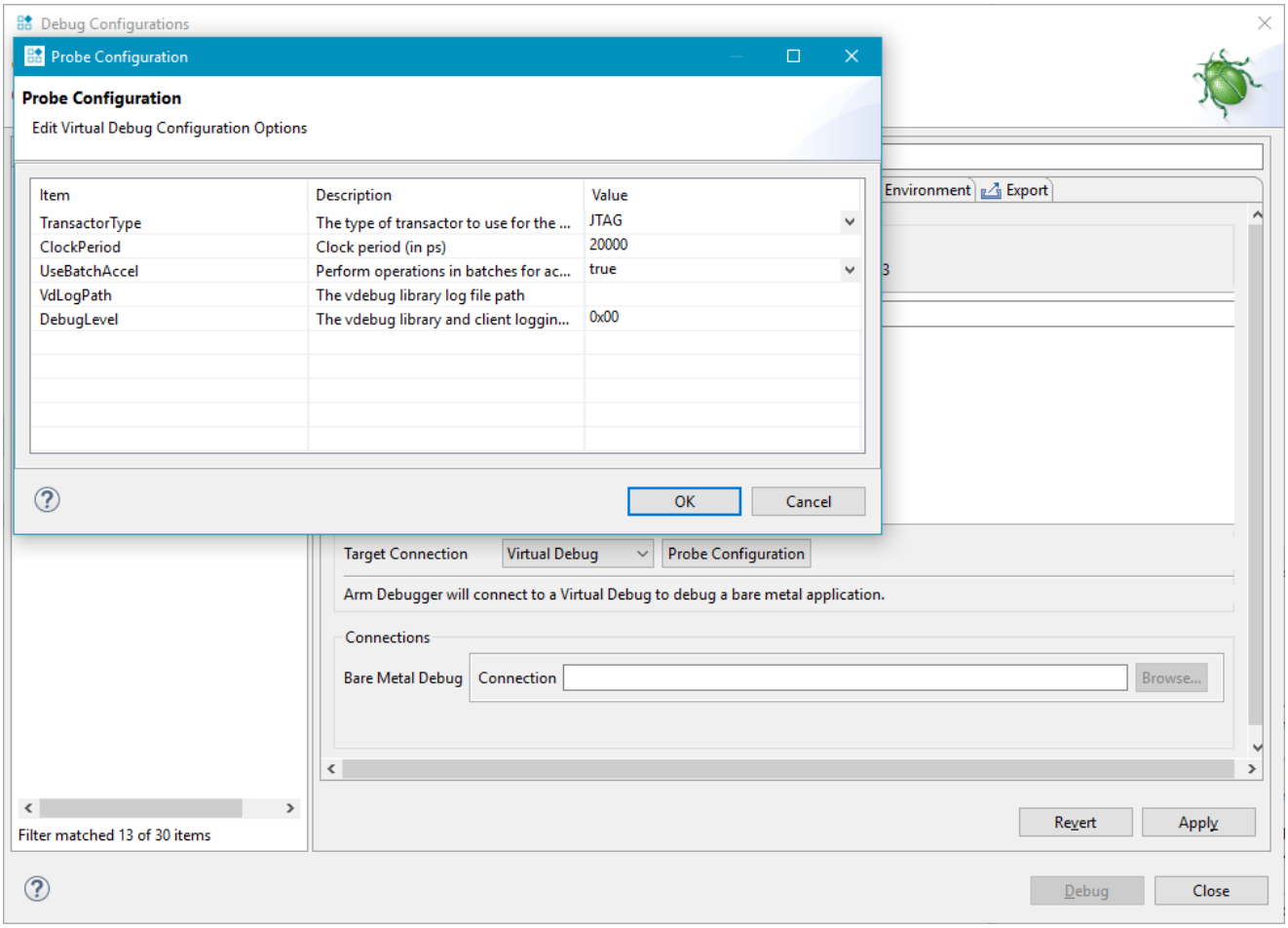


Figure 6-75 Probe Configuration dialog box

Access the Probe Configuration dialog box

To open the **Probe Configuration** dialog box:

1. In the **Debug Configurations** dialog box, select the **Connections** tab.
2. In **Target Connection**, select your probe and click **Probe Configuration**.

Note

Your selected probe must contain user-configurable options for the **Probe Configuration** option to appear.

Features

The options available depend on your probe. Check your hardware documentation for user-configurable options for your probe.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.52 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 you can add to the **Development Studio** perspective are:

- Remote Systems.
- Remote System Details.
- Remote Scratchpad.
- Terminals.

To add an RSE view to the **Development Studio** perspective:

1. Ensure that you are in the **Development Studio** perspective. You can change perspective by using the perspective toolbar or you can select **Window** > **Perspective** > **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**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.53 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.

