arm

Arm® Development Studio

Version 2022.0

User Guide

Non-Confidential

Issue 01

101470_2022.0_01_en

Arm® Development Studio
User Guide

Release information

Document history

Issue	Date	Confidentiality	Change
1800-00	27 November 2018	Non-Confidential	First release for Arm Development Studio
1800-01	18 December 2018	Non-Confidential	Documentation update 1 for Arm Development Studio 2018.0
1800-02	31 January 2019	Non-Confidential	Documentation update 2 for Arm Development Studio 2018.0
1900-00	11 April 2019	Non-Confidential	Updated document for Arm Development Studio 2019.0
1901-00	15 July 2019	Non-Confidential	Updated document for Arm Development Studio 2019.0-1
1910-00	1 November 2019	Non-Confidential	Updated document for Arm Development Studio 2019.1
2000-00	20 March 2020	Non-Confidential	Updated document for Arm Development Studio 2020.0
2000-01	3 July 2020	Non-Confidential	Documentation update 1 for Arm Development Studio 2020.0
2010-00	28 October 2020	Non-Confidential	Updated document for Arm Development Studio 2020.1
2021.0-00	19 March 2021	Non-Confidential	Updated document for Arm Development Studio 2021.0
2021.1-00	9 June 2021	Non-Confidential	Updated document for Arm Development Studio 2021.1
2021.1-01	26 August 2021	Non-Confidential	Documentation update 1 for Arm Development Studio 2021.1
2021.2-00	10 November 2021	Non-Confidential	Updated document for Arm Development Studio 2021.2

Issue	Date	Confidentiality	Change
2022.0-00	29 March 2022	Non-Confidential	Updated document for Arm Development Studio 2022.0 Beta
2022.0-01	27 April 2022	Non-Confidential	Updated document for Arm Development Studio 2022.0

Proprietary Notice

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.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

List of Tables

1 Introduction

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

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Signal names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace bold`	Language keywords when used outside example code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
⚠ Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
⚠ Warning	Requirements for the system. Not following these requirements might result in system failure or damage.

Convention	Use
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
Note	An important piece of information that needs your attention.
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2 Debugging Embedded Systems

Gives an introduction to debugging embedded systems.

2.1 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

2.2 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 information

About address spaces on page 23
Commands view on page 329
Arm Debugger commands
Address space prefixes
info memory command
info memory-parameters command

2.3 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 information

Commands view on page 329
Arm Debugger commands

2.4 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 AArch32 targets (for example, processors based on Armv7):

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

  Execution stopped in SYS mode at breakpoint 1: S:0x80000BA8.
  ```
- For address spaces in AArch64 targets (for example, processors based on Armv8-A, Armv8-R AArch64, or Armv9-A):

  ```
  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 information

About accessing AHB, APB, and AXI buses on page 21
About virtual and physical memory on page 22
About debugging hypervisors on page 24

2.5 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 information

Commands view on page 329
Arm Debugger Commands

2.6 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 information

Commands view on page 329
About debugging bare-metal symmetric multiprocessing systems on page 26
Arm Debugger Commands

2.7 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 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 information

About debugging big.LITTLE systems on page 25
About loading an image on to the target on page 584
About loading debug information into the debugger on page 585
Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
Running, stopping, and stepping through an application on page 59
Breakpoints view on page 322
Commands view on page 329
Disassembly view on page 339
Memory view on page 359
Modules view on page 373
Registers view on page 376
Variables view on page 408
Arm Debugger Commands

2.8 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.

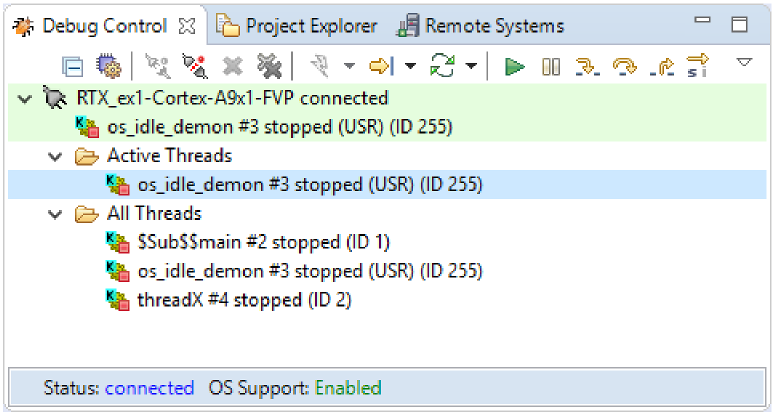
Threads are displayed in the **Debug Control** view with a unique ID that is used by the debugger and a unique ID from the Operating System (OS), for example:

```
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 2-1: Threading call stacks in the Debug Control view



```
RTX_ex1-Cortex-A9x1-FVP connected
    os_idle_demon #3 stopped (USR) (ID 255)
  Active Threads
      os_idle_demon #3 stopped (USR) (ID 255)
  All Threads
      $Sub$$main #2 stopped (ID 1)
      os_idle_demon #3 stopped (USR) (ID 255)
      threadX #4 stopped (ID 2)

Status: connected   OS Support: Enabled
```

Related information

Breakpoints view on page 322
Commands view on page 329
Disassembly view on page 339
Memory view on page 359
Modules view on page 373
Registers view on page 376
Variables view on page 408

2.9 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.

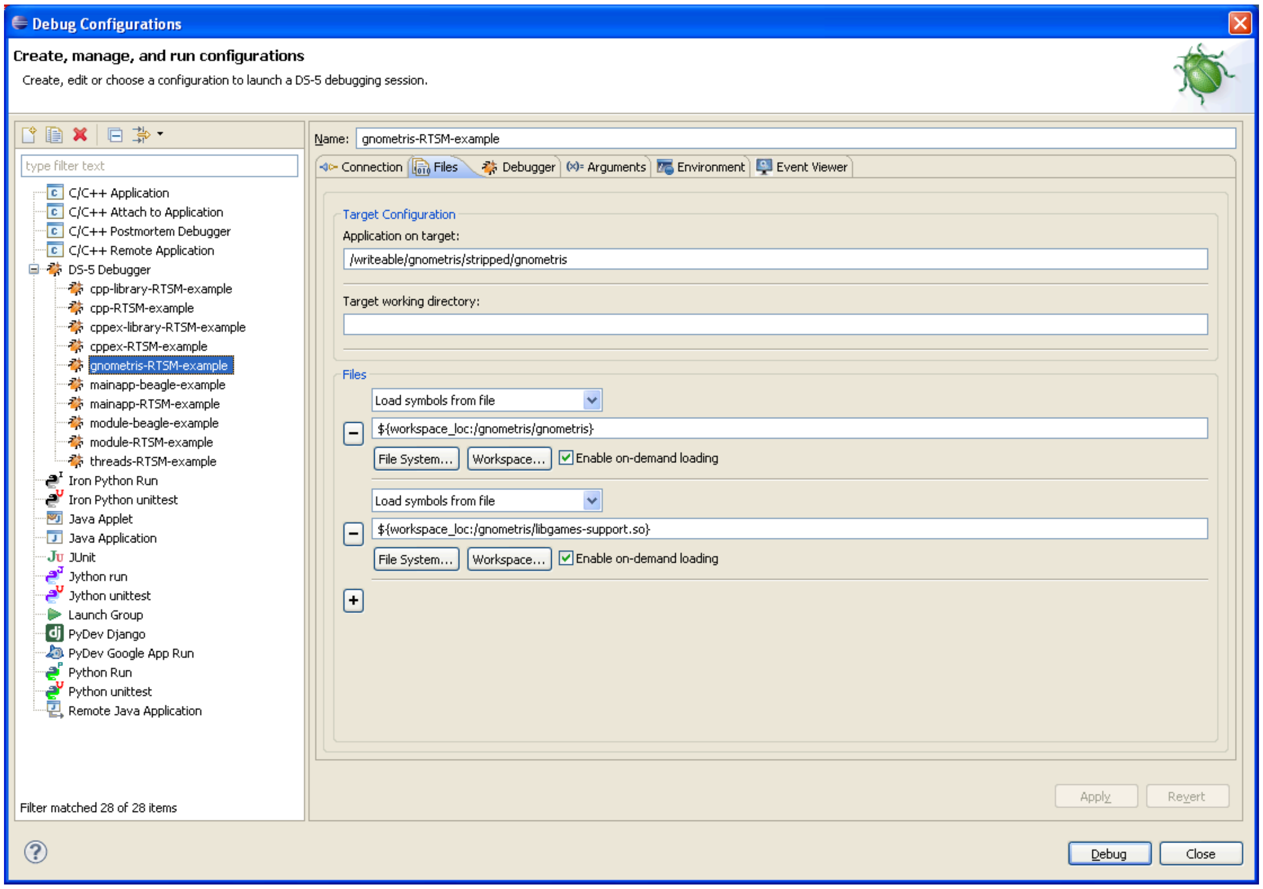
You can set standard execution breakpoints in a shared library but not until it is loaded by the application and the debug information is loaded into the debugger. Pending breakpoints however, enable you to set execution breakpoints in a shared library before it is loaded by the application.

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 2-2: Adding individual shared library files



Name: gnometris-RTSM-example

Application on target: /writeable/gnometris/stripped/gnometris

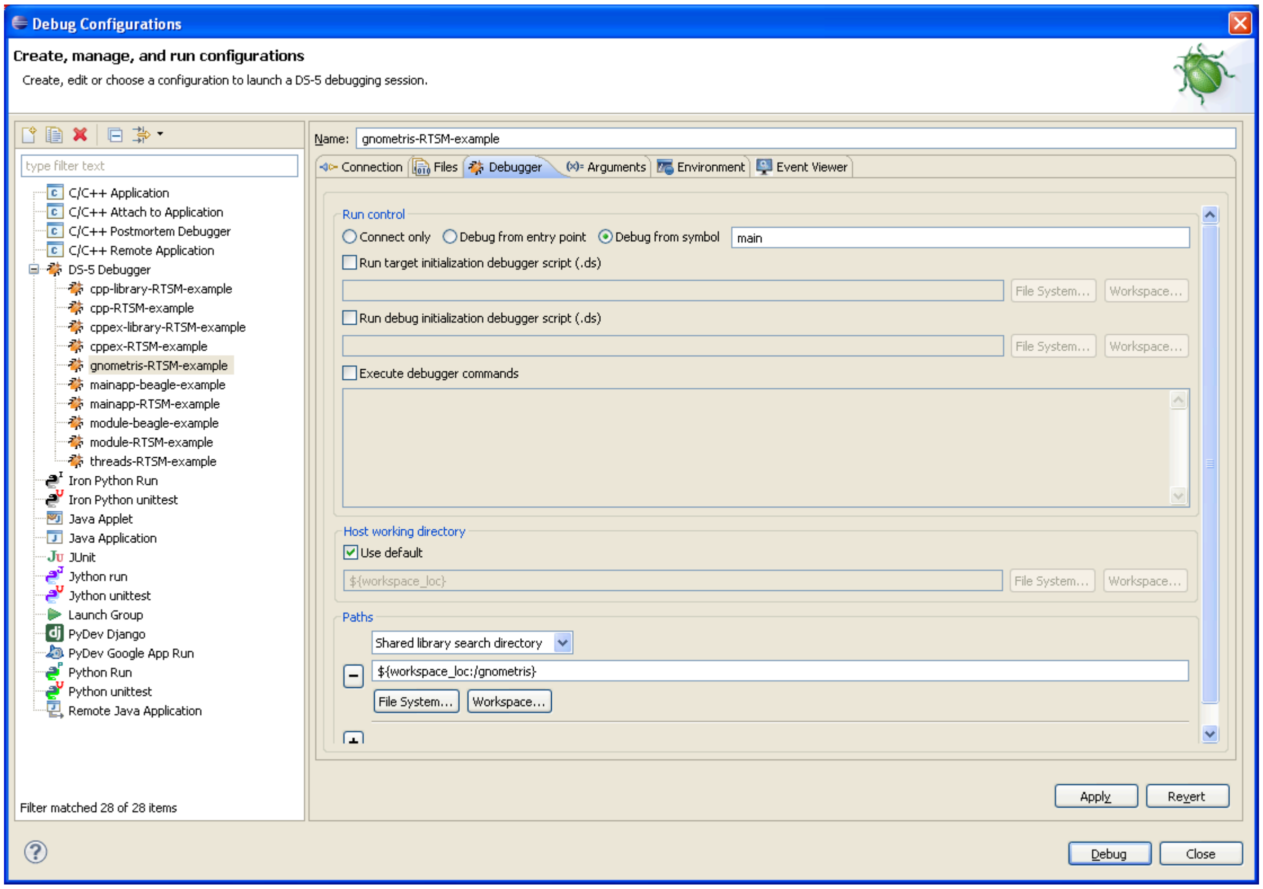
Target working directory:

Load symbols from file: ${workspace_loc:/gnometris/gnometris}
☑ Enable on-demand loading

Load symbols from file: ${workspace_loc:/gnometris/libgames-support.so}
☑ Enable on-demand loading

Alternatively if you have multiple library files then it is probably more efficient to modify the search paths in use by the debugger when searching for shared libraries. To do this you can use the **Shared library search directory** option in the Paths panel of the **Debugger** tab.

Figure 2-3: Modifying the shared library search paths



Name: gnometris-RTSM-example

Run control
( ) Connect only ( ) Debug from entry point (●) Debug from symbol main
☐ Run target initialization debugger script (.ds)
☐ Run debug initialization debugger script (.ds)
☐ Execute debugger commands

Host working directory
☑ Use default
${workspace_loc}

Paths
Shared library search directory
${workspace_loc:/gnometris}

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

Related information

Running, stopping, and stepping through an application on page 59
Examining the target execution environment on page 120
Examining the call stack on page 121
Handling UNIX signals on page 75
Handling processor exceptions on page 77
Breakpoints view on page 322
Commands view on page 329
Disassembly view on page 339
Memory view on page 359
Modules view on page 373
Registers view on page 376
Variables view on page 408

2.10 About OS awareness

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

The OS awareness support in Arm Debugger depends on the OS version and the processor architecture on the target.

Arm Debugger provides OS awareness for:

- ThreadX 5.6, 5.7: Armv5, Armv5T, Armv5TE, Armv5TEJ, Armv6-M, Armv7-M, Armv7-R, Armv7-A, Armv8-A.
- µC/OS-II 2.92: Armv6-M, Armv7-M, Armv7-R, Armv7-A.
- µC/OS-III 3.04: Armv6-M, Armv7-M, Armv7-R, Armv7-A.
- embOS 3.88: Armv5, Armv5T, Armv5TE, Armv5TEJ, Armv6-M, Armv7-M, Armv7-R, Armv7-A.
- Keil® CMSIS-RTOS RTX 4.7 and RTX 5: Armv6-M, Armv7-M, Armv7-R, Armv7-A, Armv8-M.
- FreeRTOS 10.2.1: Armv6-M, Armv7-M, Armv7-R, Armv7-A, Armv8-M.
- Freescale MQX 4.0: Freescale-based Cortex®-M4 and Cortex-A5 processors
- Quadros RTXC 1.0.2: Armv5, Armv5T, Armv5TE, Armv5TEJ, Armv7-M, Armv7-R, Armv7-A.
- Nucleus RTOS 2014.06: Armv5, Armv5T, Armv5TE, Armv5TEJ, Armv6-M, Armv7-M, Armv7-R, Armv7-A.
- µC3 Standard: Armv7-R, Armv7-A.
- µC3 Compact: Armv6-M, Armv7-M.
- PikeOS 4.1, 4.2: Armv7-A, Armv7-R, Armv8-A.
- VxWorks 7: Armv7-A, Armv7-R, Armv8-A.

Note

- By default, OS awareness is not present for an architecture or processor that is not listed above.
- OS awareness support for newer versions of the OS depends on the scope of changes to their internal data structures.
- OS awareness in Arm Debugger does not support the original non-CMSIS Keil RTX.
- OS awareness for µC3 Standard requires you to set the `vfp-flag` parameter based on the `--fpu` option that the µC3 Standard kernel was compiled with. You can set this using the **OS Awareness** tab in the **Debug Configurations** dialog box, or using the command `set os vfp-flag`. You can set the value to `disabled`, `vfpv3_16`, or `vfpv3_32`.

The Linux kernels that Arm Debugger provides OS awareness for are:

- Linux 2.6.28, Armv7-A

- Linux 2.6.38: Armv7-A
- Linux 3.0: Armv7-A
- Linux 3.11.0-rc6: Armv7-A
- Linux 3.13.0-rc3: Armv7-A
- Linux 3.6.0-rc6: Armv7-A
- Linux 3.7.0: Armv7-A
- Linux 3.9.0-rc3: Armv7-A
- Linux 3.11.0-rc6: Armv8-A

Note

Later versions of Linux are expected to work on Armv7-A Armv8-A architectures.

2.10.1 About debugging FreeRTOS

FreeRTOS is an open-source real-time operating system.

Arm® Debugger provides the following support for debugging FreeRTOS:

- Supports FreeRTOS on Cortex®-M cores.
- View FreeRTOS tasks in the **Debug Control** view.
- View FreeRTOS tasks and queues in the **RTOS Data** view.

To enable FreeRTOS support in Arm Debugger, in the **Debug Configuration** dialog box, select FreeRTOS in the **OS** tab. Debugger support is activated when FreeRTOS is initialized on the target device.

Note

Operating system support in the debugger is activated only when OS-specific debug symbols are loaded. Ensure that the debug symbols for the operating system are loaded before using any of the OS-specific views and commands.

When building your FreeRTOS image, ensure that the following compiler flags are set:

- `-DportREMOVE_STATIC_QUALIFIER`
- `-DINCLUDE_xTaskGetIdleTaskHandle`
- `-DconfigQUEUE_REGISTRY_SIZE=n` (where n >= 1)

If these flags are set incorrectly, FreeRTOS support might fail to activate in Arm Debugger See the documentation supplied with FreeRTOS to view the details of these flags.

2.10.2 About debugging a Linux kernel

Arm® Development Studio supports source level debugging of a Linux kernel. The Linux kernel (and associated device drivers) can be debugged in the same way as a standard ELF format executable. For example, you can set breakpoints in the kernel code, step through the source, inspect the call stack, and watch variables.

Note

User space parameters (marked `__user`) that are not currently mapped in cannot be read by the debugger.

To debug the kernel:

1. Compile the kernel source using the following options:

   `CONFIG_DEBUG_KERNEL=y`
   Enables the kernel debug options.

   `CONFIG_DEBUG_INFO=y`
   Builds `vmlinux` with debugging information.

   `CONFIG_DEBUG_INFO_REDUCED=n`
   Includes full debugging information when compiling the kernel.

   `CONFIG_PERF_EVENTS=n`
   Disables the performance events subsystem. Some implementations of the performance events subsystem internally make use of hardware breakpoints, disrupting the use of hardware breakpoints set by the debugger. It is recommended to disable this option if you observe the debugger failing to hit hardware breakpoints or failing to report kernel module load and unload events.

   Note

   If you are working with Arm Streamline, `CONFIG_PERF_EVENTS` must be enabled.

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

   Note

   Be aware that:

   - Other options might be required depending on the type of debugging you want to perform. Check the kernel documentation for details.
   - A Linux kernel is always compiled with full optimizations and inlining enabled, therefore:
     - Stepping through code might not work as expected due to the possible reordering of some instructions.

- Some variables might be optimized out by the compiler and therefore not be available for the debugger.

2. Load the Linux kernel on to the target.
3. Load kernel debug information into the debugger.

Note

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

4. Debug the kernel as required.

Related information

About debugging Linux kernel modules on page 35
About debugging bare-metal symmetric multiprocessing systems on page 26
Running, stopping, and stepping through an application on page 59
Examining the target execution environment on page 120
Examining the call stack on page 121
Handling UNIX signals on page 75
Handling processor exceptions on page 77
Debug Configurations - Files tab on page 434
Debug Configurations - Debugger tab on page 438
Breakpoints view on page 322
Commands view on page 329
Disassembly view on page 339
Memory view on page 359
Modules view on page 373
Registers view on page 376
Variables view on page 408
Configuring a connection to a Linux kernel

2.10.3 About debugging Linux kernel modules

Linux kernel modules provide a way to extend the functionality of the kernel, and are typically used for things such as device and file system drivers. Modules can either be built into the kernel or can be compiled as a loadable module and then dynamically inserted and removed from a running kernel during development without having to frequently recompile the kernel. However, some modules must be built into the kernel and are not suitable for loading dynamically. An example of a built-in module is one that is required during kernel boot and must be available prior to the root file system being mounted.

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

When debugging a module, you must ensure that the module on your target is the same as that on your host. The code layout must be identical, but the module on your target does not require debug information.

Built-in module

To debug a module that has been built into the kernel, the procedure is the same as for debugging the kernel itself:

1. Compile the kernel together with the module.
2. Load the kernel image on to the target.
3. Load the related kernel image with debug information into the debugger
4. Debug the module as you would for any other kernel code.

Built-in (statically linked) modules are indistinguishable from the rest of the kernel code, so are not listed by the `info os-modules` command and do not appear in the **Modules** view.

Loadable module

The procedure for debugging a loadable kernel module is more complex. From a Linux terminal shell, you can use the `insmod` and `rmmod` commands to insert and remove a module. Debug information for both the kernel and the loadable module must be loaded into the debugger. When you insert and remove a module the debugger automatically resolves memory locations for debug information and existing breakpoints. To do this, the debugger intercepts calls within the kernel to insert and remove modules. This introduces a small delay for each action whilst the debugger stops the kernel to interrogate various data structures.

Note

A connection must be established and Operating System (OS) support enabled within the debugger before a loadable module can be detected. OS support is automatically enabled when a Linux kernel image is loaded into the debugger. However, you can manually control this by using the `set os` command.

Related information

About debugging a Linux kernel on page 33
About debugging bare-metal symmetric multiprocessing systems on page 26
Running, stopping, and stepping through an application on page 59
Examining the target execution environment on page 120
Examining the call stack on page 121

Handling UNIX signals on page 75
Handling processor exceptions on page 77
Breakpoints view on page 322
Commands view on page 329
Disassembly view on page 339
Memory view on page 359
Modules view on page 373
Registers view on page 376
Variables view on page 408
Configuring a connection to a Linux kernel

2.10.4 About debugging ThreadX

ThreadX is a real-time operating system from Express Logic, Inc.

Arm® Debugger provides the following ThreadX RTOS visibility:

- Comprehensive thread list with thread status and objects on which the threads are blocked/ suspended.
- All major ThreadX objects including semaphores, mutexes, memory pools, message queues, event flags, and timers.
- Stack usage for individual threads.
- Call frames and local variables for all threads.

To enable ThreadX support in Arm Debugger, in the **Debug Configuration** dialog box, select **ThreadX** in the **OS Awareness** tab. ThreadX OS awareness is activated when ThreadX is initialized on the target device.

2.10.5 About debugging PikeOS

Arm® Development Studio supports source level debugging of PikeOS.

From a debugging perspective, PikeOS consists of mainly these parts:

- PikeOS kernel.
- PikeOS System Software (PSSW).
- Applications which run on the operating system.

All these parts have separate symbol files and you must ensure that the symbols for the relevant parts are loaded at the correct address spaces.

Debugging PikeOS after the MMU is enabled

1. Load the PikeOS image onto the target. You can use the Development Studio restore command to load the image into the target RAM.

Note

- The PikeOS image does not contain debug information, even if using the PikeOS ELF boot strategy.
- Loading the image on to your target depends on your boot strategy and target. Check the PikeOS documentation for instructions.
- If using the CODEO tool, the boot image is the output of an **Integration Project**.

2. Load the kernel debug information. You can use the Development Studio add-symbol-file command to load the kernel debug information into the debugger.

Note

- The address space where the kernel runs might differ from the current address space, especially if stopped during the boot process. Ensure that the kernel debug image is loaded to the address space that the kernel runs in, for example, EL1N.
- Do not enable OS awareness before the MMU is enabled. If you want to debug PikeOS, before the MMU is enabled, see the *Debugging PikeOS before the MMU is enabled* section later in this topic.
- The kernel debug image must exactly match the kernel with which the loaded image was compiled. Loading this information is necessary for the OS Awareness to function.
- If using the CODEO tool, you can determine the kernel that is used from the **PikeOS Kernel** section of the **Integration Project**'s **Project Configuration**. The debug image is typically stored with a `.elf` or `.unstripped` file extension.

3. Load the debug information for your application or the PikeOS System Software using the `add-symbol-file` command. Ensure that they are loaded to the correct address space.

Note

If using the CODEO tool, you can determine the location of files containing debug information from the relevant sections of the **Integration Project**'s **Project Configuration**. The debug images are typically stored with a `.elf` or `.unstripped` file extension.

This step is not required or used by the OS awareness, but improves your experience if you plan to debug either of these components.

Note

When debugging PikeOS, if you inspect unscheduled OS threads, their current Program Counter might point to an address which is not currently mapped in by the MMU.

Debugging PikeOS before the MMU is enabled

The initial enablement of the MMU is done by the Platform Support Package (PSP) during its early initialization. The point at which the MMU is enabled differs by platform. Some PSPs provide explicit symbols to mark the first virtual instructions, while others do not. To find the first point at which the MMU is enabled, inspection of the specific platform's PSP is required.

The MMU is enabled by the time the PSP starts the kernel and the kernel entry point `P4Main` is passed. When the MMU is enabled, it is safe to enable OS awareness.

The kernel debug image contains debug symbols for both the kernel and the PSP. All symbols have virtual addresses and require the MMU to be enabled. The early PSP initialization occurs before the MMU is enabled. To debug the early PSP initialization, load the kernel symbols with an offset so that their offset virtual addresses align with their physical counterparts.

Since the MMU is currently off, Arm Debugger's OS support must be disabled before the kernel debug symbols are loaded. This is to avoid the debugger from trying to read the kernel structures before they are set up (and possibly resulting in Data Aborts). To disable OS awareness, enter set os enabled off in the Development Studio **Command** view and click **Submit** or press **Enter**.

To calculate the required load offset, calculate the difference (`P-V`) between the physical start address of the boot image (`P`) and the virtual start address of the kernel image (`V`).

For example, if the kernel is linked at virtual address `0x80000000` and is loaded at physical address `0x20000000`, the offset is `-0x60000000`, which is `0x20000000 - 0x80000000` (`P-V`).

Note

Early PSP initialization typically runs at a higher Exception level than the rest of the kernel. You must take care to ensure that the offset symbols are also loaded in the correct address space. An example of a full offset for an Armv8 target is `EL2N:-0xFFFFFF7F80000000`.

See About loading debug information into the debugger for information on loading debug symbols into the debugger.

When the MMU is enabled, the previously loaded debug information must be reloaded at the unadjusted virtual addresses. To reload the debug information, first, enter file, symbol file in the Development Studio **Command** view to discard currently loaded symbols. Then, use the `add-symbol-file` command to load the kernel debug information into the debugger, but this time with zero offset.

2.11 About debugging TrustZone enabled targets

Arm® TrustZone® is a security technology designed into some Arm processors. For example, the Cortex®-A class processors. It segments execution and resources such as memory and peripherals into secure and normal worlds.

When connected to a target that supports TrustZone and where access to the secure world is permitted, then the debugger provides access to both secure and normal worlds. In this case, all addresses and address-related operations are specific to a single world. This means that any commands you use that require an address or expression must also specify the world that they apply to, with a prefix. For example `N:0x1000` or `S:0x1000`.

Where:

N:

For an address in Normal World memory.

S:

For an address in Secure World memory.

If you want to specify an address in the current world, then you can omit the prefix.

When loading images and debug information it is important that you load them into the correct world. The debug launcher panel does not provide a way to directly specify an address world for images and debug information, so to achieve this you must use scripting commands instead. The **Debugger** tab in the debugger launcher panel provides an option to run a debug initialization script or a set of arbitrary debugger commands on connection. Here are some example commands:

- Load image only to normal world (applying zero offset to addresses in the image)

```
load myimage.axf N:0
```

- Load debug information only to secure world (applying zero offset to addresses in the debug information)

```
file myimage.axf S:0
```

- Load image and debug information to secure world (applying zero offset to addresses)

```
loadfile myimage.axf S:0
```

When an operation such as loading debug symbols or setting a breakpoint needs to apply to both normal and secure worlds then you must perform the operation twice, once for the normal world and once for the secure world.

Registers such as `$PC` have no world. To access the content of memory from an address in a register that is not in the current world, you can use an expression, `N:0+$PC` . This is generally not necessary for expressions involving debug information, because these are associated with a world when they are loaded.

Related information

Breakpoints view on page 322
Commands view on page 329
Disassembly view on page 339
Memory view on page 359
Modules view on page 373
Registers view on page 376
Variables view on page 408
Arm Debugger Commands
Arm Security Technology Building a Secure System using TrustZone Technology
Technical Reference Manual
Architecture Reference Manual

2.12 About debugging a Unified Extensible Firmware Interface (UEFI)

UEFI defines a software interface to control the start-up of complex microprocessor systems. UEFI on Arm allows you to control the booting of Arm®-based servers and client computing devices.

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 for Embedded.
- 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

2.13 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:

  ```
  cache flush
  ```

Note

Flushing large caches might take a long time.

Related information

MMU/MPU view on page 368
About debugging caches on page 45
mmu commands

2.14 About Debug and Trace Services Layer (DTSL)

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

DTSL takes responsibility for:

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

DTSL is implemented as a set of Java classes which are typically implemented (and possibly extended) by Jython scripts. A typical DTSL instance is a combination of Java and Jython.

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

Note

DTSL Jython Scripting should not be confused with Arm Debugger Jython Scripting. They both use Jython but operate at different levels within the software stack. It is however possible for a debugger Jython Script to use DTSL functionality.

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

For details, see the DTSL documents and files provided with Arm Development Studio here:

```
<installation_directory>/sw/DTSL
```

Related information

Debug and Trace Services Layer (DTSL) on page 511

2.15 About CoreSight Target Access Library

CoreSight™ on-target access library allows you to interact directly with CoreSight devices. This supports use-cases such as enabling flight-recorder trace in a production system without the need to connect an external debugger.

The library offers a flexible programming interface allowing a variety of use cases and experimentation.

It also offers some advantages compared to a register-level interface. For example, it can:

- Manage any unlocking and locking of CoreSight devices via the lock register, OS Lock register, programming bit, power-down bit.
- Attempt to ensure that the devices are programmed correctly and in a suitable sequence.
- Handle variations between devices, and where necessary, work around known issues. For example, between variants of ETM/PTMs.
- Become aware of the trace bus topology and can generally manage trace links automatically. For example enabling only funnel ports in use.
- Manage 'claim bits' that coordinate internal and external use of CoreSight devices.

For details, see the CoreSight example provided with Arm® Development Studio here:

```
<installation_directory>/examples/CoreSight_Access_Library.zip
```

2.16 Debug and trace over functional I/O

CoreSight™ SoC-600 introduces access standards to the debug memory space that enable access using existing functional interfaces such as USB, ethernet, or PCIe, as an alternative to using the traditional JTAG and Serial Wire Debug interfaces, to debug your target. It also introduces an enhanced Embedded Trace Router (ETR) that supports high bandwidth streaming trace mode, which you can use to offload trace data over the functional interfaces.

For an end-to-end debug and trace over functional interfaces solution with Arm® Development Studio, you require the following software components:

- An on-target debug agent that provides an OS specific, or bare metal mechanism, to allow the on-target agent access to the CoreSight SoC-600 components.
- Functional interface drivers, such as USB or PCIe, for both the host and target.
- A transport layer protocol.
- A standard debug and trace API to the debugger.

The on-target debug agent, the functional interface drivers, and the `probes.xml` file, are provided by the SoC vendor.

Arm provides a standardized transport protocol that is agnostic of the physical link, called the CoreSight Wire Protocol (CSWP), and APIs to the debugger; RDDI MEM-AP for debug, and RDDI Streaming Trace for trace.

Although some of the software components are the responsibility of the SoC vendor, Arm provides an end-to-end open-source package that contains reference implementations of the CSWP protocol handlers, for both the host and target, and debug and trace API sets. For completeness, Arm also provides an example implementation of a Linux-based on-target debug agent, and its associated drivers.

You can find the open-source package here: https://github.com/ARM-software/coresight-wire-protocol

---

Note

You can optionally replace the Arm-provided CSWP transport protocol with a protocol from the SoC vendor. You are responsible for the host and target protocol handlers when using a protocol not provided by Arm.

A complete end-to-end solution results in an RDDI MEM-AP implementation for debug, and an RDDI Streaming Trace implementation, both in the form of a shared library that is loaded by Arm Debugger. These libraries are provided by the SoC or tool vendors. To associate the shared libraries with your new connection type, you need a probe definition file, called `probes.xml`. All dependencies of the APIs, including the transport protocol, drivers, and third-party libraries, must either be included in the shared libraries or provided alongside them.

When you have configured and associated all of these software components, the result is a new virtual probe. See Add a debug connection over functional I/O for details on how to implement a virtual probe.

Related information

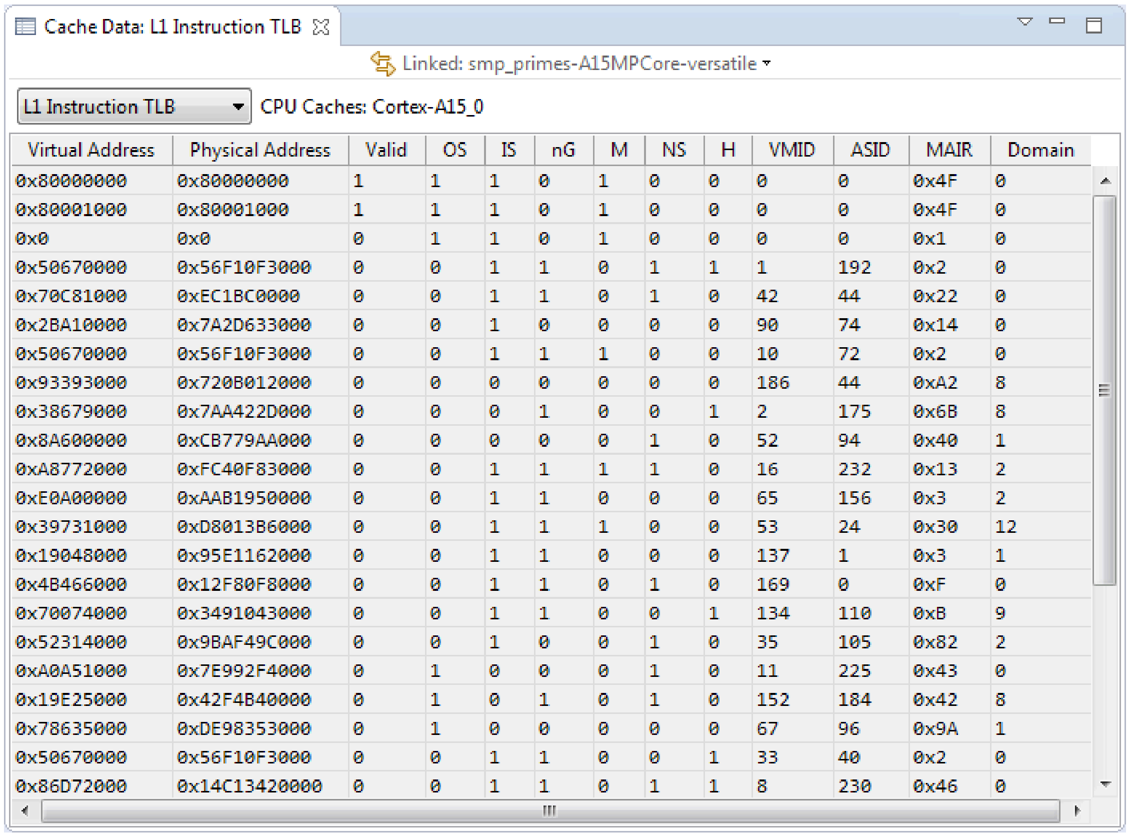
Add a debug connection over functional I/O on page 278
Add a third-party debug probe on page 274

2.17 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.

Figure 2-4: Cache Data view (showing L1 TLB cache)


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

Note

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

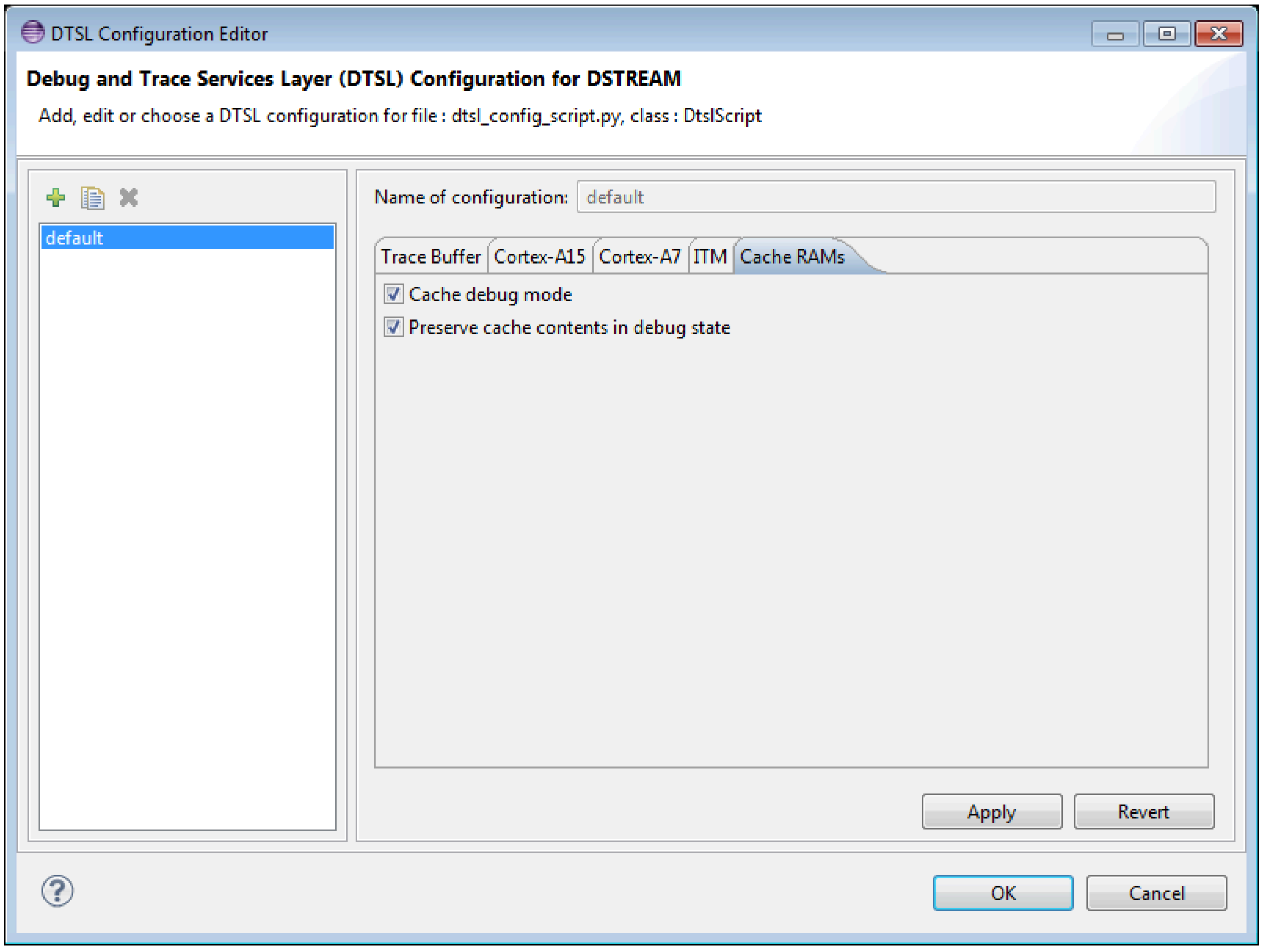
The **Cache debug mode** option in the **DTSL Configuration Editor** dialog box enables or disables the reading of cache RAMs in the **Cache Data** view. Selecting this option enables the reading of cache RAMs every time the target stops, if the **Cache Data** view is suitably configured.

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.

Figure 2-5: DTSL Configuration Editor (Shown with cache read option enabled)


DTSL Configuration Editor
Debug and Trace Services Layer (DTSL) Configuration for DSTREAM
Add, edit or choose a DTSL configuration for file : dtsl_config_script.py, class : DtslScript
default
Name of configuration: default
Trace Buffer
Cortex-A15
Cortex-A7
ITM
Cache RAMs
Apply
Revert
OK
Cancel

☑ Cache debug mode
☑ Preserve cache contents in debug state

Note

For processors based on the Armv8-A or Armv9-A 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`

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 information

Cache Data view on page 387
Memory view on page 359
DTSL Configuration Editor dialog box on page 447
cache list command
cache print command
memory command

2.18 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 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 information

Overlays view on page 386
Arm Compiler Software Development Guide: Overlay support
info overlays command
set overlays enabled command

2.19 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.

About this task

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

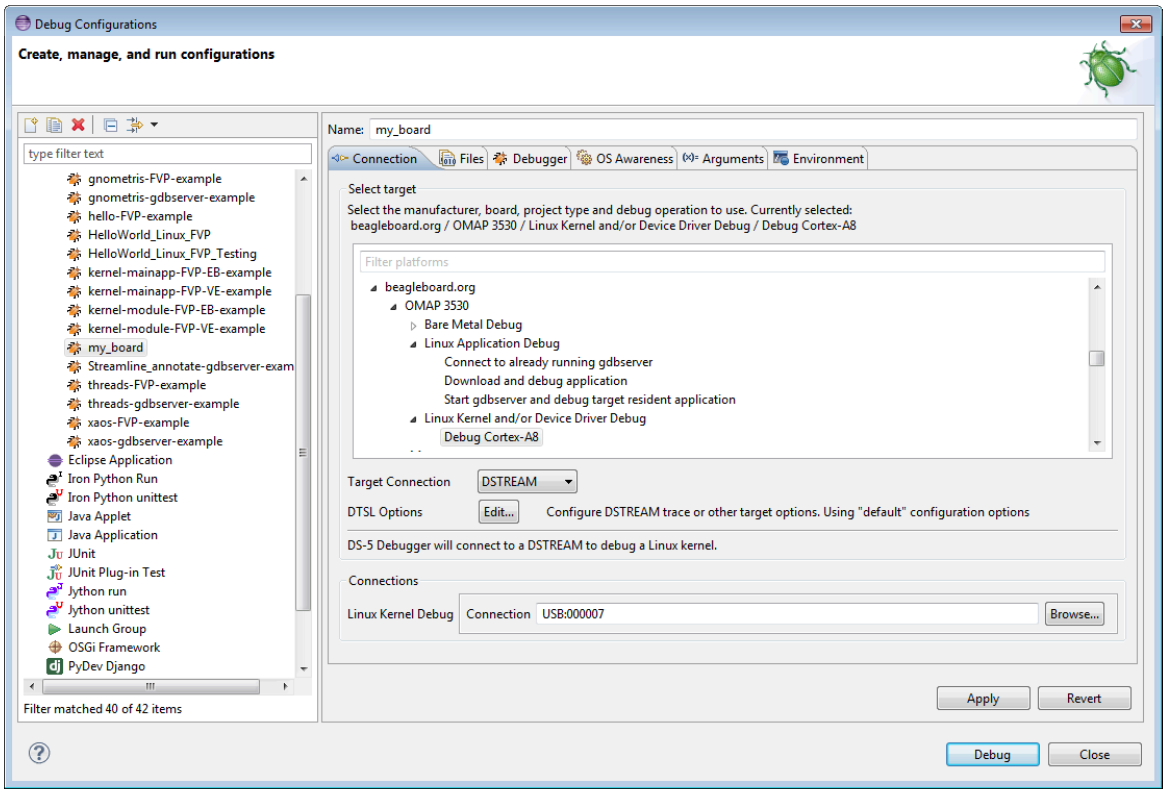
If you are working with your own module, before you can debug it, you must ensure that you:

- 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.

Figure 2-6: Typical connection settings for a Linux kernel/Device Driver Debug



Name: my_board

Select the manufacturer, board, project type and debug operation to use. Currently selected:
beagleboard.org / OMAP 3530 / Linux Kernel and/or Device Driver Debug / Debug Cortex-A8

Target Connection: DSTREAM

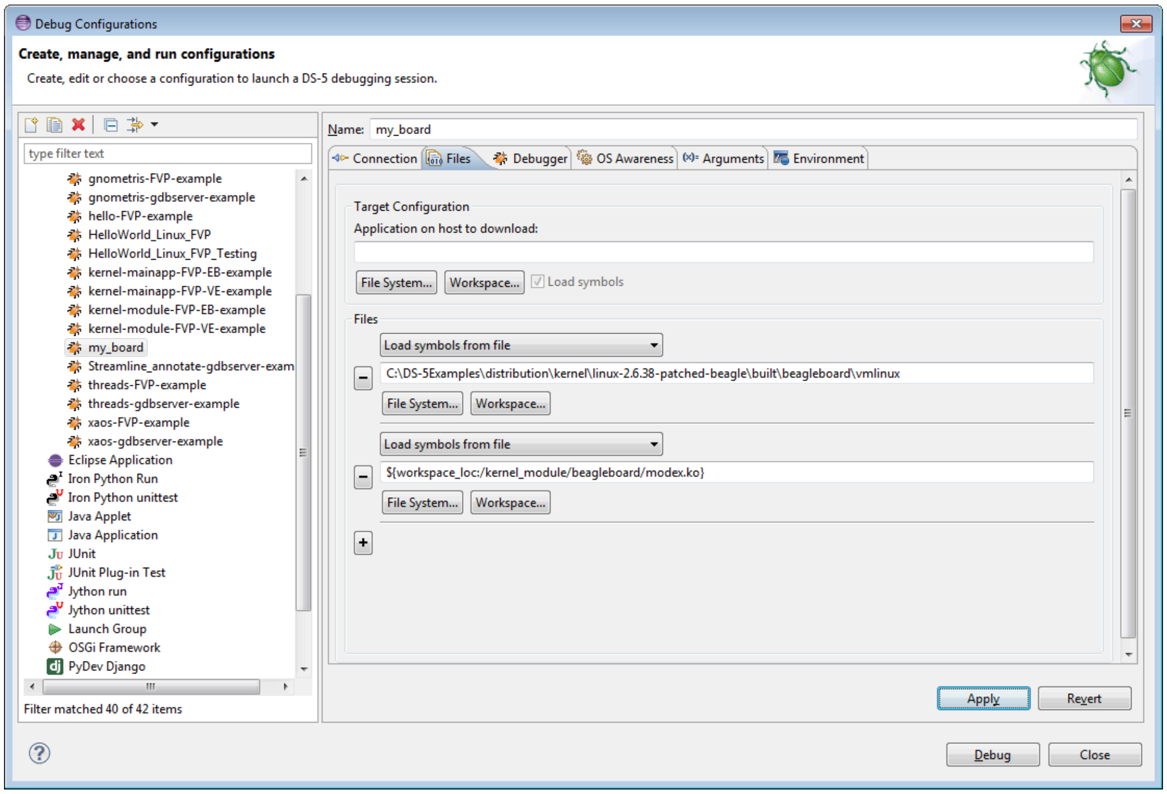
DTSL Options: Configure DSTREAM trace or other target options. Using "default" configuration options

DS-5 Debugger will connect to a DSTREAM to debug a Linux kernel.

Linux Kernel Debug — Connection: USB:000007

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

Figure 2-7: Typical Files settings for a Linux kernel/Device Driver Debug



Name: my_board

Application on host to download:

Load symbols from file: C:\DS-5Examples\distribution\kernel\linux-2.6.38-patched-beagle\built\beagleboard\vmlinux

Load symbols from file: ${workspace_loc:/kernel_module/beagleboard/modex.ko}

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.

Results

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 information

Useful commands for debugging a kernel module on page 53

2.20 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 information

Debugging a loadable kernel module on page 49

2.21 Performance analysis of the threads application running on Arm Linux

Arm® Streamline is a graphical performance analysis tool. It captures a wide variety of statistics about code running on the target and uses them to generate analysis reports. You can use these to identify problem areas at system, process, and thread level, in addition to hot spots in the code.

Before you begin

This tutorial uses the `threads_v7A` example application to show how to use Arm Streamline to capture and analyze profiling data from a Linux target. `threads_v7A` and `threads_v8A` are two of the Arm Linux application examples that are provided with Arm Development Studio.

Before capturing the data, ensure that:

1. You have built the `threads_v7A` application.
2. You know the IP address or network name of the target. To find the IP address, you can use the `ifconfig` application in a Linux console. The IP address is denoted by the `inet addr`.
3. The Linux kernel on the target is configured to work with Arm Streamline.
4. The gator daemon, `gatord`, is running on the target. If not, the simplest way to install and run `gatord` on the target is to use the **Setup Target...** button in the **Connection Browser** dialog box. The **Connection Browser** dialog box is accessible through the **Streamline Data** view by clicking on the **Browse for a target** button.
5. SSH and `gdbserver` are running on the target.

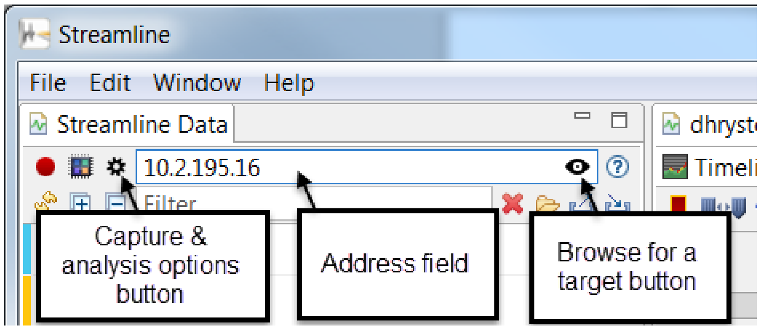
Note

- For more information about building and running the `threads` application on a Linux target see the `readme.html` supplied in the same directory as the source code for the example.
- For more information about how to configure your target for Arm Streamline, see the Arm Streamline User Guide.

Procedure

1. Launch Arm Development Studio.
2. In the **Remote Systems** view, click and define a connection to the target
3. Launch the Arm Streamline application.
4. Specify the IP address or network name of the target in the **Address** field. Alternatively, use the **Browse for a target** button, as shown in the following screenshot:

Figure 2-8: Streamline Data view

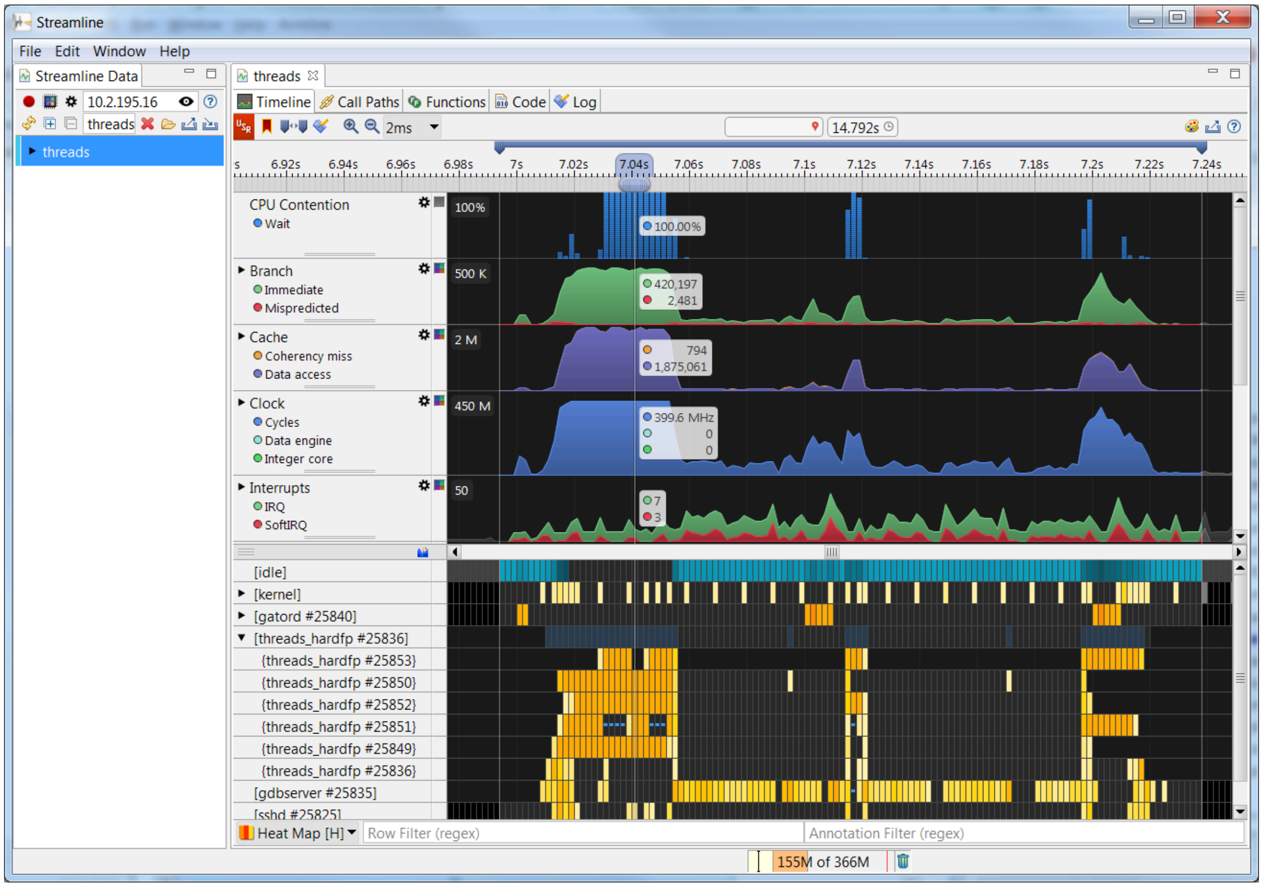


5. Click the **Capture & analysis options** button. In the **Program Images** section, select the `threads` image from the workspace, then select **Save**.
6. In Arm DS select **Run** > **Debug configurations...** then select the `threads-gdbserver` debug configuration. This configuration downloads the application to the target, starts `gdbserver` on the target and starts executing the application, stopping at `main()`.
7. Connect to the target either by clicking **Debug** in the **Debug Configurations** dialog box, or by right-clicking on the connection in the **Debug Control** view and selecting **Connect to target**.
8. The program stops at `main()`. To start capturing data, switch to the **Streamline** application and click ● . Give the capture file a unique name. The **Live** view opens in **Streamline**, displaying the capture data in real time.
9. In Arm DS, click **Continue** to continue executing the code.
10. When the application terminates, in Arm Streamline, click 🛑 to stop the capture.

Results

Arm Streamline automatically analyzes the capture data and produces a report, which it displays in the **Timeline** view, as shown in the following screenshot:

Figure 2-9: analysis report for the threads application



3 Controlling Target Execution

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

3.1 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 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 are found.

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 information

Working with breakpoints on page 61
Working with watchpoints on page 62
Conditional breakpoints on page 66

Pending breakpoints and watchpoints on page 73

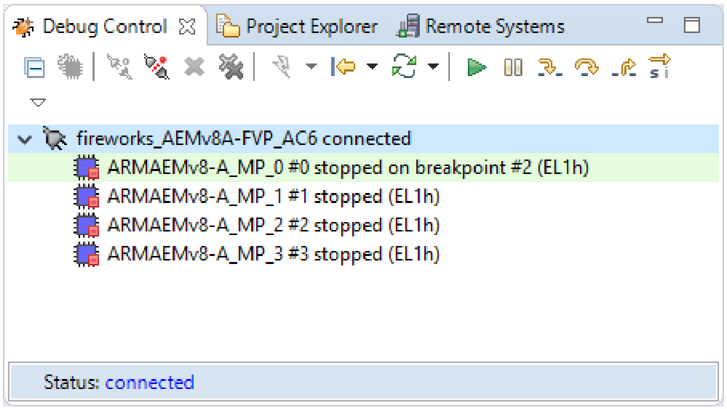
3.2 Running, stopping, and stepping through an application

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 for more information.

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

Figure 3-1: Debug Control view



fireworks_AEMv8A-FVP_AC6 connected
ARMAEMv8-A_MP_0 #0 stopped on breakpoint #2 (EL1h)
ARMAEMv8-A_MP_1 #1 stopped (EL1h)
ARMAEMv8-A_MP_2 #2 stopped (EL1h)
ARMAEMv8-A_MP_3 #3 stopped (EL1h)

Status: connected

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. See Overview: Debugging multi-core (SMP and AMP), big.LITTLE, and multi-cluster targets for more information.

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.

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 for more information.

Related information

Examining the target execution environment on page 120
Examining the call stack on page 121
Handling UNIX signals on page 75
Handling processor exceptions on page 77
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35

3.3 Working with breakpoints

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

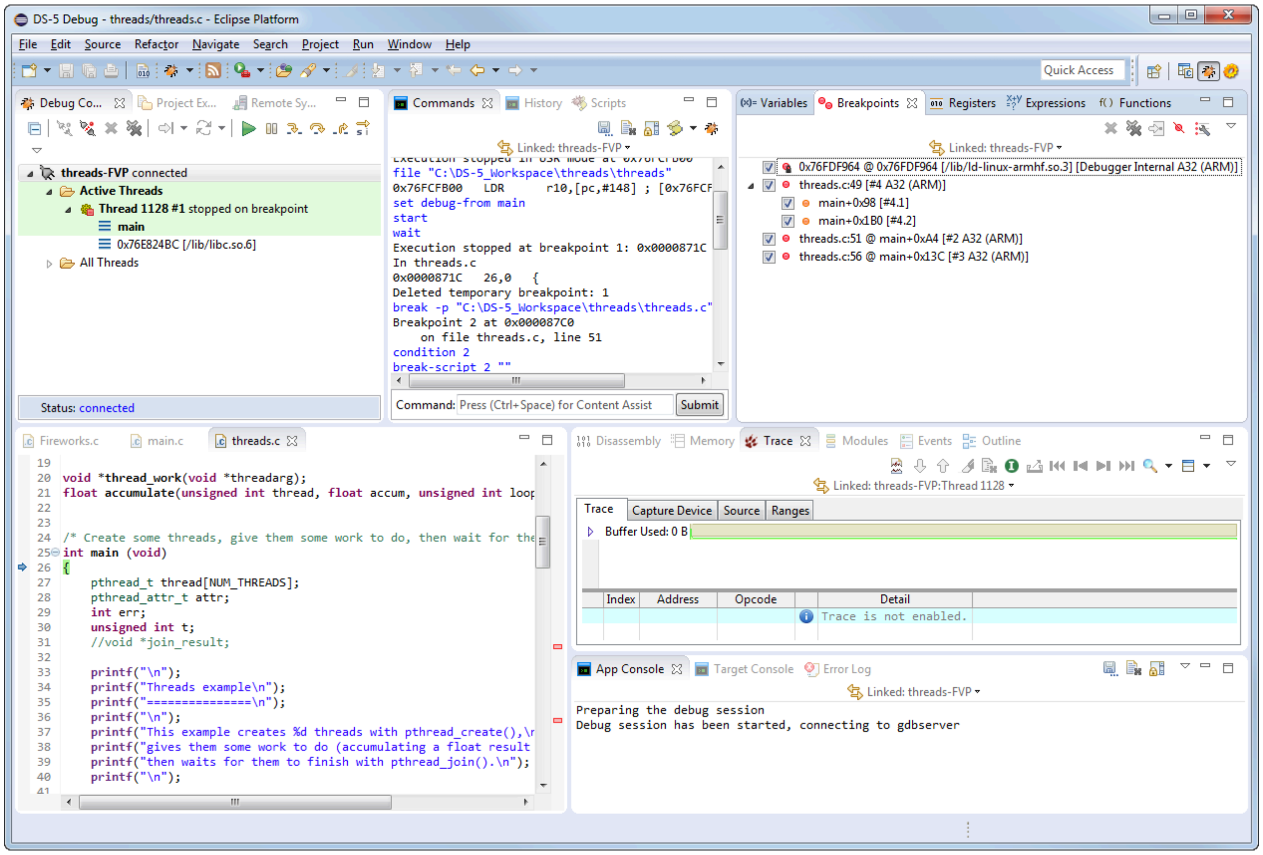
To set a breakpoint, double-click in the left-hand marker bar of the C/C++ editor or the **Disassembly** view at the position where you want to set the breakpoint. See Disassembly view for more information.

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 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 3-2: Viewing breakpoints


DS-5 Debug - threads/threads.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Debug Co... Project Ex... Remote Sy...
Commands History Scripts
Linked: threads-FVP
(x)= Variables Breakpoints Registers Expressions f() Functions
Fireworks.c main.c threads.c
Disassembly Memory Trace Modules Events Outline
Linked: threads-FVP:Thread 1128
App Console Target Console Error Log
Command: Press (Ctrl+Space) for Content Assist
Submit

threads-FVP connected
- Active Threads
  - Thread 1128 #1 stopped on breakpoint
    - main
    - 0x76E824BC [/lib/libc.so.6]
- All Threads

Status: connected

```
Execution stopped in USR mode at 0x76FCFB00
file "C:\DS-5_Workspace\threads\threads"
0x76FCFB00   LDR      r10,[pc,#148] ; [0x76FCF
set debug-from main
start
wait
Execution stopped at breakpoint 1: 0x0000871C
In threads.c
0x0000871C   26,0   {
Deleted temporary breakpoint: 1
break -p "C:\DS-5_Workspace\threads\threads.c"
Breakpoint 2 at 0x000087C0
    on file threads.c, line 51
condition 2
break-script 2 ""
```

- ☑ 0x76FDF964 @ 0x76FDF964 [/lib/ld-linux-armhf.so.3] [Debugger Internal A32 (ARM)]
- ☑ threads.c:49 [#4 A32 (ARM)]
  - ☑ main+0x98 [#4.1]
  - ☑ main+0x1B0 [#4.2]
- ☑ threads.c:51 @ main+0xA4 [#2 A32 (ARM)]
- ☑ threads.c:56 @ main+0x13C [#3 A32 (ARM)]

```
void *thread_work(void *threadarg);
float accumulate(unsigned int thread, float accum, unsigned int loop


/* Create some threads, give them some work to do, then wait for the
int main (void)
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int err;
    unsigned int t;
    //void *join_result;

    printf("\n");
    printf("Threads example\n");
    printf("===============\n");
    printf("\n");
    printf("This example creates %d threads with pthread_create(),\r
    printf("gives them some work to do (accumulating a float result
    printf("then waits for them to finish with pthread_join().\n");
    printf("\n");
```

Trace | Capture Device | Source | Ranges

Buffer Used: 0 B

Index	Address	Opcode	Detail
			Trace is not enabled.

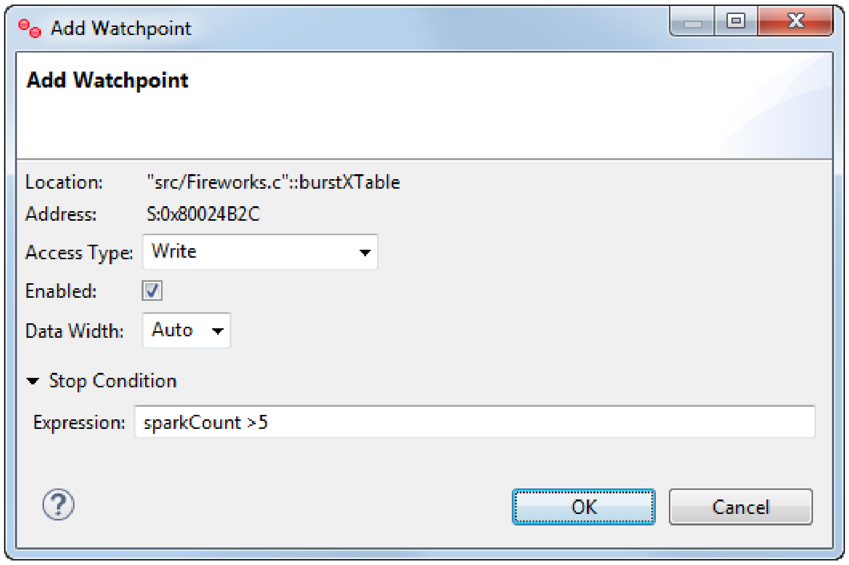
```
Preparing the debug session
Debug session has been started, connecting to gdbserver
```

3.4 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 3-3: Setting a watchpoint on a data symbol



Location: "src/Fireworks.c"::burstXTable
Address: S:0x80024B2C
Access Type: Write
Enabled: ✓
Data Width: Auto
Stop Condition
Expression: sparkCount >5

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.

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 information

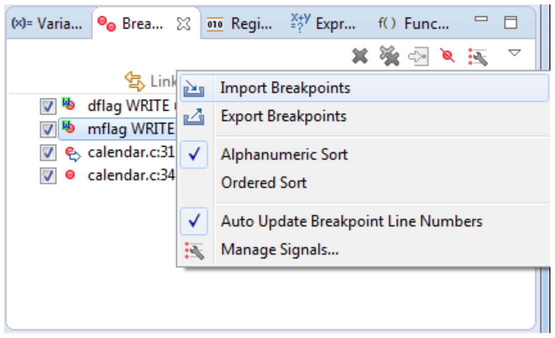
Assigning conditions to an existing watchpoint on page 71
Watchpoint Properties dialog box on page 426
awatch command
rwatch command
watch command
watch set property command

3.5 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 3-4: 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.

3.6 Viewing the properties of a breakpoint or a watchpoint

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

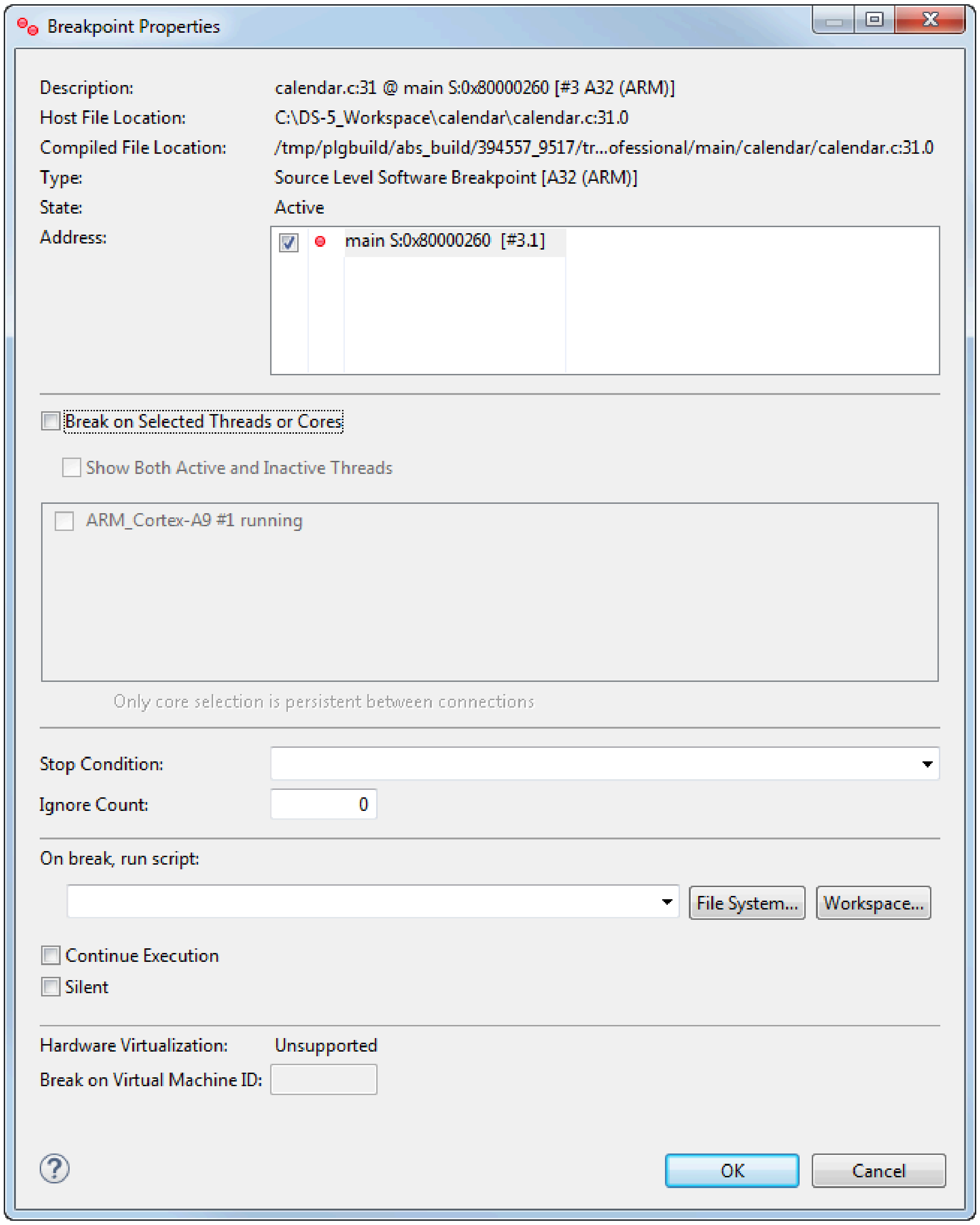
Viewing the properties of a breakpoint

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 3-5: Viewing the properties of a breakpoint



Breakpoint Properties

Description: calendar.c:31 @ main S:0x80000260 [#3 A32 (ARM)]
Host File Location: C:\DS-5_Workspace\calendar\calendar.c:31.0
Compiled File Location: /tmp/plgbuild/abs_build/394557_9517/tr...ofessional/main/calendar/calendar.c:31.0
Type: Source Level Software Breakpoint [A32 (ARM)]
State: Active
Address: [✓] ● main S:0x80000260 [#3.1]

[ ] Break on Selected Threads or Cores
[ ] Show Both Active and Inactive Threads
[ ] ARM_Cortex-A9 #1 running
Only core selection is persistent between connections

Stop Condition:
Ignore Count: 0

On break, run script:
File System... Workspace...
[ ] Continue Execution
[ ] Silent

Hardware Virtualization: Unsupported
Break on Virtual Machine ID:

OK Cancel

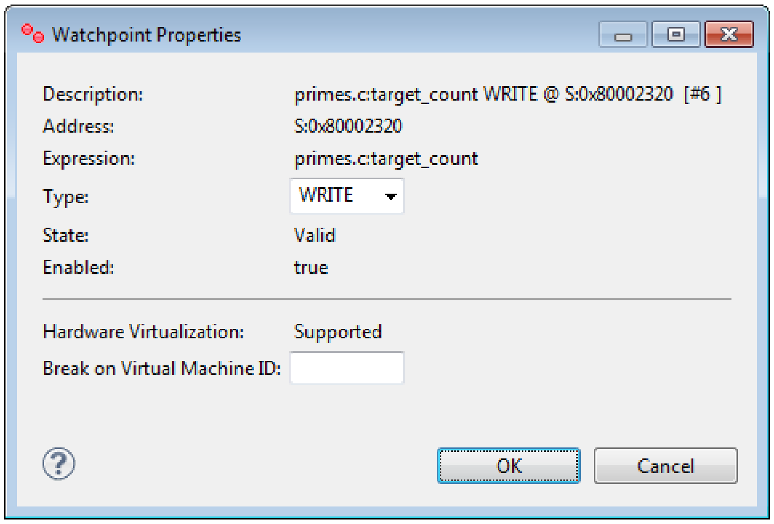
Viewing the properties of a watchpoint

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 3-6: Watchpoint Properties



Description:	primes.c:target_count WRITE @ S:0x80002320 [#6]
Address:	S:0x80002320
Expression:	primes.c:target_count
Type:	WRITE
State:	Valid
Enabled:	true
Hardware Virtualization:	Supported
Break on Virtual Machine ID:	

- 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.

3.7 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 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.
- 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.

3.8 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 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.

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 information

Arm assembler editor on page 319

Breakpoints view on page 322
C/C++ editor on page 326
Commands view on page 329
Disassembly view on page 339
Expressions view on page 350
Memory view on page 359
Registers view on page 376
Variables view on page 408

3.9 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.

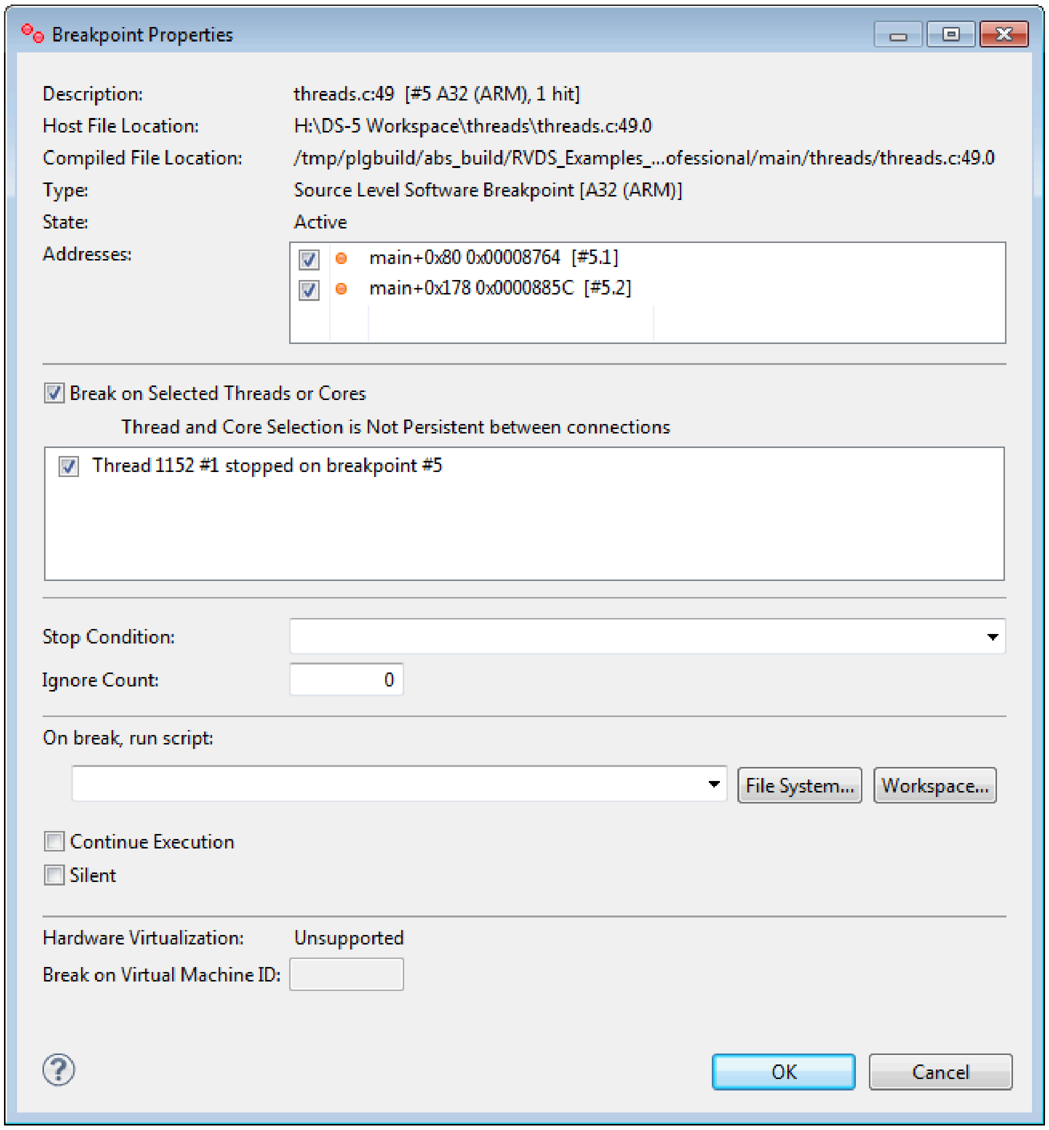
About this task

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 3-7: Breakpoint Properties dialog box



Description:	threads.c:49 [#5 A32 (ARM), 1 hit]
Host File Location:	H:\DS-5 Workspace\threads\threads.c:49.0
Compiled File Location:	/tmp/plgbuild/abs_build/RVDS_Examples_...ofessional/main/threads/threads.c:49.0
Type:	Source Level Software Breakpoint [A32 (ARM)]
State:	Active
Addresses:	☑ main+0x80 0x00008764 [#5.1] ☑ main+0x178 0x0000885C [#5.2]

☑ Break on Selected Threads or Cores

Thread and Core Selection is Not Persistent between connections

☑ Thread 1152 #1 stopped on breakpoint #5

Stop Condition:

Ignore Count: 0

On break, run script:

☐ Continue Execution
☐ Silent

Hardware Virtualization: Unsupported

Break on Virtual Machine ID:

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

Note

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

4. If you want to set a conditional expression for a specific breakpoint, then:
   a) In the **Stop Condition** field, enter a C-style expression. For example, if your application 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 information

Arm assembler editor on page 319
Breakpoints view on page 322
C/C++ editor on page 326
Commands view on page 329
Disassembly view on page 339
Expressions view on page 350
Memory view on page 359
Registers view on page 376
Variables view on page 408

3.10 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)).
- You can set a watchpoint to trigger only when a specific process is running, as defined by the current Context ID. This feature is not available on Arm®v6 processors.

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 for details about assigning a condition to watchpoint when creating it. See Assigning conditions to an existing watchpoint 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.

3.11 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.

About this task

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.

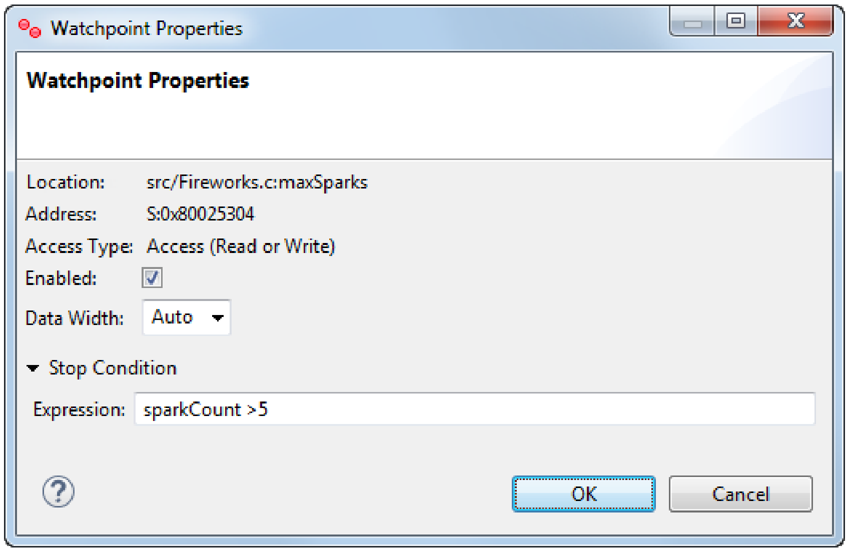
Note

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 3-8: Watchpoint Properties dialog box



Location: src/Fireworks.c:maxSparks
Address: S:0x80025304
Access Type: Access (Read or Write)
Enabled: ✓
Data Width: Auto
Stop Condition
Expression: sparkCount >5

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

Related information

Working with watchpoints on page 62
Watchpoint Properties dialog box on page 426
awatch command
rwatch command
watch command
watch set property command

3.12 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 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 information

Arm assembler editor on page 319
Breakpoints view on page 322
C/C++ editor on page 326
Commands view on page 329
Disassembly view on page 339
Expressions view on page 350
Memory view on page 359
Registers view on page 376
Variables view on page 408

3.13 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.

Note

Trace triggers are not supported on Cortex®-M series processors.

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 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 information

Arm assembler editor on page 319
Breakpoints view on page 322
C/C++ editor on page 326
Commands view on page 329
Disassembly view on page 339
Expressions view on page 350
Memory view on page 359
Registers view on page 376
Variables view on page 408

3.14 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:

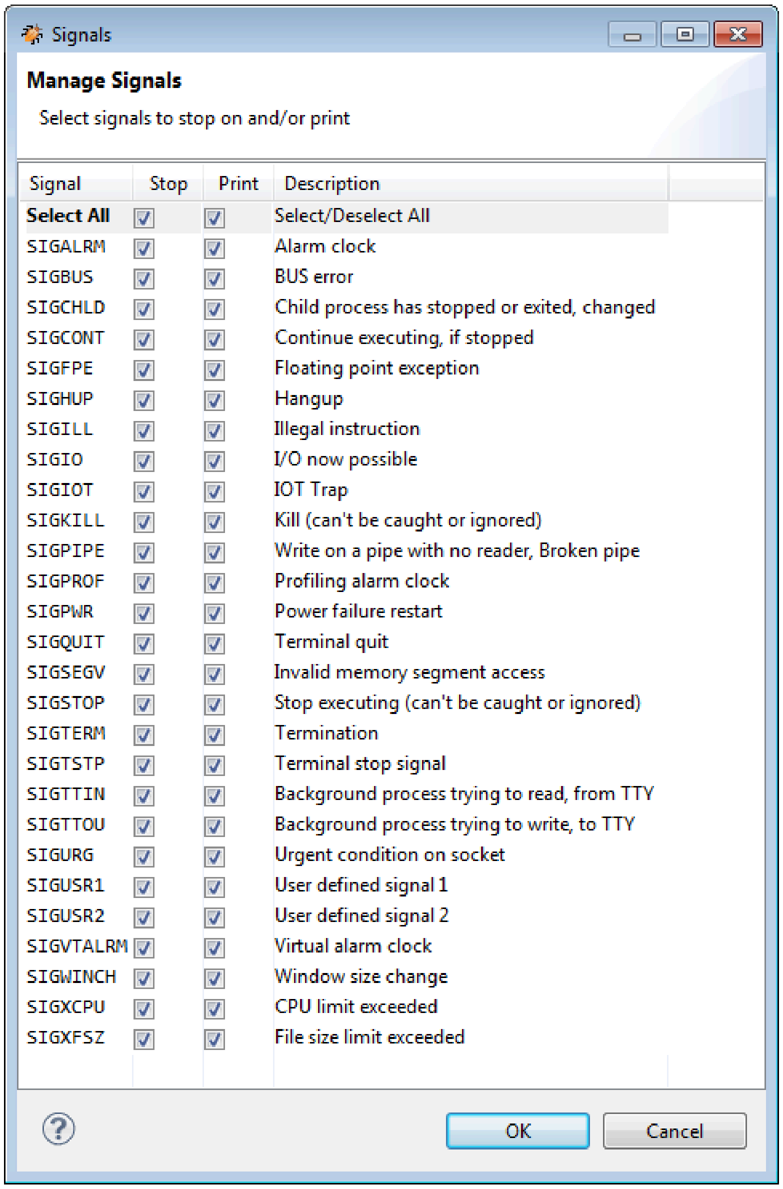
- Select **Manage Signals** from the **Breakpoints** toolbar or the view menu.

  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 3-9: Manage signals dialog box (UNIX signals)



Manage Signals

Select signals to stop on and/or print

Signal	Stop	Print	Description
Select All	✓	✓	Select/Deselect All
SIGALRM	✓	✓	Alarm clock
SIGBUS	✓	✓	BUS error
SIGCHLD	✓	✓	Child process has stopped or exited, changed
SIGCONT	✓	✓	Continue executing, if stopped
SIGFPE	✓	✓	Floating point exception
SIGHUP	✓	✓	Hangup
SIGILL	✓	✓	Illegal instruction
SIGIO	✓	✓	I/O now possible
SIGIOT	✓	✓	IOT Trap
SIGKILL	✓	✓	Kill (can't be caught or ignored)
SIGPIPE	✓	✓	Write on a pipe with no reader, Broken pipe
SIGPROF	✓	✓	Profiling alarm clock
SIGPWR	✓	✓	Power failure restart
SIGQUIT	✓	✓	Terminal quit
SIGSEGV	✓	✓	Invalid memory segment access
SIGSTOP	✓	✓	Stop executing (can't be caught or ignored)
SIGTERM	✓	✓	Termination
SIGTSTP	✓	✓	Terminal stop signal
SIGTTIN	✓	✓	Background process trying to read, from TTY
SIGTTOU	✓	✓	Background process trying to write, to TTY
SIGURG	✓	✓	Urgent condition on socket
SIGUSR1	✓	✓	User defined signal 1
SIGUSR2	✓	✓	User defined signal 2
SIGVTALRM	✓	✓	Virtual alarm clock
SIGWINCH	✓	✓	Window size change
SIGXCPU	✓	✓	CPU limit exceeded
SIGXFSZ	✓	✓	File size limit exceeded

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 information

Running, stopping, and stepping through an application on page 59
Examining the target execution environment on page 120
Examining the call stack on page 121
Handling processor exceptions on page 77
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35
Breakpoints view on page 322
Commands view on page 329
Manage Signals dialog box on page 428
Arm Debugger commands

3.15 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

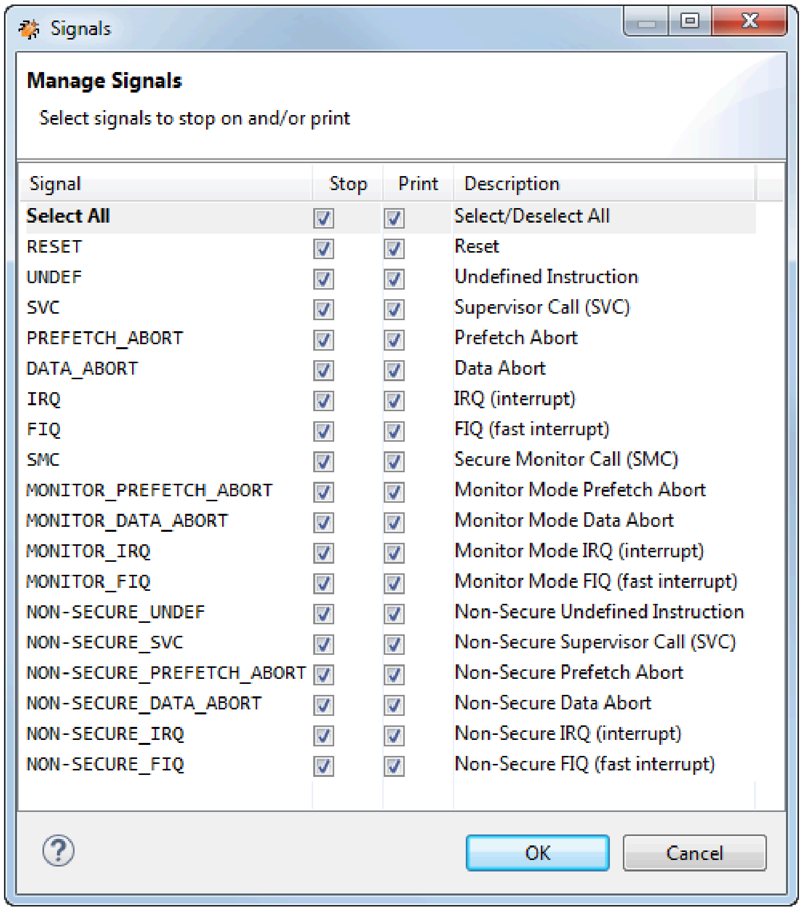
The available vector catch events are dependent on the exact processor that you are connected to.

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 3-10: Manage Signals dialog box



Signal	Stop	Print	Description
Select All	✓	✓	Select/Deselect All
RESET	✓	✓	Reset
UNDEF	✓	✓	Undefined Instruction
SVC	✓	✓	Supervisor Call (SVC)
PREFETCH_ABORT	✓	✓	Prefetch Abort
DATA_ABORT	✓	✓	Data Abort
IRQ	✓	✓	IRQ (interrupt)
FIQ	✓	✓	FIQ (fast interrupt)
SMC	✓	✓	Secure Monitor Call (SMC)
MONITOR_PREFETCH_ABORT	✓	✓	Monitor Mode Prefetch Abort
MONITOR_DATA_ABORT	✓	✓	Monitor Mode Data Abort
MONITOR_IRQ	✓	✓	Monitor Mode IRQ (interrupt)
MONITOR_FIQ	✓	✓	Monitor Mode FIQ (fast interrupt)
NON-SECURE_UNDEF	✓	✓	Non-Secure Undefined Instruction
NON-SECURE_SVC	✓	✓	Non-Secure Supervisor Call (SVC)
NON-SECURE_PREFETCH_ABORT	✓	✓	Non-Secure Prefetch Abort
NON-SECURE_DATA_ABORT	✓	✓	Non-Secure Data Abort
NON-SECURE_IRQ	✓	✓	Non-Secure IRQ (interrupt)
NON-SECURE_FIQ	✓	✓	Non-Secure FIQ (fast interrupt)

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

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 information

Running, stopping, and stepping through an application on page 59
Examining the target execution environment on page 120
Examining the call stack on page 121
Handling UNIX signals on page 75
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35
Breakpoints view on page 322
Commands view on page 329
Manage Signals dialog box on page 428
Arm Debugger commands

3.16 Cross-trigger configuration

In a multiprocessor system, when debug events from one processor are used to affect the debug sessions of other processors, it is called cross-triggering. It is sometimes useful to control all processors with a single debugger command. For example, stopping all cores when a single core hits a breakpoint.

- Hardware cross-triggering

  A hardware cross-triggering mechanism uses the cross-trigger network (composed of Cross Trigger Interface (CTI) and Cross Trigger Matrix (CTM) devices) present in a multiprocessor system. The advantage of using a hardware-based cross-triggering mechanism is low latency performance.
- Software cross-triggering

  In a software cross-triggering scenario, the mechanism is performed and managed by the debugger. Using a software cross-triggering mechanism results in increased latency.

In Arm® Development Studio, the Platform Configuration Editor (PCE) generates support for CTI-synchronized SMP and big.LITTLE debug operations platforms provided that sufficient and appropriate cores, and CTI are available in the SoC. PCE also supports for transporting trace trigger notifications across the cross-trigger network between trace sources and trace sinks.

CTI interfaces need to be programmed using the Debug and Trace Services Layer (DTSL) capabilities in Development Studio. See the DTSL documentation or contact your support representative for more information.

3.17 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.

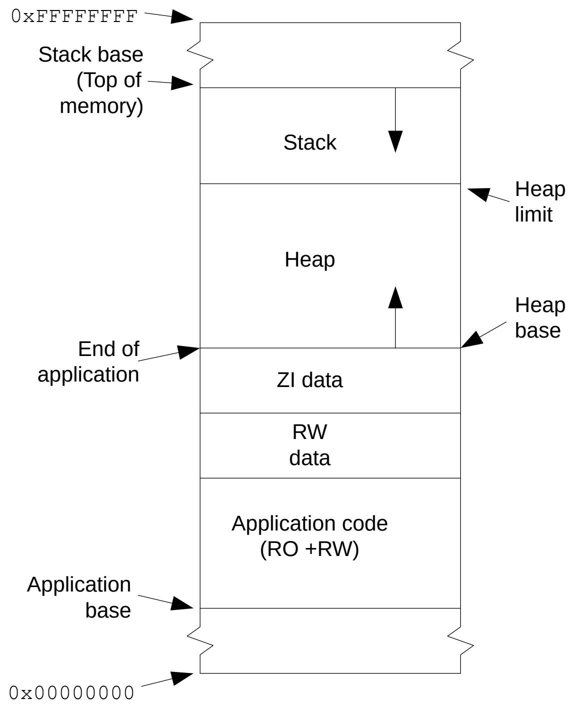
This is useful because development hardware often does not have all the input and output facilities of the final system. Semihosting enables the host computer to provide these facilities.

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 3-11: 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 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.

  See Using the C and C++ libraries with an application in a semihosting environment section in the Arm Compiler for Embedded 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 information

About passing arguments to main() on page 587
Working with semihosting on page 82
Debug Configurations - Arguments tab on page 442
App Console view on page 316
Arm Debugger commands

3.18 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 for more information.

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 information

About passing arguments to main() on page 587
Using semihosting to access resources on the host computer on page 80
Debug Configurations - Arguments tab on page 442
App Console view on page 316
Arm Debugger commands

3.19 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.

About this task

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.

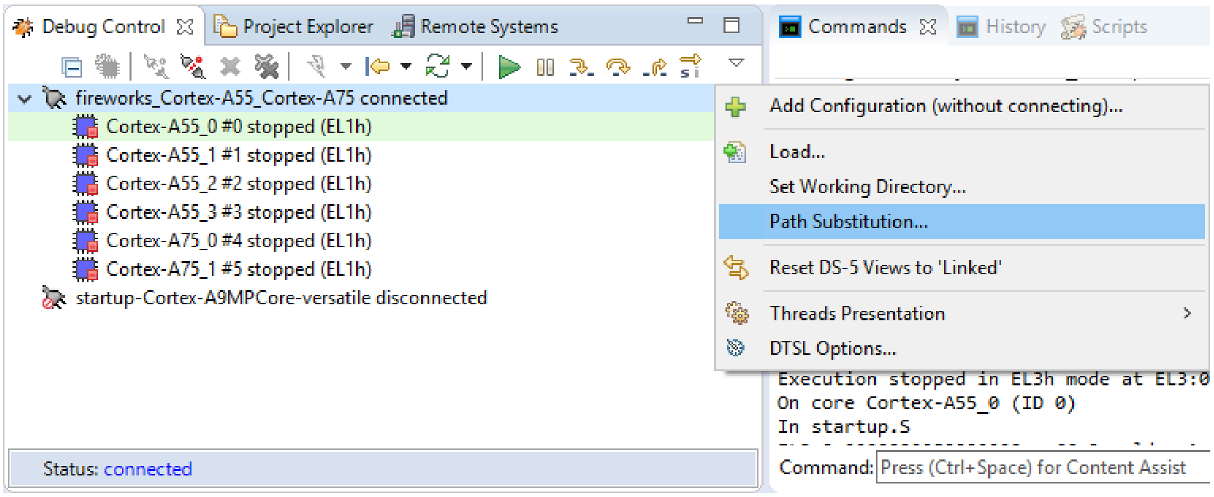
Procedure

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 3-12: Set Path Substitution

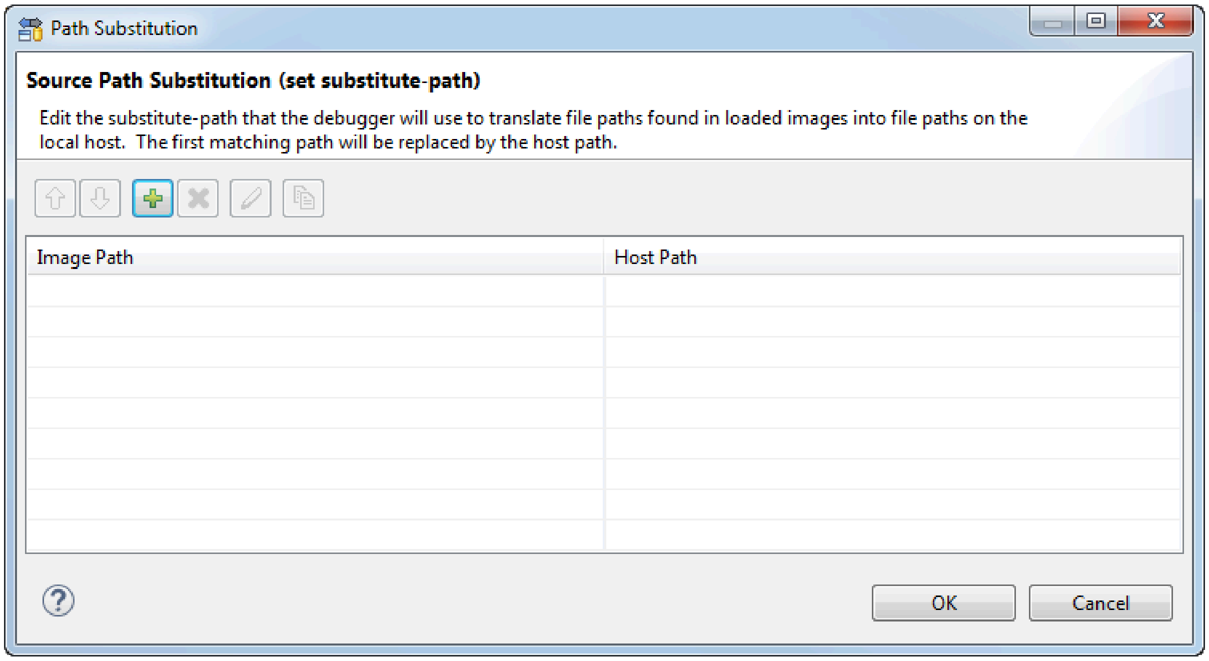


```
fireworks_Cortex-A55_Cortex-A75 connected
    Cortex-A55_0 #0 stopped (EL1h)
    Cortex-A55_1 #1 stopped (EL1h)
    Cortex-A55_2 #2 stopped (EL1h)
    Cortex-A55_3 #3 stopped (EL1h)
    Cortex-A75_0 #4 stopped (EL1h)
    Cortex-A75_1 #5 stopped (EL1h)
startup-Cortex-A9MPCore-versatile disconnected
```