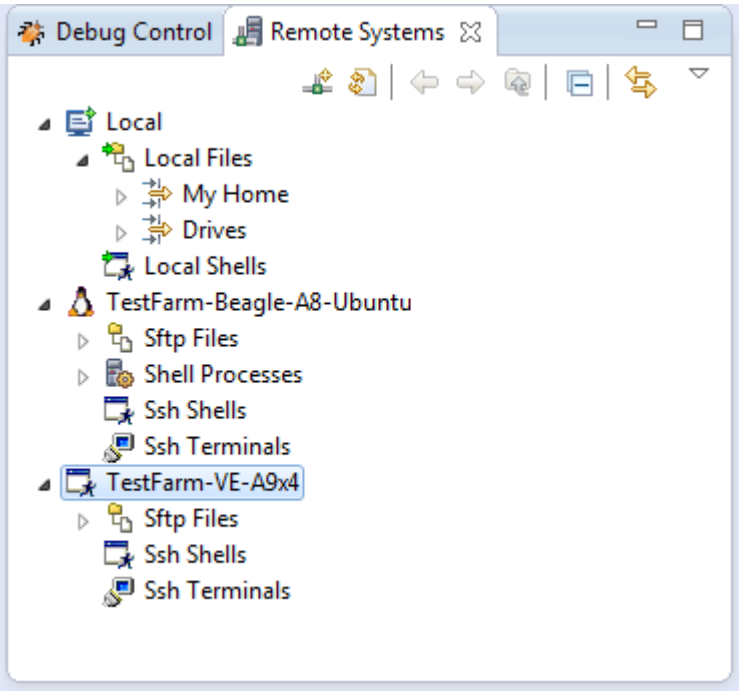


Figure 6-76 Remote Systems view

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.54 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.

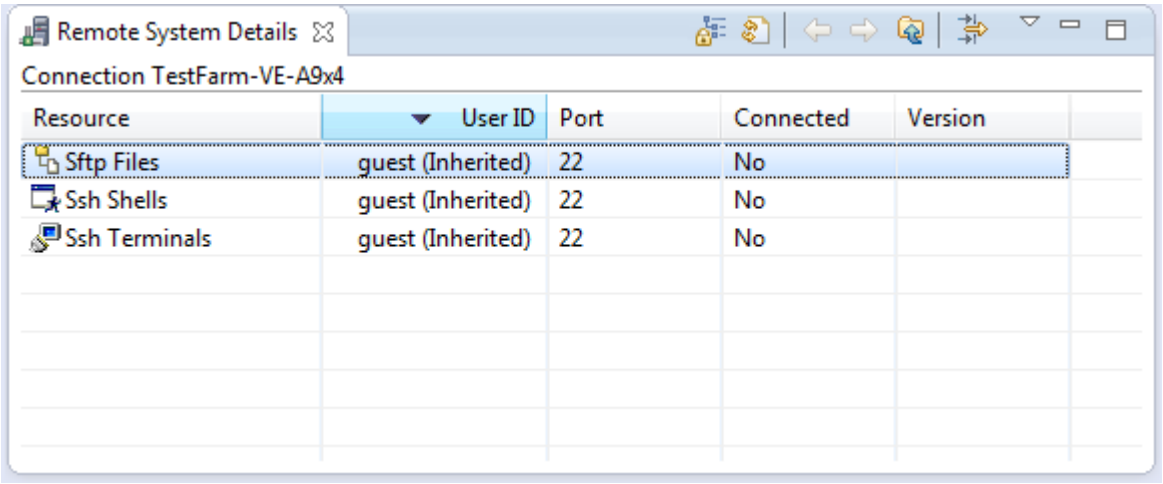


Figure 6-77 Remote System Details view

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**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.55 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.

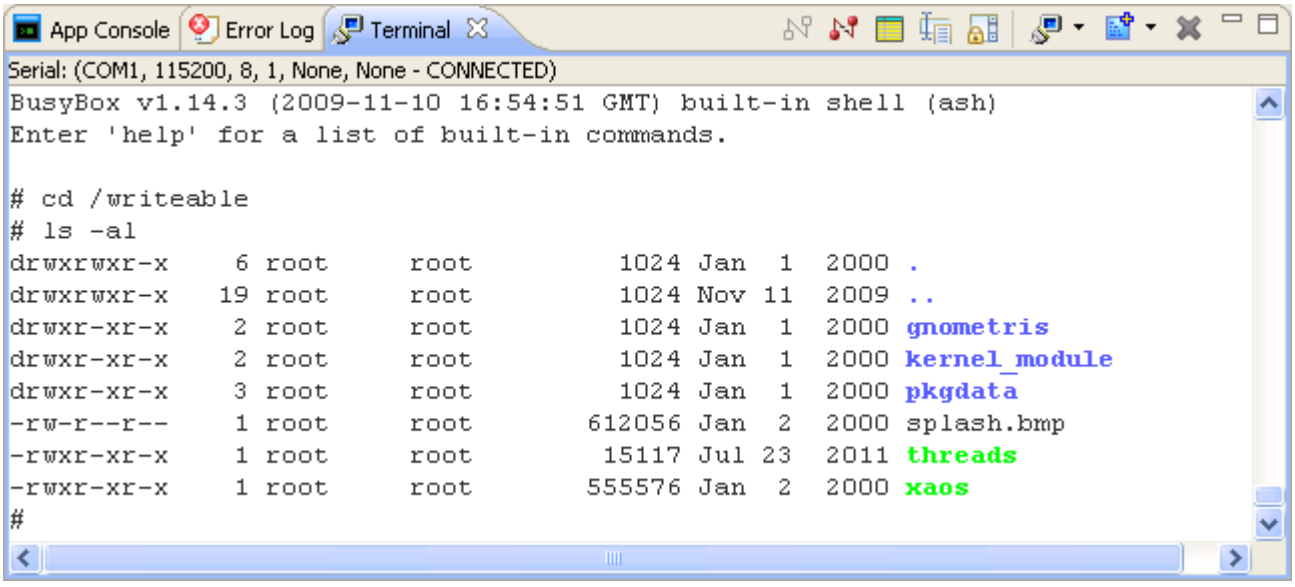


Figure 6-78 Terminal view

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 **Settings**.
5. Select the required connection type.
6. Enter the appropriate information in the **Settings** dialog box.
7. Click **OK**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.56 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.

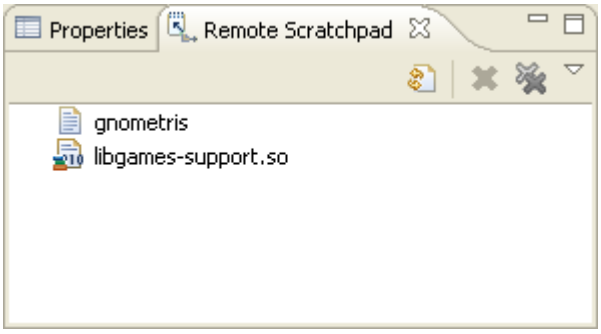


Figure 6-79 Remote Scratchpad

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**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.57 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.

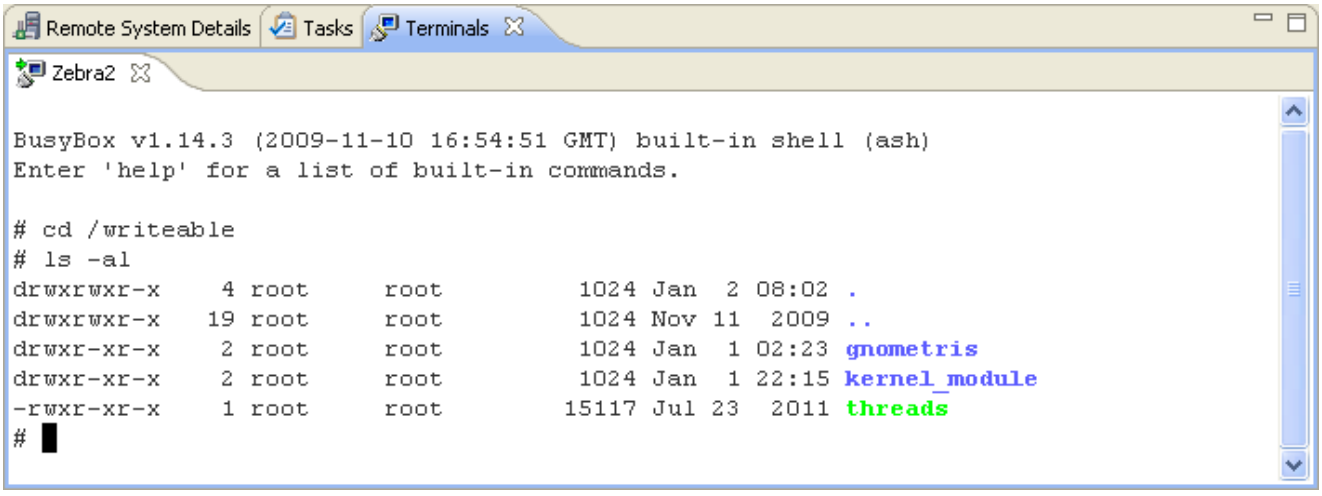


Figure 6-80 Remote Systems terminal

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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.58 Terminal Settings dialog box

Use the **Terminal Settings** dialog box to specify a connection type. You can select Telnet, Secure SHell (SSH), or Serial.

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.

Terminal Settings for Telnet

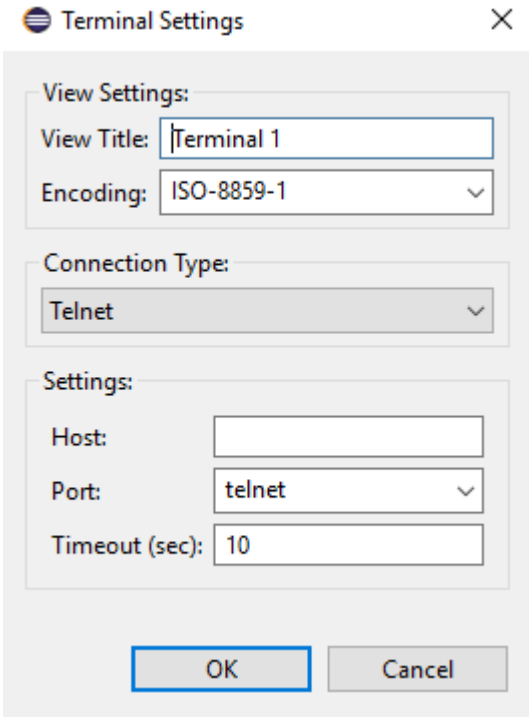


Figure 6-81 Terminal Settings (Telnet) dialog box

Host

The host to connect to.

Port

The port that the target is connected to:

- `telnet`. This is the default.
- `tgtcons`.

Timeout (sec)

The connection's timeout in seconds.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Terminal Settings for SSH

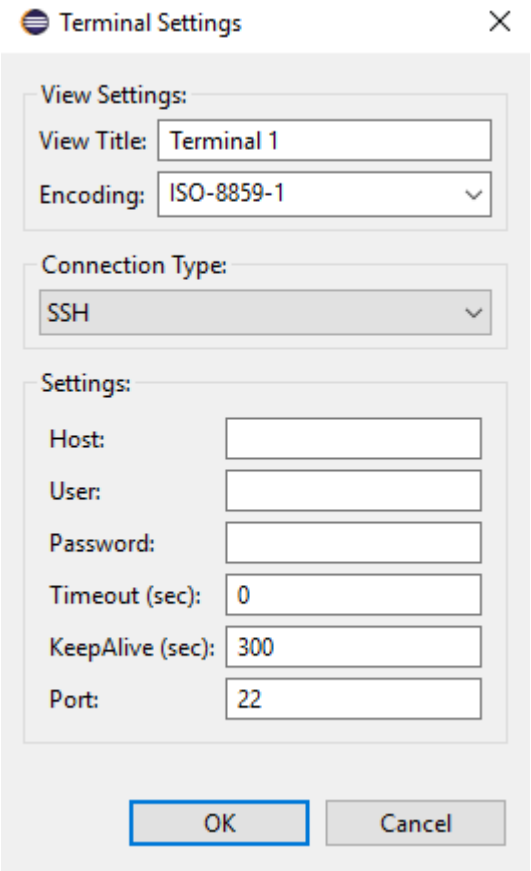


Figure 6-82 Terminal Settings (SSH) dialog box

Host

The host to connect to.