```
Execution stopped in EL3h mode at EL3:0
On core Cortex-A55_0 (ID 0)
In startup.S
```

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

Figure 3-13: Path Substitution dialog box



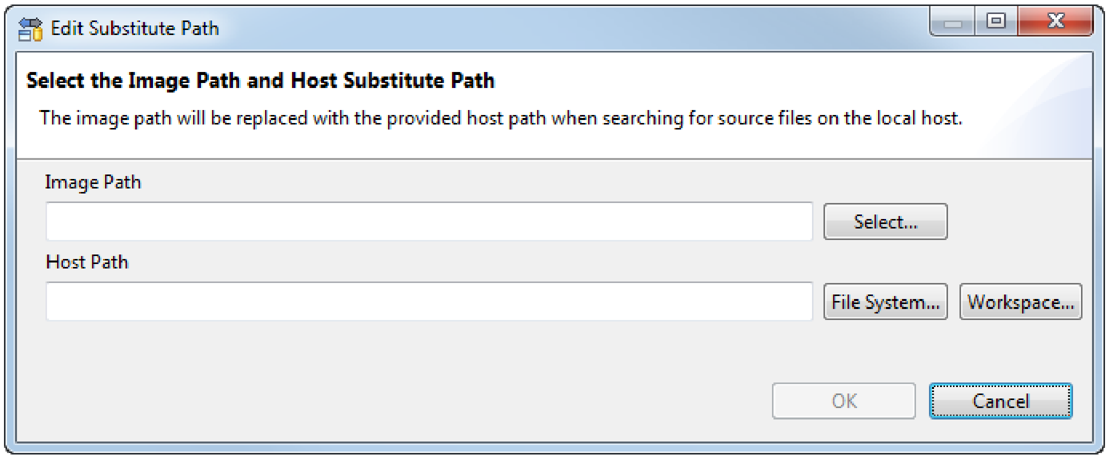
Source Path Substitution (set substitute-path)

Edit the substitute-path that the debugger will use to translate file paths found in loaded images into file paths on the local host. The first matching path will be replaced by the host path.

Image Path	Host Path

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

Figure 3-14: Edit Substitute Path dialog box



Select the Image Path and Host Substitute Path

The image path will be replaced with the provided host path when searching for source files on the local host.

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.
    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 information

About loading debug information into the debugger on page 585

4 Working with the Target Configuration Editor

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

4.1 About the Target Configuration Editor

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

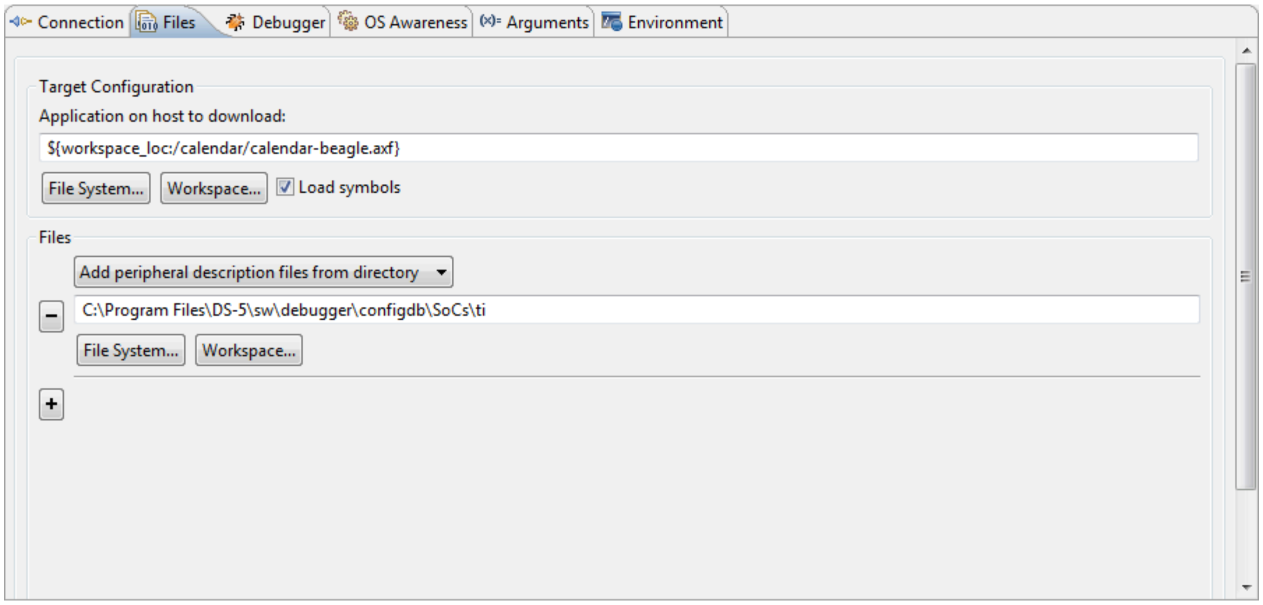
TCF files must have the file extension `.tcf` to invoke this editor.

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

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

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

Figure 4-1: Specifying TCF files in the Debug Configurations window



Target Configuration
Application on host to download:
${workspace_loc:/calendar/calendar-beagle.axf}
☑ Load symbols
Files
Add peripheral description files from directory
C:\Program Files\DS-5\sw\debugger\configdb\SoCs\ti

Related information

Target configuration editor - Overview tab on page 88
Target configuration editor - Memory tab on page 89
Target configuration editor - Peripherals tab on page 92
Target configuration editor - Registers tab on page 94
Target configuration editor - Group View tab on page 96
Target configuration editor - Enumerations tab on page 98
Target configuration editor - Configurations tab on page 98
Debug Configurations - Files tab on page 434
Scenario demonstrating how to create a new target configuration file on page 100

4.2 Target configuration editor - Overview tab

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

General Information

Consists of:

Unique Name

Unique board name (mandatory).

Category

Name of the manufacturer.

Inherits

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

Endianness

Byte order of the target.

TrustZone

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

Power Domain

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

Description

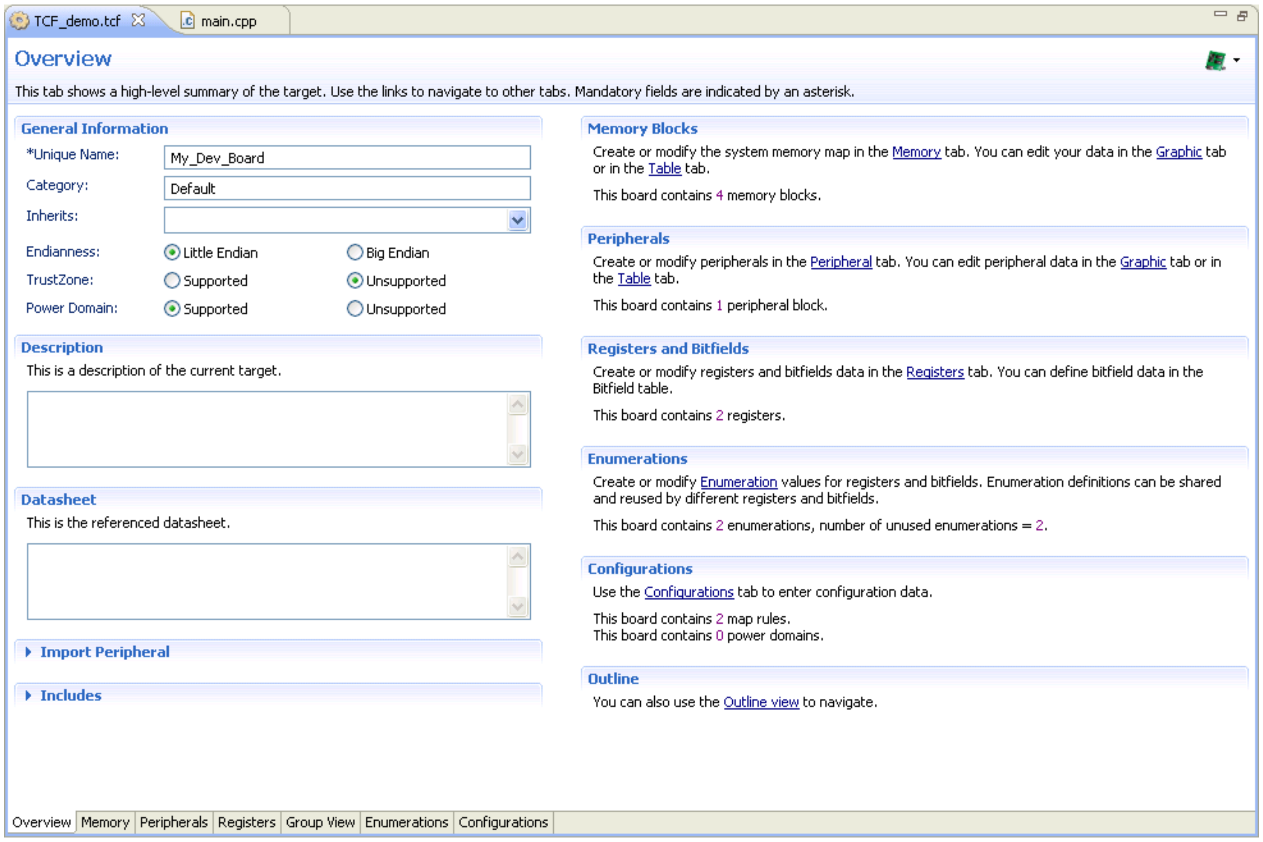
Board description.

Includes

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

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

Figure 4-2: Target configuration editor - Overview tab



Overview

This tab shows a high-level summary of the target. Use the links to navigate to other tabs. Mandatory fields are indicated by an asterisk.

General Information

*Unique Name:	My_Dev_Board	
Category:	Default	
Inherits:		
Endianness:	[✓ Little Endian]	Big Endian
TrustZone:	Supported	[✓ Unsupported]
Power Domain:	[✓ Supported]	Unsupported

Description

This is a description of the current target.

Datasheet

This is the referenced datasheet.

Import Peripheral

Includes

Memory Blocks

Create or modify the system memory map in the Memory tab. You can edit your data in the Graphic tab or in the Table tab.

This board contains 4 memory blocks.

Peripherals

Create or modify peripherals in the Peripheral tab. You can edit peripheral data in the Graphic tab or in the Table tab.

This board contains 1 peripheral block.

Registers and Bitfields

Create or modify registers and bitfields data in the Registers tab. You can define bitfield data in the Bitfield table.

This board contains 2 registers.

Enumerations

Create or modify Enumeration values for registers and bitfields. Enumeration definitions can be shared and reused by different registers and bitfields.

This board contains 2 enumerations, number of unused enumerations = 2.

Configurations

Use the Configurations tab to enter configuration data.

This board contains 2 map rules.
This board contains 0 power domains.

Outline

You can also use the Outline view to navigate.

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

Related information

Target configuration editor - Memory tab on page 89
Target configuration editor - Peripherals tab on page 92
Target configuration editor - Registers tab on page 94
Target configuration editor - Group View tab on page 96
Target configuration editor - Enumerations tab on page 98
Target configuration editor - Configurations tab on page 98
About the Target Configuration Editor on page 87
Scenario demonstrating how to create a new target configuration file on page 100
Creating a power domain for a target on page 112

4.3 Target configuration editor - Memory tab

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

Graphical view

In the graphical view, the following options are available:

View by Map Rule

Filter the graphical view based on the selected rule.

View by Address Type

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

View by Power Domain

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

Add button

Add a new memory region.

Remove button

Remove the selected memory region.

Graphical and tabular views

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

Unique Name

Name of the selected memory region (mandatory).

Name

User-friendly name for the selected memory region.

Description

Detailed description of the selected memory region.

Base Address

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

Offset

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

Size

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

Width

Access width of the selected memory region.

Access

Access mode for the selected memory region.

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

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

More... (tabular view)

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

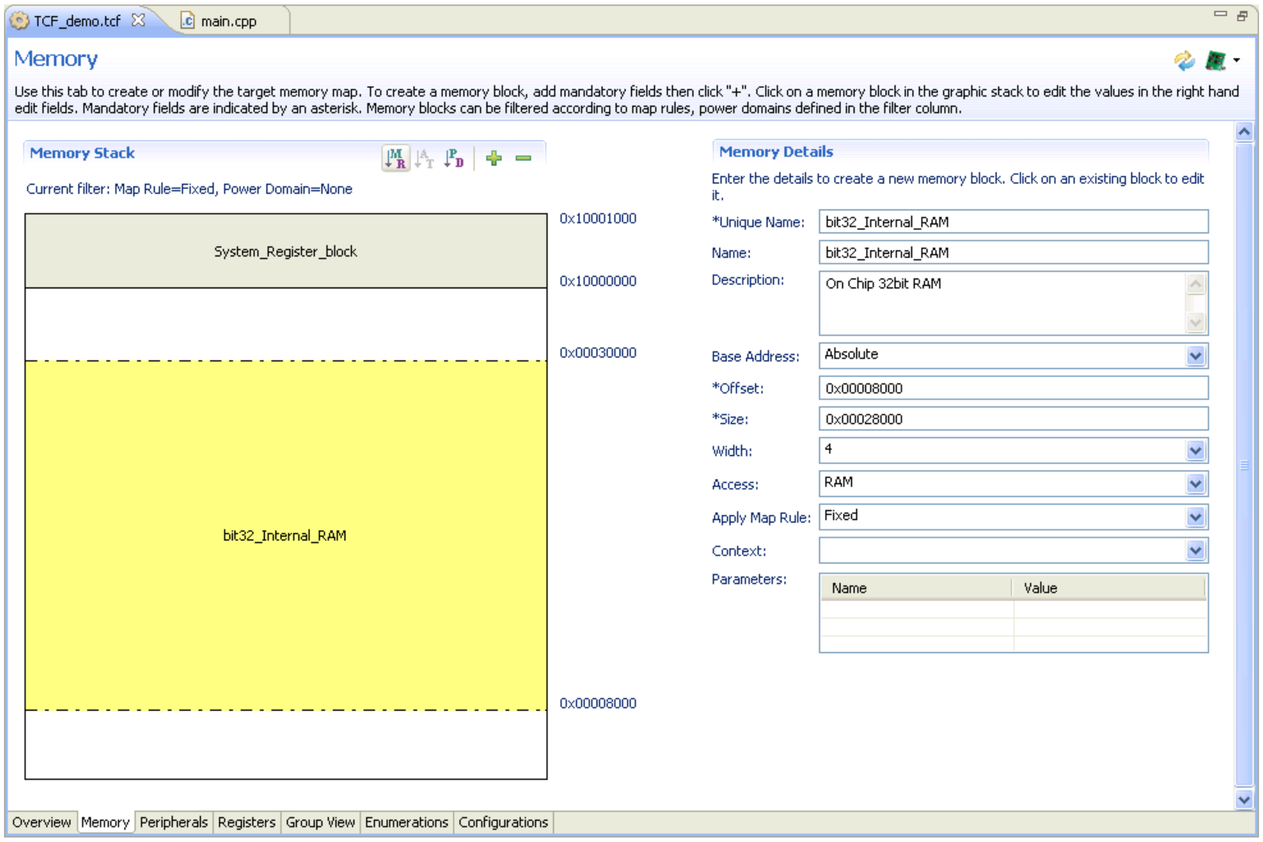
Context

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

Parameters

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

Figure 4-3: Target configuration editor - Memory tab



Memory

Use this tab to create or modify the target memory map. To create a memory block, add mandatory fields then click "+". Click on a memory block in the graphic stack to edit the values in the right hand edit fields. Mandatory fields are indicated by an asterisk. Memory blocks can be filtered according to map rules, power domains defined in the filter column.

Current filter: Map Rule=Fixed, Power Domain=None

Enter the details to create a new memory block. Click on an existing block to edit it.

*Unique Name:	bit32_Internal_RAM
Name:	bit32_Internal_RAM
Description:	On Chip 32bit RAM
Base Address:	Absolute
*Offset:	0x00008000
*Size:	0x00028000
Width:	4
Access:	RAM
Apply Map Rule:	Fixed
Context:	

Parameters:

Name	Value

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

Related information

Target configuration editor - Overview tab on page 88
Target configuration editor - Peripherals tab on page 92

Target configuration editor - Registers tab on page 94
Target configuration editor - Group View tab on page 96
Target configuration editor - Enumerations tab on page 98
Target configuration editor - Configurations tab on page 98
About the Target Configuration Editor on page 87
Creating a memory map on page 102
Creating a memory region for remapping by a control register on page 109
Applying the map rules to the overlapping memory regions on page 110

4.4 Target configuration editor - Peripherals tab

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

Graphical view

In the graphical view, the following options are available:

View by Address Type

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

View by Power Domain

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

Add button

Add a new peripheral.

Remove button

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

Graphical and tabular views

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

Unique Name

Name of the selected peripheral (mandatory).

Name

User-friendly name for the selected peripheral.

Description

Detailed description of the selected peripheral.

Base Address

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

Offset

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

Size

Size of the selected peripheral in bytes.

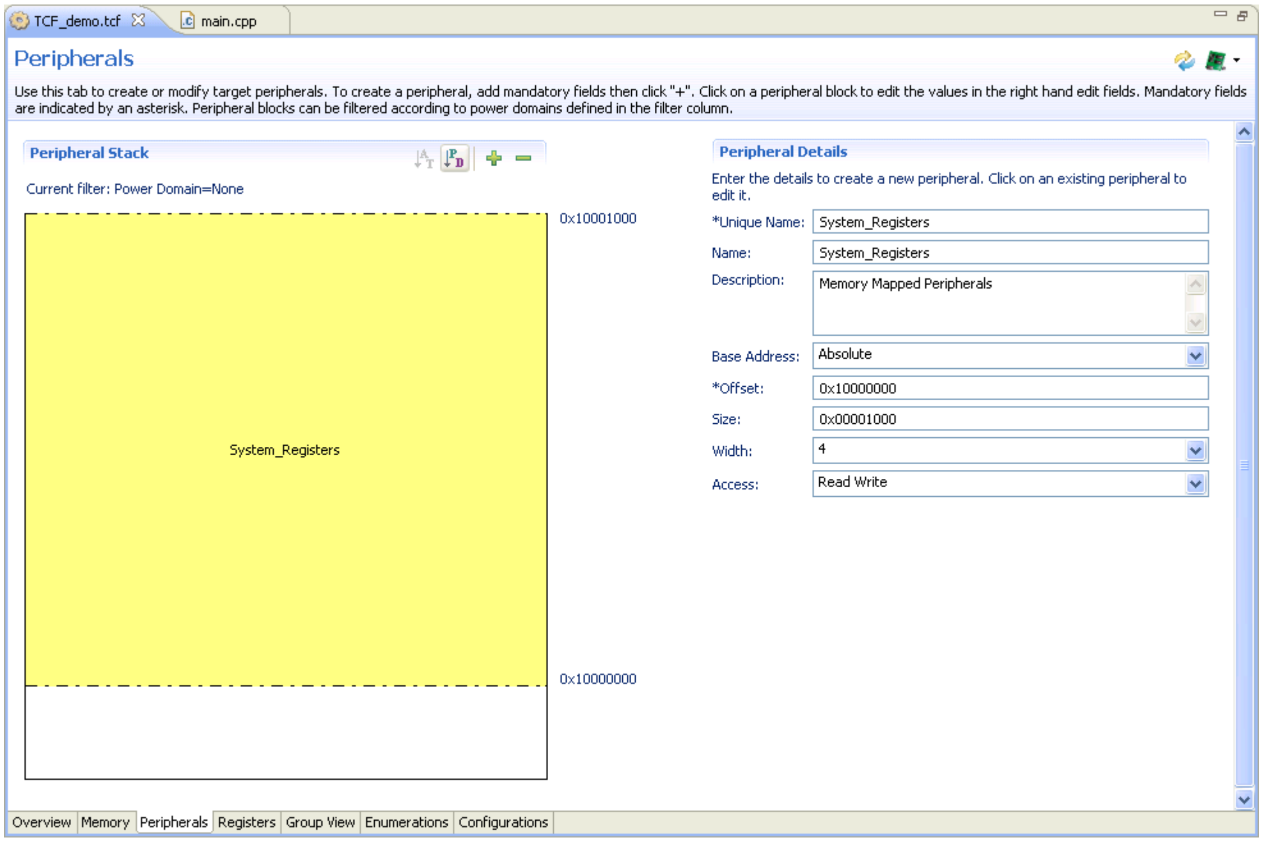
Width

Access width of the selected peripheral in bytes

Access

Access mode for the selected peripheral.

Figure 4-4: Target configuration editor - Peripherals tab



Peripherals

Use this tab to create or modify target peripherals. To create a peripheral, add mandatory fields then click "+". Click on a peripheral block to edit the values in the right hand edit fields. Mandatory fields are indicated by an asterisk. Peripheral blocks can be filtered according to power domains defined in the filter column.

Peripheral Stack

Current filter: Power Domain=None

Peripheral Details

Enter the details to create a new peripheral. Click on an existing peripheral to edit it.

*Unique Name:	System_Registers
Name:	System_Registers
Description:	Memory Mapped Peripherals
Base Address:	Absolute
*Offset:	0x10000000
Size:	0x00001000
Width:	4
Access:	Read Write

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

Related information

Target configuration editor - Overview tab on page 88
Target configuration editor - Memory tab on page 89
Target configuration editor - Registers tab on page 94
Target configuration editor - Group View tab on page 96
Target configuration editor - Enumerations tab on page 98
Target configuration editor - Configurations tab on page 98

About the Target Configuration Editor on page 87
Creating a peripheral on page 103

4.5 Target configuration editor - Registers tab

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

Unique Name
Name of the register (mandatory).

Name
User-friendly name for the register.

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

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

Size
Size of the register in bytes (mandatory).

Access size
Access width of the register in bytes.

Access
Access mode for the selected register.

Description
Detailed description of the register.

Peripheral
Associated peripheral, if applicable.

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

Unique Name
Name of the selected bitfield (mandatory).

Name
User-friendly name for the selected bitfield.

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

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

Access

Access mode for the selected bitfield.

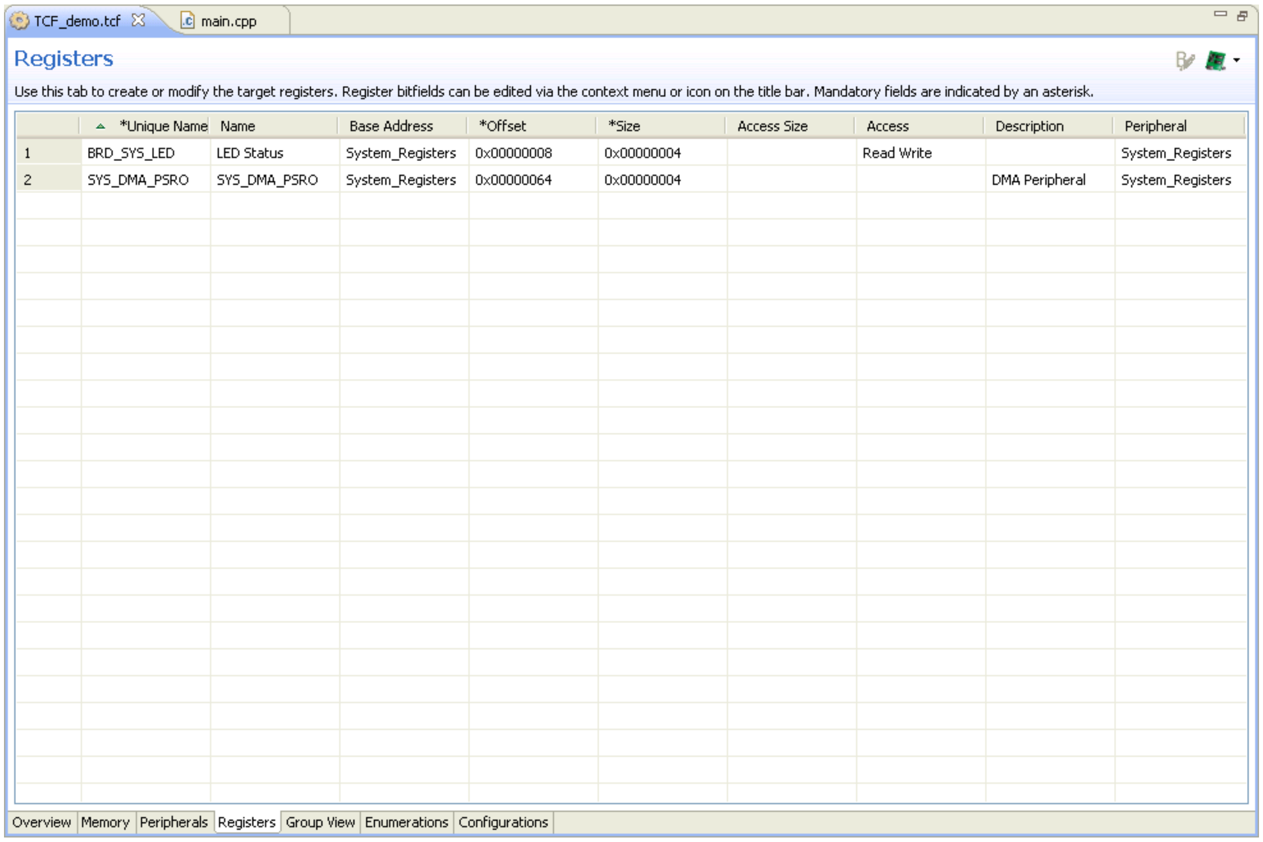
Description

Detailed description of the selected bitfield.

Enumeration

Associated enumeration for the selected bitfield, if applicable.

Figure 4-5: Target configuration editor - Registers tab



Use this tab to create or modify the target registers. Register bitfields can be edited via the context menu or icon on the title bar. Mandatory fields are indicated by an asterisk.

	*Unique Name	Name	Base Address	*Offset	*Size	Access Size	Access	Description	Peripheral
1	BRD_SYS_LED	LED Status	System_Registers	0x00000008	0x00000004		Read Write		System_Registers
2	SYS_DMA_PSRO	SYS_DMA_PSRO	System_Registers	0x00000064	0x00000004			DMA Peripheral	System_Registers

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

Related information

Target configuration editor - Overview tab on page 88
Target configuration editor - Memory tab on page 89
Target configuration editor - Peripherals tab on page 92
Target configuration editor - Group View tab on page 96
Target configuration editor - Enumerations tab on page 98
Target configuration editor - Configurations tab on page 98
About the Target Configuration Editor on page 87
Creating a standalone register on page 104

Creating a peripheral register on page 105
Assigning enumerations to a peripheral register on page 107
Creating enumerations for use with a peripheral register on page 106

4.6 Target configuration editor - Group View tab

A list view that enables you to select peripherals to be used by the debugger.

Group View List

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

Add a new group

Creates a group that you can personalize with peripherals.

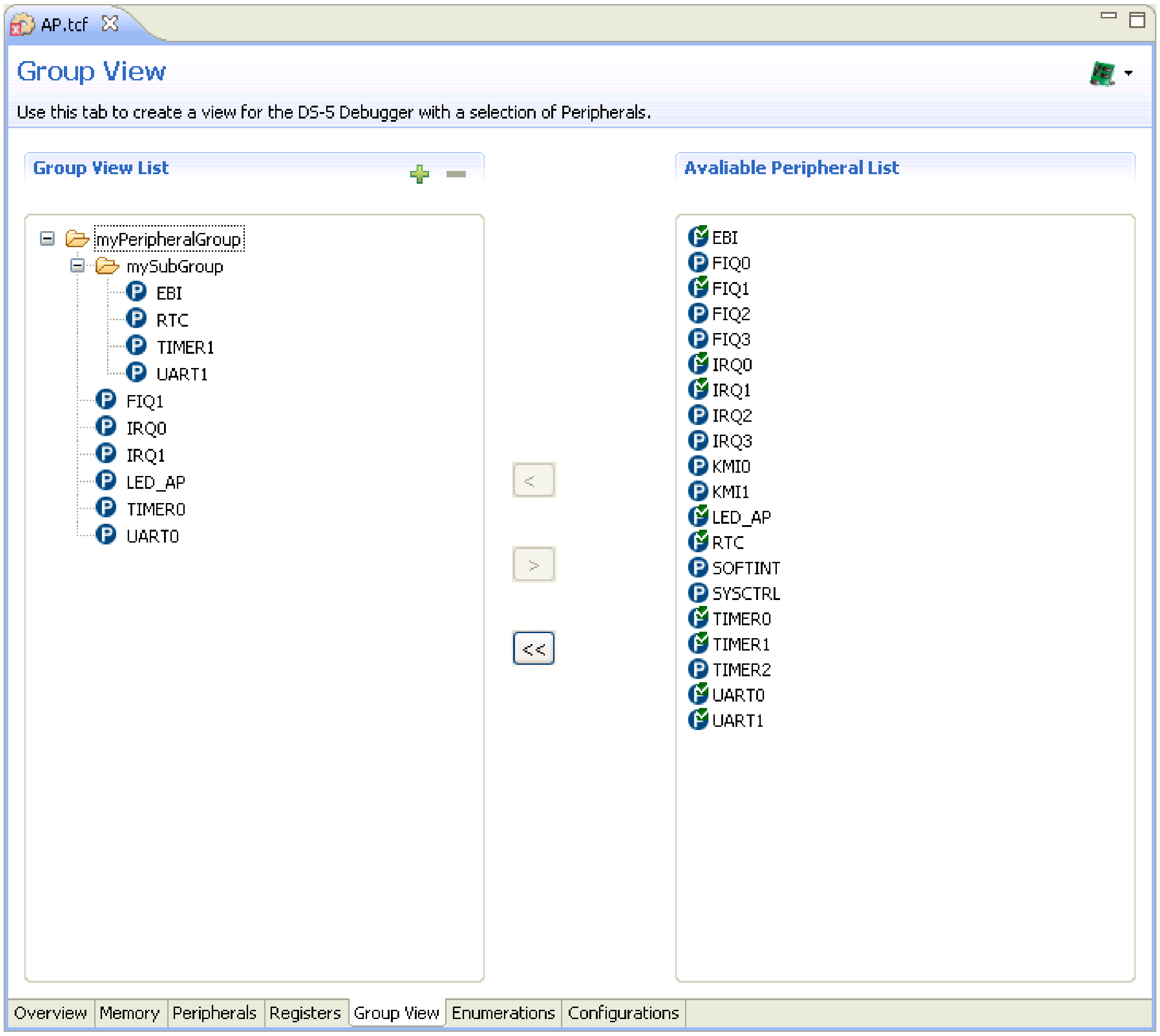
Remove the selected group

Removes a group from the list.

Available Peripheral List

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

Figure 4-6: Target configuration editor - Group View tab



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

Related information

Target configuration editor - Overview tab on page 88
Target configuration editor - Memory tab on page 89
Target configuration editor - Peripherals tab on page 92
Target configuration editor - Registers tab on page 94
Target configuration editor - Enumerations tab on page 98
Target configuration editor - Configurations tab on page 98
About the Target Configuration Editor on page 87
Creating a Group list on page 113

4.7 Target configuration editor - Enumerations tab

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

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

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

The following settings are available:

Unique Name

Name of the selected enumeration (mandatory).

Value

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

Description

Detailed description of the selected enumeration.

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

Related information

Target configuration editor - Overview tab on page 88
Target configuration editor - Memory tab on page 89
Target configuration editor - Peripherals tab on page 92
Target configuration editor - Registers tab on page 94
Target configuration editor - Group View tab on page 96
Target configuration editor - Configurations tab on page 98
About the Target Configuration Editor on page 87
Creating enumerations for use with a peripheral register on page 106
Assigning enumerations to a peripheral register on page 107

4.8 Target configuration editor - Configurations tab

A tabular view that enables you to:

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

Memory Map Configurations group

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

Unique Name
Name of the rule (mandatory).

Name
User-friendly name for the rule.

Register
Associated control register (mandatory).

Mask
Mask value (mandatory).

Value
Value for a condition (mandatory).

Trigger
Condition that changes the control register mapping (mandatory).

Power Domain Configurations group

The **Power Domain Configurations** group

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

Unique Name
Name of the power domain.

Wake-up Conditions
User-friendly name for the rule:

Register
An associated control register that you have previously created.

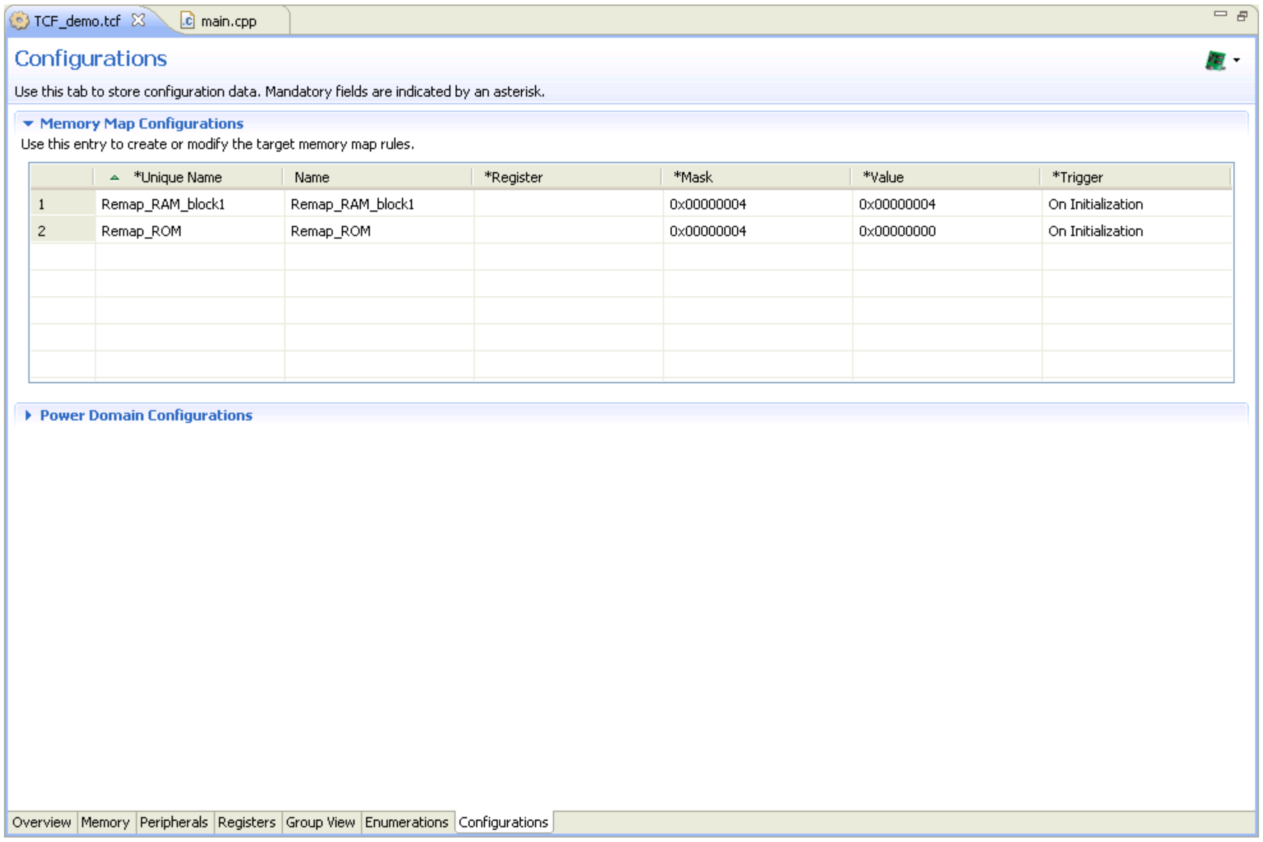
Mask
Mask value.

Value
Value for a condition.

Power State
The power state of the power domain:

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

Figure 4-7: Target configuration editor - Configuration tab



Use this tab to store configuration data. Mandatory fields are indicated by an asterisk.

Use this entry to create or modify the target memory map rules.

	*Unique Name	Name	*Register	*Mask	*Value	*Trigger
1	Remap_RAM_block1	Remap_RAM_block1		0x00000004	0x00000004	On Initialization
2	Remap_ROM	Remap_ROM		0x00000004	0x00000000	On Initialization

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

Related information

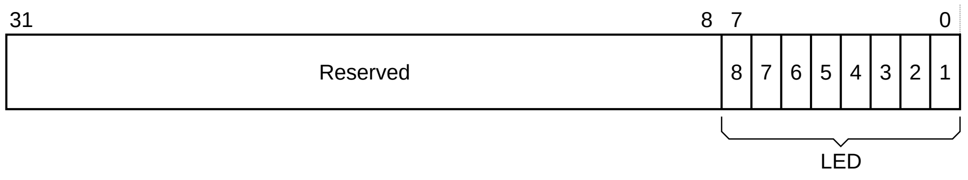
Target configuration editor - Overview tab on page 88
Target configuration editor - Memory tab on page 89
Target configuration editor - Peripherals tab on page 92
Target configuration editor - Registers tab on page 94
Target configuration editor - Group View tab on page 96
Target configuration editor - Enumerations tab on page 98
About the Target Configuration Editor on page 87
Creating a standalone register on page 104
Creating remapping rules for a control register on page 108
Creating a power domain for a target on page 112

4.9 Scenario demonstrating how to create a new target configuration file

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

- Boot ROM: `0x0 - 0x8000`
- SRAM: `0x0 - 0x8000`
- Internal RAM: `0x8000 - 0x28000`
- System Registers that contain memory mapped peripherals: `0x10000000 - 0x10001000`
  - A basic standalone LED register. This register is located at `0x10000008` and is used to write a hexadecimal value that sets the corresponding bits to 1 to illuminate the respective LEDs.

Figure 4-8: LED register and bitfields



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

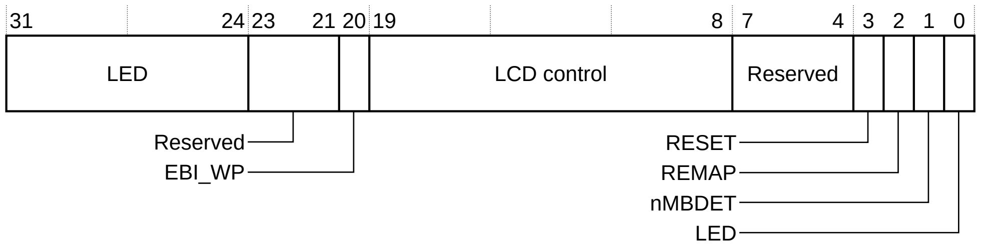
Table 4-1: DMA map register SYS_DMAPSR0

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

Bits [`4:0`]	Read/Write	FPGA peripheral mapped to this channel
		`b00000 = AACI Tx`
		`b00001 = AACI Rx`
		`b00010 = USB A`
		`b00011 = USB B`
		`b00100 = MCI 0`

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

Figure 4-9: Core module and LCD control register



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

Table 4-2: Control bit that remaps an area of memory

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

- Clearing bit 2 (CM_CTRL = 0) generates the following memory map:
    - `0x0000 - 0x8000 Boot_ROM`
    - `0x8000 - 0x28000 32bit_RAM`
- Setting bit 2 (CM_CTRL = 1) generates the following memory map:
    - `0x0000 - 0x8000 32bit_RAM_block1_alias`
    - `0x8000 - 0x28000 32bit_RAM`

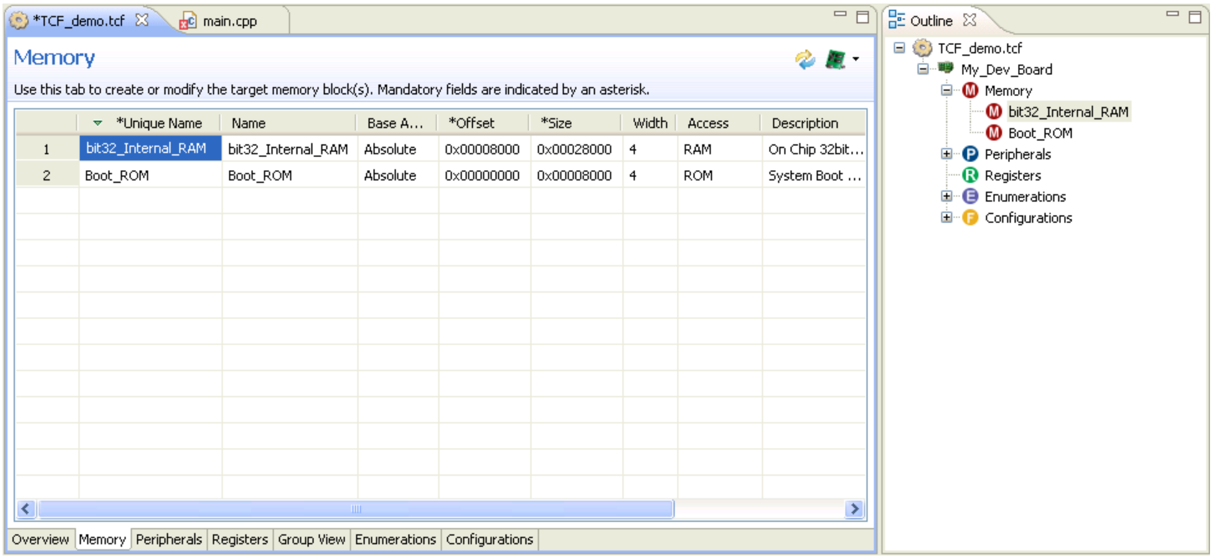
4.9.1 Creating a memory map

Describes how to create a new memory map.

Procedure

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

Figure 4-10: Creating a Memory map



Memory

Use this tab to create or modify the target memory block(s). Mandatory fields are indicated by an asterisk.

	*Unique Name	Name	Base A...	*Offset	*Size	Width	Access	Description
1	bit32_Internal_RAM	bit32_Internal_RAM	Absolute	0x00008000	0x00028000	4	RAM	On Chip 32bit...
2	Boot_ROM	Boot_ROM	Absolute	0x00000000	0x00008000	4	ROM	System Boot ...

Results

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

Related information

Creating a peripheral on page 103
Creating a standalone register on page 104
Creating a peripheral register on page 105
Creating enumerations for use with a peripheral register on page 106
Assigning enumerations to a peripheral register on page 107
Creating remapping rules for a control register on page 108
Creating a memory region for remapping by a control register on page 109
Applying the map rules to the overlapping memory regions on page 110
Target configuration editor - Memory tab on page 89

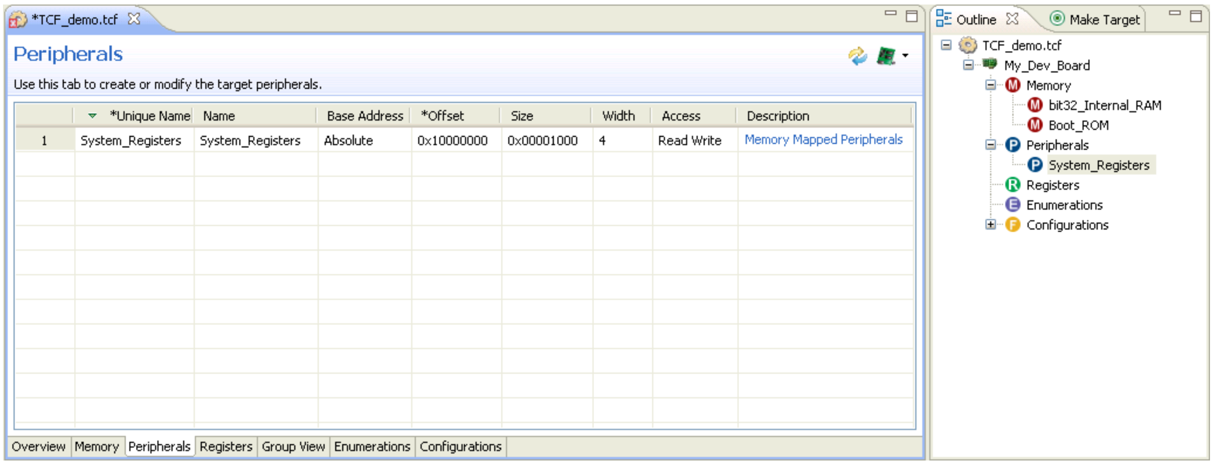
4.9.2 Creating a peripheral

Describes how to create a peripheral.

Procedure

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

Figure 4-11: Creating a peripheral



Use this tab to create or modify the target peripherals.

	*Unique Name	Name	Base Address	*Offset	Size	Width	Access	Description
1	System_Registers	System_Registers	Absolute	0x10000000	0x00001000	4	Read Write	Memory Mapped Peripherals

Related information

Creating a memory map on page 102
Creating a standalone register on page 104
Creating a peripheral register on page 105
Creating enumerations for use with a peripheral register on page 106
Assigning enumerations to a peripheral register on page 107
Creating remapping rules for a control register on page 108
Creating a memory region for remapping by a control register on page 109
Applying the map rules to the overlapping memory regions on page 110
Target configuration editor - Peripherals tab on page 92

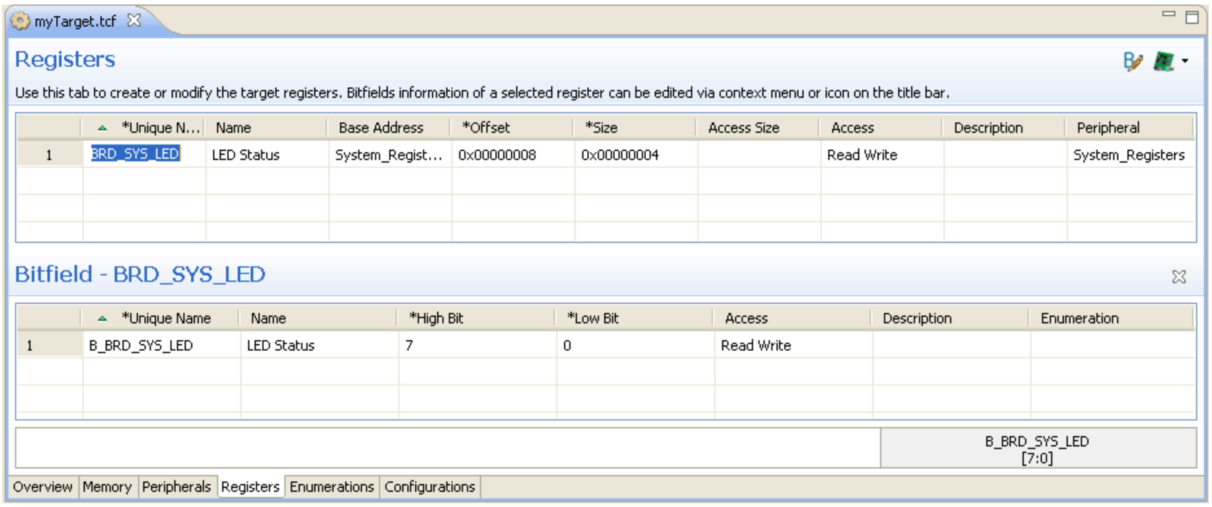
4.9.3 Creating a standalone register

Describes how to create a basic standalone register.

Procedure

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

Figure 4-12: Creating a standalone register



Registers

Use this tab to create or modify the target registers. Bitfields information of a selected register can be edited via context menu or icon on the title bar.

	*Unique N...	Name	Base Address	*Offset	*Size	Access Size	Access	Description	Peripheral
1	BRD_SYS_LED	LED Status	System_Regist...	0x00000008	0x00000004		Read Write		System_Registers

Bitfield - BRD_SYS_LED

	*Unique Name	Name	*High Bit	*Low Bit	Access	Description	Enumeration
1	B_BRD_SYS_LED	LED Status	7	0	Read Write		

B_BRD_SYS_LED
[7:0]

6. On completion, close the floating table.

Related information

Creating a memory map on page 102
Creating a peripheral on page 103
Creating a peripheral register on page 105
Creating enumerations for use with a peripheral register on page 106
Assigning enumerations to a peripheral register on page 107
Creating remapping rules for a control register on page 108
Creating a memory region for remapping by a control register on page 109
Applying the map rules to the overlapping memory regions on page 110
Target configuration editor - Registers tab on page 94
Target configuration editor - Configurations tab on page 98

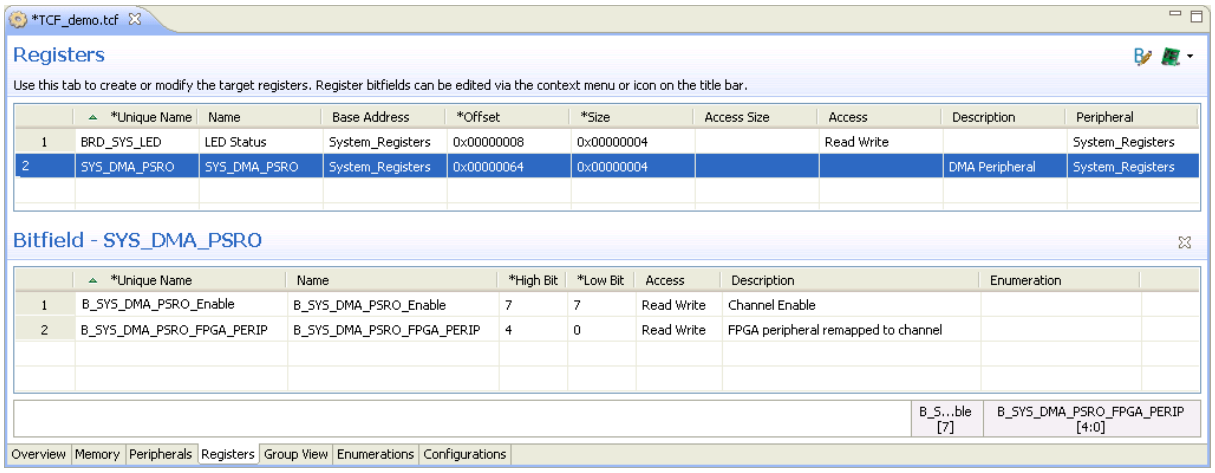
4.9.4 Creating a peripheral register

Describes how to create a peripheral register.

Procedure

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

Figure 4-13: Creating a peripheral register



Registers

Use this tab to create or modify the target registers. Register bitfields can be edited via the context menu or icon on the title bar.

	*Unique Name	Name	Base Address	*Offset	*Size	Access Size	Access	Description	Peripheral
1	BRD_SYS_LED	LED Status	System_Registers	0x00000008	0x00000004		Read Write		System_Registers
2	SYS_DMA_PSRO	SYS_DMA_PSRO	System_Registers	0x00000064	0x00000004			DMA Peripheral	System_Registers

Bitfield - SYS_DMA_PSRO

	*Unique Name	Name	*High Bit	*Low Bit	Access	Description	Enumeration
1	B_SYS_DMA_PSRO_Enable	B_SYS_DMA_PSRO_Enable	7	7	Read Write	Channel Enable	
2	B_SYS_DMA_PSRO_FPGA_PERIP	B_SYS_DMA_PSRO_FPGA_PERIP	4	0	Read Write	FPGA peripheral remapped to channel	

B_S...ble [7] | B_SYS_DMA_PSRO_FPGA_PERIP [4:0]

Related information

Creating a memory map on page 102
Creating a peripheral on page 103
Creating a standalone register on page 104
Creating enumerations for use with a peripheral register on page 106
Assigning enumerations to a peripheral register on page 107
Creating remapping rules for a control register on page 108
Creating a memory region for remapping by a control register on page 109
Applying the map rules to the overlapping memory regions on page 110
Target configuration editor - Registers tab on page 94

4.9.5 Creating enumerations for use with a peripheral register

Describes how to create enumerations for use with a peripheral.

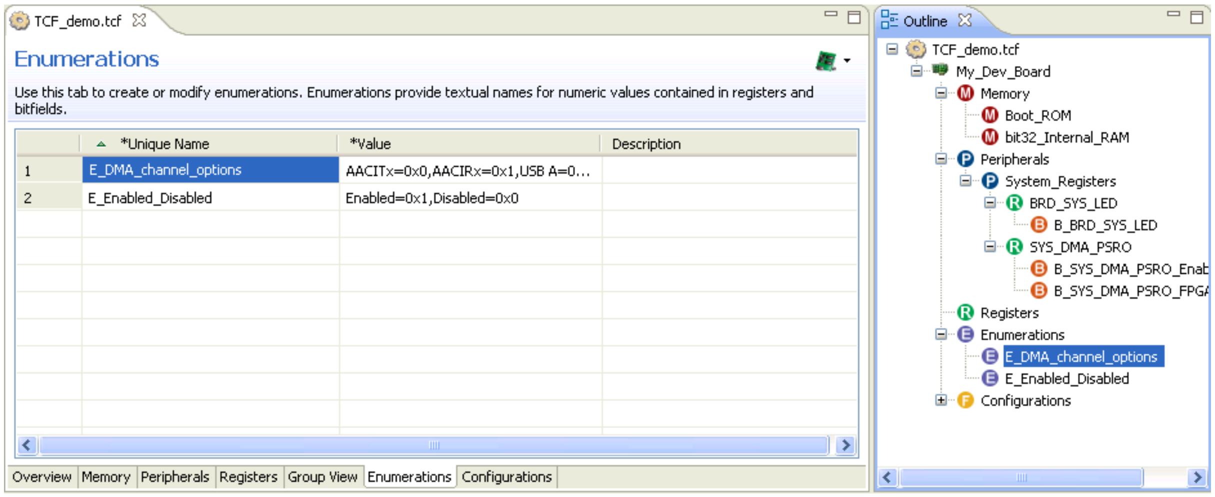
About this task

Enumerations are textual names for numeric values. For more complex peripherals, you might find it useful to create enumerations for particular peripheral bit patterns. This means that you can assign a value to a peripheral by selecting from a list of enumerated values, rather than write the equivalent hexadecimal value. (For example: Enabled/Disabled, On/Off).

Procedure

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

Figure 4-14: Creating enumerations



Enumerations

Use this tab to create or modify enumerations. Enumerations provide textual names for numeric values contained in registers and bitfields.

	*Unique Name	*Value	Description
1	E_DMA_channel_options	AACITx=0x0,AACIRx=0x1,USB A=0...	
2	E_Enabled_Disabled	Enabled=0x1,Disabled=0x0	

```
TCF_demo.tcf
  My_Dev_Board
    Memory
      Boot_ROM
      bit32_Internal_RAM
    Peripherals
      System_Registers
        BRD_SYS_LED
          B_BRD_SYS_LED
        SYS_DMA_PSRO
          B_SYS_DMA_PSRO_Enab
          B_SYS_DMA_PSRO_FPGA
    Registers
    Enumerations
      E_DMA_channel_options
      E_Enabled_Disabled
    Configurations
```

Related information

Creating a memory map on page 102
Creating a peripheral on page 103
Creating a standalone register on page 104
Creating a peripheral register on page 105
Assigning enumerations to a peripheral register on page 107
Creating remapping rules for a control register on page 108
Creating a memory region for remapping by a control register on page 109
Applying the map rules to the overlapping memory regions on page 110
Target configuration editor - Registers tab on page 94
Target configuration editor - Enumerations tab on page 98

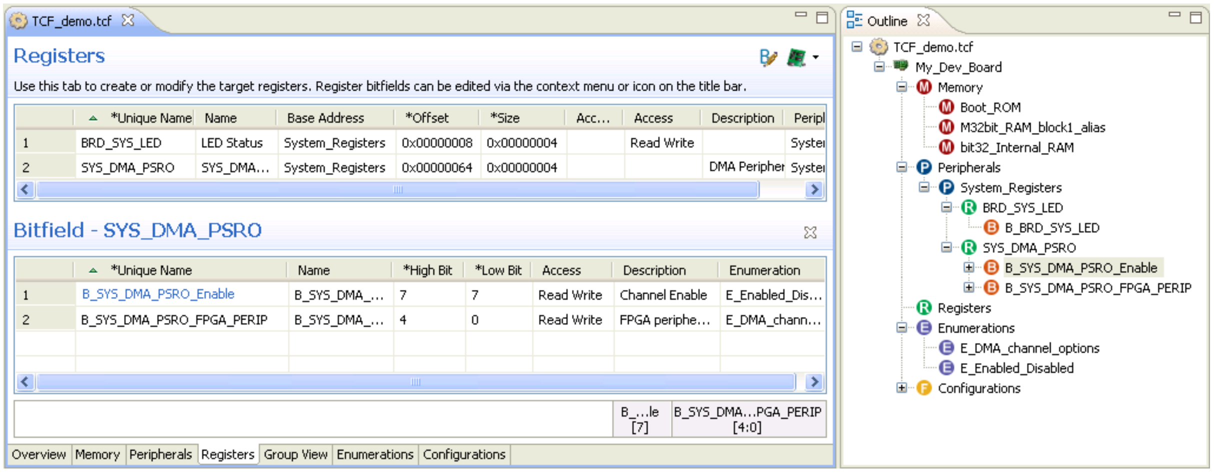
4.9.6 Assigning enumerations to a peripheral register

Describes how to assign enumerations to a peripheral register.

Procedure

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

Figure 4-15: Assigning enumerations


TCF_demo.tcf

Registers

Use this tab to create or modify the target registers. Register bitfields can be edited via the context menu or icon on the title bar.

	*Unique Name	Name	Base Address	*Offset	*Size	Acc…	Access	Description	Peripl
1	BRD_SYS_LED	LED Status	System_Registers	0x00000008	0x00000004		Read Write		Syste
2	SYS_DMA_PSRO	SYS_DMA…	System_Registers	0x00000064	0x00000004			DMA Peripher	Syste

Bitfield - SYS_DMA_PSRO

	*Unique Name	Name	*High Bit	*Low Bit	Access	Description	Enumeration
1	B_SYS_DMA_PSRO_Enable	B_SYS_DMA_…	7	7	Read Write	Channel Enable	E_Enabled_Dis…
2	B_SYS_DMA_PSRO_FPGA_PERIP	B_SYS_DMA_…	4	0	Read Write	FPGA periphe…	E_DMA_chann…

B_…le [7] | B_SYS_DMA…PGA_PERIP [4:0]

Related information

Creating a memory map on page 102
Creating a peripheral on page 103
Creating a standalone register on page 104
Creating a peripheral register on page 105
Creating enumerations for use with a peripheral register on page 106
Creating remapping rules for a control register on page 108
Creating a memory region for remapping by a control register on page 109
Applying the map rules to the overlapping memory regions on page 110
Target configuration editor - Registers tab on page 94
Target configuration editor - Enumerations tab on page 98

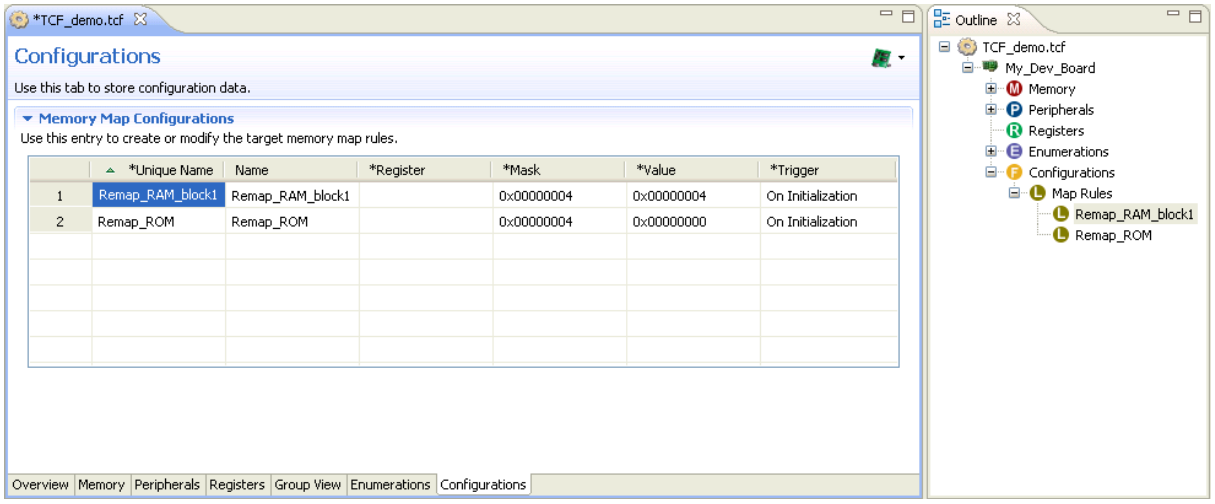
4.9.7 Creating remapping rules for a control register

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

Procedure

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

Figure 4-16: Creating remapping rules



	*Unique Name	Name	*Register	*Mask	*Value	*Trigger
1	Remap_RAM_block1	Remap_RAM_block1		0x00000004	0x00000004	On Initialization
2	Remap_ROM	Remap_ROM		0x00000004	0x00000000	On Initialization

Related information

Creating a memory map on page 102
Creating a peripheral on page 103
Creating a standalone register on page 104
Creating a peripheral register on page 105
Creating enumerations for use with a peripheral register on page 106
Assigning enumerations to a peripheral register on page 107
Creating a memory region for remapping by a control register on page 109
Applying the map rules to the overlapping memory regions on page 110
Target configuration editor - Configurations tab on page 98

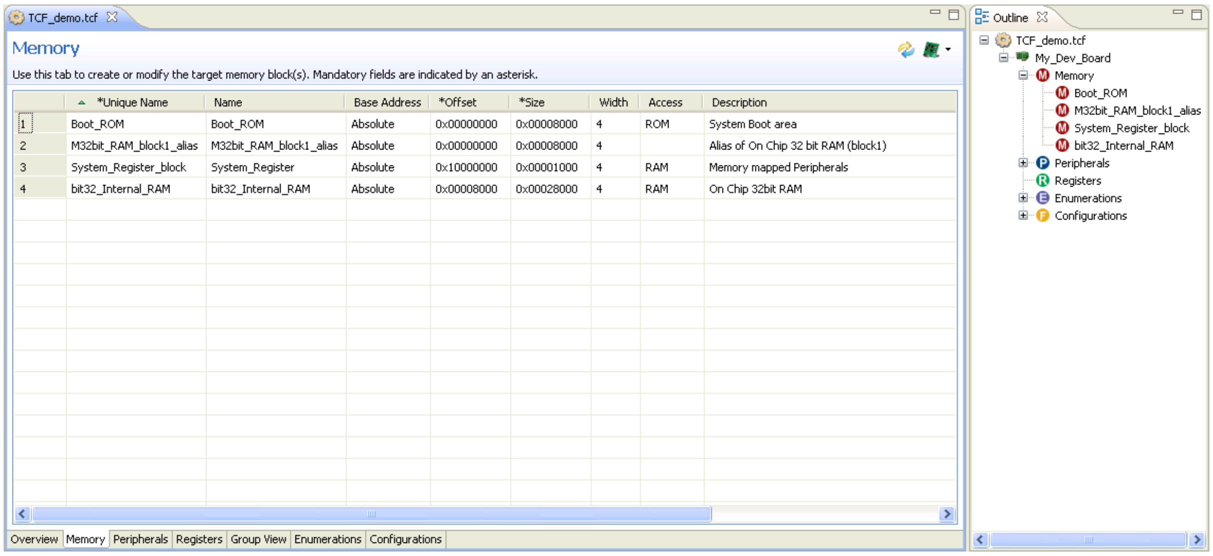
4.9.8 Creating a memory region for remapping by a control register

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

Procedure

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

Figure 4-17: Creating a memory region for remapping by a control register



Use this tab to create or modify the target memory block(s). Mandatory fields are indicated by an asterisk.

	*Unique Name	Name	Base Address	*Offset	*Size	Width	Access	Description
1	Boot_ROM	Boot_ROM	Absolute	0x00000000	0x00008000	4	ROM	System Boot area
2	M32bit_RAM_block1_alias	M32bit_RAM_block1_alias	Absolute	0x00000000	0x00008000	4		Alias of On Chip 32 bit RAM (block1)
3	System_Register_block	System_Register	Absolute	0x10000000	0x00001000	4	RAM	Memory mapped Peripherals
4	bit32_Internal_RAM	bit32_Internal_RAM	Absolute	0x00008000	0x00028000	4	RAM	On Chip 32bit RAM

Related information

Creating a memory map on page 102
Creating a peripheral on page 103
Creating a standalone register on page 104
Creating a peripheral register on page 105
Creating enumerations for use with a peripheral register on page 106
Assigning enumerations to a peripheral register on page 107
Creating remapping rules for a control register on page 108
Applying the map rules to the overlapping memory regions on page 110
Target configuration editor - Memory tab on page 89

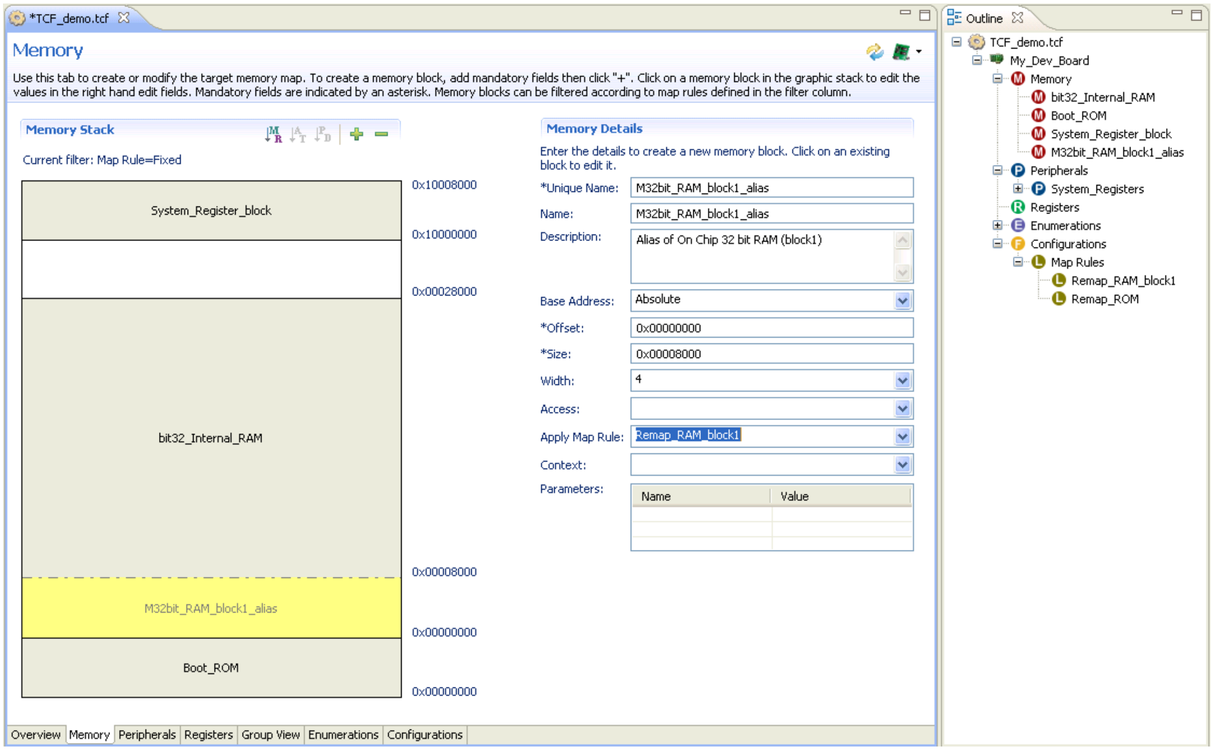
4.9.9 Applying the map rules to the overlapping memory regions

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

Procedure

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

Figure 4-18: Applying the Remap_RAM_block1 map rule



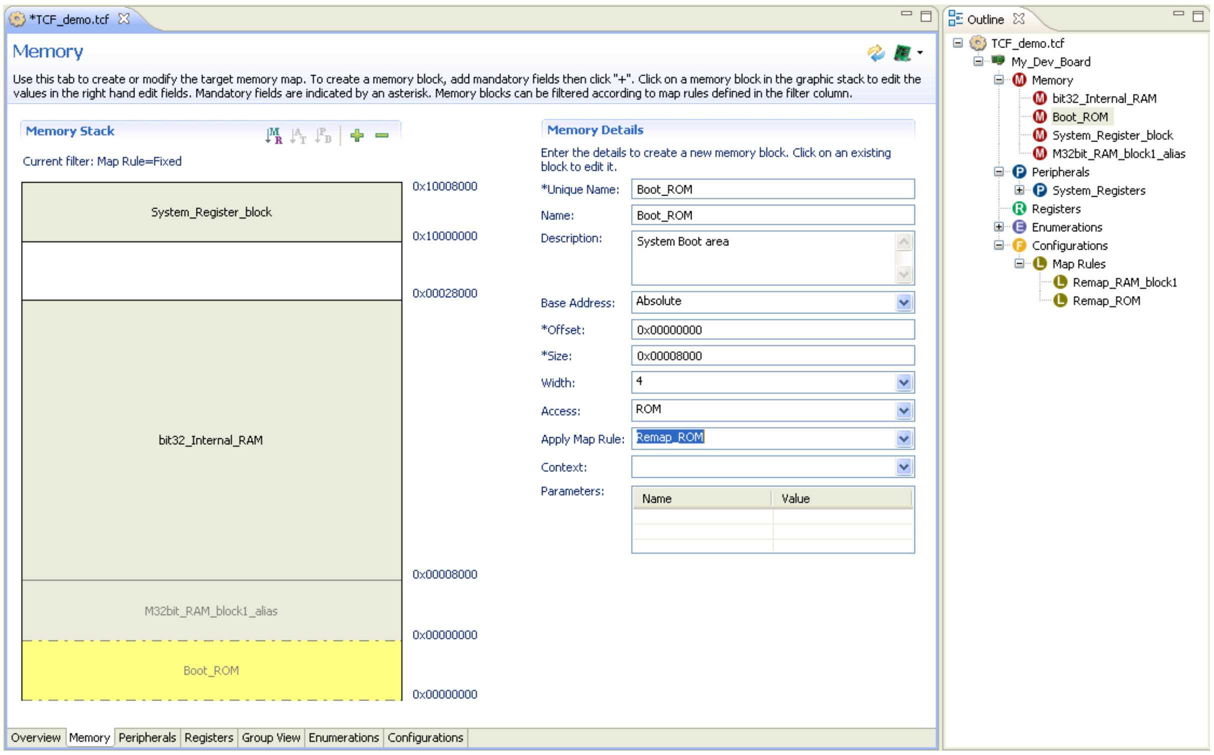
Use this tab to create or modify the target memory map. To create a memory block, add mandatory fields then click "+". Click on a memory block in the graphic stack to edit the values in the right hand edit fields. Mandatory fields are indicated by an asterisk. Memory blocks can be filtered according to map rules defined in the filter column.

Enter the details to create a new memory block. Click on an existing block to edit it.

*Unique Name:	M32bit_RAM_block1_alias
Name:	M32bit_RAM_block1_alias
Description:	Alias of On Chip 32 bit RAM (block1)
Base Address:	Absolute
*Offset:	0x00000000
*Size:	0x00008000
Width:	4
Access:	
Apply Map Rule:	Remap_RAM_block1
Context:	
Parameters:	Name / Value

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

Figure 4-19: Applying the Remap_ROM map rule



*TCF_demo.tcf

Memory

Use this tab to create or modify the target memory map. To create a memory block, add mandatory fields then click "+". Click on a memory block in the graphic stack to edit the values in the right hand edit fields. Mandatory fields are indicated by an asterisk. Memory blocks can be filtered according to map rules defined in the filter column.

Memory Stack

Current filter: Map Rule=Fixed

Block	Address
	0x10008000
System_Register_block	
	0x10000000
	0x00028000
bit32_Internal_RAM	
	0x00008000
M32bit_RAM_block1_alias	
	0x00000000
Boot_ROM	
	0x00000000

Memory Details

Enter the details to create a new memory block. Click on an existing block to edit it.

*Unique Name:	Boot_ROM
Name:	Boot_ROM
Description:	System Boot area
Base Address:	Absolute
*Offset:	0x00000000
*Size:	0x00008000
Width:	4
Access:	ROM
Apply Map Rule:	Remap_ROM
Context:	
Parameters:	Name / Value

Overview | Memory | Peripherals | Registers | Group View | Enumerations | Configurations

Outline

- TCF_demo.tcf
  - My_Dev_Board
    - Memory
      - bit32_Internal_RAM
      - Boot_ROM
      - System_Register_block
      - M32bit_RAM_block1_alias
    - Peripherals
      - System_Registers
    - Registers
    - Enumerations
    - Configurations
      - Map Rules
        - Remap_RAM_block1
        - Remap_ROM

5. Save the file.

Related information

Creating a memory map on page 102
Creating a peripheral on page 103
Creating a standalone register on page 104
Creating a peripheral register on page 105
Creating enumerations for use with a peripheral register on page 106
Assigning enumerations to a peripheral register on page 107
Creating remapping rules for a control register on page 108
Creating a memory region for remapping by a control register on page 109
Target configuration editor - Memory tab on page 89

4.10 Creating a power domain for a target

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

Before you begin

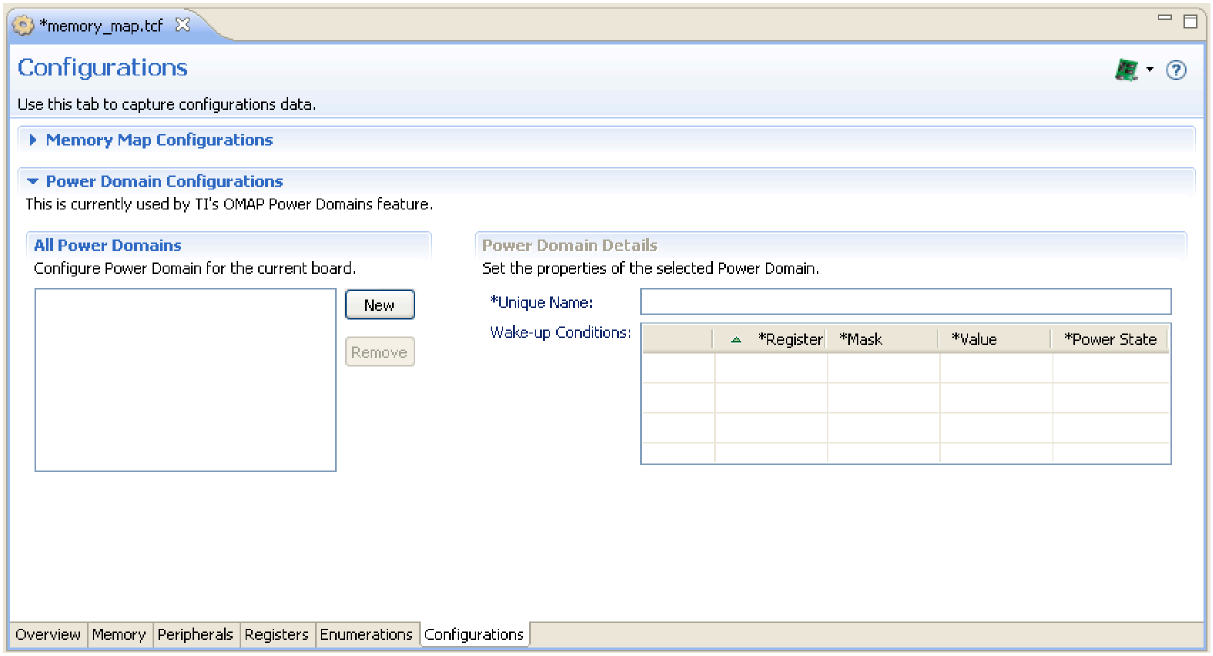
Before you create a power domain configuration, you must first create a control register.

Procedure

1. Click on the **Overview** tab.

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

Figure 4-20: Power Domain Configurations



Use this tab to capture configurations data.

This is currently used by TI's OMAP Power Domains feature.

Configure Power Domain for the current board.

Set the properties of the selected Power Domain.

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

   All settings are mandatory.

Related information

Creating a standalone register on page 104
Target configuration editor - Overview tab on page 88
Target configuration editor - Configurations tab on page 98

4.11 Creating a Group list

Describes how to create a new group list.

Procedure

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

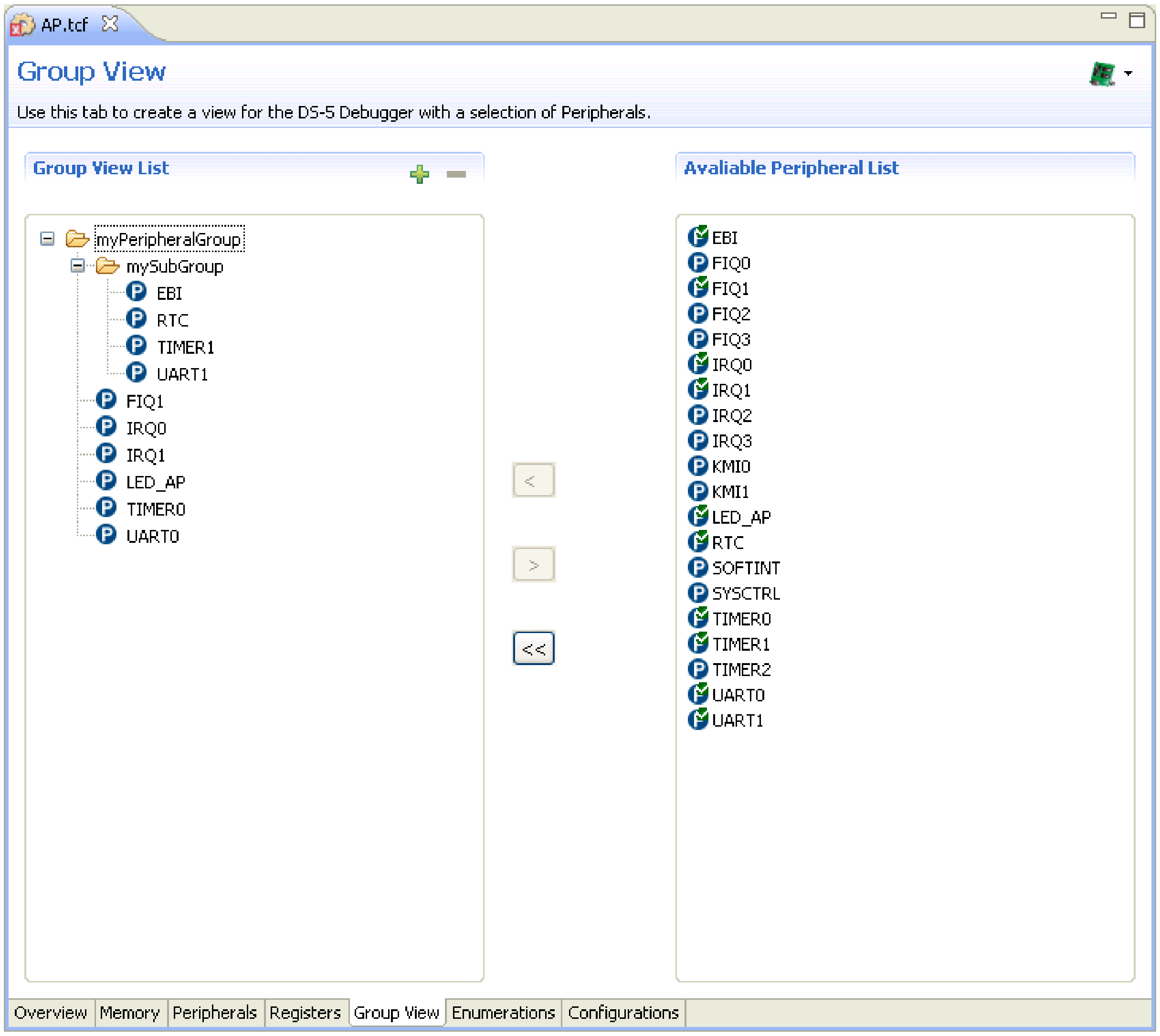
Note

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

---

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

Figure 4-21: Creating a group list



Related information

Target configuration editor - Group View tab on page 96

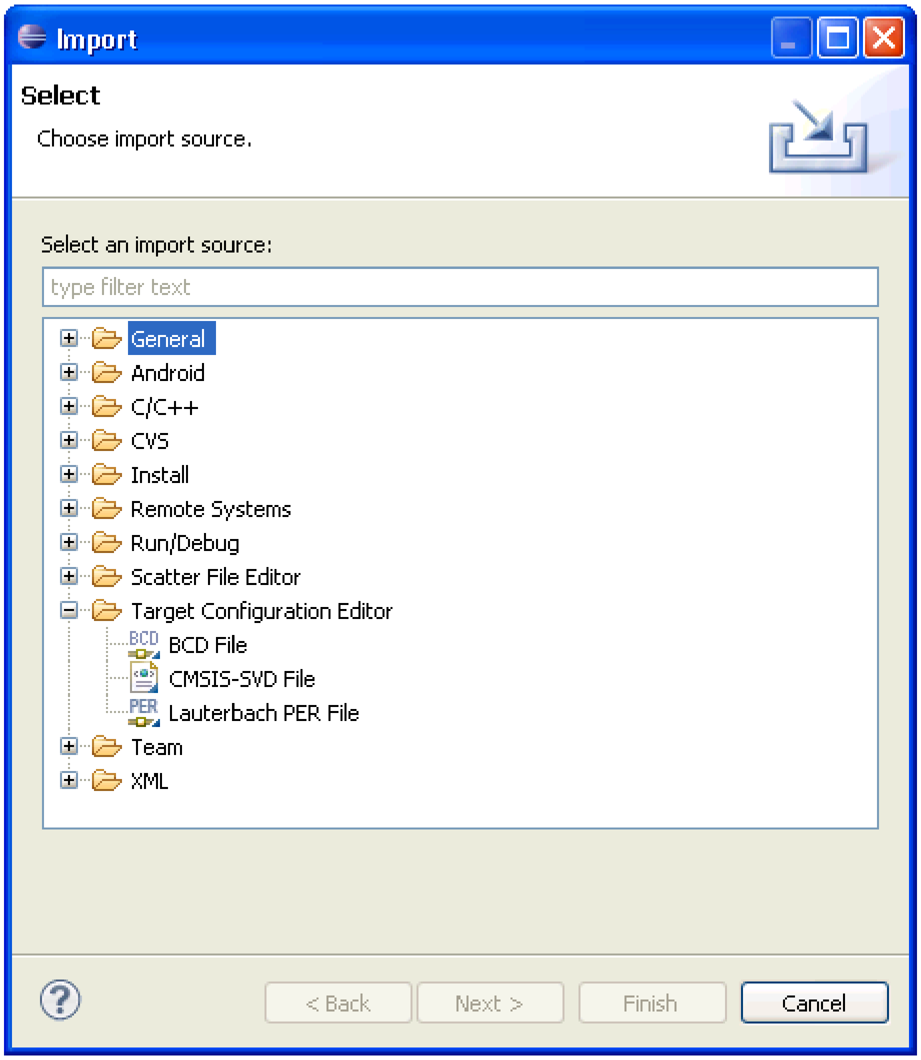
4.12 Importing an existing target configuration file

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

Procedure

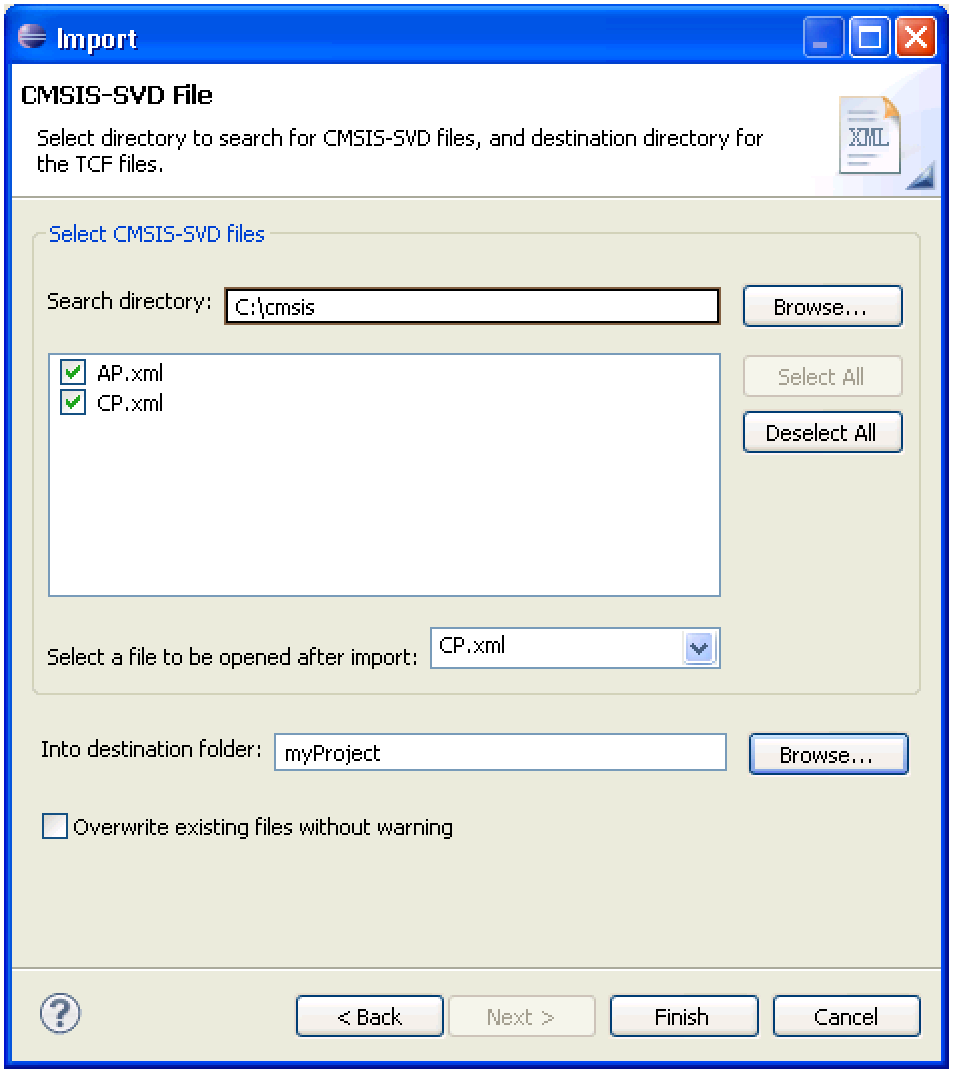
1. Select **Import** from the **File** menu.
2. Expand the **Target Configuration Editor** group.
3. Select the required file type.

Figure 4-22: Selecting an existing target configuration file



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

Figure 4-23: Importing the target configuration file



Import

CMSIS-SVD File

Select directory to search for CMSIS-SVD files, and destination directory for the TCF files.

Select CMSIS-SVD files

Search directory: C:\cmsis

✓ AP.xml
✓ CP.xml

Select a file to be opened after import: CP.xml

Into destination folder: myProject

☐ Overwrite existing files without warning

6. By default, all the files that can be imported are displayed. If the selection panel shows more than one file, select the files that you want to import.
7. Select the file that you want to automatically open in the editor.
8. In the Into destination folder field, click **Browse...** to select an existing project.
9. Click **Finish**. The new Target Configuration Files (TCF) is visible in the **Project Explorer** view.

Related information

Exporting a target configuration file on page 117

4.13 Exporting a target configuration file

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

About this task

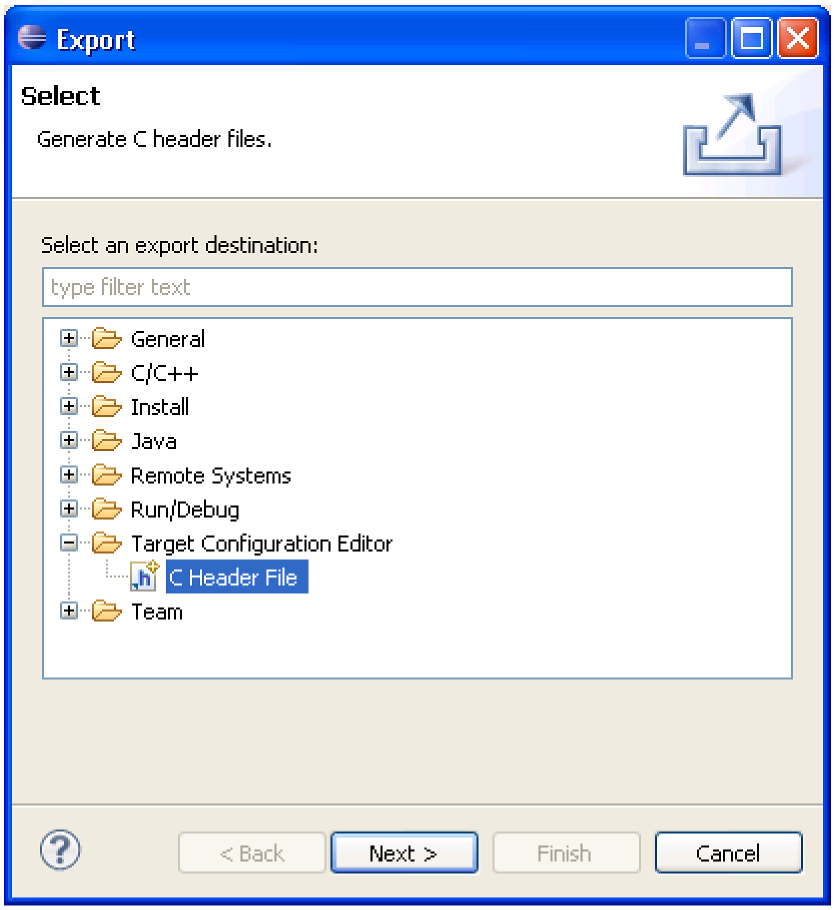
Note

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

Procedure

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

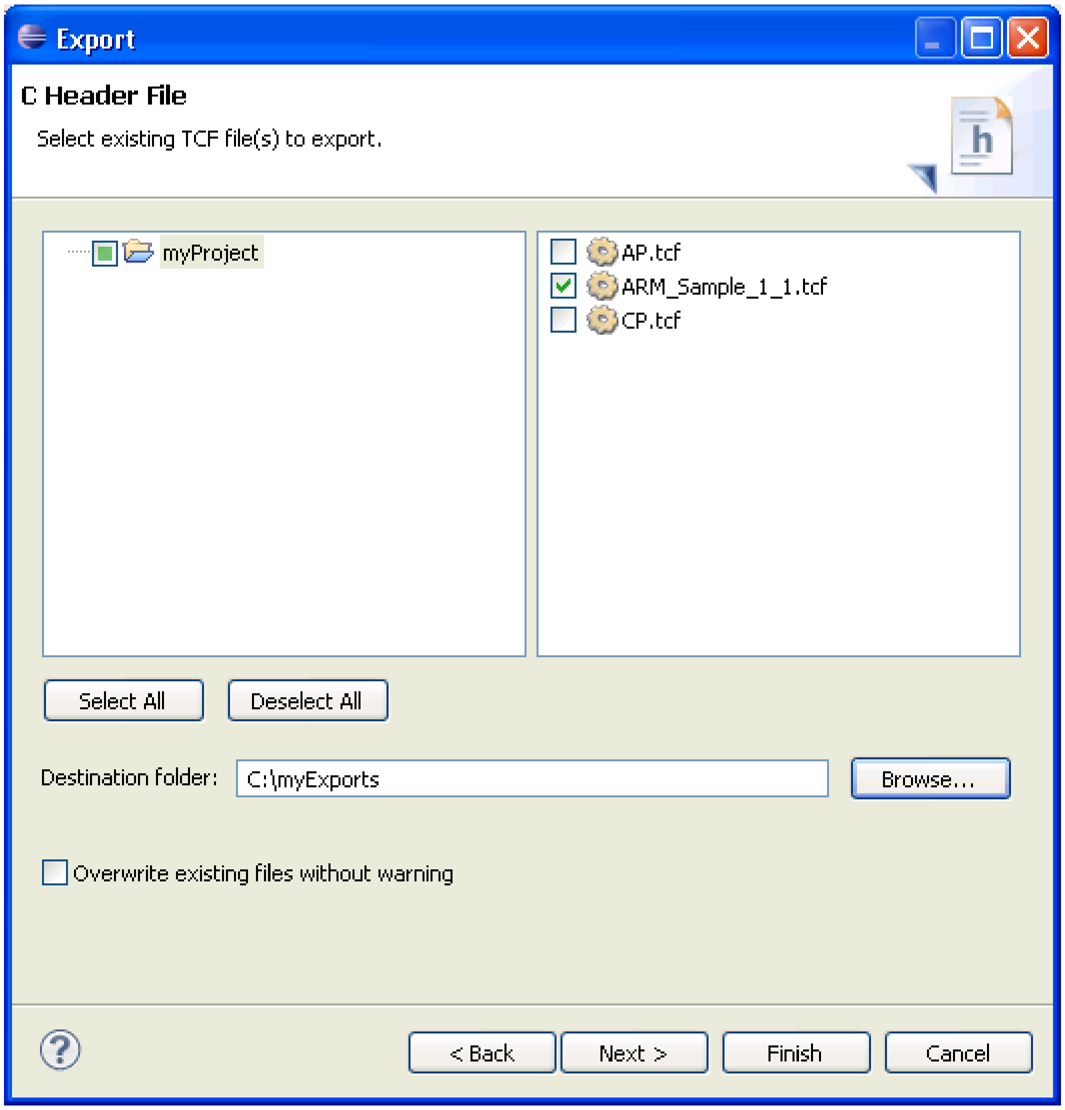
Figure 4-24: Exporting to C header file



4. Click on **Next**.

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

Figure 4-25: Selecting the files



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

Related information

Importing an existing target configuration file on page 115

5 Examining the Target

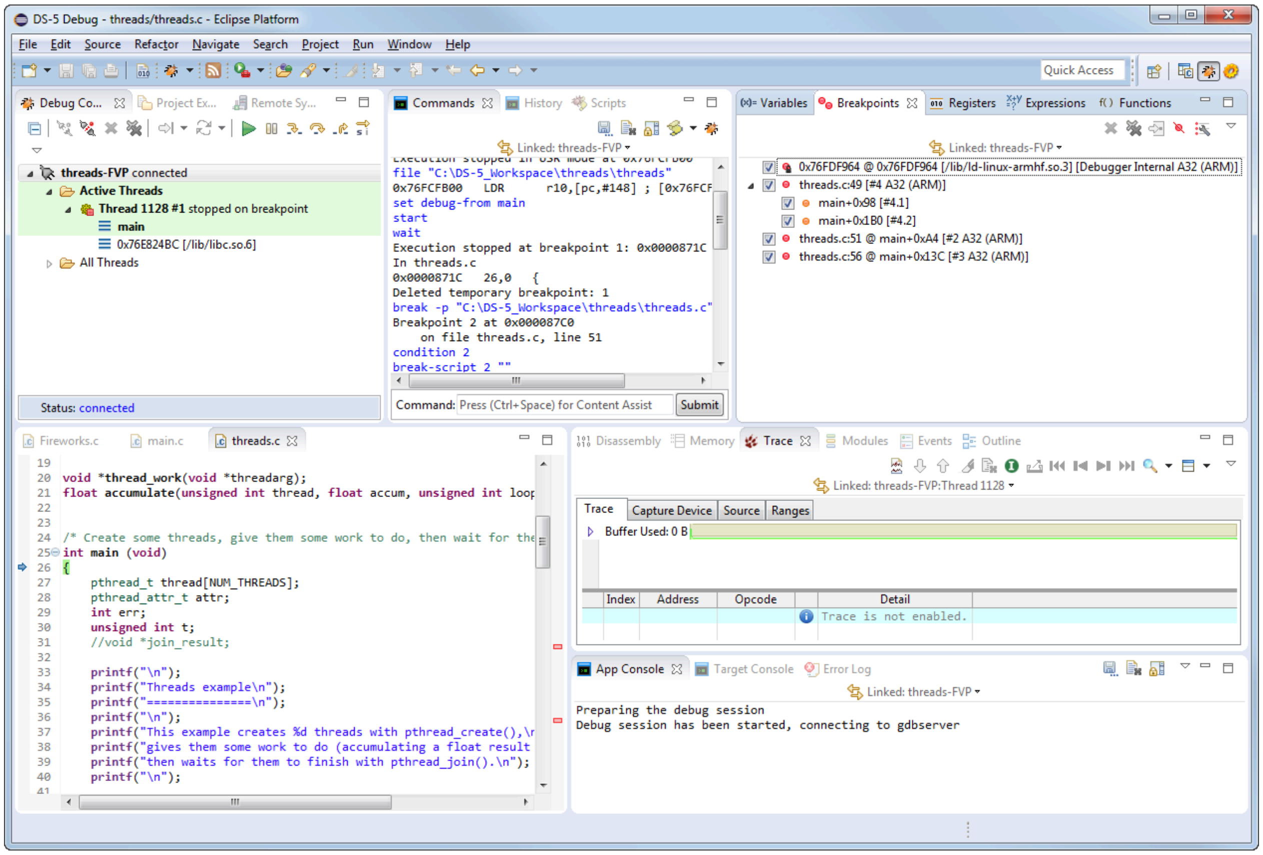
This chapter describes how to examine registers, variables, memory, and the call stack.

5.1 Examining the target execution environment

During a debug session, you might want to display the value of a register or variable, the address of a symbol, the data type of a variable, or the content of memory. The **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 5-1: Target execution environment


DS-5 Debug - threads/threads.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Debug Co... Project Ex... Remote Sy...
threads-FVP connected
Active Threads
Thread 1128 #1 stopped on breakpoint
main
0x76E824BC [/lib/libc.so.6]
All Threads
Status: connected
Commands History Scripts
Linked: threads-FVP
Command: Press (Ctrl+Space) for Content Assist
Submit
Variables Breakpoints Registers Expressions Functions
Fireworks.c main.c threads.c
Disassembly Memory Trace Modules Events Outline
Linked: threads-FVP:Thread 1128
Trace Capture Device Source Ranges
Buffer Used: 0 B
Index Address Opcode Detail
Trace is not enabled.
App Console Target Console Error Log

```
Execution stopped in USR mode at 0x76FCFB00
file "C:\DS-5_Workspace\threads\threads"
0x76FCFB00   LDR      r10,[pc,#148] ; [0x76FCF
set debug-from main
start
wait
Execution stopped at breakpoint 1: 0x0000871C
In threads.c
0x0000871C   26,0   {
Deleted temporary breakpoint: 1
break -p "C:\DS-5_Workspace\threads\threads.c"
Breakpoint 2 at 0x000087C0
    on file threads.c, line 51
condition 2
break-script 2 ""
```

```
0x76FDF964 @ 0x76FDF964 [/lib/ld-linux-armhf.so.3] [Debugger Internal A32 (ARM)]
threads.c:49 [#4 A32 (ARM)]
    main+0x98 [#4.1]
    main+0x1B0 [#4.2]
threads.c:51 @ main+0xA4 [#2 A32 (ARM)]
threads.c:56 @ main+0x13C [#3 A32 (ARM)]
```

```c

void *thread_work(void *threadarg);
float accumulate(unsigned int thread, float accum, unsigned int loop


/* Create some threads, give them some work to do, then wait for the
int main (void)
{
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int err;
    unsigned int t;
    //void *join_result;

    printf("\n");
    printf("Threads example\n");
    printf("==============\n");
    printf("\n");
    printf("This example creates %d threads with pthread_create(),\r
    printf("gives them some work to do (accumulating a float result
    printf("then waits for them to finish with pthread_join().\n");
    printf("\n");
```

```
Preparing the debug session
Debug session has been started, connecting to gdbserver
```

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 information

Running, stopping, and stepping through an application on page 59
Examining the call stack on page 121
Handling UNIX signals on page 75
Handling processor exceptions on page 77
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35

5.2 Examining the call stack

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

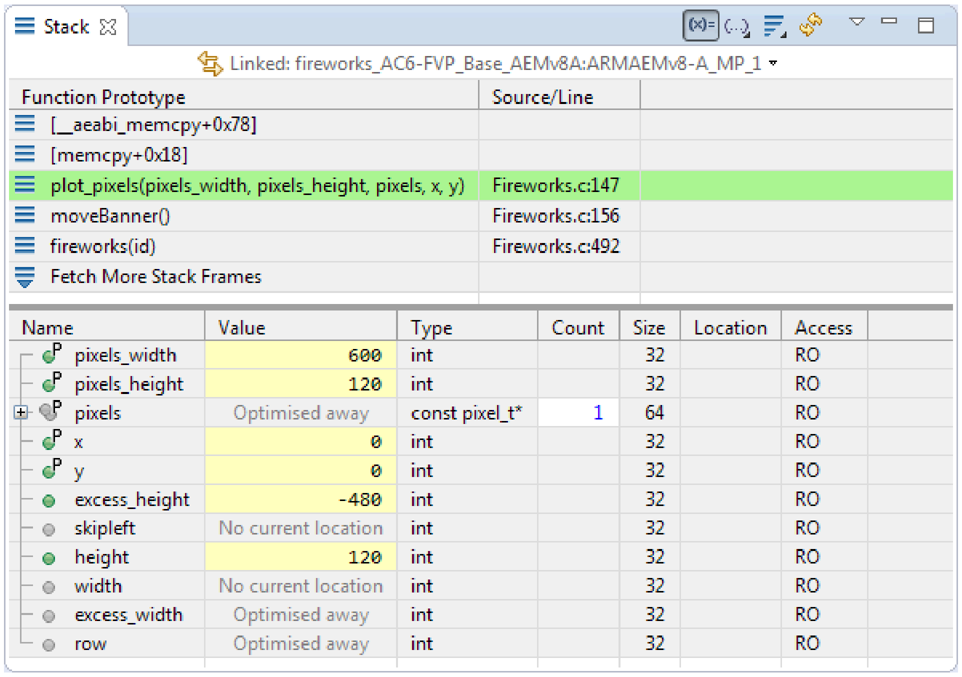
The debugger can display the calling sequence of any functions that are still in the execution path because their calling addresses are still on the call stack. However:

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

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

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 for more information.

Figure 5-2: Stack view showing information for a selectedcore



Stack

Linked: fireworks_AC6-FVP_Base_AEMv8A:ARMAEMv8-A_MP_1

Function Prototype	Source/Line
[__aeabi_memcpy+0x78]	
[memcpy+0x18]	
plot_pixels(pixels_width, pixels_height, pixels, x, y)	Fireworks.c:147
moveBanner()	Fireworks.c:156
fireworks(id)	Fireworks.c:492
Fetch More Stack Frames	

Name	Value	Type	Count	Size	Location	Access
pixels_width	600	int		32		RO
pixels_height	120	int		32		RO
pixels	Optimised away	const pixel_t*	1	64		RO
x	0	int		32		RO
y	0	int		32		RO
excess_height	-480	int		32		RO
skipleft	No current location	int		32		RO
height	120	int		32		RO
width	No current location	int		32		RO
excess_width	Optimised away	int		32		RO
row	Optimised away	int		32		RO

Related information

Running, stopping, and stepping through an application on page 59
Examining the target execution environment on page 120
Handling UNIX signals on page 75
Handling processor exceptions on page 77
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35

5.3 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.

Note

Trace triggers are not supported on Cortex®-M series processors.

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.

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.

0%

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

50%

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

99%

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

Note

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

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

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

Note

Trace trigger functionality is dependent on the target platform being able to signal to the trace capture hardware, such as ETB or 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 information

About post-mortem debugging of trace data on page 125
Overview: Running Arm Debugger from the command-line or from a script on page 156
Command-line debugger options on page 157
Specifying a custom configuration database using the command-line on page 168

5.4 About post-mortem debugging of trace data

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

Once the trace data and other files are ready, you configure the headless command-line debugger to connect to the post-mortem debug configuration from the configuration database.

You can then inspect the state of the data at the time of the trace capture.

Note

- The memory and registers are read-only.
- You can add more debug information using additional files.
- You can also decode trace and dump the output to files.

The basic steps for post-mortem debugging using the headless command-line debugger are:

1. Generate trace data files.
2. Use `--cdb-list` to list the platforms and parameters available in the configuration database.
3. Use `--cdb-entry` to specify a platform entry in the configuration database.
4. If you need to specify additional parameters, use the `--cdb-entry-param` option to specify the parameters.

Note

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

Related information

About trace support on page 122
Overview: Running Arm Debugger from the command-line or from a script on page 156
Command-line debugger options on page 157
Specifying a custom configuration database using the command-line on page 168

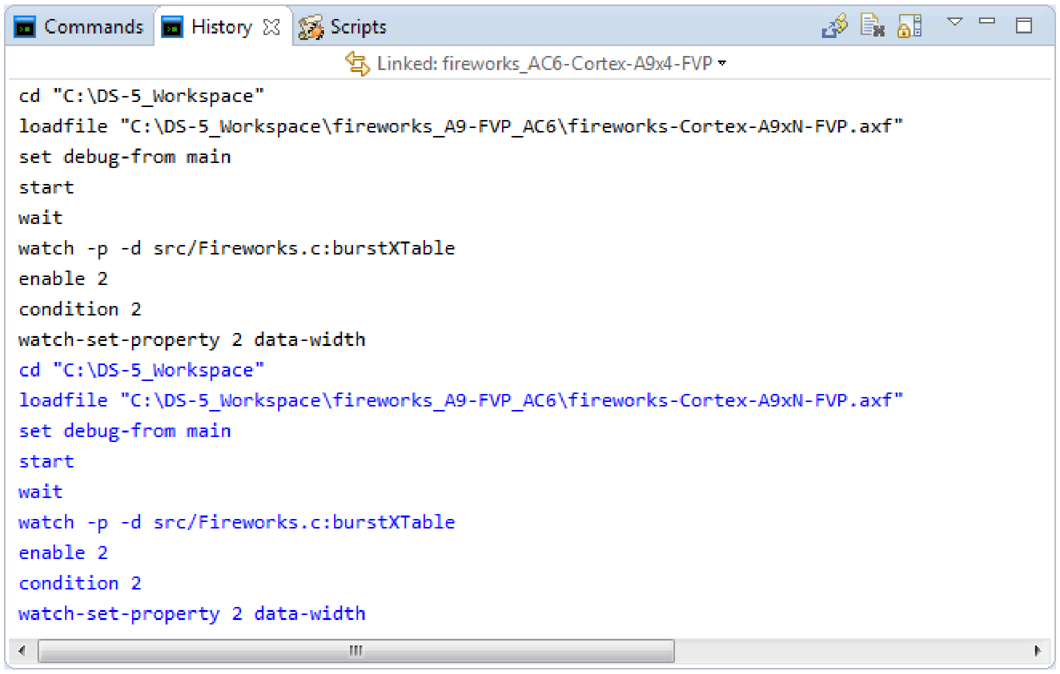
6 Debugging with Scripts

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

6.1 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 6-1: Commands generated during a debug session



```
cd "C:\DS-5_Workspace"
loadfile "C:\DS-5_Workspace\fireworks_A9-FVP_AC6\fireworks-Cortex-A9xN-FVP.axf"
set debug-from main
start
wait
watch -p -d src/Fireworks.c:burstXTable
enable 2
condition 2
watch-set-property 2 data-width
cd "C:\DS-5_Workspace"
loadfile "C:\DS-5_Workspace\fireworks_A9-FVP_AC6\fireworks-Cortex-A9xN-FVP.axf"
set debug-from main
start
wait
watch -p -d src/Fireworks.c:burstXTable
enable 2
condition 2
watch-set-property 2 data-width
```

6.2 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
```

6.3 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
```

6.4 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 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 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>`.

Related information

Scripts view on page 393
Importing and translating a CMM script on page 129
Supported CMM commands for translations on page 130

6.4.1 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.

Before you begin

- 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 Import and translate a CMM script .

   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.

Related information

Supported CMM commands for translations on page 130
Scripts view on page 393

6.4.2 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 information

Support for importing and translating CMM scripts on page 129
Importing and translating a CMM script on page 129
Scripts view on page 393

6.5 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 information

Creating a new Jython project in Arm Development Studio on page 134

Configuring an existing project to use the Arm Development Studio Jython interpreter on page 137

Creating a Jython script on page 138

Running a script on page 139

Jython script concepts and interfaces on page 132

Script Parameters dialog box on page 430

6.6 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())
```

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

6.7 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.