User

The user name.

Password

The password corresponding to the user.

Timeout (sec)

The connection's timeout in seconds. Defaults to 0.

KeepAlive (sec)

The connection's keep alive time in seconds. Defaults to 300.

Port

The port that the target is connected to. Defaults to 22.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Terminal Settings for Serial

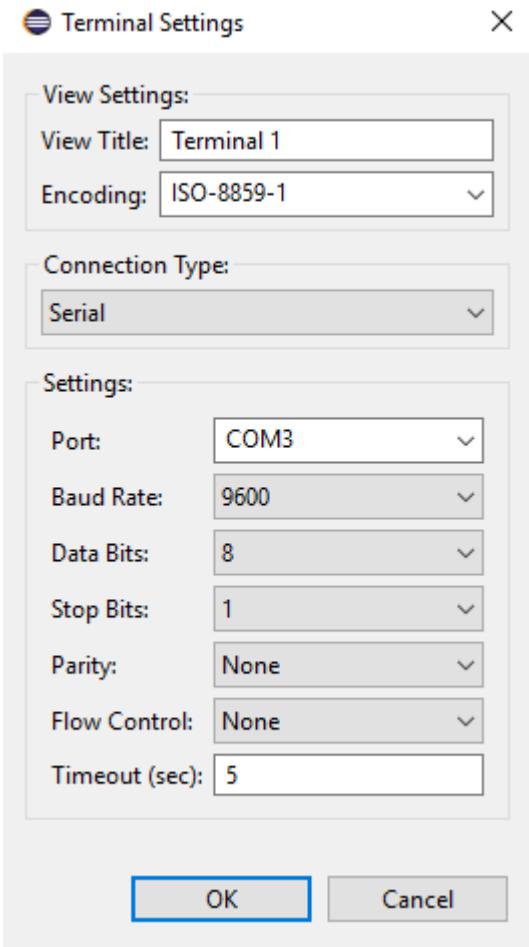


Figure 6-83 Terminal Settings (Serial) dialog box

Port

The port that the target is connected to.

Baud Rate

The connection baud rate.

Data Bits

The number of data bits.

Stop Bits

The number of stop bits for each character.

Parity

The parity type:

- None. This is the default.
- Even.
- Odd.
- Mark.
- Space.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Flow Control

The flow control of the connection:

- None. This is the default.
- RTS/CTS.
- Xon/Xoff.

Timeout (sec)

The connection's timeout in seconds.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.59 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.

The configuration process depends on how the debug hardware unit is connected to the host computer, and if your network uses Dynamic Host Configuration Protocol (DHCP). If your debug hardware unit is connected to an Ethernet network or is directly connected to the host computer using an Ethernet cross-over cable, you must configure the network settings before you can use the unit for debugging.

Note

You have to configure the network settings once for each debug hardware unit.

The following connections are possible:

- Your debug hardware unit is connected to your local network that uses DHCP. For this method, 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. For this method, you must assign a static IP address to the debug hardware unit.

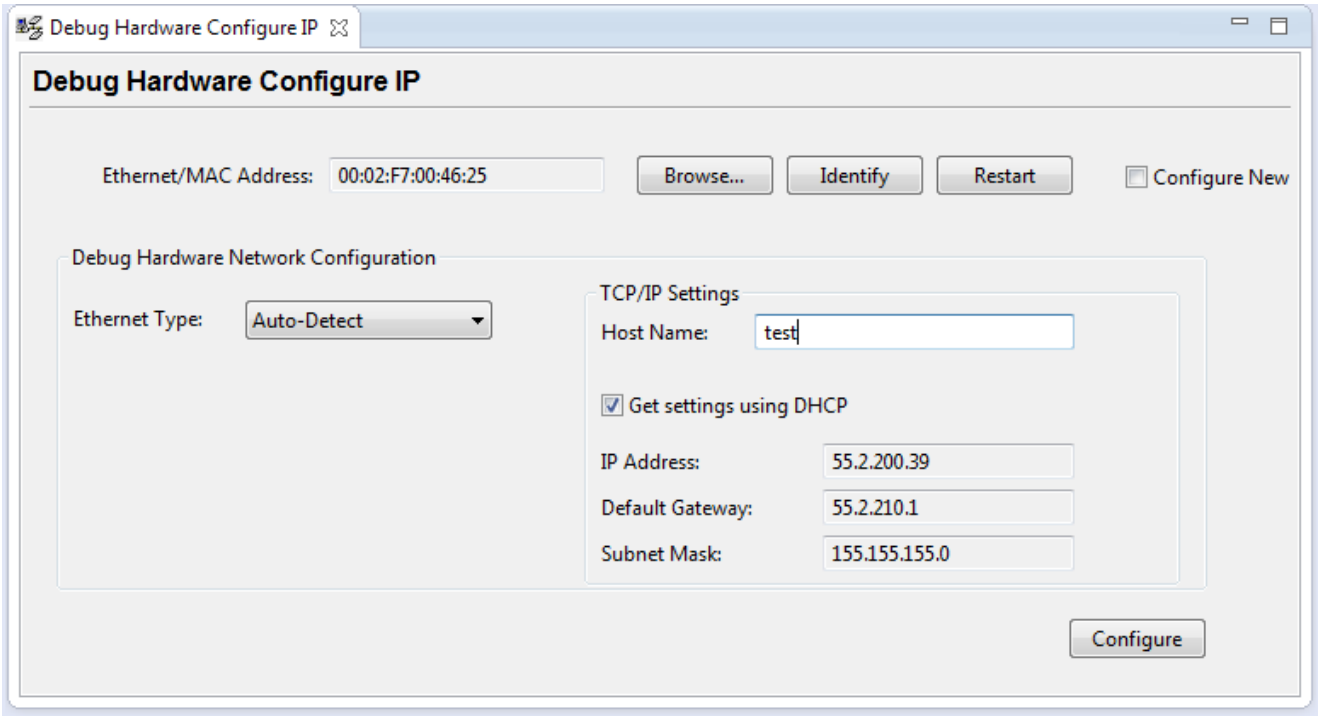


Figure 6-84 Debug Hardware Configure IP view

Access

To access the **Debug Hardware Configure IP** view from the main menu, either:

- Select **Window** > **Show View** > **Debug Hardware Configure IP**.
- Select **Window** > **Show View** > **Other...** > **Arm Debugger** > **Debug Hardware Configure IP**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

Table 6-3 Debug Hardware Configure IP view contents

Field	Description
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…	Displays the **Connection Browser** dialog box. 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 DSTREAM, the DSTREAM logo flashes during identification.
Restart	Restarts 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. For DSTREAM devices, ensure that this is set to **Auto-Detect**.
TCP/IP Settings	The following settings are available: • 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. The name 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 you use 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

6.60 Debug Hardware Firmware Installer view on page 6-282
6.61 Connection Browser dialog box on page 6-285

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.60 Debug Hardware Firmware Installer view

Use the **Debug Hardware Firmware Installer** view to update the firmware for your debug hardware.

Firmware files for Arm debug hardware units typically contain:

Templates for devices supported by the debug hardware unit

Each template defines how to communicate with the device and the settings that you can configure for that device. A firmware update might contain support for newer devices that the debug hardware unit can now support.

Firmware updates and patches

Arm periodically releases updates and patches to the firmware that is installed on a debug hardware unit. These updates or patches might extend the capabilities of your debug hardware, or might fix an issue that has become apparent.

To access the **Debug Hardware Firmware Installer** view, from the main menu, select **Window** > **Show View** > **Other** > **Arm Debugger** > **Debug Hardware Firmware Installer**.

Updating your debug hardware unit firmware using the Debug Hardware Firmware Installer view

1. Connect your debug hardware to the host workstation.
2. From the main menu, select **Window** > **Show View** > **Other** > **Arm Debugger** > **Debug Hardware Firmware Installer** to display the view.
3. In **Select Debug Hardware**, click **Browse** and select your debug hardware unit.
4. Click **Connect** to your debug hardware unit. The current firmware status of your debug hardware is displayed.
5. In **Select Firmware Update File**, click **Browse** and select the firmware file for your debug hardware unit. When the file is selected, the dialog box shows the selected firmware update file details. For example:

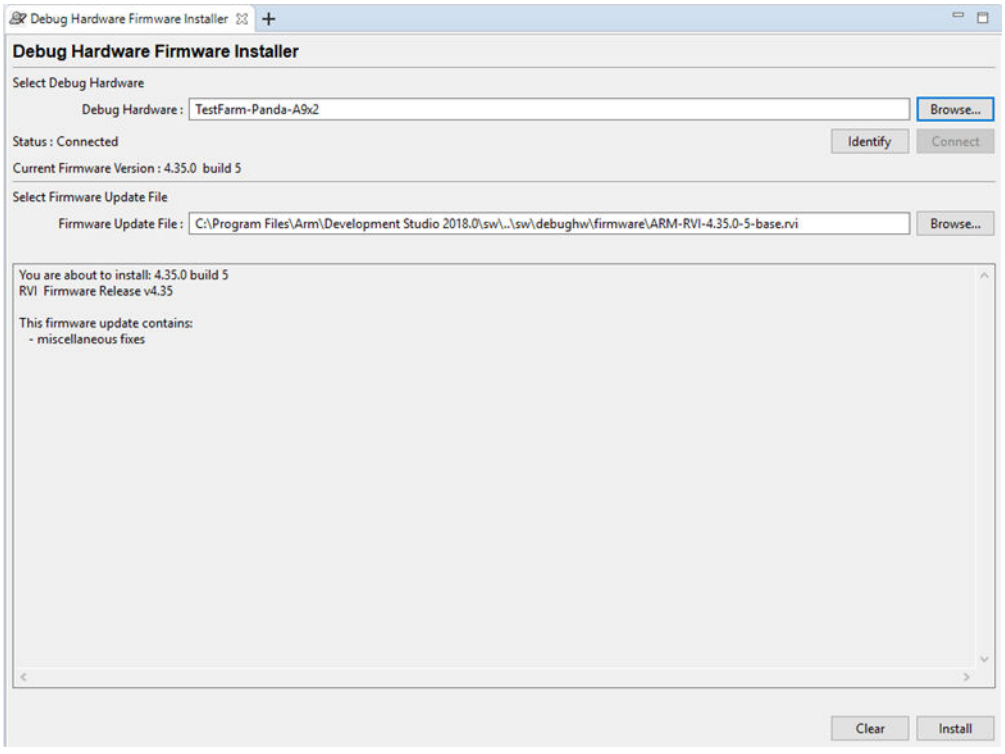


Figure 6-85 Debug Hardware Firmware Installer view

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm Development Studio only displays the firmware file applicable to your debug hardware unit, so it is easy for you to select the correct firmware file.