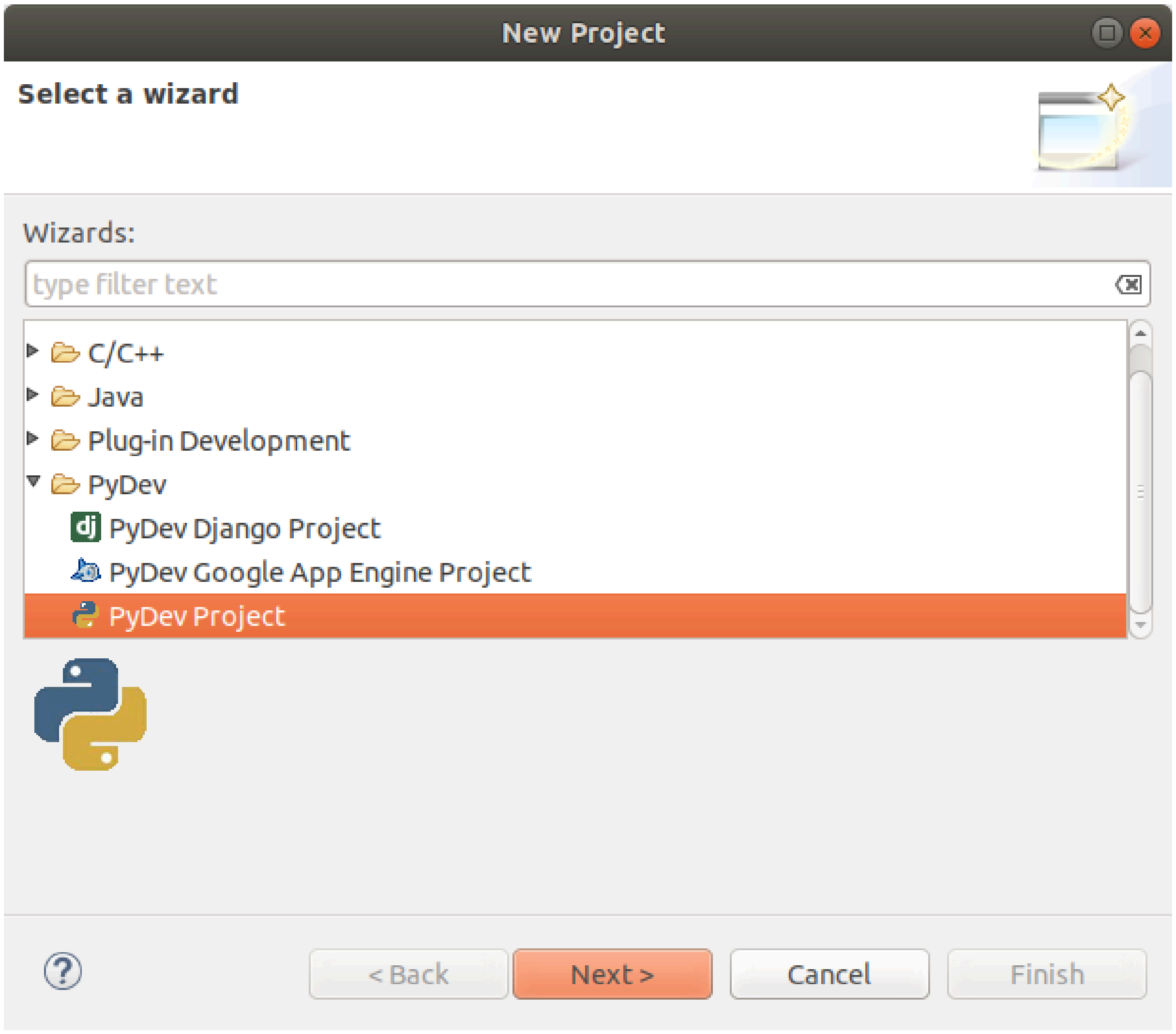
6.7.1 Creating a new Jython project in Arm Development Studio

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

Procedure

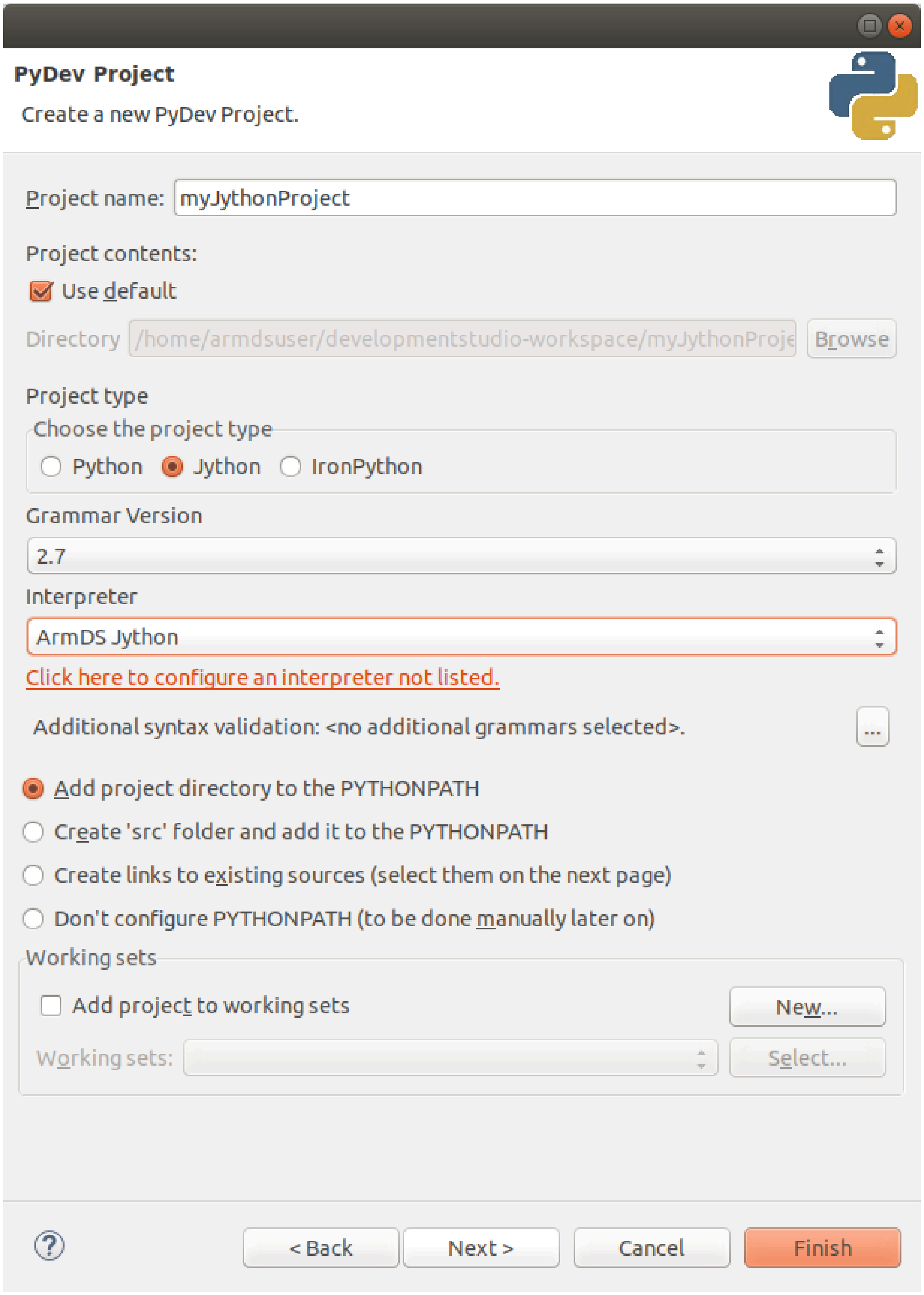
1. Select **File** > **New** > **Project...** from the main menu.
2. Expand the **PyDev** group.
3. Select **PyDev Project**.

Figure 6-2: PyDev project wizard



4. Click **Next**.
5. Enter the project name and select relevant details:
   a) In **Project name**, enter a suitable name for the project.
   b) In **Choose the project type**, select **Jython**.
   c) In **Interpreter**, select **Arm DS Jython**.

Figure 6-3: PyDev project settings


PyDev Project

Create a new PyDev Project.

Project name: myJythonProject

Project contents:

[✓] Use default

Directory /home/armdsuser/developmentstudio-workspace/myJythonProje

Project type

Choose the project type

( ) Python [✓ Jython] ( ) IronPython

Grammar Version

2.7

Interpreter

ArmDS Jython

Click here to configure an interpreter not listed.

Additional syntax validation: <no additional grammars selected>.

[✓ Add project directory to the PYTHONPATH]

( ) Create 'src' folder and add it to the PYTHONPATH

( ) Create links to existing sources (select them on the next page)

( ) Don't configure PYTHONPATH (to be done manually later on)

Working sets

[ ] Add project to working sets

Working sets:

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

Related information

Configuring an existing project to use the Arm Development Studio Jython interpreter on page 137
Creating a Jython script on page 138
Running a script on page 139
About Jython scripts on page 131
Jython script concepts and interfaces on page 132
Script Parameters dialog box on page 430

6.7.2 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 information

Creating a new Jython project in Arm Development Studio on page 134
Creating a Jython script on page 138
Running a script on page 139
About Jython scripts on page 131
Jython script concepts and interfaces on page 132
Script Parameters dialog box on page 430

6.8 Creating a Jython script

Shows a typical workflow for creating and running a Jython script in the debugger.

Procedure

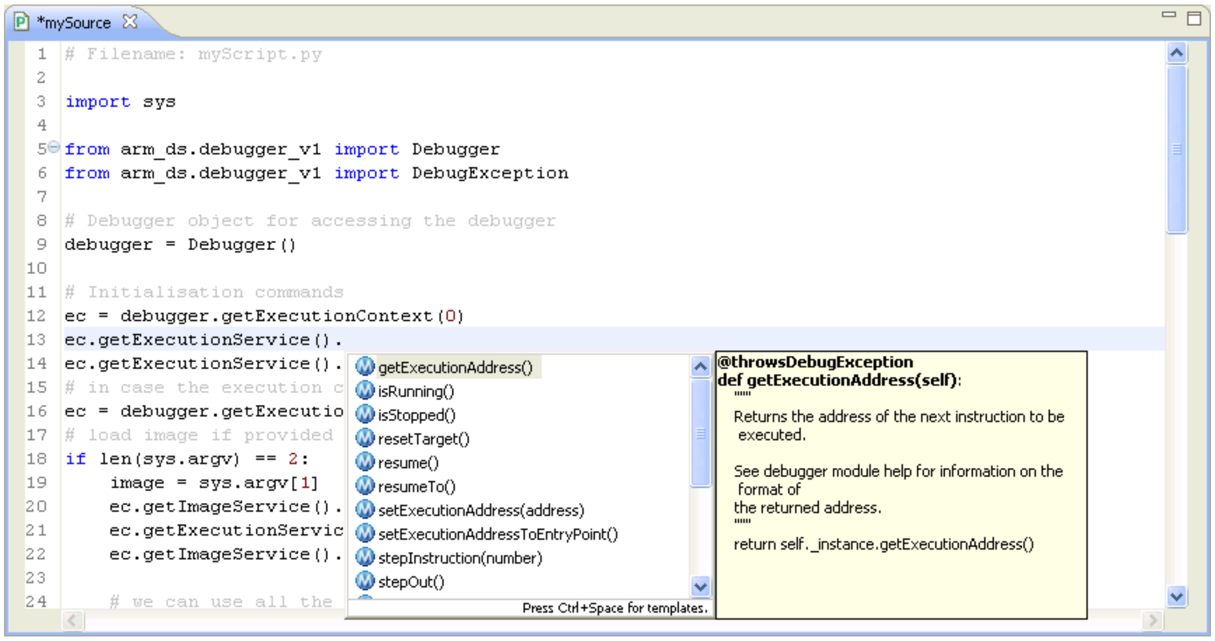
1. Create an empty Jython script file.
2. Right-click the Jython script file and select **Open**.
3. Add the following code to your file in the editor:

```
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 6-4: Jython auto-completion and help



*mySource

```
# Filename: myScript.py

import sys

from arm_ds.debugger_v1 import Debugger
from arm_ds.debugger_v1 import DebugException

# Debugger object for accessing the debugger
debugger = Debugger()

# Initialisation commands
ec = debugger.getExecutionContext(0)
ec.getExecutionService().
ec.getExecutionService().
# in case the execution c
ec = debugger.getExecutio
# load image if provided
if len(sys.argv) == 2:
    image = sys.argv[1]
    ec.getImageService().
    ec.getExecutionServic
    ec.getImageService().

    # we can use all the
```

```
getExecutionAddress()
isRunning()
isStopped()
resetTarget()
resume()
resumeTo()
setExecutionAddress(address)
setExecutionAddressToEntryPoint()
stepInstruction(number)
stepOut()
Press Ctrl+Space for templates.
```

```
@throwsDebugException
def getExecutionAddress(self):
    """
    Returns the address of the next instruction to be
     executed.

    See debugger module help for information on the
     format of
    the returned address.
    """
    return self._instance.getExecutionAddress()
```

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 information

Creating a new Jython project in Arm Development Studio on page 134
Configuring an existing project to use the Arm Development Studio Jython interpreter on page 137
Running a script on page 139
About Jython scripts on page 131
Jython script concepts and interfaces on page 132
Script Parameters dialog box on page 430

6.9 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

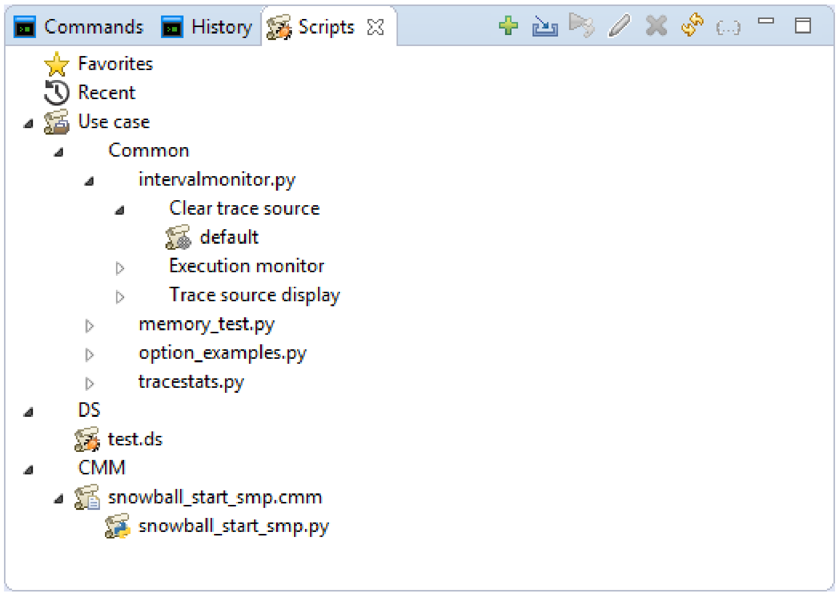
1. To run a script from the Arm Development Studio IDE:
   a) Launch Arm Development Studio IDE.
   b) Configure a connection to the target.

   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 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 6-5: Scripts view



Note

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

Related information

Exporting Arm Debugger commands generated during a debug session on page 127
Creating an Arm Debugger script on page 127
Creating a CMM-style script on page 128
Commands view on page 329
History view on page 358
Creating a new Jython project in Arm Development Studio on page 134
Configuring an existing project to use the Arm Development Studio Jython interpreter on page 137
Creating a Jython script on page 138
About Jython scripts on page 131
Jython script concepts and interfaces on page 132
Script Parameters dialog box on page 430

6.10 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.

6.11 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.

6.12 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.

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.

Example

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

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.

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
...
$Help$
```

6.13 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.

6.14 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)
```

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"),
              ]
          ),
      ]
```

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.

DTSL Options

Options are defined in a similar way to the Debug and Trace Services Layer (DTSL) options, using the same parameters and way of nesting child options.

See DTSL options and the example usecase scripts in your Arm® Development Studio installation for more information on how to construct a list of options.

6.15 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.

6.16 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
```

6.17 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.

6.18 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.

6.19 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.

6.20 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`.

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.

7 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.

There are two ways to connect to a target using the command-line:

- Using the configDB entry, and specifying the `cdb-entry`.
- Using a pre-installed CMSIS pack, and specifying the `cmsis-device`.

7.1 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 for syntax and instructions on how to launch Arm Debugger from the command line.

Related information

Command-line debugger options on page 157
Specifying a custom configuration database using the command-line on page 168

7.2 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.

The commands listed in this topic are either generic and work with all connection types, or are for use with CMSIS pack-based connections only. Those that are CMSIS-only are clearly indicated.

Connecting to a target

There are two ways to connect to a target using the command-line:

- Using a Configuration Database (configDB) entry, and specifying the `cdb-entry`.
- Using a pre-installed CMSIS pack, and specifying your `cmsis-device`.

Note

When you have 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.

Syntax

Launch the command-line debugger using the following syntax:

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

Where:

`armdbg`
Invokes the Development Studio command-line debugger.

`--option <arg>` or **`--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.

Options

`--browse`

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

Note

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

`--cdb-entry <arg>`

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

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

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

Use `--cdb-list` to determine the entries in the configuration database that the debugger can connect to. You can specify partial entries, such as `"Arm"` or `"Arm::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-ST and a USB connection, first use `--cdb-list` to identify the entries in the configuration database within `Arm`, enter:

```
armdbg --cdb-entry "Arm::Versatile Express A9x4::Bare Metal Debug::Bare Metal
 SMP Debug of all cores::Debug Cortex-A9x4 SMP::DSTREAM-ST" --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.

`Snapshot File`

Specifies a Snapshot file to load from the file system.

`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.

`model_connection_address`

Specifies the connection address and port for an already running model. Enter the TCP/IP address and port to connect in the format `ipaddress:port`. The default connection address is `127.0.0.1` and port `7100`. This parameter works as a pair with the `connect_existing_model` parameter.

`connect_existing_model`

Specifies whether the model is running locally or remotely. The default value for this parameter is `false`. Change to `true` if you are connecting to a model running on your local host.

Note

To connect to models running on the local host, you must first launch the model with the `--iris-server` switch before connecting to it. To connect to a model running on a remote host, you must first launch the model with the `--iris-server --iris-allow-remote` switches before connecting to it remotely.

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!'"
```

`--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.

`--cmsis-device <search-term>|<exact-device>`

Specify the CMSIS device you are connecting to. You can also use this option to browse for CMSIS devices. For use with CMSIS pack-based connections only.

Note

The CMSIS pack associated with your device must be installed before you run this command.

For example, to browse for a device that contains "musca" in the device name, use:

```
--cmsis-device musca
```

You can also specify the exact device you want to connect to. The device name has the following format:

```
<packfile>:<family>:<sub-family>:<device>:<processor>
```

You can use wildcards (`*`) in any part of the name.

For example:

```
--cmsis-device Musca_A1_BSP:ARM Cortex M33:Musca:Cortex-M33-0
```

`--continue_on_error= true | false`
Specifies whether the debugger stops the target and exits the current script when an error occurs.

The default is `--continue_on_error=false`.

`--disable-semihosting`
Disables semihosting operations.

`--disable_semihosting_console`
Disables all semihosting operations to the debugger console.

`--enable-semihosting`
Enables semihosting operations.

`-h` or `--help`
Displays a summary of the main command-line options.

`-b=<filename>` or `--image=<filename>`
Specifies the image file for the debugger to load when it connects to the target.

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

Note

This is the default if no script file is specified.

`--launch-config <path>`

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

For example:

```
armdbg --launch-config "C:\Workspace\debugconfiguration.cli"
```

You can create a command-line launch configuration using the Export tab 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, enter:

  ```
  armdbg --launch-config "C:\Workspace\debugconfiguration.cli" --launch-config-
  connect-only true
  ```

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

  ```
  armdbg --launch-config "C:\Workspace\debugconfiguration.cli" --launch-
  config-connect-only true --image "C:\Arm_DS_Workspace\fireworks_Cortex-
  A77\fireworks_Cortex-A77.axf"
  ```

`--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.

`--log-debug-sequences`

If your CMSIS pack files define debug sequences, specify this option if you want the debugger to log the sequences that get executed during the debug process, to the console. For use with CMSIS pack-based connections only.

`--probe "<probe-name>"`

Specify the probe you want to connect to. For use with CMSIS pack-based connections only.

The probe name can be one of the following:

- ULINKpro™
- ULINKpro D
- ULINK2
- ULINK-Plus
- CMSIS-DAP
- Cadence Virtual Debug
- FTDI MPSSE JTAG
- ST-Link
- Your own third-party debug probe. If you use this option, you must specify the probe name as it is defined in the probe definition file.

`--run-control <option>`

Specifies what to do when you connect to a target. The options are as follows:

`connect-only` - Connect to the target and then stop running. `entrypoint` - Connect to the target and run until the debugger reaches the image entry point. `symbol:<symbol-to-stop-at>` - Connect to the target and run until the debugger hits the specified symbol.

`--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`.

`--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 for more information.

`--target <location>`

The location of the CMSIS device you are connecting to. This can be a web address, a USB ID, or an SDF file. For use with CMSIS pack-based connections only.

`--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, such as, for Linux application debug or when using the DSTREAM family of devices, 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`.

Examples

- To connect to an Arm FVP Cortex®-A9x4 model and specify an image to load, enter:

```
armdbg --cdb-entry "Arm FVP::VE_Cortex_A9x4::Bare Metal Debug::Bare Metal
 Debug::Debug Cortex-A9x4 SMP" --image "C:\\Arm_DS_Workspace\\fireworks_A9x4-FVP\
\fireworks-Cortex-A9x4-FVP.axf"
```

- To connect and load an image on a single Cortex-A53 core on the Arm Cortex-A72/Cortex®-A53 big.LITTLE™ FVP running locally, enter:

```
armdbg --cdb-entry "Arm FVP::Base_A72x1_A53x1::Bare Metal Debug::Bare Metal
 Debug::Cortex-A53" --cdb-entry-param "connect_existing_model=true" --image "C:\
\Arm_DS_workspace\\fireworks_Armv8-A-FVP_AC6\\fireworks_Armv8-A-FVP_AC6.axf"
```

- To connect to a target and debug a Linux application, enter:

```
armdbg --cdb-entry "Linux Application Debug::Application Debug::Connections via
 gdbserver::gdbserver (TCP)::Connect to already running application" --cdb-entry-
param "gdb_address=TCP:10.5.196.50" --cdb-entry-param "gdb_port=5350"
```

Note

For Linux application debug, the **Download and debug application** option is not supported on the command-line debugger.

- To connect and debug a Linux kernel on Juno Arm Development Platform (r2) target using DSTREAM-ST and a USB connection, enter:

```
armdbg --cdb-entry "Arm::Juno Arm Development Platform (r2)::Linux Kernel and/or
 Device Driver Debug::Linux Kernel Debug::Debug Cortex-A53x4 SMP::DSTREAM-ST" --
cdb-entry-param="Connection=USB:000271"
```

- To connect to a single Cortex-A15 core on the Versatile Express Cortex-A15x2+A7x3 target, using DSTREAM and a TCP/IP connection, enter:

```
armdbg --cdb-entry "Arm Development Boards::Versatile_Express_V2P-CA15_A7::Bare
 Metal Debug::Bare Metal Debug::Debug Cortex-A15_0::DSTREAM" --cdb-entry-param
 "Connection=TCP:10.8.197.59"
```

- To connect to a single Cortex-M7 core on the Arm MPS2 FPGA prototyping board, using a DSTREAM debug probe and a TCP/IP connection, enter:

```
armdbg --cdb-entry "Arm::Cortex-M Prototyping System (MPS2) Cortex-M7 (SMM-
M7)::Bare Metal Debug::Bare Metal Debug::Debug Cortex-M7::DSTREAM" --cdb-entry-
param "Connection=TCP:10.62.63.21"
```

- To connect to a Juno Arm Development Platform (r0) big.LITTLE target, using DSTREAM and a TCP/IP connection, enter:

```
armdbg --cdb-entry "Arm Development Boards::Juno Arm Development Platform
 (r0)::Bare Metal Debug::Bare Metal Debug::Debug Cortex-A57/Cortex-A53
 big.LITTLE::DSTREAM" --cdb-entry-param "Connection=TCP:10.2.194.40"
```

- To connect to a Snapshot file, enter:

```
armdbg --cdb-entry "Generic::Snapshot::View snapshot::View snapshot::View
 snapshot::Snapshot" --cdb-entry-param "Snapshot File=C:\\Arm_DS_Workspace\
\CoreSight_access\\example_captures\\Juno\\snapshot.ini"
```

  You can also use the `Connection` parameter to connect to a Snapshot file. At the command prompt, enter:

```
armdbg --cdb-entry "Generic::Snapshot::View snapshot::View snapshot::View
 snapshot::Snapshot" --cdb-entry-param "Connection=C:\\Arm_DS_Workspace\
\CoreSight_access\\example_captures\\Juno\\snapshot.ini"
```

- To disconnect a command-line debugger session, enter **quit** at the command prompt.

Related information

Arm Debugger command-line console keyboard shortcuts on page 174
Exporting Arm Debugger commands generated during a debug session on page 127
Using semihosting to access resources on the host computer on page 80

7.3 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.

  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 and Capturing trace data using the command-line debugger for examples of how debugger options and arguments are used.

7.4 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.

About this task

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.
    - On Windows, select **Start** > **All Programs** > **Arm DS Command Prompt**
    - On Linux:
        - Add the `<install_directory>/bin` directory to your `PATH` environment variable. If it is already configured, then you can skip this step.
        - Open a terminal.
2. List the entries in the user-specified configuration databases. Use the following syntax:
    - On Windows, enter: :`debugger --cdb-list--cdb-root path_to_cdb1[;path_to_cdb2]`. For example, `debugger --cdb-list --cdb-root C:\\Arm_DS_Workspace\\MyConfigDB1;Arm_DS_Workspace\\MyConfigDB2`.
    - On Linux, enter: `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, on Windows:

```
debugger --cdb-entry "ARM Development Boards::Versatile Express A9x4::Bare Metal
 Debug::Bare Metal SMP Debug of all cores::Debug Cortex-A9x4 SMP::DSTREAM" --cdb-
entry-param "Connection=USB:000271" --cdb-root C:\\Arm_DS_Workspace\\MyConfigDB1
```

Where:

`debugger`

Is the command to invoke Arm Debugger.

`--cdb-entry`

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 information

Command-line debugger options on page 157

Overview: Running Arm Debugger from the command-line or from a script on page 156

7.5 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.

About this task

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:

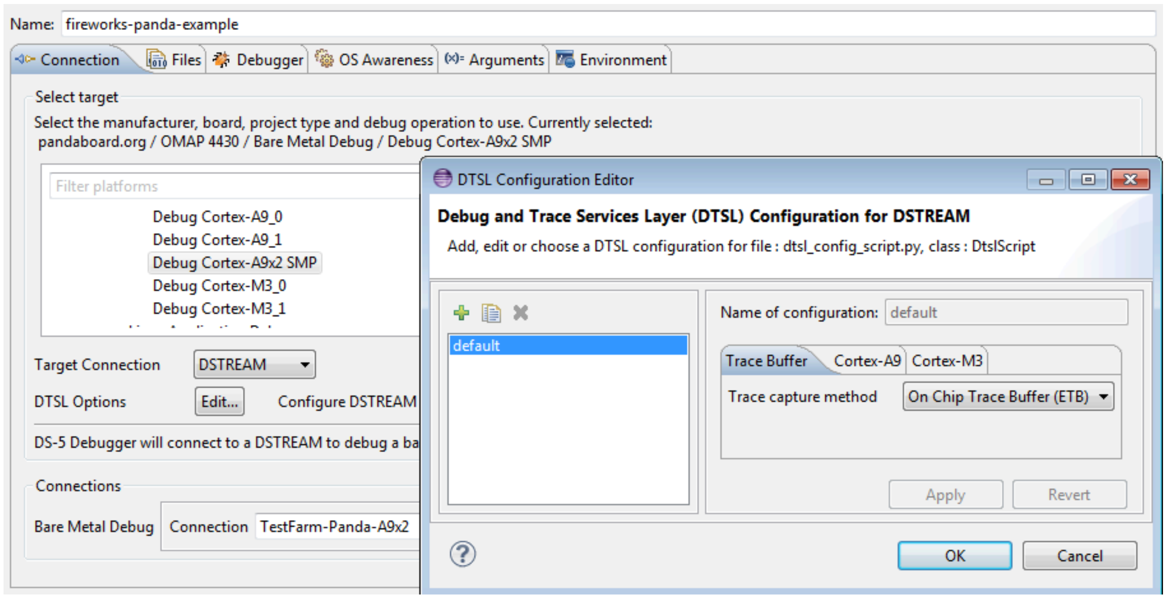
```
loadfile C:\Arm_DS_Workspace\fireworks_panda\fireworks_panda.axf # Load an image to
 debug
start                          # Start running the image after setting a temporary
 breakpoint
wait                           # Wait for a breakpoint
trace start                    # Start the trace capture 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 7-1: Enable trace in the DTSL options


Name: fireworks-panda-example

Select target
Select the manufacturer, board, project type and debug operation to use. Currently selected:
pandaboard.org / OMAP 4430 / Bare Metal Debug / Debug Cortex-A9x2 SMP

Target Connection: DSTREAM
DS-5 Debugger will connect to a DSTREAM to debug a ba

Bare Metal Debug — Connection: TestFarm-Panda-A9x2

DTSL Configuration Editor
Debug and Trace Services Layer (DTSL) Configuration for DSTREAM
Add, edit or choose a DTSL configuration for file : dtsl_config_script.py, class : DtslScript

Name of configuration: default
Trace capture method: On Chip Trace Buffer (ETB)

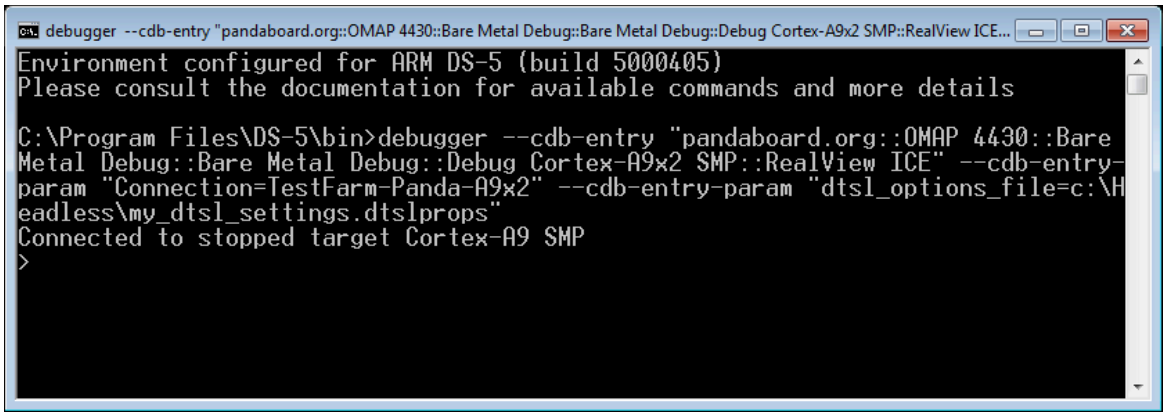
These settings are stored in a `*.dtslprops` file in your workspace. The filename is shown in the **Name of configuration** field in the dialog box. In this example, the settings are stored in the `default.dtslprops` file.

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`.
4. Open the **Arm DS Command Prompt**. In the command prompt:
   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 7-2: Command-line debugger connection with DTSL options enabled.



debugger --cdb-entry "pandaboard.org::OMAP 4430::Bare Metal Debug::Bare Metal Debug::Debug Cortex-A9x2 SMP::RealView ICE...

```
Environment configured for ARM DS-5 (build 5000405)
Please consult the documentation for available commands and more details

C:\Program Files\DS-5\bin>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:\H
eadless\my_dtsl_settings.dtslprops"
Connected to stopped target Cortex-A9 SMP
>
```

Results

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 information

Overview: Running Arm Debugger from the command-line or from a script on page 156
Command-line debugger options on page 157
Debug Configurations - Connection tab on page 431
DTSL Configuration Editor dialog box on page 447

7.6 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 an Arm FVP Cortex®-A9x4 model, and specify an image to load:

```
debugger --cdb-entry "Arm FVP::VE_Cortex_A9x4::Bare Metal Debug::Bare Metal
 Debug::Debug Cortex-A9x4 SMP" --image "C:\\Arm_DS_Workspace\\fireworks_A9-FVP_AC6\
\fireworks-Cortex-A9xN-FVP.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.

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 information

Command-line debugger options on page 157

7.7 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 information

Command-line debugger options on page 157

8 Working with the Snapshot Viewer

This chapter describes how to work with the Snapshot Viewer.

8.1 About the Snapshot Viewer

Use the Snapshot Viewer to analyze a snapshot representation of the application state of one or more processors in scenarios where interactive debugging is not possible.

To enable debugging of an application using the Snapshot Viewer, you must have the following data:

- Register Values.
- Memory Values.
- Debug Symbols.

Note

To do trace analysis, you must also have trace data.

If you are unable to provide all of this data, then the level of debug that is available is compromised. Capturing this data is specific to your application, and no tools are provided to help with this. You might have to install exception or signal handlers to catch erroneous situations in your application and dump the required data out.

You must also consider how to get the dumped data from your device onto a workstation that is accessible by the debugger. Some suggestions on how to do this are to:

- Write the data to a file on the host workstation using semihosting.
- Send the data over a UART to a terminal.
- Send the data over a socket using TCP/IP.

Register values

Register values are used to emulate the state of the original system at a particular point in time. The most important registers are those in the current processor mode. For example, on an Arm®v4 architecture processor these registers are R0-R15 and also the Program Status Registers (PSRs):

- Current Program Status Register (CPSR)
- Application Program Status Register (APSR)
- Saved Program Status Register (SPSR).

Be aware that on many Arm processors, an exception, a data abort, causes a switch to a different processor mode. In this case, you must ensure that the register values you use reflect the correct mode in which the exception occurred, rather than the register values within your exception handler.

If your application uses floating-point data and your device contains vector floating-point hardware, then you must also provide the Snapshot Viewer with the contents of the vector floating-point registers. The important registers to capture are:

- Floating-point Status and Control Register (FPSCR)
- Floating-Point EXCeption register (FPEXC)
- Single precision registers (S *n*)
- Double precision registers (D *n*)
- Quad precision registers (Q *n*).

Memory values

The majority of the application state is usually stored in memory in the form of global variables, the heap and the stack. Due to size constraints, it is often difficult to provide the Snapshot Viewer with a copy of the entire contents of memory. In this case, you must carefully consider the areas of memory that are of particular importance.

If you are debugging a crash, the most useful information to find out is often the call stack, because this shows the calling sequence of each function prior to the exception and the values of all the respective function parameters. To show the call stack, the debugger must know the current stack pointer and have access to the contents of the memory that contains the stack. By default, on Arm processors, the stack grows downwards, you must provide the memory starting from the current stack pointer and going up in memory until the beginning of the stack is reached. If you are unable to provide the entire contents of the stack, then a smaller portion starting at the current stack pointer is still useful because it provides the most recent function calls.

If your application uses global (`extern` or file `static`) data, then providing the corresponding memory values enables you to view the variables within the debugger.

If you have local or global variables that point to heap data, then you might want to follow the relevant pointers in the debugger to examine the data. To do this you must have provided the contents of the heap to the Snapshot Viewer. Be aware that heap can often occupy a large memory range, so it might not be possible to capture the entire heap. The layout of the heap in memory and the data structures that control heap allocation are often specific to the application or the C library, see the relevant documentation for more information.

To debug at the disassembly level, the debugger must have access to the memory values where the application code is located. It is often not necessary to capture the contents of the memory containing the code, because identical data can often be extracted directly from the image using processing tools such as `fromelf` . However, some complications to be aware of are:

- Self-modifying code where the values in the image and memory can vary.
- Dynamic relocation of the memory address within the image at runtime.

Debug symbols

The debugger requires debug information to display high-level information about your application, for example:

- Source code.
- Variable values and types.
- Structures.
- Call stack.

This information is stored by the compiler and linker within the application image, so you must ensure that you have a local debug copy of the same image that you are running on your device. The amount of debug information that is stored in the image, and therefore the resulting quality of your debug session, can be affected by the debug and optimization settings passed to the compiler and linker.

It is common to strip out as much of the debug information as possible when running an image on an embedded device. In such cases, try to use the original unstripped image for debugging purposes.

Related information

Connecting to the Snapshot Viewer on page 180
Components of a Snapshot Viewer initialization file on page 178
Considerations when creating debugger scripts for the Snapshot Viewer on page 182

8.2 Components of a Snapshot Viewer initialization file

Describes the groups and sections used to create a Snapshot Viewer initialization file.

The Snapshot Viewer initialization file is a simple text file consisting of one or more sections that emulate the state of the original system. Each section uses an `<option>=<value>` structure.

Before creating a Snapshot Viewer initialization file you must ensure that you have:

- One or more binary files containing a snapshot of the application that you want to analyze.

Note

The binary files must be formatted correctly in accordance with the following restrictions.

- Details of the type of processor.
- Details of the memory region addresses and offset values.
- Details of the last known register values.

To create a Snapshot Viewer initialization file, you must add grouped sections as required from the following list and save the file with `.ini` for the file extension.

[device]

A section for information about the processor or device. The following options can be used:

name

This is the name that is reported from RDDI and is used to identify the device. This value is necessary and must be unique to each device.

class

The general type of device, for example `core`, or `trace_source`.

type

The specific device type, for example `Cortex-A9`, or `ETM`.

location

This describes how the agent that produced the snapshot locates the device.

[dump]

One or more sections for contiguous memory regions stored in a binary file. The following options can be used:

file

Location of the binary file.

address

Memory start address for the specified region.

length

Length of the region. If none specified then the default is the rest of file from the offset value.

offset

Offset of the specified region from the start of the file. If none specified then the default is zero.

[regs]

A section for standard Arm register names and values, for example, `0x0`.

Banked registers can be explicitly specified using their names from the *Arm Architecture Reference Manual*, for example, `R13_fiq`. In addition, the current mode is determined from the Program Status Registers (PSRs), enabling register names without mode suffixes to be identified with the appropriate banked registers.

The values of the PSRs and PC registers must always be provided. The values of other registers are only required if it is intended to read them from the debugger.

Consider:

```
[regs]
CPSR=0x600000D2 ; IRQ
SP=0x8000
R14_irq=0x1234
```

Reading the registers named `SP`, `R13`, or `R13_irq` all yield the value `0x8000`.

Reading the registers named `LR`, `R14`, or `R14_irq` all yield the value `0x1234`.

For more information about Snapshot Viewer file formats, see the documentation in `<installation_directory>\sw\debugger\snapshot`.

Restrictions

The following restrictions apply:

- Consecutive bytes of memory must appear as consecutive bytes in one or more dump files.
- Address ranges representing memory regions must not overlap.

Examples

```
[device]
name=cpu_0
class=core
type=Cortex-A7                ; Selected processor
location=address:0x1200013000
; Location of a contiguous memory region stored in a dump file
[dump]
file="path/ dumpfile1.bin"     ; File location (full path must be specified)
address=0x8000                ; Memory start address for specific region
length=0x0090                 ; Length of region
                              ; (optional, default is rest of file from offset)
; Location of another contiguous memory region stored in a dump file
[dump]
file="path/ dumpfile2.bin"     ; File location
address=0x8090                ; Memory start address for specific region
offset=0x0024                 ; Offset of region from start of file
                              ; (optional, default is 0)
; Arm registers
[regs]
R0=0x000080C8
R1=0x0007C000
R2=0x0007C000
R3=0x0007C000
R4=0x00000363
R5=0x00008EEC
R6=0x00000000
R7=0x00000000
R8=0x00000000
R9=0xB3532737
R10=0x00008DE8
R11=0x00000000
R12=0x00000000
SP=0x0007FFF8
LR=0x0000808D
PC=0x000080B8
```

Related information

Considerations when creating debugger scripts for the Snapshot Viewer on page 182
About the Snapshot Viewer on page 176
Connecting to the Snapshot Viewer on page 180
Arm Architecture Reference Manual

8.3 Connecting to the Snapshot Viewer

Describes how to launch Arm® Debugger from a command-line console or Arm Development Studio IDE, and connect to the **Snapshot Viewer**.

Before you begin

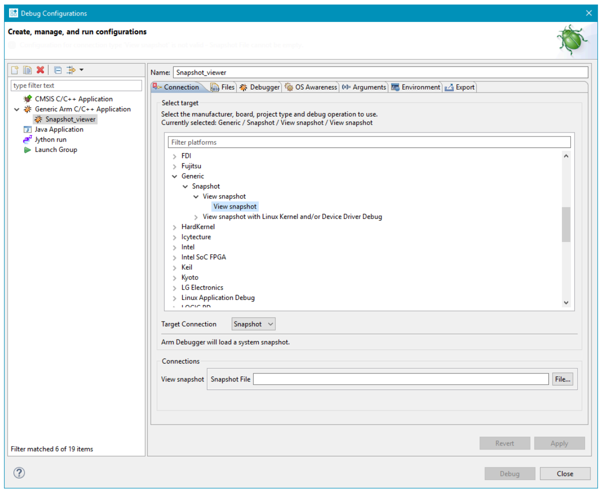
Before connecting, ensure that you have a **Snapshot Viewer** initialization file that contains static information about a target at a specific point in time. For example, your file might contain the contents of registers, memory, and processor state.

About this task

The **Snapshot Viewer** provides a virtual target that you can use to analyze a snapshot of a known system state using the debugger.

Procedure

1. Connect to the **Snapshot Viewer**:

Connecting from Arm Development Studio	Connecting from the command-line
a. Open the **Debug Configurations** dialog box. From the Arm Development Studio menu, click **Run -> Debug Configurations**. b. Select or create a debug configuration under **Generic Arm C/C++ Application**. c. In the **Connection** tab, select **Generic -> Snapshot -> View snapshot -> View snapshot**. **Figure 8-1: Connecting to the Snapshot Viewer through Debug Configurations.**  Debug Configurations Create, manage, and run configurations Name: Snapshot_viewer Select target Select the manufacturer, board, project type and debug operation to use. Currently selected: Generic / Snapshot / View snapshot / View snapshot Target Connection: Snapshot Arm Debugger will load a system snapshot. Connections View snapshot: Snapshot File Filter matched 6 of 19 items d. In the **Connections** section, add your **Snapshot File**. e. Click **Debug**.	a. Launch Arm Debugger in the command-line console. b. Use the `--cdb-entry-param` option to pass your **Snapshot Viewer** initialization file to the debugger: `armdbg --cdb-entry "Generic::Snapshot::View snapshot::View snapshot::View snapshot::Snapshot" --cdb-entry-param "Snapshot File=int.ini" --script=int.cmm`

2. Result: You can now analyze the data from your **Snapshot Viewer** initialization file using Arm Debugger.

Example 8-1: Example

To connect to a Snapshot file from the Arm Development Studio command-line, enter:

```
armdbg --cdb-entry "Generic::Snapshot::View snapshot::View snapshot::View
 snapshot::Snapshot" --cdb-entry-param "Snapshot File=C:\\Arm_DS_Workspace\
\CoreSight_access\\example_captures\\Juno\\snapshot.ini"
```

You can also use the `Connection` parameter to connect to a Snapshot file. At the command prompt, enter:

```
armdbg --cdb-entry "Generic::Snapshot::View snapshot::View snapshot::View
 snapshot::Snapshot" --cdb-entry-param "Connection=C:\\Arm_DS_Workspace\
\CoreSight_access\\example_captures\\Juno\\snapshot.ini"
```

Related information

About the Snapshot Viewer on page 176
Components of a Snapshot Viewer initialization file on page 178
Considerations when creating debugger scripts for the Snapshot Viewer on page 182
Command-line debugger options on page 157
Arm Debugger command-line console keyboard shortcuts on page 174

8.4 Considerations when creating debugger scripts for the Snapshot Viewer

The **Snapshot Viewer** uses an initialization file that emulates the state of the original system. The symbols are loaded from the image using the `data.load.elf` command with the `/nocode /noreg` arguments.

Note

The snapshot data and registers are read-only and so the commands you can use are limited.

The following example shows a script using CMM-style commands to analyze the contents of the `types_m3.axf` image.

```
var.print "Connect and load symbols:"
system.up
data.load.elf "types_m3.axf"  /nocode /noreg
;Arrays and pointers to arrays
var.print ""
var.print "Arrays and pointers to arrays:"
var.print "Value of i_array[9999] is " i_array[9999]
var.print "Value of *(i_array+9999) is " *(i_array+9999)
var.print "Value of d_array[1][5] is " d_array[1][5]
var.print "Values of *((*d_array)+9) is " *((*d_array)+9)
var.print "Values of *((*d_array)) is " *((*d_array))
var.print "Value of &d_array[5][5] is " &d_array[5][5]
```

```
;Display 0x100 bytes from address in register PC
var.print ""
var.print "Display 0x100 bytes from address in register PC:"
data.dump r(PC)++0x100
;Structures and bit-fields
var.print ""
var.print "Structures and bit-fields:"
var.print "Value of values2.no is " values2.no
var.print "Value of ptr_values->no is " ptr_values->no
var.print "Value of values2.name is " values2.name
var.print "Value of ptr_values->name is " ptr_values->name
var.print "Value of values2.name[0] is " values2.name[0]
var.print "Value of (*ptr_values).name is " (*ptr_values).name
var.print "Value of values2.f1 is " values2.f1
var.print "Value of values2.f2 is " values2.f2
var.print "Value of ptr_values->f1 is " ptr_values->f1
var.print ""
var.print "Disconnect:"
system.down
```

Related information

Components of a Snapshot Viewer initialization file on page 178
About the Snapshot Viewer on page 176
Connecting to the Snapshot Viewer on page 180

9 Platform Configuration

In this section we describe how to configure your debug hardware units, hardware, and model platforms in Development Studio.

9.1 Platform Configuration and the Platform Configuration Editor (PCE)

This section describes platform configuration in Development Studio.

You can import and manage configurations for model and hardware platforms using the Development Studio perspective.

To access the Development Studio perspective, from the main menu, select **Window** > **Perspective** > **Open Perspective** > **Other...** > **Development Studio**.

To configure your platforms, use the Development Studio perspective to:

- Create a configuration database.
- View and configure hardware platforms in the Platform Configuration Editor (PCE).
- View and configure models in the Model Configuration Editor.
- Import custom model and hardware platform configurations into a configuration database.
- Create a launch configuration for a model or hardware platform configuration.

9.1.1 Platform Configuration in Development Studio

You can configure your hardware platform and model platform targets using the configuration editors in the Development Studio perspective:

- The Platform Configuration Editor (PCE) enables you create or modify configurations and connections for hardware target platforms. For more information, see Platform Configuration Editor (PCE).
- The Model Configuration Editor enables you to to create or modify configurations and connections for model target platforms. For more information, see Model Configuration Editor.

Use these views to configure debug and trace support information for targets through DSTREAM, ULINK, or model connections.

Warning

If you are autodetecting hardware target information, sometimes it is not possible for Development Studio to read all of the information that it needs from a platform. This can be caused by a variety of issues which are described in Hardware platform bring-up in Development Studio.

If you experience autodetection issues and know the details about the debug system of the platform, use manual platform configuration. For more information, see Manual platform configuration.

The configuration database in Development Studio stores the platform configuration and connection settings in Development Studio. To extend the default Development Studio configuration database, you can create platform configurations in user configuration databases.

Note

If your platform configuration is already in the configuration database, you can use the existing configuration to connect to the platform. For more information, see Debugging Code. You do not have to use the Platform Configuration Editor unless you want to modify the platform configuration.

To create a new configuration, Development Studio uses information from:

- A configuration file for a platform, created and saved using the Platform Configuration Editor (PCE). See Create a platform configuration.
- A configuration file for a model that provides a CADI server, created and saved using the Model Configuration Editor. The model can be already running or you can specify the path and filename to the executable file. See Create a new model configuration.

You can create the following debug operations:

- Single processor and Symmetric Multi Processing (SMP) bare-metal debug for hardware and models.
- Single processor and SMP Linux kernel debug for hardware.
- Linux application debug configurations for hardware.
- big.LITTLE™ configurations for cores that support big.LITTLE operation, such as Cortex®-A15 and Cortex-A7.

Note

For more information on SMP, see Debugging SMP systems.

Debug and Trace Services Layer (DTSL) options are produced for hardware targets with a trace subsystem. These can include:

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

The PCE does not create debug operations that configure non-instruction trace macrocells, except for ITM and STM.

For SMP configurations, the Cross Trigger Interface (CTI) synchronization is used on targets where a suitable CTI is present. A CTI produces a much tighter synchronization with a very low latency, in the order of cycles. Synchronization without using a CTI has a much higher latency, but makes no assumptions about implementation or usage.

Note

The CTI must be fully implemented and connected in line with the Arm reference designs. The CTI must not be used for any other purpose.

For multiplexed pins, you might have to manually configure off-chip Trace Port Interface Unit (TPIU) trace, and also perform calibrations to handle signal timing issues.

Note

Sometimes calibration needs to be performed even if the trace pins are not multiplexed.

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

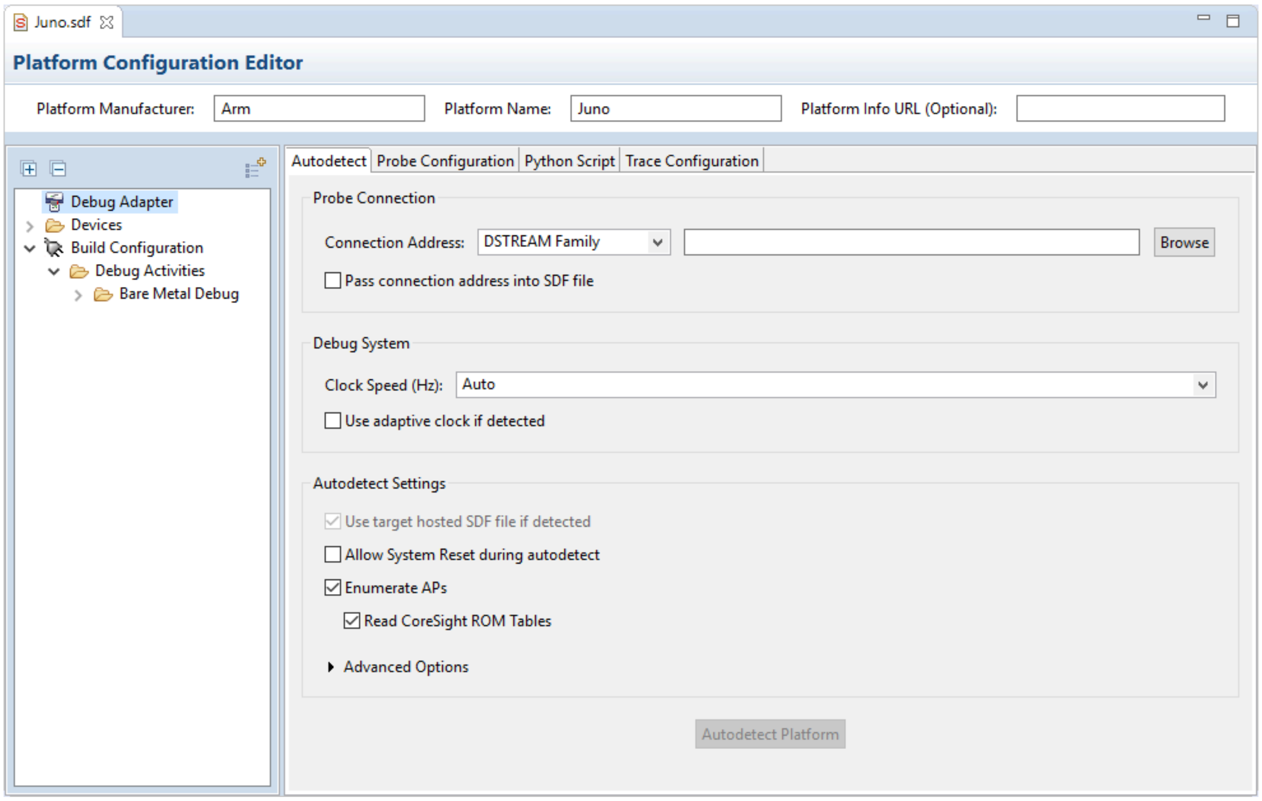
Related information

Platform Configuration Editor (PCE) on page 186
Hardware platform bring-up in Development Studio on page 194
Model Configuration Editor on page 227
Model platform bring-up in Development Studio on page 217
Add a configuration database on page 232
Debug Hardware Firmware Installer view on page 462
Debug Hardware Configure IP view on page 460

9.1.2 Platform Configuration Editor (PCE)

Use the Platform Configuration Editor (PCE) view to create or modify configurations and connections for hardware target platforms.

Figure 9-1: The Platform Configuration Editor (PCE).



Juno.sdf

Platform Configuration Editor

Platform Manufacturer: Arm
Platform Name: Juno
Platform Info URL (Optional):

Debug Adapter
Devices
Build Configuration
Debug Activities
Bare Metal Debug

Autodetect | Probe Configuration | Python Script | Trace Configuration

Probe Connection
Connection Address: DSTREAM Family
Browse
☐ Pass connection address into SDF file

Debug System
Clock Speed (Hz): Auto
☐ Use adaptive clock if detected

Autodetect Settings
☑ Use target hosted SDF file if detected
☐ Allow System Reset during autodetect
☑ Enumerate APs
☑ Read CoreSight ROM Tables
Advanced Options

Autodetect Platform

PCE enables you to easily specify the debug topology by defining the connections between the various processors, CoreSight™ components, and debug IP on the platform. This enables Arm® Debugger to create the DTSL script for the debug connection to the platform.

You can also use the PCE to:

- Review the devices on your development platform.
- Modify device information or add new devices that Arm Debugger was unable to autodetect.
- Configure your debug hardware unit and target-related features that are appropriate to correctly debug on your development platform.
- Review or modify the debug activities for the various processors on the platform.
- Build and save the platform configuration to an RDDI configuration file which Arm Debugger uses to connect to the target processors on your development platform.

Note

To autodetect the devices on the platform, Arm Debugger connects to the platform. It does not maintain the connection to the target after reading the device information.

Related information

Platform Configuration in Development Studio on page 184

Device hierarchy in the PCE view on page 190

9.1.3 PCE with ADIv6 Debug systems

The Arm Debug Interface (ADI) provides access to the debug components in your System on Chip (SoC). The ADI is based on the IEEE 1149.1 JTAG interface, and it is compatible with the CoreSight™ architecture.

The main components of the ADI are split between the Access Port (AP) architecture and the Debug Port (DP) architecture.

The Debug and Access Port (DAP) is an implementation of the ADI.

ADIv6 introduces new functionality for AP and DAP devices, including the ability to nest AP devices. To nest devices, you must specify a Base Address for each nested device. The Platform Configuration Editor (PCE) in Arm Development Studio enables you to manually configure or edit a platform configuration. See Manual platform configuration and Edit a platform configuration. In the PCE, you can also use autodetection to automatically add the base addresses and any nesting of AP devices. For more information, see Create a platform configuration.

Table 9-1: Architecture differences between ADIv5 and ADIv6

Architecture	ADIv5	ADIv6
AP	APv1	APv2 - this architecture is not backwards compatible.
DP	DPv2	DPv3 - this architecture is not backwards compatible.

Example: Manually nest AP devices

To manually nest your AP devices, add them to the device tree using the **Devices Panel**. For each nested device, you must specify its base address.

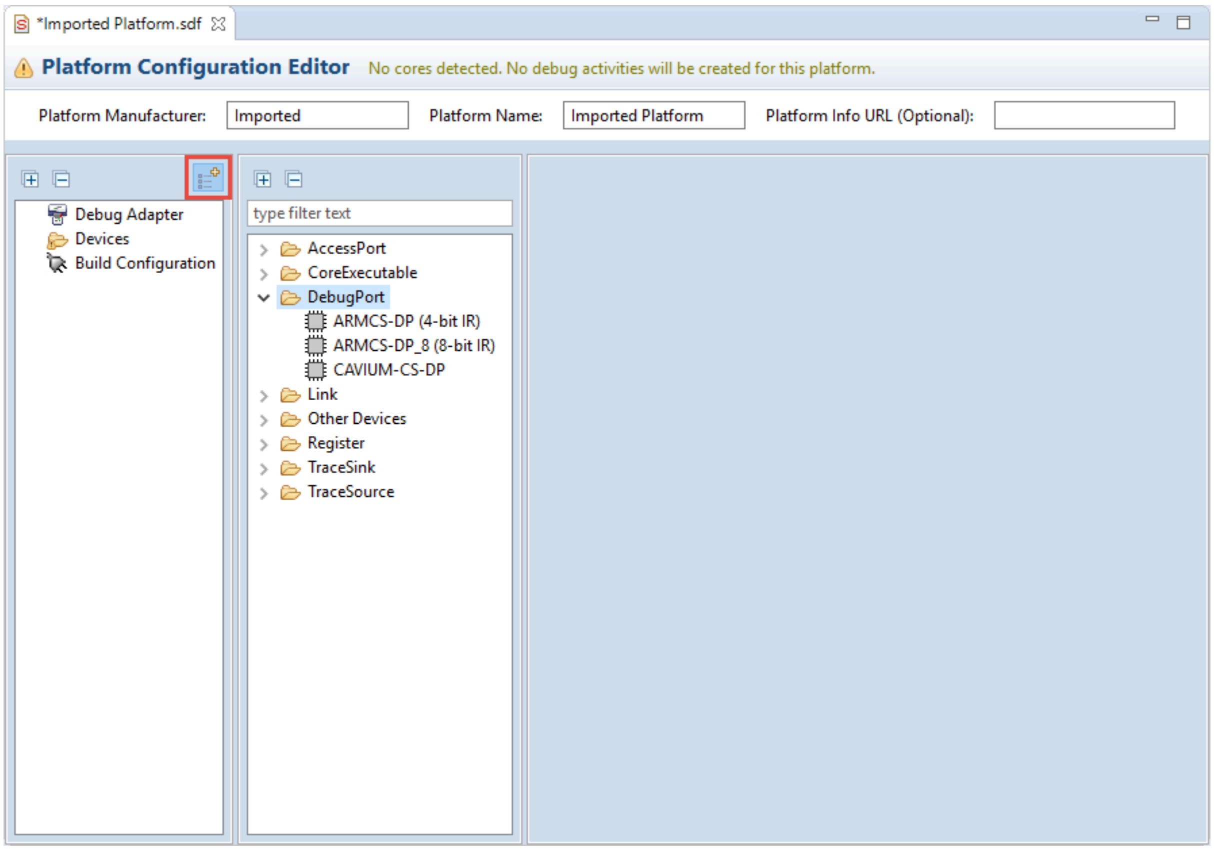
Warning

- You cannot mix and match AP types. If your configuration uses an APv1 device, then your other devices must be APv1 as well. If using an APv2 device, then your other devices must be APv2 devices.
- For ADIv6 systems, all AP devices must be APv2 devices.
- If a system has APv2 devices, all APv2 devices must have a base address.

To specify a base address for a nested AP device:

1. Open the **Devices Panel**. In the PCE, click the **Toggle Devices Panel** icon at the top-right of the Device hierarchy in the PCE view.

Figure 9-2: How to open the device browser


*Imported Platform.sdf

Platform Configuration Editor No cores detected. No debug activities will be created for this platform.

Platform Manufacturer: Imported
Platform Name: Imported Platform
Platform Info URL (Optional):

2. Add an `ARMCS-DP` device as the root device.

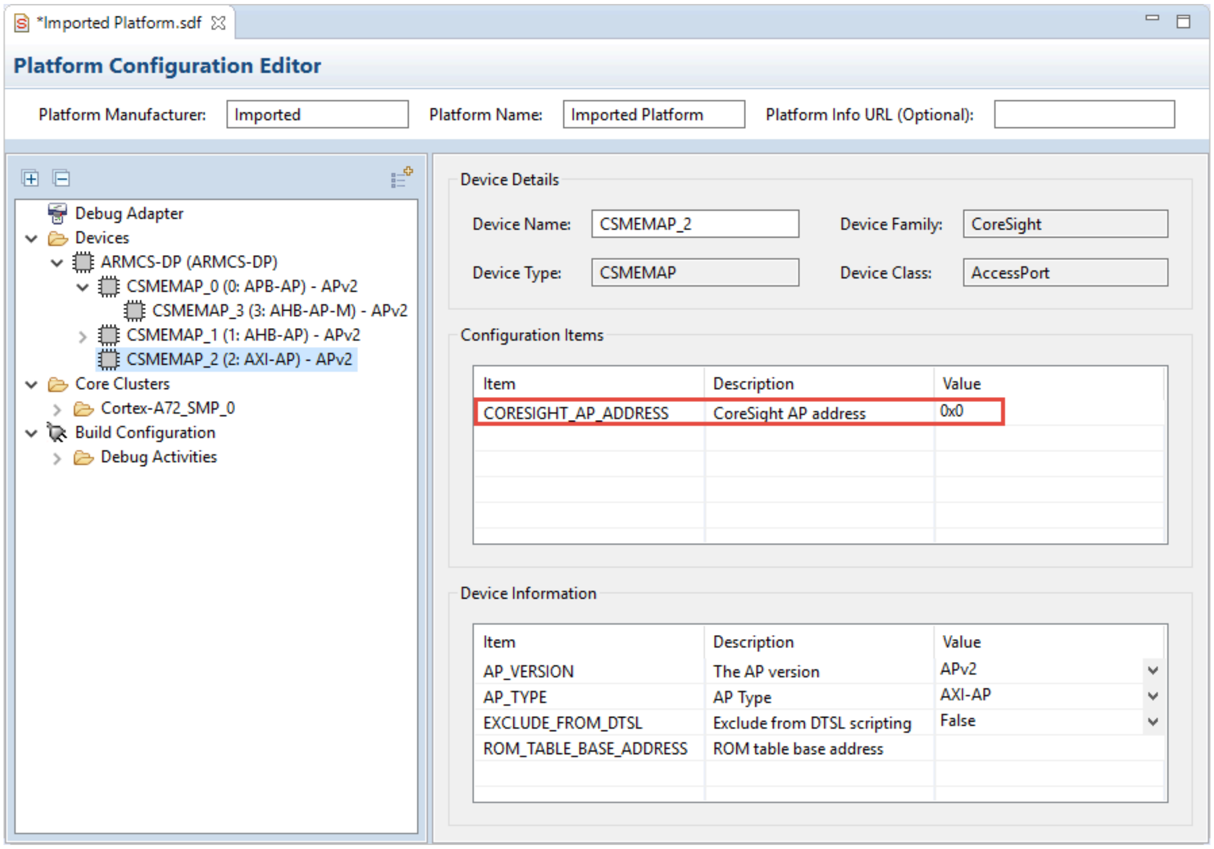
3. Drag-and-drop AP devices. Use the search box at the top of the **Devices Panel** to quickly locate AP devices. Drag them to your device tree.

Note

You can only add AP devices under the root `ARMCS-DP` device, and `CSMEMAP` devices.

4. Specify the APv2 base addresses. For each nested APv2 device, in the **Configuration Items** table specify the base address in the `CORESIGHT_AP_ADDRESS` field:

Figure 9-3: Specifying a Base Address using the PCE


*Imported Platform.sdf
Platform Configuration Editor
Debug Adapter
Devices
ARMCS-DP (ARMCS-DP)
CSMEMAP_0 (0: APB-AP) - APv2
CSMEMAP_3 (3: AHB-AP-M) - APv2
CSMEMAP_1 (1: AHB-AP) - APv2
CSMEMAP_2 (2: AXI-AP) - APv2
Core Clusters
Cortex-A72_SMP_0
Build Configuration
Debug Activities

Platform Manufacturer: Imported
Platform Name: Imported Platform
Platform Info URL (Optional):

Device Details

Device Name: CSMEMAP_2
Device Family: CoreSight
Device Type: CSMEMAP
Device Class: AccessPort

Configuration Items

Item	Description	Value
CORESIGHT_AP_ADDRESS	CoreSight AP address	0x0

Device Information

Item	Description	Value
AP_VERSION	The AP version	APv2
AP_TYPE	AP Type	AXI-AP
EXCLUDE_FROM_DTSL	Exclude from DTSL scripting	False
ROM_TABLE_BASE_ADDRESS	ROM table base address	

Note

- If you input an invalid address, the PCE reverts the value to the previous valid address (which might be the default).
- For an APv1 device, set the CoreSight AP index using the `CORESIGHT_AP_INDEX` configuration item.

5. Save and build the platform. Select **File** > **Save**.

Related information

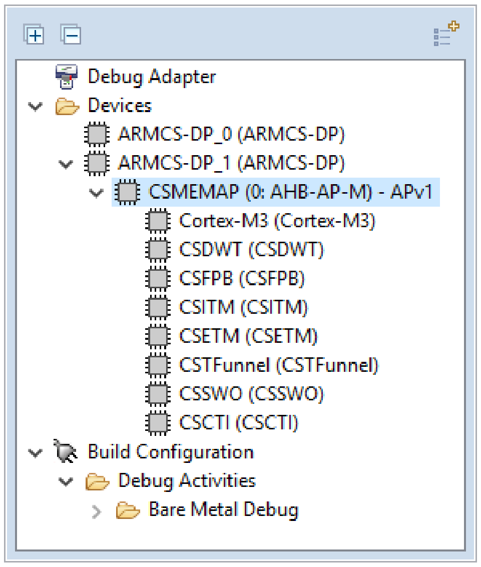
Platform Configuration Editor (PCE) on page 186
Create a platform configuration on page 195
Manual platform configuration on page 212
Edit a platform configuration on page 203

9.1.4 Device hierarchy in the PCE view

The device hierarchy in the PCE view shows the devices on the platform.

In the PCE view, you can add or remove devices to configure the platform for how you want to debug it. This figure shows the device hierarchy of an example platform. Development Studio might not autodetect all the devices on the platform. To use these undetected devices in the debug session, you must add them to the device hierarchy and configure them.

Figure 9-4: Device hierarchy



```
Debug Adapter
Devices
    ARMCS-DP_0 (ARMCS-DP)
    ARMCS-DP_1 (ARMCS-DP)
        CSMEMAP (0: AHB-AP-M) - APv1
            Cortex-M3 (Cortex-M3)
            CSDWT (CSDWT)
            CSFPB (CSFPB)
            CSITM (CSITM)
            CSETM (CSETM)
            CSTFunnel (CSTFunnel)
            CSSWO (CSSWO)
            CSCTI (CSCTI)
Build Configuration
    Debug Activities
        Bare Metal Debug
```

If you do not need some of the autodetected devices, you can remove them from the device hierarchy. To remove a device, right click on the device and select **Remove Device**.

Access

To access, either:

- In the Project Explorer, right-click on an SDF file, and select **Open With** > **Platform Configuration Editor**.
- Double-click an SDF file (where the SDF association has not been overridden).
- In the Project Explorer, right-click, and select **File** > **New** > **Platform Configuration**. After autodetection or manual configuration of a platform, the PCE view opens.
- Select **File** > **New** > **Other...** > **Configuration Database** > **Platform Configuration**. After autodetection or manual configuration of a platform, the PCE view opens.
- The hardware connection dialog box provides the option to enter the Platform Configuration Editor (PCE) at the final stages of new connection creation: **File** > **New** > **Hardware Connection**. At the target selection step, click **Add a new platform...**. It can also be accessed at the end of the target selection flow for a CMSIS device; click **Target Configuration**.

Contents

The context menu for the device hierarchy contains:

Table 9-2: Device hierarchy view contents

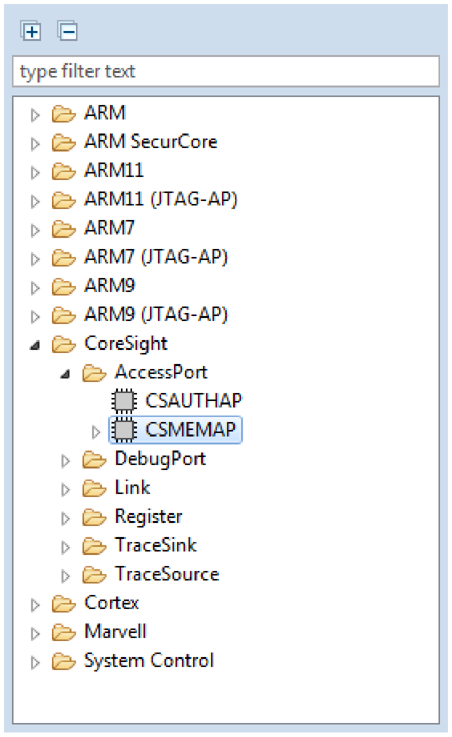
Field	Description
Toggle Devices Panel	Shows or hides the **Devices Panel** which lists the devices that you can add to the JTAG scan chain or device hierarchy.
Debug Adapter	Shows the configuration for your debug hardware unit, for example, a DSTREAM unit. For more information on the contents displayed, see Debug Adapter configuration in the PCE.
Devices	Shows the scan chain and device hierarchy of your platform. The device hierarchy usually consists of one or more Debug Access Ports (DAP). Each DAP consists of one or more Access Ports (AP). Each AP shows the devices that have been detected through that access port. • Device Table - Shows information about the RDDI ID, device name, type, family, class, AP Index, and Base Address. • Component Connections - Shows component connection information, including master, slave, and link type, details, and origin.
Device	Shows the device name, along with device and configuration information. See Device configuration panel for more information.
Debug Activities	Shows the type of debug activities you can perform on the target. The debug activities are accessible from the **Debug Configurations** dialog box when you want to start a debug session.
Enumerate APs	Available for Debug Access Ports (DAP) on the device hierarchy. This enumerates the Access Ports under the DAP.
Read CoreSight™ ROM Tables	Reads the CoreSight ROM tables to obtain more information about the devices from the various access ports. This might cause certain devices on the platform to become unresponsive. If so, during autodetection and after selecting your debug hardware **Probe**, deselect **Read CoreSight ROM Tables** under the **Autodetect** tab in the **Debug Adapter** pane.
Add Custom JTAG Device	Adds a custom device to the JTAG scan chain.
Add core cluster components	Opens a dialog box which you can use to add a cluster of cores and their associated devices (ETM/PTM, CTI, and PMU) in groups. If a base address is specified, then base addresses for all components are set, and topology links are added between all added components. If you select the option to add funnel connections, all ETM/PTM devices are linked to the funnel specified by the Funnel Base Address. If there is no device at this address, a new funnel is created.
Autodetect Component Connections	Starts the autodetection. It detects the connections between the various components on the platform.
Add Link From This Device	Adds a topology link between the selected device and another device. The selected device is the link master. If no links from the device can be created (for example, the device is already linked, or there are no devices to which a valid link can be made) then this menu item is not available.

Field	Description
Add Link To This Device	Adds a topology link between the selected device and another device. The selected device is the link slave. If no links to the device can be created (for example, the device is already linked, or there are no devices from which a valid link can be made) then this menu item is not available.
Remove Device	Removes the device from the device hierarchy.

Usage

To add a device as a sibling or as a child, drag-and-drop from the **Devices Panel** to the appropriate place in the device hierarchy.

Figure 9-5: Devices Panel



Any device that you add or remove from the hierarchy changes the topology of the SoC. You must ensure that the topology is appropriate for your platform. After adding new devices, you can configure the devices in the right-hand pane in the PCE view.

Related information

Platform Configuration Editor (PCE) on page 186
Debug Adapter configuration in the PCE on page 290
Add core cluster components dialog box on page 216

9.2 Hardware targets

This section describes how to configure hardware platforms in Development Studio.

9.2.1 Hardware platform bring-up in Development Studio

Effective debug and trace support requires that Arm® Debugger has the necessary information about the platform it needs to debug.

The complexity of modern System-on-Chip (SoC) based platforms is increasing. The debugger needs to know:

- What devices are present on the SoC.
- The type and configuration details of each device.
- The base addresses of the CoreSight™ components.
- The type and index of the Access Ports (AP) to access the various CoreSight components.
- How the different devices relate or connect to each other (their topology).

Development Studio automatically detects most of this information, enabling a simple and efficient platform bring-up process for all Arm CoreSight-based SoCs. However, it is common that Development Studio is unable to detect certain features on a complex SoC. The reasons might be:

- The SoC does not make the information available to the debugger.
- The information from the SoC might be missing when parts of the SoC are powered down.
- Devices inside the SoC might interfere with topology detection.
- JTAG routing or security devices might prevent Development Studio from discovering details of physical JTAG devices.
- Debug or trace might be partially or fully disabled.
- Devices might be powered down, or their clocks might be disabled. This can make devices unresponsive to requests for information, and can affect individual devices, entire clusters, or all the devices in a ROM table.
- ROM tables might be missing, incomplete, or at the wrong address.
- Integration Test registers might not be fully implemented, or their operation might be limited by other devices.
- The platform might contain unsupported devices.
- Component IDs are not recognized, or Development Studio does not know how to correctly detect their connections. This might cause autodetection to fail.

Development Studio does not make any assumptions about the platform configuration. If Development Studio has limited information about the platform, it can only provide limited debug and trace functionality. This means that it is common for Arm Debugger to provide limited trace functionality for certain processors on the platform. You can add to the automatically detected platform configuration by manually providing data based on your knowledge of the SoC, for more information see Manual platform configuration. Manual platform configuration enables you to provide any missing parts of the topology definition and generate correct DTSL script.

Related information

Create a platform configuration on page 195
Edit a platform configuration on page 203
Manual platform configuration on page 212
Custom devices on page 214

9.2.2 Create a platform configuration

Use the **New Platform** dialog box in Development Studio to create debug configurations for hardware platforms.

Before you begin

- If you autodetect your target, ensure you connect your debug adapter and targets, or that you have the connection address.
- If you import from an existing RDDI configuration file, SDF file (`*.rcf`, `*.rvc`, `*.sdf`), CoreSight Creator file (`*.xml`), or CMM script (`*.cmm`), ensure you have access to these files.

About this task

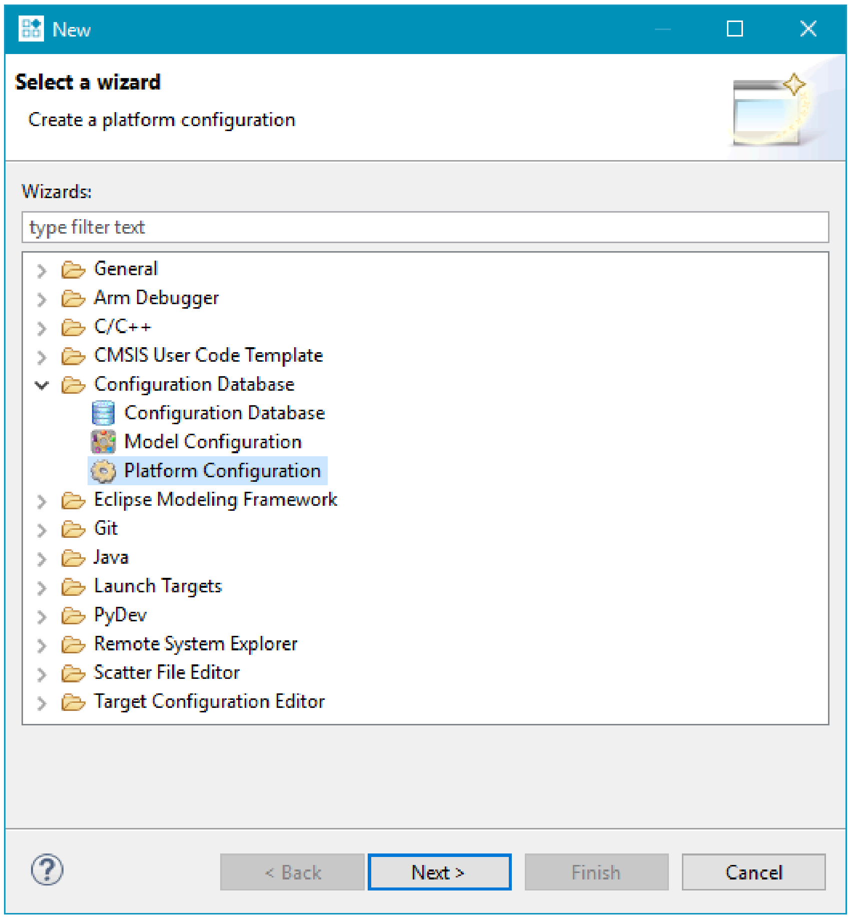
Warning

If you choose to autodetect your platform, depending on your target, you might receive warnings and errors. If Development Studio is unable to detect the connection information from the platform, no assumptions are made about how the devices are connected. You must provide this information in the PCE view. For more information on the possible causes, see Hardware platform bring-up in Development Studio.

Procedure

1. Open the new project dialog box. In the **Project Explorer**, right-click, and select **File** > **New** > **Platform Configuration**.
2. Select **Configuration Database** > **Platform Configuration** and then click **Next** .

Figure 9-6: Select create a platform configuration.



New

Select a wizard

Create a platform configuration

Wizards:

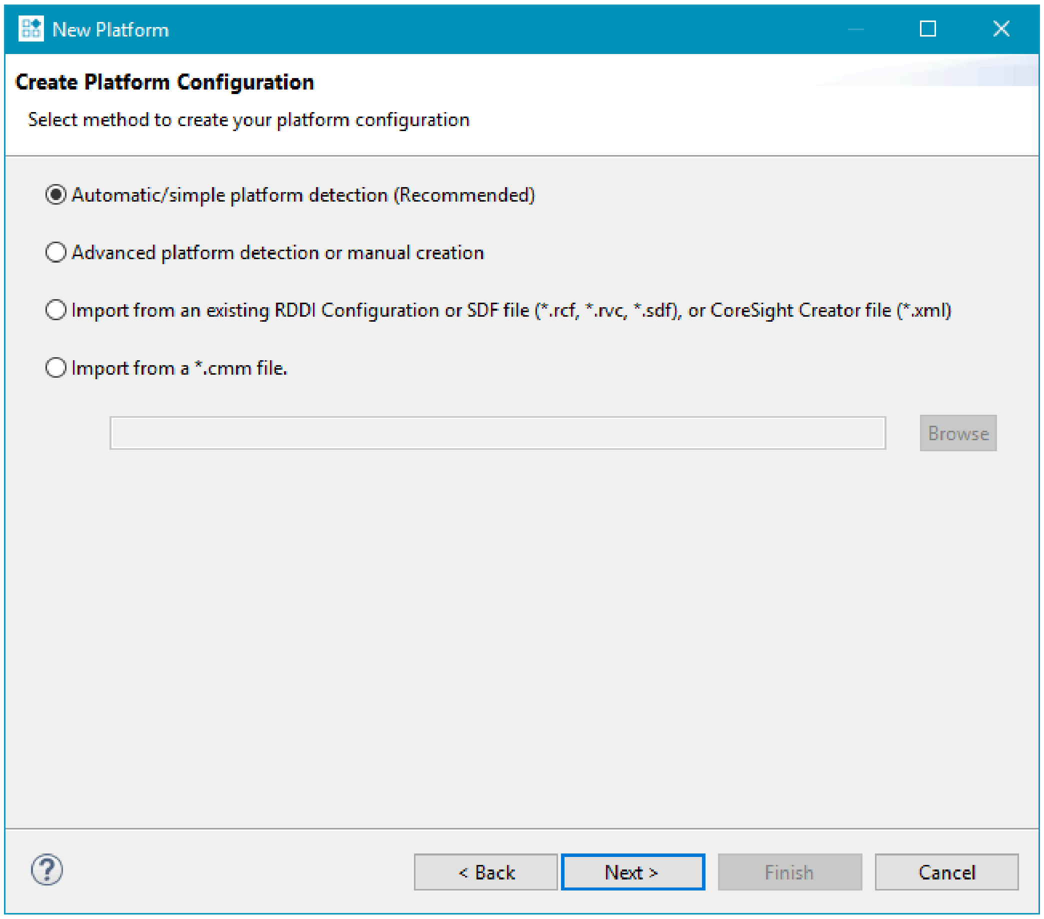
type filter text

- General
- Arm Debugger
- C/C++
- CMSIS User Code Template
- Configuration Database
  - Configuration Database
  - Model Configuration
  - Platform Configuration
- Eclipse Modeling Framework
- Git
- Java
- Launch Targets
- PyDev
- Remote System Explorer
- Scatter File Editor
- Target Configuration Editor

This shows the **Create Platform Configuration** dialog box.

3. Select a method to create the configuration for your platform and click **Next**.

Figure 9-7: Platform creation options


New Platform

Create Platform Configuration

Select method to create your platform configuration

(●) Automatic/simple platform detection (Recommended)

( ) Advanced platform detection or manual creation

( ) Import from an existing RDDI Configuration or SDF file (*.rcf, *.rvc, *.sdf), or CoreSight Creator file (*.xml)

( ) Import from a *.cmm file.

Choose from:

- **Automatic/simple platform detection**

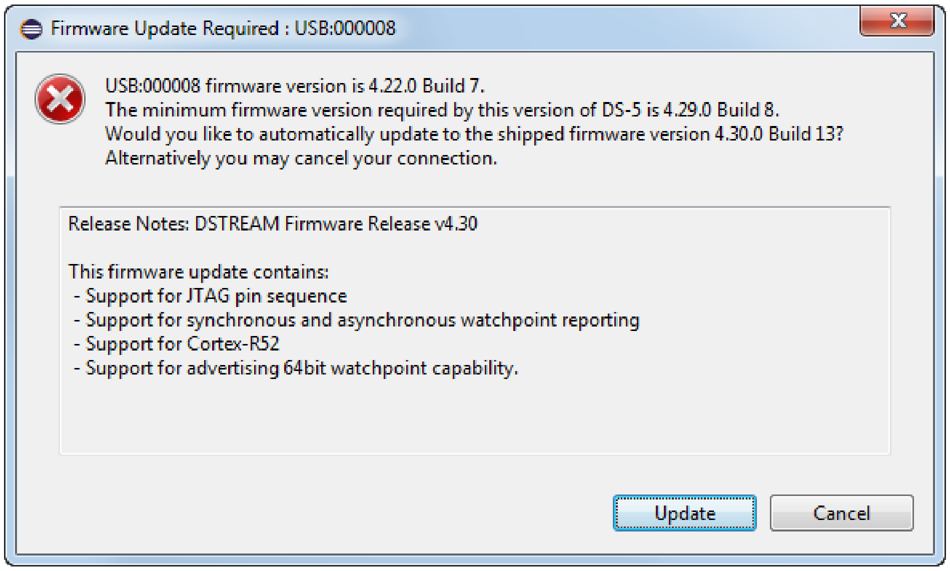
  Note: Arm recommends this option.

  To automatically detect the devices present on your platform, use this option. After autodetection, you can add more devices and specify how the devices are interconnected.

  You must supply the details for the debug hardware adapter attached to your platform, or to specify its **Connection Address**.

  If you are using outdated firmware, Development Studio warns you during the platform detection process. For example:

Figure 9-8: Debug hardware firmware update notification during platform configuration



USB:000008 firmware version is 4.22.0 Build 7.
The minimum firmware version required by this version of DS-5 is 4.29.0 Build 8.
Would you like to automatically update to the shipped firmware version 4.30.0 Build 13?
Alternatively you may cancel your connection.

Release Notes: DSTREAM Firmware Release v4.30

This firmware update contains:
- Support for JTAG pin sequence
- Support for synchronous and asynchronous watchpoint reporting
- Support for Cortex-R52
- Support for advertising 64bit watchpoint capability.

To update your debug hardware firmware, click **Update**.

- **Advanced platform detection or manual creation**

  Gives you control over the individual stages that are involved in reading the device information from your platform. For more information, see Manual platform configuration.

  Note

  This option is useful to read certain device information that can make the platform unresponsive.

- **Import from an existing RDDI configuration file or SDF file (*.rcf, *.rvc, *.sdf), or CoreSight Creator file (*.xml)**

  Note

  Use this option if you already have a configuration file for your platform.

  Imports minimal information about the components available on your platform, such as the base address. After importing, you can manually provide additional information and links between the components to enable full debug and trace support.

- **Import from a *.cmm file**

  Imports a platform configuration from a CMM script (`*.cmm`) file.

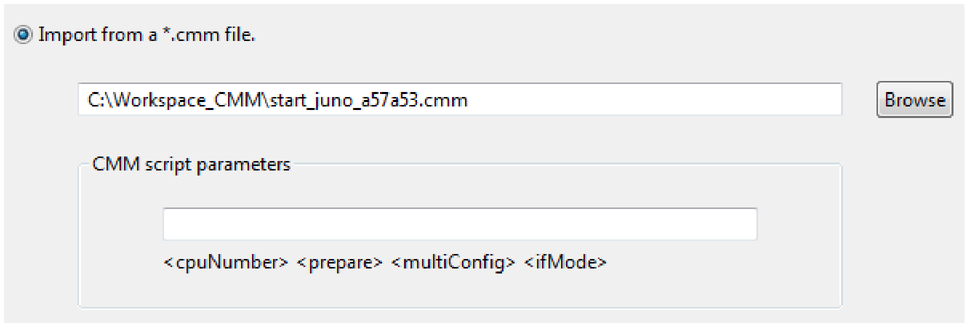
  CMM scripts can contain target description information such as:

- JTAG pre- and post- IR/DR bit information.
- Core types.
- Device base addresses.
- CoreSight topology information.

The PCE uses this information to create a Development Studio platform configuration, complete with custom DTSL control tabs for trace configuration.

Some CMM scripts describe different targets (or different cores and trace devices in the same target) depending on the value of parameters that are passed to the script. If a CMM script requires parameters, enter them in the **CMM script parameters** field.

Figure 9-9: Enter CMM script parameters



Import from a *.cmm file.

C:\Workspace_CMM\start_juno_a57a53.cmm

CMM script parameters

<cpuNumber> <prepare> <multiConfig> <ifMode>

Warning

To detect a new platform, ensure your debug hardware has the minimum firmware version for the Arm® Development Studio version installed.

If autodetecting using **Automatic/simple platform detection** or **Advanced platform detection or manual creation** (using **Autodetect Platform** in the **Debug Adapter** panel), Development Studio connects to the platform and reads all the device information that it can from the platform.

Warning

Depending on your target, you might receive warnings and errors. Where Development Studio is unable to obtain this information from the platform, no assumptions are made about how the devices are connected. You must provide this information in the PCE view.

Debug functionality will succeed if all cores are correctly detected. Where the connections between cores are not detected, debug will succeed but trace and cross-triggering functionality might be limited or missing.

4. (Optional - **Automatic/simple platform detection** only) Save or edit your autodetected configuration. The following list describes the next steps available for your autodetected platform:

- **Save a Debug-Only Arm DS Platform Configuration**

  Saves a debug-only configuration to your configuration database.
- **Save a Debug and Trace Arm DS Platform Configuration**

  Saves a debug and trace configuration to your configuration database. You can open this in Development Studio later to modify it or you can use it to connect to and debug the platform later.
- **Edit platform in Arm DS Platform Configuration Editor**

  Saves the configuration to your configuration database and opens the PCE view.

  In the PCE view, you can provide information about the platform that Development Studio was unable to autodetect.

  For more information, see Edit a platform configuration.

Note

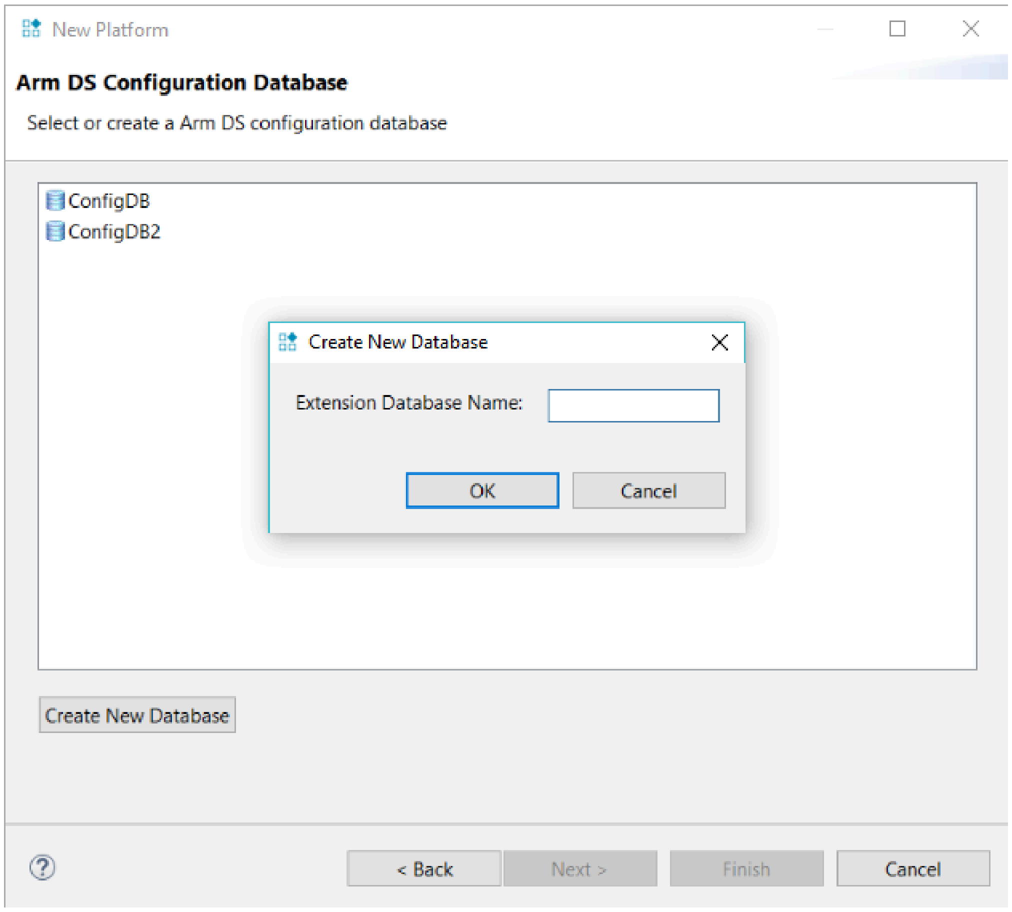
- All Development Studio platform configurations must be stored in a configuration database.
- You can open the configurations in Development Studio later to modify it or you can use it to connect to and debug the platform later.

Select an option and click **Next**.

The **New Platform** dialog box opens.

5. Select an existing configuration database from the list, or create a new one. To create a new configuration database, click **Create New Database**, provide a name for your new configuration database in the prompt, then click **OK** to save your configuration database.

Figure 9-10: Create new configuration database



New Platform

Arm DS Configuration Database

Select or create a Arm DS configuration database

ConfigDB
ConfigDB2

Create New Database

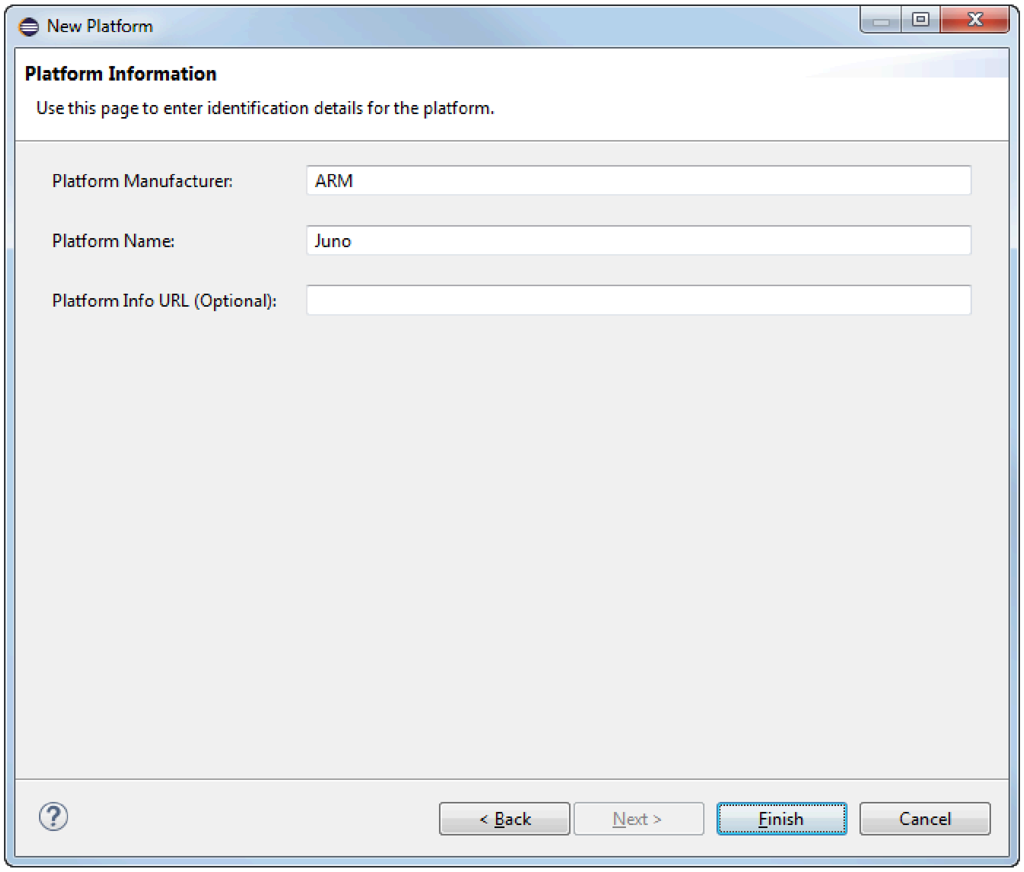
Extension Database Name:

6. Click **Next >**.
   The **Platform Information** dialog box opens.
7. Enter the **Platform Manufacturer**, for example Arm. Enter the **Platform Name**, for example **Juno** . Optionally, if you want to provide a URL link to information about the platform, enter it in **Platform Info URL**.

Note

When you select a debug activity for the platform, the URL appears in the **Debug Configurations** panel.

Figure 9-11: New platform information



Platform Information

Use this page to enter identification details for the platform.

Platform Manufacturer: ARM

Platform Name: Juno

Platform Info URL (Optional):

8. Click **Finish**.
9. (Optional - **Advanced platform detection or manual creation** only) Complete your configuration:
    - Configure your debug hardware in **Debug Adapter** panel.
    - Autodetect your platform. In **Debug Adapter** panel under **Autodetect** tab, click **Autodetect Platform**.

Warning

Depending on your target, you might receive warnings and errors. Where Development Studio is unable to obtain this information from the platform, no assumptions are made about how the devices are connected. You must provide this information in the PCE view.

Always review the information that has been collected before deciding what further action to take. If Development Studio fails to read information, it might be an indication of a deeper problem. For more information, see Hardware platform bring-up in Development Studio.

- Check, edit, and save your configuration. If required, edit the configuration then select **File** > **Save**.

Results

The **Platform Configuration Editor** (PCE) view opens in the right-hand panel. You can view the configuration database and platform in the **Project Explorer**.

Related information

Manual platform configuration on page 212
Edit a platform configuration on page 203
Platform Configuration in Development Studio on page 184
Configuration Database panel on page 230

9.2.3 Edit a platform configuration

You can use the PCE view to identify missing component connections and to add them to your platform configuration.

Before you begin

- You need a platform configuration available to edit.
- You need the topological information that describes your platform.

About this task

After you create a new platform configuration in Development Studio, you can review it in the Platform Configuration Editor (PCE). Development Studio might not detect all the devices on the platform or might not know how the devices are connected to each other. You can use the **Component Connections** table in PCE to:

- Describe the relationship between the cross-triggers.
- Describe the trace topology, for example which trace source is connected to which trace sink.

Warning

Cross-triggering or trace might not work if the fields within the component connections table are not correctly populated. When the CTI trigger links are not detected, SMP debug reverts to use loose synchronization. Loose synchronization is when the cores are being stopped separately by the debugger, instead of using the Cross-Trigger Matrix.

For a more detailed description of the possible reasons Development Studio might not autodetect your platform correctly, see Hardware platform bring-up in Development Studio.

Procedure

1. Select **Devices** in the Device hierarchy in the PCE view.
2. Navigate to the component connection information. In the right-hand pane, select the **Component Connections** tab.
3. Add a new component connection to a master or slave device:
   a) Click **Add Link**. This shows the **Add Link** dialog box.
   b) Select a device for the **Master**, the <master-device>, and select a device for the **Slave**, the <slave-device>. Click **Ok** to add the component connection.

Note

Depending on the device you add a component connection for, you might have to specify additional parameters. For example:

- If you add a CTI as a master, then you must specify the trigger output port.
- If you add a CTI as a slave, then you must specify the trigger input port.
- If you add a Trace Replicator as a master, then you must specify the master interface.
- If you add a Trace Funnel as a slave, then you must specify the slave interface.

The **Component Connections** tab shows the user-added component connections in the PCE view.

4. Save the platform configuration. Select **File** > **Save**.
5. Customize the build. By default, CTI synchronization and trace are enabled. When CTI sync is failing or causing problems, it is useful to disable these options. To build the platform configuration without these options, deselect them in the **Platform Builder** tab in the **Properties** dialog box and select **Apply**.
   When starting from scratch with a complex system, to test the connection with your target and debug your cores, you might prefer to disable trace in the configuration.
6. Build the platform configuration. Right-click the project in the **Project Explorer** view and select **Build Platform**.
   If the PCE suspects that some topology information is missing, a **The system topology (component connections) may not be correct** dialog box appears. This can happen, for example, if you are using trace components that do not have a programming model. If the dialog box appears:

   - Select **Full Debug and Trace** to regenerate the debug configuration files with full debug and trace information.
   - Select **Debug Only** to regenerate the debug configuration files that only contain the configuration for a debug session without trace capability.
   - Select **Return to PCE** to manually provide the component interconnect information.

Note

If you return to the PCE, check that the component interconnect information is accurate. If required, edit the information, save the platform configuration, and rebuild the platform configuration.

---

7. Select your platform and debug activity:
   a) Open the **Debug Configurations** view. Right-click in the **Project Explorer** view and select **Debug As** > **Debug Configurations...**.
   b) Select your platform and debug activity from the **Connection** tab in the **Debug Configurations** dialog box.

Note

To see your platform in the **Debug Configurations** list, your configuration database must be specified in **Window** > **Preferences**. Expand **Arm DS** and select **Configuration Database**.

---

8. Check your new device is trace capable for the <master-device> processor. In the **Connection** tab in the **Debug Configurations** dialog box, click **Edit** on **DTSL Options**.

Related information

Hardware platform bring-up in Development Studio on page 194
Create a platform configuration on page 195
Manual platform configuration on page 212
Device hierarchy in the PCE view on page 190

9.2.4 Add topology information for an autodetected Cortex-M3 processor

If it is not added automatically, you can add topology information for an autodetected platform. The following example shows you how to add trace topology information for an autodetected Cortex®-M3 processor.

Before you begin

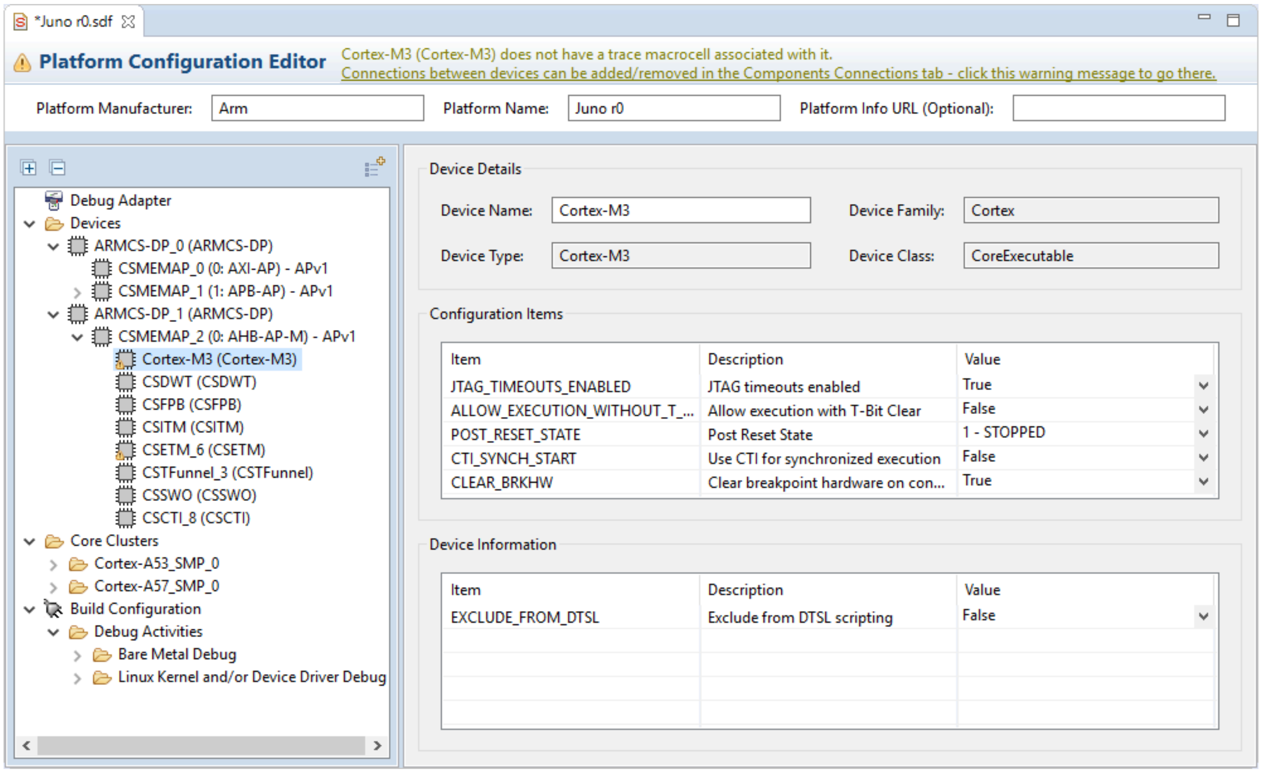
- You need a Cortex-M3 processor platform configuration available to edit.
- You need the topological information that describes your platform.

About this task

This example shows how to edit topology information for a specific platform configuration. See Edit a platform configuration for general instructions on editing a platform configuration.

The figure shows the configuration of an example platform in the Platform Congiguration Editor (PCE) view after autodetection in Development Studio.

Figure 9-12: Component Connections


*Juno r0.sdf

Platform Configuration Editor Cortex-M3 (Cortex-M3) does not have a trace macrocell associated with it.
Connections between devices can be added/removed in the Components Connections tab - click this warning message to go there.

Platform Manufacturer: Arm Platform Name: Juno r0 Platform Info URL (Optional):

- Debug Adapter
- Devices
  - ARMCS-DP_0 (ARMCS-DP)
    - CSMEMAP_0 (0: AXI-AP) - APv1
    - CSMEMAP_1 (1: APB-AP) - APv1
  - ARMCS-DP_1 (ARMCS-DP)
    - CSMEMAP_2 (0: AHB-AP-M) - APv1
      - Cortex-M3 (Cortex-M3)
      - CSDWT (CSDWT)
      - CSFPB (CSFPB)
      - CSITM (CSITM)
      - CSETM_6 (CSETM)
      - CSTFunnel_3 (CSTFunnel)
      - CSSWO (CSSWO)
      - CSCTI_8 (CSCTI)
- Core Clusters
  - Cortex-A53_SMP_0
  - Cortex-A57_SMP_0
- Build Configuration
  - Debug Activities
    - Bare Metal Debug
    - Linux Kernel and/or Device Driver Debug

Device Details

Device Name: Cortex-M3 Device Family: Cortex
Device Type: Cortex-M3 Device Class: CoreExecutable

Configuration Items

Item	Description	Value
JTAG_TIMEOUTS_ENABLED	JTAG timeouts enabled	True
ALLOW_EXECUTION_WITHOUT_T_...	Allow execution with T-Bit Clear	False
POST_RESET_STATE	Post Reset State	1 - STOPPED
CTI_SYNCH_START	Use CTI for synchronized execution	False
CLEAR_BRKHW	Clear breakpoint hardware on con...	True

Device Information

Item	Description	Value
EXCLUDE_FROM_DTSL	Exclude from DTSL scripting	False

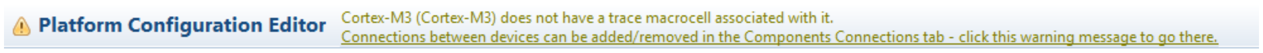
The title bar of the PCE view warns about the information that autodetection was unable to determine.

Note

To get more information, click on the device in the device hierarchy.

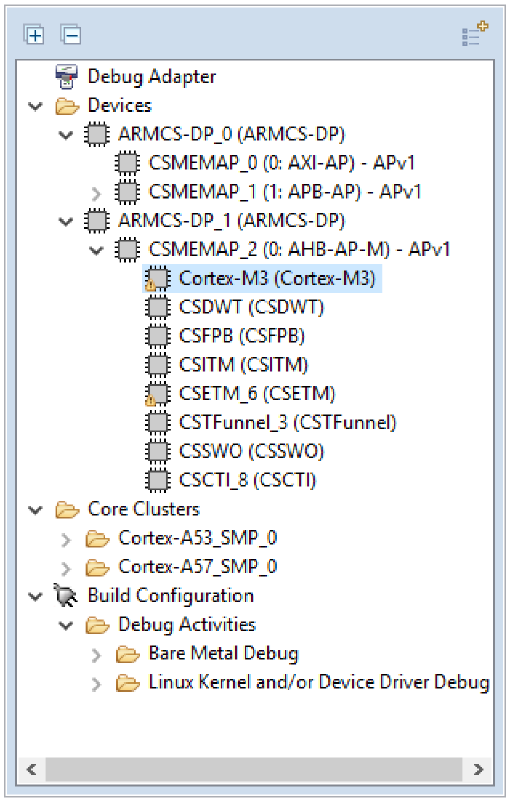
When you select the Cortex-M3 processor, Arm® Debugger displays a warning message if there is any missing information.

Figure 9-13: Missing trace macrocell


Platform Configuration Editor Cortex-M3 (Cortex-M3) does not have a trace macrocell associated with it.
Connections between devices can be added/removed in the Components Connections tab - click this warning message to go there.

In this example, the Cortex-M3 processor does not have a trace macrocell associated with it. Autodetection was unable to determine the topology information for it. The Device hierarchy in the PCE view shows that the Cortex-M3 processor and the trace source, `CSETM_6`, are under the same access port, `CSMEMAP_2`.

Figure 9-14: Device hierarchy



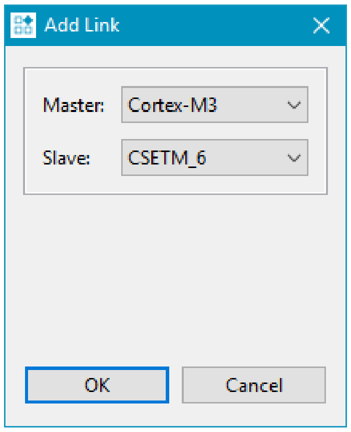
Debug Adapter
Devices
ARMCS-DP_0 (ARMCS-DP)
CSMEMAP_0 (0: AXI-AP) - APv1
CSMEMAP_1 (1: APB-AP) - APv1
ARMCS-DP_1 (ARMCS-DP)
CSMEMAP_2 (0: AHB-AP-M) - APv1
Cortex-M3 (Cortex-M3)
CSDWT (CSDWT)
CSFPB (CSFPB)
CSITM (CSITM)
CSETM_6 (CSETM)
CSTFunnel_3 (CSTFunnel)
CSSWO (CSSWO)
CSCTI_8 (CSCTI)
Core Clusters
Cortex-A53_SMP_0
Cortex-A57_SMP_0
Build Configuration
Debug Activities
Bare Metal Debug
Linux Kernel and/or Device Driver Debug

The following procedure explains how to edit a platform configuration and add a component connection between a Cortex-M3 processor and the Embedded Trace Macrocell (ETM):

Procedure

1. Select **Devices** in the Device hierarchy in the PCE view.
2. In the right-hand pane, select the **Component Connections** tab.
3. Add a new component connection:
   a) Click **Add Link**. This shows the **Add Link** dialog box.
   b) Select Cortex-M3 for the **Master** and select `CSETM_6` for the **Slave**. To add the component connection, click **OK**.

Figure 9-15: Add Core Trace

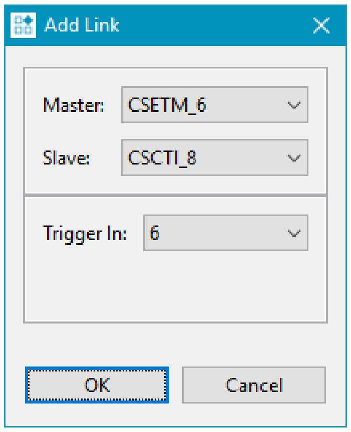


Master: Cortex-M3

Slave: CSETM_6

c) Add slave connections. The PCE view shows that `CSETM_6` does not have any slave connections. The device hierarchy shows that there is a `CSCTI_8` component under the same access port. To add this component connection to `CSETM_6`, click **Add Link** and select `CSETM_6` for the **Master**, and `CSCTI_8` for the **Slave** .

Figure 9-16: Add CTI Trigger



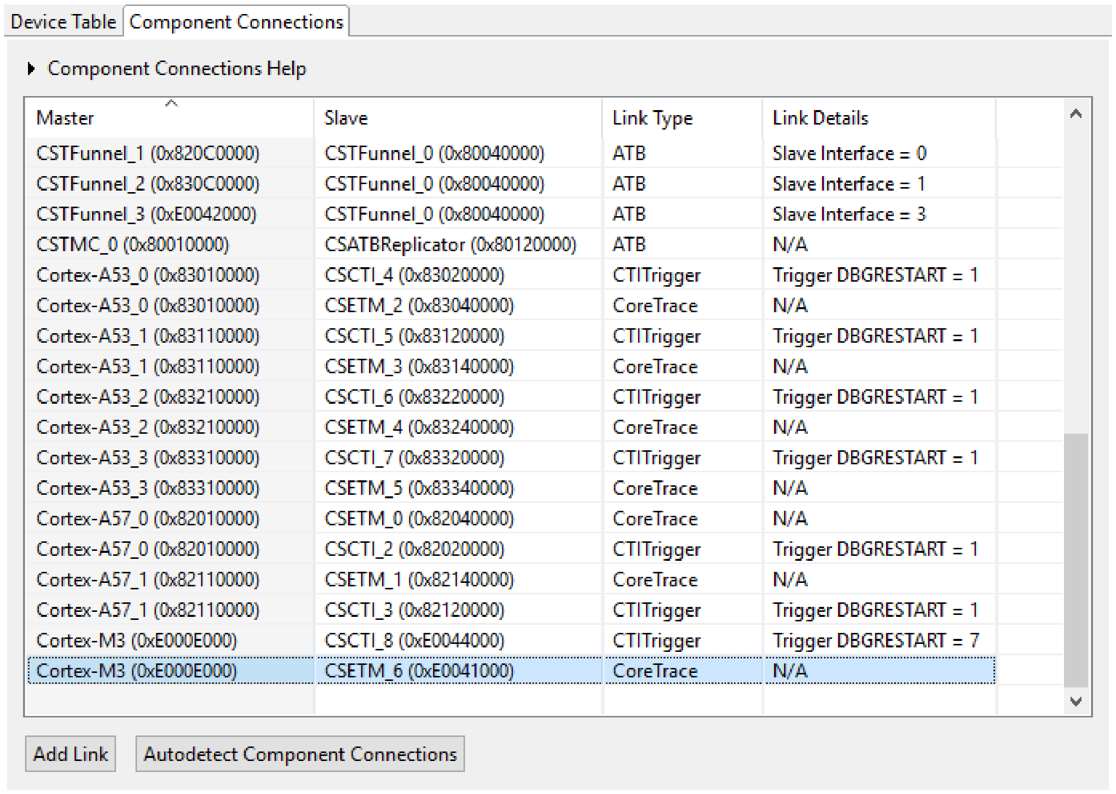
Master: CSETM_6

Slave: CSCTI_8

Trigger In: 6

4. Save the platform configuration. Select **File** > **Save**.

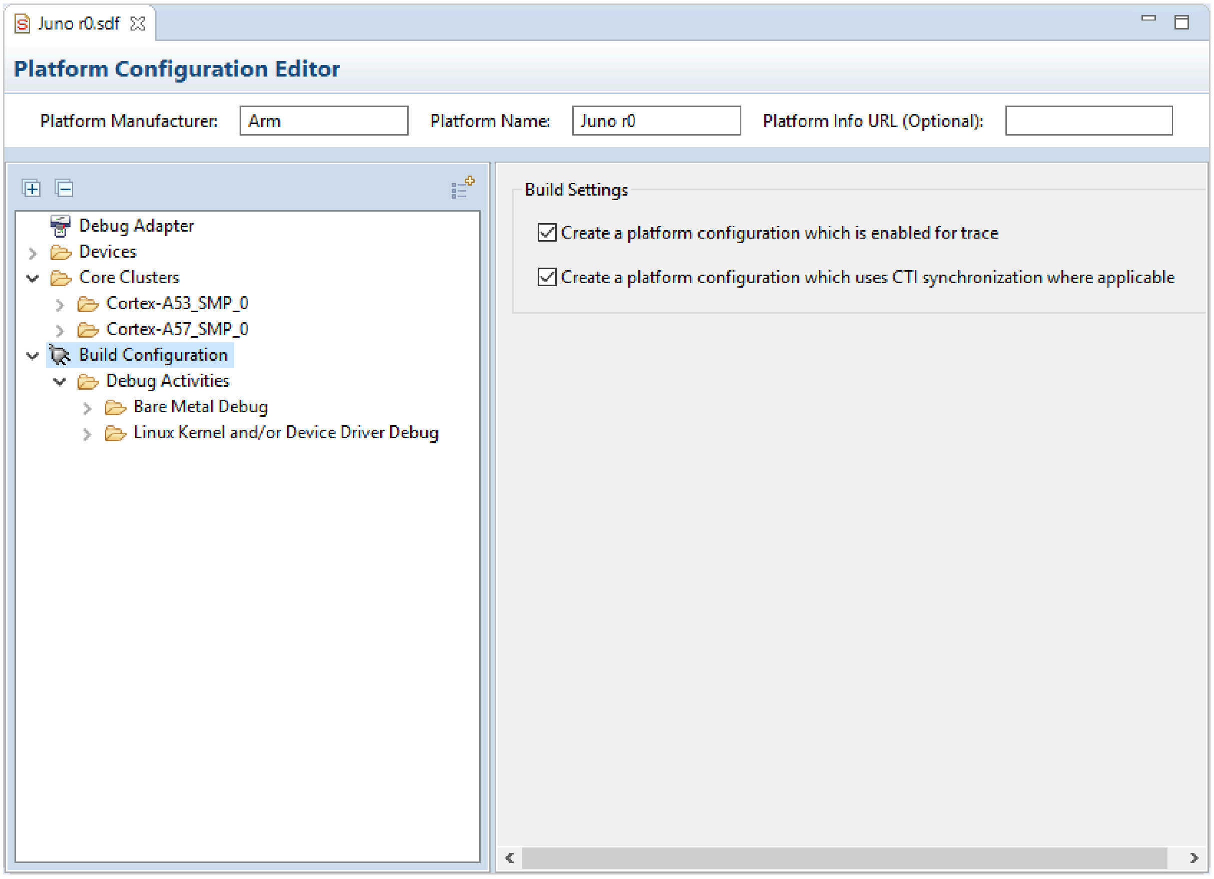
Figure 9-17: User added component connections



Master	Slave	Link Type	Link Details
CSTFunnel_1 (0x820C0000)	CSTFunnel_0 (0x80040000)	ATB	Slave Interface = 0
CSTFunnel_2 (0x830C0000)	CSTFunnel_0 (0x80040000)	ATB	Slave Interface = 1
CSTFunnel_3 (0xE0042000)	CSTFunnel_0 (0x80040000)	ATB	Slave Interface = 3
CSTMC_0 (0x80010000)	CSATBReplicator (0x80120000)	ATB	N/A
Cortex-A53_0 (0x83010000)	CSCTI_4 (0x83020000)	CTITrigger	Trigger DBGRESTART = 1
Cortex-A53_0 (0x83010000)	CSETM_2 (0x83040000)	CoreTrace	N/A
Cortex-A53_1 (0x83110000)	CSCTI_5 (0x83120000)	CTITrigger	Trigger DBGRESTART = 1
Cortex-A53_1 (0x83110000)	CSETM_3 (0x83140000)	CoreTrace	N/A
Cortex-A53_2 (0x83210000)	CSCTI_6 (0x83220000)	CTITrigger	Trigger DBGRESTART = 1
Cortex-A53_2 (0x83210000)	CSETM_4 (0x83240000)	CoreTrace	N/A
Cortex-A53_3 (0x83310000)	CSCTI_7 (0x83320000)	CTITrigger	Trigger DBGRESTART = 1
Cortex-A53_3 (0x83310000)	CSETM_5 (0x83340000)	CoreTrace	N/A
Cortex-A57_0 (0x82010000)	CSETM_0 (0x82040000)	CoreTrace	N/A
Cortex-A57_0 (0x82010000)	CSCTI_2 (0x82020000)	CTITrigger	Trigger DBGRESTART = 1
Cortex-A57_1 (0x82110000)	CSETM_1 (0x82140000)	CoreTrace	N/A
Cortex-A57_1 (0x82110000)	CSCTI_3 (0x82120000)	CTITrigger	Trigger DBGRESTART = 1
Cortex-M3 (0xE000E000)	CSCTI_8 (0xE0044000)	CTITrigger	Trigger DBGRESTART = 7
Cortex-M3 (0xE000E000)	CSETM_6 (0xE0041000)	CoreTrace	N/A

5. Customize the build. By default, Cross Trigger Interface (CTI) synchronization and trace are enabled. When CTI sync or trace fails or causes problems, we recommend disabling these options. To build the platform configuration without these options, click **Build Configuration** and under **Build Settings** deselect the options. Save your configuration after making the changes.

Figure 9-18: Disable Trace or CTIs



Platform Manufacturer: Arm
Platform Name: Juno r0
Platform Info URL (Optional):

Build Settings

☑ Create a platform configuration which is enabled for trace

☑ Create a platform configuration which uses CTI synchronization where applicable

---

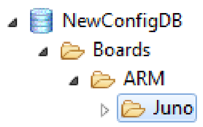
Tip

When starting from scratch with a complex system, to test the connection with your target and debug your cores, we recommend disabling trace in the configuration.

---

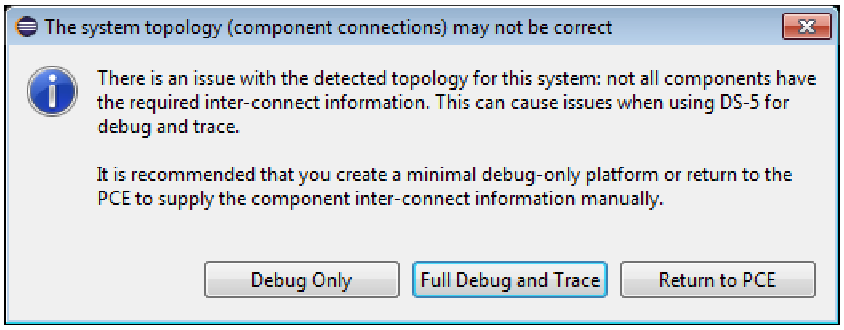
6. Build the platform configuration:
   a) Right-click the project in the **Project Explorer** view and select **Build Platform**.

   Figure 9-19: Project Explorer

   b) **The system topology (component connections) may not be correct** dialog box appears. To regenerate the debug configuration files with the added CoreSight™ trace components, select **Full Debug and Trace**.

Figure 9-20: Full debug and trace



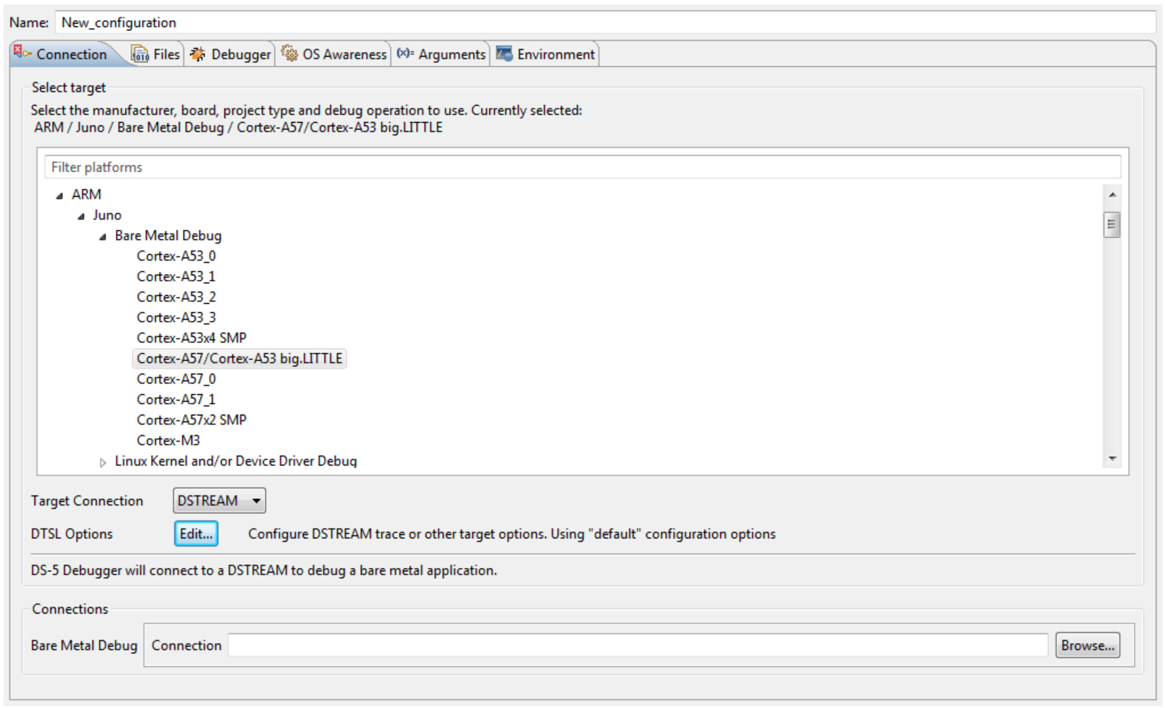
The system topology (component connections) may not be correct

There is an issue with the detected topology for this system: not all components have the required inter-connect information. This can cause issues when using DS-5 for debug and trace.

It is recommended that you create a minimal debug-only platform or return to the PCE to supply the component inter-connect information manually.

7. Select your platform and debug activity:
   a) Open the **Debug Configurations** view. Right-click in the **Project Explorer** view and select **Debug As** > **Debug Configurations...**.

   Figure 9-21: Debug Activities



Name: New_configuration

Select target

Select the manufacturer, board, project type and debug operation to use. Currently selected:
ARM / Juno / Bare Metal Debug / Cortex-A57/Cortex-A53 big.LITTLE

Filter platforms

- ARM
  - Juno
    - Bare Metal Debug
      - Cortex-A53_0
      - Cortex-A53_1
      - Cortex-A53_2
      - Cortex-A53_3
      - Cortex-A53x4 SMP
      - Cortex-A57/Cortex-A53 big.LITTLE
      - Cortex-A57_0
      - Cortex-A57_1
      - Cortex-A57x2 SMP
      - Cortex-M3
    - Linux Kernel and/or Device Driver Debug

Target Connection DSTREAM

DTSL Options Edit... Configure DSTREAM trace or other target options. Using "default" configuration options

DS-5 Debugger will connect to a DSTREAM to debug a bare metal application.

Connections

Bare Metal Debug Connection Browse...

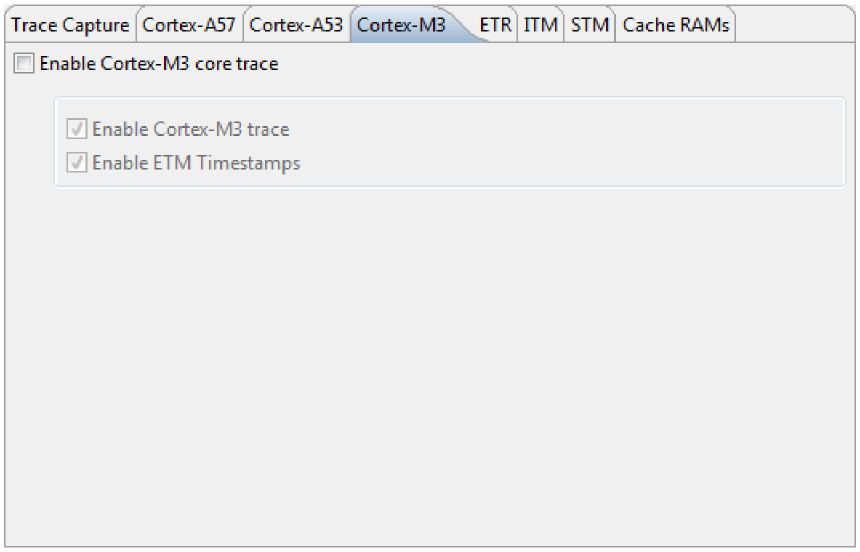
   b) Select your platform and debug activity from the **Connection** tab in the **Debug Configurations** dialog box.

Note

To see your new platform in the **Debug Configurations** list, you must specify your configuration database in **Window** > **Preferences**. Expand **Arm DS* and select **Configuration Database**.

8. Check your new device is trace capable for the <master-device> processor. From the **Connection** tab in the **Debug Configurations** dialog box, click **Edit** on **DTSL Options**.

Figure 9-22: DTSL Options



☐ Enable Cortex-M3 core trace

✓ Enable Cortex-M3 trace
✓ Enable ETM Timestamps

Related information

Hardware platform bring-up in Development Studio on page 194
Create a platform configuration on page 195
Edit a platform configuration on page 203
Manual platform configuration on page 212
Device hierarchy in the PCE view on page 190

9.2.5 Manual platform configuration

In Development Studio, you can manually configure platforms that are not detected automatically.

Before you begin

You will need the topological information that describes your platform.

About this task

To create a custom platform configuration, Development Studio uses the information that it reads from the platform. However, sometimes it is not possible for Development Studio to read all of the information that it needs from a platform. For more information, see Hardware platform bring-up in Development Studio.

Warning

- Always review the information that has been collected before deciding what further action to take. If Development Studio fails to read information, it might be an indication of a deeper problem. For more information, see Hardware platform bring-up in Development Studio.
- If you autoconfigure your platform, it can cause the target to stop and reset. If you do not want to reset the target, you can manually create the platform configuration using Development Studio.

Procedure

1. To manually configure a platform configuration, create a new configuration database to manually configure, or select an existing configuration database:
    - To create a new configuration database, follow the instructions in Create a platform configuration. In the **New Platform** dialog box, select **Advanced platform detection or manual creation**.

      On completion, the Platform Configuration Editor (PCE) opens.
    - To select an existing configuration database, in the **Project Explorer** double-click on the `*.sdf` file for the platform configuration to edit.

      The Platform Configuration Editor (PCE) opens.
2. Add or edit the existing topology information for your platform. See the topology diagram for your platform. When you know the topology, add the components using these steps:
    a) Create the JTAG scan chain by adding all the devices that are on the scan chain. Drag-and-drop devices to the scan chain from the device hierarchy into the **Devices** folder in the PCE view. The devices must be in the correct order on the JTAG scan chain.

    Note

    - For more information about using the device hierarchy, see Device hierarchy in the PCE view.
    - For more information about adding custom devices, see the Custom devices topic.

    b) Add your CoreSight™ devices or Cortex® processors to the device hierarchy. Before you begin, you must add a CoreSight Debug Access Port (DAP), for example ARMCS-DP. For each DAP, you must add the CoreSight Memory Access Ports (AP) that you need, for example CSMEMAP. You must specify the correct index and type of each AP.
    c) Add the Cortex processors and CoreSight devices to the correct AP on the correct DAP. To do this, drag-and-drop them from the devices panel into the correct AP. As the CoreSight devices are memory-mapped, you can add them in any order. However, you need to ensure that the device type and ROM table base address are correct.
    d) Specify how the devices connect to each other. Edit a platform configuration describes how to do this.

    Warning

    Always review the information that has been collected before deciding what further action to take. If Development Studio fails to read information, it might be an indication of a deeper problem. For example, if Development Studio fails to discover the base addresses because the devices are powered down, it might not be possible to provide debug support. This can also happen for manually configured platforms because powered down devices are not responsive to Arm® Debugger. You might need to perform other operations, such as enabling clocks or powering processor clusters, before debug and trace are possible.
3. Save and build the `*.sdf` file. Click **File** > **Save**.

Results

When you next connect to a target through **File** > **New** > **<|Hardware|Linux Application|Model> Connection**, the manually configured platform is visible as an available target.

Related information

Hardware platform bring-up in Development Studio on page 194
Create a platform configuration on page 195
Device configuration panel on page 214
Edit a platform configuration on page 203

9.2.6 Custom devices

You can add custom devices to the JTAG scan chain in the PCE.

To add a custom device, right-click on the **Devices** folder, and select **Add Custom Device**.

When you add a custom device, you must specify the correct JTAG Instruction Register (IR) length. You can provide any name for the custom device. You cannot debug a custom device. However, if you add the custom device in the correct order with the correct length, you will be able to debug the supported devices in the same scan chain.

Note

You cannot consolidate multiple custom devices on the scan chain. For example, you cannot replace two custom devices with instructions lengths of 4 and 5 bits, by a single custom device of instruction length 9 bits.

Related information

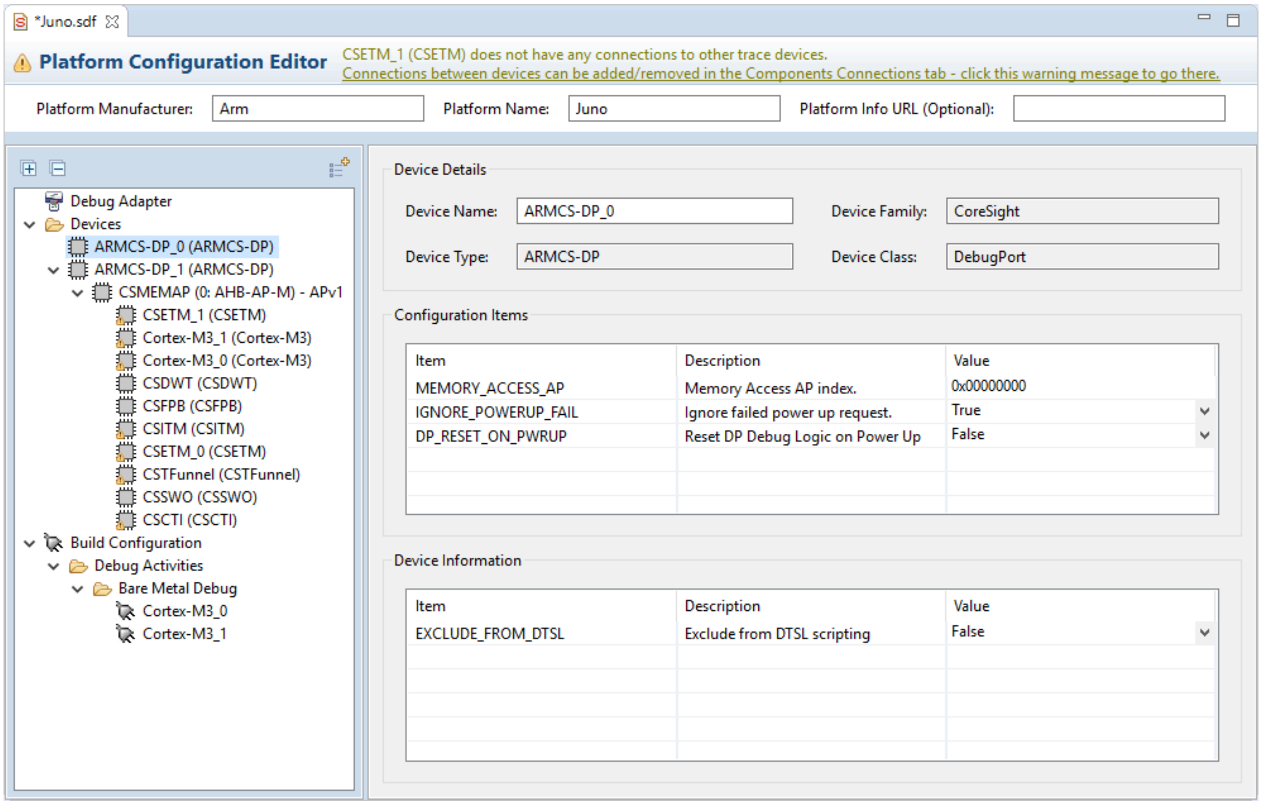
Manual platform configuration on page 212
Edit a platform configuration on page 203

9.2.7 Device configuration panel

The **Device** configuration panel in the PCE shows the configuration information for the devices on your platform.

Edit an existing device within the Platform Configuration Editor (PCE) using the **Device** configuration panel.

Figure 9-23: The Device configuration panel in the PCE.


*Juno.sdf

Platform Configuration Editor

CSETM_1 (CSETM) does not have any connections to other trace devices.
Connections between devices can be added/removed in the Components Connections tab - click this warning message to go there.

Platform Manufacturer: Arm
Platform Name: Juno
Platform Info URL (Optional):

- Debug Adapter
- Devices
  - ARMCS-DP_0 (ARMCS-DP)
  - ARMCS-DP_1 (ARMCS-DP)
    - CSMEMAP (0: AHB-AP-M) - APv1
      - CSETM_1 (CSETM)
      - Cortex-M3_1 (Cortex-M3)
      - Cortex-M3_0 (Cortex-M3)
      - CSDWT (CSDWT)
      - CSFPB (CSFPB)
      - CSITM (CSITM)
      - CSETM_0 (CSETM)
      - CSTFunnel (CSTFunnel)
      - CSSWO (CSSWO)
      - CSCTI (CSCTI)
- Build Configuration
  - Debug Activities
    - Bare Metal Debug
      - Cortex-M3_0
      - Cortex-M3_1

Device Details

Device Name: ARMCS-DP_0
Device Family: CoreSight
Device Type: ARMCS-DP
Device Class: DebugPort

Configuration Items

Item	Description	Value
MEMORY_ACCESS_AP	Memory Access AP index.	0x00000000
IGNORE_POWERUP_FAIL	Ignore failed power up request.	True
DP_RESET_ON_PWRUP	Reset DP Debug Logic on Power Up	False

Device Information

Item	Description	Value
EXCLUDE_FROM_DTSL	Exclude from DTSL scripting	False

Access

You can access the panel using one of the following methods:

- In the **Project Explorer**, right-click on an `*.sdf` file, and select **Open With** > **Platform Configuration Editor**. Select a device.
- Double-click an `*.sdf` file (where the `*.sdf` association has not been overridden), then select a device.
- **File** > **New** > **Other...** > **Configuration Database** > **Platform Configuration**. At the end of the wizard, you have the option to open the PCE to check or modify the platform. In the PCE, select a device.
- The target connection wizard provides an option to enter the PCE: **File** > **New** > **Hardware Connection--> Add a new platform**. In the PCE, select a device.

Contents

The **Device** configuration panel contains:

Table 9-3: Device configuration panel contents

Field	Description
Device Details	Shows the general information about the device. Fields include: • Device Name • Device Type • Device Family • Device Class
Configuration Items	Shows the configuration items for the device. These are used by Arm® Debugger to configure the device when connecting to the target. Changing these configuration items affects the behavior of the target.
Device Information	Shows information about the device. The Arm Debugger uses this information to generate the platform configuration. If you change the device information, the platform configuration generated by Arm Debugger changes, which might result in different debug and trace options being available.

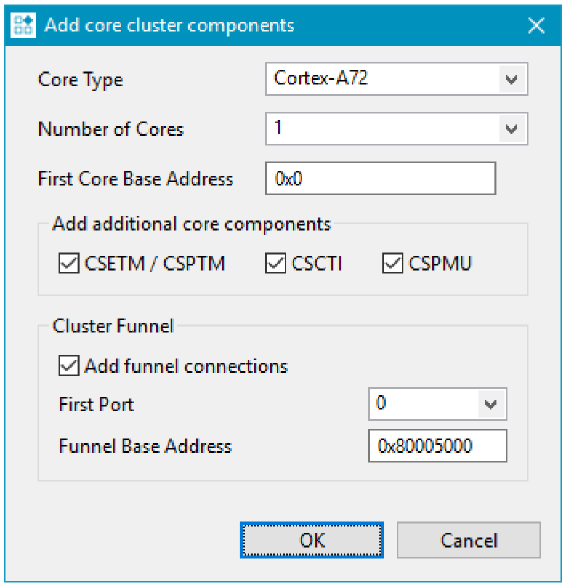
Related information

Add core cluster components dialog box on page 216
Edit a platform configuration on page 203
Manual platform configuration on page 212
Platform Configuration Editor (PCE) on page 186

9.2.8 Add core cluster components dialog box

Use the **Add core cluster components** dialog box to add cores to your platform configuration, including the topology links.

Figure 9-24: Add core cluster components dialog box.


Add core cluster components

Core Type: Cortex-A72
Number of Cores: 1
First Core Base Address: 0x0

Add additional core components
☑ CSETM / CSPTM ☑ CSCTI ☑ CSPMU

Cluster Funnel
☑ Add funnel connections
First Port: 0
Funnel Base Address: 0x80005000

OK Cancel

Access

Right-click a MEM-AP device (Memory Access Port device) in the Device hierarchy in the PCE view and select **Add core cluster components**.

Contents

The **Add core cluster components** dialog box contains:

Table 9-4: Add core cluster components dialog box contents

Field	Description
Core Type	Select the core type.
Number of Cores	Select the number of cores to add.
First Core Base Address	Enter the base address of your first core.
Add additional core components	Select the core components to add. Choose from: • CSETM/CSPTM • CSCTI • CSPMU
Cluster Funnel	Configure a cluster funnel, using: • **Add funnel connections** - Select to add funnel connections. • **First Port** - Select the number of the first port of the funnel to which the ETM output is routed. Choose a port number from 0 to 7. • **Funnel Base Address** - Enter the base address of the funnel.

Related information

Device hierarchy in the PCE view on page 190
Platform Configuration Editor (PCE) on page 186

9.3 Model targets

This section describes how to configure model platforms in Development Studio.

9.3.1 Model platform bring-up in Development Studio

To start a debug connection to a model, Arm® Debugger needs a model platform configuration. A model platform configuration consists of:

- A `dtsl_config_script.py` file.
- A `project_types.xml` file.
- A `cadi_config.xml` file (if using Component Architecture Debug Interface (CADI) as the connection interface) or a `iris_config.xml` file (if using Iris as the connection interface).

Note

You do not need an `*.sdf` file.

When Development Studio connects to a running model, the Model Configuration Editor connects to, and interrogates, the model for information about its devices. The information detected is used to create an `*.mdf` file (Model Description File). The `*.mdf` file is used to generate the model platform configuration, but it is not required for connection.

To enable a simple and efficient model platform bring-up process, Arm Debugger automatically detects most of the model configuration information. However, Arm Debugger might not recognize the core type of new models with custom or pre-released cores. To resolve this, open the `*.mdf` file in Model Configuration Editor and change the unrecognized core types to core types that are supported by Arm Debugger.

For more information on creating model configurations, follow the instructions in Create a new model configuration. For more information about the Model Configuration Editor, see Model Configuration Editor.

Related information

Create a new model configuration on page 220
Model Configuration Editor on page 227
Specifying a custom configuration database using the command-line on page 168
Configuring a connection to an external Fixed Virtual Platform (FVP) for bare-metal application debug
Configuring a connection to a Linux application using gdbserver
Configuring a connection to a Linux kernel
Configuring a connection to a bare-metal hardware target
Overview: Debug connections in Arm Debugger
Component Architecture Debug Interface Developer Guide
Iris Developer Guide

9.3.2 Set up environment variables for models not provided with Arm Development Studio

Arm® Debugger provides built-in support for connecting to a large range of Arm Fast Models products. To use any other simulation model with Arm Debugger, you must set up your host operating system environment variables so that your models are available to Development Studio.

Before you begin

- You might require local administrator access on the host operating system to make any changes to the system environment variables.

Procedure

1. Add the `install_directory/bin` directory to your `PATH` environment variable:
   - For Windows, enter `set PATH=<your model path>\bin;%PATH%`.
   - For Linux, enter `export PATH=<your model path>/bin:$PATH`.
2. Restart Development Studio.
3. Ensure that the modified path is available for future sessions:
   - **For Windows:**
     a. Right-click **My Computer** > **Properties** > **Advanced system settings** > **Environment Variables**.
     b. Under **User Variables**, either create a `PATH` variable with the value `<your model path>\bin`, or append `;<your model path>\bin` to any existing `PATH` variable.
   - For Linux, set up the `PATH` in the appropriate shell configuration file. For example, in `.bashrc`, add the line `export PATH=<your model path>/bin:$PATH`.

Results

The models are now available to be used with Development Studio.

Related information

Create a new model configuration on page 220
Launch a Fast Model for use with Arm Development Studio on page 219

9.3.3 Launch a Fast Model for use with Arm Development Studio

If you want to connect to an already running model using Arm® Development Studio, you need to first launch the model with the appropriate model connection interface. You can launch a Fast Models as a library file or as an executable.

Procedure

1. Launch your model and start the model connection interface server:

Table 9-5: Launch your model with the appropriate model connection interface

To launch models with Component Architecture Debug Interface (CADI) as the model connection interface server:	To launch models with Iris as the model connection interface server:
• If your model is a library file: ◦ On Windows, select **Start** > **All Programs** > **Arm Development Studio** > **Arm Development Studio Command Prompt** and enter, either: ▪ `model_shell -m <your model path and name> -S` ▪ `<your model path and executable name> -S` ◦ On Linux, open a new terminal and run: `install_directory /bin/model_shell -m <your model path and name> -S`. • If your model is an executable file, at the command prompt, enter `<your model path and name> -S`.	• If your model is a library file: ◦ On Windows, select **Start** > **All Programs** > **Arm Development Studio** > **Arm Development Studio Command Prompt** and enter, either: ▪ `model_shell -m <your model path and name> -I` ▪ `<your model path and executable name> -I` ◦ On Linux, open a new terminal and run: `install_directory /bin/model_shell -m <your model path and name> -I`. • If your model is an executable file, at the command prompt, enter `<your model path and name> -I`.

2. Connect to the running model using the Development Studio connection options.

Note

- For more information about the options available with the `model_shell` utility in Development Studio, enter `model_shell --help` at the Development Studio command prompt.
- For more information about the switches available for either CADI or the Iris model connection interfaces, see the Fast Models documentation.

Related information

Create a new model configuration on page 220

Set up environment variables for models not provided with Arm Development Studio on page 218

9.3.4 Create a new model configuration

Use the **Model Configuration** wizard in Development Studio to create debug configurations for new models.

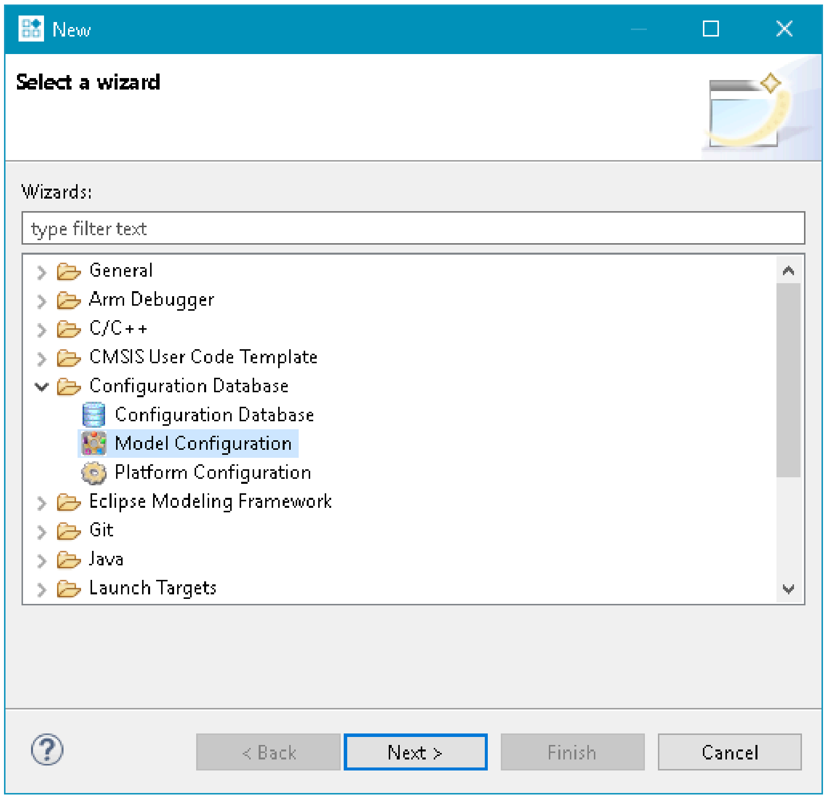
Before you begin

- If you are importing an existing model configuration file (`*.mdf`), ensure you have access to this file.
- Some of the options below require you to launch your model before connecting to it. Before using these options, ensure you have launched your model with the appropriate model interface switches before attempting to connect to it.

Procedure

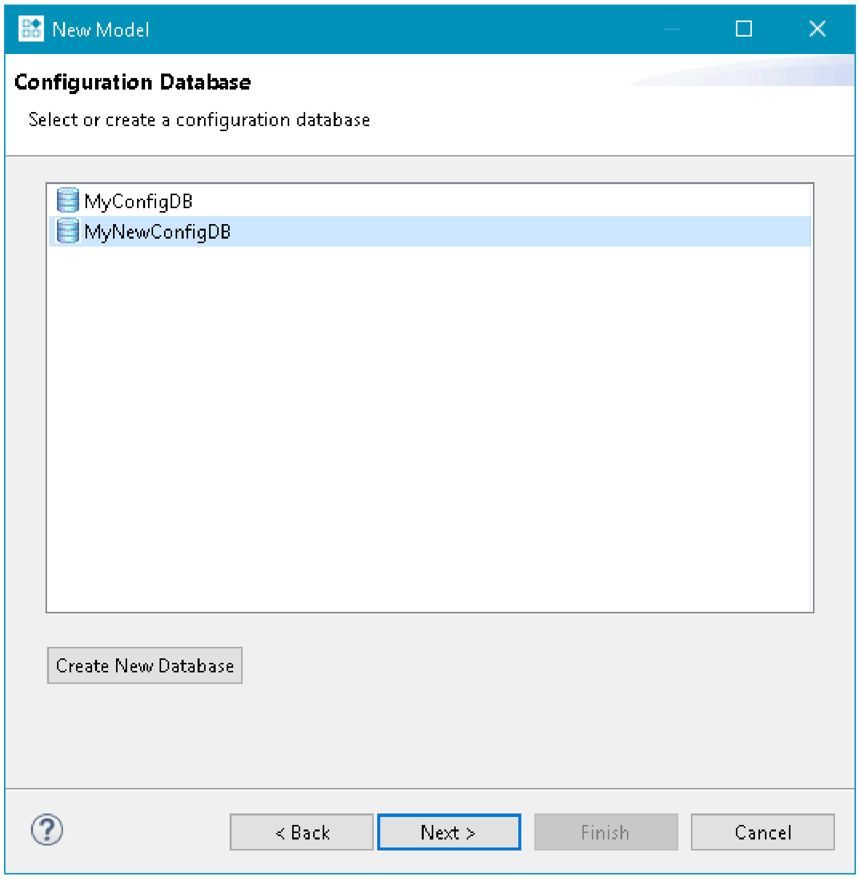
1. Open the **Model Configuration** wizard. From the main menu, select **File** > **New** > **Other** > **Model Configuration** and click **Next**.

Figure 9-25: Select Model Configuration wizard



2. Either:
    - Select the configuration database where you want to add your model.

Figure 9-26: Select a database for your new model configuration.



- Click **Create New Database** to create a new configuration database.

  If you create a new database, enter a name at the prompt and click **OK** to save it.

3. Click **Next**. The **Select Method for Connecting to Model** dialog box opens.
4. Select a model interface for connecting to your model. You have two interface options - Component Architecture Debug Interface (CADI) or Iris.

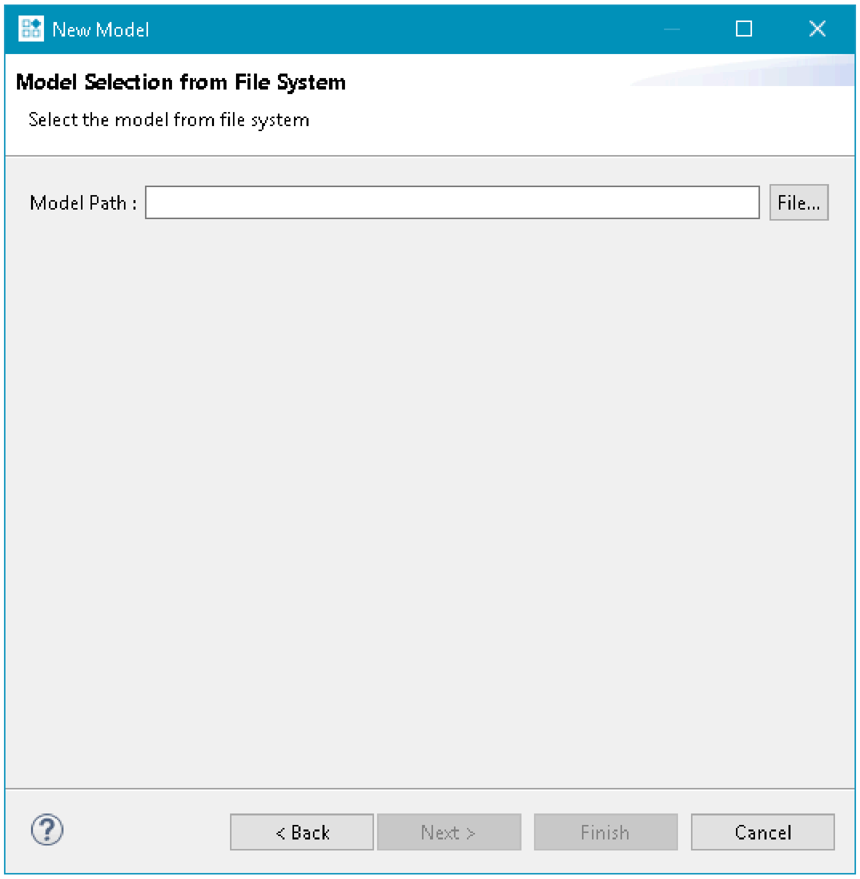
   CADI model interface:

   - To launch and connect to a specific model from your local file system using CADI:

     a. Select the **Launch and connect to a specific model** option and click **Next**.

     b. In the **Model Selection from File System** dialog box, click **File** to browse for a model and select it.

Figure 9-27: Select model from file system


New Model
Model Selection from File System
Select the model from file system
Model Path :
File...
< Back
Next >
Finish
Cancel

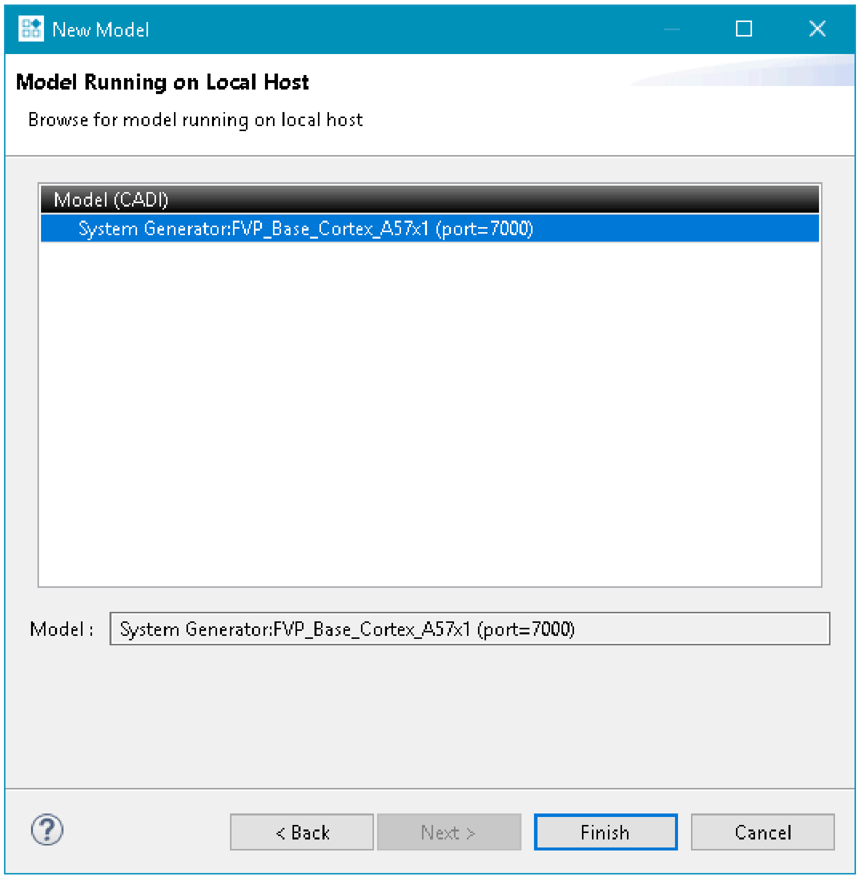
c. Click **Open**, and then click **Finish**.

- To connect to a model running on the local host:

  a. Select the **Browse for model running on local host** option and click **Next**.

  b. Select the model you require from the listed models.

Figure 9-28: Browse for model running on local host


New Model
Model Running on Local Host
Browse for model running on local host
Model (CADI)
System Generator:FVP_Base_Cortex_A57x1 (port=7000)
Model :
System Generator:FVP_Base_Cortex_A57x1 (port=7000)
< Back
Next >
Finish
Cancel

c. Click **Finish** and connect to the model.

Iris model interface:

- To launch and connect to a specific model from your local file system using Iris:
  a. Select the **Launch and connect to a specific model** option and click **Next**.
  b. In the **Model Selection from File System** dialog box, click **File** to browse for a model and select it.
  c. Click **Open**, and then click **Finish**.
- To connect to a model running on the local host:

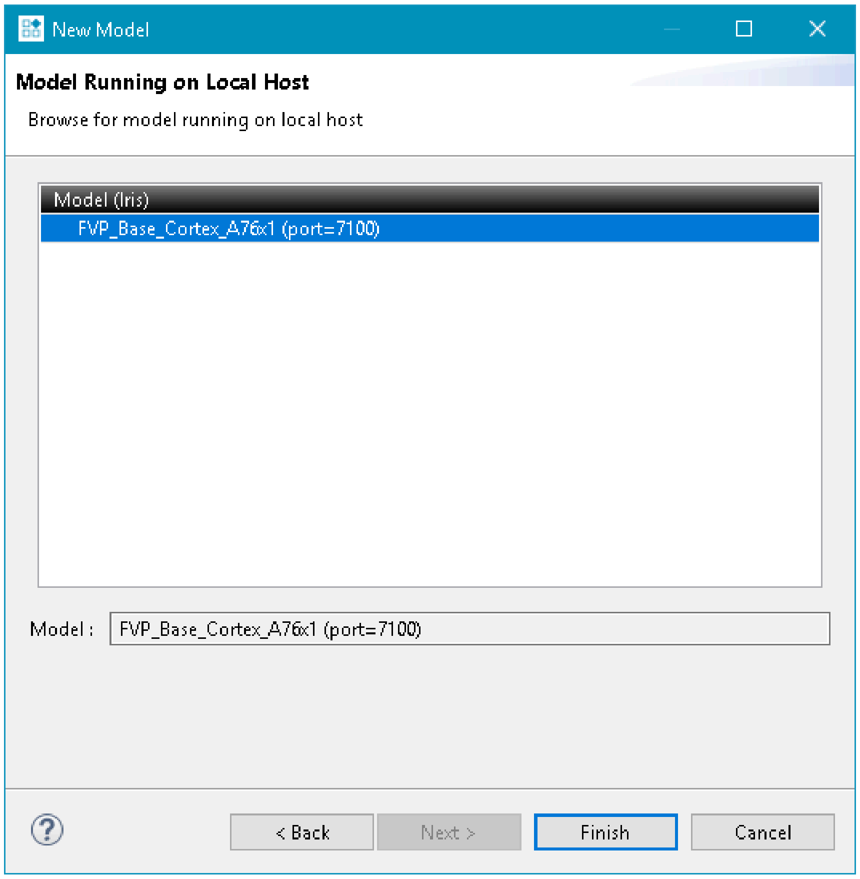
Note

To connect to models running on the local host, you must launch the model with the `--iris-server` switch before connecting to it.

a. Select the **Browse for model running on local host** option and click **Next**.

b. Select the model you require from the listed models.

Figure 9-29: Browse for model running on local host


New Model
Model Running on Local Host
Browse for model running on local host
Model (Iris)
FVP_Base_Cortex_A76x1 (port=7100)
Model : FVP_Base_Cortex_A76x1 (port=7100)
< Back
Next >
Finish
Cancel

c. Click **Finish** and connect to the model.

- To connect to a model using its address and port number, running either on the local or a remote host:

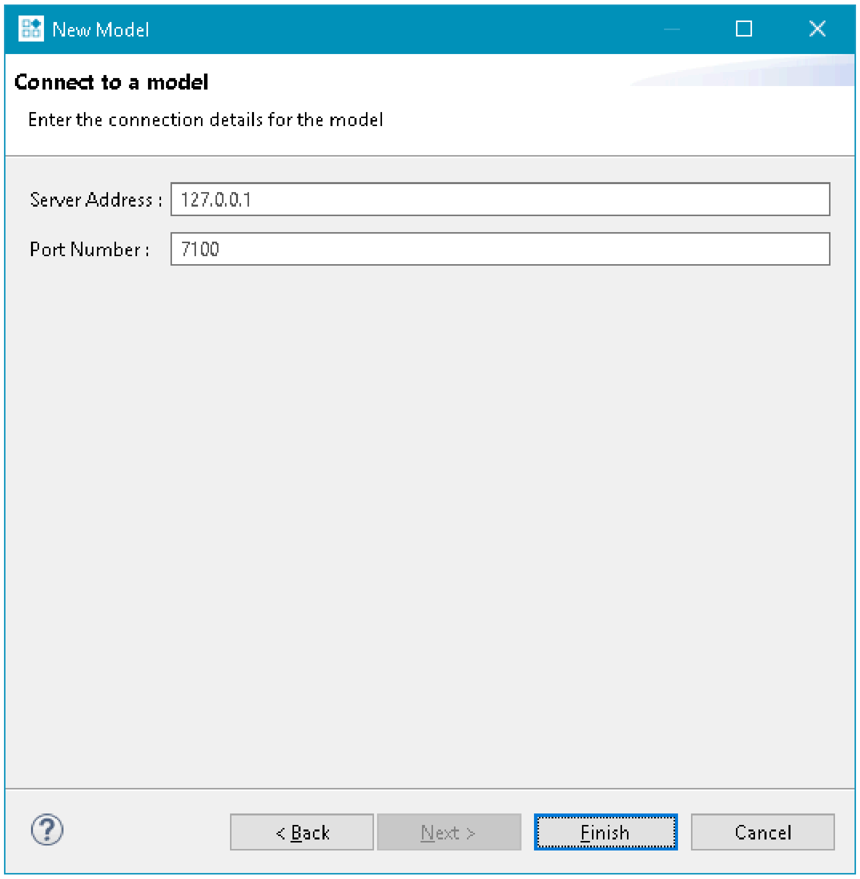
Note

To connect to models running on the local host, you must first launch the model with the `--iris-server` switch before connecting to it. To connect to models running on a remote host, you must first launch the model with the `--iris-server --iris-allow-remote` switches before connecting to it remotely.

a. Select the **Connect to model running on either local or remote host** option and click **Next**.

b. Enter the connection address and port number of the model.

Figure 9-30: Connect to model running on either local or remote host



Server Address : 127.0.0.1
Port Number : 7100

c. Click **Finish**.

The selected model is imported and the `*.mdf` created. The **Model Configuration Editor** opens and loads the imported model file. You can view the configuration database and model in the **Project Explorer**.

5. (Optional) Rename the **Manufacturer Name** and **Platform Name**, and if required, use the Model Configuration Editor to complete the model configuration.

Note

If you do not enter a **Manufacturer Name**, the platform is listed under **Imported** in the **Debug Configurations** dialog box.

Next steps

Make any changes to the model in the **Model Configuration Editor**. To save the changes to the model, click **Save**.

To import and rebuild the Development Studio configuration database, click **Import**.

Click **Debug** to open the **Debug Configurations** dialog box to create, manage, and run configurations for this target.

Related information

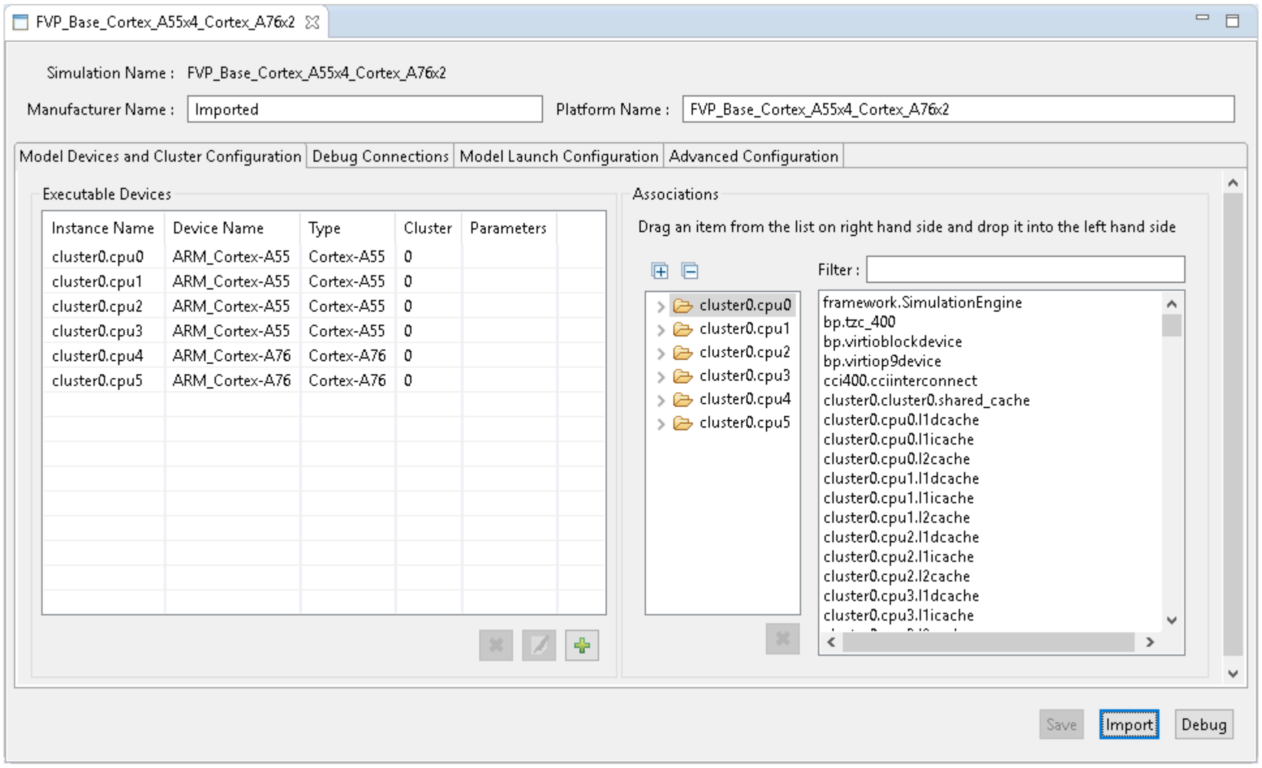
Model Configuration Editor on page 227
Model platform bring-up in Development Studio on page 217
Launch a Fast Model for use with Arm Development Studio on page 219
FVP command-line options

9.3.5 Model Configuration Editor

Use the **Model Configuration Editor** to create or modify configurations and connections for model target platforms.

In the **Model Configuration Editor**, you can add or remove executable devices to configure the model for debug. This figure shows the **Model Devices and Cluster Configuration** of an example model platform.

Figure 9-31: Model Devices and Cluster Configuration tab


FVP_Base_Cortex_A55x4_Cortex_A76x2
Model Devices and Cluster Configuration
Debug Connections
Model Launch Configuration
Advanced Configuration
Save
Import
Debug

Simulation Name : FVP_Base_Cortex_A55x4_Cortex_A76x2
Manufacturer Name : Imported
Platform Name : FVP_Base_Cortex_A55x4_Cortex_A76x2

Executable Devices

Instance Name	Device Name	Type	Cluster	Parameters
cluster0.cpu0	ARM_Cortex-A55	Cortex-A55	0	
cluster0.cpu1	ARM_Cortex-A55	Cortex-A55	0	
cluster0.cpu2	ARM_Cortex-A55	Cortex-A55	0	
cluster0.cpu3	ARM_Cortex-A55	Cortex-A55	0	
cluster0.cpu4	ARM_Cortex-A76	Cortex-A76	0	
cluster0.cpu5	ARM_Cortex-A76	Cortex-A76	0	

Associations

Drag an item from the list on right hand side and drop it into the left hand side

Filter :

```
cluster0.cpu0
cluster0.cpu1
cluster0.cpu2
cluster0.cpu3
cluster0.cpu4
cluster0.cpu5
```

```
framework.SimulationEngine
bp.tzc_400
bp.virtioblockdevice
bp.virtiop9device
cci400.cciinterconnect
cluster0.cluster0.shared_cache
cluster0.cpu0.l1dcache
cluster0.cpu0.l1icache
cluster0.cpu0.l2cache
cluster0.cpu1.l1dcache
cluster0.cpu1.l1icache
cluster0.cpu1.l2cache
cluster0.cpu2.l1dcache
cluster0.cpu2.l1icache
cluster0.cpu2.l2cache
cluster0.cpu3.l1dcache
cluster0.cpu3.l1icache
```

Access the Model Configuration Editor

- Double-click an `*.mdf` file.

- In the **Project Explorer**, right-click on a `*.mdf` file, and select **Open with** > **Platform Configuration Editor**.
- In the **Project Explorer**, right-click, and select **File** > **New** > **Model Configuration**. After connecting to a model, the Model Configuration Editor opens.
- Select **File** > **New** > **Other...** > **Configuration Database** > **Model Configuration**. After selecting a configuration database, and connecting to a model, you have the option to open the Model Configuration Editor to check or modify the model.
- The target connection wizard provides the option to enter the Model Configuration Editor at the final stages of new connection creation: **File** > **New** > **Model Connection**. At the target selection step, click **Add a new model...**.

Contents

The **Model Configuration Editor** contains:

Table 9-6: Model Configuration Editor contents

Field	Description
Manufacturer Name	Model platform manufacturer.
Platform Name	Model platform name.
Model Devices and Cluster Configuration tab	View and configure the devices in the model. • The **Executable Devices** section lists the cores available within the model. Add, remove, or edit the available cores. • The **Associations** section lists the non-executable devices within the model. Expand the associations to see the mapping of the non-executable devices. Delete items from the associations view or add items from the list of available non-executable devices. **Note:** There are two important conventions for an **Instance Name**: • Each name must be unique. • For multi-cluster models, each name must be in the clusterX.cpuY format.
Debug Connections tab	View and configure the debug connections. Drag cores or clusters from the **Cores and Clusters** section and drop them into the node connection under **Debug Activities**. To enable Linux application debug, select the **Enable Linux Application Debug** option.

Field	Description
Model Launch Configuration tab	View and configure the launch settings for a specific model: • To enable options and provide the path of the model you want to launch, select **Launch and connect to a specific model**. • **Model Interface** shows the interface used for creating the model configuration. • To use the default launcher script, select **Use Default Launcher Script**. To provide an alternative launch script, deselect the option, and provide a script path in **Launch Script**. • To provide model launch parameters, specify them in the **Model launch parameters** panel. Click **Parameters** to display available model launch parameters. Select the parameters to use, and click **OK** to apply them. • To provide model run-time parameters, specify them in the **Model run-time parameters** panel.
Advanced Configuration tab	View and select configure advanced configuration options: • To create a model log, select the **Enable Model Log** option. • To specify a RDDI log file, select the **Enable RDDI Log** option, and specify the file path. **Note:** To enable RDDI logging, you must specify a RDDI Log file to use.
Save	Saves the model configuration.
Import	Imports configuration database files and adds the project to Development Studio preferences.
Debug	Launches the Development Studio **Debug Configurations** dialog box.

Usage

Configure your model and connections in the Model Configuration Editor.

To save the changes to the configuration, click **Save**. To import the configuration database files and add the projects to the Development Studio preferences, click **Import**.

Warning

Changes to an `*.mdf` file must be imported into the Development Studio preferences to take effect.

To launch the Development Studio **Debug Configurations** dialog box, click **Debug**.

Related information

Create a new model configuration on page 220
Launch a Fast Model for use with Arm Development Studio on page 219
Set up environment variables for models not provided with Arm Development Studio on page 218

9.4 Configuration database

This section describes how to add, edit, and extend configuration databases in Development Studio.

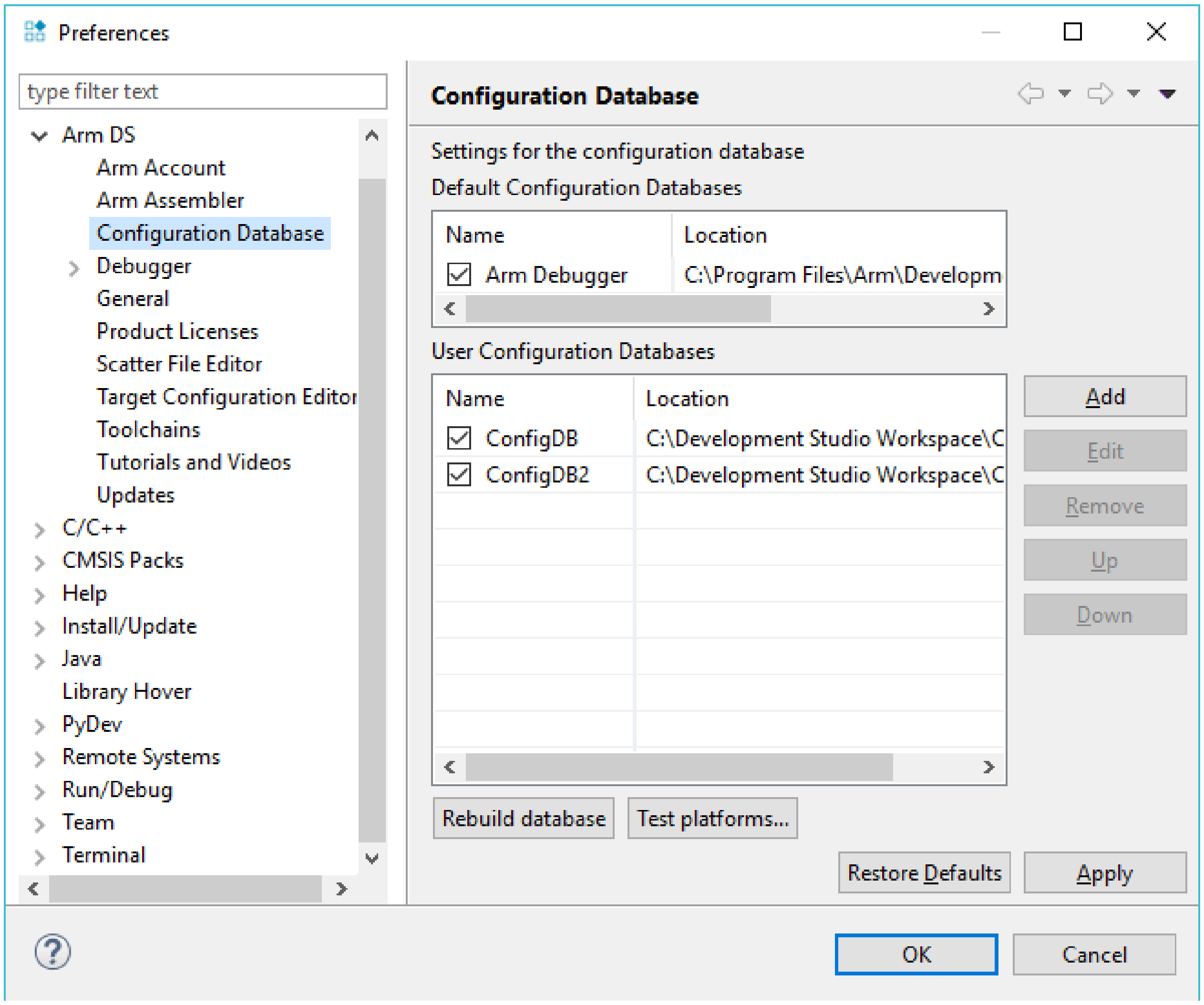
9.4.1 Configuration Database panel

Use the **Configuration Database** panel to manage the Development Studio configuration database settings.

The Development Studio configuration database is made up of **Default Configuration Databases** and **User Configuration Databases**.

In the **Configuration Database** panel you can add, remove, edit, and organize the default and user-provided configuration databases. You can also rebuild the Development Studio configuration database and test platforms.

Figure 9-32: Configuration Database panel



Configuration Database

Settings for the configuration database

Default Configuration Databases

Name	Location
☑ Arm Debugger	C:\Program Files\Arm\Developm

User Configuration Databases

Name	Location
☑ ConfigDB	C:\Development Studio Workspace\C
☑ ConfigDB2	C:\Development Studio Workspace\C

Access

Select **Window** > **Preferences**. In the **Preferences** dialog box, expand **Arm DS** and select **Configuration Database**.

Contents

The **Configuration Database** panel contains:

Table 9-7: Configuration Database panel contents

Field	Description
Default Configuration Databases	Displays the default Development Studio configuration databases. **Note:** Arm recommends that you do not disable these.
User Configuration Databases	Displays the user-provided configuration databases.

Field	Description
Add	Opens a dialog box to select a configuration database to add.
Edit	Opens a dialog box to modify the name and location of the selected configuration database.
Remove	Removes the selected configuration database.
Up	Moves the selected configuration database up the list.
Down	Moves the selected configuration database down the list.
Rebuild database	Rebuilds the Development Studio configuration database.
Test platforms...	Enables you to test platforms and report any errors found. This is useful when trying to determine why a platform configuration is not loading correctly.
Restore Defaults	Removes all the configuration databases that do not belong to the Development Studio default system.
Apply	Saves the current configuration database settings.

Usage

Development Studio configuration database reads and processes the default and user configuration databases sequentially from top to bottom in the list. The information read in each configuration database appends, or overwrites, the information read in the previous databases positioned above it in the list. This means the information in the configuration database that is positioned at the bottom of the list has the highest priority. The information in the configuration database that is positioned at the top of the list has the lowest priority. For example, if you produced a modified core definition with different registers, you would add it to the database at the bottom of the list so that the Development Studio configuration database reads it last. Development Studio then uses this information instead of the core definitions in the higher-positioned or default databases.

Ensure that you rebuild the Development Studio configuration database after you add, remove, or edit any configuration databases. To rebuild the Development Studio configuration database, click **Rebuild database**.

Related information

Add a configuration database on page 232
Add a Platform Configuration to a Configuration Database on page 234
Create a platform configuration on page 195
How do I add a custom components.xml file to an Arm Development Studio Configuration Database

9.4.2 Add a configuration database

You can add configuration databases from other sources into your installation of Arm® Development Studio.

Before you begin

You must have an existing configuration database.

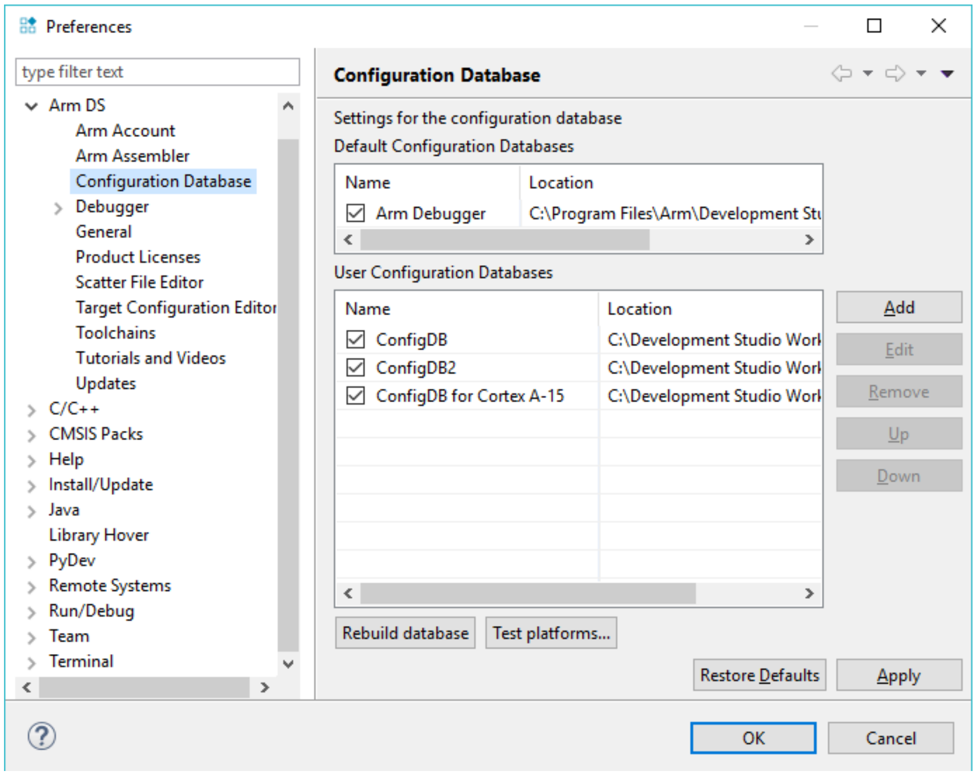
Procedure

1. Open the **Preferences** dialog box. From the main menu, select **Window** > **Preferences**.
2. Expand **Arm DS** and select **Configuration Database**.
3. Add your new configuration database and configure the ordering:
   a) Click **Add**. The **Add configuration database location** dialog box opens.
   b) Enter a **Name** for the configuration database, for example, **MyConfigDB**.
   c) Click **Browse**. Find and select the database, then click **OK**.
   d) Click **OK** to close the **Add configuration database location** dialog box.
   The new configuration database is listed as an option under **User Configuration Databases**.
   e) (Optional) If required, change the order of the user configuration databases. Select the new database and click **Up** or **Down** as required.

Note

Development Studio processes the user configuration databases from top to bottom, with the information in the lower databases replacing information in the higher databases. For example, if you want to produce a modified Cortex®-A15 processor definition with different registers, add those changes to a new configuration database lower in the list of user databases.

Figure 9-33: Reorder the configuration databases



Configuration Database

Settings for the configuration database

Default Configuration Databases

Name	Location
☑ Arm Debugger	C:\Program Files\Arm\Development Stu

User Configuration Databases

Name	Location
☑ ConfigDB	C:\Development Studio Worl
☑ ConfigDB2	C:\Development Studio Worl
☑ ConfigDB for Cortex A-15	C:\Development Studio Worl

Note

Development Studio provides built-in databases containing a default set of target configurations. You can enable or disable these databases, but you cannot delete them.

---

4. Click **Rebuild database** to rebuild your Development Studio configuration database.
5. Click **OK** to close the dialog box and save the settings.

Results

You can view your new configuration database in the **Connection** tab of the **Debug Configuration** dialog box, under the name specified in the procedure.

Related information

Create a platform configuration on page 195
Create a new model configuration on page 220
Add a Platform Configuration to a Configuration Database on page 234
Configuration Database panel on page 230

9.4.3 Add a Platform Configuration to a Configuration Database

You can add a new platform configuration to your Development Studio configuration database by adding the platform to a user configuration database, then rebuilding the Development Studio configuration database.

Before you begin

- You need the platform configuration files for the new platform.

Note

Your platform configuration must include the `*.sdf` file, but might also include files such as the `dtsl_config_script.py` and the `project_types.xml` files.

- You need to have an existing user configuration database to update to add the platform configuration.

About this task

Note

If you create a platform configuration using the PCE, Development Studio automatically saves it to the configuration database you chose or create within the **New Platform** wizard.

Procedure

1. Navigate to the user configuration database to update in your file system.

2. Add the new platform configuration directory (containing the `*.sdf` file) to the **Boards** directory.
3. Open the **Configuration Database** dialog box in Development Studio. Select **Window** > **Preferences**. In the **Preferences** dialog box which opens, select **Arm DS** > **Configuration Database**.
4. Rebuild the Development Studio configuration database. Ensure the updated user configuration database is selected, and click **Rebuild database**.

Note

If the updated user configuration database is not listed, follow the instructions in Add a configuration database to add it as a new configuration database.

---

5. Click **OK**.

Results

You can view the new platform in the **Connection** tab of the **Debug Configuration** dialog box. The new platform is listed under `<Platform Manufacturer>/<Platform Name>` as specified in the `*.sdf` file.

Related information

Add a configuration database on page 232

Configuration Database panel on page 230

How do I add a custom components.xml file to an Arm Development Studio Configuration Database

9.4.4 Add Arm debug hardware support to an existing platform configuration

Support for the latest debug hardware systems from Arm is available in the latest releases of Arm® Development Studio. Use the Platform Configuration Editor (PCE) in Arm Development Studio to add the latest debug hardware system support to your existing platform configurations.

Before you begin

- Install the latest release of Arm Development Studio which contains support for the latest debug hardware from Arm.
- You require an existing platform configuration.

Note

- Your platform configuration must include the `*.sdf` file, but might also include files such as the `dtsl_config_script.py` and the `project_types.xml` files.

About this task

Warning

Generating a new platform configuration overwrites any DTSL scripts and other platform files that might exist for your current configuration. Backup your configuration if you have applied manual changes to your configuration.

Procedure

1. In the *Project Explorer*, right-click the `*.sdf` file, and select **Open With** > **Platform Configuration Editor**.
2. Click *Debug Activities* to view hardware debug units supported by your version of Arm Development Studio.
3. Select the debug units that you require support for in your platform configuration.
4. Build the platform configuration. Right-click the project in the **Project Explorer** view and select **Build Platform**.

   If your platform contains incomplete or invalid topology, a **This platform contains warnings** dialog box appears. If the dialog box appears:

   - Select **Debug Only** to regenerate the debug configuration files containing the configuration for a debug session without trace capability.
   - Select **Full Debug and Trace** to regenerate the debug configuration files with full debug and trace information.
   - Select **Return to PCE** to manually provide the missing information.

   Note

   If you return to the PCE, check that the component interconnect information is accurate. If required, edit the information, save the platform configuration, and rebuild the platform configuration.

5. After the platform has built successfully, reapply any manual changes you had made to your previous platform configuration.

10 Using debug probes with Arm Development Studio

In this section, we describe how to connect a debug probe between Arm® Development Studio and your target to enable additional debug and trace functionality.

10.1 Overview: Debug Probes and Arm Development Studio

Connect a debug probe between Arm® Development Studio and your target to enable additional debug and trace functionality.

Supported probes

Arm Development Studio automatically recognizes the Arm DSTREAM family and Keil® ULINK™ family debug probes, as well as some third party probes. You can use other probes too, but they require additional configuration for Arm Development Studio to recognize them.

The DSTREAM and ULINK probes implement JTAG and SWD interfaces that communicate with the CoreSight™ debug components on your target.

- DSTREAM family:
  - Arm DSTREAM-ST
  - Arm DSTREAM-PT
  - Arm DSTREAM-HT
  - Arm DSTREAM-XT

Note

Although the Arm DSTREAM probe is supported, it is discontinued and is no longer available to purchase.

- ULINK family:
  - Keil ULINK2
  - Keil ULINKpro
  - Keil ULINKpro D
  - Keil ULINKplus
- Third party debug probes:
  - ST-Link
  - Cadence virtual debug

- FTDI MPSSE JTAG

Note

If you are using the FTDI MPSSE JTAG adapter on Linux, the OS automatically installs an incorrect driver when you connect this adapter. For details on how to fix this issue, see Troubleshooting: FTDI probe incompatible driver error in the *Arm Development Studio User Guide*.

- USB Blaster II

Note

If you are using the USB-Blaster debug units, Arm Debugger can connect to Arria V SoC, Arria 10 SoC, Cyclone V SoC and Stratix 10 boards. To enable the connections, ensure that the environment variable `QUARTUS_ROOTDIR` is set and contains the path to the Quartus tools installation directory:

- On Windows, this environment variable is usually set by the Quartus tools installer.
- On Linux, you might have to manually set the environment variable to the Quartus tools installation path. For example, `~/<quartus_tools_installation_directory>/qprogrammer`.

For information on installing device drivers for USB-Blaster and USB-Blaster II, consult your Quartus tools documentation.

Using a debug probe with Arm Development Studio

To use your debug probe with Arm Development Studio, you must add it to your debug connection:

- If you are using a supported probe, Arm Development Studio automatically detects your device when you plug it in. You can then configure your debug connection using the **Debug Configurations** editor.
- If you are using a probe that is not listed as supported, you must provide Arm Development Studio with its details. See Add a third-party debug probe for guidance on how to do this.

Related information

Arm Developer - Debug probes

10.2 Configure DSTREAM-HT trace using the Arm Development Studio Platform Configuration Editor

Use the Platform Configuration Editor (PCE) in Arm® Development Studio to add support for the DSTREAM-HT system to your platform configuration.

10.2.1 Create a DSTREAM-HT enabled platform configuration

Use the **New Platform** dialog box in Development Studio to create a new platform configuration with DSTREAM-HT support.

Before you begin

- Install the latest release of Arm® Development Studio which contains support for the latest debug hardware from Arm.
- Ensure your DSTREAM-HT system has the latest firmware version installed.
- If you autodetect your target, ensure that you connect your debug adapter and targets, or that you have the connection address.
- If you import an existing RDDI configuration file, SDF file (`.rcf`, `*.rvc`, `*.sdf`), CoreSight Creator file (`*.xml`), or CMM script (`*.cmm`), ensure that you have access to these files.

About this task

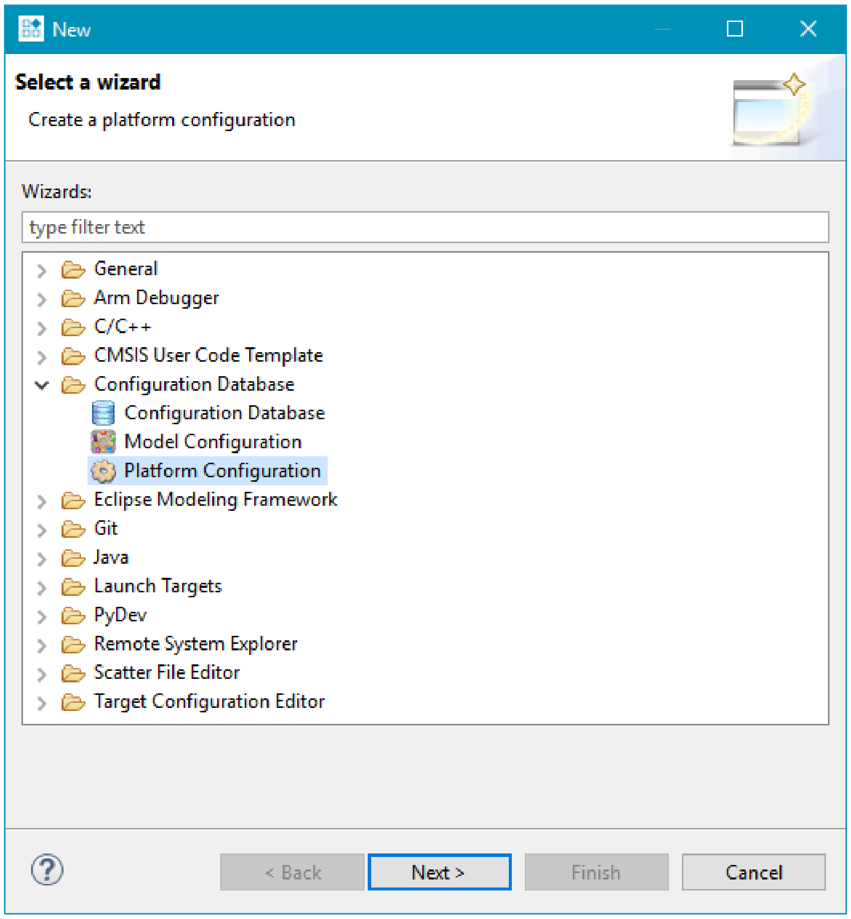
Warning

Generating a new platform configuration overwrites any DTSL scripts and other platform files that might exist for your current configuration. Backup your configuration if you have an existing configuration and have applied manual changes to your configuration.

Procedure

1. Open the new project dialog box. In the **Project Explorer**, right-click, and select **File** > **New** > **Other** > **Platform Configuration**.
2. Select **Configuration Database** > **Platform Configuration** and then click **Next** to view the **Create Platform Configuration** dialog box.

Figure 10-1: Select create a platform configuration.



New

Select a wizard

Create a platform configuration

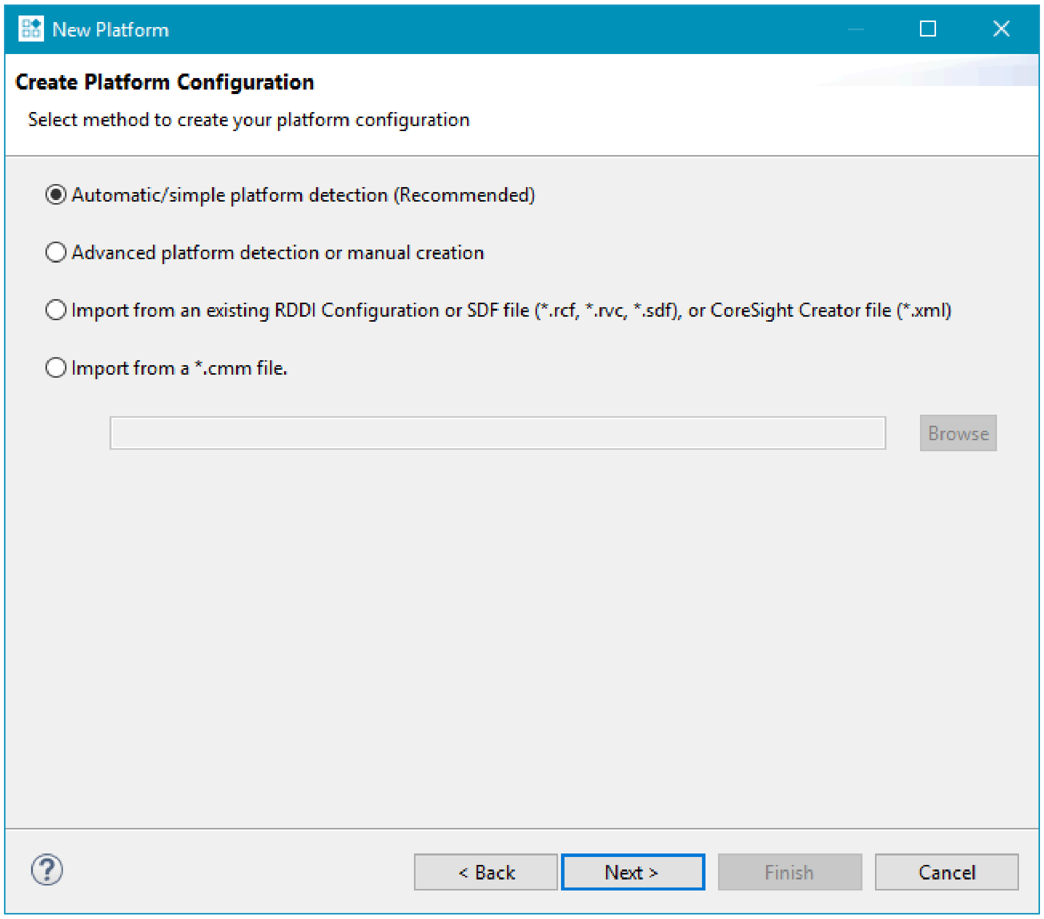
Wizards:

type filter text

- General
- Arm Debugger
- C/C++
- CMSIS User Code Template
- Configuration Database
  - Configuration Database
  - Model Configuration
  - Platform Configuration
- Eclipse Modeling Framework
- Git
- Java
- Launch Targets
- PyDev
- Remote System Explorer
- Scatter File Editor
- Target Configuration Editor

3. Select a method to create the configuration for your platform and click **Next**.

Figure 10-2: Platform creation options



Create Platform Configuration

Select method to create your platform configuration

- [✓ Automatic/simple platform detection (Recommended)]
- Advanced platform detection or manual creation
- Import from an existing RDDI Configuration or SDF file (*.rcf, *.rvc, *.sdf), or CoreSight Creator file (*.xml)
- Import from a *.cmm file.

Choose from:

- **Automatic/simple platform detection**

  Note: Arm recommends this option.

  Automatically detects devices that are present on your platform, use this option. After autodetection, you can add more devices and specify how the devices are interconnected.

- **Advanced platform detection or manual creation**

  This option gives you control over the individual stages that are involved in reading the device information from your platform. This option is useful, for example, to read certain device information that might make the platform unresponsive. For more information, see Manual platform configuration.

- **Import from an existing RDDI configuration file or SDF file (*.rcf, *.rvc, *.sdf), or CoreSight Creator file (*.xml)**

Use this option if you already have a configuration file for your platform.

Imports minimal information about the components available on your platform, such as the base address. After importing, you can manually provide additional information and links between the components to enable full debug and trace support.

- **Import from a *.cmm file**

  Imports a platform configuration from a CMM script (`*.cmm`) file.

  CMM scripts can contain target description information such as:

  ◦ JTAG pre- and post- IR/DR bit information.

  ◦ Core types.

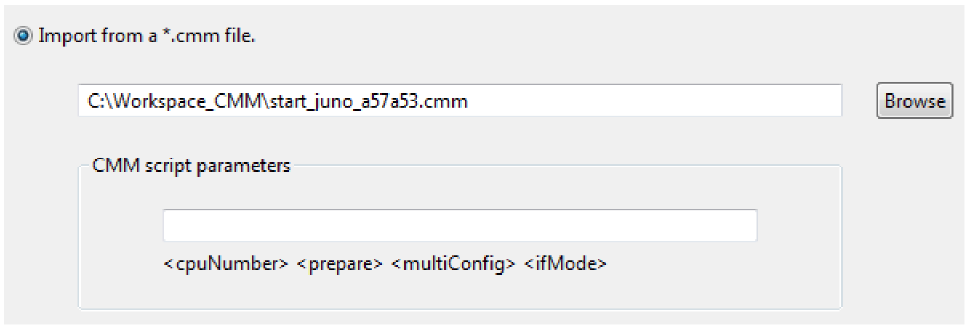
  ◦ Device base addresses.

  ◦ CoreSight topology information.

  The PCE uses this information to create a Development Studio platform configuration, complete with custom DTSL control tabs for trace configuration.

  Depending on the value of parameters that are passed to the script, some CMM scripts describe different targets, or different cores and trace devices in the same target. If a CMM script requires parameters, enter them in the **CMM script parameters** field.

Figure 10-3: Enter CMM script parameters



Import from a *.cmm file.

C:\Workspace_CMM\start_juno_a57a53.cmm

CMM script parameters

<cpuNumber> <prepare> <multiConfig> <ifMode>

If you are autodetecting using **Automatic/simple platform detection** or **Advanced platform detection or manual creation** (using **Autodetect Platform in **Debug Adapter** panel), Development Studio connects to the platform and reads all the device information that it can from the platform.

Warning

Depending on your target, you might receive warnings and errors. When Development Studio cannot obtain this information from the platform, it does not make any assumptions about how the devices are connected. You must provide this information in the PCE view.

Debug functionality succeeds if all cores are correctly detected. Where the connections between cores are not detected, debug succeeds, but trace and cross-triggering functionality might be limited or missing.

---

4. (Optional - **Automatic/simple platform detection** only) Save or edit your autodetected configuration. The following list describes the next steps available for your autodetected platform:

    - **Save a Debug-Only Arm DS Platform Configuration**

      Saves a debug-only configuration to your configuration database.

    - **Save a Debug and Trace Arm DS Platform Configuration**

      Saves a debug and trace configuration to your configuration database. You can open this in Development Studio later to modify it, or you can use it to connect to and debug the platform later.

    - **Edit platform in Arm DS Platform Configuration Editor**

      Saves the configuration to your configuration database and opens the PCE view.

      In the PCE view, you can provide information about the platform that Development Studio was unable to autodetect.

      For more information, see Edit a platform configuration.

    Note

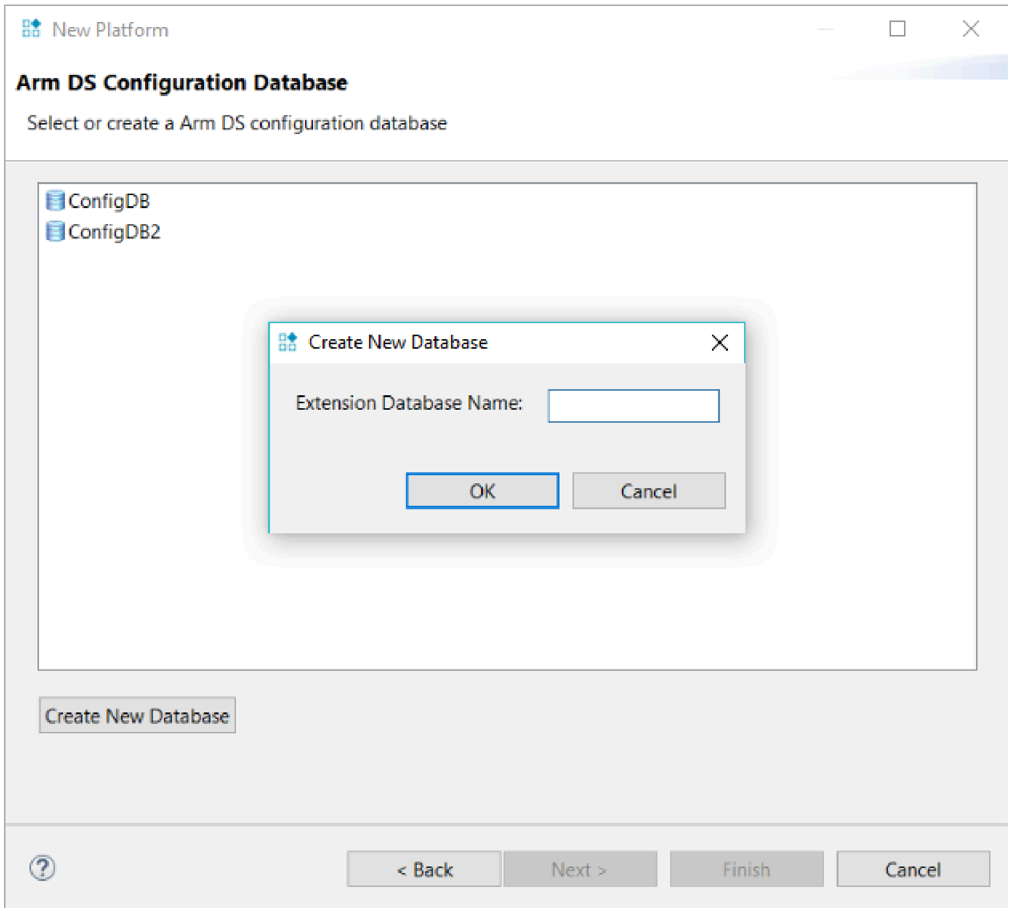
    - All Development Studio platform configurations must be stored in a configuration database.
    - You can open the configuration in Development Studio later to modify it, or you can use it to connect to and debug the platform.

    Select an option and click **Next**.

    The **New Platform** dialog box opens.

5. Select an existing configuration database from the list, or create a new one. To create a new configuration database, click **Create New Database**, provide a name for your new configuration database in the prompt, then click **OK** to save your configuration database.

Figure 10-4: Create new configuration database

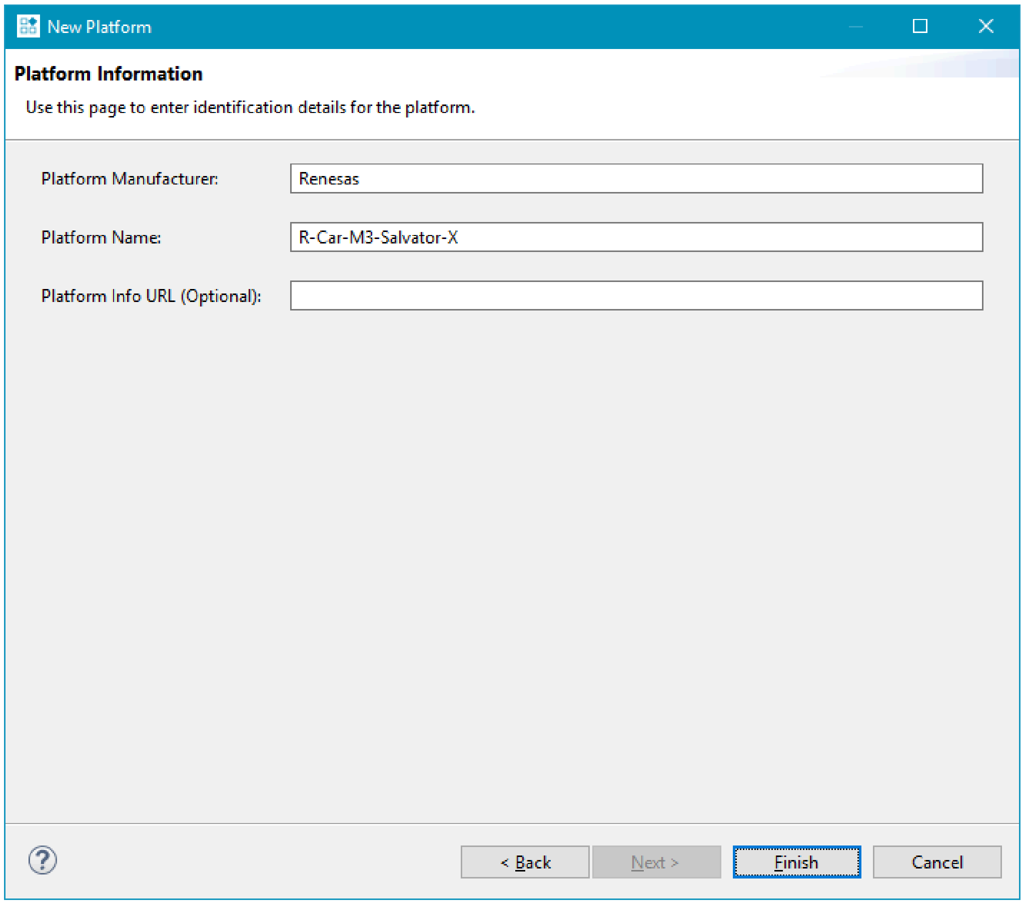


6. Click **Next >**.
   The **Platform Information** dialog box opens.
7. Enter the **Platform Manufacturer**, for example **Renesas**. Enter the **Platform Name**, for example **R-Car-M3-Salvator-X**. Optionally, if you want to provide a URL link to information about the platform, enter it in **Platform Info URL**.

Note

When you select a debug activity for the platform, the URL appears in the **Debug Configurations** panel.

Figure 10-5: New platform information



Platform Information

Use this page to enter identification details for the platform.

Platform Manufacturer:	Renesas
Platform Name:	R-Car-M3-Salvator-X
Platform Info URL (Optional):	

8. Click **Finish**.
9. (Optional - **Advanced platform detection or manual creation** only) Complete your configuration:
   - Configure your debug hardware in **Debug Adapter** panel.
   - Autodetect your platform. In **Debug Adapter** panel under **Autodetect** tab, click **Autodetect Platform**.

Warning

Depending on your target, you might receive warnings and errors. When Development Studio cannot obtain this information from the platform, it does not make any assumptions about how the devices are connected. You must provide this information in the PCE view.

Always review the information that has been collected before deciding what further action to take. If Development Studio fails to read information, it might indicate a deeper problem. For more information, see Hardware platform bring-up in Development Studio.

- Check, edit, and save your configuration. If required, edit the configuration then select **File** > **Save**.

Results

The Platform Configuration Editor (PCE) view opens in the right-hand panel. You can view the configuration database and platform in the **Project Explorer**.

Next steps

After successfully creating the platform configuration with DSTREAM-HT support, the next step is to modify the configuration usecase scripts to accept target-specific values.

10.2.2 Customize the configuration usecase script for your target

After creating the platform configuration for your platform, you must customize the configuration script to initialize and train the HSSTP trace link at connection time.

Before you begin

- You require a platform configuration for your target generated using the Platform Configuration Editor (PCE) in Arm® Development Studio.
- You require your target SoC user documentation. As part of the steps below you must modify the `configureTargetHSSTPLink(memAccessDevice)` and `startTargetHSSTPTraining(memAccessDevice)` functions with target-specific values. Refer to your target SoC documentation for the required values.

About this task

The HSSTP configuration that is generated by the PCE contains the `hsstp_usecase.py` file. The contents of the file:

- Contains target-specific functions to initialize the HSSTP trace subsystem.
- Trains the HSSTP trace link at connection time.
- Initiates a training sequence.
- Finally, starts HSSTP trace output.

Procedure

1. In the **Project Explorer**, browse to the configuration database which contains the configuration for the platform you require.
2. Locate the `hsstp_usecase.py` file and open it with your preferred text editor.
3. In the `hsstp_usecase.py` file contents, locate the `configureTargetHSSTPLink(memAccessDevice)` and

`startTargetHSSTPTraining(memAccessDevice)` functions and modify it to the values specific to your target.

Results

The configuration usecase script specific to your target is now ready to run.

Next steps

Run the script after you create a debug configuration for your application and target.

10.2.3 Create debug configuration and connect to the target

For DSTREAM-HT, create a debug hardware connection to your target to download, execute, debug your application, and capture trace.

Before you begin

- Ensure that you have customized your configuration usecase script for your specific target.
- Ensure that your target is connected correctly to the DSTREAM-HT unit.
- Ensure that your target is powered on. Refer to the documentation supplied with the target for more information.
- Ensure that the debug hardware probe connecting your target to your workstation is powered on and working.

About this task

After customizing the configuration usecase script for your target, create a debug hardware connection and connect to your hardware.

Note

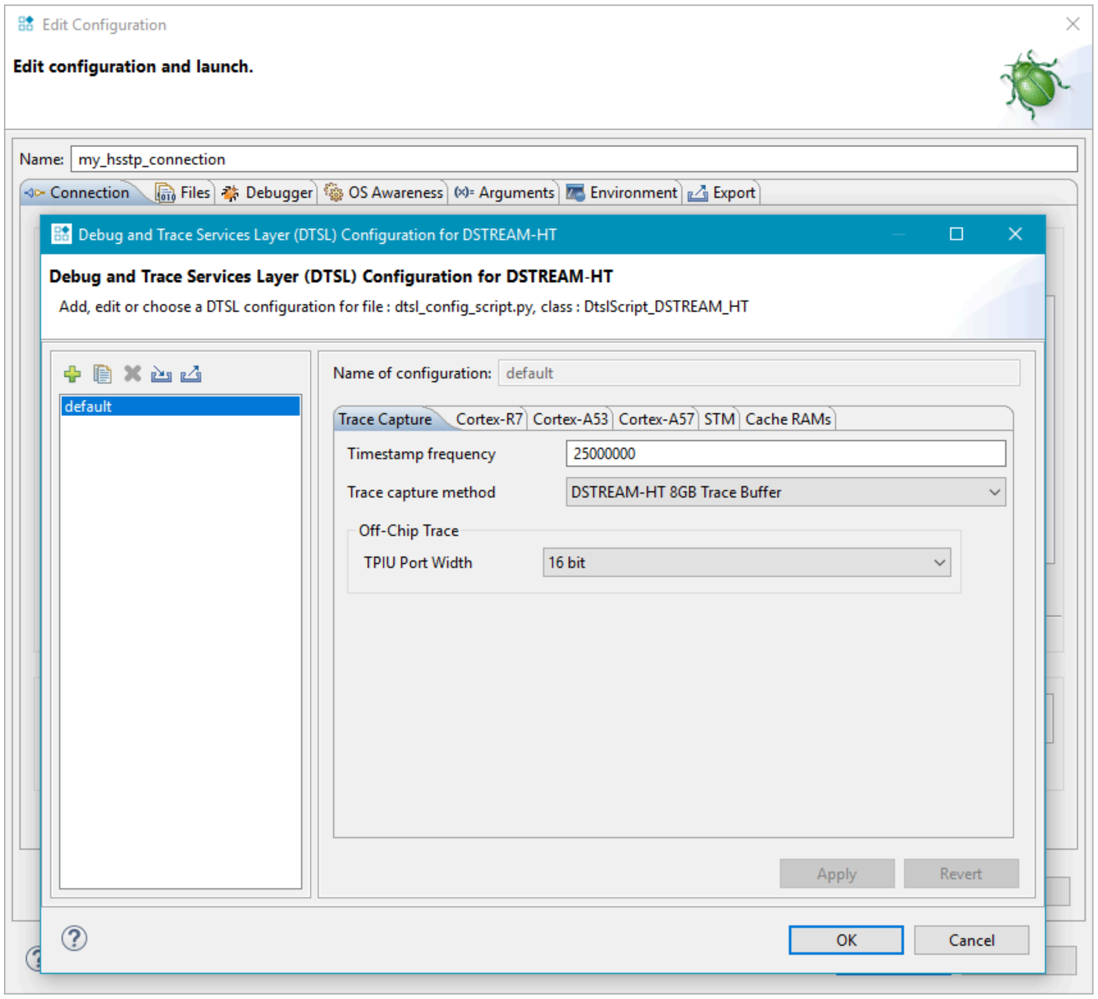
The following steps use the Renesas R-Car H3 target as an example, but demonstrates the general concept. Refer to your target SoC documentation for any specific values required by your target.

Procedure

1. From the Arm® Development Studio main menu, select **File** > **New** > **Hardware Connection**.
2. In the **Hardware Connection** dialog box, specify the details of the connection:
   a) In **Debug Connection** give the debug connection a name, for example **my_hsstp_connection** and click **Next**.
   b) In **Target Selection** select a target, for example **Renesas** > **R-Car H3** and click **Finish**. This completes the initial connection configuration and opens the **Edit Configuration** dialog.
3. To specify the target and connection settings, in the **Edit Configuration** dialog box:
   a) Select the **Connection** tab.
   b) In the **Select target** panel confirm the target that is selected.
   c) In the **Target Connection** list, select **DSTREAM Family**.

d) In the **Connections** area, enter the **Connection** name or IP address of your debug hardware adapter. If your connection is local, click **Browse** and select the connection using the **Connection Browser**.
e) In **DSTL Options**, click **Edit** to display the **Debug and Trace Services Layer (DTSL) Configuration for DSTREAM-HT** dialog.
f) In the **Trace Capture** tab, set the **Trace capture method** as **DSTREAM-HT 8GB Trace Buffer**

Figure 10-6: Edit the DTSL settings



Name: my_hsstp_connection

Debug and Trace Services Layer (DTSL) Configuration for DSTREAM-HT

Add, edit or choose a DTSL configuration for file : dtsl_config_script.py, class : DtslScript_DSTREAM_HT

default

Name of configuration: default

Timestamp frequency: 25000000

Trace capture method: DSTREAM-HT 8GB Trace Buffer

Off-Chip Trace

TPIU Port Width: 16 bit

g) Select the other trace settings you require.
h) Click **OK** to close the dialog box and return to the **Edit Configuration** dialog box.

4. Click the **Files** tab to specify your application and additional resources to download to the target:
   a) To load your application on the target at connection time, in the **Target Configuration** area, specify your application in the **Application on host to download** field.
   b) To debug your application at source level, select **Load symbols**.
   c) To load additional resources, for example, additional symbols or peripheral description files from a directory, add them in the **Files** area. Click **+** to add resources, click **-** to remove resources.
5. Use the **Debugger** tab to configure debugger settings.

a) In the **Run control** area:
   - Specify if you want to **Connect only** to the target or **Debug from entry point**. If you want to start debugging from a specific symbol, select **Debug from symbol**.
   - To run target or debugger initialization scripts, select the relevant options and specify the script paths.
   - To specify at debugger start up, select **Execute debugger commands** options and specify the commands.

b) The debugger uses your workspace as the default working directory on the host. To change the default location, deselect the **Use default** option under **Host working directory** and specify the new location.

c) In the **Paths** area, specify any directories on the host that contain your application files in the **Source search directory** field.

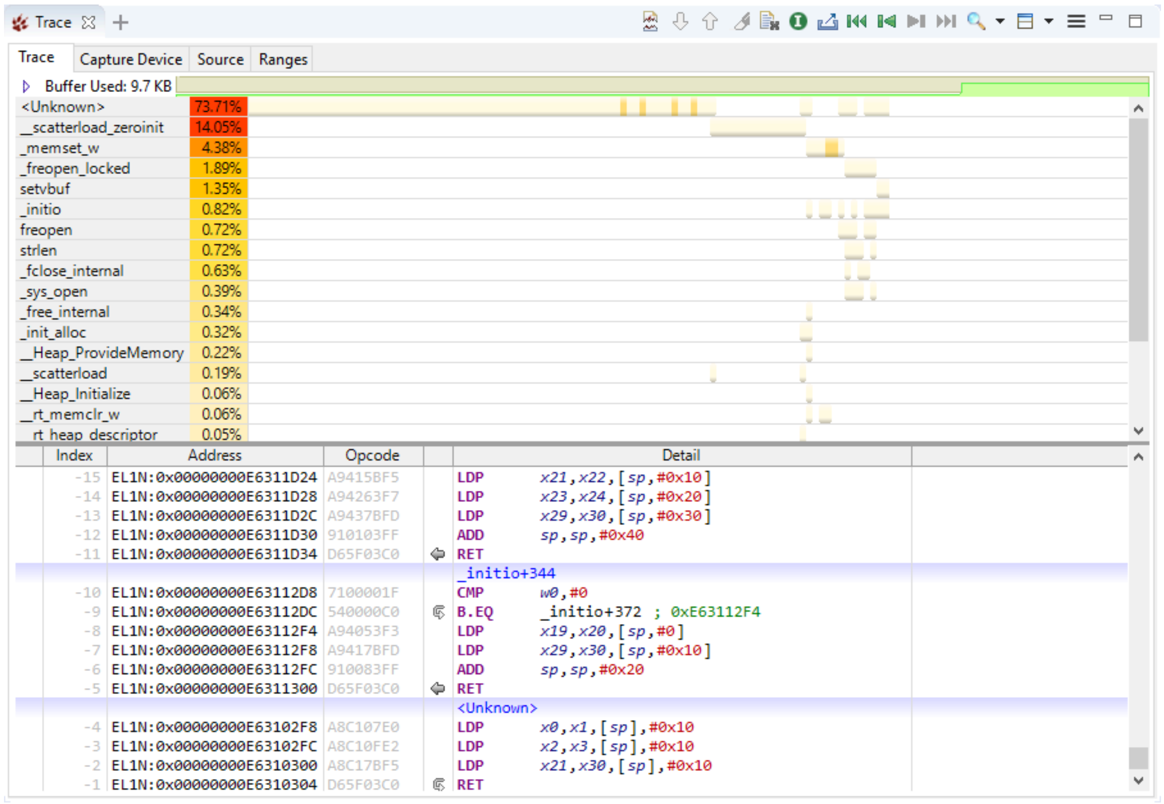
d) To use additional resources, click **Add resource (+)** to add resources. Click **Remove resources (-)** to remove resources.

6. [Optional] Use the **Arguments** tab to enter arguments that are passed to the `main()` function of the application when the debug session starts. The debugger uses semihosting to pass arguments to `main()`.
7. [Optional] Use the **Environment** tab to create and configure environment variables to pass into the launch configuration when it is executed.
8. Click **Apply** and then **Debug** to connect to the target and start debugging session.

Results

The trace is now setup for the HSSTP target. You can view the trace output in the **Trace** view.

Figure 10-7: Trace view HSSTP output



<Unknown>	73.71%
__scatterload_zeroinit	14.05%
_memset_w	4.38%
_freopen_locked	1.89%
setvbuf	1.35%
_initio	0.82%
freopen	0.72%
strlen	0.72%
_fclose_internal	0.63%
_sys_open	0.39%
_free_internal	0.34%
_init_alloc	0.32%
__Heap_ProvideMemory	0.22%
__scatterload	0.19%
__Heap_Initialize	0.06%
__rt_memclr_w	0.06%
rt_heap_descriptor	0.05%