6. Click **Install**. The firmware update process starts. When complete, a message is displayed indicating the status of the update.

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 box. 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 DSTREAM, the DSTREAM logo flashes during identification.

Connect

Click to connect to your debug hardware. When connected, the dialog box shows the current firmware status.

Select Firmware Update File

Click **Browse** and select the firmware file. Development Studio only displays the firmware applicable to your debug hardware unit. If you want to use a different firmware file, use the **Browse** button and select the firmware file you need. When the file is selected, the dialog box shows the selected firmware update file details.

Note

In Development Studio, the latest firmware files are available at: `<install_directory>\sw\debughw\firmware\`. If you want to choose a different firmware file, click **Browse** and select the new location and 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`.

bld

Is a build number.

type

Is either:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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.

dstream2

For the newer family of DSTREAM debug and trace units.

rvi

For an RVI debug unit.

Related references

6.59 Debug Hardware Configure IP view on page 6-280
6.61 Connection Browser dialog box on page 6-285
7.4 Updating multiple debug hardware units on page 7-295

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.61 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 box, 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**.

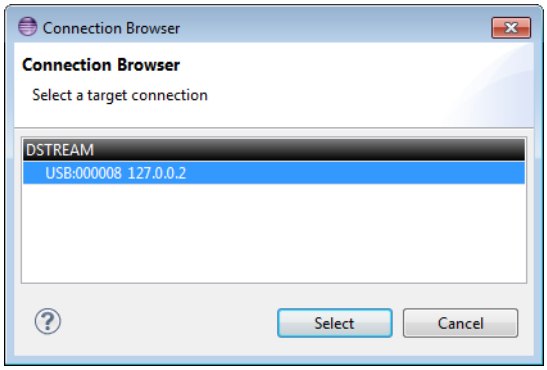


Figure 6-86 Connection Browser (Showing a USB connected DSTREAM)

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.

Related references

6.59 Debug Hardware Configure IP view on page 6-280

6.60 Debug Hardware Firmware Installer view on page 6-282

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.62 Preferences dialog box

You can customize your settings using the **Preferences** dialog box.

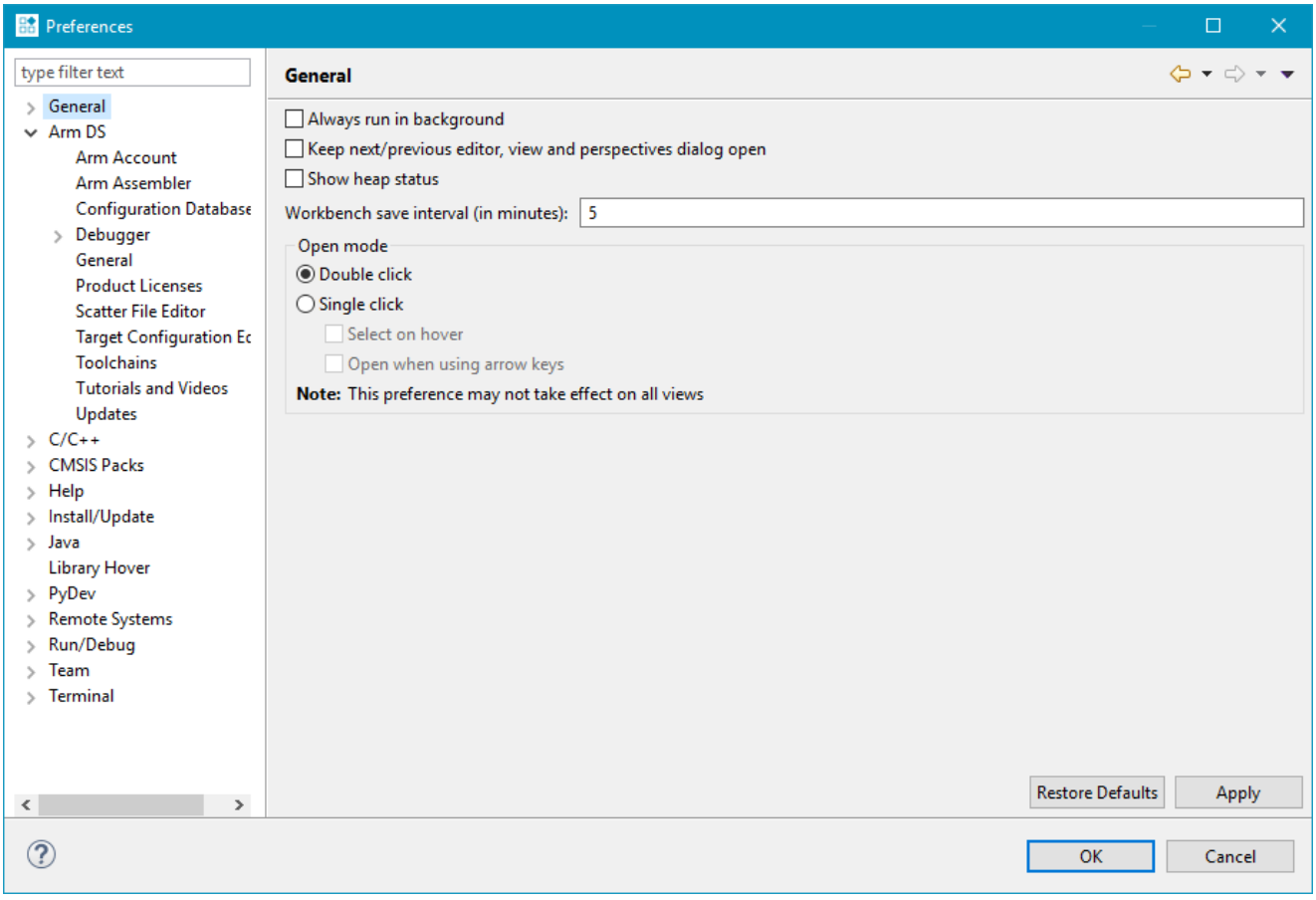


Figure 6-87 Window preferences dialog box

To access the **Preferences** dialog box, select **Preferences** from the **Window** menu. Changes to these settings are saved in the current workspace. If you want to copy your settings to another workspace, use the **Export** wizard.