```
Index  Address                    Opcode       Detail
  -15  EL1N:0x00000000E6311D24   A9415BF5     LDP     x21,x22,[sp,#0x10]
  -14  EL1N:0x00000000E6311D28   A94263F7     LDP     x23,x24,[sp,#0x20]
  -13  EL1N:0x00000000E6311D2C   A9437BFD     LDP     x29,x30,[sp,#0x30]
  -12  EL1N:0x00000000E6311D30   910103FF     ADD     sp,sp,#0x40
  -11  EL1N:0x00000000E6311D34   D65F03C0     RET
                                              _initio+344
  -10  EL1N:0x00000000E63112D8   7100001F     CMP     w0,#0
   -9  EL1N:0x00000000E63112DC   540000C0     B.EQ    _initio+372 ; 0xE63112F4
   -8  EL1N:0x00000000E63112F4   A94053F3     LDP     x19,x20,[sp,#0]
   -7  EL1N:0x00000000E63112F8   A9417BFD     LDP     x29,x30,[sp,#0x10]
   -6  EL1N:0x00000000E63112FC   910083FF     ADD     sp,sp,#0x20
   -5  EL1N:0x00000000E6311300   D65F03C0     RET
                                              <Unknown>
   -4  EL1N:0x00000000E63102F8   A8C107E0     LDP     x0,x1,[sp],#0x10
   -3  EL1N:0x00000000E63102FC   A8C10FE2     LDP     x2,x3,[sp],#0x10
   -2  EL1N:0x00000000E6310300   A8C17BF5     LDP     x21,x30,[sp],#0x10
   -1  EL1N:0x00000000E6310304   D65F03C0     RET
```

If your target requires additional setup, make changes to the `hsstp_usecase.py` in the **Scripts** view and run the `default` component of the script.

10.2.4 Additional HSSTP target configuration setup

You can make additional changes to your target HSSTP configuration setup without disconnecting from the target.

Before you begin

- Ensure that you have customized your configuration usecase script for your specific target.
- Ensure that your debug connection is set up and working.

About this task

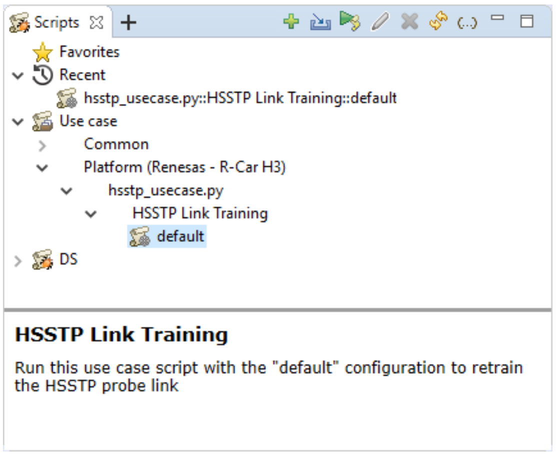
For additional target HSSTP configuration changes, in the **Scripts** view, make your changes to `hsstp_usecase.py` file. Then run the `default` component of the script. This initiates a training sequence for your target and restarts the HSSTP trace output.

Procedure

1. Open the **Scripts** view, and locate the `hsstp_usecase.py` file that you modified for your target.
2. Make any additional changes that you require.

3. Double-click the **default** stub to initiate the training sequence for your target and start HSSTP trace output.

Figure 10-8: Scripts view HSSTP training



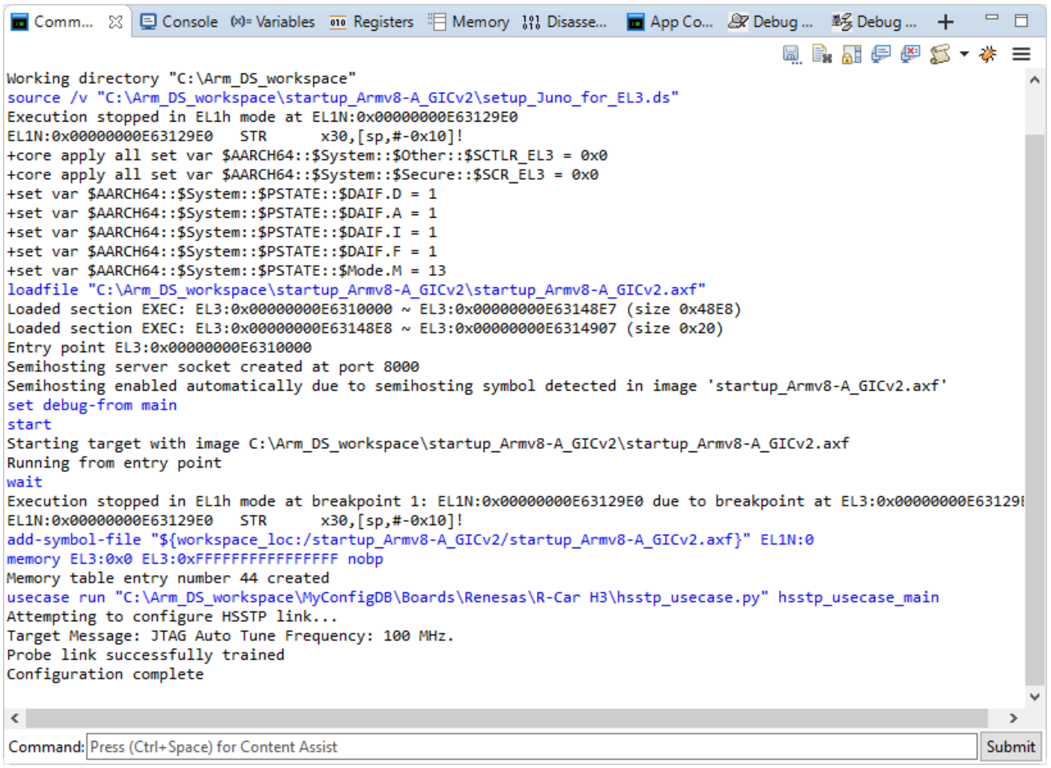
HSSTP Link Training

Run this use case script with the "default" configuration to retrain the HSSTP probe link

Results

You can view the status of the script execution in the **Commands** view.

Figure 10-9: Scripts execution status



```
Working directory "C:\Arm_DS_workspace"
source /v "C:\Arm_DS_workspace\startup_Armv8-A_GICv2\setup_Juno_for_EL3.ds"
Execution stopped in EL1h mode at EL1N:0x00000000E63129E0
EL1N:0x00000000E63129E0   STR      x30,[sp,#-0x10]!
+core apply all set var $AARCH64::$System::$Other::$SCTLR_EL3 = 0x0
+core apply all set var $AARCH64::$System::$Secure::$SCR_EL3 = 0x0
+set var $AARCH64::$System::$PSTATE::$DAIF.D = 1
+set var $AARCH64::$System::$PSTATE::$DAIF.A = 1
+set var $AARCH64::$System::$PSTATE::$DAIF.I = 1
+set var $AARCH64::$System::$PSTATE::$DAIF.F = 1
+set var $AARCH64::$System::$PSTATE::$Mode.M = 13
loadfile "C:\Arm_DS_workspace\startup_Armv8-A_GICv2\startup_Armv8-A_GICv2.axf"
Loaded section EXEC: EL3:0x00000000E6310000 ~ EL3:0x00000000E63148E7 (size 0x48E8)
Loaded section EXEC: EL3:0x00000000E63148E8 ~ EL3:0x00000000E6314907 (size 0x20)
Entry point EL3:0x00000000E6310000
Semihosting server socket created at port 8000
Semihosting enabled automatically due to semihosting symbol detected in image 'startup_Armv8-A_GICv2.axf'
set debug-from main
start
Starting target with image C:\Arm_DS_workspace\startup_Armv8-A_GICv2\startup_Armv8-A_GICv2.axf
Running from entry point
wait
Execution stopped in EL1h mode at breakpoint 1: EL1N:0x00000000E63129E0 due to breakpoint at EL3:0x00000000E63129
EL1N:0x00000000E63129E0   STR      x30,[sp,#-0x10]!
add-symbol-file "${workspace_loc:/startup_Armv8-A_GICv2/startup_Armv8-A_GICv2.axf}" EL1N:0
memory EL3:0x0 EL3:0xFFFFFFFFFFFFFFFF nobp
Memory table entry number 44 created
usecase run "C:\Arm_DS_workspace\MyConfigDB\Boards\Renesas\R-Car H3\hsstp_usecase.py" hsstp_usecase_main
Attempting to configure HSSTP link...
Target Message: JTAG Auto Tune Frequency: 100 MHz.
Probe link successfully trained
Configuration complete
```

Related information

Create a DSTREAM-HT enabled platform configuration on page 238

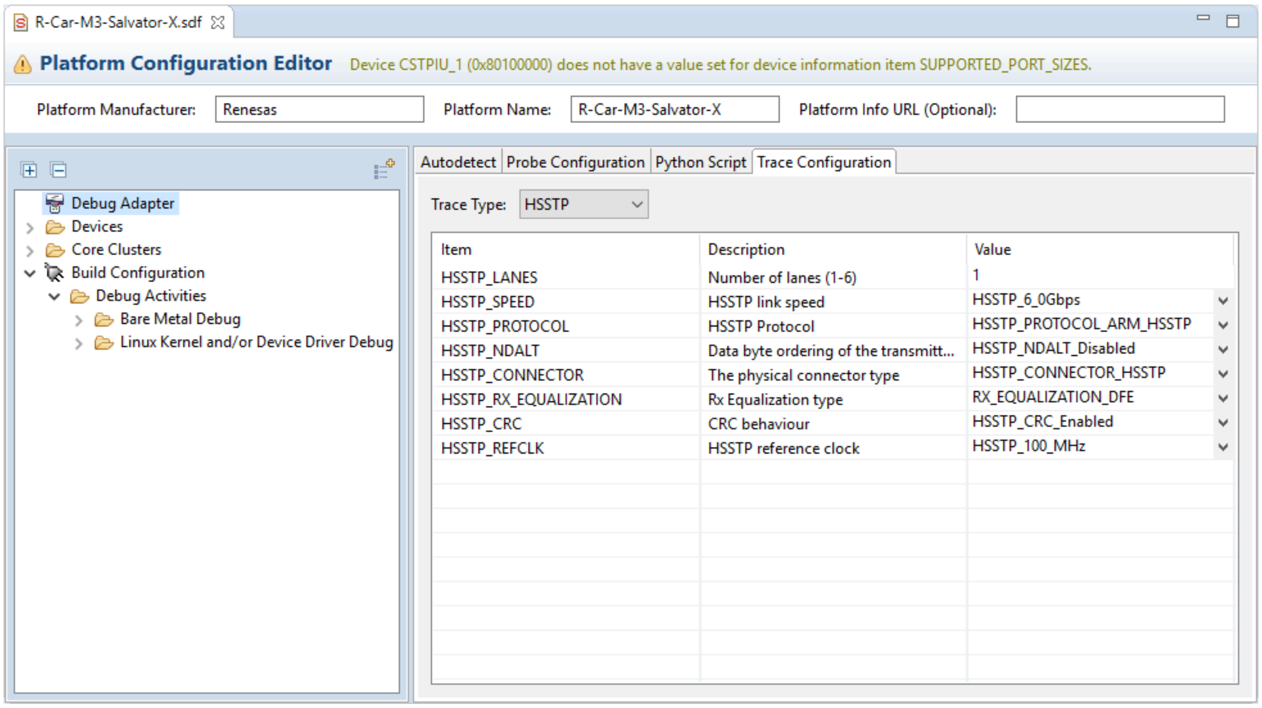
10.2.5 DSTREAM-HT trace probe configuration

As part of configuring trace for your hardware, you can configure HSSTP trace.

Configure the DSTREAM-HT trace probe using the **Trace Configuration** settings available in the **Platform Configuration Editor** (PCE).

In PCE, select the configuration file, and then click **Debug Adapter** > **Trace Configuration** > **Trace Type** > **HSSTP**

Figure 10-10: Trace Configuration tab



Device CSTPIU_1 (0x80100000) does not have a value set for device information item SUPPORTED_PORT_SIZES.

Platform Manufacturer: Renesas
Platform Name: R-Car-M3-Salvator-X
Platform Info URL (Optional):

Trace Type: HSSTP

Item	Description	Value
HSSTP_LANES	Number of lanes (1-6)	1
HSSTP_SPEED	HSSTP link speed	HSSTP_6_0Gbps
HSSTP_PROTOCOL	HSSTP Protocol	HSSTP_PROTOCOL_ARM_HSSTP
HSSTP_NDALT	Data byte ordering of the transmitt...	HSSTP_NDALT_Disabled
HSSTP_CONNECTOR	The physical connector type	HSSTP_CONNECTOR_HSSTP
HSSTP_RX_EQUALIZATION	Rx Equalization type	RX_EQUALIZATION_DFE
HSSTP_CRC	CRC behaviour	HSSTP_CRC_Enabled
HSSTP_REFCLK	HSSTP reference clock	HSSTP_100_MHz

Table 10-1: HSSTP trace configuration options

Configuration item name	Type	Description and supported values
HSSTP_LANES	Int32	Number of HSSTP lanes to use for trace. Currently, support is provided for 1 and 2-lane Arm HSSTP/Serial-ETM trace. Additional lane support is planned for later Arm® Development Studio releases. Supported values: 1-6.

Configuration item name	Type	Description and supported values
HSSTP_SPEED	Str	The HSSTP link speed to use. **Supported values:** • HSSTP_2_5Gbps • HSSTP_3_0Gbps • HSSTP_3_125Gbps • HSSTP_4_25Gbps • HSSTP_5_0Gbps • HSSTP_6_0Gbps • HSSTP_6_25Gbps • HSSTP_8_0Gbps • HSSTP_10_0Gbps • HSSTP_10_3125Gbps • HSSTP_12_0Gbps • HSSTP_12_5Gbps • HSSTP_SETM_1_5Gbps • HSSTP_SETM_3_0Gbps
HSSTP_PROTOCOL	Str	The HSSTP protocol to use. **Supported values:** • HSSTP_PROTOCOL_Arm_HSSTP • HSSTP_PROTOCOL_8_BIT_SETM • HSSTP_PROTOCOL_16_BIT_SETM • HSSTP_PROTOCOL_32_BIT_SETM
HSSTP_NDALT	Str	Data byte ordering of the transmitted data. **Supported values:** • HSSTP_NDALT_Disabled • HSSTP_NDALT_Enabled
HSSTP_CONNECTOR	Str	The physical connector type. **Supported values:** • HSSTP_CONNECTOR_HSSTP • HSSTP_CONNECTOR_SMA
HSSTP_RX_EQUALIZATION	Str	Rx equalization type **Supported values:** • RX_EQUALIZATION_DFE • RX_EQUALIZATION_LPM

Configuration item name	Type	Description and supported values
HSSTP_CRC	Str	Enable CRC check on HSSTP data. **Supported values:** • HSSTP_CRC_Disabled - Disable HSSTP CRC checks (See HSSTP specification). • HSSTP_CRC_Enabled - Enable HSSTP CRC checks. • HSSTP_CRC_Reversed - Enable HSSTP bit reversed CRC checks (necessary on some targets). • HSSTP_CRC_Ignored - Ignores an incorrectly implemented CRC.

10.2.6 Example HSSTP configurations provided with Arm Development Studio

The latest releases of Arm® Development Studio provides HSSTP configuration examples.

You can locate these example target configurations in the Configuration Database under `<install_directory>/sw/debugger/configdb/Boards`.

View the `hsstp_usecase.py` files in the example configurations to see how the `configureTargetHSSTPLink(memAccessDevice)` and `startTargetHSSTPTraining(memAccessDevice)` functions are implemented. Although the examples are target-specific, it demonstrates the general concept of the modifications required for your platform.

Related information

Configuration database on page 229

10.3 Configure DSTREAM-XT debug and trace using the Arm Development Studio Platform Configuration Editor

Use the Platform Configuration Editor (PCE) in Arm® Development Studio to add support for the DSTREAM-XT system to your platform configuration.

There are three use-cases for DSTREAM-XT:

- PCIe debug-only (no trace)
- JTAG/SWD debug with PCIe trace
- PCIe debug with PCIe trace

After you have created your platform configuration for DSTREAM-XT, you need to configure the debug and/or trace configuration settings, depending on your use-case. You must configure these settings before you create your debug connection and connect to your target.

10.3.1 Create a DSTREAM-XT enabled platform configuration

Use the **New Platform** dialog box in Development Studio to create a new platform configuration with DSTREAM-XT support.

Before you begin

- Install the latest release of Arm® Development Studio which contains support for the latest debug hardware from Arm.

Note

You must install Arm Development Studio version 2021.1/2021.b or later.

- Ensure your DSTREAM-XT system has the latest firmware version installed.

Note

- In Arm Development Studio, the latest firmware files are available at: `<Arm_Development_Studio_install_directory>/sw/debughw/firmware/`
- You must use firmware version 7.6 or later to detect the XT probe.

- If you autodetect your target, ensure that you have either physically connected your debug adapter and targets using USB, or that you have the connection address for a remote connection using TCP.
- If you import an existing RDDI configuration file, SDF file (`.rcf`, `*.rvc`, `*.sdf`), CoreSight™ Creator file (`*.xml`), or CMM script (`*.cmm`), ensure that you have access to these files.

About this task

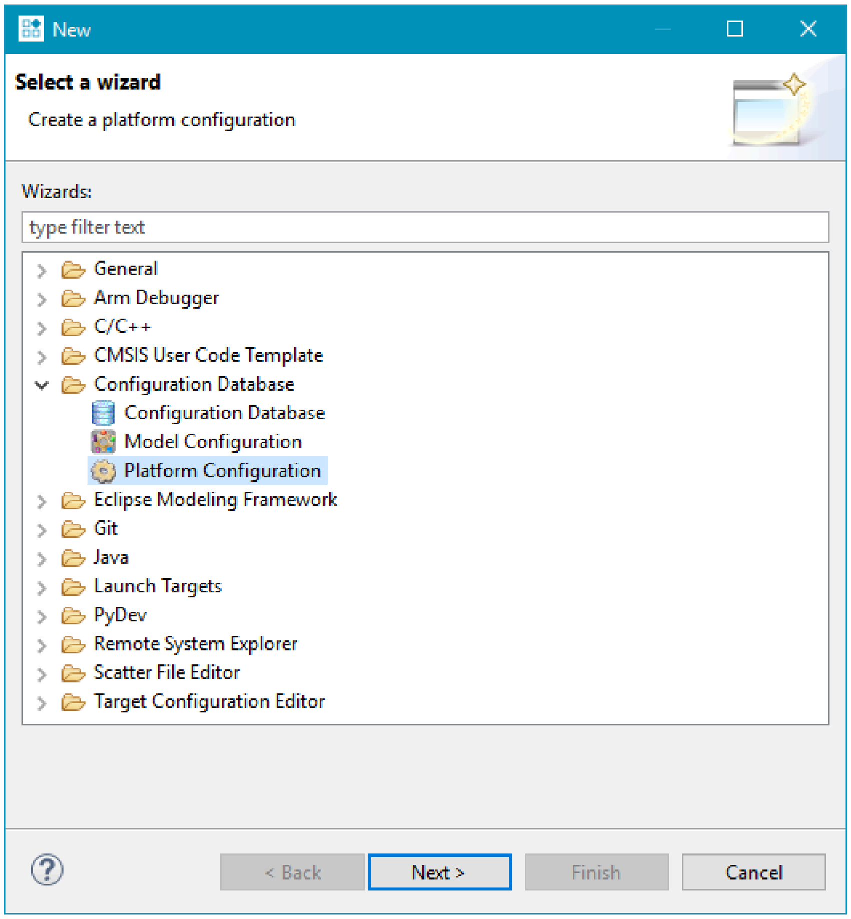
Warning

Generating a new platform configuration overwrites any DTSL scripts and other platform files that might exist for your current configuration. Backup your configuration if you have an existing configuration, and you have applied manual changes to your configuration.

Procedure

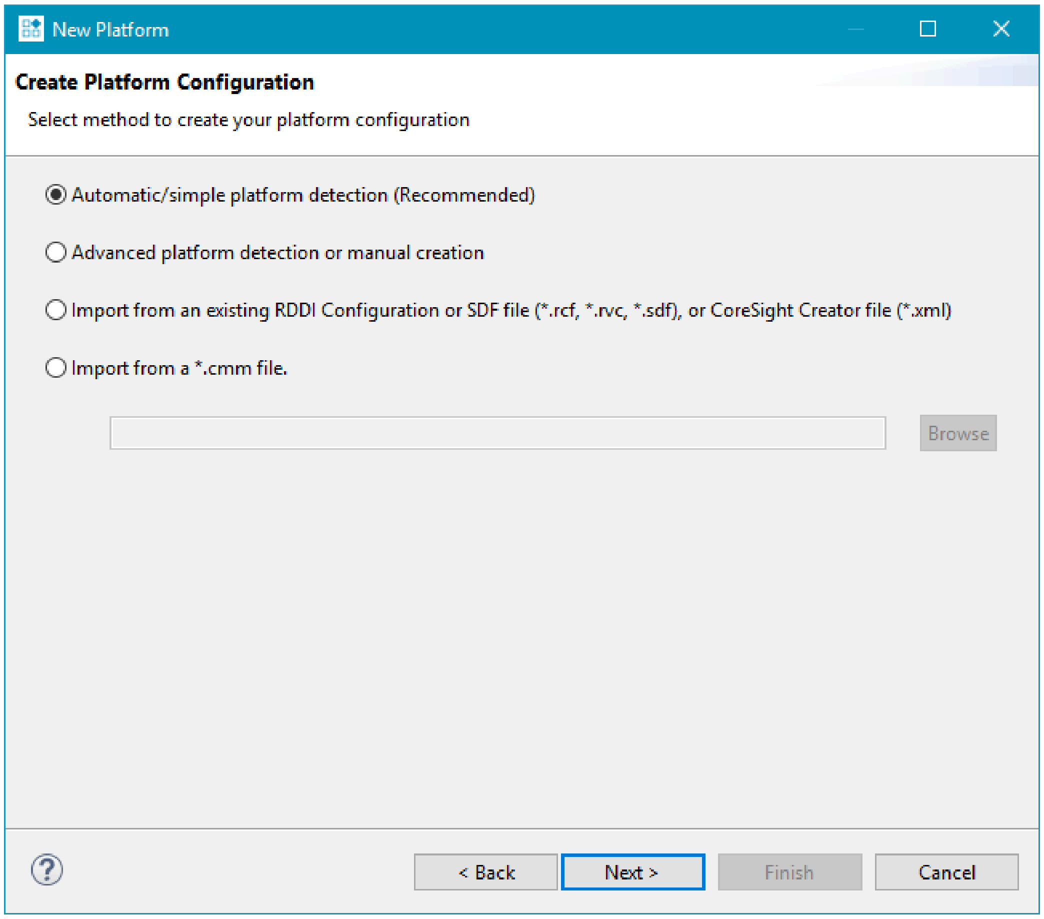
1. In the **Project Explorer**, right-click, and select **File** > **New** > **Other** > **Platform Configuration**.
2. Select **New** > **Platform Configuration** and then click **Next** to view the **Create Platform Configuration** dialog box.

Figure 10-11: Select create a platform configuration.



3. Select a method to create the configuration for your platform and click **Next**.

Figure 10-12: Platform creation options



New Platform

Create Platform Configuration

Select method to create your platform configuration

- [✓ Automatic/simple platform detection (Recommended)]
- Advanced platform detection or manual creation
- Import from an existing RDDI Configuration or SDF file (*.rcf, *.rvc, *.sdf), or CoreSight Creator file (*.xml)
- Import from a *.cmm file.

Choose from:

- **Automatic/simple platform detection**

  Note

  Arm recommends this option.

  Use this option to automatically detect devices that are present on your platform. After autodetection, you can add more devices and specify how the devices are interconnected.

- **Advanced platform detection or manual creation**

  This option gives you control over the individual stages that are involved in reading the device information from your platform. This option is useful, for example, to read certain device information that might make the platform unresponsive. For more information, see Manual platform configuration.

- **Import from an existing RDDI configuration file or SDF file (*.rcf, *.rvc, *.sdf), or CoreSight Creator file (*.xml)**

Use this option if you already have a configuration file for your platform.

Imports minimal information about the components available on your platform, such as the base address. After importing, you can manually provide additional information and links between the components to enable full debug and trace support.

- **Import from a *.cmm file**

  Imports a platform configuration from a CMM script (`*.cmm`) file.

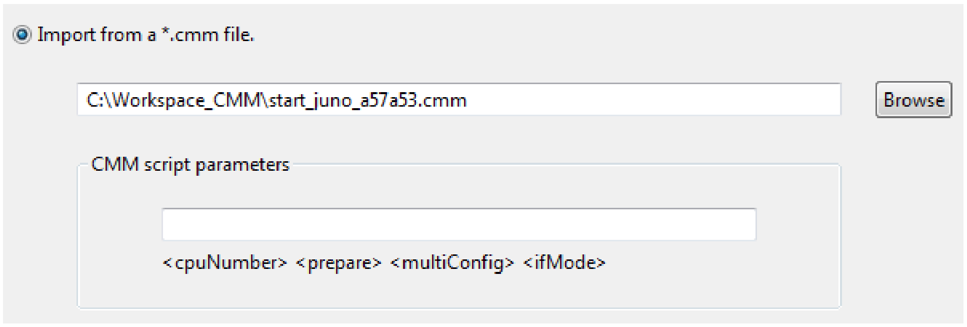
  CMM scripts can contain target description information such as:

  - JTAG pre- and post- IR/DR bit information.
  - Core types.
  - Device base addresses.
  - CoreSight topology information.

  The PCE uses this information to create a Development Studio platform configuration, complete with custom DTSL control tabs for trace configuration.

  Depending on the value of parameters that are passed to the script, some CMM scripts describe different targets, or different cores and trace devices in the same target. If a CMM script requires parameters, you can enter them in the **CMM script parameters** field.

Figure 10-13: Enter CMM script parameters



C:\Workspace_CMM\start_juno_a57a53.cmm

<cpuNumber> <prepare> <multiConfig> <ifMode>

If you are autodetecting using **Automatic/simple platform detection** or **Advanced platform detection or manual creation** (using **Autodetect Platform** in **Debug Adapter** panel), Development Studio connects to the platform and reads all the device information that it can collect from the platform.

Warning

Depending on your target, you might receive warnings and errors. When Development Studio cannot obtain device information from the platform, it does not make any assumptions about how the devices are connected. You must provide this information in the PCE view.

Debug functionality succeeds if all cores are correctly detected. Where the connections between cores are not detected, debug succeeds, but trace and cross-triggering functionality might be limited or missing.

---

4. (Optional - **Automatic/simple platform detection** only) Save or edit your autodetected configuration. The following list describes the next steps available for your autodetected platform:

   - **Save a Debug-Only Arm DS Platform Configuration**

     Saves a debug-only configuration to your configuration database.
   - **Save a Debug and Trace Arm DS Platform Configuration**

     Saves a debug and trace configuration to your configuration database. You can open this in Development Studio later to modify it, or you can use it to connect to and debug the platform later.
   - **Edit platform in Arm DS Platform Configuration Editor**

     Saves the configuration to your configuration database and opens the PCE view.

     In the PCE view, you can provide information about the platform that Development Studio was unable to autodetect.

     For more information, see Edit a platform configuration.

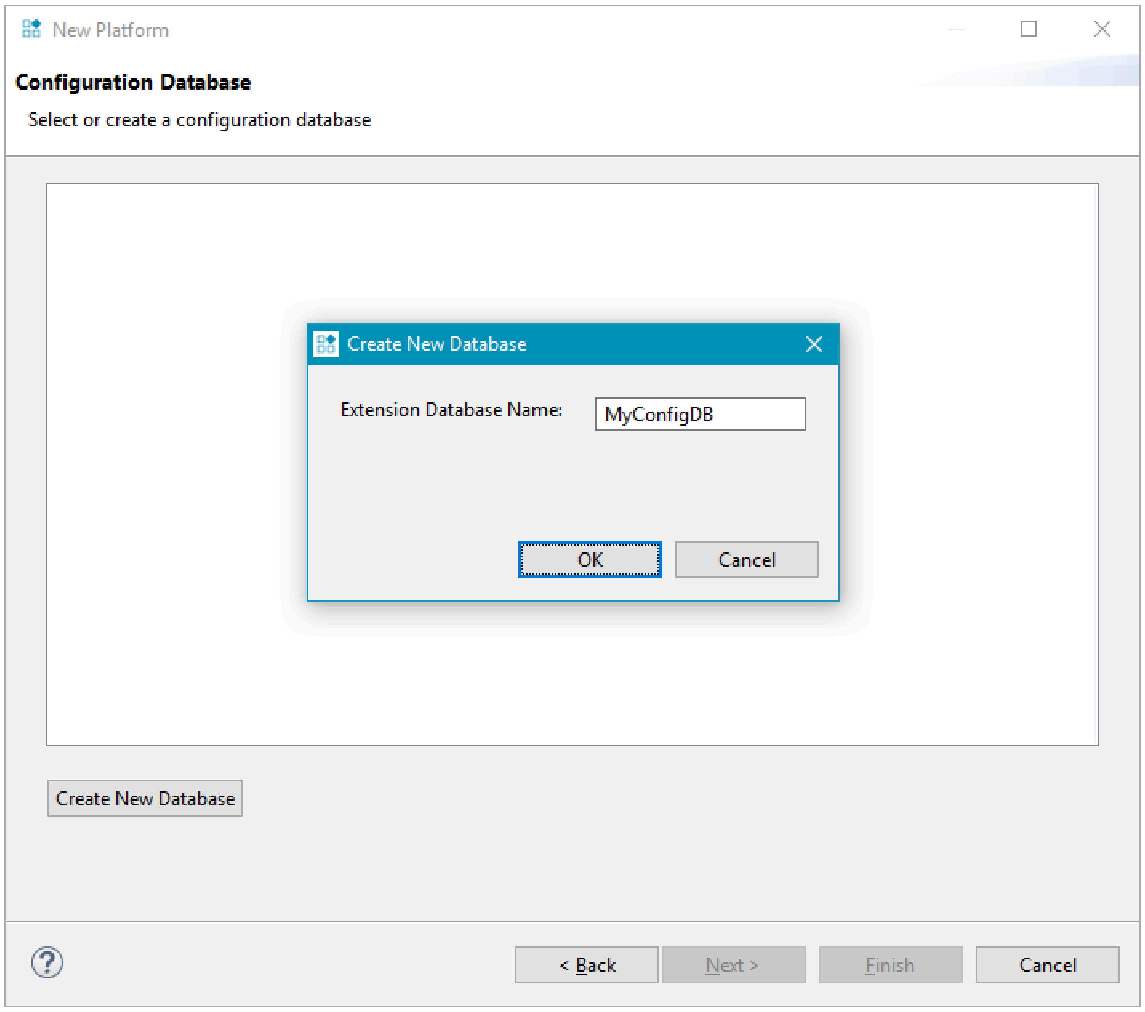
   Note

   - All Development Studio platform configurations must be stored in a configuration database.
   - You can open the configuration in Development Studio later to modify it, or you can use it to connect to and debug the platform.

   Select an option and click **Next**.

   The **New Platform** dialog box opens.
5. Select an existing configuration database from the list, or create a new one. To create a new configuration database, click **Create New Database**, provide a name for your new configuration database in the prompt, then click **OK** to save your configuration database.

Figure 10-14: Create new configuration database

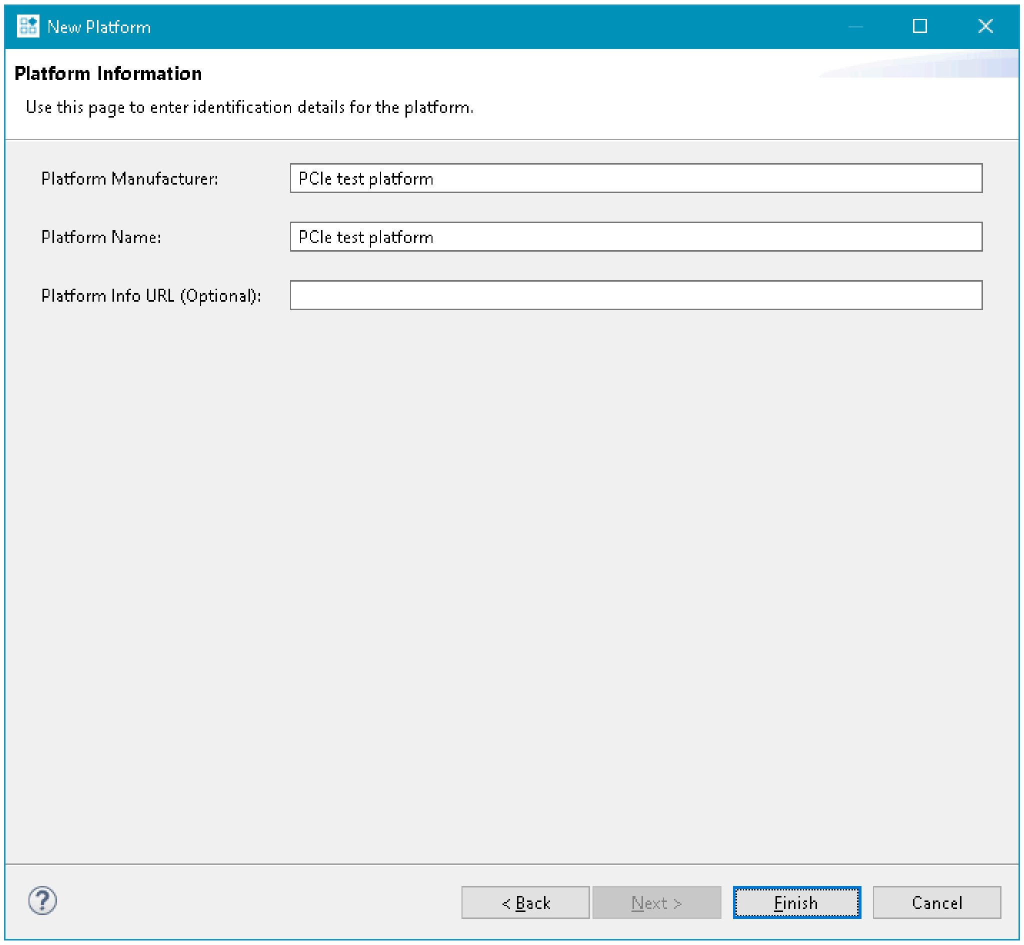


6. Click **Next >**.
   The **Platform Information** dialog box opens.
7. Enter the **Platform Manufacturer** and the **Platform Name**. Optionally, if you want to provide a URL link to information about the platform, enter it in **Platform Info URL**.

Note

When you select a debug activity for the platform, the URL appears in the **Debug Configurations** panel.

Figure 10-15: New platform information


New Platform
Platform Information
Use this page to enter identification details for the platform.
< Back
Next >
Finish
Cancel

Platform Manufacturer: PCIe test platform
Platform Name: PCIe test platform
Platform Info URL (Optional):

8. Click **Finish**.
9. (Optional - **Advanced platform detection or manual creation** only) Complete your configuration:
   - Configure your debug hardware in **Debug Adapter** panel.
   - Autodetect your platform. In **Debug Adapter** panel under **Autodetect** tab, click **Autodetect Platform**.

Warning

Depending on your target, you might receive warnings and errors. When Development Studio cannot obtain this information from the platform, it does not make any assumptions about how the devices are connected. You must provide this information in the PCE view.

Always review the information that has been collected before deciding what further action to take. If Development Studio fails to read information, it might indicate a deeper problem. For more information, see Hardware platform bring-up in Development Studio.

- Check, edit, and save your configuration. If required, edit the configuration then select **File** > **Save**.

Results

The **Platform Configuration Editor (PCE)** view opens in the right-hand panel. You can view the configuration database and platform in the **Project Explorer**.

Next steps

1. After successfully creating the platform configuration with DSTREAM-XT support, the next step is to configure the debug or trace settings in the generated SDF file.

   There are three ways to use DSTREAM-XT;

   - PCIe debug-only (no trace)
   - JTAG/SWD debug with PCIe trace
   - PCIe debug with PCIe trace

   The configuration items mentioned in DSTREAM-XT trace configuration are used by Arm Development Studio to manage the PCIe link. Therefore, if you want to use any of the PCIe link functionality for debug, you must modify both the *DSTREAM-XT debug* and *DSTREAM-XT trace* configuration items. If you only want to use PCIe trace functionality, then you only need to modify the *DSTREAM-XT trace* configuration items.

   The use-cases and items to configure are as follows:

   - PCIe debug-only (no trace): Configure all the items listed in DSTREAM-XT debug configuration and DSTREAM-XT trace configuration
   - JTAG/SWD debug with PCIe trace: Configure all the items listed in DSTREAM-XT trace configuration
   - PCIe debug with PCIe trace: Configure all the items listed in DSTREAM-XT debug configuration and DSTREAM-XT trace configuration
2. When you have finished editing the configuration items in the SDF file, right-click the file and select **Build Platform**.
3. The final item to edit, is the `pcie_bringup.py` script, to configure your link connection settings.
4. When you have finished editing these files, you can then Create debug configuration and connect to the target.

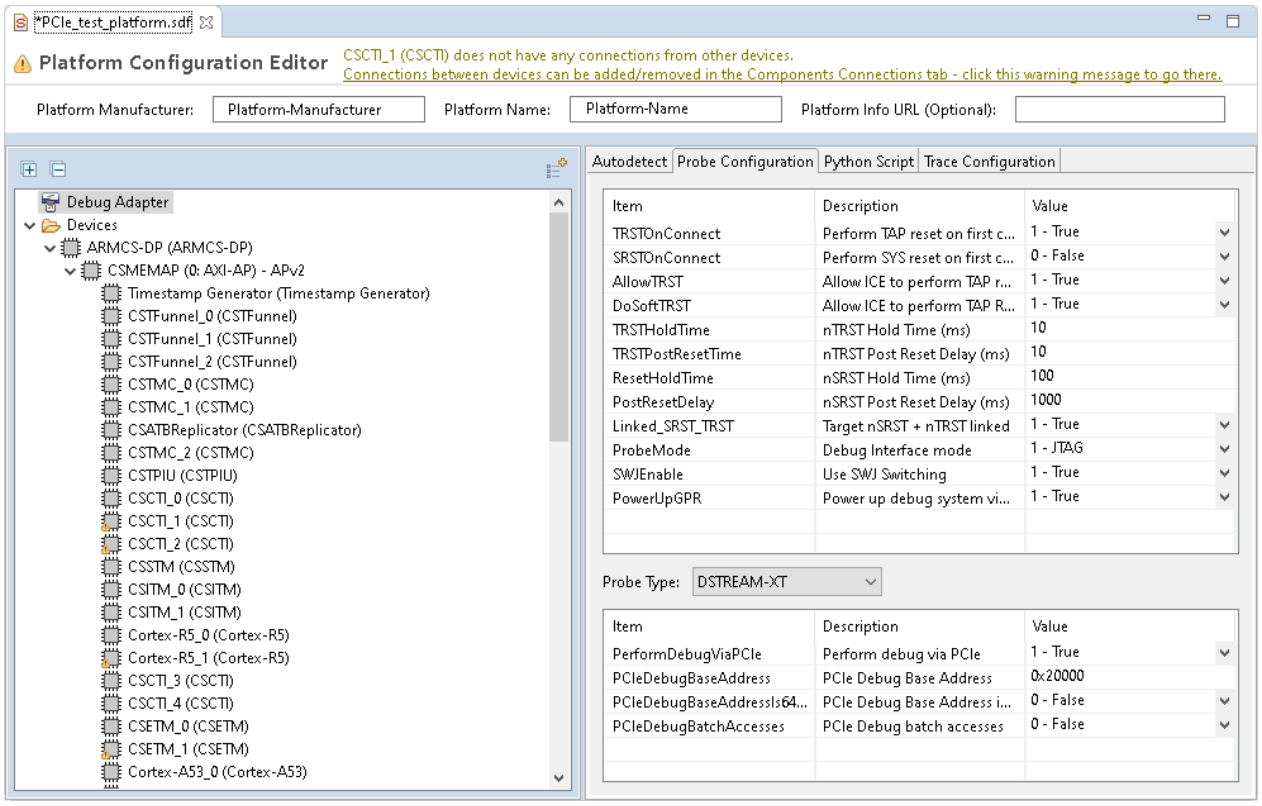
10.3.2 DSTREAM-XT debug configuration

You can perform debug using PCIe with the DSTREAM-XT system.

Configure the DSTREAM-XT debug settings using the debug configuration settings available in the **Platform Configuration Editor** (PCE).

In PCE, select the configuration file, and then click **Debug Adapter** > **Probe Configuration** > **Probe Type** > **DSTREAM-XT**.

Figure 10-16: Probe Configuration tab



*PCIe_test_platform.sdf

Platform Configuration Editor

CSCTI_1 (CSCTI) does not have any connections from other devices.
Connections between devices can be added/removed in the Components Connections tab - click this warning message to go there.

Platform Manufacturer: Platform-Manufacturer
Platform Name: Platform-Name
Platform Info URL (Optional):

- Debug Adapter
- Devices
  - ARMCS-DP (ARMCS-DP)
    - CSMEMAP (0: AXI-AP) - APv2
      - Timestamp Generator (Timestamp Generator)
      - CSTFunnel_0 (CSTFunnel)
      - CSTFunnel_1 (CSTFunnel)
      - CSTFunnel_2 (CSTFunnel)
      - CSTMC_0 (CSTMC)
      - CSTMC_1 (CSTMC)
      - CSATBReplicator (CSATBReplicator)
      - CSTMC_2 (CSTMC)
      - CSTPIU (CSTPIU)
      - CSCTI_0 (CSCTI)
      - CSCTI_1 (CSCTI)
      - CSCTI_2 (CSCTI)
      - CSSTM (CSSTM)
      - CSITM_0 (CSITM)
      - CSITM_1 (CSITM)
      - Cortex-R5_0 (Cortex-R5)
      - Cortex-R5_1 (Cortex-R5)
      - CSCTI_3 (CSCTI)
      - CSCTI_4 (CSCTI)
      - CSETM_0 (CSETM)
      - CSETM_1 (CSETM)
      - Cortex-A53_0 (Cortex-A53)

Autodetect | Probe Configuration | Python Script | Trace Configuration

Item	Description	Value
TRSTOnConnect	Perform TAP reset on first c...	1 - True
SRSTOnConnect	Perform SYS reset on first c...	0 - False
AllowTRST	Allow ICE to perform TAP r...	1 - True
DoSoftTRST	Allow ICE to perform TAP R...	1 - True
TRSTHoldTime	nTRST Hold Time (ms)	10
TRSTPostResetTime	nTRST Post Reset Delay (ms)	10
ResetHoldTime	nSRST Hold Time (ms)	100
PostResetDelay	nSRST Post Reset Delay (ms)	1000
Linked_SRST_TRST	Target nSRST + nTRST linked	1 - True
ProbeMode	Debug Interface mode	1 - JTAG
SWJEnable	Use SWJ Switching	1 - True
PowerUpGPR	Power up debug system vi...	1 - True

Probe Type: DSTREAM-XT

Item	Description	Value
PerformDebugViaPCIe	Perform debug via PCIe	1 - True
PCIeDebugBaseAddress	PCIe Debug Base Address	0x20000
PCIeDebugBaseAddressIs64...	PCIe Debug Base Address i...	0 - False
PCIeDebugBatchAccesses	PCIe Debug batch accesses	0 - False

Table 10-2: PCIe debug configuration options

Configuration item name	Description	Value
`PerformDebugVIAPCIe`	Perform debug using PCIe or using JTAG/ SWD.	If using PCIe to debug, select `1-True`. If using JTAG/SWD to debug, select `0-False`.
`PCIeDebugBaseAddress`	The base address of the PCIe memory access.	The base memory address. For example, `0x20000`
`PCIeDebugBaseAddressIs64Bit`	Select if the PCIe debug address is 32-bit or 64-bit	If 32-bit, select `0-False`. If 64-bit, select `1-True`.

Configuration item name	Description	Value
`PCIeDebugBatchAccesses`	Select if the PCIe debug read and write instructions to consecutive addresses are batched together. This improves performance. To use this feature, your target must support read and write transactions with length greater than one `dword`.	If the read and write must be batched, select `1-True`, else select `0-False`.

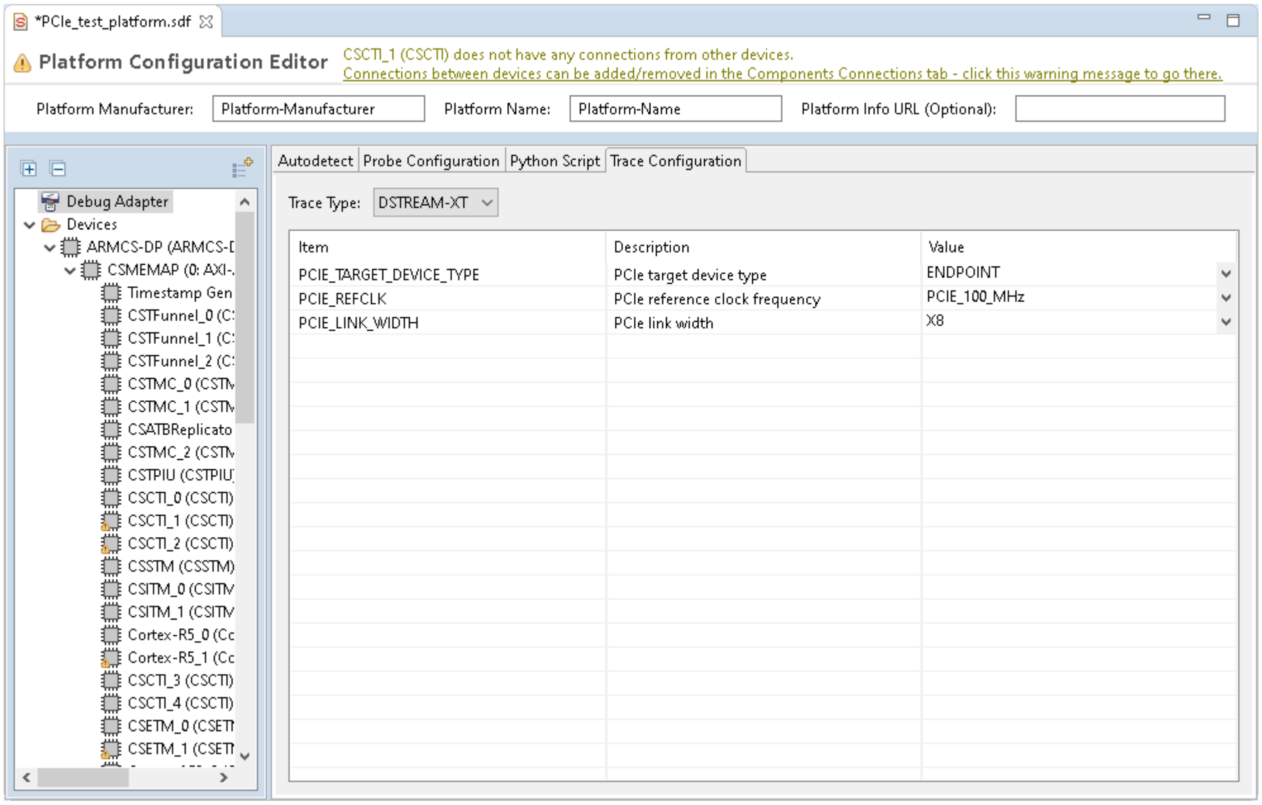
10.3.3 DSTREAM-XT trace configuration

As part of configuring trace for your hardware, you can configure the trace over PCIe interface.

Configure the DSTREAM-XT trace settings using the **Trace Configuration** settings available in the **Platform Configuration Editor** (PCE).

In PCE, select the configuration file, and then click **Debug Adapter** > **Trace Configuration** > **Trace Type** > **DSTREAM-XT**

Figure 10-17: Trace Configuration tab



*PCIe_test_platform.sdf

Platform Configuration Editor

CSCTI_1 (CSCTI) does not have any connections from other devices.
Connections between devices can be added/removed in the Components Connections tab - click this warning message to go there.

Platform Manufacturer: Platform-Manufacturer Platform Name: Platform-Name Platform Info URL (Optional):

Debug Adapter
Devices
ARMCS-DP (ARMCS-D
CSMEMAP (0: AXI-
Timestamp Gen
CSTFunnel_0 (C
CSTFunnel_1 (C
CSTFunnel_2 (C
CSTMC_0 (CSTM
CSTMC_1 (CSTM
CSATBReplicato
CSTMC_2 (CSTM
CSTPIU (CSTPIU
CSCTI_0 (CSCTI)
CSCTI_1 (CSCTI)
CSCTI_2 (CSCTI)
CSSTM (CSSTM)
CSITM_0 (CSITM
CSITM_1 (CSITM
Cortex-R5_0 (Co
Cortex-R5_1 (Co
CSCTI_3 (CSCTI)
CSCTI_4 (CSCTI)
CSETM_0 (CSETI
CSETM_1 (CSETI

Autodetect | Probe Configuration | Python Script | Trace Configuration

Trace Type: DSTREAM-XT

Item	Description	Value
PCIE_TARGET_DEVICE_TYPE	PCIe target device type	ENDPOINT
PCIE_REFCLK	PCIe reference clock frequency	PCIE_100_MHz
PCIE_LINK_WIDTH	PCIe link width	X8

Table 10-3: PCIe trace configuration options

Configuration item name	Description	Value
`PCIE_TARGET_DEVICE_TYPE`	The PCIe target device type.	`ENDPOINT/ROOT_PORT`

Configuration item name	Description	Value
`PCIE_REFCLK`	PCIe reference clock frequency required by the target.	`PCIE_100_MHz/PCIE_125_MHz/PCIE_250_MHz`
`PCIE_LINK_WIDTH`	PCIe link width of the target.	`x1/x2/x4/x8`

10.3.4 Create debug configuration and connect to the target

Before you begin

- Ensure that your target is available in the configuration database.
- Ensure that your target is connected correctly to the DSTREAM-XT unit.
- Ensure that your target is powered on. Refer to the documentation supplied with the target for more information.
- Ensure that the debug hardware probe connecting your target to your workstation is powered on and working.

About this task

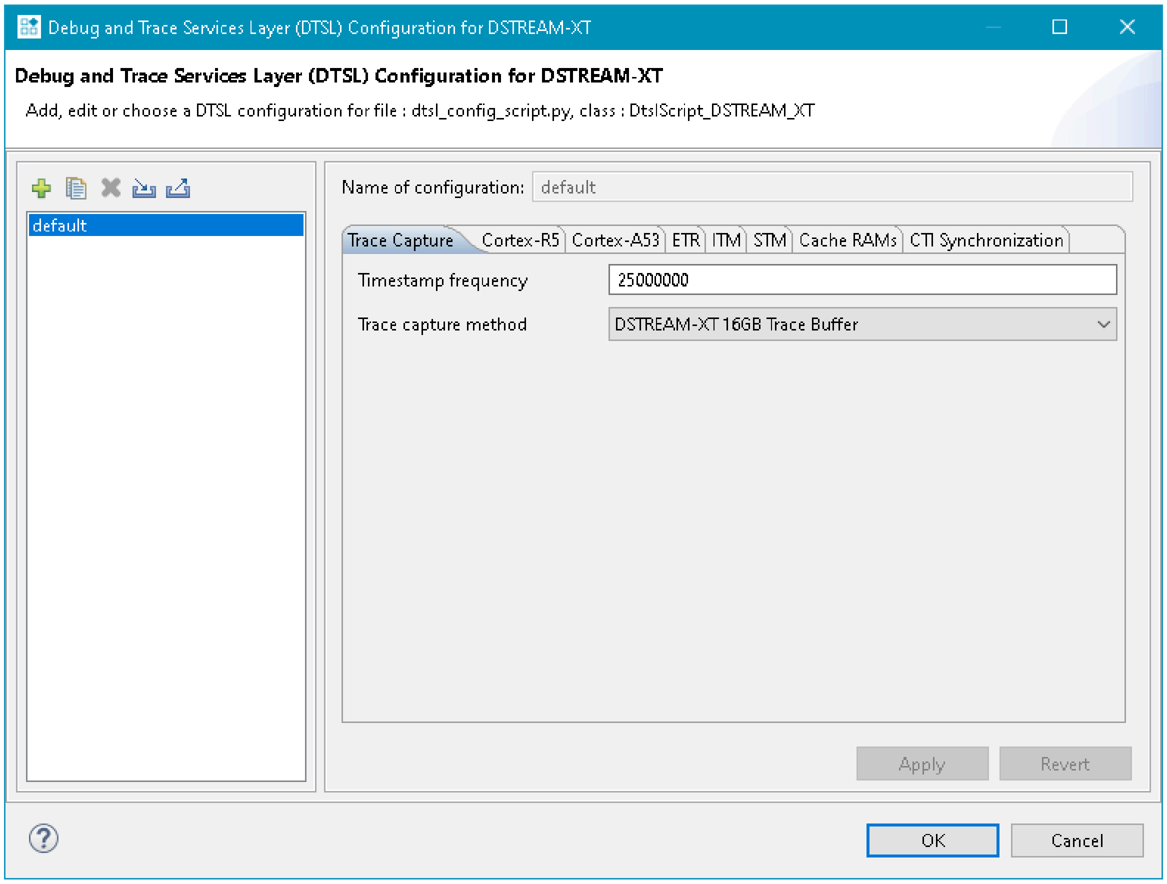
This task describes the workflow for the *JTAG/SWD debug with PCIe trace* use-case.

After creating the platform configuration for your target, create a debug hardware connection and connect to your hardware.

Procedure

1. From the Arm® Development Studio main menu, select **File** > **New** > **Hardware Connection**.
2. In the **Hardware Connection** dialog box, specify the details of the connection:
   a) In **Debug Connection** give the debug connection a name, for example **my_dstream_xt_connection** and click **Next**.
   b) In **Target Selection** select your **<Platform manufacturer>** > **<Platform name>** and click **Finish**. This completes the initial connection configuration and opens the **Edit Configuration** dialog.
3. To specify the target and connection settings, in the **Edit Configuration** dialog box:
   a) Select the **Connection** tab.
   b) In the **Select target** panel, confirm the target that is selected.
   c) In the **Target Connection** list, select **DSTREAM Family**.
   d) In the **Connections** area, enter the **Connection** name or IP address of your debug hardware adapter. If your connection is local, click **Browse** and select the connection using the **Connection Browser**.
   e) In **DSTL Options**, click **Edit** to display the **Debug and Trace Services Layer (DTSL) Configuration for DSTREAM-XT** dialog.
   f) In the **Trace Capture** tab, set the **Trace capture method** as **DSTREAM-XT 16GB Trace Buffer**

Figure 10-18: Edit the DTSL settings


Debug and Trace Services Layer (DTSL) Configuration for DSTREAM-XT

Debug and Trace Services Layer (DTSL) Configuration for DSTREAM-XT

Add, edit or choose a DTSL configuration for file : dtsl_config_script.py, class : DtslScript_DSTREAM_XT

Name of configuration: default

Timestamp frequency: 25000000

Trace capture method: DSTREAM-XT 16GB Trace Buffer

g) In the **ETR** tab, enable the **Configure the system memory trace buffer to be used by the CSTMC_2/ETR device** checkbox, and then specify the **Start address** and **Size in bytes**.
h) Select any other trace settings you require.
i) Click **OK** to close the dialog box and return to the **Edit Configuration** dialog box.

4. Click the **Files** tab to specify your application and additional resources to download to the target:
   a) To load your application on the target at connection time, in the **Target Configuration** area, specify your application in the **Application on host to download** field.
   b) To debug your application at source level, select **Load symbols**.
   c) To load additional resources, for example, additional symbols or peripheral description files from a directory, add them in the **Files** area. Click **+** to add resources, click **-** to remove resources.
5. Use the **Debugger** tab to configure debugger settings.
   a) In the **Run control** area:
      - Specify if you want to **Connect only** to the target or **Debug from entry point**. If you want to start debugging from a specific symbol, select **Debug from symbol**.
      - To run target or debugger initialization scripts, select the relevant options and specify the script paths.
      - To execute additional debugger commands after connecting, select **Execute debugger commands** options. Specify the commands on separate lines in the text entry field.

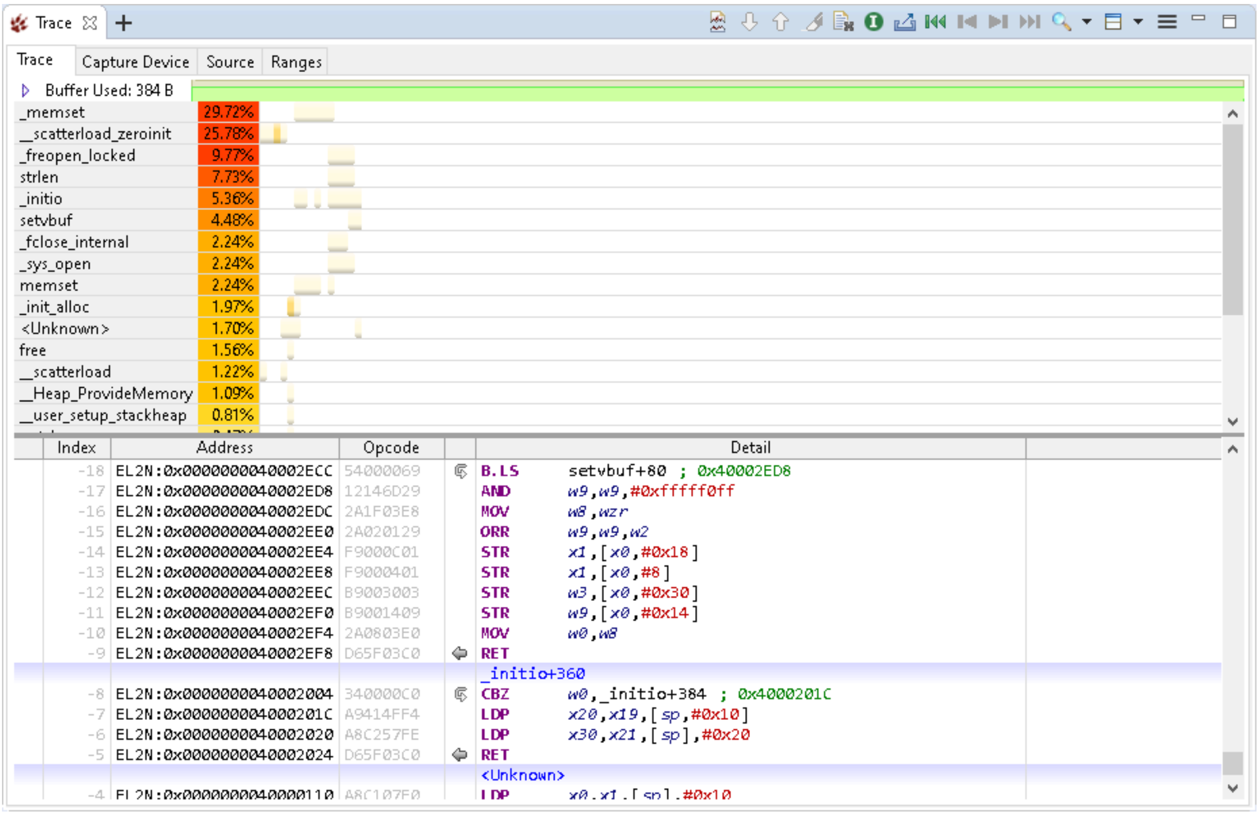
   b) The debugger uses your workspace as the default working directory on the host. To change the default location, deselect the **Use default** option under **Host working directory** and specify the new location.
   c) In the **Paths** area, specify any directories on the host that contain your application files in the **Source search directory** field.
   d) To use additional resources, click **+** to add resources. Click **-** to remove resources.

6. (Optional) Use the **Arguments** tab to enter arguments that are passed to the `main()` function of the application when the debug session starts. The debugger uses semihosting to pass arguments to `main()`.
7. (Optional) Use the **Environment** tab to create and configure environment variables to pass into the launch configuration when it is executed.
8. Click **Apply** and then **Debug** to connect to the target and start debugging session.

Results

The trace is now setup for the target. You can view the trace output in the **Trace** view.

Figure 10-19: Trace view output



Buffer Used: 384 B

_memset	29.72%
__scatterload_zeroinit	25.78%
_freopen_locked	9.77%
strlen	7.73%
_initio	5.36%
setvbuf	4.48%
_fclose_internal	2.24%
_sys_open	2.24%
memset	2.24%
_init_alloc	1.97%
<Unknown>	1.70%
free	1.56%
__scatterload	1.22%
__Heap_ProvideMemory	1.09%
__user_setup_stackheap	0.81%

Index	Address	Opcode	Detail
-18	EL2N:0x0000000040002ECC	54000069	B.LS setvbuf+80 ; 0x40002ED8
-17	EL2N:0x0000000040002ED8	12146D29	AND w9,w9,#0xfffff0ff
-16	EL2N:0x0000000040002EDC	2A1F03E8	MOV w8,wzr
-15	EL2N:0x0000000040002EE0	2A020129	ORR w9,w9,w2
-14	EL2N:0x0000000040002EE4	F9000C01	STR x1,[x0,#0x18]
-13	EL2N:0x0000000040002EE8	F9000401	STR x1,[x0,#8]
-12	EL2N:0x0000000040002EEC	B9003003	STR w3,[x0,#0x30]
-11	EL2N:0x0000000040002EF0	B9001409	STR w9,[x0,#0x14]
-10	EL2N:0x0000000040002EF4	2A0803E0	MOV w0,w8
-9	EL2N:0x0000000040002EF8	D65F03C0	RET
			_initio+360
-8	EL2N:0x0000000040002004	340000C0	CBZ w0,_initio+384 ; 0x4000201C
-7	EL2N:0x000000004000201C	A9414FF4	LDP x20,x19,[sp,#0x10]
-6	EL2N:0x0000000040002020	A8C257FE	LDP x30,x21,[sp],#0x20
-5	EL2N:0x0000000040002024	D65F03C0	RET
			<Unknown>
-4	EL2N:0x0000000040000110	A8C107F0	LDP x0,x1,[sp],#0x10

10.4 Debug Hardware configuration

This section describes how to configure your debug hardware unit in Development Studio.

Note

For information about the Debug Hardware Firmware Installer and Debug Hardware Configure IP views, see Debug Hardware Firmware Installer view and Debug Hardware Configure IP view.

10.4.1 Arm Debug and Trace Architecture

Modern Arm processors consist of several debug and trace components. Here, we describe what some of the main components do and how they connect to each other.

Debug units, such as DSTREAM, use JTAG to connect directly to physical devices. To detect the devices, the debug unit sends clock signals around the JTAG scan chain, which pass through all the physical devices in sequence. The physical devices on the JTAG scan chain can include:

- Legacy Arm processors, such as Arm7™, Arm9™, and Arm11™ processors.
- Arm® CoreSight™ Debug Access Port (DAP).
- Custom devices that are not based on Arm.

Arm® Cortex® processors and CoreSight devices are not located directly on the JTAG scan chain. Instead, Arm CoreSight Debug Access Ports provide access to CoreSight Memory Access Ports. These Memory Access Ports provide access to additional JTAG devices or memory-mapped virtual devices, such as Arm Cortex processors and CoreSight devices. Each virtual device provides memory-mapped registers that a debugger can use to control and configure the device, or to read information from it.

There are different types of CoreSight devices.

- Embedded Trace Macrocell (ETM) and Program Flow Trace Macrocell (PTM) are trace sources. They attach directly to a Cortex processor and non-invasively generate information about the operations performed by the processor. Each Cortex processor has a revision of ETM or PTM that has specific functionality for that processor.
- Instrumentation Trace Macrocell (ITM) and System Trace Macrocell (STM) are trace sources. They generate trace information about software and hardware events occurring across the System-on-Chip.
- Embedded Trace Buffer (ETB), Trace Memory Controller (TMC), and Trace Port Interface Unit (TPIU) are trace sinks. They receive trace information generated by the trace sources. Trace sinks either store the trace information or route it to a physical trace port.
- Funnels and Replicators are trace links. They route trace information from trace sources to trace sinks.
- Cross Trigger Interface (CTI) devices route events between other devices. The CTI network has a variety of uses, including:

  - Halt processors when trace buffers become full.
  - Route trace triggers.
  - Ensure tight synchronization between processors. For example, when one processor in a cluster halts, the other processors in the cluster halt with minimal latency.

A debugger needs the details of the physical JTAG scan chain, and the details of individual Cortex processors and CoreSight devices, including:

- CoreSight AP index.
- ROM table base address.
- Device type.
- Revision.
- Implementation detail.

However, a debugger also needs to know how the devices are connected to each other. For example, which processors are part of the same cluster, how the CTI network can pass event information between devices, and the topology of the trace subsystem. Without this information, a debugger might not be able to provide all of the control and configuration services that are available. To provide this information to Development Studio, use the device hierarchy in the Platform Configuration Editor (PCE). For more information, see Device hierarchy in the PCE view and Device configuration panel. For instructions on configuring your debug hardware unit, see Configure your debug hardware unit for Platform Autodetection.

Related information

Device hierarchy in the PCE view on page 190
Device configuration panel on page 214
Configure your debug hardware unit for Platform Autodetection on page 270
Overview: Arm CoreSight debug and trace components

10.4.2 Hardware configurations created by the PCE

The Development Studio Platform Configuration Editor (PCE) can create DTSL configurations from system description files. You can create DTSL configurations manually in the PCE, or through the wizard-based autodetection workflow.

Warning

After autodetecting your platform, check the system description within the PCE, and add any missing information. For more information on platform configuration, see Platform Configuration in Development Studio.

When you build a platform, three files are created as part of the debug configuration:

system_description.sdf

The system description file has the same name as the platform described in its contents. For example, a platform with the name Juno has an `*.sdf` file named `Juno.sdf`.

This file contains:

- Information about the devices present within the system.
- Details about the version and specific configuration of these devices.
- The topology information which describes their interconnections.
- Debug probe configuration settings.
- The information that is required to create and configure the DTSL configuration objects that are used to debug the target.

Warning

Incomplete information within the `*.sdf` file might result in failing debug connections or trace connections failing to produce valid trace. For example, missing or incomplete information about ATB trace topology, CTI trigger topology, or SMP connections not using CTI synchronization.

project_types.xml

Contains details about the debug activities that are available to Development Studio when connecting to the target. For more information about this file, see About project_types.xml.

dtsl_config_script.py

This is the DTSL Jython script file. It is responsible for the instantiation of the configuration object and all associated devices. The file is created using the information directly from the `*.sdf` file.

This Jython file enables you to add user-defined types, modify default trace and debug options, and provide target-specific initialization, where required. For example, register writes to power up the debug subsystem.

For more information about the structure of the DTSL Jython configuration file, see DTSL Jython configuration file structure.

For more information about the DTSL configuration execution flow, see DTSL configuration execution flow.

Related information

Platform Configuration in Development Studio on page 184
DTSL Jython configuration file structure on page 281
DTSL configuration execution flow on page 287

10.4.3 Configure your debug hardware unit for Platform Autodetection

To automatically detect the correct configuration for your platform, you must correctly configure your debug hardware unit.

Before you begin

- You must know the target platform.
- You need the configuration information for your debug hardware unit, for example:
  - Probe connection address (the connection address (TCP/USB) of the debug hardware unit).
  - Clock speed (the JTAG clock speed of the target).
  - The reset hold and delay times (hold: time in milliseconds the target is held in a hardware reset state, delay: time in ms that the target waits after reset is released, before attempting any other debugging operations).
  - Drive strengths (to set the debug signals strengths from the debug hardware unit into the target).
  - Reset behavior during autodetection (controls whether a system reset can take place during the autodetection process, for example, on some targets that have a board with a tap controller a system reset may cause autodetection to fail).
  - Cable pin configurations (some boards can have the CoreSight 20 connector pins laid out differently, changing this option allows the debugger and target to communicate properly).

  These configuration settings must be correct for your specific target platform.

  Note

  For most cases, you can use the default settings.

- If you are using a third-party debug probe, add it using the instructions in Add a third-party debug probe.
- You need an existing platform configuration to edit the debug hardware unit autodetection configuration.

About this task

In the **Autodetect** tab of the Platform Configuration Editor (PCE), you can configure your debug hardware unit and the settings that are used for autodetection. For example, to use JTAG or Serial Wire Debug (SWD), and the clock speed.

Procedure

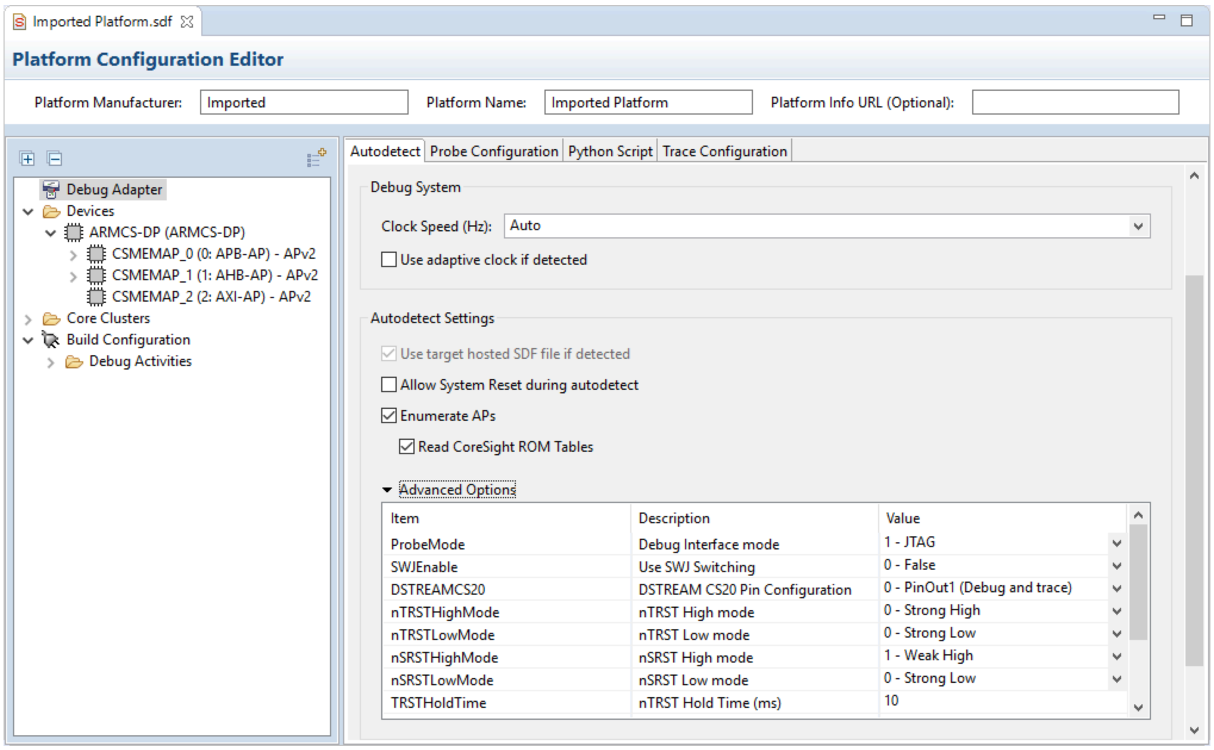
1. Navigate to the device hierarchy in the PCE. Double-click an existing `*.sdf` file.

Note

For more information about the device hierarchy, see the Device hierarchy in the PCE view topic.

2. Select **Debug Adapter** and click the **Autodetect** tab.

Figure 10-20: Autodetect settings



Imported Platform.sdf

Platform Configuration Editor

Platform Manufacturer:	Imported
Platform Name:	Imported Platform
Platform Info URL (Optional):	

- Debug Adapter
- Devices
  - ARMCS-DP (ARMCS-DP)
    - CSMEMAP_0 (0: APB-AP) - APv2
    - CSMEMAP_1 (1: AHB-AP) - APv2
    - CSMEMAP_2 (2: AXI-AP) - APv2
- Core Clusters
- Build Configuration
  - Debug Activities

Autodetect | Probe Configuration | Python Script | Trace Configuration

Debug System

Clock Speed (Hz): Auto

☐ Use adaptive clock if detected

Autodetect Settings

☑ Use target hosted SDF file if detected

☐ Allow System Reset during autodetect

☑ Enumerate APs

☑ Read CoreSight ROM Tables

Advanced Options

Item	Description	Value
ProbeMode	Debug Interface mode	1 - JTAG
SWJEnable	Use SWJ Switching	0 - False
DSTREAMCS20	DSTREAM CS20 Pin Configuration	0 - PinOut1 (Debug and trace)
nTRSTHighMode	nTRST High mode	0 - Strong High
nTRSTLowMode	nTRST Low mode	0 - Strong Low
nSRSTHighMode	nSRST High mode	1 - Weak High
nSRSTLowMode	nSRST Low mode	0 - Strong Low
TRSTHoldTime	nTRST Hold Time (ms)	10

Note

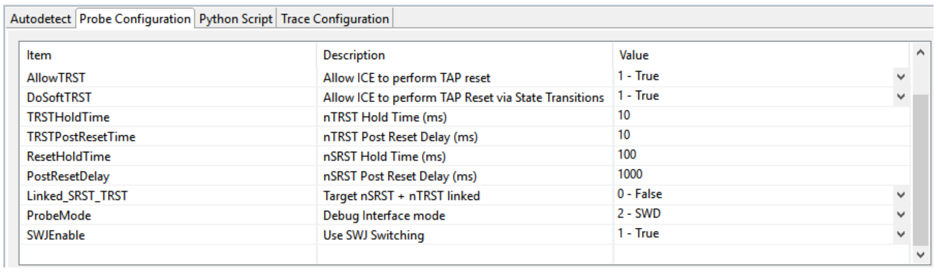
For more information about the Debug Adapter view, see the Debug Adapter configuration in the PCE topic.

3. Set your probe **Connection Address**:
   a) Choose a debug adapter type from the drop-down.
   b) Provide a probe connection address, either:
      - Manually enter a probe connection address.
      - Click **Browse** and select a probe connection in the **Connection Browser**.
4. Expand **Advanced Options** and configure your debug hardware unit.
   For more information about the advanced options, see the Debug Adapter configuration in the PCE topic.

Note

- If you use a JTAG connection, you must set the JTAG type and clock speed. Other important autodetection options are:
    - The reset hold and delay times.
    - Drive strengths.
    - Reset behavior during autodetection.
    - Cable pin configurations.
- If your target supports both JTAG and SWD, you must also enable `Use SWJ Switching`. Configure this setting, and other configuration options, in the **Probe Configurations** tab.
- To configure the probe for an SWD connection, select the **Probe Configuration** tab, change the **ProbeMode** item to `SWD`, and change the **SWJEnable** item to `True`.

Figure 10-21: SWD connection



Item	Description	Value
AllowTRST	Allow ICE to perform TAP reset	1 - True
DoSoftTRST	Allow ICE to perform TAP Reset via State Transitions	1 - True
TRSTHoldTime	nTRST Hold Time (ms)	10
TRSTPostResetTime	nTRST Post Reset Delay (ms)	10
ResetHoldTime	nSRST Hold Time (ms)	100
PostResetDelay	nSRST Post Reset Delay (ms)	1000
Linked_SRST_TRST	Target nSRST + nTRST linked	0 - False
ProbeMode	Debug Interface mode	2 - SWD
SWJEnable	Use SWJ Switching	1 - True

5. Check that the configuration information for your debug hardware unit is correct in each of the Probe Configuration, Python Script, and Trace Configuration tabs.

Note

Descriptions of the configuration options in the **Debug Adapter** view are available in the Debug Adapter configuration in the PCE topic.

6. Click **File** > **Save** and save any configuration changes.

Next steps

Next, click **Autodetect Platform**. The debug hardware unit interrogates the scan chain at the current clock speed. If the clock speed is too high, some devices on the scan chain might not be detected. If you suspect that some devices on the scan chain are not being detected, decrease the clock speed.

When platform autodetection has finished, review the detected configuration information and add any missing topology links. Finally, rebuild the platform configuration.

For more information about creating and connecting to hardware platforms, see Hardware platform bring-up in Development Studio.

Related information

Add a third-party debug probe on page 274
Debug Adapter configuration in the PCE on page 290
Device hierarchy in the PCE view on page 190

10.4.4 Third-party Debug Probe API

Use the Debug Probe API package to add your debug probe to Arm® Debugger.

The Debug Probe API package is a working but non-optimized solution for you to modify as required. It contains:

- A stubbed third-party probe solution that implements RDDI APIs, including the 'RDDI 3rd Party' API. You can use this API as a starting point when integrating your probe with Arm Development Studio.
- A reference CMSIS-DAP example is included, for demonstration purposes.

The API package is located in `<installation_directory>/sw/debugger/DebugProbeAPI/`.

Detailed instructions are available in the `README_RDDI3rdParty.txt` file, located in the DebugProbeAPI directory.

After you have modified the package, you can then add your probe to Arm Debugger.

Related information

Add a third-party debug probe on page 274

10.4.5 Add a third-party debug probe

Create a probe configuration database entry, and then use the Platform Configuration Editor (PCE) to add your third-party debug probe to Arm® Development Studio.

Before you begin

You need one of the following, provided by the probe vendor:

- Implementation files:
    - A probe definition file. This file is an XML file that defines your probe name and the RDDI library file. It might also contain `config_items` and `capabilities`.
    - An RDDI library file. The probe vendor might provide both a Windows and a Linux variant of the file:
        - Windows: `rddi_example_2.dll`.
        - Linux: `librddi_example.so.2`.

Note

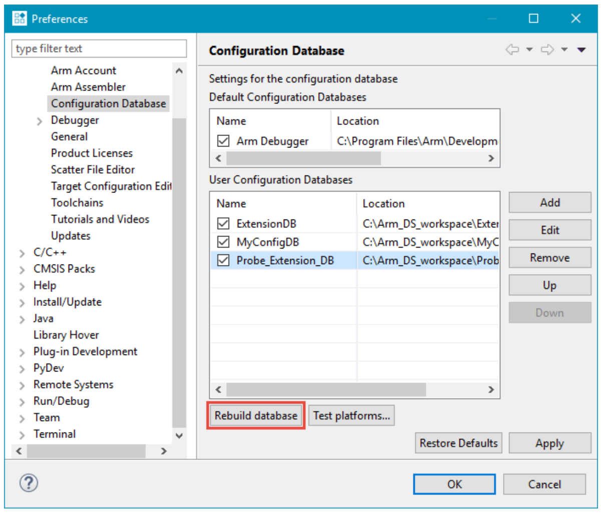
To create your own third-party probe implementation files, see Third-party Debug Probe API.

- A configuration database that contains these implementation files.

Procedure

1. Set up your configuration database using the files provided by the probe vendor. The method for doing this varies, depending on your scenario.

Table 10-4: Methods for setting up your configuration database

You have the implementation files, but no configuration database:	You have an existing configuration database:
a. (Optional) Create a configuration database: **Note:** This step is optional. If you want to edit an existing configuration database to add a third-party debug probe, you can jump to the next step. 1. In the **Project Explorer** view, right-click and select **New** > **Configuration Database**. 2. Enter a name for your database, and click **FINISH**. Result: A configuration database directory is added to your workspace. It contains an empty directory called `Boards`. b. To store third-party probes, edit the configuration database: 1. Navigate to your Development Studio workspace in your file system. Locate and open the configuration database to edit. 2. Create a `Probes` directory at the same level as the `Boards` directory. 3. Open the `Probes` directory and add the probe definition and the library files. 4. Rebuild your database. Open Development Studio and select **Window** > **Preferences** > **Arm DS** > **Configuration Database**, and click **Rebuild database**. **Warning:** This rebuilds all of your databases. **Figure 10-22: How to manually rebuild your database** 	Import the database: a. Right-click in the **Project Explorer**, and select **Import...** > **General** > **Existing Projects into Workspace**, and click **Next**. b. **Browse** to the file and click **OK** to select. c. Select the **Copy projects into workspace** checkbox, and click **Finish**. d. If your workspace is not configured to automatically build projects, you must add the new database in your settings: 1. Select **Window -> Preferences** to open the **Preferences** dialog box. Select **Arm DS -> Configuration Database**. 2. Add your newly imported database to **User Configuration Databases** and click **Rebuild database**. Result: The database is imported into your workspace and gets automatically rebuilt.

Configuration Database

Settings for the configuration database

Default Configuration Databases

Name	Location
☑ Arm Debugger	C:\Program Files\Arm\Developm

User Configuration Databases

Name	Location
☑ ExtensionDB	C:\Arm_DS_workspace\Exter
☑ MyConfigDB	C:\Arm_DS_workspace\MyC
☑ Probe_Extension_DB	C:\Arm_DS_workspace\Prob

Note

- Rebuild your database every time you change the probe definition file.
- To get the probe to appear in the PCE, close and re-open any open `*.sdf` files.

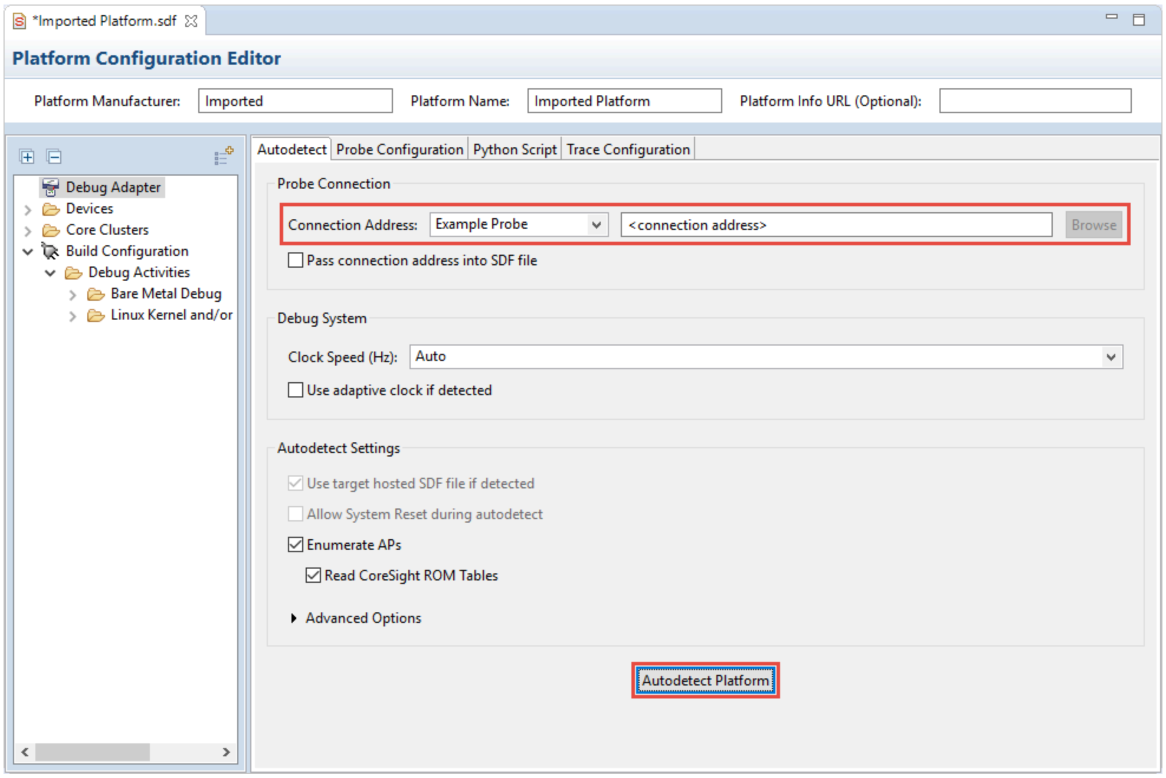
2. Detect the platform using the probe:

Warning

If you need to make any changes to the probe configuration values before autodetecting your target, use manual platform configuration to connect to your target. For more information on manual platform configuration, see Manual platform configuration. Follow the manual configuration instructions and make any probe configuration changes before you add or edit the existing topology for your platform.

a) In the device hierarchy list, select **Debug Adapter** and then select the **Autodetect** tab.
b) Add the connection address for your probe. In the **Connection Address** field, select your debug adapter type, and click **Browse** to display the **Connection Browser** dialog box. Select your debug adapter.
c) To configure the settings for the selected probe, click **Autodetect Platform**.

Figure 10-23: Autodetect your platform using the PCE



Platform Manufacturer: Imported
Platform Name: Imported Platform
Platform Info URL (Optional):

Probe Connection
Connection Address: Example Probe <connection address>
☐ Pass connection address into SDF file

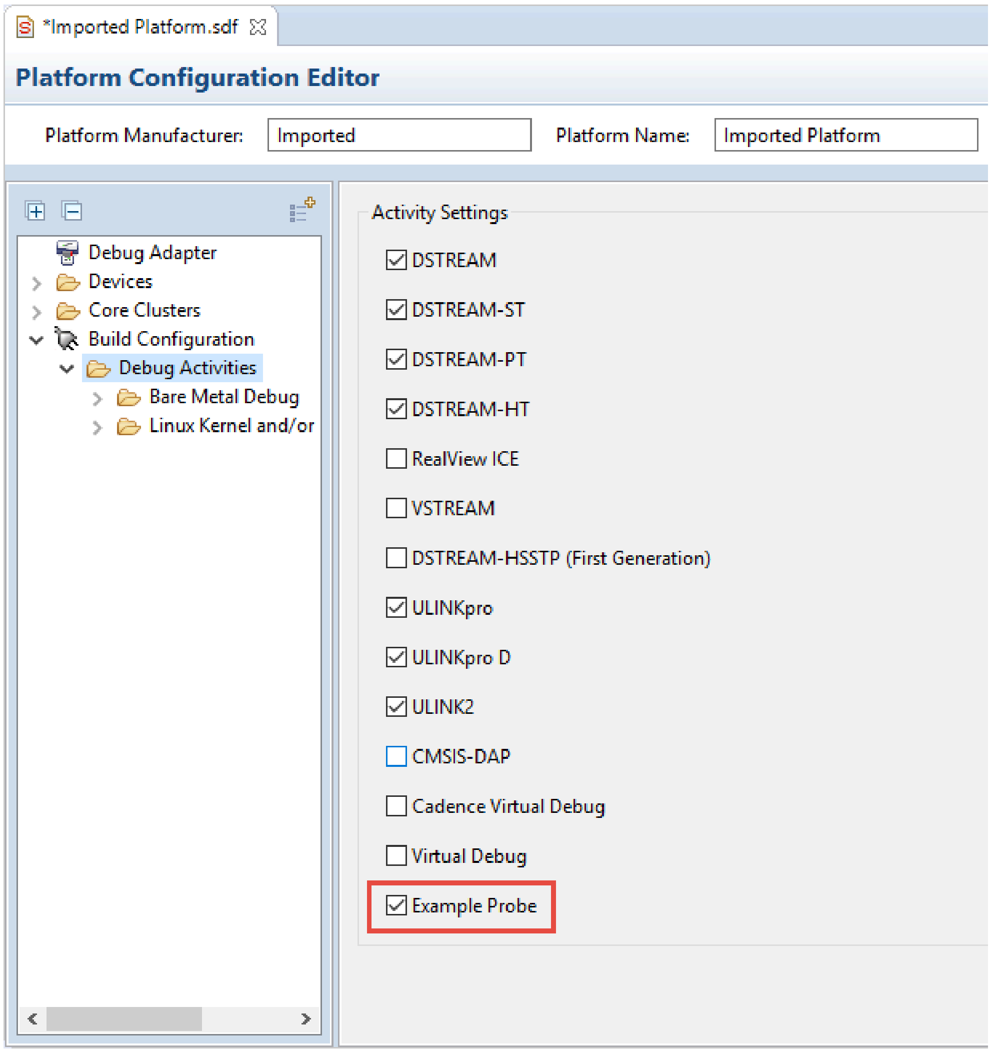
Debug System
Clock Speed (Hz): Auto
☐ Use adaptive clock if detected

Autodetect Settings
☑ Use target hosted SDF file if detected
☐ Allow System Reset during autodetect
☑ Enumerate APs
☑ Read CoreSight ROM Tables

If your platform is detected correctly, an `Autodetection Complete` message is displayed in the **Console** view.

3. Open the activity settings for your platform:
   a) In the device hierarchy panel, select the **Debug Activities** directory or select any of the child nodes in the **Debug Activities** directory.
   The **Activity Settings** panel opens.

Figure 10-24: Specify debug activities


*Imported Platform.sdf

Platform Configuration Editor

Platform Manufacturer: Imported
Platform Name: Imported Platform

Activity Settings

- [x] DSTREAM
- [x] DSTREAM-ST
- [x] DSTREAM-PT
- [x] DSTREAM-HT
- [ ] RealView ICE
- [ ] VSTREAM
- [ ] DSTREAM-HSSTP (First Generation)
- [x] ULINKpro
- [x] ULINKpro D
- [x] ULINK2
- [ ] CMSIS-DAP
- [ ] Cadence Virtual Debug
- [ ] Virtual Debug
- [x] Example Probe

   b) Check that your new probe is selected. If not, select it, and it saves automatically.
4. Rebuild your platform. The PCE enables your probe in the platform configuration.

Related information

Third-party Debug Probe API on page 274
Configure your debug hardware unit for Platform Autodetection on page 270
Debug Adapter configuration in the PCE on page 290

10.4.6 Add a debug connection over functional I/O

Describes how to add a virtual debug connection that utilizes existing functional interfaces, such as USB, ethernet, and PCIe.

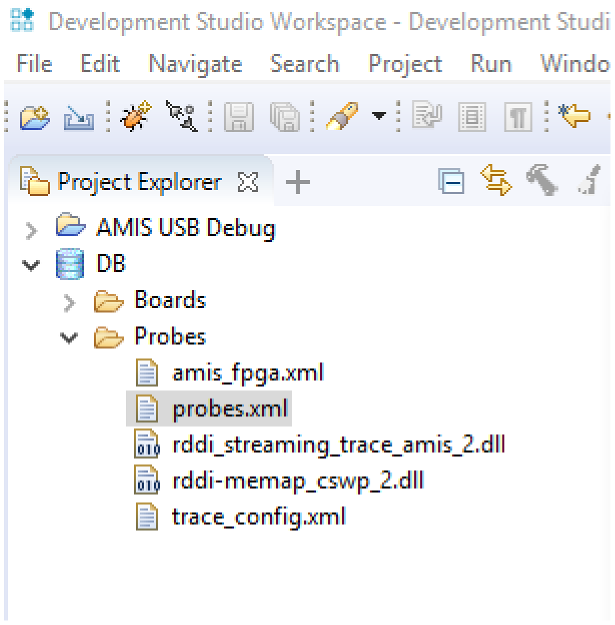
Before you begin

- Make sure that you have all of the software components that are listed in Debug and trace over functional I/O, and that they are loaded as follows:
  - The debug agent must be running on your target.
  - The functional interface drivers must be installed on your machine.
  - You need a directory that is called `Probes`, that contains your probe definition file (`probes.xml`), the library files that implement the debug API and CSWP protocol, and other configuration files if needed. All of these files are provided by your SoC vendor.

Procedure

1. In Arm® Development Studio IDE, create a new **Configuration Database** connection:
   a) **File** > **New** > **Other** > **Configuration Database** > **Configuration Database**.
   b) Enter a name, and click **Finish**.

   Result: The new database is listed in the **Project Explorer**.
2. Copy the configuration and library files into your new database:
   a) Open the parent directory of the `Probes` directory.
   b) Drag and drop the `Probes` directory into your new **Configuration Database**.


Development Studio Workspace - Development Studi
File Edit Navigate Search Project Run Windo
Project Explorer
AMIS USB Debug
DB
Boards
Probes
amis_fpga.xml
probes.xml
rddi_streaming_trace_amis_2.dll
rddi-memap_cswp_2.dll
trace_config.xml

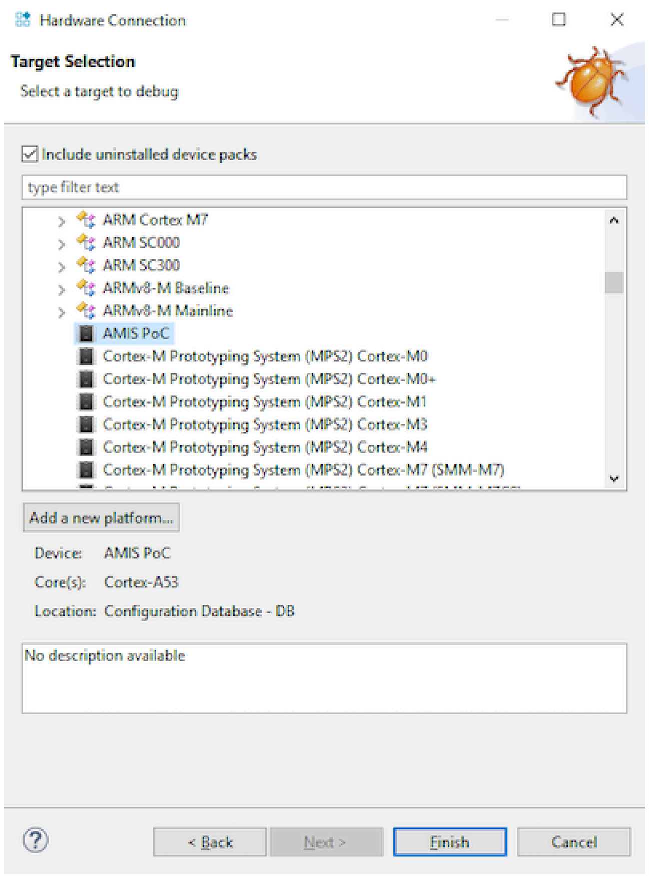
3. Rebuild your database:
   a) Open the **Configuration Database Preferences** dialog box: **Window** > **Preferences** > **Arm DS** > **Configuration Database**.
   b) Select your new database, and click **Rebuild database**.
   c) When it has finished rebuilding, click **Apply** and then **OK** to close the dialog box.

4. Create a new platform configuration:
   a) Open the **Platform Configuration** dialog box: **File** > **New** > **Other** > **Configuration Database** > **Platform Configuration**.
   b) In the **Create Platform Configuration** dialog box, select **Automatic/simple platform detection**. Click **Next**.
   c) In the **Debug Adapter Connection** dialog box, select your virtual probe from the **Connection Type** drop-down, and click **Next**.
   d) Providing there are no connection issues, in the **Summary** dialog box select **Save a Debug and Trace Platform Configuration** and leave the **Debug target after saving configuration** box unchecked. Click **Next**.
   e) In the **Platform Information** dialog box, enter a descriptive **Platform Manufacturer** and **Platform Name**, and then click **Finish**,

   Result: The Platform Configuration Editor opens.
5. Configure your new platform. This is specific to your implementation. For further information, see the documentation for your platform and debug architecture.
   When you have finished, click **Autodetect Platform**.

Results

On completion, your new platform is listed as an available platform, when you create a new debug configuration.


Hardware Connection
Target Selection
Select a target to debug
Include uninstalled device packs
type filter text
ARM Cortex M7
ARM SC000
ARM SC300
ARMv8-M Baseline
ARMv8-M Mainline
AMIS PoC
Cortex-M Prototyping System (MPS2) Cortex-M0
Cortex-M Prototyping System (MPS2) Cortex-M0+
Cortex-M Prototyping System (MPS2) Cortex-M1
Cortex-M Prototyping System (MPS2) Cortex-M3
Cortex-M Prototyping System (MPS2) Cortex-M4
Cortex-M Prototyping System (MPS2) Cortex-M7 (SMM-M7)
Add a new platform...
< Back
Next >
Finish
Cancel

Device: AMIS PoC
Core(s): Cortex-A53
Location: Configuration Database - DB

No description available

Next steps

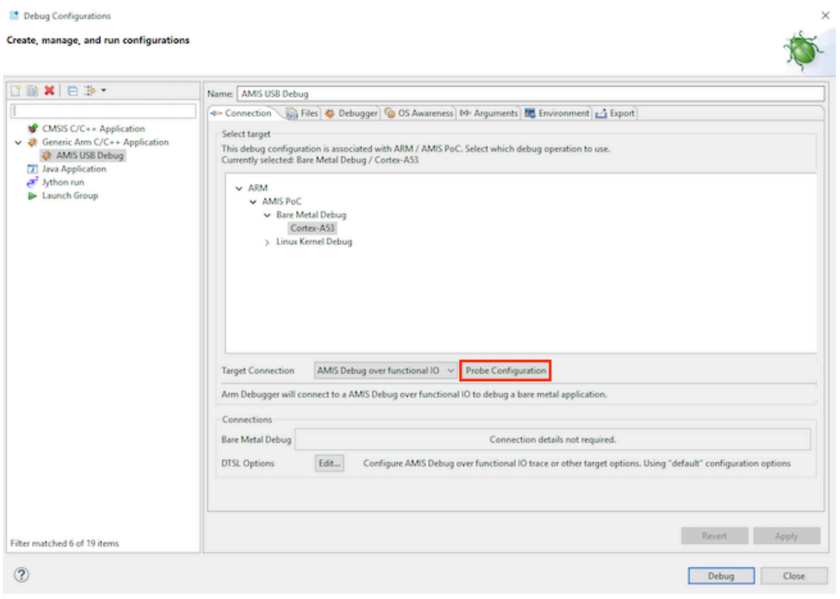
Now you can connect to your new virtual probe when you create a new debug configuration.

Note

`RDDI MEM-AP` virtual probes are not available for CMSIS connections. Only `RDDI-DAP` virtual probes are available for CMSIS connections.

When you create a new debug configuration that uses your new probe, there might be a few differences, including:

- The **Connection Address** field might be disabled if it is not requirement by the implementation. You can enable this in the `probes.xml` file, or using the Arm Development Studio IDE.
- You might need to explicitly tell your target to connect to your virtual probe. You can do this using the **Platform Configuration Editor**.
- If you have configurable connection settings for your virtual probe, you can edit them using the **Probe Configuration** button:



Name: AMIS USB Debug

Select target

This debug configuration is associated with ARM / AMIS PoC. Select which debug operation to use.
Currently selected: Bare Metal Debug / Cortex-A53

Target Connection: AMIS Debug over functional IO

Arm Debugger will connect to a AMIS Debug over functional IO to debug a bare metal application.

Connections

Bare Metal Debug: Connection details not required.

DTSL Options: Configure AMIS Debug over functional IO trace or other target options. Using "default" configuration options

Filter matched 6 of 19 items

Related information

Debug and trace over functional I/O on page 44
Add a third-party debug probe on page 274
Probe Configuration dialog box on page 449

10.4.7 DTSL Jython configuration file structure

The DTSL configuration script (`dtsl_config_script.py` file) is executed when a connection to the target is made, and is responsible for the instantiation and configuration of DTSL objects. These objects, along with the configuration object itself, are used by a debugger to control the debug and trace of the target.

The Jython configuration object can be split into three distinct areas of functionality:

1. DTSL options provision

   Options are provided through a static `getOptionsList()` function. Details of available option types and their use can be found in DTSL options. For some smaller targets, no options are appropriate, and this function returns an empty list. However, user-defined options might still be added.
2. Device and trace capture initialization

   All device and trace source instantiation is performed in the `discoverDevices()` function, and all trace capture devices within the `createTraceCapture()` function.
3. Option call-back response

   The `optionValuesChanged()` function is provided to allow the configuration to respond to changes to DTSL options. See DTSL options for details.

Example of a configuration script

This example uses a trace-enabled big.LITTLE™ configuration. The target consists of two Cortex®-A57 cores, and two Cortex-A53 cores.

```
from com.arm.debug.dtsl.configurations import ConfigurationBaseSDF
from com.arm.debug.dtsl.configurations import DTSLv1
from com.arm.debug.dtsl.components import FormatterMode
from com.arm.debug.dtsl.components import AXIAP
from com.arm.debug.dtsl.components import AHBAP
from com.arm.debug.dtsl.configurations import TimestampInfo
from com.arm.debug.dtsl.components import Device
from com.arm.debug.dtsl.configurations.options import IIntegerOption
from com.arm.debug.dtsl.components import CSTMC
from com.arm.debug.dtsl.components import TMCETBTraceCapture
from com.arm.debug.dtsl.components import DSTREAMSTStoredTraceCapture
from com.arm.debug.dtsl.components import DSTREAMTraceCapture
from com.arm.debug.dtsl.components import CSCTI
from com.arm.debug.dtsl.components import ETMv4TraceSource
from com.arm.debug.dtsl.components import CSTPIU
# The lists below are used within discover devices to initialize core and SMP
 devices
clusterNames = ["Cortex-A53_SMP_0", "Cortex-A57_SMP_0"]
clusterCores = [["Cortex-A53_0", "Cortex-A53_1"], ["Cortex-A57_0", "Cortex-A57_1"]]
coreNames_cortexA57 = ["Cortex-A57_0", "Cortex-A57_1"]
coreNames_cortexA53 = ["Cortex-A53_0", "Cortex-A53_1"]
blCores = [["Cortex-A57_0", "Cortex-A57_1"], ["Cortex-A53_0", "Cortex-A53_1"]]
TRACE_RANGE_DESCRIPTION = '''Limit trace capture to the specified range. This is
 useful for restricting trace capture to an OS (e.g. Linux kernel)'''
# Import core specific functions
import a57_rams
import a53_rams
class DtslScript(ConfigurationBaseSDF):
    @staticmethod
    def getOptionList():
        return [
            DTSLv1.tabSet("options", "Options", childOptions=
                [DTSLv1.tabPage("trace", "Trace Capture", childOptions=[
                    DTSLv1.enumOption('traceCapture', 'Trace capture method',
 defaultValue="none",
                        values = [("none", "None"), ("CSTMC", "On Chip Trace Buffer
 (CSTMC/ETF)"), ("DSTREAM", "DSTREAM 4GB TraceBuffer")],
                        setter=DtslScript.setTraceCaptureMethod),
                    DTSLv1.infoElement("traceOpts", "Trace Options", childOptions=[
                        DTSLv1.integerOption('timestampFrequency', 'Timestamp
 frequency', defaultValue=25000000, isDynamic=False, description="This value will be
 used to set the Counter Base Frequency ID Register of the Timestamp generator.\nIt
 represents the number of ticks per second and is used to translate the timestamp
 value reported into anumber of seconds.\nNote that changing this value may not
 result in a change in the observed frequency."),
                    ]),
                    DTSLv1.infoElement("offChip", "Off-Chip Trace", childOptions=[
                        DTSLv1.enumOption('tpiuPortWidth', 'TPIU Port Width',
 defaultValue="16",
                             values = [("1", "1 bit"), ("2", "2 bit"), ("3", "3
 bit"), ("4", "4 bit"), ("5", "5 bit"), ("6", "6 bit"), ("7", "7 bit"), ("8", "8
 bit"), ("9", "9 bit"), ("10", "10 bit"), ("11", "11 bit"), ("12", "12 bit"),("13",
 "13 bit"), ("14", "14 bit"), ("15", "15 bit"), ("16", "16 bit")], isDynamic=False),
                    ]),
                ])]
                +[DTSLv1.tabPage("Cortex-A53_SMP_0", "Cortex-A53", childOptions=[
                    DTSLv1.booleanOption('coreTrace', 'Enable Cortex-A53 core
 trace', defaultValue=False,
                        childOptions =
                            # Allow each source to be enabled/disabled individually
                             [ DTSLv1.booleanOption('Cortex-A53_SMP_0_%d' % core,
 "Enable " + clusterCores[0][core] + " trace",defaultValue=True)
                            for core in range(len(clusterCores[0])) ] +
                             [ DTSLv1.booleanOption('timestamp', "Enable ETM
 Timestamps", description="Controls the output oftimestamps into the ETM output
 streams", defaultValue=True) ] +
                             [ DTSLv1.booleanOption('contextIDs', "Enable ETM
 Context IDs", description="Controls the output ofcontext ID values into the ETM
 output streams", defaultValue=True)
                            ] +

 [ ETMv4TraceSource.cycleAccurateOption(DtslScript.getSourcesForCluster("Cortex-
A53_SMP_0"))] +
                            [ # Trace range selection (e.g. for linux kernel)
                            DTSLv1.booleanOption('traceRange', 'Trace capture
 range',
                                description=TRACE_RANGE_DESCRIPTION,
                                defaultValue = False,
                                childOptions = [
                                    DTSLv1.integerOption('start', 'Start address',
                                        description='Start address for trace
 capture',
                                        defaultValue=0,
                                        display=IIntegerOption.DisplayFormat.HEX),
                                    DTSLv1.integerOption('end', 'End address',
                                        description='End address for trace capture',
                                        defaultValue=0xFFFFFFFF,
                                        display=IIntegerOption.DisplayFormat.HEX)
                                ])
                            ]
                        ),
                ])]
                +[DTSLv1.tabPage("Cortex-A57_SMP_0", "Cortex-A57", childOptions=[
                    DTSLv1.booleanOption('coreTrace', 'Enable Cortex-A57 core
 trace', defaultValue=False,
                        childOptions =
                            # Allow each source to be enabled/disabled individually
                             [ DTSLv1.booleanOption('Cortex-A57_SMP_0_%d' % core,
 "Enable " + clusterCores[1][core] + " trace",defaultValue=True)
                            for core in range(len(clusterCores[1])) ] +
                             [ DTSLv1.booleanOption('timestamp', "Enable ETM
 Timestamps", description="Controls the output oftimestamps into the ETM output
 streams", defaultValue=True) ] +
                             [ DTSLv1.booleanOption('contextIDs', "Enable ETM
 Context IDs", description="Controls the output ofcontext ID values into the ETM
 output streams", defaultValue=True)
```

```
                             ] +

 [ ETMv4TraceSource.cycleAccurateOption(DtslScript.getSourcesForCluster("Cortex-
A57_SMP_0"))] +
                             [ # Trace range selection (e.g. for linux kernel)
                             DTSLv1.booleanOption('traceRange', 'Trace capture
 range',
                                 description=TRACE_RANGE_DESCRIPTION,
                                 defaultValue = False,
                                 childOptions = [
                                     DTSLv1.integerOption('start', 'Start address',
                                         description='Start address for trace
 capture',
                                         defaultValue=0,
                                         display=IIntegerOption.DisplayFormat.HEX),
                                     DTSLv1.integerOption('end', 'End address',
                                         description='End address for trace capture',
                                         defaultValue=0xFFFFFFFF,
                                         display=IIntegerOption.DisplayFormat.HEX)
                                 ])
                             ]
                         ),
                     ])]
                 )
             ]
     def __init__(self, root):
         ConfigurationBaseSDF.__init__(self, root)
         self.discoverDevices()
         self.createTraceCapture()
     # +----------------------------+
     # | Target dependent functions |
     # +----------------------------+
     def discoverDevices(self):
         '''Find and create devices'''
         #MemAp devices
         AXIAP(self, self.findDevice("CSMEMAP_0"), "CSMEMAP_0")
         AHBAP(self, self.findDevice("CSMEMAP_1"), "CSMEMAP_1")
         # The ATB stream ID which will be assigned to trace sources.
         streamID = 1
         self.cortexA57cores = []
         for coreName in (coreNames_cortexA57):
             # Create core
             coreDevice = a57_rams.A57CoreDevice(self, self.findDevice(coreName),
 coreName)
             self.cortexA57cores.append(coreDevice)
             self.addDeviceInterface(coreDevice)
             a57_rams.registerInternalRAMs(coreDevice)
             # Create CTI (if a CTI exists for this core)
             ctiName = self.getCTINameForCore(coreName)
             if not ctiName is None:
                 coreCTI = CSCTI(self, self.findDevice(ctiName), ctiName)
             # Create Trace Macrocell (if a macrocell exists for this core -
 disabled by default - will enable with option)
             tmName = self.getTraceSourceNameForCore(coreName)
             if not tmName == None:
                 tm = ETMv4TraceSource(self, self.findDevice(tmName), streamID,
 tmName)
                 streamID += 2
                 tm.setEnabled(False)
         self.cortexA53cores = []
         for coreName in (coreNames_cortexA53):
             # Create core
             coreDevice = a53_rams.A53CoreDevice(self, self.findDevice(coreName),
 coreName)
             self.cortexA53cores.append(coreDevice)
             self.addDeviceInterface(coreDevice)
             a53_rams.registerInternalRAMs(coreDevice)
             # Create CTI (if a CTI exists for this core)
             ctiName = self.getCTINameForCore(coreName)
             if not ctiName is None:
                 coreCTI = CSCTI(self, self.findDevice(ctiName), ctiName)
```

```
            # Create Trace Macrocell (if a macrocell exists for this core -
 disabled by default - will enable with option)
            tmName = self.getTraceSourceNameForCore(coreName)
            if not tmName == None:
                tm = ETMv4TraceSource(self, self.findDevice(tmName), streamID,
 tmName)
                streamID += 2
                tm.setEnabled(False)
        tmc = CSTMC(self, self.findDevice("CSTMC"), "CSTMC")
        tmc.setMode(CSTMC.Mode.ETF)
        tpiu = CSTPIU(self, self.findDevice("CSTPIU"), "CSTPIU")
        tpiu.setEnabled(False)
        tpiu.setFormatterMode(FormatterMode.CONTINUOUS)
        # Create and Configure Funnels
        self.createFunnel("CSTFunnel")
        self.setupCTISyncSMP()
        self.setupCTISyncBigLittle(blCores)
    def createTraceCapture(self):
        # ETF Devices
        etfTrace = TMCETBTraceCapture(self, self.getDeviceInterface("CSTMC"),
 "CSTMC")
        self.addTraceCaptureInterface(etfTrace)
        # DSTREAM
        self.createDSTREAM()
        self.addTraceCaptureInterface(self.DSTREAM)
    def createDSTREAM(self):
        self.DSTREAM = DSTREAMTraceCapture(self, "DSTREAM")
    # +--------------------------------+
    # | Callback functions for options |
    # +--------------------------------+
    def optionValuesChanged(self):
        '''Callback to update the configuration state after options are changed'''
        if not self.isConnected():
            self.setInitialOptions()
        self.updateDynamicOptions()
    def setInitialOptions(self):
        '''Set the initial options'''
        traceMode = self.getOptionValue("options.trace.traceCapture")
        coreTraceEnabled = self.getOptionValue("options.Cortex-
A53_SMP_0.coreTrace")
        for core in range(len(clusterCores[0])):
            thisCoreTraceEnabled = self.getOptionValue("options.Cortex-
A53_SMP_0.coreTrace.Cortex-A53_SMP_0_%d" % core)
            tmName = self.getTraceSourceNameForCore(clusterCores[0][core])
            coreTM=self.getDeviceInterface(tmName)
            enableSource = coreTraceEnabled and thisCoreTraceEnabled
            self.setTraceSourceEnabled(tmName, enableSource)
            if(self.getOptionValue("options.Cortex-
A53_SMP_0.coreTrace.traceRange")):
                coreTM.clearAllTraceRanges()
                coreTM.addTraceRange(self.getOptionValue("options.Cortex-
A53_SMP_0.coreTrace.traceRange.start"),
                                     self.getOptionValue("options.Cortex-
A53_SMP_0.coreTrace.traceRange.end"))
            coreTM.setTimestampingEnabled(self.getOptionValue("options.Cortex-
A53_SMP_0.coreTrace.timestamp"))
            self.setContextIDEnabled(coreTM,
                                     self.getOptionValue("options.Cortex-
A53_SMP_0.coreTrace.contextIDs"),
                                     "32")
        coreTraceEnabled = self.getOptionValue("options.Cortex-
A57_SMP_0.coreTrace")
        for core in range(len(clusterCores[1])):
            thisCoreTraceEnabled = self.getOptionValue("options.Cortex-
A57_SMP_0.coreTrace.Cortex-A57_SMP_0_%d" % core)
            tmName = self.getTraceSourceNameForCore(clusterCores[1][core])
            coreTM=self.getDeviceInterface(tmName)
            enableSource = coreTraceEnabled and thisCoreTraceEnabled
            self.setTraceSourceEnabled(tmName, enableSource)
            if(self.getOptionValue("options.Cortex-
A57_SMP_0.coreTrace.traceRange")):
```

```
                 coreTM.clearAllTraceRanges()
                 coreTM.addTraceRange(self.getOptionValue("options.Cortex-
A57_SMP_0.coreTrace.traceRange.start"),
                                      self.getOptionValue("options.Cortex-
A57_SMP_0.coreTrace.traceRange.end"))
             coreTM.setTimestampingEnabled(self.getOptionValue("options.Cortex-
A57_SMP_0.coreTrace.timestamp"))
             self.setContextIDEnabled(coreTM,
                                      self.getOptionValue("options.Cortex-
A57_SMP_0.coreTrace.contextIDs"),
                                      "32")
         if self.getOptions().getOption("options.trace.offChip.tpiuPortWidth"):

 self.setPortWidth(int(self.getOptionValue("options.trace.offChip.tpiuPortWidth")))
         if self.getOptions().getOption("options.trace.offChip.traceBufferSize"):

 self.setTraceBufferSize(self.getOptionValue("options.trace.offChip.traceBufferSize"))
         self.configureTraceCapture(traceMode)
     def updateDynamicOptions(self):
         '''Update the dynamic options'''
         for core in range(len(self.cortexA57cores)):
             a57_rams.applyCacheDebug(configuration = self,
                                      optionName = "options.rams.cacheDebug",
                                      device = self.cortexA57cores[core])
             a57_rams.applyCachePreservation(configuration = self,
                                             optionName =
 "options.rams.cachePreserve",
                                             device = self.cortexA57cores[core])
         for core in range(len(self.cortexA53cores)):
             a53_rams.applyCacheDebug(configuration = self,
                                      optionName = "options.rams.cacheDebug",
                                      device = self.cortexA53cores[core])
             a53_rams.applyCachePreservation(configuration = self,
                                             optionName =
 "options.rams.cachePreserve",
                                             device = self.cortexA53cores[core])
     def setTraceCaptureMethod(self, method):
         '''Simply call into the configuration to enable the trace capture device.
         CTI devices associated with the capture will also be configured'''
         self.enableTraceCapture(method)
     #Configurations may also contain static functions such as as this, which
     #allow the return of device instances via parameter-binding
     @staticmethod
     def getSourcesForCluster(cluster):
         '''Get the Trace Sources for a given coreType
            Use parameter-binding to ensure that the correct Sources
            are returned for the core type and cluster passed only'''
         def getClusterSources(self):
             return self.getTraceSourcesForCluster(cluster)
         return getClusterSources
     # +-----------------------------+
     # | Target independent functions |
     # +-----------------------------
     def postConnect(self):
         ConfigurationBaseSDF.postConnect(self)
         try:
             freq =
 self.getOptionValue("options.trace.traceOpts.timestampFrequency")
         except:
             return
         # Update the value so the trace decoder can access it
         tsInfo = TimestampInfo(freq)
         self.setTimestampInfo(tsInfo)
..
```

Related information

DTSL configuration execution flow on page 287
Hardware configurations created by the PCE on page 269

10.4.8 DTSL configuration execution flow

The DTSL script allows you to customize device behavior and configuration, and enables you to use specialized implementations of DTSL devices and trace capture objects.

The execution flow of the script is:

Static option provision

The static function `getOptionsList()` is available so that a client can acquire a list of options without instantiating the configuration. Details of how Arm® Debugger uses this function can be found in DTSL options.

Script Construction

The constructor of a default (for example, PCE built) script calls two functions - `discoverDevices()` and `createTraceCapture()`.

The `discoverDevices()` function is responsible for the creation of all DTSL objects which implement the `IDevice` interface. Configuration of the DTSL devices which are created by default, is performed here. You can also initialize and configure any devices here. At this point, the configuration is not connected, so it is important not to try to access the physical device. Most of all objects that are created are not assigned to a field, but an `IDeviceConnection` interface for any device can always be retrieved from the configuration using the `getDeviceInterface(deviceName)` function. The `getDeviceInterface(deviceName)` function is provided by the `ConfigurationBase` DTSL class, from which all DTSL configurations are derived. For example, `self.funnel = getDeviceInterface("CSTFunnel_1")`.

To configure SMP, big.LITTLE™, and DynamIQ™ DTSL devices for synchronized execution debug, the `discoverDevices()` function also, where appropriate for the target, makes internal calls to the `ConfigurationBaseSDF` class.

Next, the constructor calls `createTraceCapture()`. Within this function, all trace capture devices are created and added to the configuration. The capture devices are added to the configuration using the function `addTraceCaptureInterface()`. The `addTraceCaptureInterface()` function also configures the trace/formatter mode of the capture device, and disables it. If you instantiate more devices here, they must implement the `ITraceCapture` interface and any specific configuration which does not require a target connection. A target connection is a connection that is not yet made at the point of initialization of the configuration object. Trace capture devices are not assigned to a field, but can be acquired at any point after initialization using `getTraceCaptureDevices()[devicename]`. For example, `self.ETF =getTraceCaptureDevices()["CSTMC_1"]`.

After the configuration is constructed, all DTSL objects should be created and configured so far as they are ready to start interacting with the target.

Commit of option values

Immediately after construction, `optionValuesChanged()` is called to give the configuration an opportunity to configure the DTSL devices. The DTSL devices are configured according to the selected option values. Further information about how Arm Development Studio stores and uses the option values can be found in DTSL options.

Connection and trace start

Now that the configuration and all devices are initialized correctly, connection to the target occurs. If configured, trace starts for all enabled source and sink devices. The configuration, and also the DTSL devices, provide overridable hooks. The overridable hooks are called at key events during connection or tracing. For more information, see the following sections.

Adding connection-event functionality hooks

Overridable hooks are provided to perform any additional actions at key events during the connection phase. These hooks can be added to the DTSL file and default behavior augmented, or overridden completely, by omitting a call to the parent class. The following hooks are available:

postRDDIConnect()

Called immediately after an RDDI interface has been opened, but no connection to the debug server has been made. This method can be overridden to perform low-level initialization of the target. For example, a JTAG interface can be acquired and scans performed to unlock a TAP Controller.

postDebugConnect()

Called immediately after a connection is made to the debug server. No target interaction has occurred, but devices can be instantiated and connected here to perform any target initialization which, if omitted, would cause the DTSL connection to fail. For example, writing through a DAP to power debug components.

postConnect()

Called after the RDDI debug interface is opened, and all devices to which the configuration can connect, are connected. Any connected devices can be used directly in this function.

preDisconnect()

Called immediately before the connection is closed. This is an opportunity to perform any target-specific clean up after every debug session.

All the hooks above are also available on the individual DTSL device objects, which are called as part of the configuration event-hooks, as described above. For further information and some example uses of these hooks, see Performing custom actions on connect.

Adding trace-event functionality hooks

In addition to the connection hooks, the configuration also provides hooks for events related to trace. The hooks apply both to the configuration as a whole (for example, called on configuration once per event), and also to individual trace objects (for example, called once per event on all `ITraceDevice` based objects).

preTraceStart()

No trace data is captured. The component or configuration performs any actions necessary before capture is started.

traceStart()

Trace capture has started. The component or configuration performs actions to start trace data production.

postTraceStart()

The trace capture system is now capturing trace from all components that are configured to produce it.

preTraceStop()

Called before trace stop.

traceStop()

Called at the point of stopping trace. Trace does not stop until the ancestor method is called.

postTraceStop()

Called after trace stop. No components are now producing trace.

Overriding script behavior

Modifying the synchronized execution method

Where configurations support synchronized debug of SMP/AMP cores, the `discoverDevices()` function makes internal calls to `setupCTISyncSMP()`. If appropriate, it also makes internal calls to `setupCTISyncBigLittle()`. This is known as hardware synchronization, and is described in Hardware synchronization.

If you see any problems while debugging using these devices (for example, cores fail to restart), it is possible to use a synchronization method which does not use the CTI devices. This might resolve debug issues where CTI halt or restart is not working as expected. To use this method, you should change the SMP setup calls to `setupSimpleSyncSMP()` and `setupSimpleSyncBigLittle()`. This is 'tight' synchronization, as described in Tight synchronization.

Changing the cores for big.LITTLE SMP debug

For targets which support big.LITTLE core configurations, the cores debugged by the big.LITTLE device are identified at the top of the configuration file. For example:

```
blCores = [["Cortex-A57_0", "Cortex-A57_1"],["Cortex-A53_0"], ["Cortex-A53_1"]]
```

You might be required to modify cores that are considered for big.LITTLE SMP devices because the Platform Configuration Editor groups all 'big' and 'LITTLE' cores into a single big.LITTLE SMP device. You can add or remove cores from this list. Only cores of the same type can be added to any single list, and all cores must support big.LITTLE configurations.

Overriding CTM channels

The CTM channels for synchronized execution and trace triggering are configured by default as:

Table 10-5: CTM channels default configuration for synchronized execution and trace triggering

	Sync Stop	Sync Start	Trace Trigger
v8 systems	0	1	2
v7 systems	2	1	3
Function to override	`getCtmChannelStop()`	`getCtmChannelStart()`	`GetCtmChannelTrigger()`

To modify the channel used for any of the triggers, you can add the appropriate function to the script, and have it return the required channel number.

For example:

```
def getCtmChannelStop(self):
     return 1
```

Subclassing existing DTSL classes and including user-defined classes

You can provide specialized implementations of trace devices which are instantiated in the `discoverDevices()` or `createTraceCapture()` functions. For information related to the DTSL class hierarchies and requirements for interface implementation of derived or specialized types, see Main DTSL classes and hierarchy.

Related information

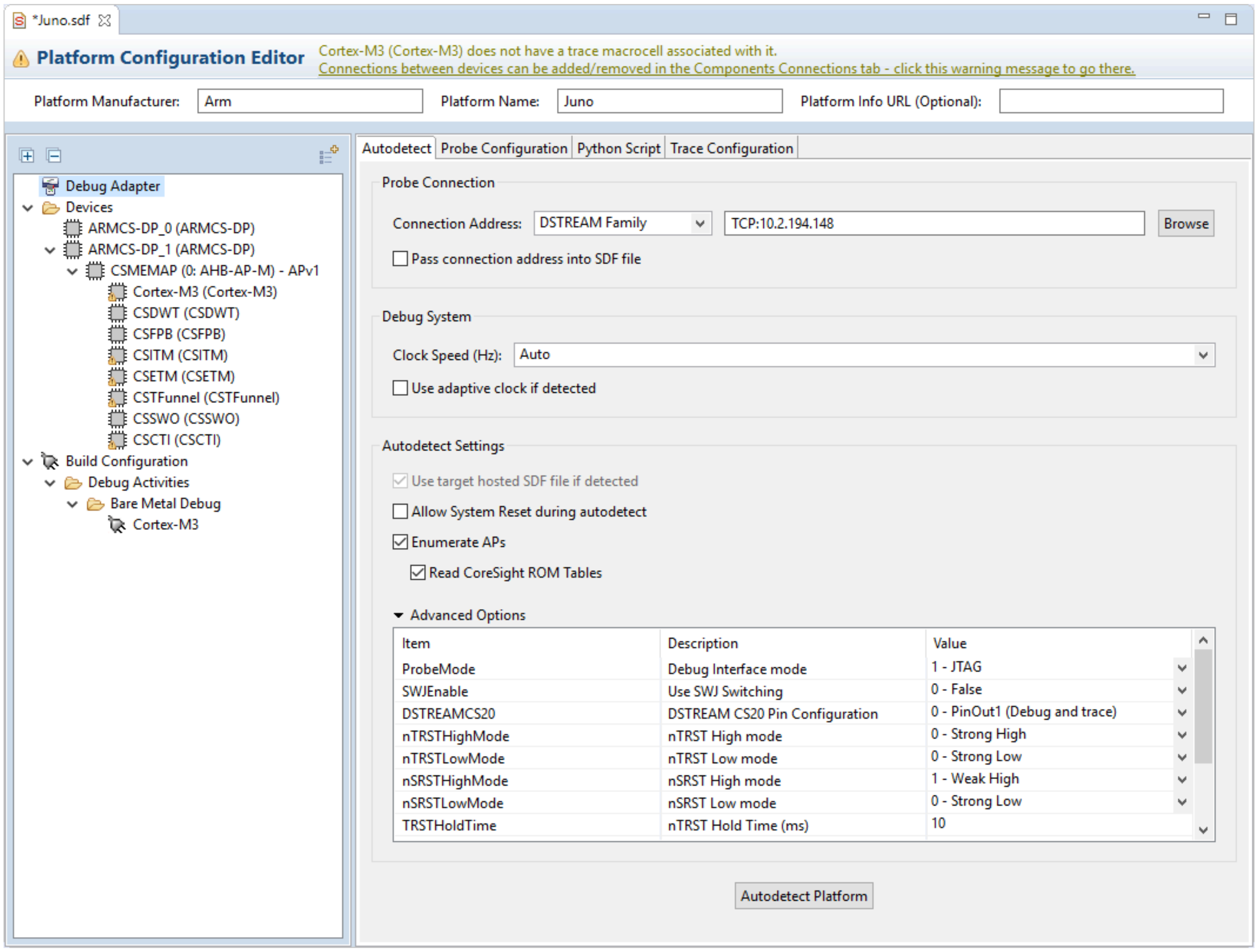
DTSL Jython configuration file structure on page 281
Hardware configurations created by the PCE on page 269

10.4.9 Debug Adapter configuration in the PCE

View configuration information for your debug adapter in the **Platform Configuration Editor**.

To see the configuration settings for your debug adapter, in the **Platform Configuration Editor** view, under the device hierarchy, select **Debug Adapter**.

Figure 10-25: Device Adapter tabs



*Juno.sdf

Platform Configuration Editor

Cortex-M3 (Cortex-M3) does not have a trace macrocell associated with it.
Connections between devices can be added/removed in the Components Connections tab - click this warning message to go there.

Platform Manufacturer: Arm
Platform Name: Juno
Platform Info URL (Optional):

- Debug Adapter
- Devices
  - ARMCS-DP_0 (ARMCS-DP)
  - ARMCS-DP_1 (ARMCS-DP)
    - CSMEMAP (0: AHB-AP-M) - APv1
      - Cortex-M3 (Cortex-M3)
      - CSDWT (CSDWT)
      - CSFPB (CSFPB)
      - CSITM (CSITM)
      - CSETM (CSETM)
      - CSTFunnel (CSTFunnel)
      - CSSWO (CSSWO)
      - CSCTI (CSCTI)
- Build Configuration
  - Debug Activities
    - Bare Metal Debug
      - Cortex-M3

Autodetect | Probe Configuration | Python Script | Trace Configuration

Probe Connection

Connection Address: DSTREAM Family TCP:10.2.194.148 Browse

☐ Pass connection address into SDF file

Debug System

Clock Speed (Hz): Auto

☐ Use adaptive clock if detected

Autodetect Settings

☑ Use target hosted SDF file if detected
☐ Allow System Reset during autodetect
☑ Enumerate APs
☑ Read CoreSight ROM Tables

Advanced Options

Item	Description	Value
ProbeMode	Debug Interface mode	1 - JTAG
SWJEnable	Use SWJ Switching	0 - False
DSTREAMCS20	DSTREAM CS20 Pin Configuration	0 - PinOut1 (Debug and trace)
nTRSTHighMode	nTRST High mode	0 - Strong High
nTRSTLowMode	nTRST Low mode	0 - Strong Low
nSRSTHighMode	nSRST High mode	1 - Weak High
nSRSTLowMode	nSRST Low mode	0 - Strong Low
TRSTHoldTime	nTRST Hold Time (ms)	10

Autodetect Platform

Access

- In the **Project Explorer**, right-click on a `.sdf` file, and select **Open With** > **Platform Configuration Editor**.
- Double-click a `.sdf` file (where the SDF association has not been overridden).
- In the **Project Explorer**, right-click, and select **File** > **New** > **Platform Configuration**. After autodetection or manual configuration of a platform, the PCE view opens.
- Select **File** > **New** > **Other...** > **Configuration Database** > **Platform Configuration**. After autodetection or manual configuration of a platform, the PCE view opens.
- Enter the PCE and access debug adapter settings at the final stages of creating a new **Hardware Connection**. Select **File** > **New** > **Hardware Connection**. At the target selection step, click **Add a new platform...** and follow the steps in the wizard.
- At the end of the target selection flow for a CMSIS device; click **Target Configuration**.

Contents

The **Debug Adapter** configuration tabs contain:

Table 10-6: Debug Adapter tabs contents

Tab	Description
Autodetect	Sets the autodetection configuration, using: • Probe Connection - Add or browse for a connection address, and choose whether to pass the connection address into the SDF file. • Debug System - Configure the clock speed and choose to use adaptive clock if detected. • Autodetect settings - Choose to allow system reset, enumerate APs, read CoreSight™ ROM Tables, and more advanced options. • Autodetect Platform - Run to autodetect the platform.
Probe Configuration	View descriptions and set values for each probe configuration item. For example, use the *UserOut_nn* configuration item to set User I/O pin values for your debug adapter (where *nn* is the *UserOut* item number as displayed). Locate and select the *UserOut_nn* item and select either *0 - Low* or *1 - High*.
Python Script	Adds a Python script to hook debug events and provide custom behavior. This script is added to the configuration file as a configuration item.
Trace Configuration	Selects parallel or HSSTP trace type, view a description for each trace type item, and configure the value set for each item.

Related information

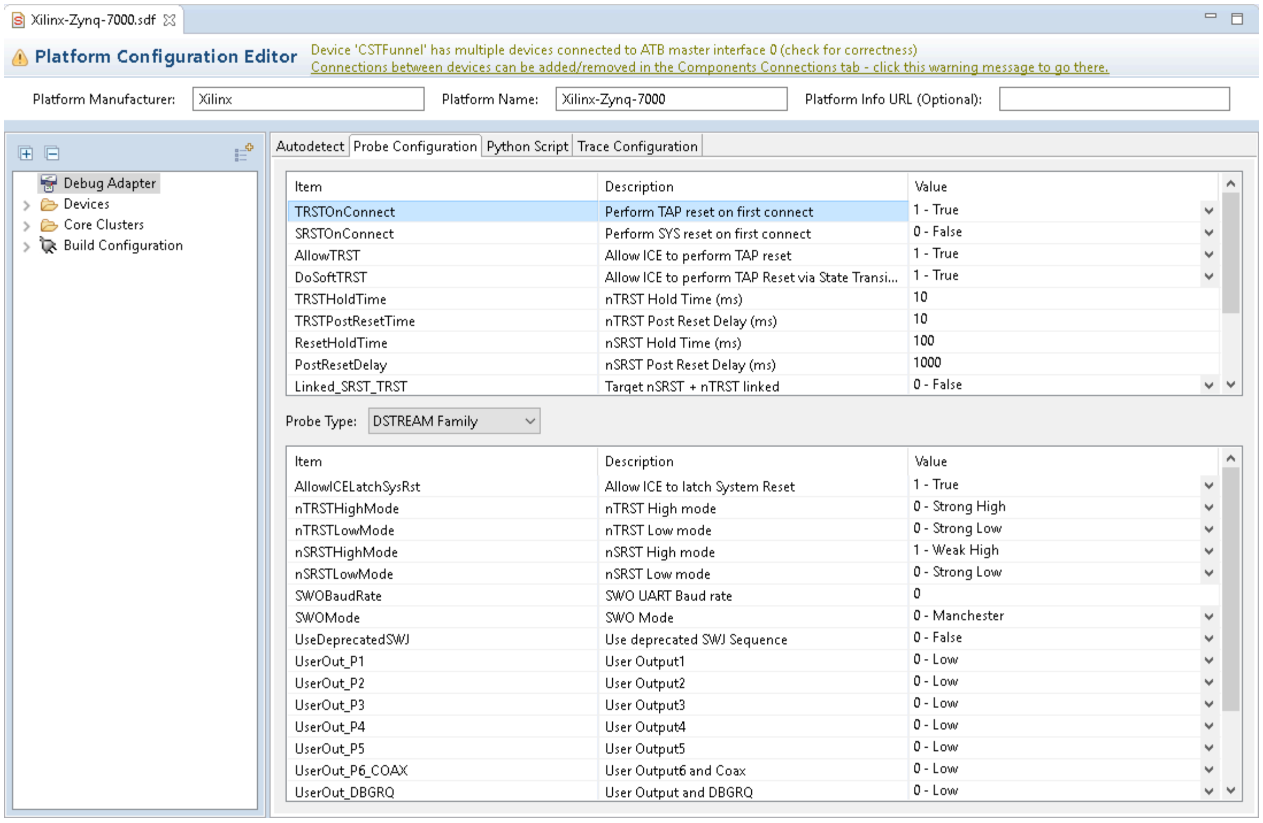
Platform Configuration Editor (PCE) on page 186
Device hierarchy in the PCE view on page 190
Add core cluster components dialog box on page 216

10.4.10 Debug adapter advanced configuration options

Depending on the debug probe connected to your development platform, Arm® Development Studio provides options to set values for each probe configuration item.

You can set the values in the Platform Configuration Editor (PCE) under **Debug Adapter** in the **Probe Configuration** tab.

Figure 10-26: Debug adapter advanced configuration example



Device 'CSTFunnel' has multiple devices connected to ATB master interface 0 (check for correctness)
Connections between devices can be added/removed in the Components Connections tab - click this warning message to go there.

Platform Manufacturer: Xilinx
Platform Name: Xilinx-Zynq-7000
Platform Info URL (Optional):

Item	Description	Value
TRSTOnConnect	Perform TAP reset on first connect	1 - True
SRSTOnConnect	Perform SYS reset on first connect	0 - False
AllowTRST	Allow ICE to perform TAP reset	1 - True
DoSoftTRST	Allow ICE to perform TAP Reset via State Transi...	1 - True
TRSTHoldTime	nTRST Hold Time (ms)	10
TRSTPostResetTime	nTRST Post Reset Delay (ms)	10
ResetHoldTime	nSRST Hold Time (ms)	100
PostResetDelay	nSRST Post Reset Delay (ms)	1000
Linked_SRST_TRST	Target nSRST + nTRST linked	0 - False

Probe Type: DSTREAM Family

Item	Description	Value
AllowICELatchSysRst	Allow ICE to latch System Reset	1 - True
nTRSTHighMode	nTRST High mode	0 - Strong High
nTRSTLowMode	nTRST Low mode	0 - Strong Low
nSRSTHighMode	nSRST High mode	1 - Weak High
nSRSTLowMode	nSRST Low mode	0 - Strong Low
SWOBaudRate	SWO UART Baud rate	0
SWOMode	SWO Mode	0 - Manchester
UseDeprecatedSWJ	Use deprecated SWJ Sequence	0 - False
UserOut_P1	User Output1	0 - Low
UserOut_P2	User Output2	0 - Low
UserOut_P3	User Output3	0 - Low
UserOut_P4	User Output4	0 - Low
UserOut_P5	User Output5	0 - Low
UserOut_P6_COAX	User Output6 and Coax	0 - Low
UserOut_DBGRQ	User Output and DBGRQ	0 - Low

Note

The options available depend on the selected debug **Probe Type**.

Table 10-7: Debug adapter advanced configuration options

Configuration item name	Supported probes	Description and supported values
`AllowICELatchSysRst`	DSTREAM family only	Allow the debug probe to latch System Reset. `True` - This is the default. Your debug hardware unit performs a `nTRST` reset holding `nSRST`. When `True`, the debug hardware unit can extend the time the target controller holds the target in reset. It uses the `ResetHoldTime` configuration item to set the time that reset is held. The debug hardware unit can then apply breakpoint settings before the processor starts execution. This option is useful for debugging a target from reset. It allows the unit to stop the processor on the first instruction fetch after reset is released by the unit. Setting the option `True` also ensures that the Test Access Port (TAP) state machine and associated debug logic is properly reset. `False` - When `False`, the breakpoint settings only take effect after the processor has already started execution, preventing debugging of the handler.
`UseDeprecatedSWJ`	DSTREAM family only	Specify that your debug hardware unit must use the deprecated SWJ switching sequence. Use this item for older SWD-compatible targets which use the SWJ switching sequence.
`nSRSTHighMode`	DSTREAM family only	Set the drive strength to use when the reset signal is in the `high` or `inactive` state. You can set the strength to be one of the following values: `strong`, `weak high`, or `not driven` (tristate).
`nSRSTLowMode`	DSTREAM family only	Set the drive strength to use when the reset signal is in the `low` or `active` state. You can set the strength to be one of the following values: `strong`, `weak low`, or `not driven` (tristate).
`nTRSTHighMode`	DSTREAM family only	Reset mode used for the `nTRST` signal in the high state.
`nTRSTLowMode`	DSTREAM family only	Set the drive strength to use when the reset signal is in the `low`, or `active` state. You can set the strength to be one of the following values: `strong`, `weak low`, or `not driven` (tristate).

Configuration item name	Supported probes	Description and supported values
`SWOMode`	DSTREAM family only	Protocol used to carry `SWO` data. Depending on the target mode, set to `Manchester` or `UART`. If the `SWO Mode` is set to `UART`, the debug hardware unit can detect the SWO UART Baud rate. This setting has no effect in `Manchester` mode.
`SWOBaudRate`	DSTREAM family only	SWO UART Baud rate. Specify the frequency of the incoming data. If you set this to `0`, the system attempts to autodetect the baud rate. `UART` mode in the `SWO` context also means `Non Return to Zero` (NRZ) mode.
`UserOut_nn`	DSTREAM family only	Sets the state of the USER IO pins on the DSTREAM family of debug probes. Specify the User I/O pin values for your debug adapter (where `nn` is the `UserOut` item number as displayed). Locate and select the UserOut_nn item and select either `0 - Low` or `1 - High`.
`UserOut_P6_COAX`	DSTREAM family only	User-defined hardware output pin 6 on header, and output co-axial connector.
`UserOut_DBGRQ`	DSTREAM family only	User-defined hardware output to `DBGRQ` on JTAG connector.
`PowerFilterTime`	DSTREAM family only	The power status filter. Limit sending power status notification messages.
`DSTREAMCS20`	DSTREAM family only	DSTREAM CoreSight™ 20 Pin Configuration
`JTAGAutoMaxFreq`	DSTREAM family only	Set maximum frequency (Hz) used by the JTAG auto tune process in DSTREAM-ST.

Configuration item name	Supported probes	Description and supported values
`AllowTRST`	All probes	Allow the debug probe to perform a TAP Reset. A TAP reset performs a JTAG state machine reset and resets any TAP Controllers that are receiving commands from the debug hardware unit. `True` - This is the default. Your debug hardware unit holds the `nTRST` line active long enough to perform any post-reset setup that might be required after a target-initiated reset. Use the `TRSTHoldTime` config item to extend the time the target is held in reset. `False`- Your debug hardware unit does not assert the reset line. Note that post-reset setup might not be complete before the target starts to run. To use JTAG state transitions to reset just the state machine, set this item to `False`, and use the `DoSoftTRST` item instead.
`DoSoftTRST`	All probes	Allow debug probe to perform a TAP reset through JTAG state transitions. Perform a JTAG sequence using the Test-Logic-Reset to force a reset. This is done in addition to asserting the `nTRST` line if the `AllowTRST` configuration item is set to `True`. **Note:** This option replaces `DoSoftTAPReset`.
`ProbeMode`	All probes	The debug interface mode. You can set the interface that connects to the target DAP to be either `JTAG` or `SWD`. If set to `SWD`, your debug probe connects to your target using the `SWD` protocol instead of `JTAG`.
`SWJEnable`	All probes	Use SWJ to switch the target between SWD and JTAG. If a target supports both JTAG and SWD, you must enable this setting before you autoconfigure the target.
`ResetHoldTime`	All probes	`nSRST` hold time in milliseconds. Specify the time in milliseconds the target is held in a hardware reset state.
`TRSTHoldTime`	All probes	`nTRST` hold time in milliseconds. Specify the time in milliseconds the target is held in a hardware reset state.

Configuration item name	Supported probes	Description and supported values
`PostResetDelay`	All probes	`nSRST` post reset delay time in milliseconds. Specify the time in milliseconds that the debug probe must wait after the reset is released, before attempting any other debugging operation.
`TRSTPostResetTime`	All probes	`nTRST` post reset delay time in milliseconds. Specify the time in milliseconds that the debug probe must wait after the reset is released, before attempting any other debugging operations.
`TRSTOnConnect`	All probes	Perform `TAP` reset on connection to the first available core. Reset the target hardware by asserting the `nTRST` signal when connecting to the first available core in a debug session.
`SRSTOnConnect`	All probes	Perform `SYS` reset on connection to the first available core. Reset the target hardware by asserting the `nSRST` signal when connecting to the first available core in a debug session.
`DoSoftTAPReset`	All probes	Reset the JTAG logic in the target hardware by forcing transitions within its state machine. This is done in addition to holding the `nTRST TAP` reset signal `LOW`. Specify this option if `nTRST` is not connected, or if the target hardware requires that you force a reset of the JTAG logic whenever resetting. **Note:** This option is deprecated. Arm recommends using `DoSoftTRST` instead.
`Linked_SRST_TRST`	All probes	Set `TRUE` if the target hardware has the `nSRST` and `nTRST` signals physically linked.
`PythonScript`	All probes	The python script to hook debug events and provide custom behavior. You can specify the python script in the **Python script** tab in the PCE.
`PowerUpGPR`	All probes	Set the item to enable powerup using any ROM table GPRs (ADIv6 debug systems only).

Related information

Add a third-party debug probe on page 274
Debug Adapter configuration in the PCE on page 290
Configure your debug hardware unit for Platform Autodetection on page 270

Device hierarchy in the PCE view on page 190

10.4.11 DSTREAM-PT trace modes

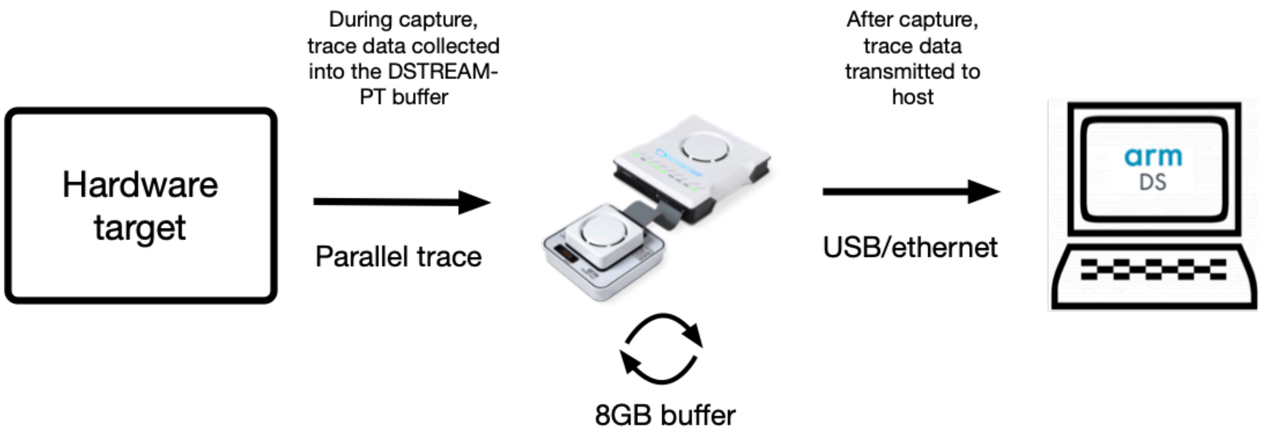
There are two trace modes available in DSTREAM-PT, *store and forward mode*, and *streaming mode*.

Store and forward mode

Store and forward mode is when the DSTREAM-PT probe uses its onboard storage buffer to hold captured trace data. When the trace capture is completed, the data is then forwarded, on demand, to the host machine.

If the onboard storage reaches the 8GB maximum capacity and there is still incoming trace data, the buffer wraps.

Figure 10-27: store and forward mode diagram


During capture, trace data collected into the DSTREAM-PT buffer
After capture, trace data transmitted to host
Hardware target
Parallel trace
USB/ethernet
arm DS
8GB buffer

Streaming mode

Streaming mode is when the DSTREAM-PT probe forwards trace data to the host machine, as it is capturing trace.

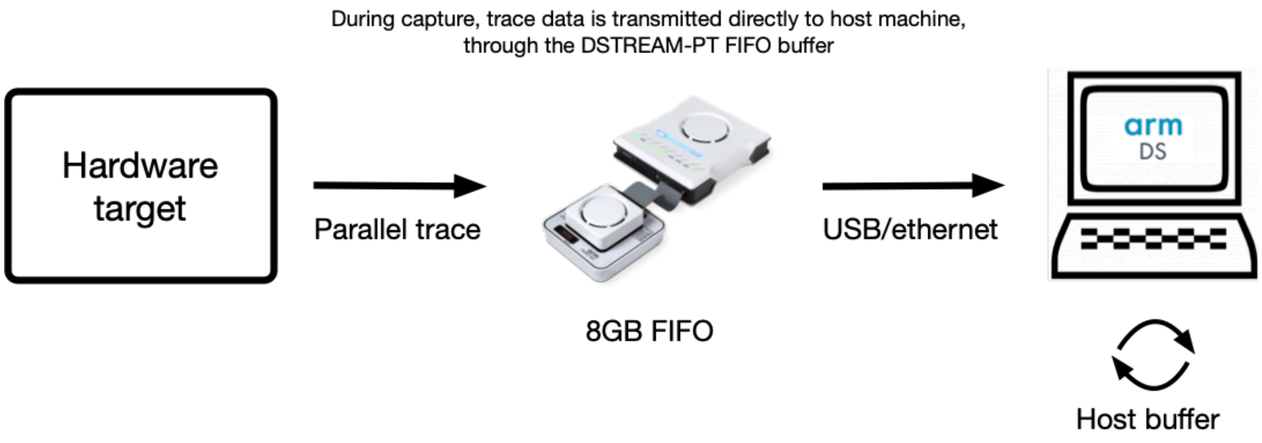
When using streaming mode:

- The storage buffer is on the host machine, and can be any size. The Arm® Development Studio IDE lists 128GB as the maximum size, but you can overwrite this in the DTSL script.
- Live decode of trace events (ITM/STM) is supported.
- Stop on trigger is not supported.

DSTREAM-PT streaming trace mode has an 8GB First-In, First-Out (FIFO) buffer in between the target and the host. This stabilizes the transfer rate of trace data to the host machine. It also means that the buffer does not overflow if there is a sudden burst of trace data from the target, or if the host computer is temporarily distracted from receiving the incoming trace.

When you stop trace capture, DSTREAM-PT ensures that any data in the FIFO buffer gets drained before the transfer completes. You can view the captured trace data in the Trace view.

Figure 10-28: streaming mode diagram


During capture, trace data is transmitted directly to host machine, through the DSTREAM-PT FIFO buffer
Hardware target
Parallel trace
8GB FIFO
USB/ethernet
arm DS
Host buffer

Caution

If the average rate of trace incoming from the target is higher than what the host is receiving, the buffer eventually overflows when the 8GB FIFO is full. In this situation, DSTREAM-PT stops capturing new trace, and the existing 8GB of trace data in the FIFO is delivered to the host. If this happens, a warning message is displayed in the Commands view.

Which mode to use

Both trace modes are suitable for most targets. However there are some situations when one mode is preferable to the other:

Use *store and forward mode* when:

- Your target is producing fast trace.
- You are using a slow connection, such as ethernet rather than USB.

Use *streaming mode* when:

- You want to collect more than 8GB of trace data.
- You want to decode trace events live.

Related information

Configure DSTREAM-PT trace mode on page 299
Trace view on page 399
Trace Control view on page 404
Commands view on page 329
Events view on page 344
DTSL options on page 535
Configuring trace for bare-metal or Linux kernel targets

10.4.12 Configure DSTREAM-PT trace mode

Configure Arm® Development Studio to use either *store and forward mode*, or *streaming mode*, when you are connecting to a DSTREAM-PT debug probe.

Before you begin

- You need a debug connection to a target that is connected to a DSTREAM-PT probe.

About this task

For more information on these modes, see DSTREAM-PT trace modes.

Procedure

1. Open the **Debug and Trace Services Layer (DTSL) Configuration** dialog box:
   a) In the **Debug Control** view, right-click your connection and select **Debug Configurations**.
   b) In the **Connection** tab, click the DTSL Options **Edit** button.
2. Select a trace mode from the **Trace Buffer** tab:

Tip

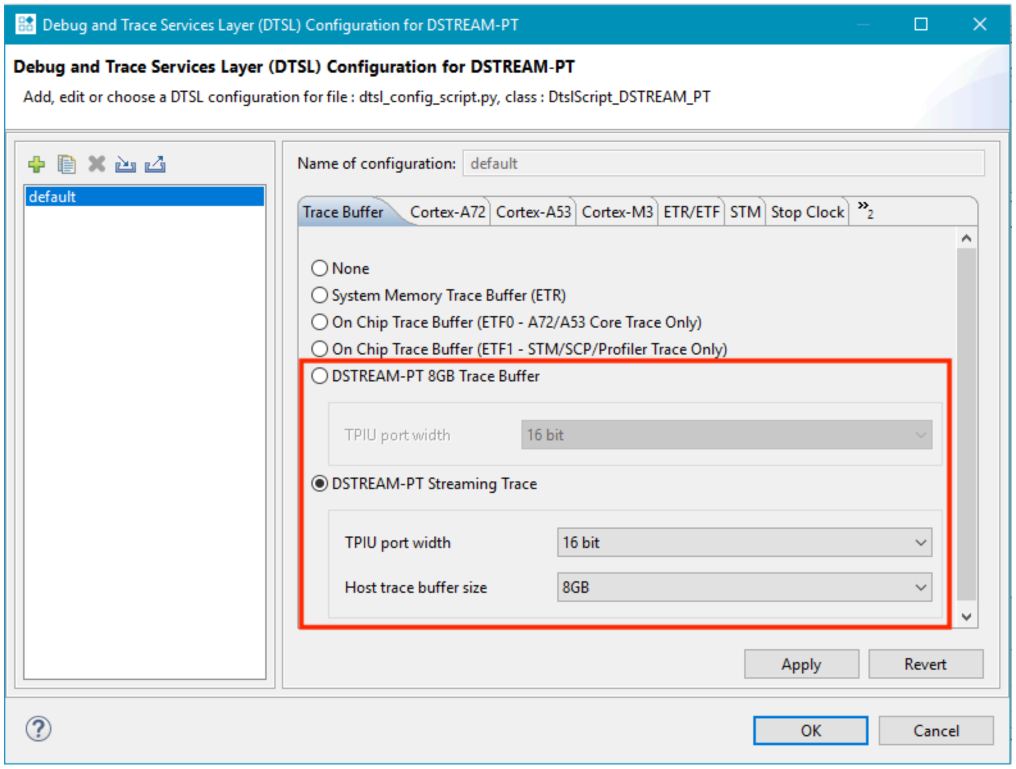
See the DSTREAM-PT trace modes topic for guidance on which off-chip trace mode to use.

- To use the store and forward mode, select the **DSTREAM-PT 8GB Trace Buffer** option and modify the **TPIU port width** if required.
- To use streaming mode, select the **DSTREAM-PT Streaming Trace** option:
  a. Modify the **TPIU port width** if required.
  b. Select a **Host trace buffer size**, that is, how much trace data you want to store on your host machine.

Note

Store and forward mode and *streaming mode* are for off-chip trace. If your target has on-chip trace, you might see additional trace modes listed.

Figure 10-29: Select a trace mode


Debug and Trace Services Layer (DTSL) Configuration for DSTREAM-PT

Debug and Trace Services Layer (DTSL) Configuration for DSTREAM-PT

Add, edit or choose a DTSL configuration for file : dtsl_config_script.py, class : DtslScript_DSTREAM_PT

default

Name of configuration: default

Trace Buffer | Cortex-A72 | Cortex-A53 | Cortex-M3 | ETR/ETF | STM | Stop Clock | »2

- None
- System Memory Trace Buffer (ETR)
- On Chip Trace Buffer (ETF0 - A72/A53 Core Trace Only)
- On Chip Trace Buffer (ETF1 - STM/SCP/Profiler Trace Only)
- DSTREAM-PT 8GB Trace Buffer
  - TPIU port width: 16 bit
- [✓ DSTREAM-PT Streaming Trace]
  - TPIU port width: 16 bit
  - Host trace buffer size: 8GB

Apply | Revert

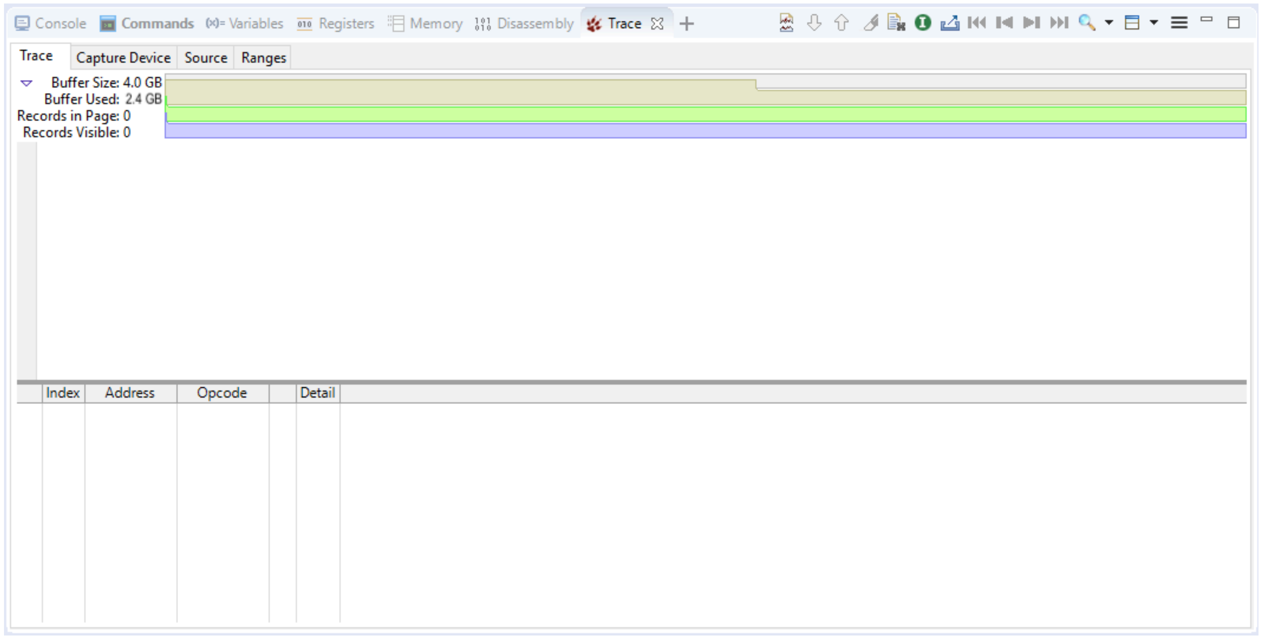
OK | Cancel

3. Enable trace for your core. The **DTSL Configuration** dialog box shows the processors on the target that are capable of trace:
   a) Open the relevant processor tab, and select the **Enable <core_name> core trace** checkbox.
   b) If your target supports ITM/STM trace, open the relevant ITM/STM tab and select the **Enable ITM/STM trace** checkbox.
4. Click **Apply** and then **OK** to apply the settings and close the **DTSL Configuration** dialog box.
5. [Optional] Add the application that you want to run to generate trace data.
   a) In the **Debug Configurations** window, open the **Files** tab.
   b) In the **Target Configuration** section, the `.axf` file that you want to run.
6. [Optional] For targets that have an ITM/STM trace source, you can also view the trace events. To enable this, you need to:
   - Run application code that causes ITM/STM trace events.
   - Configure the **Events View** settings for your target. See the **ITM_Cortex-M4_MPS2** example in `<installation-directory>/examples/Bare-metal_examples_Armv7/`, which describes how to configure the settings for the **MPS2_M4** target.
7. Click **Debug** to start capturing trace.

Results

During the trace capture session, open the **Trace View** to see the live capture of the trace data:

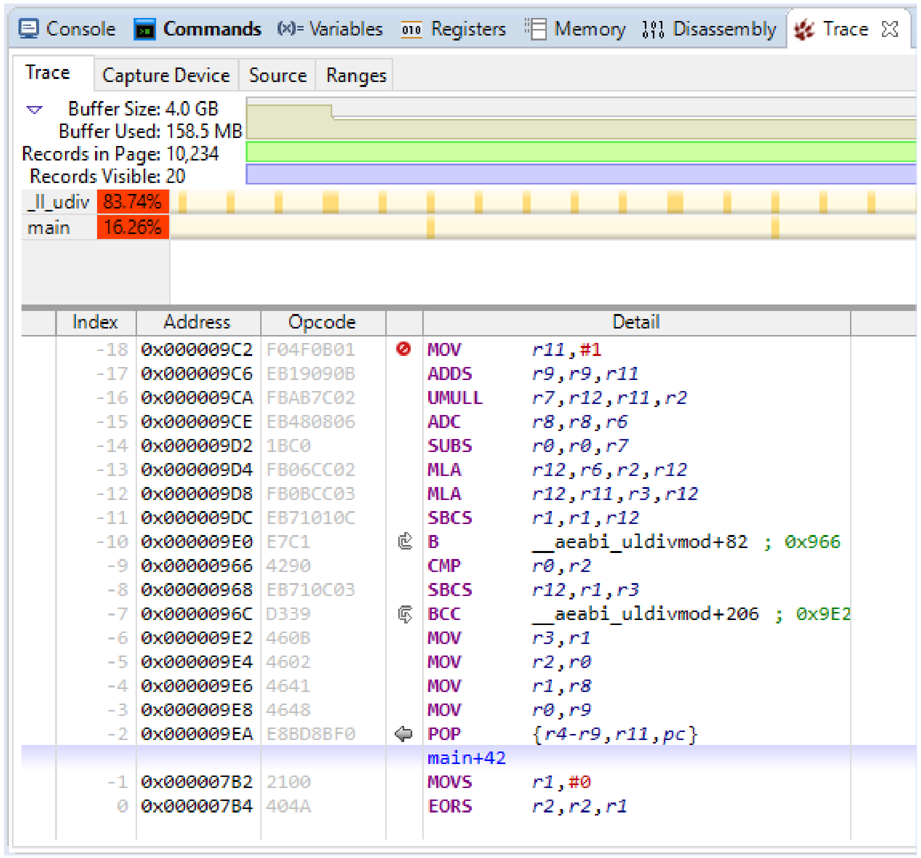
Figure 10-30: Trace view showing a trace data capture session



Buffer Size: 4.0 GB
Buffer Used: 2.4 GB
Records in Page: 0
Records Visible: 0

When you stop the capture session, the **Trace View** shows a list of executed instructions:

Figure 10-31: Trace view showing captured trace data



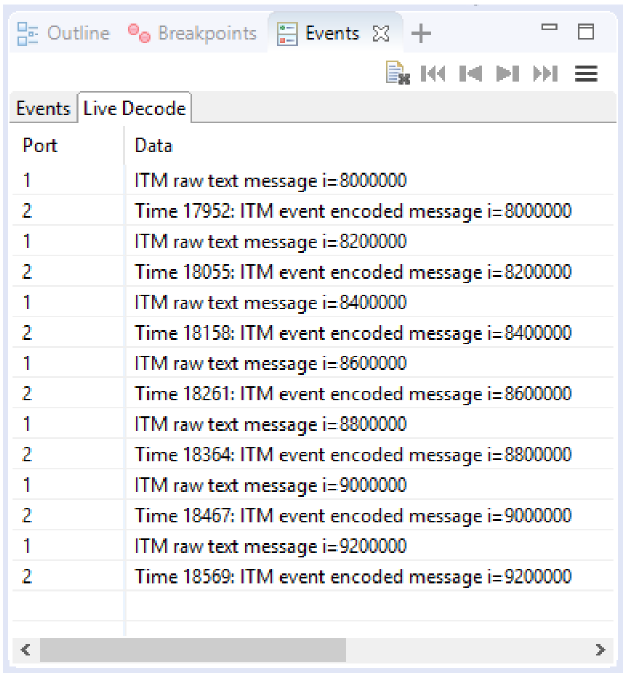
```
Buffer Size: 4.0 GB
Buffer Used: 158.5 MB
Records in Page: 10,234
Records Visible: 20
_ll_udiv 83.74%
main 16.26%
```

Index	Address	Opcode		Detail
-18	0x000009C2	F04F0B01		MOV r11,#1
-17	0x000009C6	EB19090B		ADDS r9,r9,r11
-16	0x000009CA	FBAB7C02		UMULL r7,r12,r11,r2
-15	0x000009CE	EB480806		ADC r8,r8,r6
-14	0x000009D2	1BC0		SUBS r0,r0,r7
-13	0x000009D4	FB06CC02		MLA r12,r6,r2,r12
-12	0x000009D8	FB0BCC03		MLA r12,r11,r3,r12
-11	0x000009DC	EB71010C		SBCS r1,r1,r12
-10	0x000009E0	E7C1		B __aeabi_uldivmod+82 ; 0x966
-9	0x00000966	4290		CMP r0,r2
-8	0x00000968	EB710C03		SBCS r12,r1,r3
-7	0x0000096C	D339		BCC __aeabi_uldivmod+206 ; 0x9E2
-6	0x000009E2	460B		MOV r3,r1
-5	0x000009E4	4602		MOV r2,r0
-4	0x000009E6	4641		MOV r1,r8
-3	0x000009E8	4648		MOV r0,r9
-2	0x000009EA	E8BD8BF0		POP {r4-r9,r11,pc}
				main+42
-1	0x000007B2	2100		MOVS r1,#0
0	0x000007B4	404A		EORS r2,r2,r1

If your target has an ITM/STM trace source, and you have enabled and configured trace events decode:

- When using streaming mode, during trace capture open the **Events View**. Click the **Live Decode** tab to see a list of events as they are received:

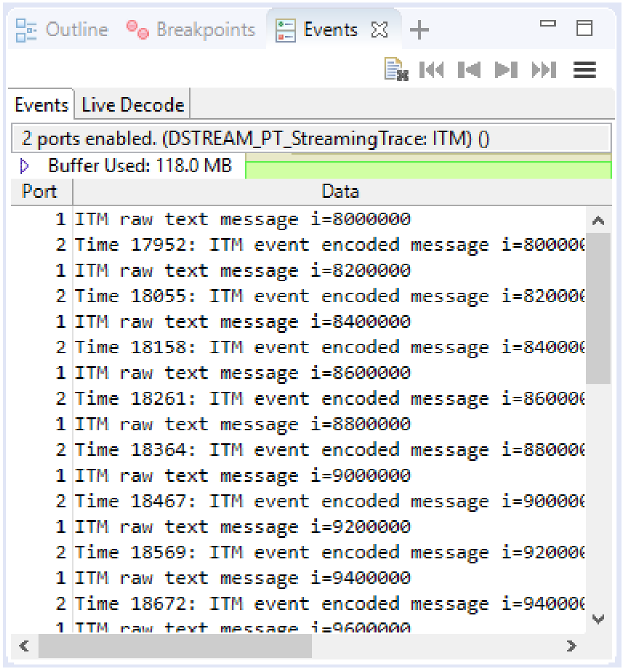
Figure 10-32: Events view showing a trace data capture session with incoming trace events



Port	Data
1	ITM raw text message i=8000000
2	Time 17952: ITM event encoded message i=8000000
1	ITM raw text message i=8200000
2	Time 18055: ITM event encoded message i=8200000
1	ITM raw text message i=8400000
2	Time 18158: ITM event encoded message i=8400000
1	ITM raw text message i=8600000
2	Time 18261: ITM event encoded message i=8600000
1	ITM raw text message i=8800000
2	Time 18364: ITM event encoded message i=8800000
1	ITM raw text message i=9000000
2	Time 18467: ITM event encoded message i=9000000
1	ITM raw text message i=9200000
2	Time 18569: ITM event encoded message i=9200000

- When using store and forward mode, live decode is not available. Instead, open the **Events view** and when the capture session has stopped, you can see a list of all the trace events.

Figure 10-33: Events view showing captured trace events



2 ports enabled. (DSTREAM_PT_StreamingTrace: ITM) ()

Buffer Used: 118.0 MB

```
Port                Data
   1 ITM raw text message i=8000000
   2 Time 17952: ITM event encoded message i=800000
   1 ITM raw text message i=8200000
   2 Time 18055: ITM event encoded message i=820000
   1 ITM raw text message i=8400000
   2 Time 18158: ITM event encoded message i=840000
   1 ITM raw text message i=8600000
   2 Time 18261: ITM event encoded message i=860000
   1 ITM raw text message i=8800000
   2 Time 18364: ITM event encoded message i=880000
   1 ITM raw text message i=9000000
   2 Time 18467: ITM event encoded message i=900000
   1 ITM raw text message i=9200000
   2 Time 18569: ITM event encoded message i=920000
   1 ITM raw text message i=9400000
   2 Time 18672: ITM event encoded message i=940000
   1 ITM raw text message i=9600000
```

Related information
DSTREAM-PT trace modes on page 298
Trace view on page 399
Trace Control view on page 404
Commands view on page 329
Events view on page 344
Configuring trace for bare-metal or Linux kernel targets

10.5 DSTREAM dashboard

Use the DSTREAM dashboard to view the state of a remote connection to a DSTREAM or DSTREAM-ST unit. Use the DSTREAM Web API to retrieve data from the remote unit.

10.5.1 DSTREAM dashboard overview

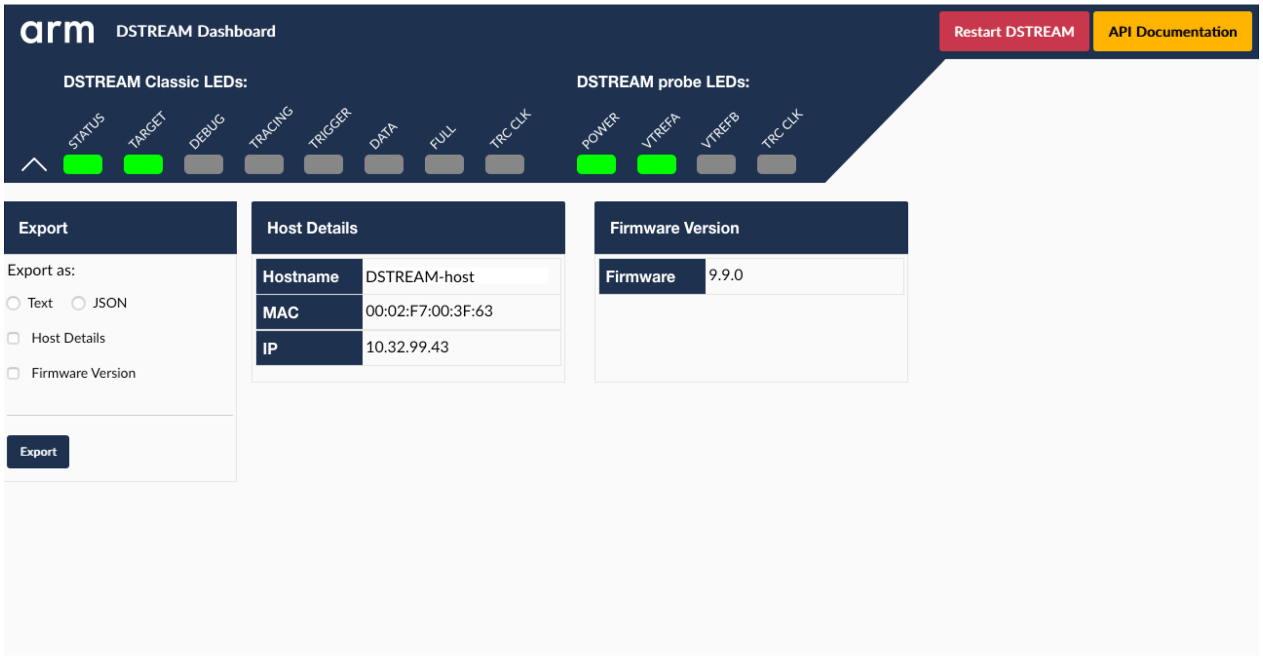
Use the DSTREAM dashboard to connect to a legacy DSTREAM or DSTREAM-ST unit remotely.

When connected, you can view the status LEDs on the unit, restart the unit, access the API documentation, and more.

Note

If you are using DSTREAM-PT or DSTREAM-XT, you can only view information about your DSTREAM-ST unit. If you are using a DSTREAM-HT, see DSTREAM-HT dashboard overview.

Figure 10-34: DSTREAM dashboard



Hostname	DSTREAM-host
MAC	00:02:F7:00:3F:63
IP	10.32.99.43

Firmware	9.9.0

Access the DSTREAM dashboard

Open a web browser and enter the IP address or hostname of your DSTREAM unit in the address bar.

See Connect to a DSTREAM unit remotely for details on how to find this information for your unit.

Features

Table 10-8: DSTREAM dashboard features

Name	Description
View the status of the unit	The dashboard displays a visual representation of the status LEDs that are physically present on the unit.
Restart the unit	Remotely restart your unit.
Export data	Export the host and firmware version details, in either plain text or JSON format.
Access the API documentation	Read how to access all the information that is presented on the dashboard by using the Web APIs instead.

Using the DSTREAM dashboard

Restart the unit

1. Click the **Restart DSTREAM** button to restart the unit. A dialog asks you to confirm the action.
2. Wait a few seconds and then refresh the page.

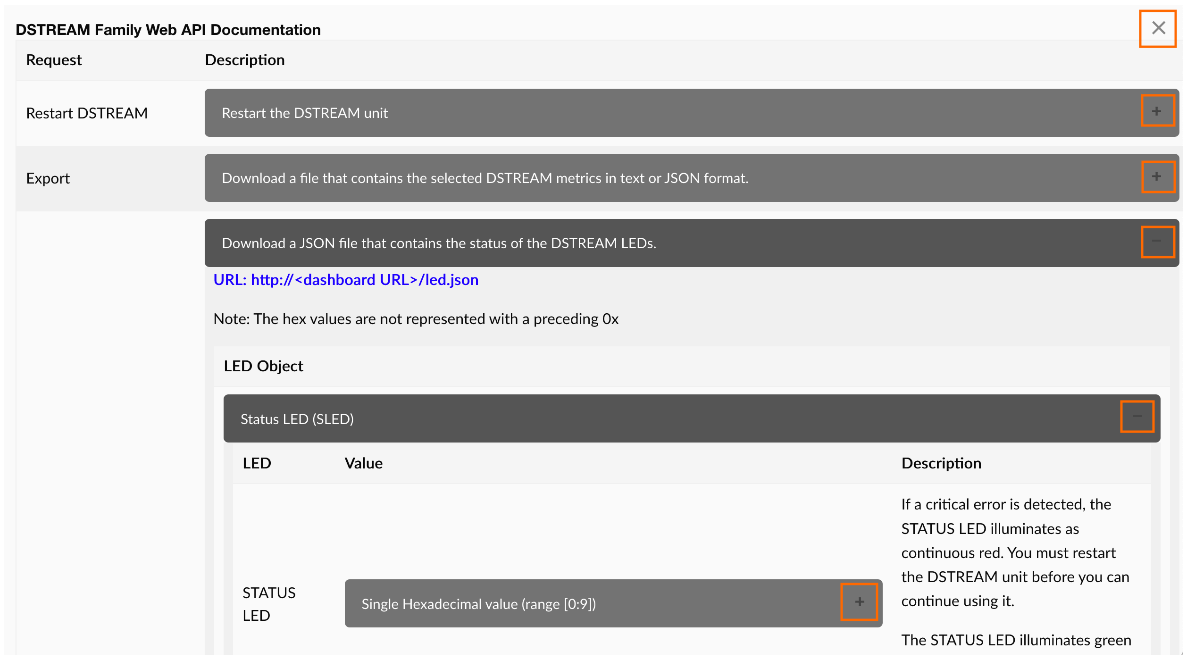
Export data

- Use the **Export** section to select the type and data you want to export. You can export data from the **Host Details** and the **Firmware Version** sections.

Access the API documentation

Click the **API Documentation** button and the documentation screen opens.

Figure 10-35: DSTREAM dashboard



DSTREAM Family Web API Documentation

Request	Description
Restart DSTREAM	Restart the DSTREAM unit
Export	Download a file that contains the selected DSTREAM metrics in text or JSON format.
	Download a JSON file that contains the status of the DSTREAM LEDs.

URL: http://<dashboard URL>/led.json

Note: The hex values are not represented with a preceding 0x

LED Object

Status LED (SLED)

LED	Value	Description
STATUS LED	Single Hexadecimal value (range [0:9])	If a critical error is detected, the STATUS LED illuminates as continuous red. You must restart the DSTREAM unit before you can continue using it. The STATUS LED illuminates green

To expand and collapse the sections that are exposed by the APIs, use the **+** and **-** buttons.

To return to the dashboard, use the **X** button in the top-right corner.

Note

You can only access the API documentation when you have an active network connection to a DSTREAM or DSTREAM-ST unit.

Related information

DSTREAM-HT dashboard overview on page 312
Connect to a DSTREAM unit remotely on page 307
DSTREAM Web API on page 309

10.5.2 Connect to a DSTREAM unit remotely

Connect to a legacy DSTREAM or DSTREAM-ST unit over a network.

Before you begin

- Your DSTREAM or DSTREAM-ST unit must use firmware version 7.4.0-1 or higher.

- DSTREAM-HT dashboard functionality is available in DSTREAM-ST firmware version 7.8.0.1 and later.
- Your unit must be connected to a network.

About this task

Note

If you are using DSTREAM-PT or DSTREAM-XT, you can only view information about your DSTREAM-ST unit. If you are using a DSTREAM-HT, see DSTREAM-HT dashboard overview.

Procedure

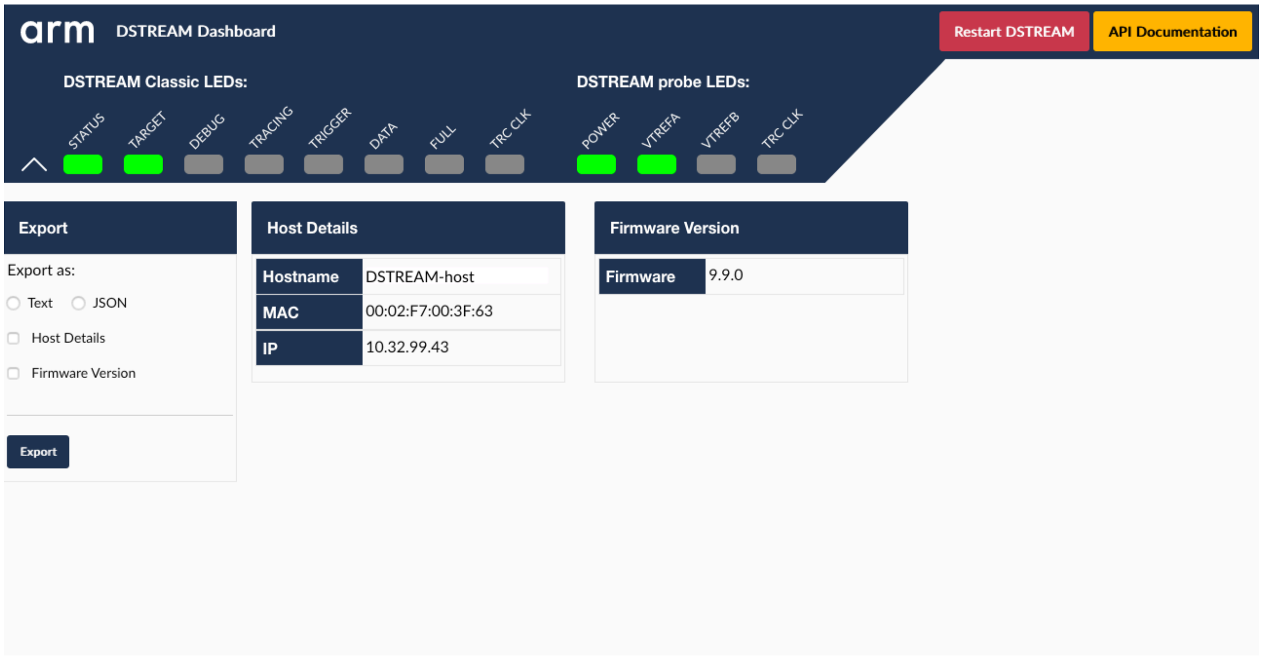
1. Get the IP address or host name of your unit:
   a) Either open one of the following views in Arm® Development Studio:
      - Debug Hardware Configure IP view.
      - Debug Hardware Firmware Installer view.

   b) Or open the Debug Configurations - Connection tab.
   c) Copy the IP address or host name.
2. Open a web browser, and paste the IP address or host name into the address bar, and press **Enter**.

Results

The DSTREAM dashboard opens, displaying the state of the specified unit.

Figure 10-36: DSTREAM dashboard



Hostname	DSTREAM-host
MAC	00:02:F7:00:3F:63
IP	10.32.99.43

Firmware	9.9.0

Next steps

When connected, you can:

- View the state of the unit.
- Restart the unit.

For more information, see DSTREAM dashboard overview.

You can also interact with the unit using APIs. See DSTREAM Web API for more information.

Related information

DSTREAM dashboard overview on page 305
DSTREAM-HT dashboard overview on page 312
DSTREAM Web API on page 309

10.5.3 DSTREAM Web API

Retrieve information about your legacy DSTREAM or DSTREAM-ST unit in JSON format, using the Web API.

Note

- If you are using DSTREAM-PT or DSTREAM-XT, you can only view information about your DSTREAM-ST unit. If you are using a DSTREAM-HT, see DSTREAM-HT dashboard overview.
- To use the Web API and to access the API documentation, you require an active network connection to your DSTREAM or DSTREAM-ST unit.

Syntax

URL: `http://<dashboard-URL>/<request>`

To get your dashboard URL, see Connect to a DSTREAM unit remotely.

Where `request` is one of the following:

`cgi-bin/restart`
Request a restart of the unit.

`info.json`
Request information about your unit.

`led.json`
Request the LED statuses of your unit.

Note

To send these requests, you can use any of the following:

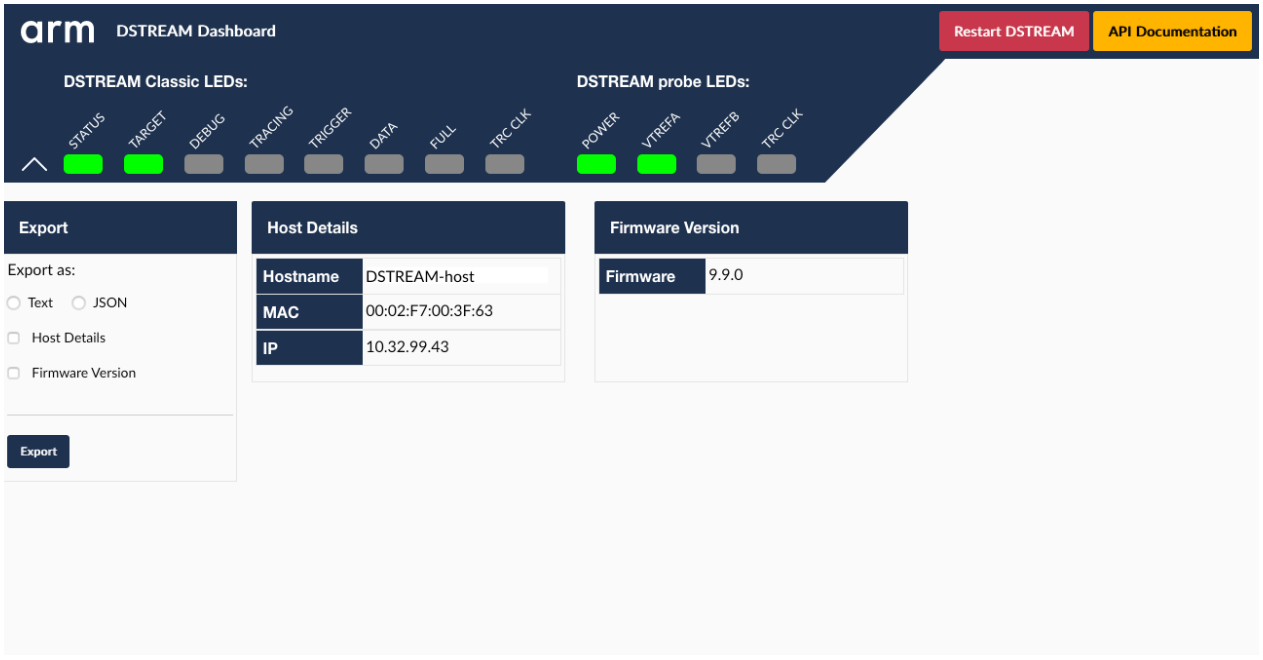
- A web browser.
- A command line utility, for example `curl` or `wget`.

- An HTTP library of any programming language.

Decode the LED data

To decode the retrieved LED data, see the API documentation. It is available from the dashboard:

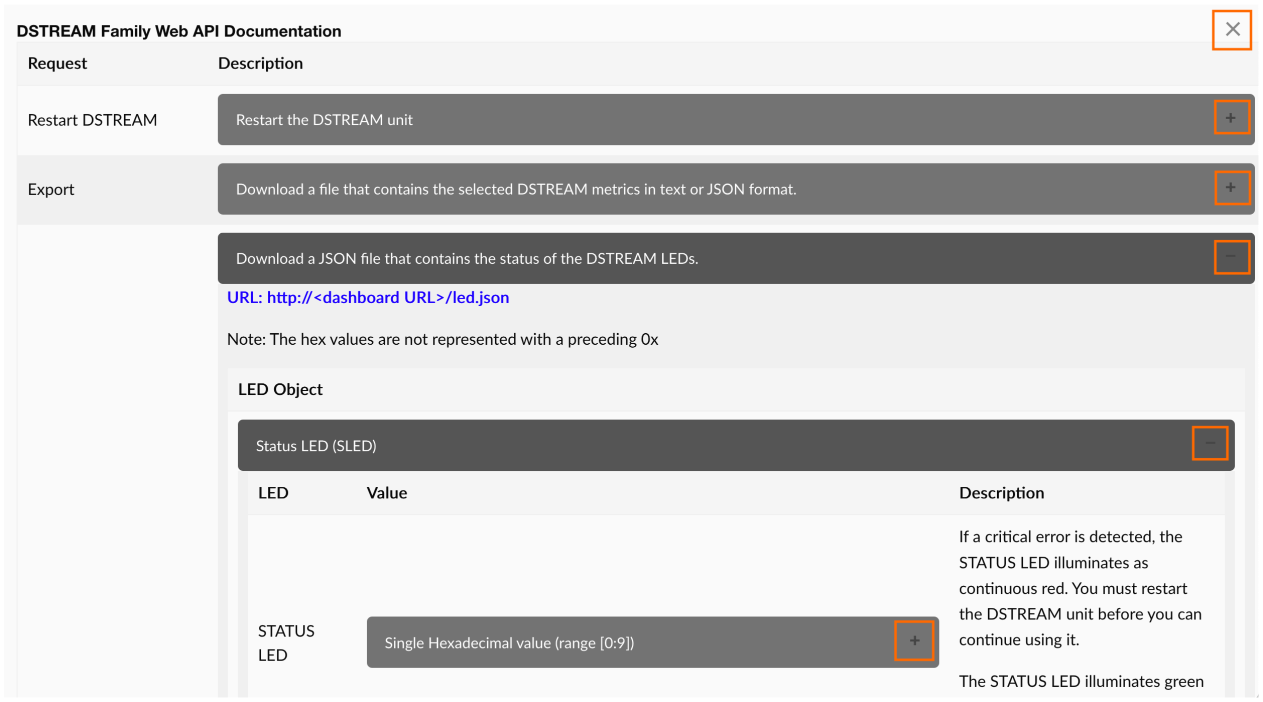
Figure 10-37: DSTREAM dashboard



Hostname	DSTREAM-host
MAC	00:02:F7:00:3F:63
IP	10.32.99.43

Firmware	9.9.0

Figure 10-38: DSTREAM Web API documentation



DSTREAM Family Web API Documentation

Request	Description
Restart DSTREAM	Restart the DSTREAM unit
Export	Download a file that contains the selected DSTREAM metrics in text or JSON format.
	Download a JSON file that contains the status of the DSTREAM LEDs.

URL: http://<dashboard URL>/led.json

Note: The hex values are not represented with a preceding 0x

LED Object

Status LED (SLED)

LED	Value	Description
STATUS LED	Single Hexadecimal value (range [0:9])	If a critical error is detected, the STATUS LED illuminates as continuous red. You must restart the DSTREAM unit before you can continue using it. The STATUS LED illuminates green

Example: Restart the DSTREAM unit

```
http://192.168.0.1/cgi-bin/restart
```

Restarts the unit.

Example: Request device information

```
http://192.168.0.1/info.json
```

Returns a JSON file, that contains information similar to:

```
{"Host Details":{"Hostname":"DSTREAM-
host","MAC":"00:01:02:03:04:05","IP":"192.168.0.1"},"Firmware Version":
{"Firmware":"1.2.3"}}
```

Example: Request the status of the LEDs

```
http://192.168.0.1/led.json
```

Returns a JSON file, that contains information similar to:

```
{"Sled":"1","Dled":"00000020","Tled":"00000000","Pled":"200","Fled":""}
```

Related information

Connect to a DSTREAM unit remotely on page 307
DSTREAM dashboard overview on page 305
DSTREAM-HT dashboard overview on page 312

10.5.4 DSTREAM-HT dashboard overview

Use the DSTREAM-HT dashboard to connect to a DSTREAM-HT system remotely.

In addition to the functionality listed in DSTREAM dashboard overview, with the DSTREAM-HT dashboard you can also view:

- The status LEDs on the HSSTP probe.
- The *Bit Error Rate* (BER) on the incoming data stream, represented in the Data Eye View.

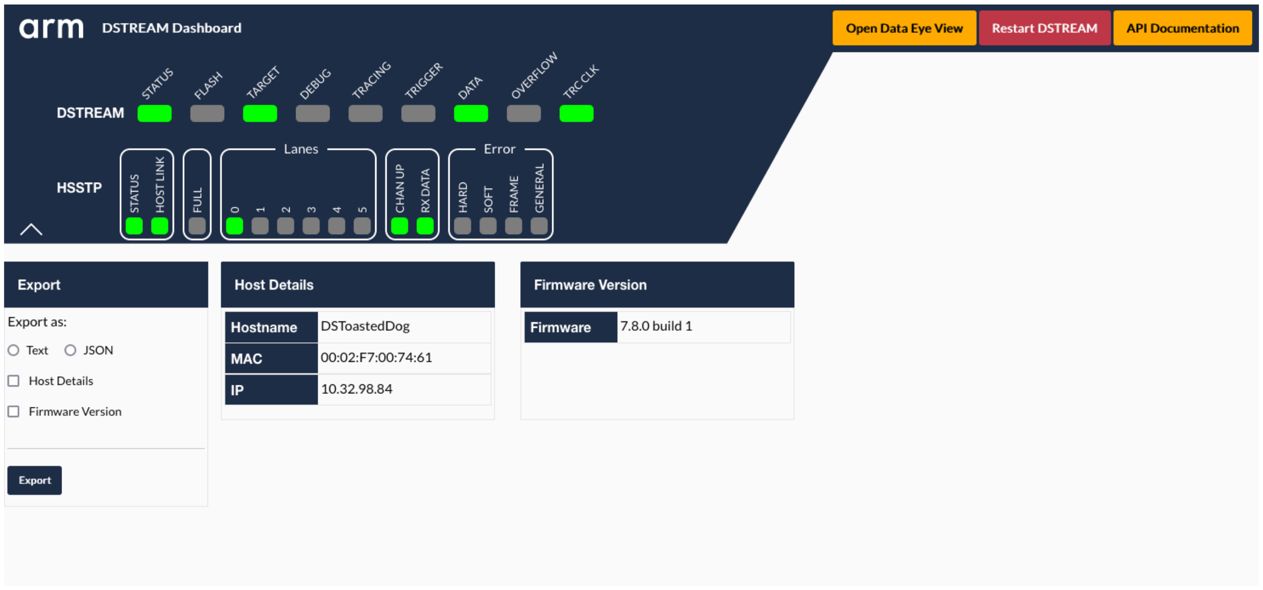
Note

The DSTREAM-HT dashboard functionality is available in DSTREAM-ST firmware version 7.8.0.1 and later.

DSTREAM-HT dashboard

To access the dashboard, open a web browser and enter the IP address or hostname of your DSTREAM-HT system in the address bar. See Connect to a DSTREAM unit remotely for more information.

Figure 10-39: DSTREAM-HT dashboard



Hostname	DSToastedDog
MAC	00:02:F7:00:74:61
IP	10.32.98.84

Firmware	7.8.0 build 1

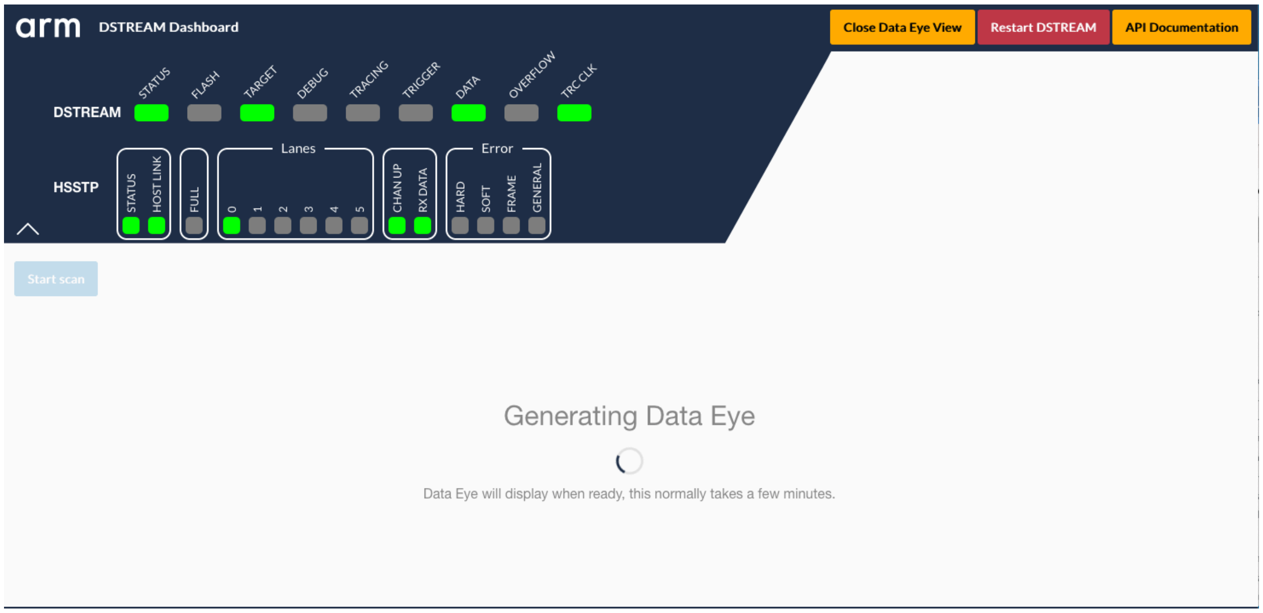
Data Eye View

The Data Eye View shows the BER on an incoming data stream.

To access the View, click the **Open Data Eye View** button at the top right of the screen.

To generate a scan, click the **Start Scan** button, and wait a few minutes.

Figure 10-40: Generate Data Eye



Generating Data Eye

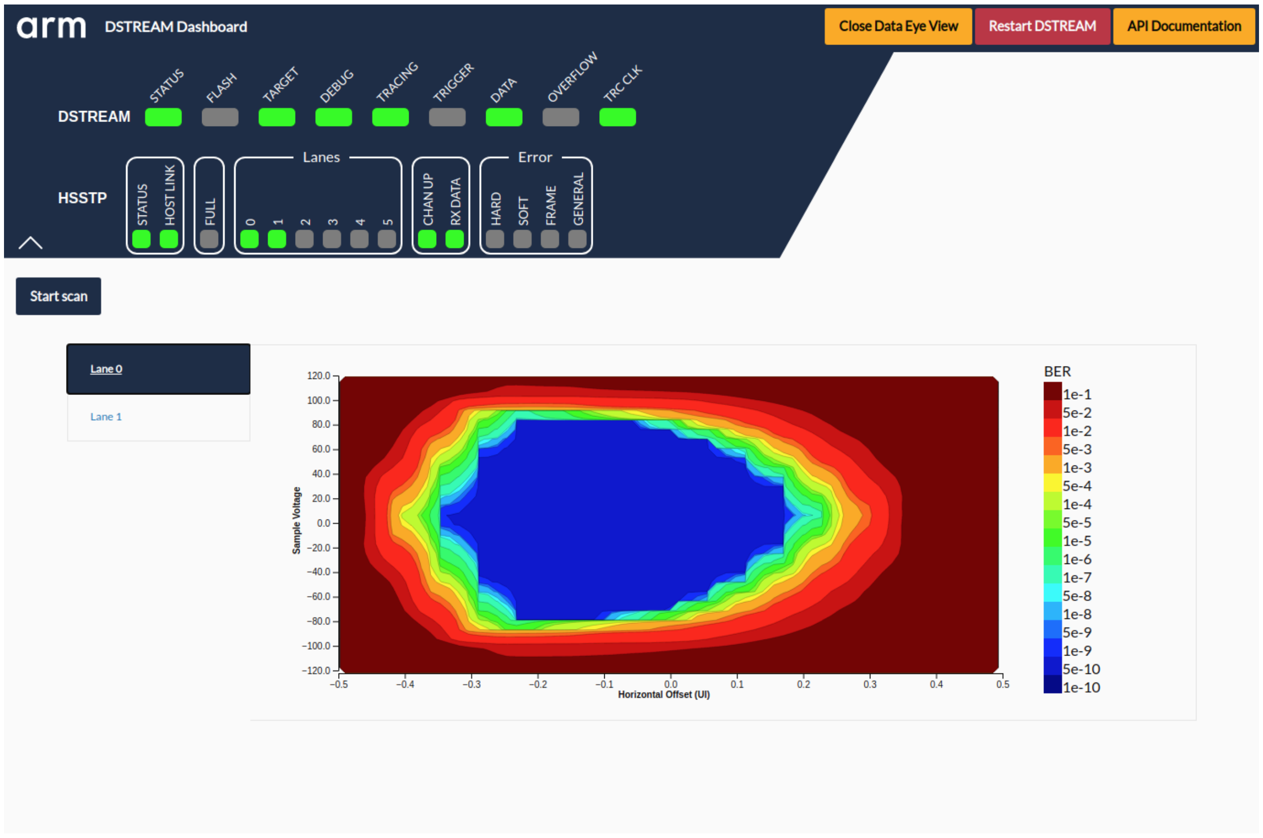
Data Eye will display when ready, this normally takes a few minutes.

After it has finished generating, the graph appears automatically on the screen.

Note

The selected Lane is highlighted dark blue.

Figure 10-41: DSTREAM-HT Data Eye View



Download Data Eye data

When the Data Eye has finished generating, you can download the data as a CSV file using the DSTREAM Web API.

Syntax:

`http://<dashboard-url>/lane<lane-number>.csv`

Returns a CSV file containing Voltage, Rx Sampling Point, and the BER used for generating BER Data Eye.

You must specify the `<lane-number>` for the data that you want to download.

Related information

DSTREAM dashboard overview on page 305
Connect to a DSTREAM unit remotely on page 307
DSTREAM Web API on page 309

11 Perspectives and Views

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

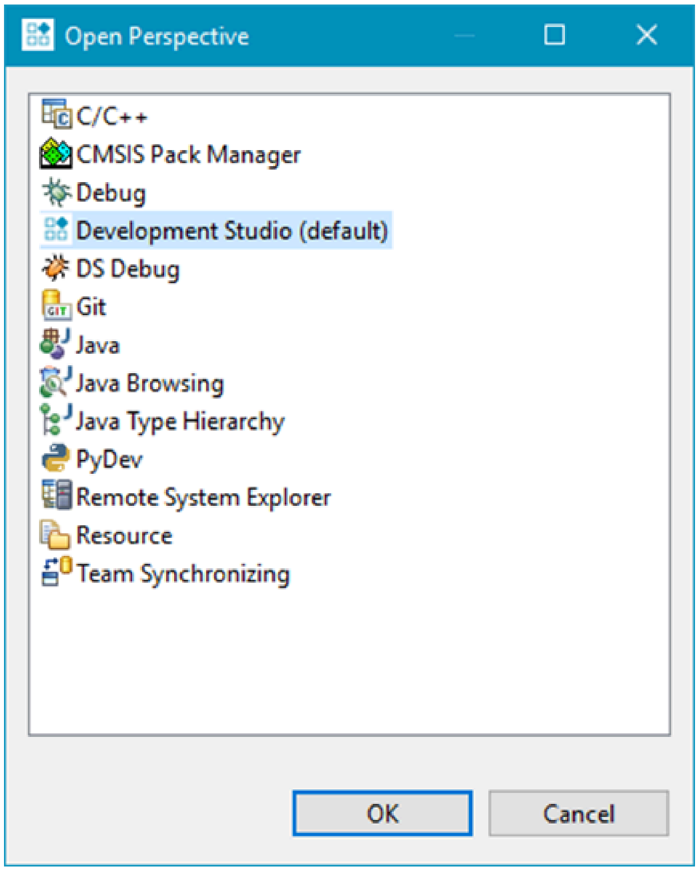
11.1 Perspectives in Arm Development Studio

Perspectives are preset configurations of views and editors in the Arm® Development Studio IDE. Each perspective has its own menus and toolbars. By default, Arm Development Studio starts in the **Development Studio** perspective.

Access other perspectives in Arm Development Studio

1. Go to **Window** > **Perspective** > **Open Perspective** > **Other...**. This opens the **Open Perspective** dialog box.
2. Select the perspective that you want to open, and click **OK**.

Figure 11-1: Open Perspective dialog box


Open Perspective

C/C++
CMSIS Pack Manager
Debug
Development Studio (default)
DS Debug
Git
Java
Java Browsing
Java Type Hierarchy
PyDev
Remote System Explorer
Resource
Team Synchronizing

Features

Table 11-1: Perspectives in Arm Development Studio

Perspective	Description
Development Studio (default)	The default perspective in Arm Development Studio. Use this perspective for managing debug tasks. In this perspective, you can do the following: • Manage projects in the **Project Explorer** view. • Manage debug connections in the **Debug Control** view. • View and modify source code files in the **Editor** window. • View debug output in the **Console** and **Target Console** views.
CMSIS-Pack Manager	Use this perspective to manage CMSIS-Packs. In this perspective, you can import packs and pack examples for various processors and boards. For more information, see the *CMSIS C/C++ Development User's Guide* at **Help** > **Help Contents** > **CMSIS C/C++ Development User's Guide**.

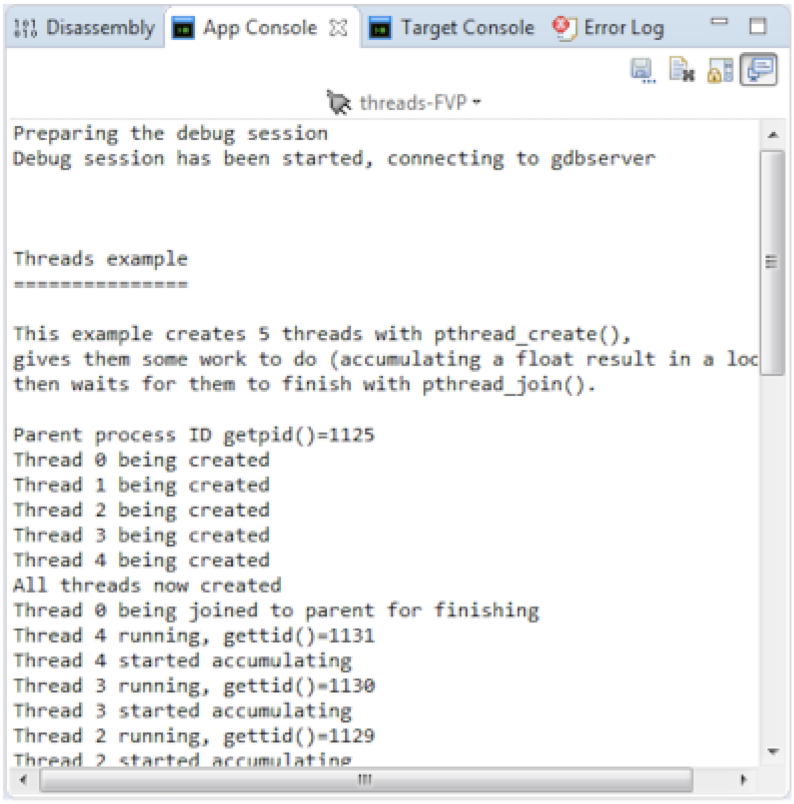
Related information

Integrated Development Environment (IDE) Overview
Using the IDE

11.2 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 11-2: App Console view



```
Preparing the debug session
Debug session has been started, connecting to gdbserver



Threads example
===============

This example creates 5 threads with pthread_create(),
gives them some work to do (accumulating a float result in a loc
then waits for them to finish with pthread_join().

Parent process ID getpid()=1125
Thread 0 being created
Thread 1 being created
Thread 2 being created
Thread 3 being created
Thread 4 being created
All threads now created
Thread 0 being joined to parent for finishing
Thread 4 running, gettid()=1131
Thread 4 started accumulating
Thread 3 running, gettid()=1130
Thread 3 started accumulating
Thread 2 running, gettid()=1129
Thread 2 started accumulating
```

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.

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 information

Using semihosting to access resources on the host computer on page 80
Working with semihosting on page 82
Perspectives and Views on page 315

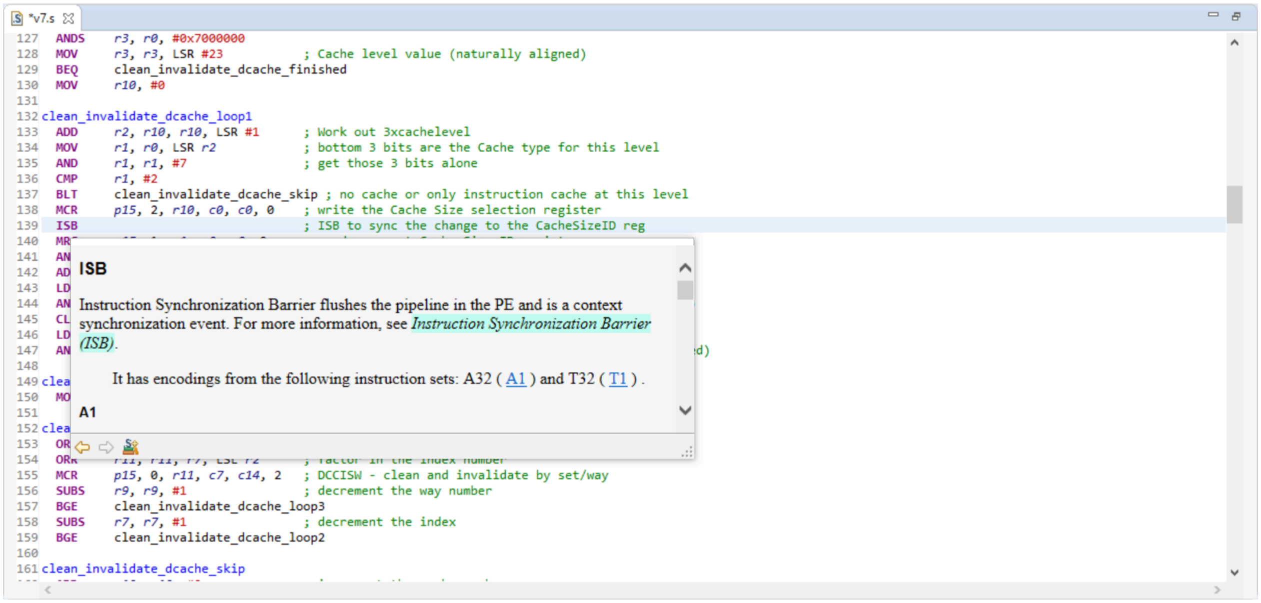
11.3 Arm Asm Info view

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 11-3: Arm Asm Info pop-up



*v7.s

```
    ANDS    r3, r0, #0x7000000
    MOV     r3, r3, LSR #23           ; Cache level value (naturally aligned)
    BEQ     clean_invalidate_dcache_finished
    MOV     r10, #0

clean_invalidate_dcache_loop1
    ADD     r2, r10, r10, LSR #1      ; Work out 3xcachelevel
    MOV     r1, r0, LSR r2            ; bottom 3 bits are the Cache type for this level
    AND     r1, r1, #7                ; get those 3 bits alone
    CMP     r1, #2
    BLT     clean_invalidate_dcache_skip ; no cache or only instruction cache at this level
    MCR     p15, 2, r10, c0, c0, 0    ; write the Cache Size selection register
    ISB                               ; ISB to sync the change to the CacheSizeID reg
    MR
    AN
    AD
    LD
    AN
    CL
    LD
    AN

clea
    MO

clea
    OR
    ORR
    MCR     p15, 0, r11, c7, c14, 2   ; DCCISW - clean and invalidate by set/way
    SUBS    r9, r9, #1                ; decrement the way number
    BGE     clean_invalidate_dcache_loop3
    SUBS    r7, r7, #1                ; decrement the index
    BGE     clean_invalidate_dcache_loop2

clean_invalidate_dcache_skip
```

ISB

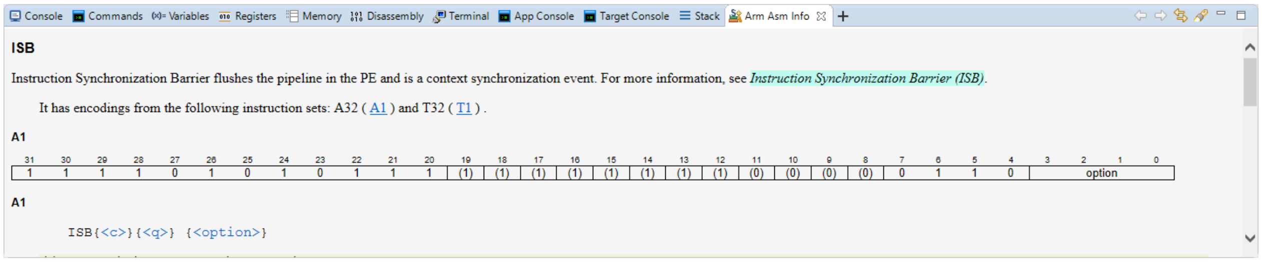
Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

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

Figure 11-4: Arm Asm Info view



ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	0	option			

A1

```
ISB{<c>}{<q>} {<option>}
```

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 information

Perspectives and Views on page 315
Arm assembler editor on page 319

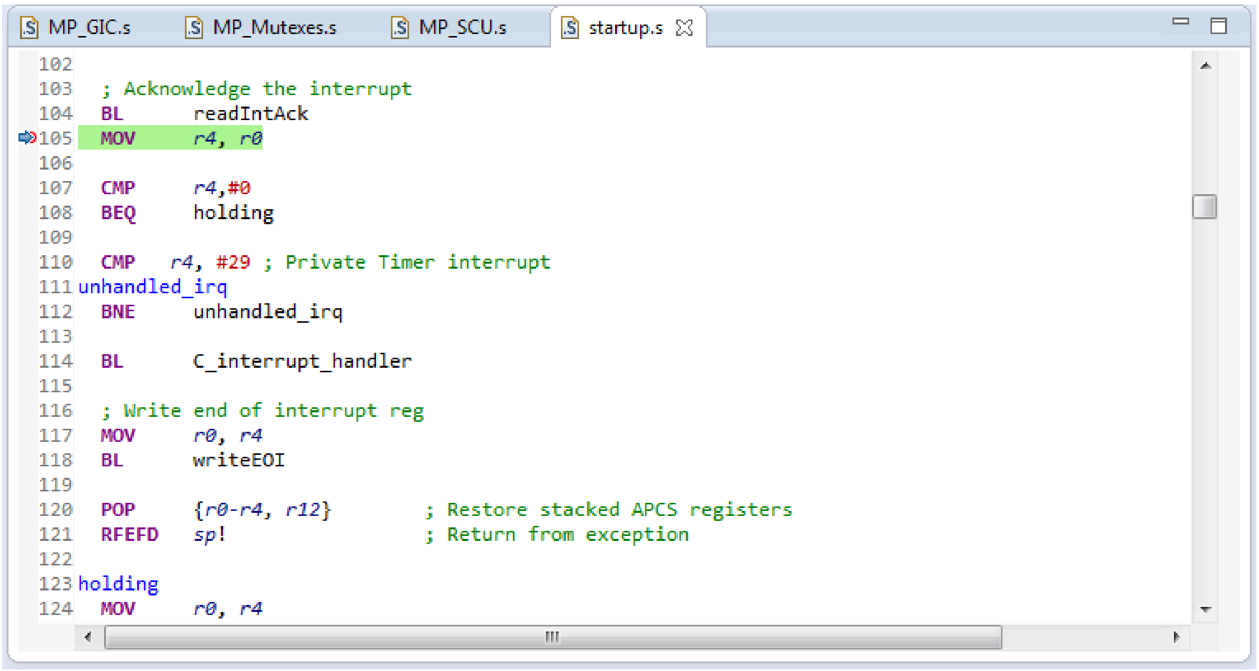
11.4 Arm assembler editor

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 11-5: Assembler editor



```

  ; Acknowledge the interrupt
  BL      readIntAck
  MOV     r4, r0

  CMP     r4,#0
  BEQ     holding

  CMP   r4, #29 ; Private Timer interrupt
unhandled_irq
  BNE     unhandled_irq

  BL      C_interrupt_handler

  ; Write end of interrupt reg
  MOV     r0, r4
  BL      writeEOI

  POP     {r0-r4, r12}         ; Restore stacked APCS registers
  RFEFD   sp!                  ; Return from exception

holding
  MOV     r0, r4
```

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.

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 information

Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
Perspectives and Views on page 315

Examining the target execution environment on page 120
Examining the call stack on page 121
Arm Asm Info view on page 318

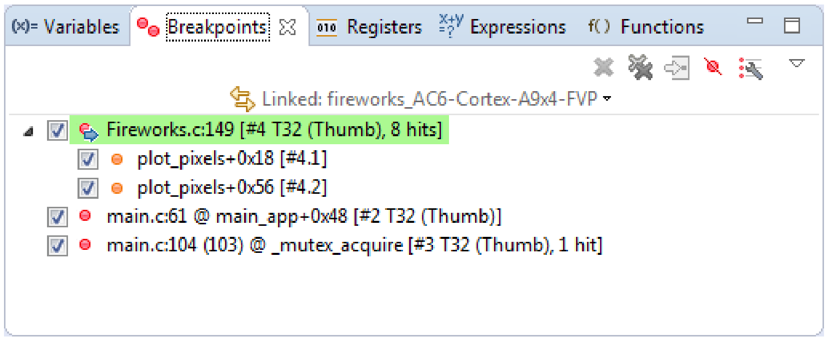
11.5 Breakpoints view

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 11-6: Breakpoints view showing breakpoints and sub-breakpoints



Linked: fireworks_AC6-Cortex-A9x4-FVP

Fireworks.c:149 [#4 T32 (Thumb), 8 hits]
plot_pixels+0x18 [#4.1]
plot_pixels+0x56 [#4.2]
main.c:61 @ main_app+0x48 [#2 T32 (Thumb)]
main.c:104 (103) @ _mutex_acquire [#3 T32 (Thumb), 1 hit]

Syntax of a breakpoint entry

A breakpoint entry has the following syntax:

```
<source_file>:<linenum> @ <function>+<offset> <address> [#<ID> <instruction_set>,
 ignore = <num>/<count>, <nHits> hits, (<condition>)]
```

where:

`<source_file>:<linenum>`

If the source file is available, the file name and line number in the file where the breakpoint is set, for example `main.c:44`.

`<function>+<offset>`

The name of the function in which the breakpoint is set and the number of bytes from the start of the function. For example, `main_app+0x12` shows that the breakpoint is 18 bytes from the start of the `main_app()` function.

`<address>`

The address at which the breakpoint is set.

`<ID>`

The breakpoint ID number, `#<N>`. In some cases, such as in a *for* loop, a breakpoint might comprise a number of sub-breakpoints. These are identified as `<N>.<n>`, where `<N>` is the number of the parent. The syntax of a sub-breakpoint entry is:

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

`<instruction_set>`

The instruction set of the instruction at the address of the breakpoint, `A32 (Arm)` or `T32 (Thumb)`.

`ignore = <num>/<count>`

An `ignore` count, if set, where:

`<num>` equals `count` initially, and decrements on each pass until it reaches zero.

`<count>` is the value you have specified for the `ignore` count.

`<nHits> hits`

A counter that increments each time the breakpoint is hit. This is not displayed until the first hit. If you set an `ignore` count, `hits` count does not start incrementing until the `ignore` count reaches zero.

`<condition>`

The stop condition you have specified.

Syntax of a watchpoint entry

A watchpoint entry has the following syntax:

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

where:

`<name>`

The name of the variable where the watchpoint is set.

`<type>`

The access type of the watchpoint.

`<address>`

The address at which the watchpoint is set.

`<ID>`

The watchpoint ID number.

Syntax of a tracepoint entry

A tracepoint entry has the following syntax:

```
<source_file>:<linenum>
```

where:

`<source_file>:<linenum>`
If the source file is available, the file name and line number in the file where the tracepoint is set.

Toolbar and context menu options

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

`Linked: <context>`
Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively, you can link the view to a different connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Delete
Removes the selected breakpoints, watchpoints, and tracepoints.

Delete All
Removes all breakpoints, watchpoints, and tracepoints.

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

Skip All Breakpoints
Deactivates all 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.

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 information

Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
About debugging multi-threaded applications on page 28
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35
About debugging TrustZone enabled targets on page 39
Perspectives and Views on page 315
Examining the target execution environment on page 120
Examining the call stack on page 121

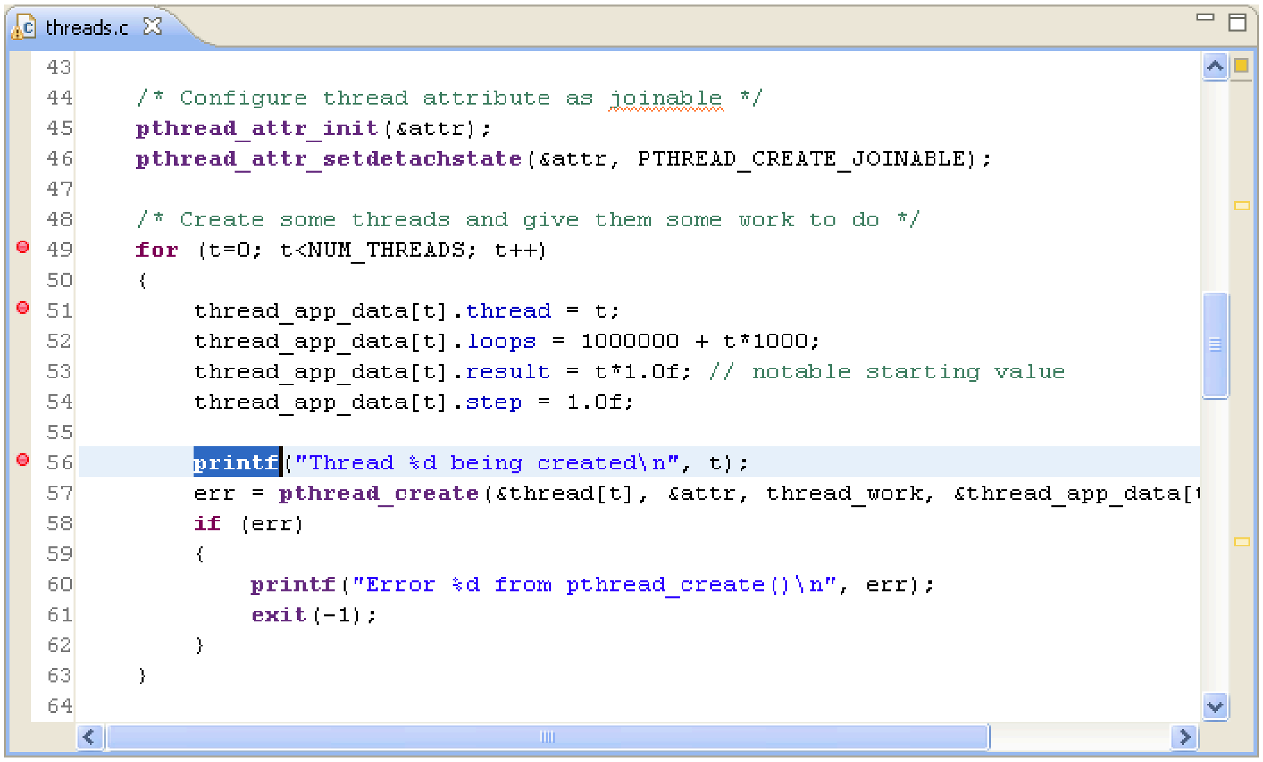
11.6 C/C++ editor

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 11-7: C/C++ editor



```

    /* Configure thread attribute as joinable */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    /* Create some threads and give them some work to do */
    for (t=0; t<NUM_THREADS; t++)
    {
        thread_app_data[t].thread = t;
        thread_app_data[t].loops = 1000000 + t*1000;
        thread_app_data[t].result = t*1.0f; // notable starting value
        thread_app_data[t].step = 1.0f;

        printf("Thread %d being created\n", t);
        err = pthread_create(&thread[t], &attr, thread_work, &thread_app_data[
        if (err)
        {
            printf("Error %d from pthread_create()\n", err);
            exit(-1);
        }
    }

```

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:

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

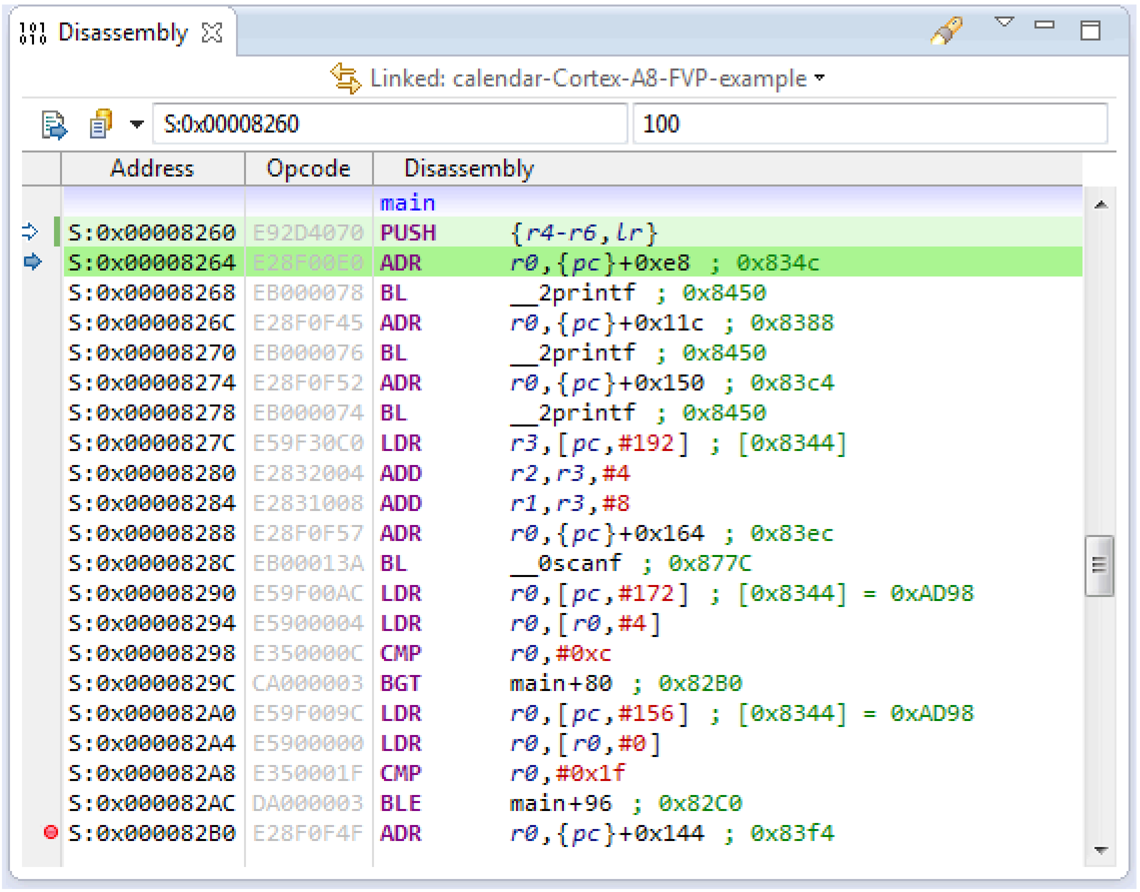
Runs to the selected source line.

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 11-8: Show disassembly for selected source line



```
Address         Opcode    Disassembly
                          main
S:0x00008260    E92D4070  PUSH     {r4-r6,lr}
S:0x00008264    E28F00E8  ADR      r0,{pc}+0xe8  ; 0x834c
S:0x00008268    EB000078  BL       __2printf  ; 0x8450
S:0x0000826C    E28F0F45  ADR      r0,{pc}+0x11c  ; 0x8388
S:0x00008270    EB000076  BL       __2printf  ; 0x8450
S:0x00008274    E28F0F52  ADR      r0,{pc}+0x150  ; 0x83c4
S:0x00008278    EB000074  BL       __2printf  ; 0x8450
S:0x0000827C    E59F30C0  LDR      r3,[pc,#192]  ; [0x8344]
S:0x00008280    E2832004  ADD      r2,r3,#4
S:0x00008284    E2831008  ADD      r1,r3,#8
S:0x00008288    E28F0F57  ADR      r0,{pc}+0x164  ; 0x83ec
S:0x0000828C    EB00013A  BL       __0scanf  ; 0x877C
S:0x00008290    E59F00AC  LDR      r0,[pc,#172]  ; [0x8344] = 0xAD98
S:0x00008294    E5900004  LDR      r0,[r0,#4]
S:0x00008298    E350000C  CMP      r0,#0xc
S:0x0000829C    CA000003  BGT      main+80  ; 0x82B0
S:0x000082A0    E59F009C  LDR      r0,[pc,#156]  ; [0x8344] = 0xAD98
S:0x000082A4    E5900000  LDR      r0,[r0,#0]
S:0x000082A8    E350001F  CMP      r0,#0x1f
S:0x000082AC    DA000003  BLE      main+96  ; 0x82C0
S:0x000082B0    E28F0F4F  ADR      r0,{pc}+0x144  ; 0x83f4
```

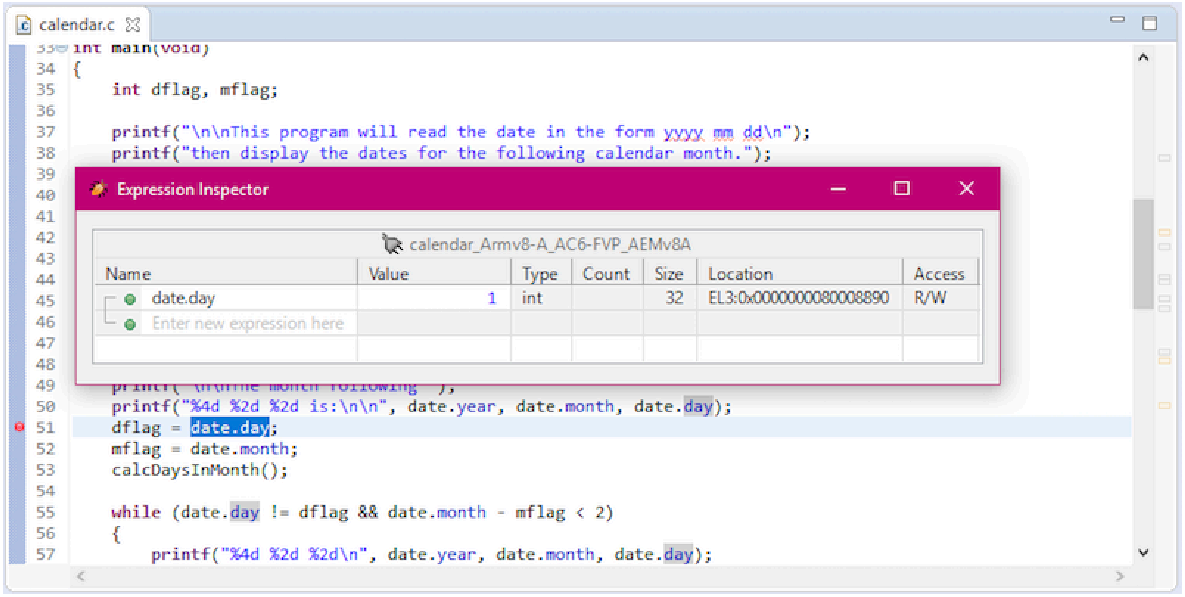
Inspect

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 11-9: Inspect the value of the highlighted variable



```
int main(void)
{
    int dflag, mflag;

    printf("\n\nThis program will read the date in the form yyyy mm dd\n");
    printf("then display the dates for the following calendar month.");
```

Name	Value	Type	Count	Size	Location	Access
date.day	1	int		32	EL3:0x0000000080008890	R/W
Enter new expression here						

```
    printf("\n\nThe month following");
    printf("%4d %2d %2d is:\n\n", date.year, date.month, date.day);
    dflag = date.day;
    mflag = date.month;
    calcDaysInMonth();

    while (date.day != dflag && date.month - mflag < 2)
    {
        printf("%4d %2d %2d\n", date.year, date.month, date.day);
```

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

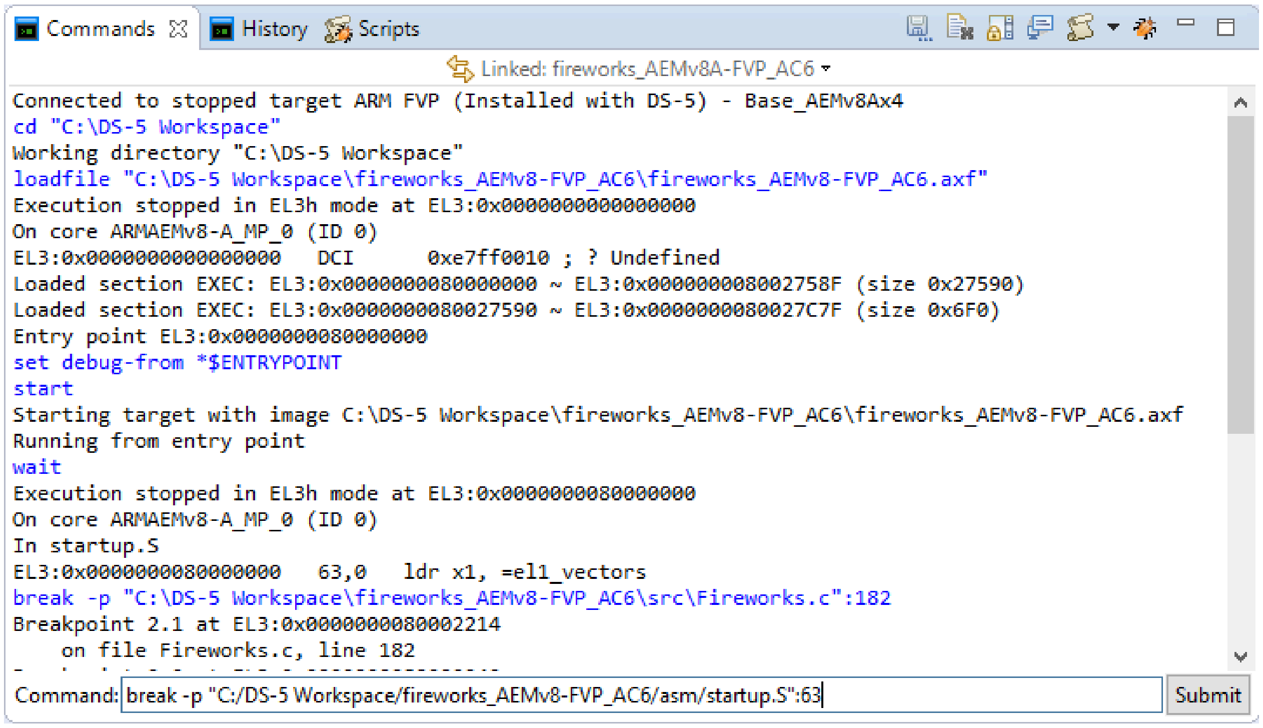
Related information

Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
Expressions view on page 350
Perspectives and Views on page 315

11.7 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 11-10: Commands view



```
Connected to stopped target ARM FVP (Installed with DS-5) - Base_AEMv8Ax4
cd "C:\DS-5 Workspace"
Working directory "C:\DS-5 Workspace"
loadfile "C:\DS-5 Workspace\fireworks_AEMv8-FVP_AC6\fireworks_AEMv8-FVP_AC6.axf"
Execution stopped in EL3h mode at EL3:0x0000000000000000
On core ARMAEMv8-A_MP_0 (ID 0)
EL3:0x0000000000000000   DCI      0xe7ff0010 ; ? Undefined
Loaded section EXEC: EL3:0x0000000080000000 ~ EL3:0x000000008002758F (size 0x27590)
Loaded section EXEC: EL3:0x0000000080027590 ~ EL3:0x0000000080027C7F (size 0x6F0)
Entry point EL3:0x0000000080000000
set debug-from *$ENTRYPOINT
start
Starting target with image C:\DS-5 Workspace\fireworks_AEMv8-FVP_AC6\fireworks_AEMv8-FVP_AC6.axf
Running from entry point
wait
Execution stopped in EL3h mode at EL3:0x0000000080000000
On core ARMAEMv8-A_MP_0 (ID 0)
In startup.S
EL3:0x0000000080000000   63,0   ldr x1, =el1_vectors
break -p "C:\DS-5 Workspace\fireworks_AEMv8-FVP_AC6\src\Fireworks.c":182
Breakpoint 2.1 at EL3:0x0000000080002214
    on file Fireworks.c, line 182
```

```
Command: break -p "C:/DS-5 Workspace/fireworks_AEMv8-FVP_AC6/asm/startup.S":63
```

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:

`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.

New Commands View

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

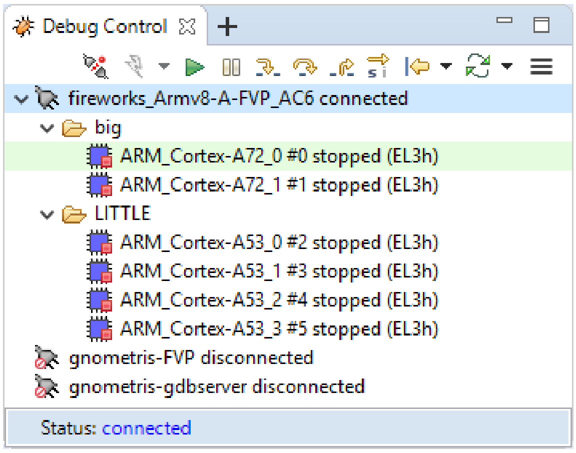
Related information

Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
About debugging multi-threaded applications on page 28
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35
About debugging TrustZone enabled targets on page 39
Perspectives and Views on page 315
Development Studio perspective keyboard shortcuts
Arm Debugger commands listed in alphabetical order

11.8 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 11-11: Debug Control view



fireworks_Armv8-A-FVP_AC6 connected
big
ARM_Cortex-A72_0 #0 stopped (EL3h)
ARM_Cortex-A72_1 #1 stopped (EL3h)
LITTLE
ARM_Cortex-A53_0 #2 stopped (EL3h)
ARM_Cortex-A53_1 #3 stopped (EL3h)
ARM_Cortex-A53_2 #4 stopped (EL3h)
ARM_Cortex-A53_3 #5 stopped (EL3h)
gnometris-FVP disconnected
gnometris-gdbserver disconnected
Status: connected

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:

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.

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.

Threads Presentation

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

Related information

Stack view on page 336
About debugging multi-threaded applications on page 28
About debugging Linux kernel modules on page 35
About debugging a Linux kernel on page 33
About debugging shared libraries on page 29

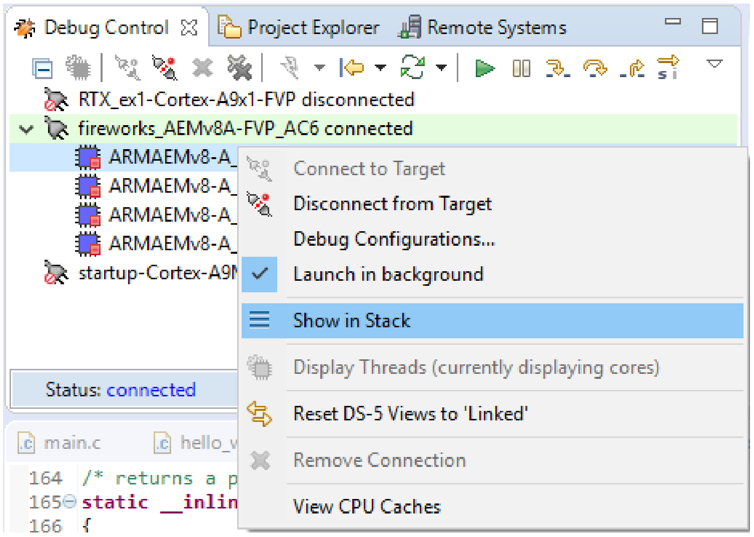
11.9 Stack view

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.

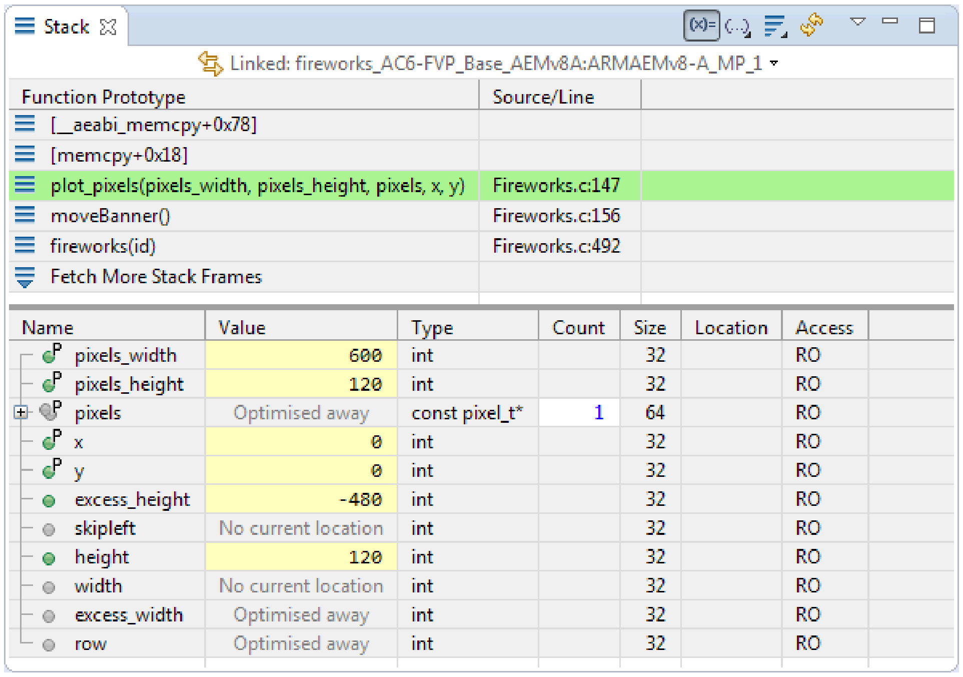
Figure 11-12: Show in Stack



```
/* returns a p
static __inlin
{
```

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

Figure 11-13: Stack view showing information for a selected core



Function Prototype	Source/Line
[__aeabi_memcpy+0x78]	
[memcpy+0x18]	
plot_pixels(pixels_width, pixels_height, pixels, x, y)	Fireworks.c:147
moveBanner()	Fireworks.c:156
fireworks(id)	Fireworks.c:492
Fetch More Stack Frames	

Name	Value	Type	Count	Size	Location	Access
pixels_width	600	int		32		RO
pixels_height	120	int		32		RO
pixels	Optimised away	const pixel_t*	1	64		RO
x	0	int		32		RO
y	0	int		32		RO
excess_height	-480	int		32		RO
skipleft	No current location	int		32		RO
height	120	int		32		RO
width	No current location	int		32		RO
excess_width	Optimised away	int		32		RO
row	Optimised away	int		32		RO

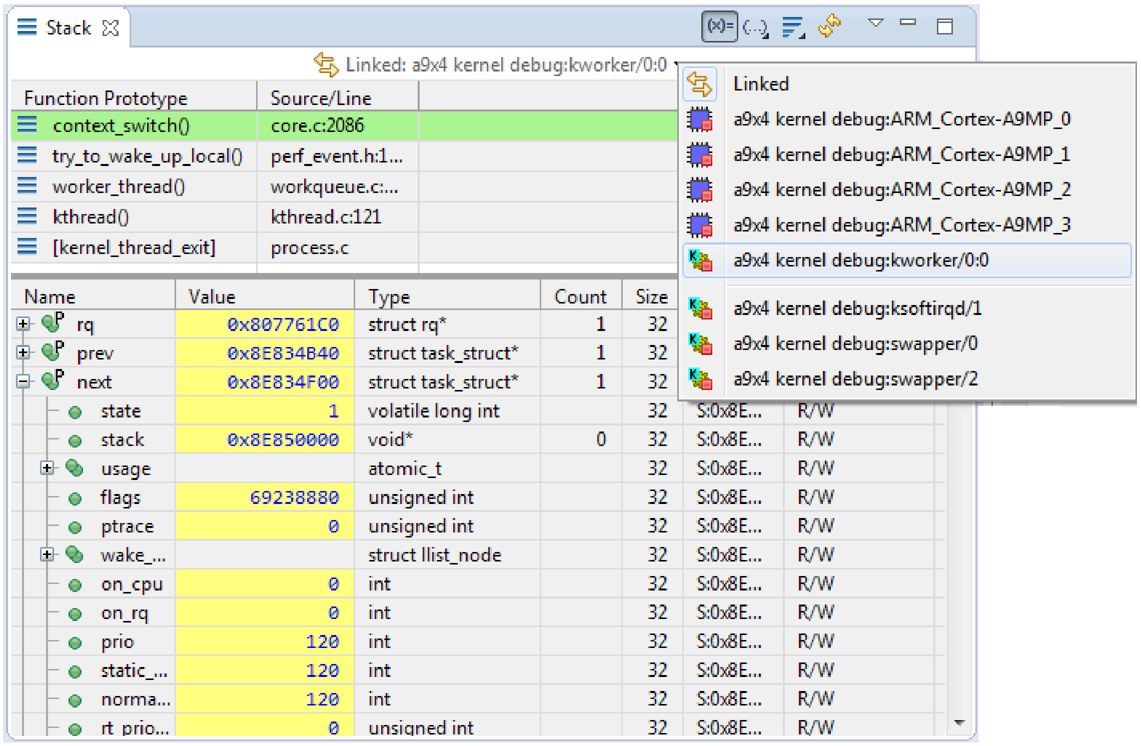
Some of the views in the **Development Studio** perspective are associated with the currently selected stack frame. Each associated view is synchronized accordingly.

You can also:

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 11-14: Stack view locked to a selected context


Stack
Linked: a9x4 kernel debug:kworker/0:0
Linked
a9x4 kernel debug:ARM_Cortex-A9MP_0
a9x4 kernel debug:ARM_Cortex-A9MP_1
a9x4 kernel debug:ARM_Cortex-A9MP_2
a9x4 kernel debug:ARM_Cortex-A9MP_3
a9x4 kernel debug:kworker/0:0
a9x4 kernel debug:ksoftirqd/1
a9x4 kernel debug:swapper/0
a9x4 kernel debug:swapper/2

Function Prototype	Source/Line
context_switch()	core.c:2086
try_to_wake_up_local()	perf_event.h:1...
worker_thread()	workqueue.c:...
kthread()	kthread.c:121
[kernel_thread_exit]	process.c

Name	Value	Type	Count	Size		
rq	0x807761C0	struct rq*	1	32		
prev	0x8E834B40	struct task_struct*	1	32		
next	0x8E834F00	struct task_struct*	1	32		
state	1	volatile long int		32	S:0x8E...	R/W
stack	0x8E850000	void*	0	32	S:0x8E...	R/W
usage		atomic_t		32	S:0x8E...	R/W
flags	69238880	unsigned int		32	S:0x8E...	R/W
ptrace	0	unsigned int		32	S:0x8E...	R/W
wake_...		struct llist_node		32	S:0x8E...	R/W
on_cpu	0	int		32	S:0x8E...	R/W
on_rq	0	int		32	S:0x8E...	R/W
prio	120	int		32	S:0x8E...	R/W
static_...	120	int		32	S:0x8E...	R/W
norma...	120	int		32	S:0x8E...	R/W
rt_prio...	0	unsigned int		32	S:0x8E...	R/W

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 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.

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 information

Debug Control view on page 332

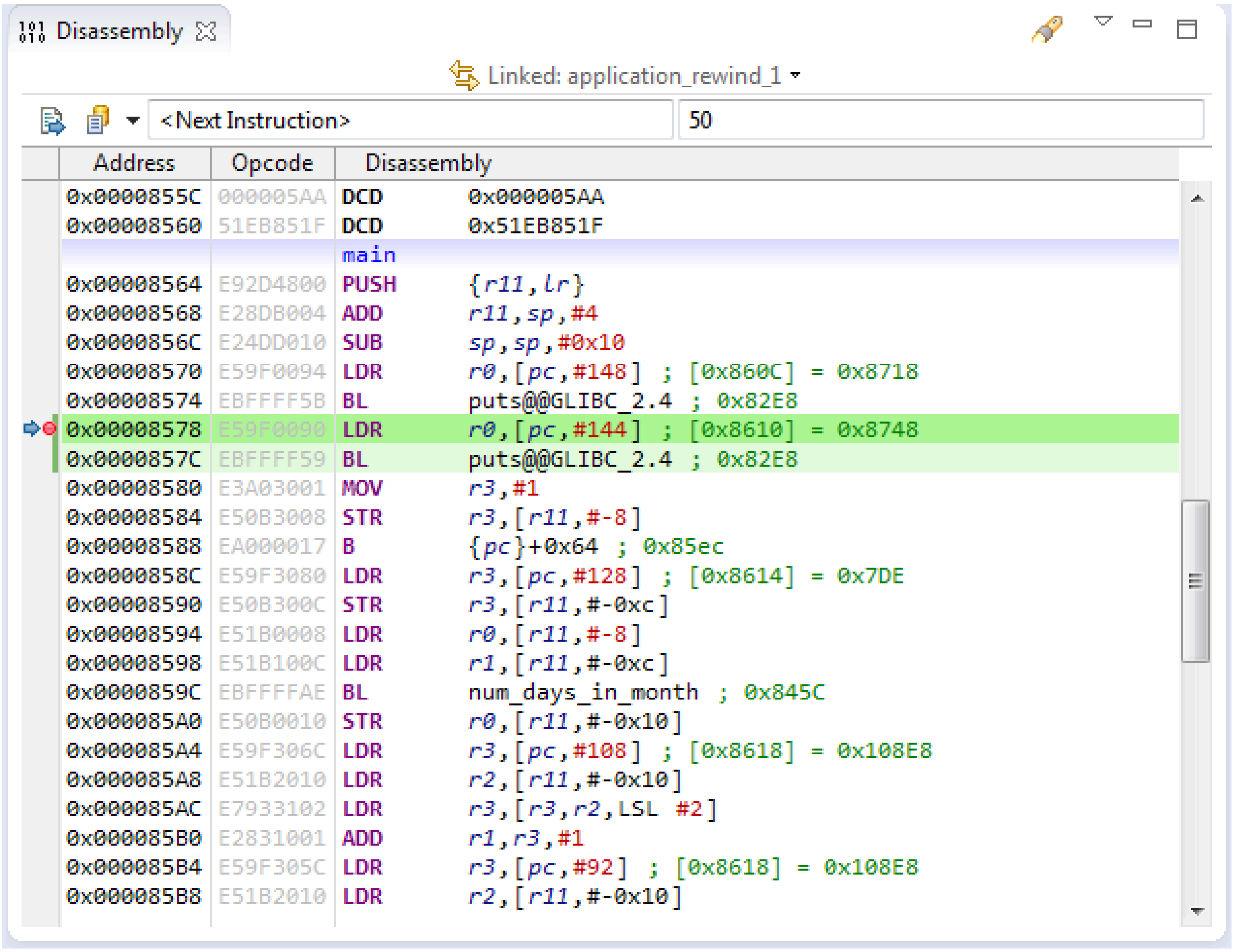
11.10 Disassembly view

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 11-15: Disassembly view



```
Address     Opcode    Disassembly
0x0000855C  000005AA  DCD      0x000005AA
0x00008560  51EB851F  DCD      0x51EB851F
                      main
0x00008564  E92D4800  PUSH     {r11,lr}
0x00008568  E28DB004  ADD      r11,sp,#4
0x0000856C  E24DD010  SUB      sp,sp,#0x10
0x00008570  E59F0094  LDR      r0,[pc,#148] ; [0x860C] = 0x8718
0x00008574  EBFFFF5B  BL       puts@@GLIBC_2.4 ; 0x82E8
0x00008578  E59F0090  LDR      r0,[pc,#144] ; [0x8610] = 0x8748
0x0000857C  EBFFFF59  BL       puts@@GLIBC_2.4 ; 0x82E8
0x00008580  E3A03001  MOV      r3,#1
0x00008584  E50B3008  STR      r3,[r11,#-8]
0x00008588  EA000017  B        {pc}+0x64 ; 0x85ec
0x0000858C  E59F3080  LDR      r3,[pc,#128] ; [0x8614] = 0x7DE
0x00008590  E50B300C  STR      r3,[r11,#-0xc]
0x00008594  E51B0008  LDR      r0,[r11,#-8]
0x00008598  E51B100C  LDR      r1,[r11,#-0xc]
0x0000859C  EBFFFFAE  BL       num_days_in_month ; 0x845C
0x000085A0  E50B0010  STR      r0,[r11,#-0x10]
0x000085A4  E59F306C  LDR      r3,[pc,#108] ; [0x8618] = 0x108E8
0x000085A8  E51B2010  LDR      r2,[r11,#-0x10]
0x000085AC  E7933102  LDR      r3,[r3,r2,LSL #2]
0x000085B0  E2831001  ADD      r1,r3,#1
0x000085B4  E59F305C  LDR      r3,[pc,#92] ; [0x8618] = 0x108E8
0x000085B8  E51B2010  LDR      r2,[r11,#-0x10]
```

Gradient shading in the **Disassembly** view shows the start of each function.

Solid shading in the **Disassembly** view shows the instruction at the address of the current PC register followed by any related instructions that correspond to the current source line.

In the left-hand margin of the **Disassembly** view you can find a marker bar that displays view markers associated with specific locations in the disassembly code.

To set a breakpoint, double-click in the marker bar at the position where you want to set the breakpoint. To delete a breakpoint, double-click on the breakpoint marker.

Note

If you have sub-breakpoints to a parent breakpoint then double-clicking on the marker also deletes the related sub-breakpoints.

Toolbar and context menu options

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

`Linked: context`

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a different connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Next Instruction

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

History

Addresses and expressions you specify in the **Address** field are added to the drop down box, and persist until you clear the history list or exit Eclipse. If you want to keep an expression for later use, add it to the **Expressions** view.

Address field

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

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.

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 information

Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
About debugging multi-threaded applications on page 28
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33

About debugging Linux kernel modules on page 35
About debugging TrustZone enabled targets on page 39
Perspectives and Views on page 315

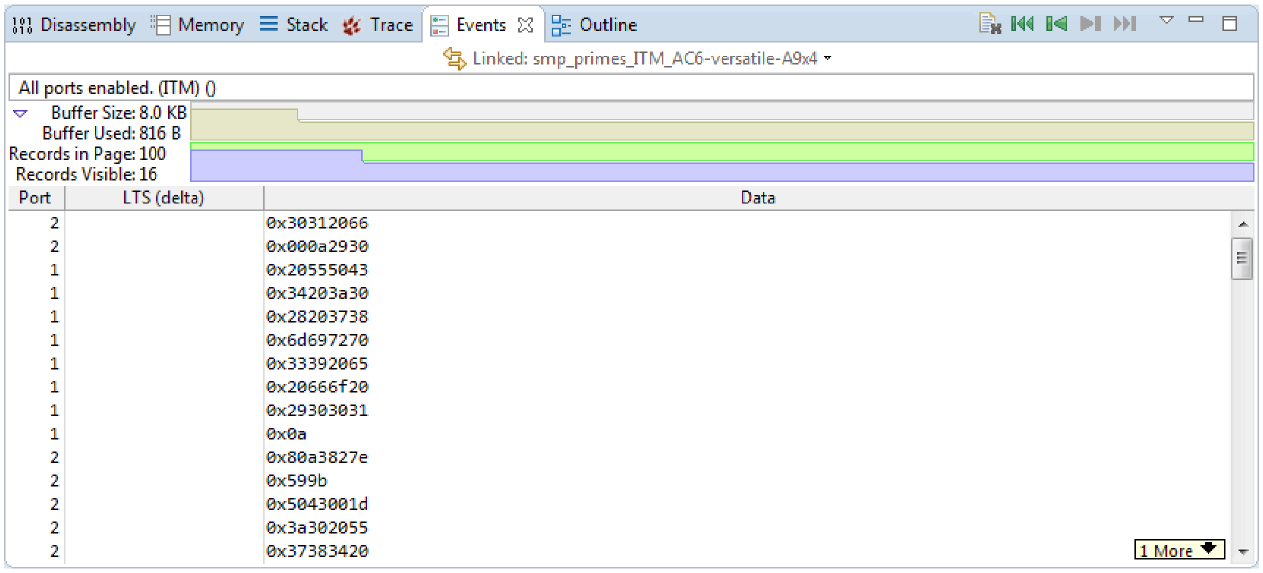
11.11 Events view

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 11-16: Events view (Shown with all ports enabled for an ETB:ITM trace source)



Linked: smp_primes_ITM_AC6-versatile-A9x4

All ports enabled. (ITM) ()

Buffer Size: 8.0 KB
Buffer Used: 816 B
Records in Page: 100
Records Visible: 16

Port	LTS (delta)	Data
2		0x30312066
2		0x000a2930
1		0x20555043
1		0x34203a30
1		0x28203738
1		0x6d697270
1		0x33392065
1		0x20666f20
1		0x29303031
1		0x0a
2		0x80a3827e
2		0x599b
2		0x5043001d
2		0x3a302055
2		0x37383420

Note

- 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 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.

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 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.

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 information

Perspectives and Views on page 315
Event Viewer Settings dialog box on page 346
CoreSight
CoreSight System Trace Macrocell Technical Reference Manual
Armv7-M Architecture Reference Manual Documentation

11.12 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.

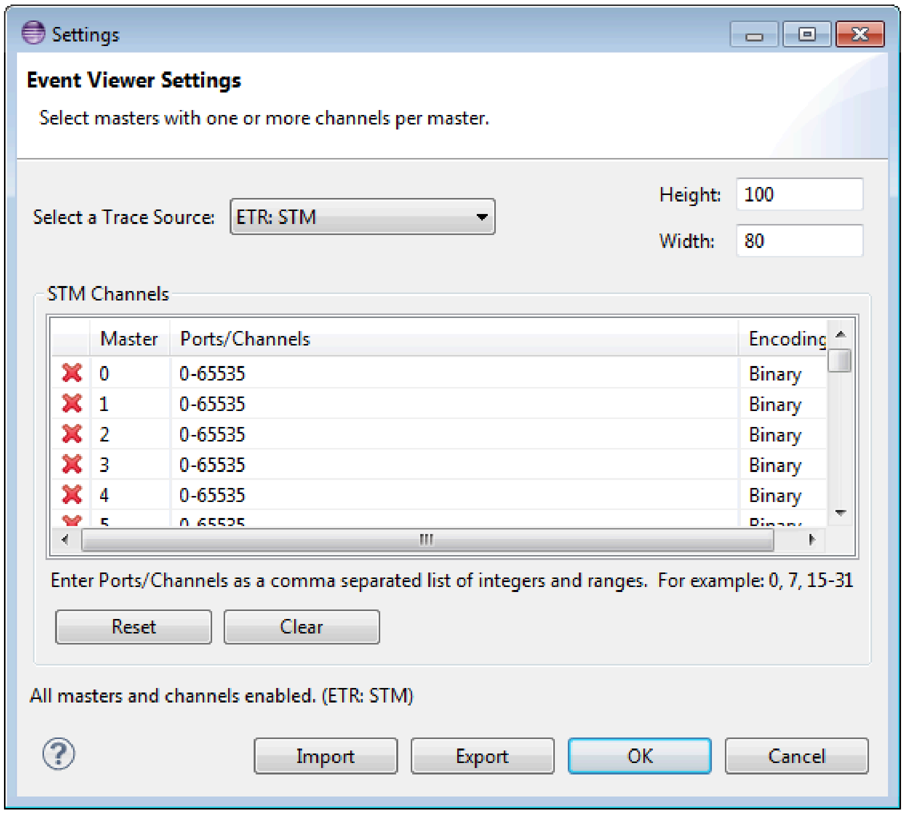
OK

Reorganizes the current channels into a canonical form, saves the settings, and closes the dialog box.

Cancel

Enables you to cancel unsaved changes.

Figure 11-17: Event Viewer Settings (Shown with all Masters and Channels enabled for an ETR:STM trace source)



Settings

Event Viewer Settings

Select masters with one or more channels per master.

Select a Trace Source: ETR: STM

Height: 100

Width: 80

STM Channels

	Master	Ports/Channels	Encoding
✖	0	0-65535	Binary
✖	1	0-65535	Binary
✖	2	0-65535	Binary
✖	3	0-65535	Binary
✖	4	0-65535	Binary

Enter Ports/Channels as a comma separated list of integers and ranges. For example: 0, 7, 15-31

All masters and channels enabled. (ETR: STM)

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.

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.

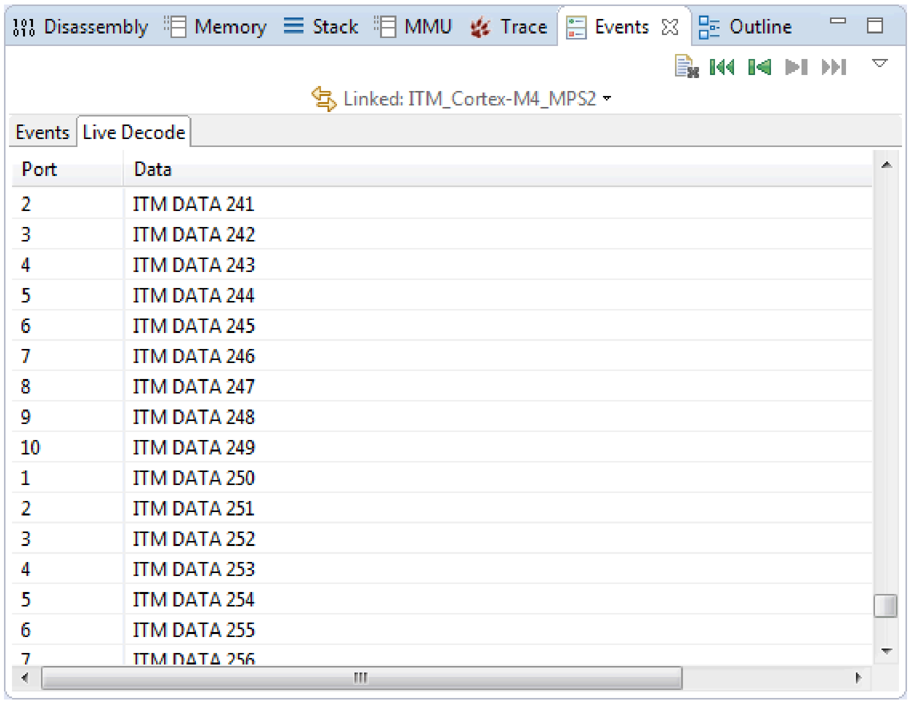
Show data tracing (M-profile only)

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 11-18: Events view - Live Decode tab



Port	Data
2	ITM DATA 241
3	ITM DATA 242
4	ITM DATA 243
5	ITM DATA 244
6	ITM DATA 245
7	ITM DATA 246
8	ITM DATA 247
9	ITM DATA 248
10	ITM DATA 249
1	ITM DATA 250
2	ITM DATA 251
3	ITM DATA 252
4	ITM DATA 253
5	ITM DATA 254
6	ITM DATA 255
7	ITM DATA 256

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.

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**.

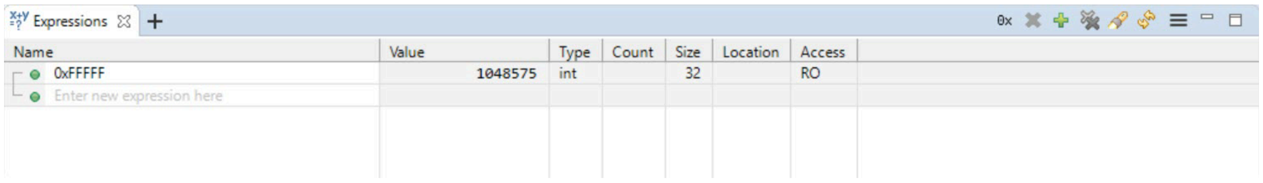
Related information

Events view on page 344
CoreSight Components Technical Reference Manual
CoreSight System Trace Macrocell Technical Reference Manual
Armv7-M Architecture Reference Manual Documentation

11.13 Expressions view

Use the **Expressions** view to create and work with expressions.

Figure 11-19: Expressions view



Name	Value	Type	Count	Size	Location	Access
0xFFFFF	1048575	int		32		RO
Enter new expression here						

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

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.

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.

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 Arm®v8 in SVC mode, with 8K stack size:

```
(struct thread_info*)($SP_SVC &~0x1FFF)
```

For Armv8 AArch64 in EL1, with 16K stack size:

```
(struct thread_info*)($SP_EL1 &~0x3FFF)
```

Related information

Expression Inspector on page 354
Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
C/C++ editor on page 326
Perspectives and Views on page 315

11.14 Expression Inspector

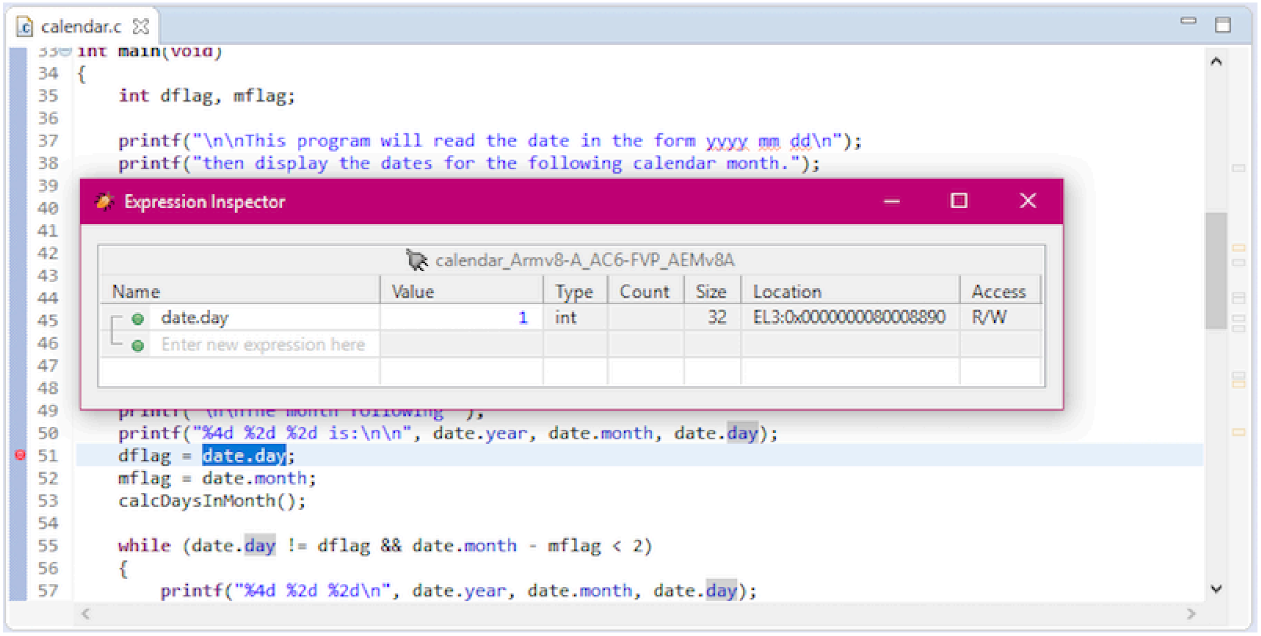
The **Expression Inspector** is a light-weight version of the **Expressions** View. It allows you to monitor the values of the highlighted variables or expressions, and manually enter variables or expressions to monitor.

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 11-20: Inspect the value of the highlighted variable



```
int main(void)
{
    int dflag, mflag;

    printf("\n\nThis program will read the date in the form yyyy mm dd\n");
    printf("then display the dates for the following calendar month.");
```

Name	Value	Type	Count	Size	Location	Access
date.day	1	int		32	EL3:0x0000000080008890	R/W
Enter new expression here						

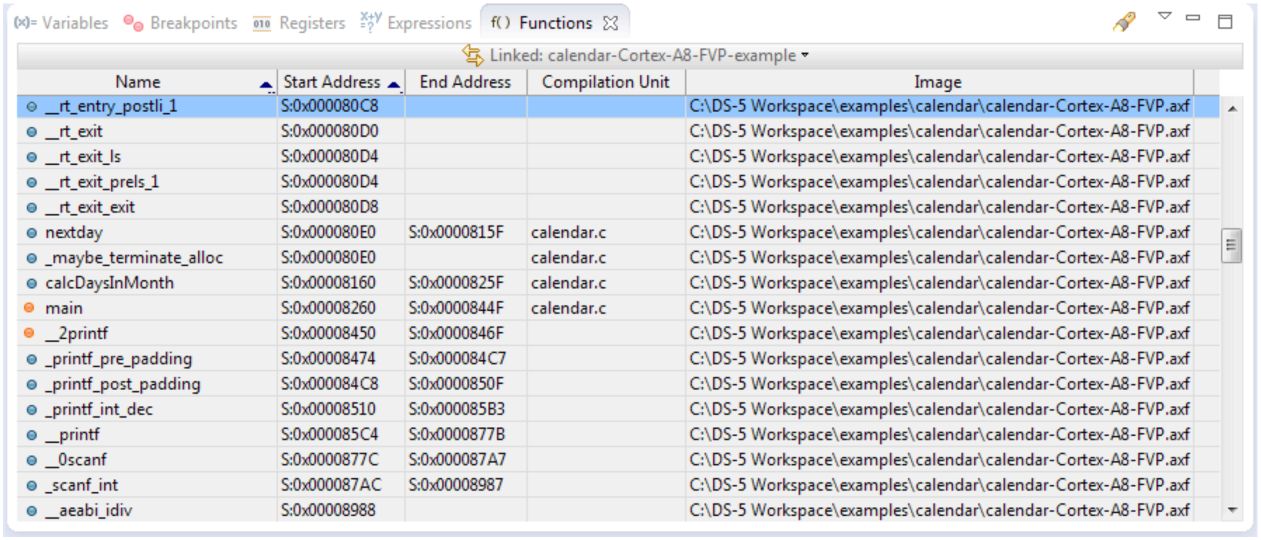
```
    printf("\n\nThe month following");
    printf("%4d %2d %2d is:\n\n", date.year, date.month, date.day);
    dflag = date.day;
    mflag = date.month;
    calcDaysInMonth();

    while (date.day != dflag && date.month - mflag < 2)
    {
        printf("%4d %2d %2d\n", date.year, date.month, date.day);
```

11.15 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.

Figure 11-21: Functions view



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

Right-click on the column headers to select the columns that you want displayed:

Name

The name of the function.

Mangled Name

The C++ mangled name of the function.

Base Address

The function entry point.

Start Address

The start address of the function.

End Address

The end address of the function.

Size

The size of the function in bytes.

Compilation Unit

The name of the compilation unit containing the function.

Image

The location of the ELF image containing the function.

Show All Columns

Displays all columns.

Reset Columns

Resets the columns displayed and their widths to the default. The `Name`, `StartAddress`, `End Address`, `Compilation Unit`, and `Image` columns are displayed by default.

In the **Functions** view, the functions are represented as:

Table 11-2: Function icons

Icon	Description	Icon	Description
●	Function	●	Function with a breakpoint set
S	Static function	S	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.

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 information

Perspectives and Views on page 315

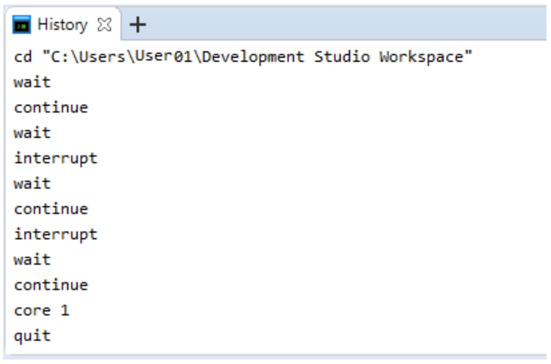
11.16 History view

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 11-22: History view



```
cd "C:\Users\User01\Development Studio Workspace"
wait
continue
wait
interrupt
wait
continue
interrupt
wait
continue
core 1
quit
```

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.

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 information

Perspectives and Views on page 315

11.17 Memory view

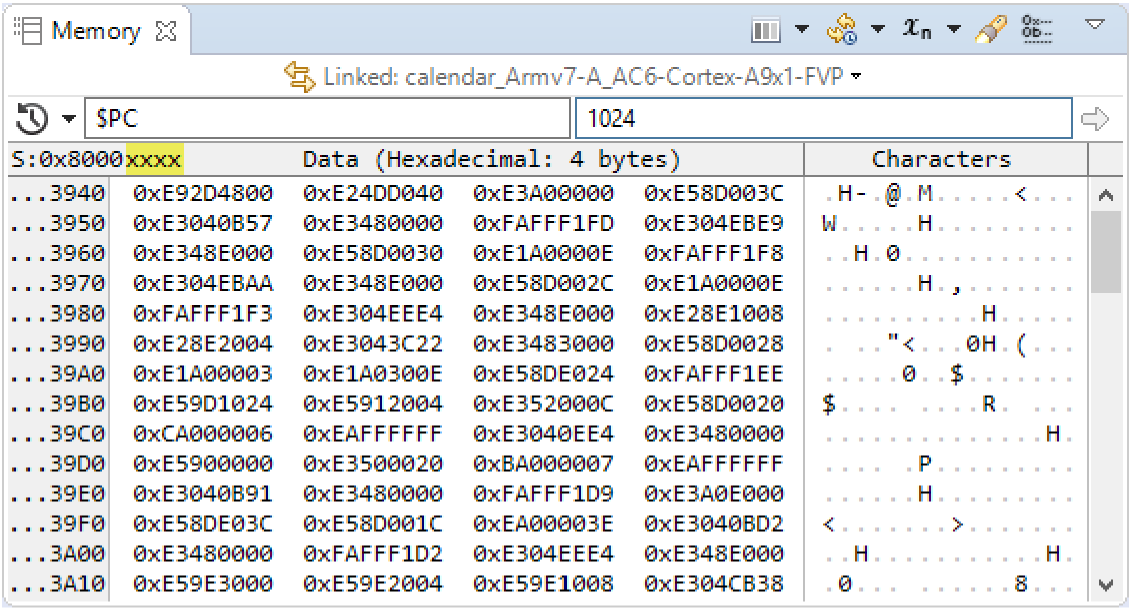
Use the **Memory** view to display and modify the contents of memory.

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 11-23: Memory view



```
$PC                                   1024
S:0x8000xxxx   Data (Hexadecimal: 4 bytes)                    Characters
...3940  0xE92D4800  0xE24DD040  0xE3A00000  0xE58D003C   .H-.@.M.....<...
...3950  0xE3040B57  0xE3480000  0xFAFFF1FD  0xE304EBE9   W.....H.........
...3960  0xE348E000  0xE58D0030  0xE1A0000E  0xFAFFF1F8   ..H.0...........
...3970  0xE304EBAA  0xE348E000  0xE58D002C  0xE1A0000E   ......H.,.......
...3980  0xFAFFF1F3  0xE304EEE4  0xE348E000  0xE28E1008   ..........H.....
...3990  0xE28E2004  0xE3043C22  0xE3483000  0xE58D0028   . .."<...0H.(...
...39A0  0xE1A00003  0xE1A0300E  0xE58DE024  0xFAFFF1EE   .....0..$.......
...39B0  0xE59D1024  0xE5912004  0xE352000C  0xE58D0020   $.... ....R. ...
...39C0  0xCA000006  0xEAFFFFFF  0xE3040EE4  0xE3480000   ..............H.
...39D0  0xE5900000  0xE3500020  0xBA000007  0xEAFFFFFF   .... .P.........
...39E0  0xE3040B91  0xE3480000  0xFAFFF1D9  0xE3A0E000   ......H.........
...39F0  0xE58DE03C  0xE58D001C  0xEA00003E  0xE3040BD2   <.......>.......
...3A00  0xE3480000  0xFAFFF1D2  0xE304EEE4  0xE348E000   ..H...........H.
...3A10  0xE59E3000  0xE59E2004  0xE59E1008  0xE304CB38   .0... ......8...
```

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.

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 dialog box where you can specify these options as default.

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`.

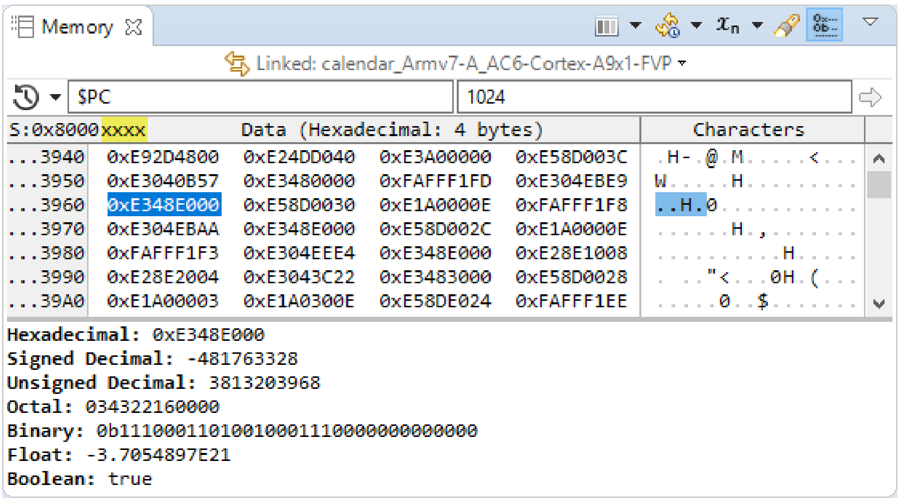
Search

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 11-24: Memory view with details panel



```
S:0x8000xxxx         Data (Hexadecimal: 4 bytes)               Characters
...3940  0xE92D4800  0xE24DD040  0xE3A00000  0xE58D003C   .H-.@.M.....<...
...3950  0xE3040B57  0xE3480000  0xFAFFF1FD  0xE304EBE9   W.....H.........
...3960  0xE348E000  0xE58D0030  0xE1A0000E  0xFAFFF1F8   ..H.0...........
...3970  0xE304EBAA  0xE348E000  0xE58D002C  0xE1A0000E   ......H.,.......
...3980  0xFAFFF1F3  0xE304EEE4  0xE348E000  0xE28E1008   ..........H.....
...39A0  0xE1A00003  0xE1A0300E  0xE58DE024  0xFAFFF1EE   ......0..$......
```

```
Hexadecimal: 0xE348E000
Signed Decimal: -481763328
Unsigned Decimal: 3813203968
Octal: 034322160000
Binary: 0b11100011010010001110000000000000
Float: -3.7054897E21
Boolean: true
```

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.

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

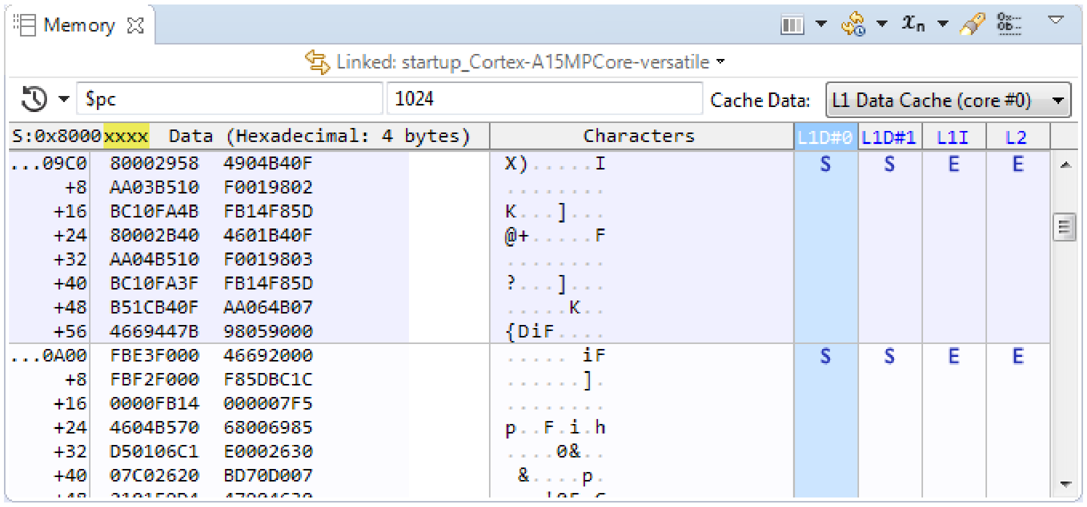
Shows the least significant bytes of the address that are not repeating.

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 11-25: Memory view with Show Cache option enabled



```
$pc          1024          Cache Data: L1 Data Cache (core #0)

S:0x8000xxxx  Data (Hexadecimal: 4 bytes)   Characters   L1D#0 L1D#1 L1I L2
...09C0   80002958   4904B40F               X)......I    S     S     E   E
     +8   AA03B510   F0019802               .........
    +16   BC10FA4B   FB14F85D               K...]...
    +24   80002B40   4601B40F               @+......F
    +32   AA04B510   F0019803               .........
    +40   BC10FA3F   FB14F85D               ?...]...
    +48   B51CB40F   AA064B07               ......K..
    +56   4669447B   98059000               {DiF....
...0A00   FBE3F000   46692000               ......iF     S     S     E   E
     +8   FBF2F000   F85DBC1C               .......].
    +16   0000FB14   000007F5               .........
    +24   4604B570   68006985               p..F.i.h
    +32   D50106C1   E0002630               ....0&..
    +40   07C02620   BD70D007                &....p.
```

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.

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.

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>

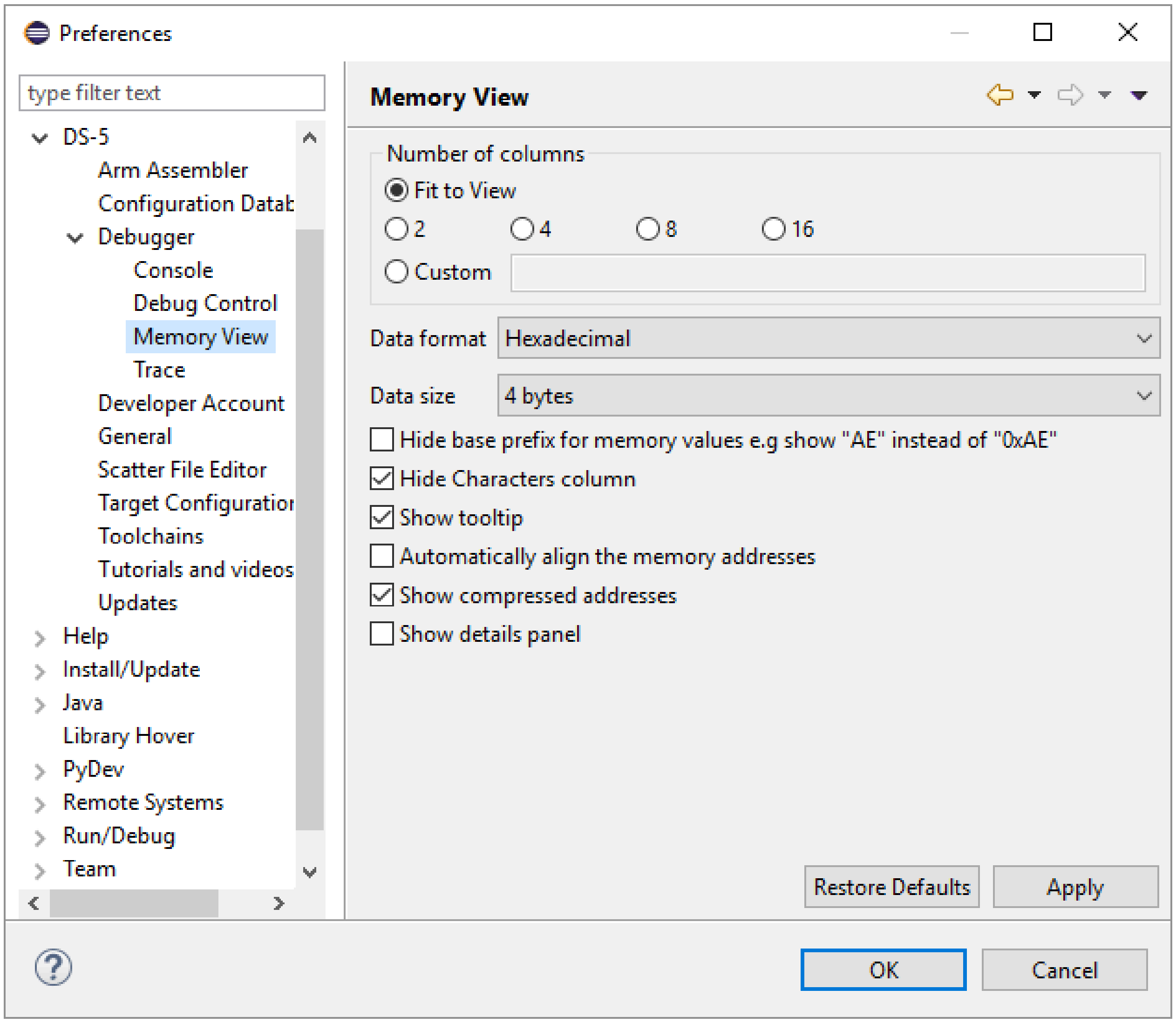
Displays the **MMU** view and translates the address of the selected value in memory.

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 11-26: Memory View preferences



Memory View

Number of columns
[✓ Fit to View] 2 4 8 16 Custom

Data format: Hexadecimal

Data size: 4 bytes

- [ ] Hide base prefix for memory values e.g show "AE" instead of "0xAE"
- [x] Hide Characters column
- [x] Show tooltip
- [ ] Automatically align the memory addresses
- [x] Show compressed addresses
- [ ] Show details panel

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.

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:

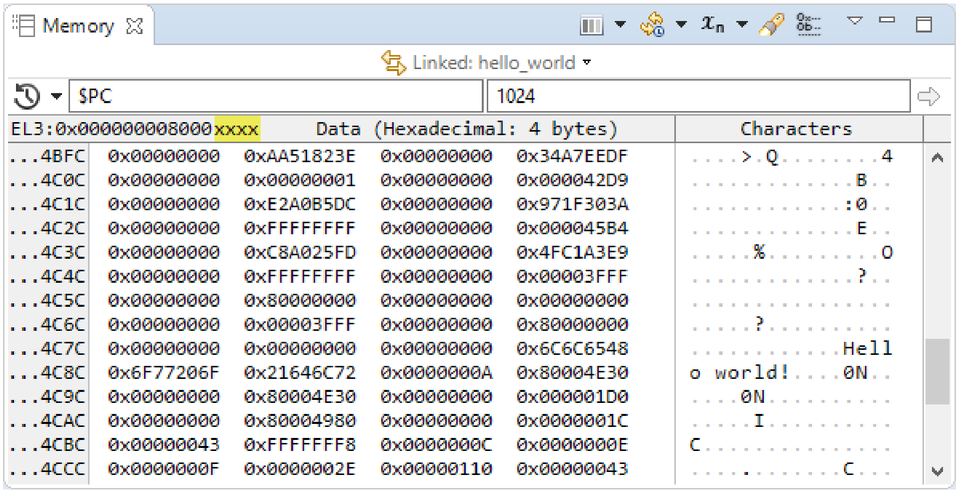
- For hexadecimal values, this option hides the preceding **0x**. So the value **0xEA000016** is shown as **EA000016**.
- For binary values, this option hides the preceding **0b**. So the value **0b00010110** is shown as **00010110**.
- For octal values, this option hides the preceding **0**. So the value **035200000026** is shown as **35200000026**.

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**

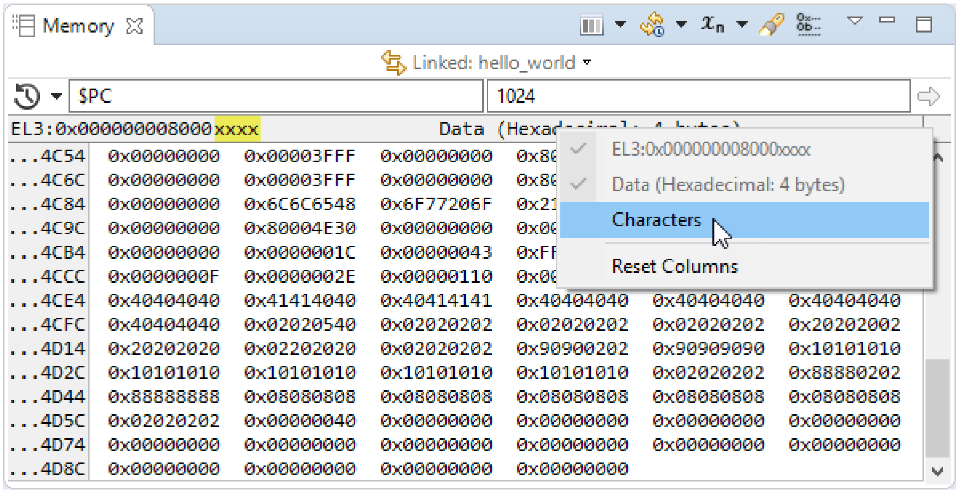
Figure 11-27: Show Characters column in the Memory view



```
EL3:0x000000008000xxxx     Data (Hexadecimal: 4 bytes)             Characters
...4BFC  0x00000000  0xAA51823E  0x00000000  0x34A7EEDF    ....>.Q........4
...4C0C  0x00000000  0x00000001  0x00000000  0x000042D9    .............B..
...4C1C  0x00000000  0xE2A0B5DC  0x00000000  0x971F303A    ............:0..
...4C2C  0x00000000  0xFFFFFFFF  0x00000000  0x000045B4    .............E..
...4C3C  0x00000000  0xC8A025FD  0x00000000  0x4FC1A3E9    .....%.........O
...4C4C  0x00000000  0xFFFFFFFF  0x00000000  0x00003FFF    .............?..
...4C5C  0x00000000  0x80000000  0x00000000  0x00000000    ................
...4C6C  0x00000000  0x00003FFF  0x00000000  0x80000000    .....?..........
...4C7C  0x00000000  0x00000000  0x00000000  0x6C6C6548    ............Hell
...4C8C  0x6F77206F  0x21646C72  0x0000000A  0x80004E30    o world!....0N..
...4C9C  0x00000000  0x80004E30  0x00000000  0x000001D0    ....0N..........
...4CAC  0x00000000  0x80004980  0x00000000  0x0000001C    .....I..........
...4CBC  0x00000043  0xFFFFFFF8  0x0000000C  0x0000000E    C...............
...4CCC  0x0000000F  0x0000002E  0x00000110  0x00000043    ............C...
```

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 11-28: Select Characters from context menu to show the ASCII characters



```
EL3:0x000000008000xxxx                Data (Hexad
...4C54  0x00000000  0x00003FFF  0x00000000  0x80
...4C6C  0x00000000  0x00003FFF  0x00000000  0x80
...4C84  0x00000000  0x6C6C6548  0x6F77206F  0x21
...4C9C  0x00000000  0x80004E30  0x00000000  0x00
...4CB4  0x00000000  0x0000001C  0x00000043  0xFF
...4CCC  0x0000000F  0x0000002E  0x00000110  0x00
...4CE4  0x40404040  0x41414040  0x40414141  0x40404040  0x40404040  0x40404040
...4CFC  0x40404040  0x02020540  0x02020202  0x02020202  0x02020202  0x20202002
...4D14  0x20202020  0x02202020  0x02020202  0x90900202  0x90909090  0x10101010
...4D2C  0x10101010  0x10101010  0x10101010  0x10101010  0x02020202  0x88880202
...4D44  0x88888888  0x08080808  0x08080808  0x08080808  0x08080808  0x08080808
...4D5C  0x02020202  0x00000040  0x00000000  0x00000000  0x00000000  0x00000000
...4D74  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
...4D8C  0x00000000  0x00000000  0x00000000  0x00000000
```

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.

Related information

Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
About debugging multi-threaded applications on page 28
About debugging shared libraries on page 29
About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35
About debugging TrustZone enabled targets on page 39
Perspectives and Views on page 315

11.18 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

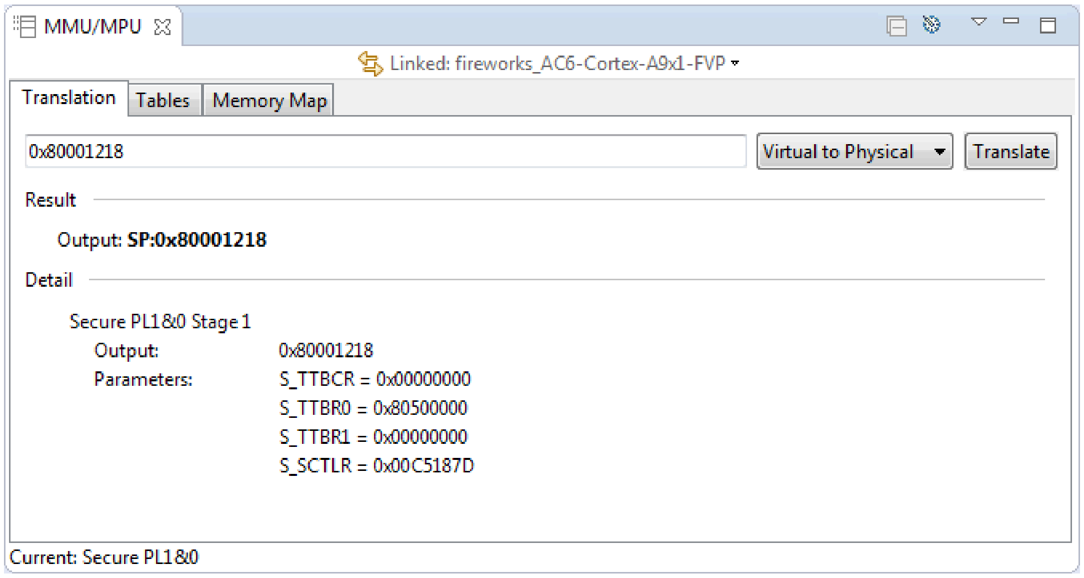
MMU awareness is only supported on Arm®v7-A, Armv8-A, Armv8-R AArch64, and Armv9-A architectures. MPU awareness is only supported on Armv8-M and Armv8-R architectures.

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 11-29: MMU/MPU Translation tab view



0x80001218
Virtual to Physical

Result

Output: **SP:0x80001218**

Detail

Secure PL1&0 Stage 1