The contents of the preferences hierarchy tree include the following groups:

General

Controls the workspace, perspectives, editors, build order, linked resources, file associations, path variables, background operations, keyboard and mouse settings.

Arm DS

Controls the default Arm Development Studio environment settings, presentation and formatting for Development Studio editors and views, target configuration database search locations, and the automatic checks for Development Studio product updates. You can also view Arm Development Studio license information.

C/C++

Controls the C/C++ environment settings, CDT build variables, syntax formatting, and default project wizard settings.

Help

Controls how the context help is displayed.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Install/Update

Controls the update history, scheduler, and policy.

Remote Systems

Controls the settings used by the **Remote System Explorer**.

Run/Debug

Controls the default perspectives, breakpoint, build, and launch settings before running and debugging.

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

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.63 Properties dialog box

You can customize project settings using the **Properties** dialog box.

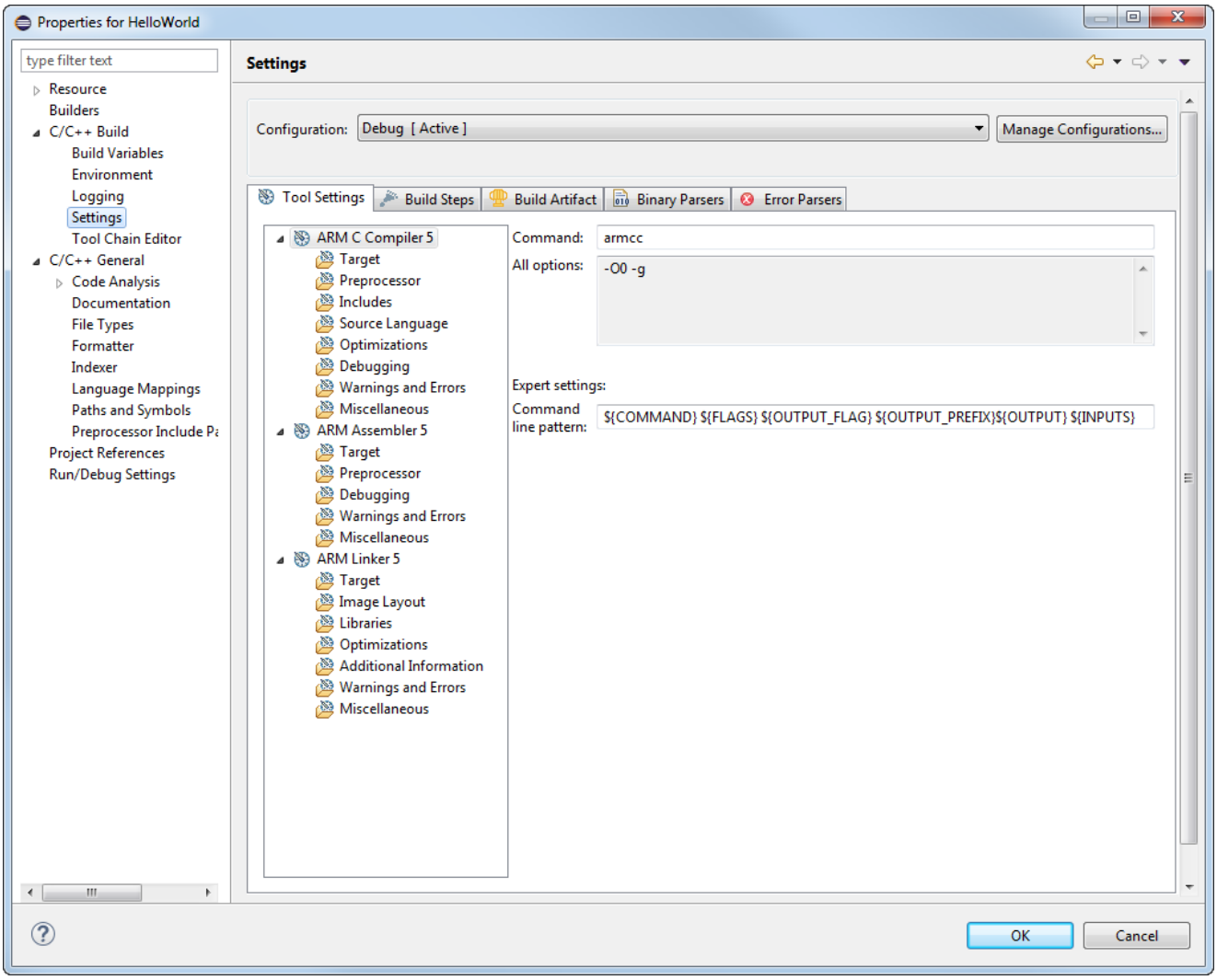


Figure 6-88 Project properties dialog box

To access the **Properties** dialog box select a project and then select **Properties…** from the **Project** menu. Changes to the customized settings are saved in the project folder in your workspace. You can also customize the C/C++ properties for a single file for example, if you want to apply a specific compiler option to a file during the build.