```
Output:          0x80001218
Parameters:      S_TTBCR = 0x00000000
                 S_TTBR0 = 0x80500000
                 S_TTBR1 = 0x00000000
                 S_SCTLR = 0x00C5187D
```

Current: Secure PL1&0

To perform an address translation in the **Translation** tab:

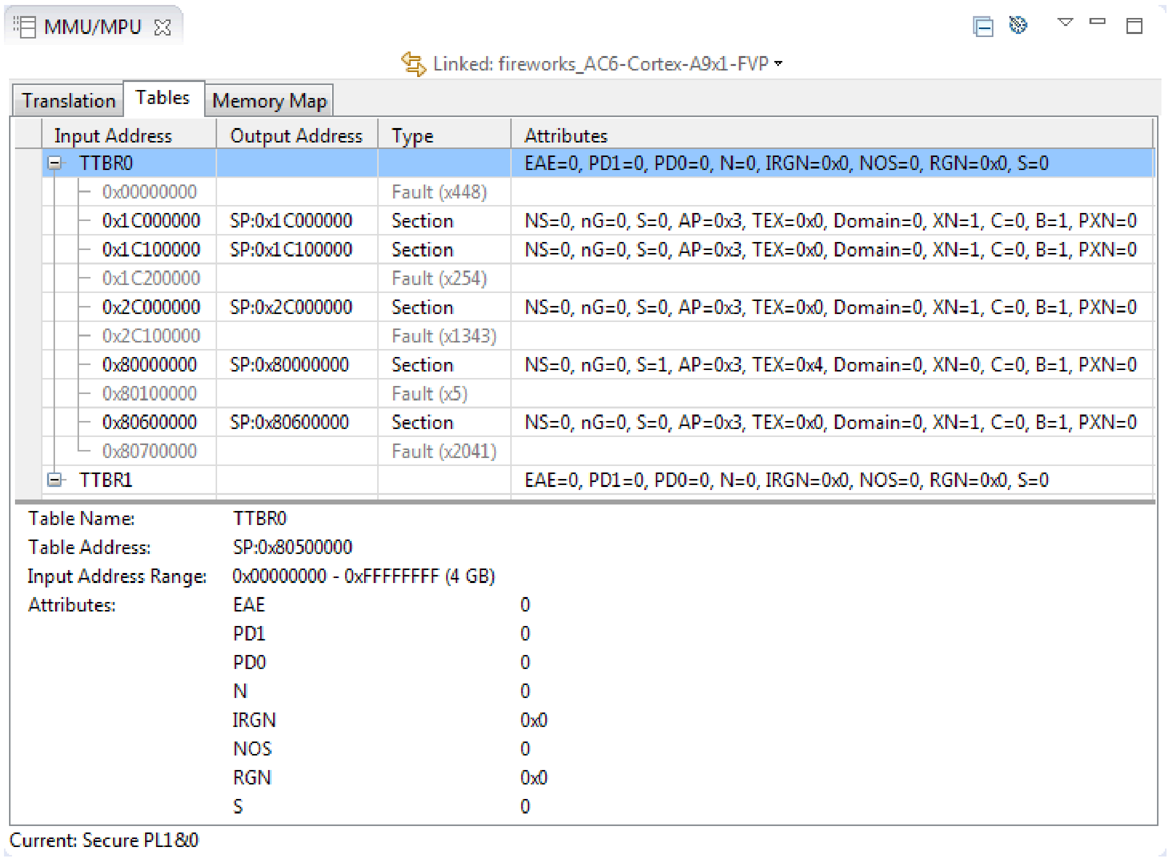
1. Enter a physical or virtual address in the address field. You can also enter an expression that evaluates to an address.
2. Select **Physical to Virtual** or **Virtual to Physical** depending on the translation type.
3. Click **Translate** to perform the address translation.

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.

Figure 11-30: MMU/MPU Tables tab view



Input Address	Output Address	Type	Attributes
TTBR0			EAE=0, PD1=0, PD0=0, N=0, IRGN=0x0, NOS=0, RGN=0x0, S=0
0x00000000		Fault (x448)	
0x1C000000	SP:0x1C000000	Section	NS=0, nG=0, S=0, AP=0x3, TEX=0x0, Domain=0, XN=1, C=0, B=1, PXN=0
0x1C100000	SP:0x1C100000	Section	NS=0, nG=0, S=0, AP=0x3, TEX=0x0, Domain=0, XN=1, C=0, B=1, PXN=0
0x1C200000		Fault (x254)	
0x2C000000	SP:0x2C000000	Section	NS=0, nG=0, S=0, AP=0x3, TEX=0x0, Domain=0, XN=1, C=0, B=1, PXN=0
0x2C100000		Fault (x1343)	
0x80000000	SP:0x80000000	Section	NS=0, nG=0, S=1, AP=0x3, TEX=0x4, Domain=0, XN=0, C=0, B=1, PXN=0
0x80100000		Fault (x5)	
0x80600000	SP:0x80600000	Section	NS=0, nG=0, S=0, AP=0x3, TEX=0x0, Domain=0, XN=1, C=0, B=1, PXN=0
0x80700000		Fault (x2041)	
TTBR1			EAE=0, PD1=0, PD0=0, N=0, IRGN=0x0, NOS=0, RGN=0x0, S=0

Table Name:	TTBR0	
Table Address:	SP:0x80500000	
Input Address Range:	0x00000000 - 0xFFFFFFFF (4 GB)	
Attributes:	EAE	0
	PD1	0
	PD0	0
	N	0
	IRGN	0x0
	NOS	0
	RGN	0x0
	S	0

Current: Secure PL1&0

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.

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.

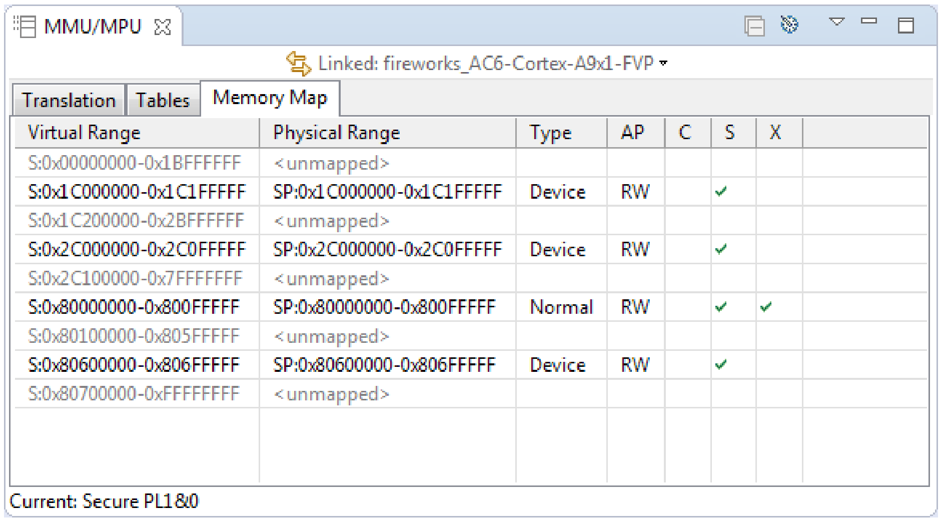
Attributes

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 11-31: MMU/MPU Memory Map tab view



Virtual Range	Physical Range	Type	AP	C	S	X
S:0x00000000-0x1BFFFFFF	<unmapped>					
S:0x1C000000-0x1C1FFFFF	SP:0x1C000000-0x1C1FFFFF	Device	RW		✓	
S:0x1C200000-0x2BFFFFFF	<unmapped>					
S:0x2C000000-0x2C0FFFFF	SP:0x2C000000-0x2C0FFFFF	Device	RW		✓	
S:0x2C100000-0x7FFFFFFF	<unmapped>					
S:0x80000000-0x800FFFFF	SP:0x80000000-0x800FFFFF	Normal	RW		✓	✓
S:0x80100000-0x805FFFFF	<unmapped>					
S:0x80600000-0x806FFFFF	SP:0x80600000-0x806FFFFF	Device	RW		✓	
S:0x80700000-0xFFFFFFFF	<unmapped>					

Current: Secure PL1&0

Toolbar and context menu options

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

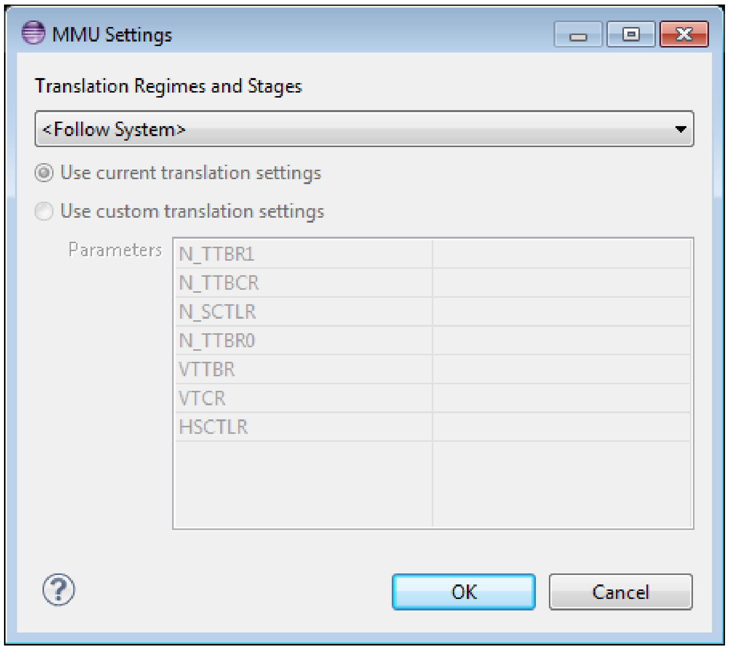
Linked: `<context>`

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:

Figure 11-32: MMU settings



Translation Regimes and Stages

<Follow System>

[✓ Use current translation settings]

Use custom translation settings

Parameters

N_TTBR1	
N_TTBCR	
N_SCTLR	
N_TTBR0	
VTTBR	
VTCR	
HSCTLR	

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.

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.

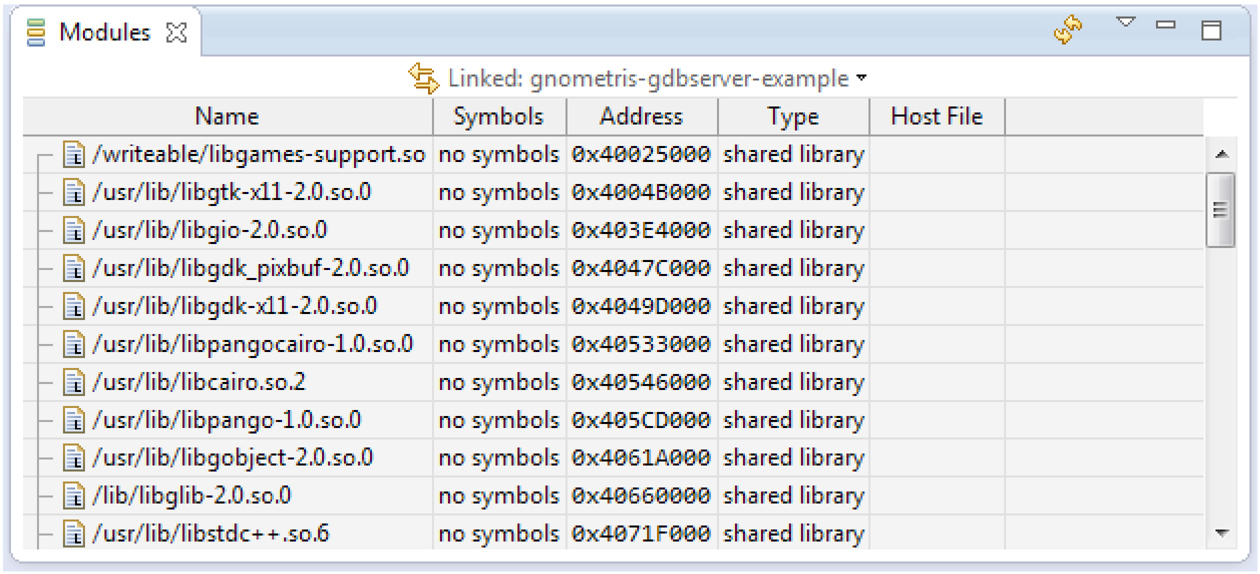
Coalesce Invalid Entries

Condenses the contiguous rows of faulty or invalid input addresses into a single row in the **Tables** tab.

11.19 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 11-33: Modules view showing shared libraries



Name	Symbols	Address	Type	Host File
/writeable/libgames-support.so	no symbols	0x40025000	shared library	
/usr/lib/libgtk-x11-2.0.so.0	no symbols	0x4004B000	shared library	
/usr/lib/libgio-2.0.so.0	no symbols	0x403E4000	shared library	
/usr/lib/libgdk_pixbuf-2.0.so.0	no symbols	0x4047C000	shared library	
/usr/lib/libgdk-x11-2.0.so.0	no symbols	0x4049D000	shared library	
/usr/lib/libpangocairo-1.0.so.0	no symbols	0x40533000	shared library	
/usr/lib/libcairo.so.2	no symbols	0x40546000	shared library	
/usr/lib/libpango-1.0.so.0	no symbols	0x405CD000	shared library	
/usr/lib/libgobject-2.0.so.0	no symbols	0x4061A000	shared library	
/lib/libglib-2.0.so.0	no symbols	0x40660000	shared library	
/usr/lib/libstdc++.so.6	no symbols	0x4071F000	shared library	

Note

A connection must be established and OS support enabled within the debugger before a loadable module can be detected. OS support is automatically enabled when a Linux kernel image is loaded into the debugger. However, you can manually control this by using the `set os` command.

Right-click on the column headers to select the columns that you want displayed:

Name
Displays the name and location of the component on the target.

Symbols
Displays whether the symbols are currently loaded for each object.

Address
Displays the load address of the object.

Size
Displays the size of the object.

Type
Displays the component type. For example, shared library or OS module.

Host File
Displays the name and location of the component on the host workstation.

Show All Columns

Displays all columns.

Reset Columns

Resets the columns displayed and their widths to the default.

The `Name`, `Symbols`, `Address`, `Type`, and `Host File` columns are displayed by default.

Toolbar and context menu options

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

`Linked: context`

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a different connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Copy

Copies the selected data.

Select All

Selects all the displayed data.

Load Symbols

Loads debug information into the debugger from the source file displayed in the Host File column. This option is disabled if the host file is unknown before the file is loaded.

Add Symbol File...

Opens a dialog box where you can select a file from the host workstation containing the debug information required by the debugger.

Discard Symbols

Discards debug information relating to the selected file.

Show in Memory

Displays the **Memory** view starting at the load address of the selected object.

Show in Disassembly

Displays the **Disassembly** view starting at the load address of the selected object.

View Menu

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

Update View When Hidden

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

Refresh

Refreshes the view.

Related information

About debugging multi-threaded applications on page 28

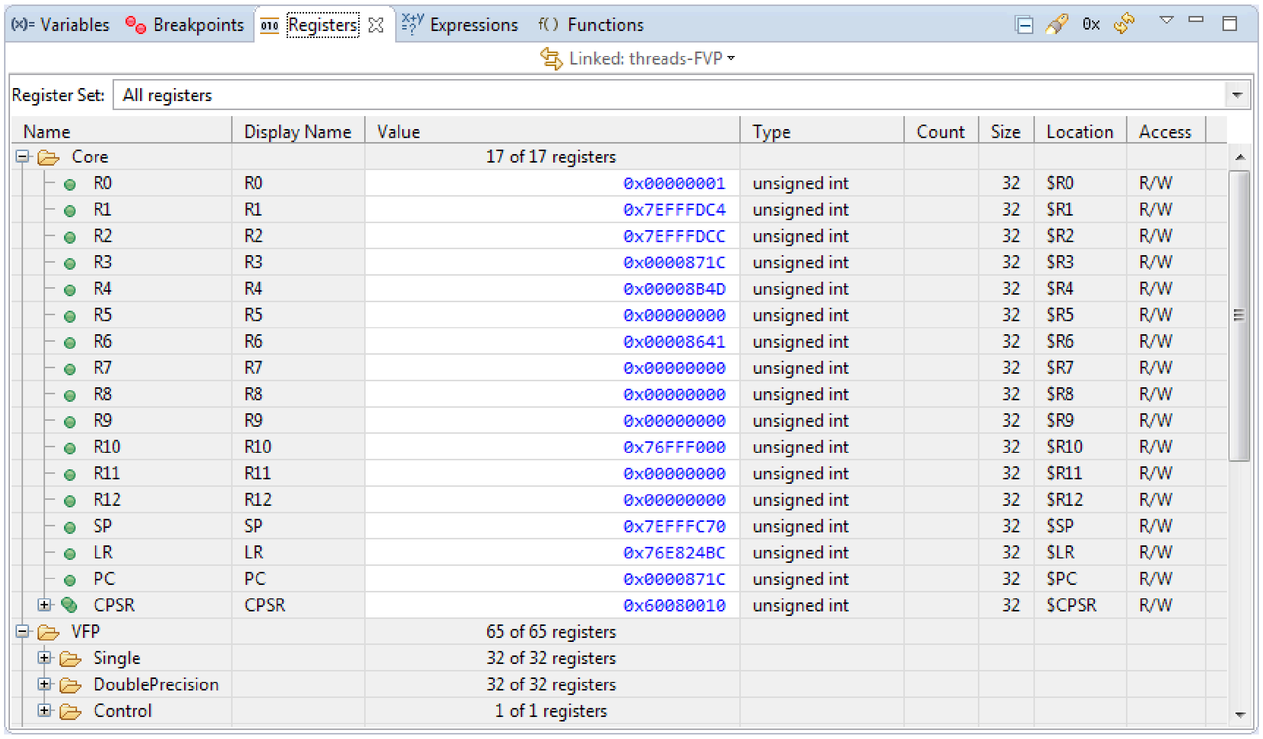
About debugging shared libraries on page 29

About debugging a Linux kernel on page 33
About debugging Linux kernel modules on page 35
About debugging TrustZone enabled targets on page 39
Perspectives and Views on page 315

11.20 Registers view

Use the **Registers** view to work with the contents of processor and peripheral registers available on your target.

Figure 11-34: Registers view (with all columns displayed)



Linked: threads-FVP

Register Set: All registers

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					

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 ⟳ .

Search for a specific register

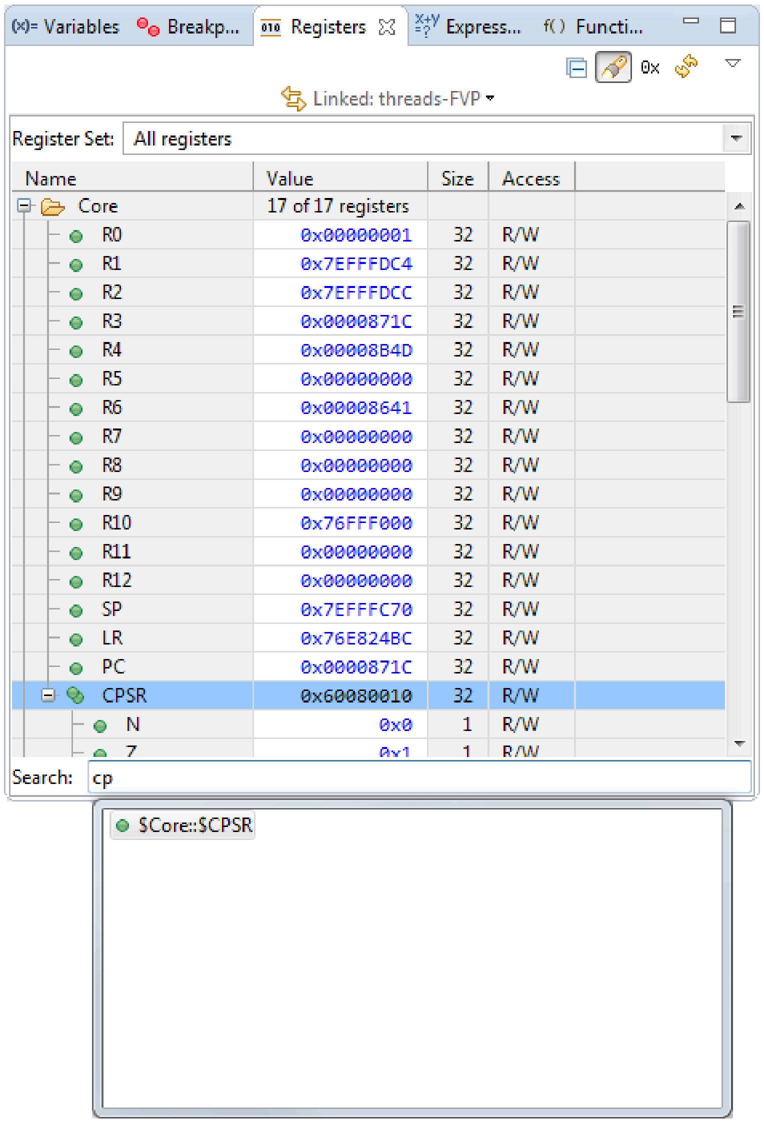
You can use the search feature in the **Registers** view to search for a specific register or group.

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 11-35: Registers - CP



Register Set: All registers

Name	Value	Size	Access
Core	17 of 17 registers		
R0	0x00000001	32	R/W
R1	0x7EFFFDC4	32	R/W
R2	0x7EFFFDCC	32	R/W
R3	0x0000871C	32	R/W
R4	0x00008B4D	32	R/W
R5	0x00000000	32	R/W
R6	0x00008641	32	R/W
R7	0x00000000	32	R/W
R8	0x00000000	32	R/W
R9	0x00000000	32	R/W
R10	0x76FFF000	32	R/W
R11	0x00000000	32	R/W
R12	0x00000000	32	R/W
SP	0x7EFFFC70	32	R/W
LR	0x76E824BC	32	R/W
PC	0x0000871C	32	R/W
CPSR	0x60080010	32	R/W
N	0x0	1	R/W
Z	0x1	1	R/W

Search: cp

$Core::$CPSR

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.

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.

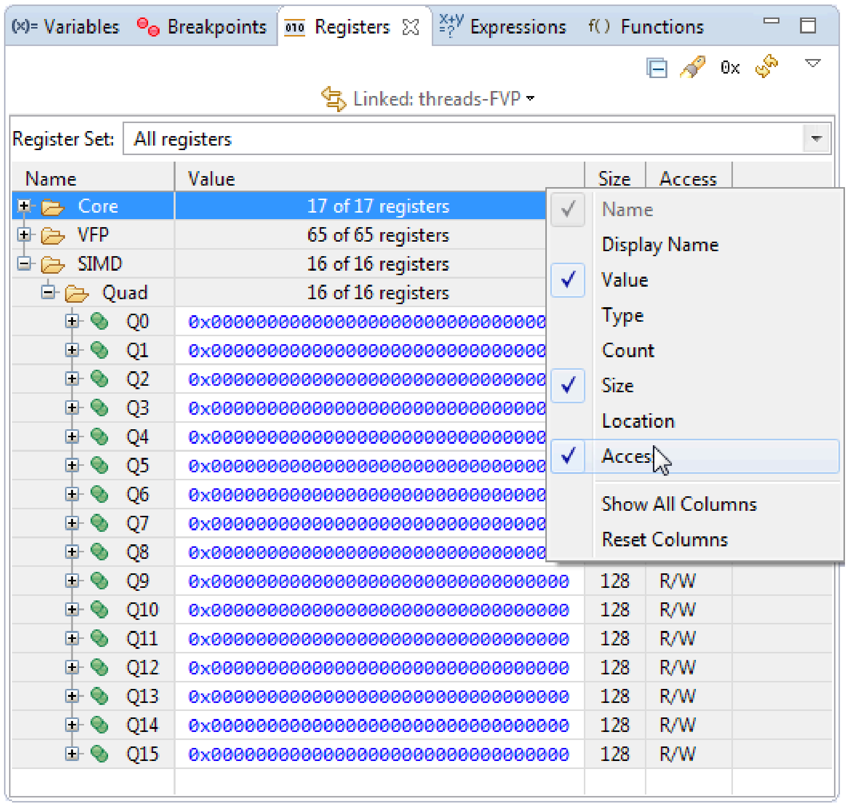
To manage a register set:

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.

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 11-36: Registers access rights



Register Set: All registers

Name	Value	Size	Access
Core	17 of 17 registers		
VFP	65 of 65 registers		
SIMD	16 of 16 registers		
Quad	16 of 16 registers		
Q0	0x000000000000000000000000000000		
Q1	0x000000000000000000000000000000		
Q2	0x000000000000000000000000000000		
Q3	0x000000000000000000000000000000		
Q4	0x000000000000000000000000000000		
Q5	0x000000000000000000000000000000		
Q6	0x000000000000000000000000000000		
Q7	0x000000000000000000000000000000		
Q8	0x000000000000000000000000000000		
Q9	0x00000000000000000000000000000000	128	R/W
Q10	0x00000000000000000000000000000000	128	R/W
Q11	0x00000000000000000000000000000000	128	R/W
Q12	0x00000000000000000000000000000000	128	R/W
Q13	0x00000000000000000000000000000000	128	R/W
Q14	0x00000000000000000000000000000000	128	R/W
Q15	0x00000000000000000000000000000000	128	R/W

Drag and drop an address held in a register from the Registers view to other views

Drag and drop an address held in a register from this view into either the **Memory** view to see the memory at that address, or into the **Disassembly** view to disassemble from that address.

Change the display format of register values

You can set the format of individual bits for Program Status Registers (PSRs).

Freeze the selected view to prevent the values being updated by a running target

Select **Freeze Data** from the view menu to prevent values updating automatically when the view refreshes.

Toolbar and context menu options

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

Linked: `<context>`

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a different connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Copy

Copies the selected registers. If a register contains bitfields, you must expand the bitfield to copy the individual bitfield values.

It can be useful to copy registers to a text editor in order to compare the values when execution stops at another location.

Select All

Selects all registers currently expanded in the view.

Show Memory Pointed to By <register name>

Displays the **Memory** view starting at the address held in the register.

Show Disassembly Pointed to By <register name>

Displays the **Disassembly** view starting at the address held in the register.

Translate Address in <register name>

Displays the **MMU** view and translates the address held in the register.

Send to <selection>

Displays a sub menu that enables you to add register filters to a specific **Expressions** view.

<Format list>

A list of formats you can use for the register values. 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.

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 information

Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
About debugging a Linux kernel on page 33
Pending breakpoints and watchpoints on page 73
About debugging multi-threaded applications on page 28

About debugging shared libraries on page 29
About debugging Linux kernel modules on page 35
About debugging TrustZone enabled targets on page 39
Perspectives and Views on page 315

11.21 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 Arm®v6-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.

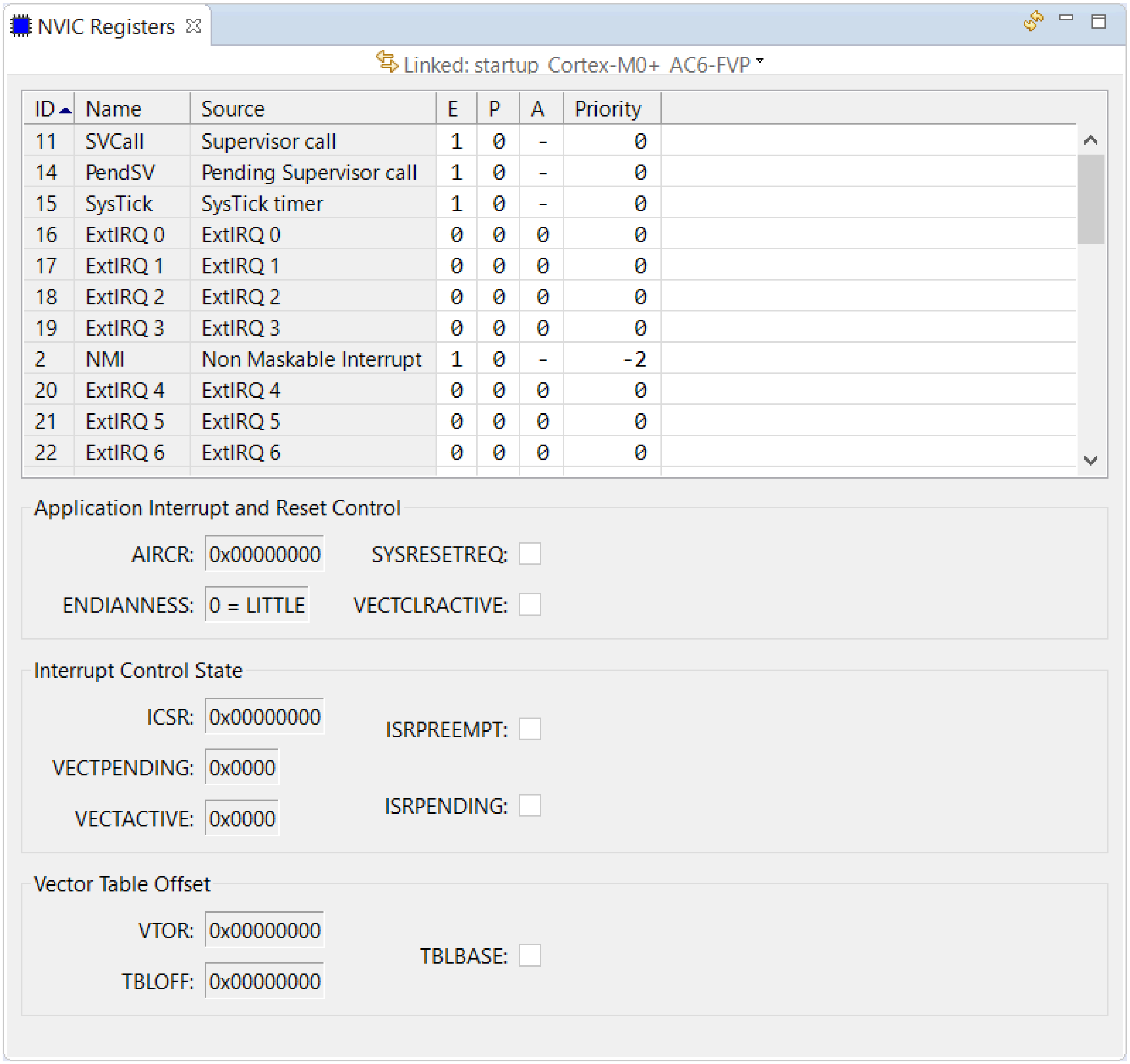
Pending

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.

Figure 11-37: NVIC Registers view


NVIC Registers

Linked: startup_Cortex-M0+_AC6-FVP

ID	Name	Source	E	P	A	Priority
11	SVCall	Supervisor call	1	0	-	0
14	PendSV	Pending Supervisor call	1	0	-	0
15	SysTick	SysTick timer	1	0	-	0
16	ExtIRQ 0	ExtIRQ 0	0	0	0	0
17	ExtIRQ 1	ExtIRQ 1	0	0	0	0
18	ExtIRQ 2	ExtIRQ 2	0	0	0	0
19	ExtIRQ 3	ExtIRQ 3	0	0	0	0
2	NMI	Non Maskable Interrupt	1	0	-	-2
20	ExtIRQ 4	ExtIRQ 4	0	0	0	0
21	ExtIRQ 5	ExtIRQ 5	0	0	0	0
22	ExtIRQ 6	ExtIRQ 6	0	0	0	0

Application Interrupt and Reset Control

AIRCR: 0x00000000 SYSRESETREQ: ☐

ENDIANNESS: 0 = LITTLE VECTCLRACTIVE: ☐

Interrupt Control State

ICSR: 0x00000000 ISRPREEMPT: ☐

VECTPENDING: 0x0000

VECTACTIVE: 0x0000 ISRPENDING: ☐

Vector Table Offset

VTOR: 0x00000000 TBLBASE: ☐

TBLOFF: 0x00000000

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.

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.

11.22 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.

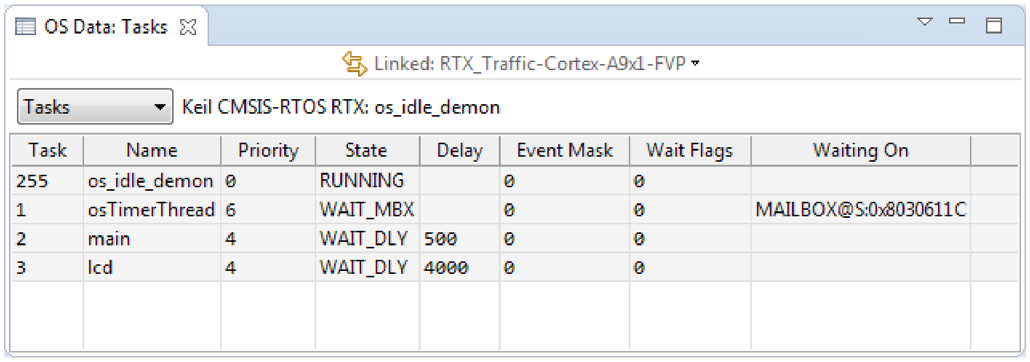
See About OS awareness for information about OS awareness support in Arm® Development Studio.

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.

Figure 11-38: OS Data view (showing Keil CMSIS-RTOS RTX Tasks)



Keil CMSIS-RTOS RTX: os_idle_demon

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	

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.

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.

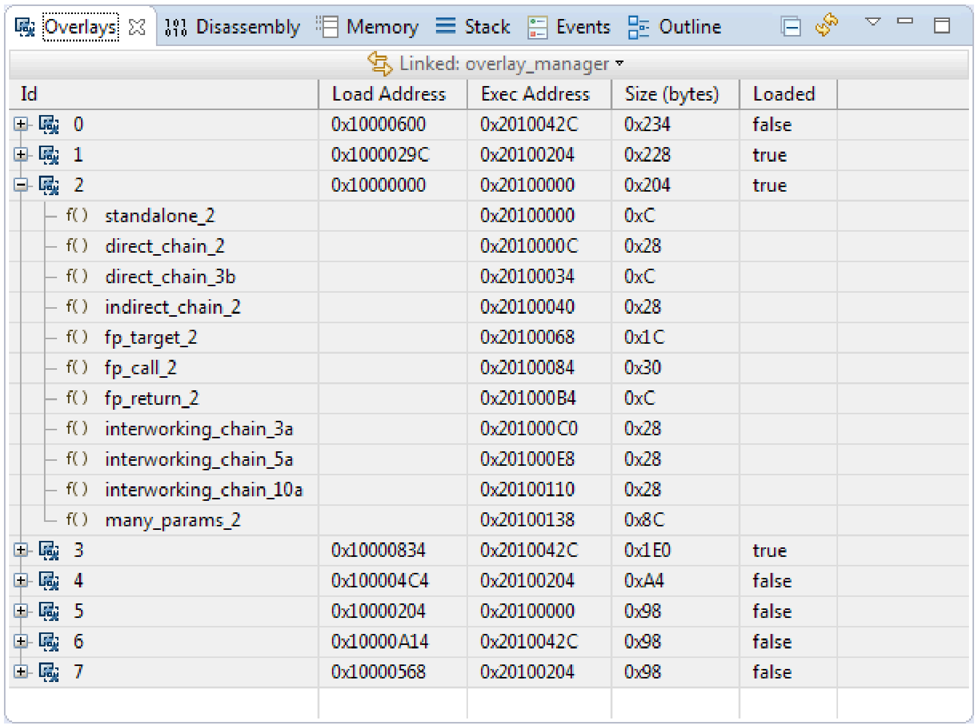
Select All

Selects all text.

11.23 Overlays view

Use the **Overlays** view to interact with the currently loaded overlaid application.

Figure 11-39: Overlays view



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

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 information

About Arm Debugger support for overlays on page 48
info overlays command
set overlays enabled command

11.24 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.

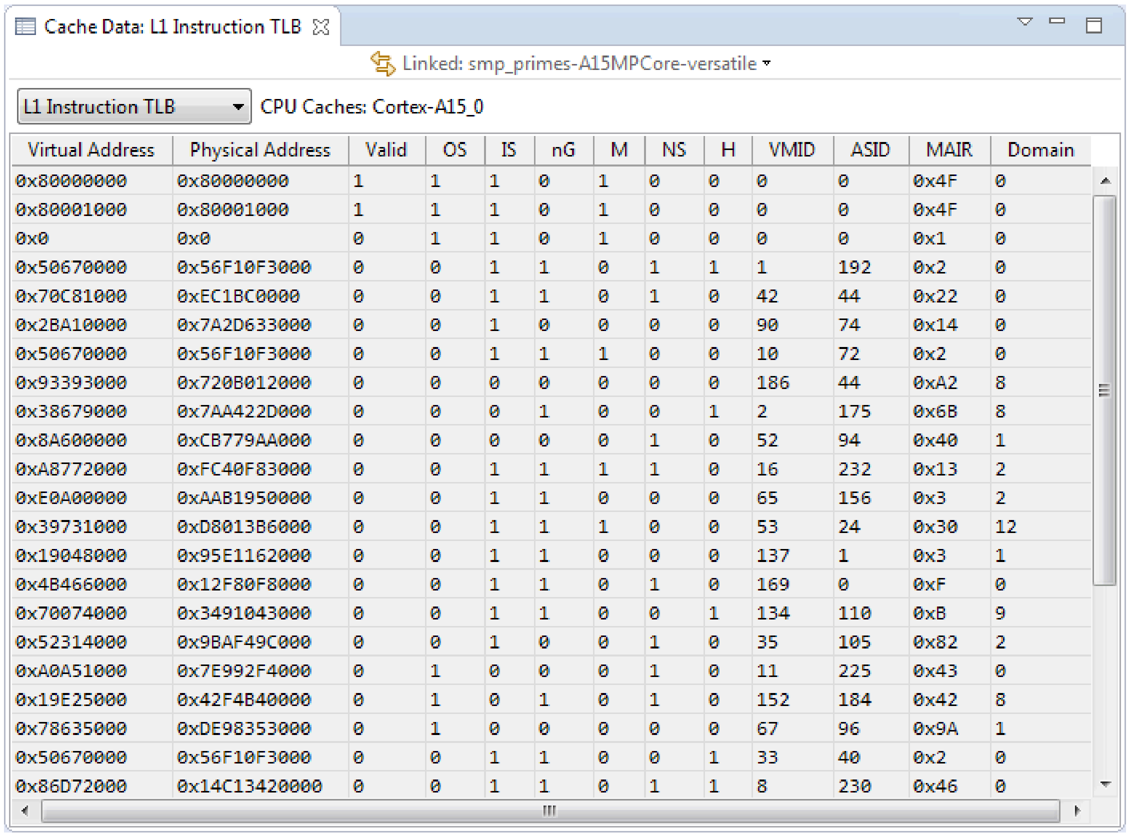
To access the **Cache Data** view, you must enable **Cache debug mode** in the **DTSL Configuration Editor** dialog box.

Note

Cache debug mode is only supported on certain target CPUs. For more information, refer to your CPU documentation.

Select the cache you want to view from the **CPU Caches** menu.

Figure 11-40: Cache Data view (showing L1 TLB cache)


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

Alternatively, you can use the `cache list` and `cache print` commands in the **Commands** view to show information about the caches.

Note

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

Toolbar and context menu options

`Linked: context`

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively, you can link the view to a different connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Show linked data in other Data views

Shows selected data in a view that is linked to another view.

View Menu

This menu contains the following options:

New Cache Data View
Displays a new instance of the **Cache Data** view.

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

Refresh
Refreshes the view.

Freeze Data
Toggles the freezing of data in the current view. 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 information

About debugging caches on page 45
DTSL Configuration Editor dialog box on page 447
Memory view on page 359
Debug commands: cache

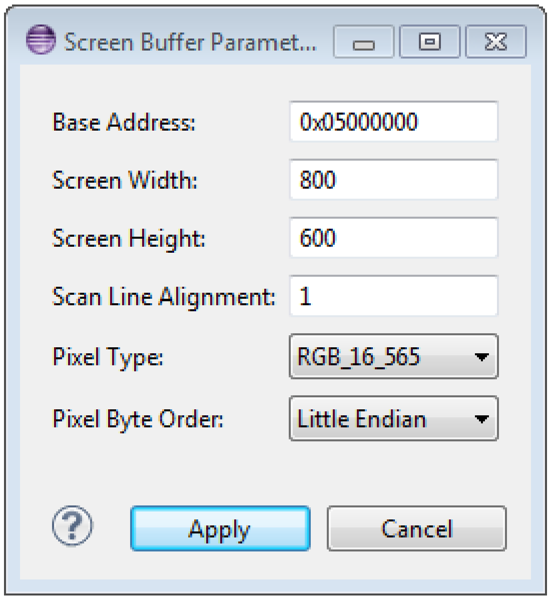
11.25 Screen view

Use the **Screen** view to display the contents of the screen buffer.

This view enables you to:

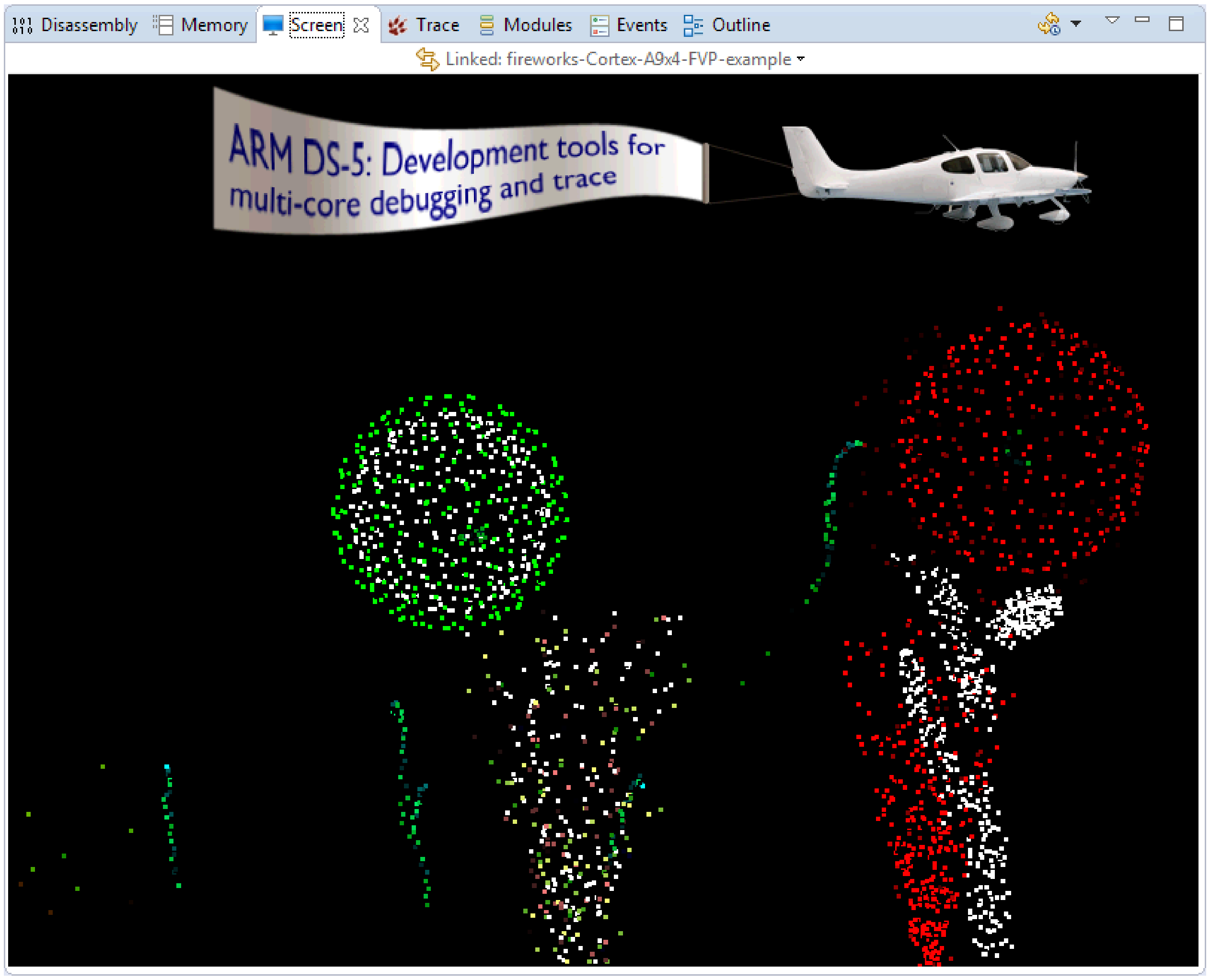
- Configure when view updates should occur and the interval between updates.
- Freeze the view to prevent it being updated by the running target when it next updates.
- Set the screen buffer parameters appropriate for the target:

Figure 11-41: Screen buffer parameters



Base Address:	0x05000000
Screen Width:	800
Screen Height:	600
Scan Line Alignment:	1
Pixel Type:	RGB_16_565
Pixel Byte Order:	Little Endian

Figure 11-42: Screen view


Disassembly
Memory
Screen
Trace
Modules
Events
Outline
Linked: fireworks-Cortex-A9x4-FVP-example
ARM DS-5: Development tools for multi-core debugging and trace

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.

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 information

Perspectives and Views on page 315

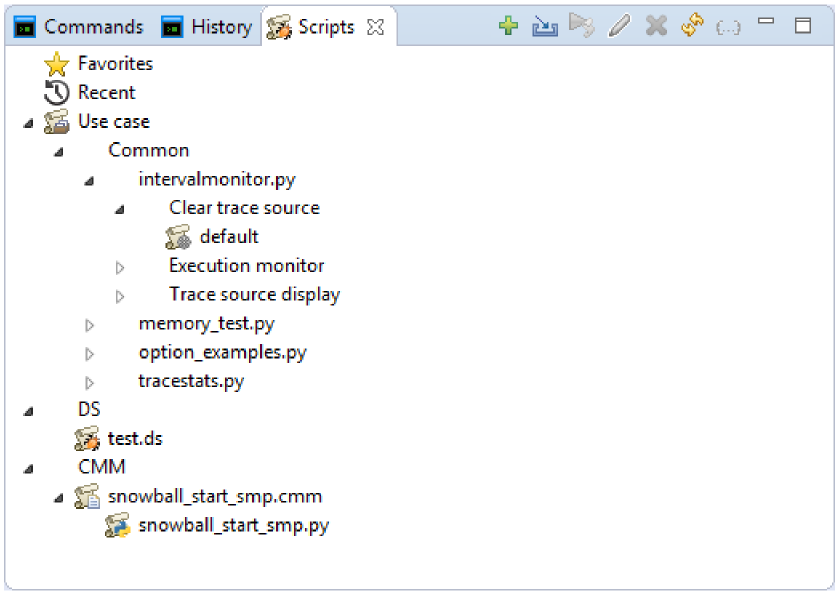
11.26 Scripts view

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 11-43: Scripts view



Using the **Scripts** view, you can:

Create a script

To create a script, click Create 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 Import a DS or Jython script and browse and select your file.

Import and translate a CMM script

To import and translate 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.

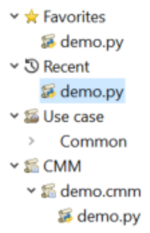
Configure script options

Select a script, and click to configure script options.

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.

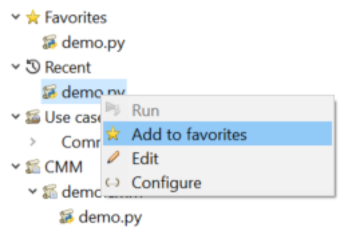
Figure 11-44: Recent scripts



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**.

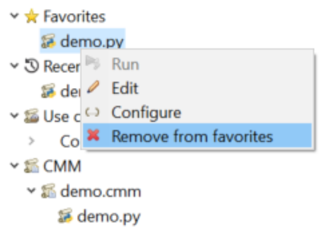
Figure 11-45: Add to favorites



Remove from favorites

To remove your scripts from the **Favorites** list, right-click a script and select **Remove from favorites**.

Figure 11-46: Remove from favorites

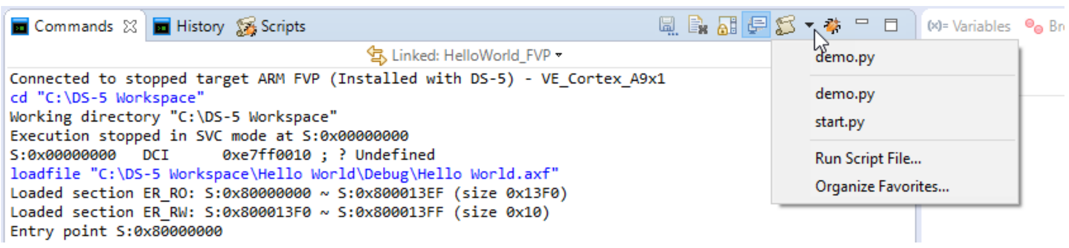


To delete multiple scripts, select them with Ctrl+click and press the **Delete** key.

Tip

You can also access your favorites and recently accessed scripts from the Commands view.

Figure 11-47: Running scripts from the Commands view



```
Connected to stopped target ARM FVP (Installed with DS-5) - VE_Cortex_A9x1
cd "C:\DS-5 Workspace"
Working directory "C:\DS-5 Workspace"
Execution stopped in SVC mode at S:0x00000000
S:0x00000000   DCI      0xe7ff0010 ; ? Undefined
loadfile "C:\DS-5 Workspace\Hello World\Debug\Hello World.axf"
Loaded section ER_RO: S:0x80000000 ~ S:0x800013EF (size 0x13F0)
Loaded section ER_RW: S:0x800013F0 ~ S:0x800013FF (size 0x10)
Entry point S:0x80000000
```

Related information

Debugging with Scripts on page 127
Use case scripts on page 140
Running Arm Debugger from the operating system command-line or from a script on page 156
Support for importing and translating CMM scripts on page 129

11.27 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.

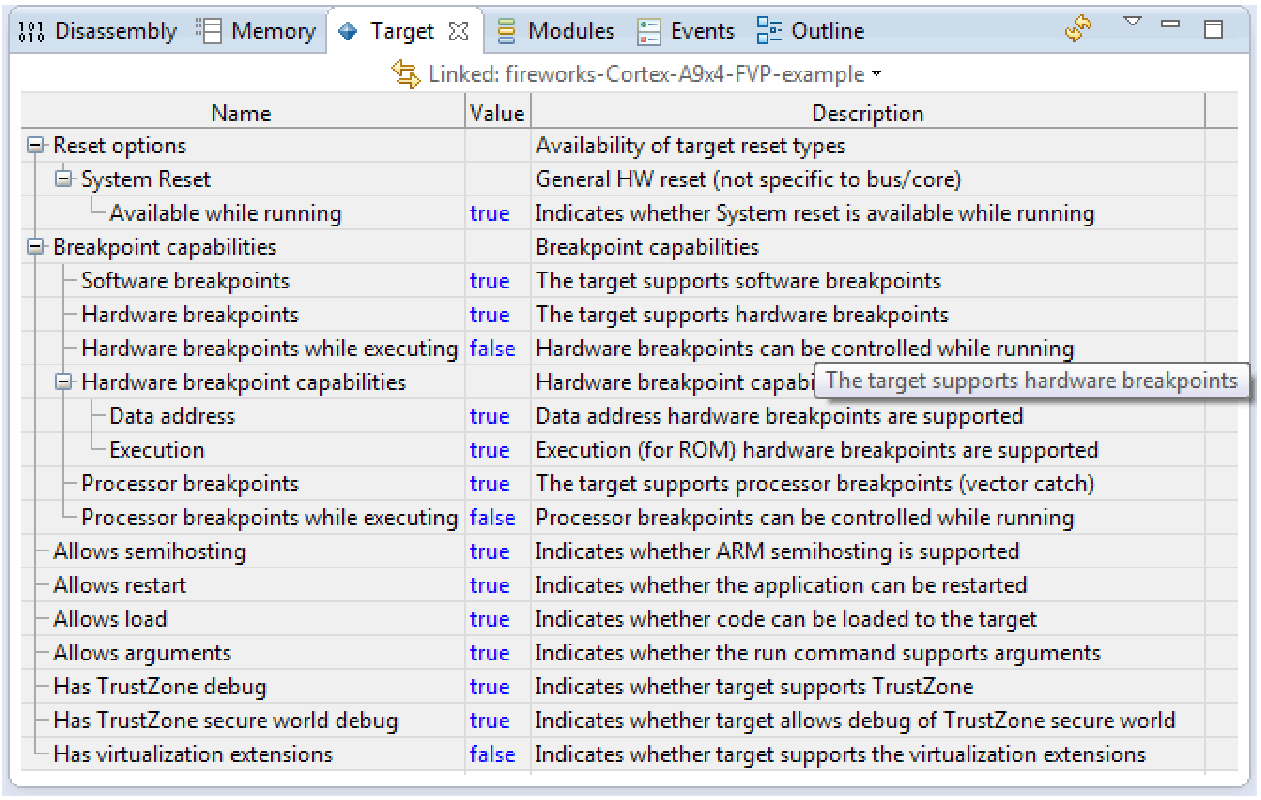
Related information

Perspectives and Views on page 315

11.28 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.

Figure 11-48: Target view



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 The target supports hardware breakpoints
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

Right-click on the column headers to select the columns that you want displayed:

Name
The name of the target capability.

Value
The value of the target capability.

Key
The name of the target capability. This is used by some commands in the **Commands** view.

Description
A brief description of the target capability.

Show All Columns
Displays all columns.

Reset Columns
Resets the columns displayed and their widths to the default.

The `Name`, `Value`, and `Description` columns are displayed by default.

The **Target** view is not visible by default. To add this view:

1. Ensure that you are in the **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 information

Perspectives and Views on page 315

11.29 Trace view

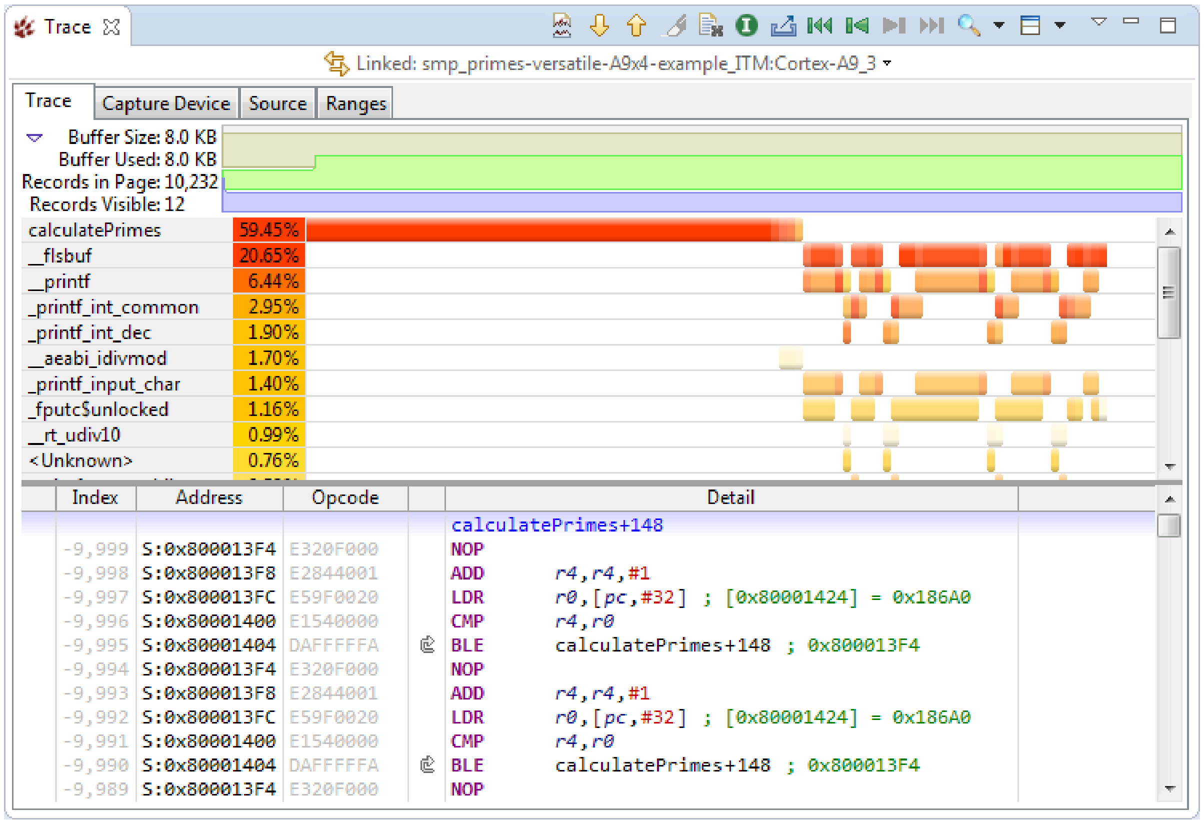
Use the **Trace** view to display a graphical navigation chart that shows function executions with a navigational timeline. In addition, the disassembly trace shows function calls with associated addresses and if selected, instructions. Clicking on a specific time in the chart synchronizes the **Disassembly** 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 11-49: Trace view with a scale of 100:1



Buffer Size: 8.0 KB
Buffer Used: 8.0 KB
Records in Page: 10,232
Records Visible: 12

calculatePrimes	59.45%
_flsbuf	20.65%
_printf	6.44%
_printf_int_common	2.95%
_printf_int_dec	1.90%
__aeabi_idivmod	1.70%
_printf_input_char	1.40%
_fputc$unlocked	1.16%
__rt_udiv10	0.99%
<Unknown>	0.76%

```
Index   Address       Opcode            Detail
                                        calculatePrimes+148
-9,999  S:0x800013F4  E320F000          NOP
-9,998  S:0x800013F8  E2844001          ADD     r4,r4,#1
-9,997  S:0x800013FC  E59F0020          LDR     r0,[pc,#32]  ; [0x80001424] = 0x186A0
-9,996  S:0x80001400  E1540000          CMP     r4,r0
-9,995  S:0x80001404  DAFFFFFA          BLE     calculatePrimes+148  ; 0x800013F4
-9,994  S:0x800013F4  E320F000          NOP
-9,993  S:0x800013F8  E2844001          ADD     r4,r4,#1
-9,992  S:0x800013FC  E59F0020          LDR     r0,[pc,#32]  ; [0x80001424] = 0x186A0
-9,991  S:0x80001400  E1540000          CMP     r4,r0
-9,990  S:0x80001404  DAFFFFFA          BLE     calculatePrimes+148  ; 0x800013F4
-9,989  S:0x800013F4  E320F000          NOP
```

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.

The **Trace** tab also shows:

Buffer Size

Size of the trace buffer to store trace records. This is determined by the trace capture device. The trace records can be instruction records or non-instruction records.

Buffer Used

Amount of the trace buffer that is already used for trace records.

Records in Page

The total number of instruction records and non-instruction records in the current **Trace** view.

Records Visible

The number of trace records visible in the disassembly area of the **Trace** view.

Toolbar and context menu options

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

`Linked: context`

Links this view to the selected connection in the**Debug Control** view. This is the default. Alternatively you can link the view to a different connection or processor in a Symmetric MultiProcessing (SMP) connection. If the connection you want is not shown in the drop-down list you might have to select it first in the **Debug Control** view.

Updating view when hidden, Not updating view when hidden

Toggles the updating of the view when it is hidden behind other views. By default the view does not update when it is hidden, which might cause loss of trace data.

Show Next Match

Moves the focus of the navigation chart and disassembly trace to the next matching occurrence for the selected function or instruction.

Show Previous Match

Moves the focus of the navigation chart and disassembly trace to the previous matching occurrence for the selected function or instruction.

Don't mark other occurrences - click to start marking**Mark other occurrences - click to stop marking**

When function trace is selected, marks all occurrences of the selected function with a shaded highlight. This is disabled when instruction trace is selected.

Clear Trace

Clears the raw trace data that is currently contained in the trace buffer and the trace view.

Showing instruction trace - click to switch to functions, Showing function trace - click to switch to instructions

Toggles the disassembly trace between instructions and functions.

Export Trace Report

Displays the **Export Trace Report** dialog box to save the trace data to a file.

Home

Where enabled, moves the trace view to the beginning of the trace buffer. Changes might not be visible if the trace buffer is too small.

Page Back

Where enabled, moves the trace view back one page. You can change the page size by modifying the **Set Maximum Instruction Depth** setting.

Page Forward

Where enabled, moves the trace view forward one page. You can change the page size by modifying the **Set Maximum Instruction Depth** setting.

End

Where enabled, moves the trace view to the end of the trace buffer. Changes might not be visible if the trace buffer is too small.

Switch between navigation resolutions

Changes the timeline resolution in the navigation chart.

Switch between alternate views

Changes the view to display the navigation chart, disassembly trace or both.

Focus Here

At the top of the list, displays the function being executed in the selected time slot. The remaining functions are listed in the order in which they are executed after the selected point in time. Any functions that do not appear after that point in time are placed at the bottom and ordered by total time.

Order By Total Time

Displays the functions ordered by the total time spent within the function. This is the default ordering.

View Menu

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

New Trace View

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

Set Trace Page Size...

Displays a dialog box in which you can enter the maximum number of instructions to display in the disassembly trace. The number must be within the range of 1,000 to 1,000,000 instructions.

Find Trace Trigger Event

Enables you to search for trigger events in the trace capture buffer.

Find Timestamp...

Displays a dialog box in which you can enter either a numeric timestamp as a 64 bit value or in the h:m:s format.

Find Function...

Enables you to search for a function by name in the trace buffer.

Find Instruction by Address...

Enables you to search for an instruction by address in the trace buffer.

Find ETM data access in trace buffer...

Enables you to search for a data value or range of values in the trace buffer.

Find Instruction Index...

Enables you to search for an instruction by index. A positive index is relative to the start of the trace buffer and a negative index is relative to the end.

DTSL Options...

Displays a dialog box in which you can add, edit, or choose a DTSL configuration.

Note

This clears the trace buffer.

Open Trace Control View

Opens the **Trace Control View**.

Refresh

Discards all the data in the view and rereads it from the current trace buffer.

Freeze Data

Toggles the freezing of data in the current view. 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

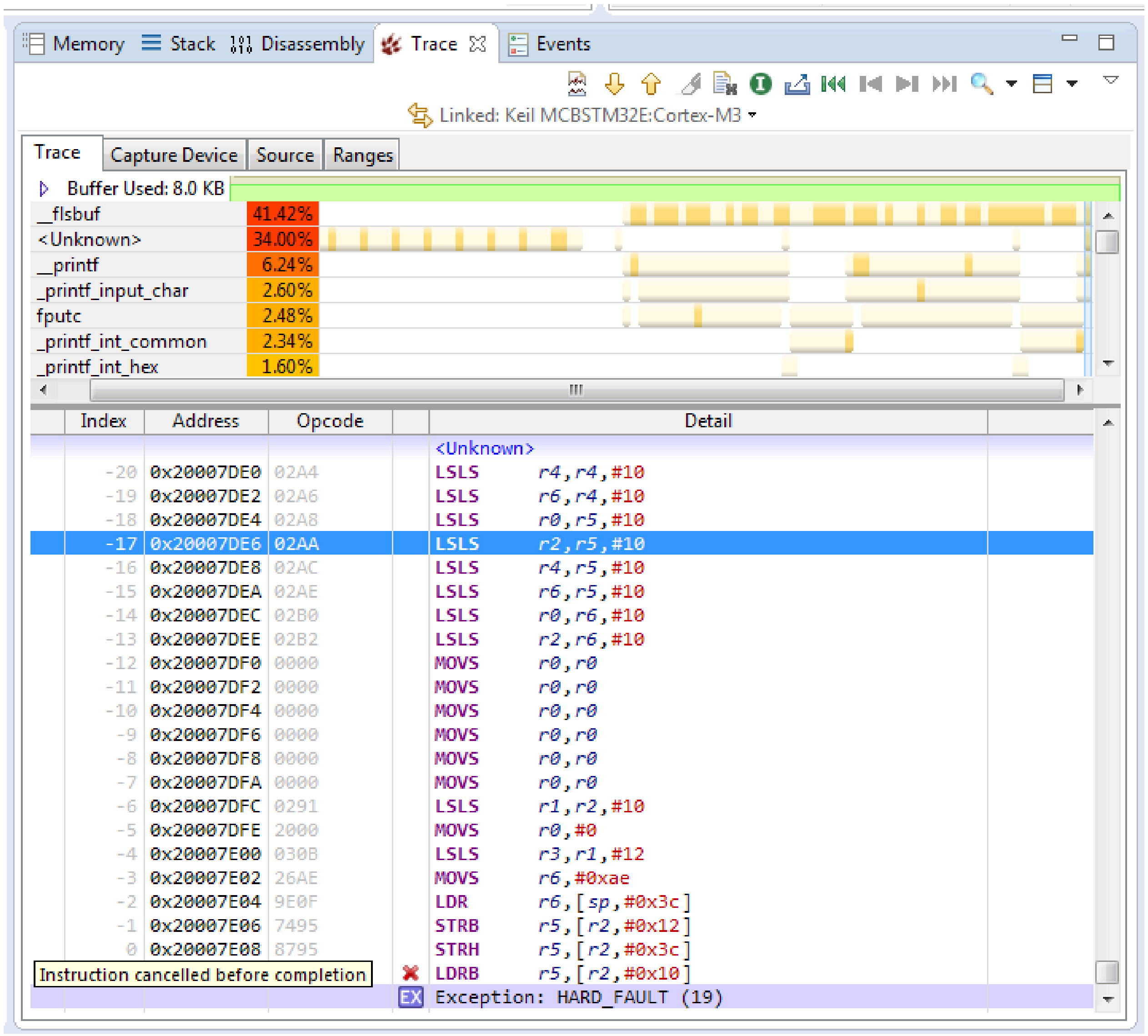
The opcode for the instruction expressed in hexadecimal.

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.

Figure 11-50: Trace view for Cortex-M3 Thumb instructions



Buffer Used: 8.0 KB

_flsbuf	41.42%
<Unknown>	34.00%
_printf	6.24%
_printf_input_char	2.60%
fputc	2.48%
_printf_int_common	2.34%
_printf_int_hex	1.60%

Index	Address	Opcode	Detail
			<Unknown>
-20	0x20007DE0	02A4	LSLS r4,r4,#10
-19	0x20007DE2	02A6	LSLS r6,r4,#10
-18	0x20007DE4	02A8	LSLS r0,r5,#10
-17	0x20007DE6	02AA	LSLS r2,r5,#10
-16	0x20007DE8	02AC	LSLS r4,r5,#10
-15	0x20007DEA	02AE	LSLS r6,r5,#10
-14	0x20007DEC	02B0	LSLS r0,r6,#10
-13	0x20007DEE	02B2	LSLS r2,r6,#10
-12	0x20007DF0	0000	MOVS r0,r0
-11	0x20007DF2	0000	MOVS r0,r0
-10	0x20007DF4	0000	MOVS r0,r0
-9	0x20007DF6	0000	MOVS r0,r0
-8	0x20007DF8	0000	MOVS r0,r0
-7	0x20007DFA	0000	MOVS r0,r0
-6	0x20007DFC	0291	LSLS r1,r2,#10
-5	0x20007DFE	2000	MOVS r0,#0
-4	0x20007E00	030B	LSLS r3,r1,#12
-3	0x20007E02	26AE	MOVS r6,#0xae
-2	0x20007E04	9E0F	LDR r6,[sp,#0x3c]
-1	0x20007E06	7495	STRB r5,[r2,#0x12]
0	0x20007E08	8795	STRH r5,[r2,#0x3c]
Instruction cancelled before completion		✗	LDRB r5,[r2,#0x10]
		EX	Exception: HARD_FAULT (19)

Related information

Perspectives and Views on page 315

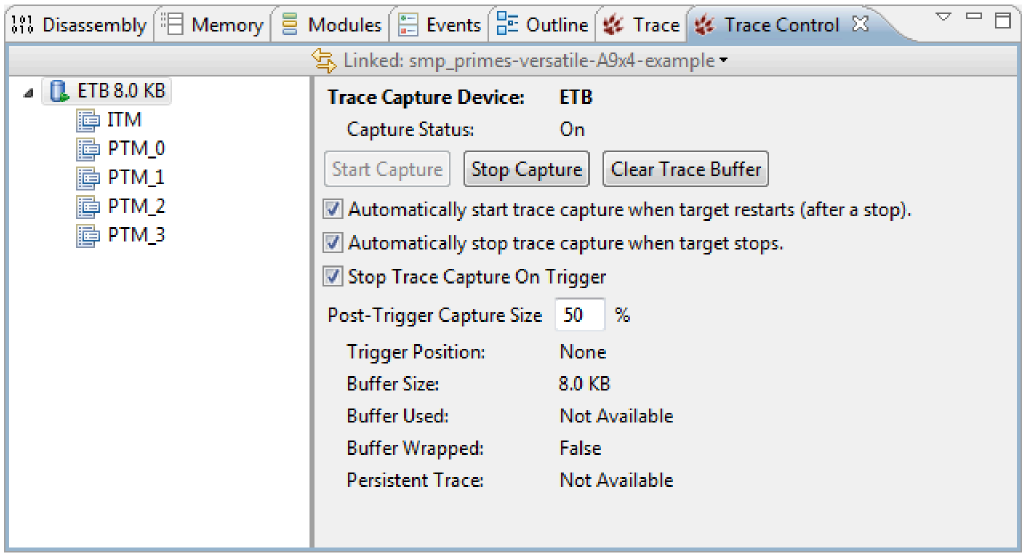
Capturing trace data using the command-line debugger on page 170

11.30 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 11-51: Trace Control view



Trace Capture Device: ETB
Capture Status: On

☑ Automatically start trace capture when target restarts (after a stop).
☑ Automatically stop trace capture when target stops.
☑ Stop Trace Capture On Trigger

Post-Trigger Capture Size 50 %

Trigger Position:	None
Buffer Size:	8.0 KB
Buffer Used:	Not Available
Buffer Wrapped:	False
Persistent Trace:	Not Available

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:

Trace Source

The name of the selected trace source.

Source ID

The unique ID of the selected trace source.

Source Encoding

The trace encoding format.

Core

The core associated with the trace source.

Context IDs

The tracing context IDs availability status.

Cycle Accurate Trace

The cycle accurate trace support status.

Virtualization Extensions

The virtualization extensions availability status.

Timestamps

Timestamp availability status for the trace.

Timestamp Origin

Whether a timestamp origin for the trace is set or cleared. When set, timestamps are displayed as offsets from the origin.

Trace Triggers

Trace triggers support status.

Trace Start Points

Trace start points support status.

Trace Stop Points

Trace stop points support status.

Trace Ranges

Trace ranges support status.

Note

The information displayed varies depending on the trace source.

Trace Control view options

Start Capture

Click **Start Capture** to start trace capture on the trace capture device. This is the same as the `trace start` command.

Stop Capture

Click **Stop Capture** to stop trace capture on the trace capture device. This is the same as the `trace stop` command.

Clear Trace Buffer

Click **Clear Trace Buffer** to empty the trace buffer on the trace capture device. This is the same as the `trace clear` command.

Start trace capture when target restarts (after a stop)

Select this option to automatically start trace capture after a target restarts after a stop.

Stop trace capture when target stops

Select this option to automatically stop trace capture when a target stops.

Stop trace capture on trigger

Select this option to stop trace capture after a trace capture trigger has been hit.

Post-trigger capture size

Use this option to control the percentage of the trace buffer that should be reserved for after a trigger point is hit. The range is from 0 to 99.

Note

The `trace start` and `trace stop` commands and the automatic start and stop trace options act as master switches. Trace triggers cannot override them.

Toolbar and context menu options

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

`Linked: context`

Links this view to the selected connection in the **Debug Control** view. This is the default. Alternatively you can link the view to a specific connection or processor in a Symmetric MultiProcessing (SMP) connection. If the connection you want is not shown in the drop-down list, you might have to select it first in the **Debug Control** view.

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.

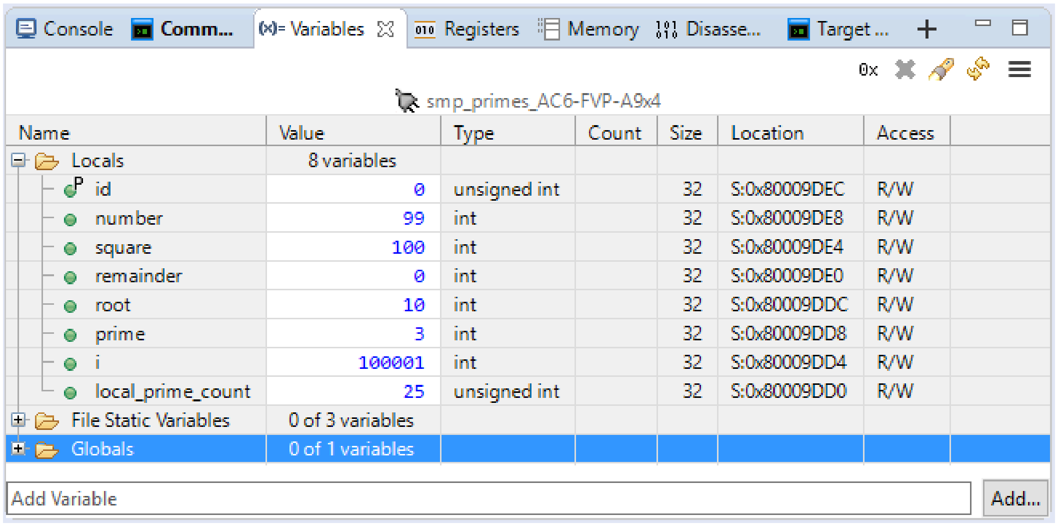
Related information

Perspectives and Views on page 315

11.31 Variables view

Use the **Variables** view to work with the contents of local, file static, and global variables in your program.

Figure 11-52: Variables view



Name	Value	Type	Count	Size	Location	Access
Locals	8 variables					
id	0	unsigned int		32	S:0x80009DEC	R/W
number	99	int		32	S:0x80009DE8	R/W
square	100	int		32	S:0x80009DE4	R/W
remainder	0	int		32	S:0x80009DE0	R/W
root	10	int		32	S:0x80009DDC	R/W
prime	3	int		32	S:0x80009DD8	R/W
i	100001	int		32	S:0x80009DD4	R/W
local_prime_count	25	unsigned int		32	S:0x80009DD0	R/W
File Static Variables	0 of 3 variables					
Globals	0 of 1 variables					

You can:

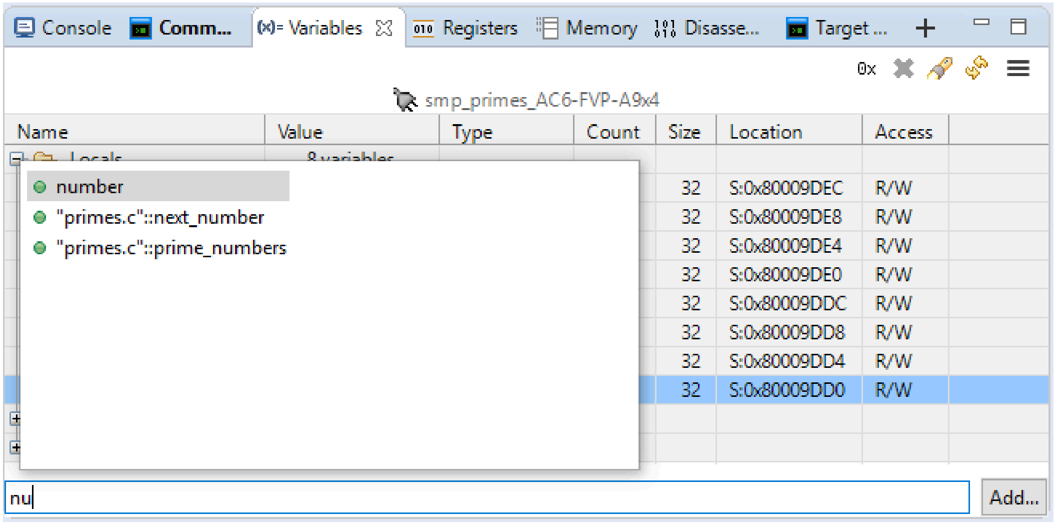
View the contents of variables that are currently in scope

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 11-53: How to add variables to the view



Name	Value	Type	Count	Size	Location	Access
Locals	8 variables					
				32	S:0x80009DEC	R/W
				32	S:0x80009DE8	R/W
				32	S:0x80009DE4	R/W
				32	S:0x80009DE0	R/W
				32	S:0x80009DDC	R/W
				32	S:0x80009DD8	R/W
				32	S:0x80009DD4	R/W
				32	S:0x80009DD0	R/W

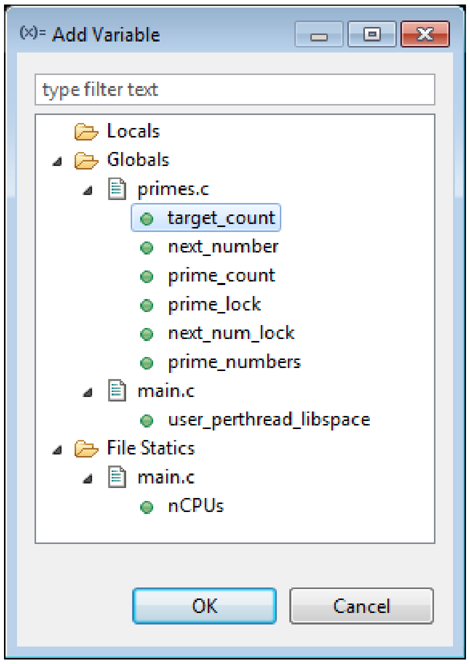
- number
- "primes.c"::next_number
- "primes.c"::prime_numbers

```
nu
```

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 11-54: Add Global Variables dialog box



- Locals
- Globals
  - primes.c
    - target_count
    - next_number
    - prime_count
    - prime_lock
    - next_num_lock
    - prime_numbers
  - main.c
    - user_perthread_libspace
- File Statics
  - main.c
    - nCPUs

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 .

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.

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

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.

Reset Columns

Resets the columns displayed and their widths to the default.

Related information

Setting a tracepoint on page 74
Conditional breakpoints on page 66
Assigning conditions to an existing breakpoint on page 68
Pending breakpoints and watchpoints on page 73
About debugging multi-threaded applications on page 28
About debugging shared libraries on page 29
Perspectives and Views on page 315
About debugging a Linux kernel on page 33

About debugging Linux kernel modules on page 35
About debugging TrustZone enabled targets on page 39

11.32 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.

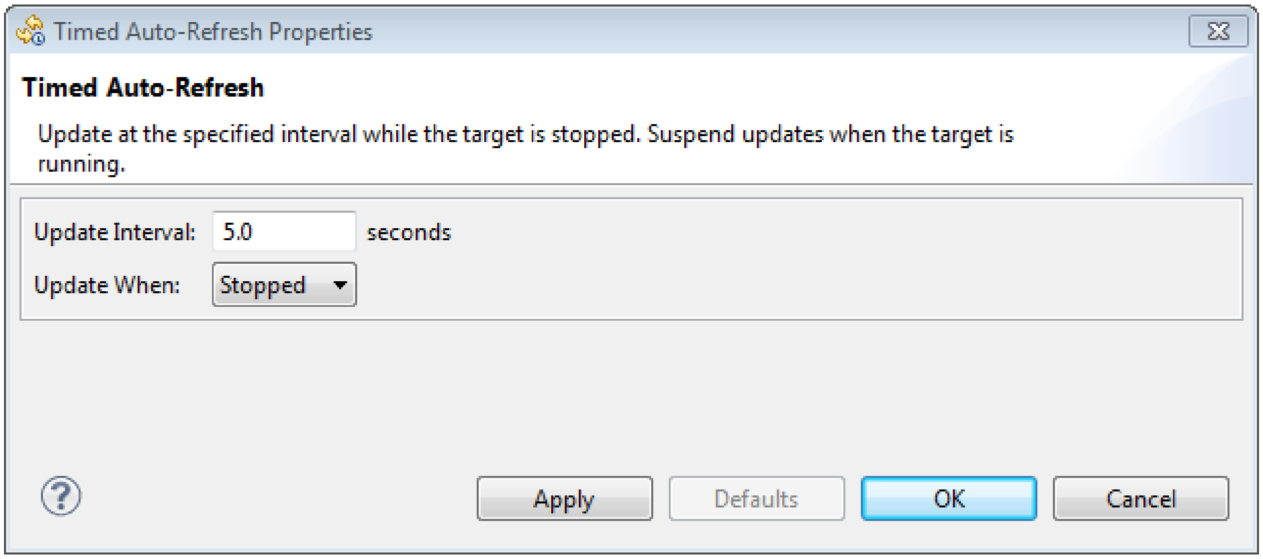
Stopped
Refreshes the view only while the target is stopped.

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 11-55: Timed Auto-Refresh properties dialog box



Timed Auto-Refresh Properties

Timed Auto-Refresh

Update at the specified interval while the target is stopped. Suspend updates when the target is running.

Update Interval: 5.0 seconds

Update When: Stopped

11.33 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).

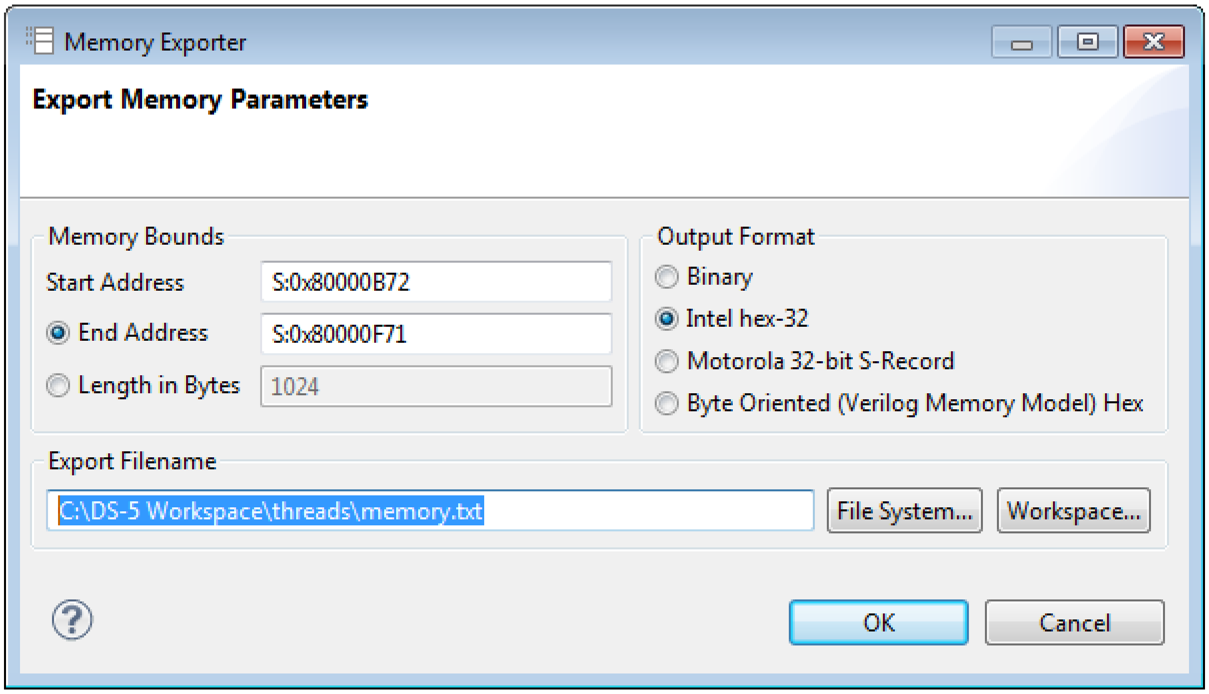
Export Filename

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 11-56: Memory Exporter dialog box



Start Address: S:0x80000B72
(●) End Address: S:0x80000F71
( ) Length in Bytes: 1024

( ) Binary
(●) Intel hex-32
( ) Motorola 32-bit S-Record
( ) Byte Oriented (Verilog Memory Model) Hex

Export Filename: C:\DS-5 Workspace\threads\memory.txt

11.34 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.

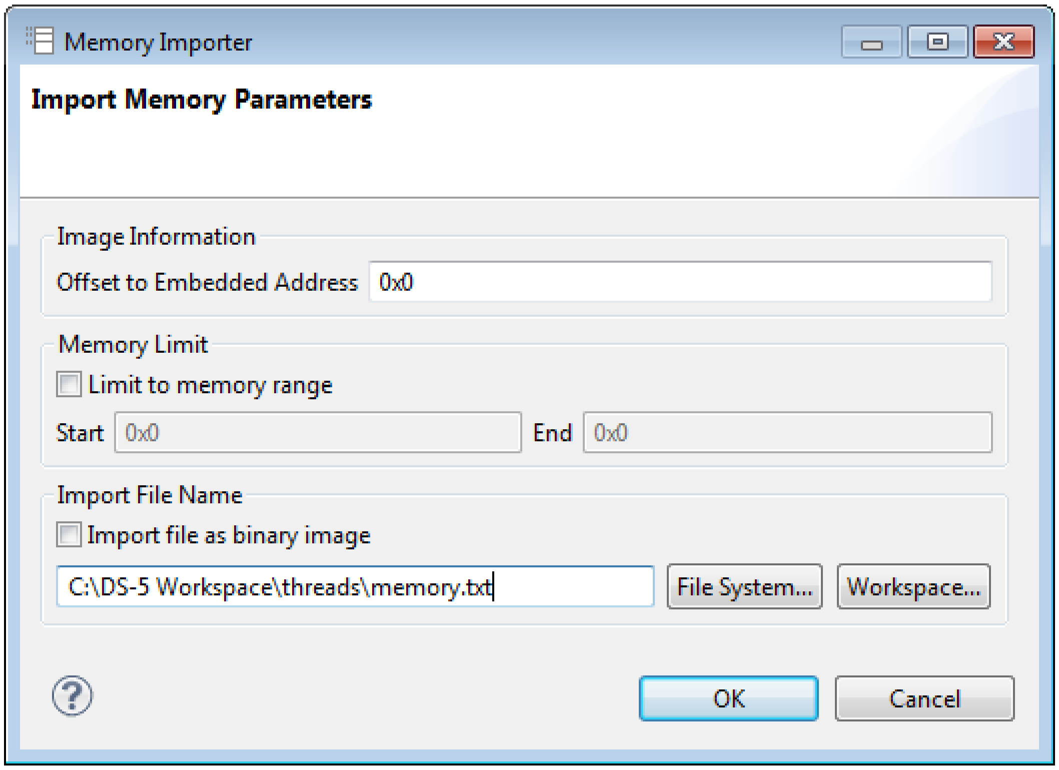
Import File Name

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 11-57: Memory Importer dialog box



Offset to Embedded Address: 0x0

[ ] Limit to memory range

Start: 0x0 End: 0x0

[ ] Import file as binary image

C:\DS-5 Workspace\threads\memory.txt

11.35 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.

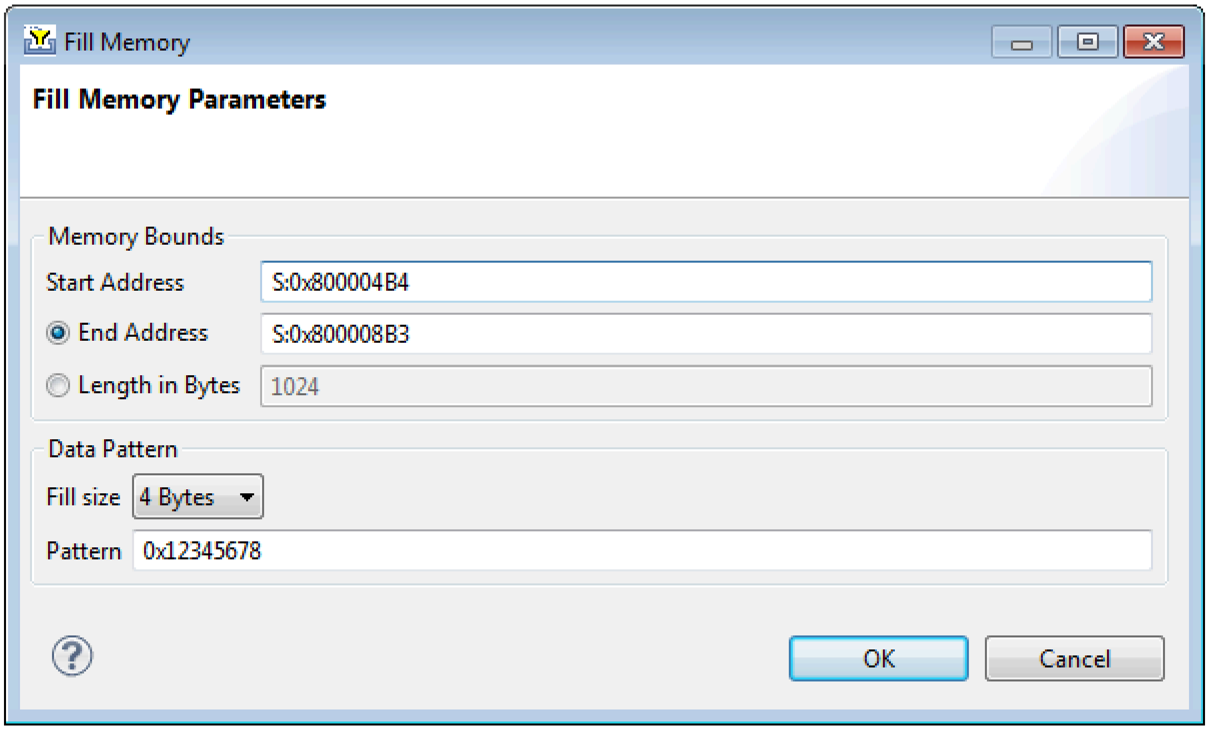
Fill size

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 11-58: Fill Memory dialog box



Memory Bounds

Start Address: S:0x800004B4

(●) End Address: S:0x800008B3

( ) Length in Bytes: 1024

Data Pattern

Fill size: 4 Bytes

Pattern: 0x12345678

11.36 Export Trace Report dialog box

Use the **Export Trace Report** dialog box to export a trace report.

Report Name

Enter the report location and name.

Base Filename

Enter the report name.

Output Folder

Enter the report folder location.

Browse

Selects the report location in the file system.

Include core

Enables you to add the core name in the report filename.

Include date time stamp

Enables you to add the date time stamp to the report filename.

Split Output Files

Splits the output file when it reaches the selected size.

Select source for trace report

Selects the required trace data.

Use trace view as report source

Instructions that are decoded in the **Trace** view.

Use trace buffer as report source

Trace data that is currently contained in the trace buffer.

Note

When specifying a range, ensure that the range is large enough otherwise you might not get any trace output. This is due to the trace packing format used in the buffer.

Report Format

Configures the report.

Output Format

Selects the output format.

Include column headers

Enables you to add column headers in the first line of the report.

Select columns to export

Enables you to filter the trace data in the report.

Record Filters

Enables or disables trace filters.

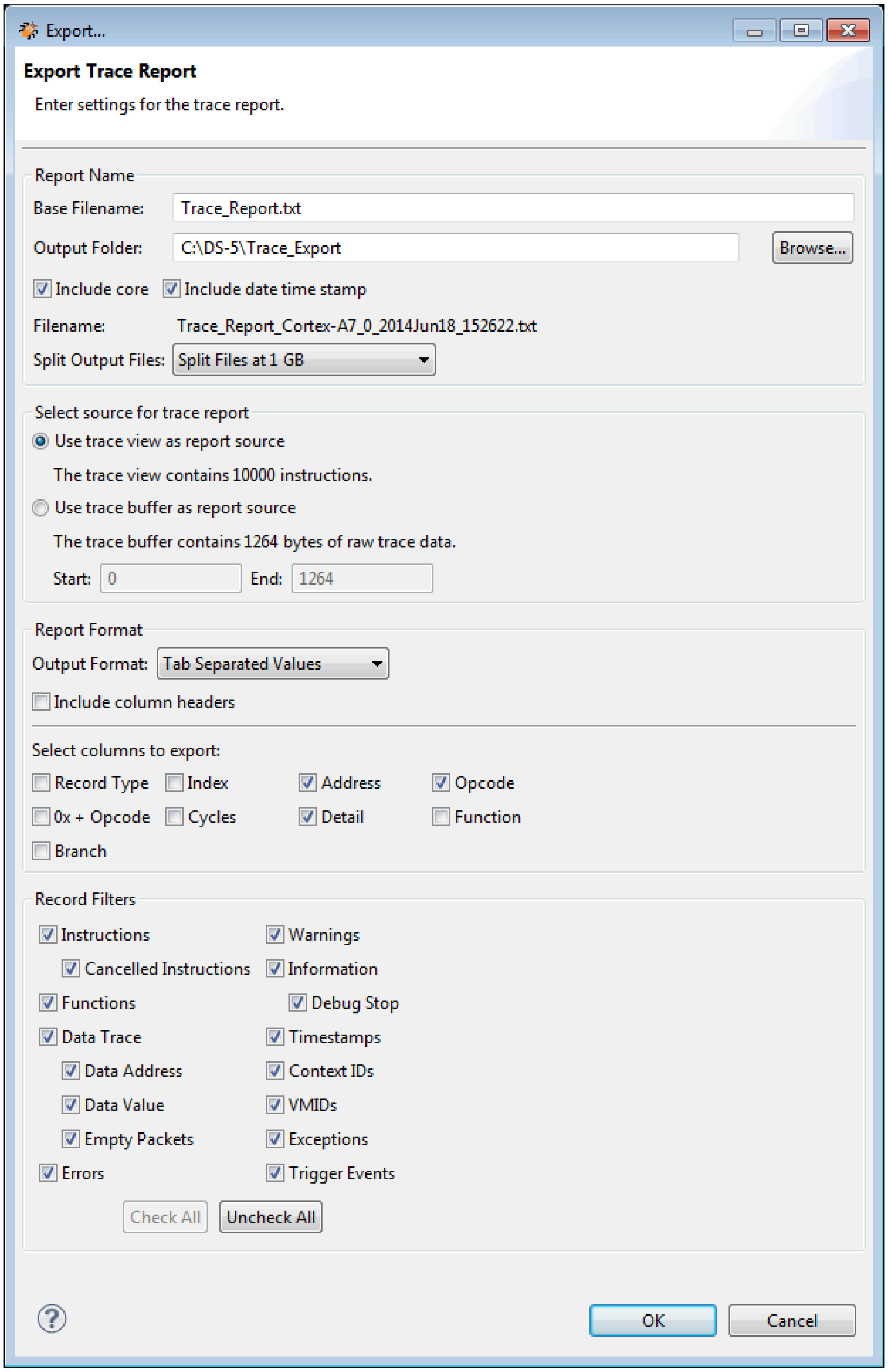
Check All

Enables you to select all the trace filters.

Uncheck All

Enables you to unselect all the trace filters.

Figure 11-59: Export Trace Report dialog box



Export...

Export Trace Report

Enter settings for the trace report.

Report Name

Base Filename: Trace_Report.txt

Output Folder: C:\DS-5\Trace_Export Browse...

[✓] Include core [✓] Include date time stamp

Filename: Trace_Report_Cortex-A7_0_2014Jun18_152622.txt

Split Output Files: Split Files at 1 GB

Select source for trace report

● Use trace view as report source

The trace view contains 10000 instructions.

○ Use trace buffer as report source

The trace buffer contains 1264 bytes of raw trace data.

Start: 0 End: 1264

Report Format

Output Format: Tab Separated Values

[ ] Include column headers

Select columns to export:

[ ] Record Type [ ] Index [✓] Address [✓] Opcode
[ ] 0x + Opcode [ ] Cycles [✓] Detail [ ] Function
[ ] Branch

Record Filters

[✓] Instructions [✓] Warnings
[✓] Cancelled Instructions [✓] Information
[✓] Functions [✓] Debug Stop
[✓] Data Trace [✓] Timestamps
[✓] Data Address [✓] Context IDs
[✓] Data Value [✓] VMIDs
[✓] Empty Packets [✓] Exceptions
[✓] Errors [✓] Trigger Events

Check All Uncheck All

OK Cancel

11.37 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.

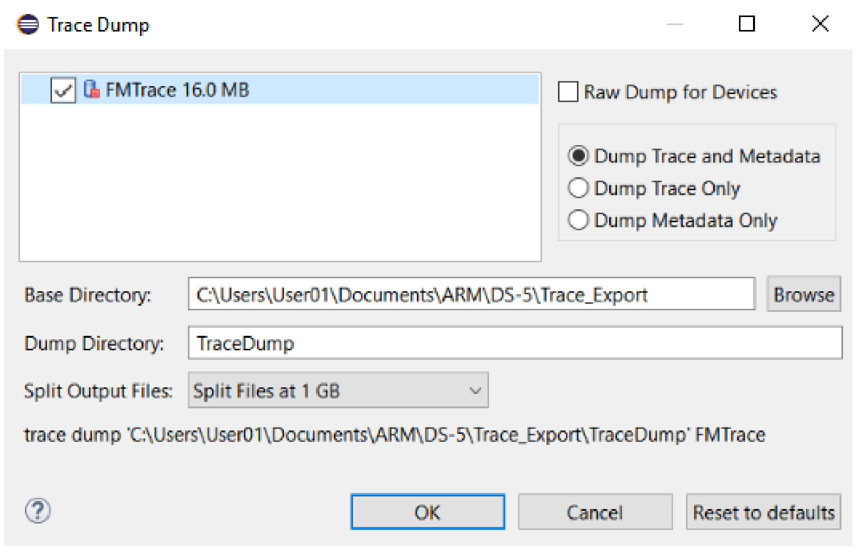
Cancel

Exit the **Trace Dump** dialog box.

Reset defaults

Reset all configuration options to their default values.

Figure 11-60: Trace Dump dialog box



Trace Dump

[✓] FMTrace 16.0 MB

[ ] Raw Dump for Devices

(● Dump Trace and Metadata) ○ Dump Trace Only ○ Dump Metadata Only

Base Directory: C:\Users\User01\Documents\ARM\DS-5\Trace_Export

Dump Directory: TraceDump

Split Output Files: Split Files at 1 GB

trace dump 'C:\Users\User01\Documents\ARM\DS-5\Trace_Export\TraceDump' FMTrace

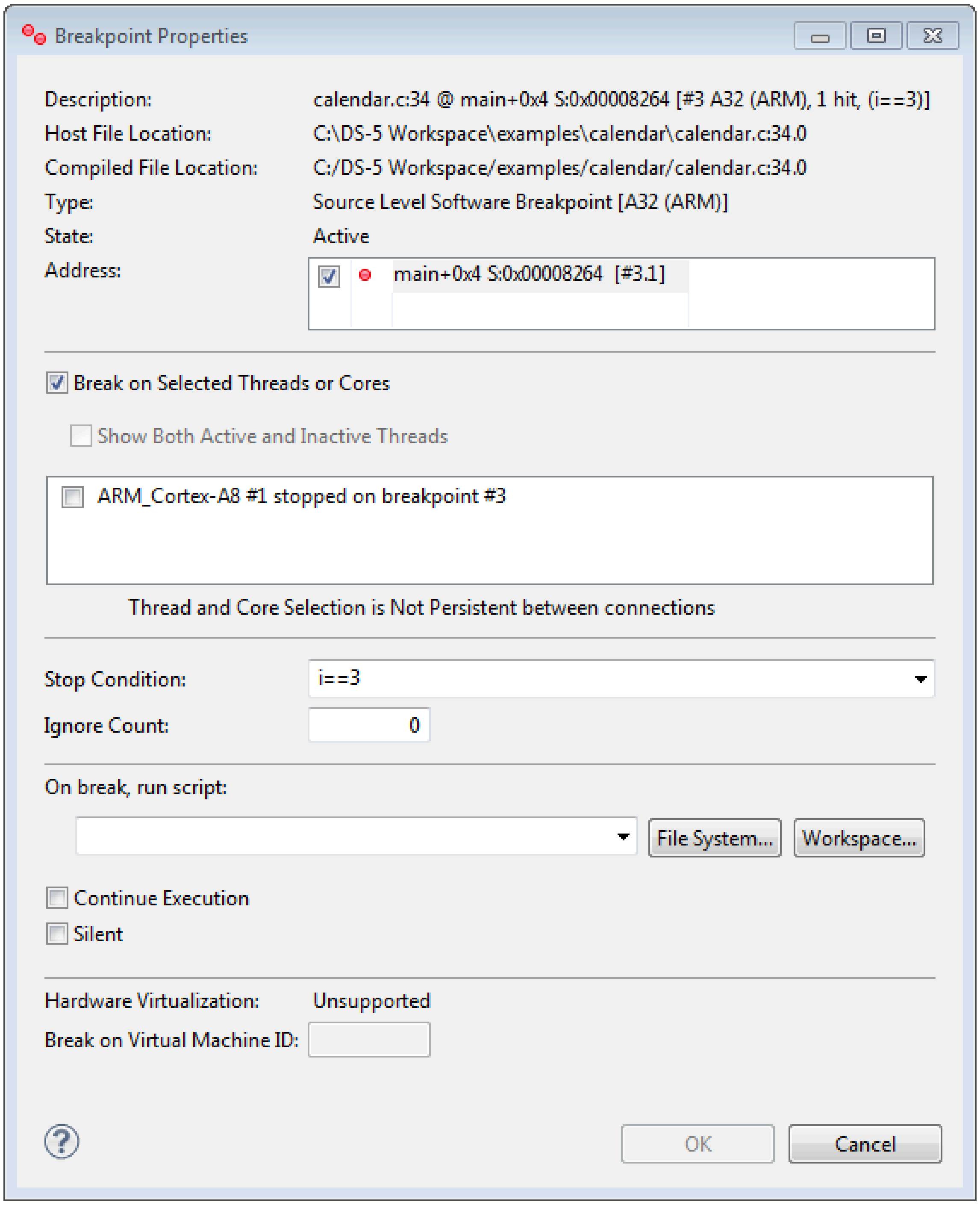
11.38 Breakpoint Properties dialog box

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 11-61: Breakpoint properties dialog box



Breakpoint Properties

Description: calendar.c:34 @ main+0x4 S:0x00008264 [#3 A32 (ARM), 1 hit, (i==3)]
Host File Location: C:\DS-5 Workspace\examples\calendar\calendar.c:34.0
Compiled File Location: C:/DS-5 Workspace/examples/calendar/calendar.c:34.0
Type: Source Level Software Breakpoint [A32 (ARM)]
State: Active
Address: ☑ ● main+0x4 S:0x00008264 [#3.1]

☑ Break on Selected Threads or Cores
☐ Show Both Active and Inactive Threads
☐ ARM_Cortex-A8 #1 stopped on breakpoint #3
Thread and Core Selection is Not Persistent between connections

Stop Condition: i==3
Ignore Count: 0

On break, run script:

☐ Continue Execution
☐ Silent

Hardware Virtualization: Unsupported
Break on Virtual Machine ID:

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.

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.

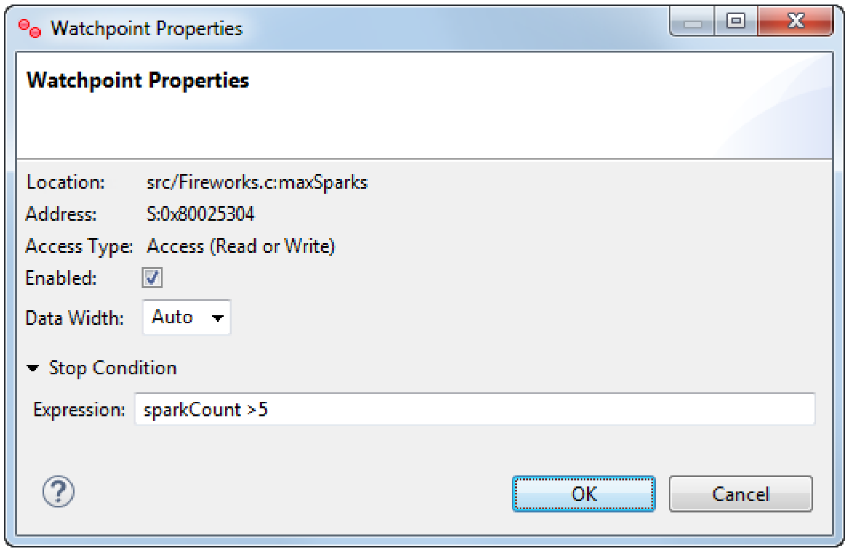
11.39 Watchpoint Properties dialog box

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 11-62: Watchpoint Properties dialog box



Location: src/Fireworks.c:maxSparks
Address: S:0x80025304
Access Type: Access (Read or Write)
Enabled: ✓
Data Width: Auto
Stop Condition
Expression: sparkCount >5

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.

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**.

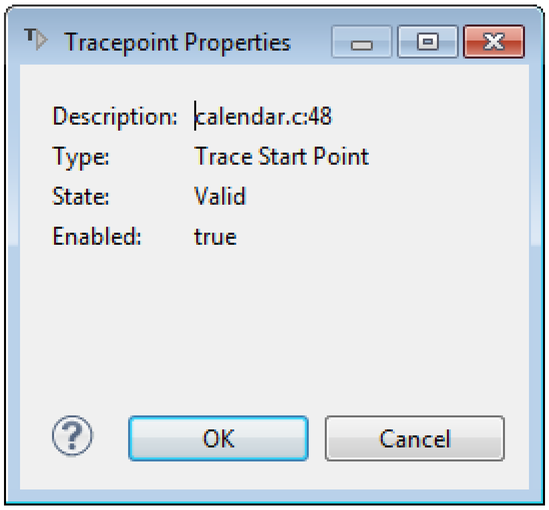
Note

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.

11.40 Tracepoint Properties dialog box

Use the **Tracepoint Properties** dialog box to display the properties of a tracepoint.

Figure 11-63: Tracepoint Properties dialog box



Description: calendar.c:48
Type: Trace Start Point
State: Valid
Enabled: true

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.

11.41 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