Note

If you specify different options for a single file, it overrides the options specified in the project configuration panels that apply to all related source files.

The contents of the properties hierarchy tree for a project include the following:

Resource

Displays the resource location, modification state, and file type.

Builders

Controls builders available for the selected project.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C/C++ Build

Controls the environment, build, and tool chain settings for the active configuration.

C/C++ General

Controls documentation, file types, indexer and path/symbol settings.

Project References

Controls project dependencies.

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

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 7
Reference

Lists other information that might be useful when working with Arm Debugger.

It contains the following sections:

- *7.1 About loading an image on to the target* on page 7-291.
- *7.2 About loading debug information into the debugger* on page 7-292.
- *7.3 About passing arguments to main()* on page 7-294.
- *7.4 Updating multiple debug hardware units* on page 7-295.
- *7.5 Standards compliance in Arm® Debugger* on page 7-296.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

7.1 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.

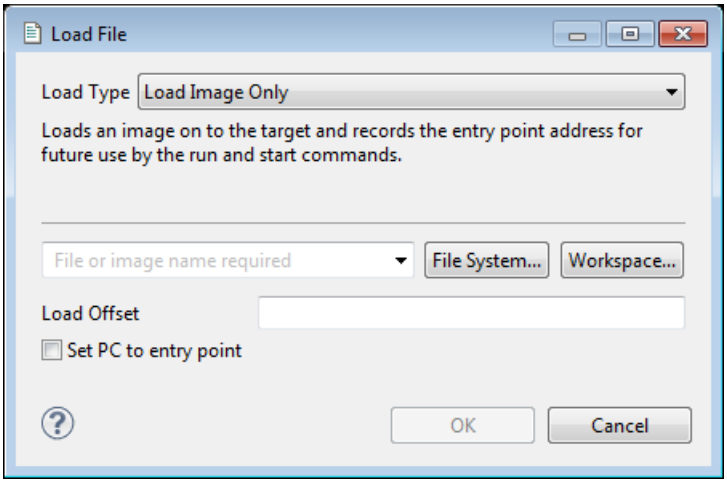


Figure 7-1 Load File dialog box

Related concepts

7.2 About loading debug information into the debugger on page 7-292

Related references

6.6 Commands view on page 6-148

Related information

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

7.2 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 an image is loaded to a target, 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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

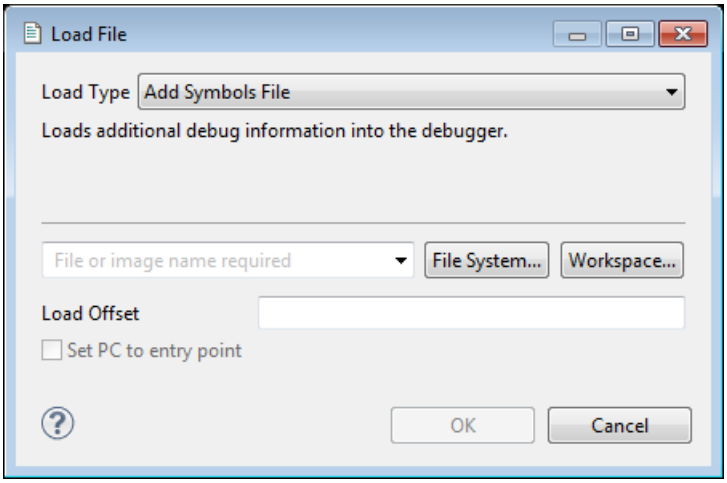


Figure 7-2 Load additional debug information dialog box

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* on page 2-70 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

7.1 About loading an image on to the target on page 7-291

Related tasks

2.18 Configuring the debugger path substitution rules on page 2-70

Related references

6.6 Commands view on page 6-148

Related information

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

7.3 About passing arguments to main()

Arm Debugger enables you to pass arguments to the `main()` function of your application.

You can use 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

2.16 Using semihosting to access resources on the host computer on page 2-66
2.17 Working with semihosting on page 2-68
6.47 Debug Configurations - Arguments tab on page 6-262

Related information

Arm Debugger commands

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

7.4 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 units.
- View the firmware versions on a group of DSTREAM units.

The input to **dbghw_batchupdater** is a file containing a list of DSTREAM units. Each line in the input file is a string that specifies a single DSTREAM connection. Firmware images are available within a subdirectory of the Arm Development Studio installation.

Syntax

```
dbghw_batchupdater -list <file>[-<option>]...
```

Where:

`list <file>`

Specifies the file containing a list of DSTREAM connection strings.

`option:`

Is one or more of the following:

`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.

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 2017-11-10 10:45:36
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\Arm
\Development Studio <version>\sw\debughw\firmware\ARM-RVI-4.34.0-22-base.dstream' -log
out.log
```

Related references

6.59 Debug Hardware Configure IP view on page 6-280

6.60 Debug Hardware Firmware Installer view on page 6-282

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

7.5 Standards compliance in Arm® Debugger

Arm Debugger conforms to various formats and protocols.

Executable and Linkable Format (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.

Note

The DWARF 2 and DWARF 3 standards are 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.

Trace Protocols

The debugger can interpret trace that complies with the Embedded Trace Macrocell (ETM) (v3 and above), Instrumentation Trace Macrocell (ITM), and System Trace Macrocell (STM) protocols.

Related information

ELF for the Arm Architecture

DWARF for the Arm Architecture

The DWARF Debugging Standard

International Organization for Standardization

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential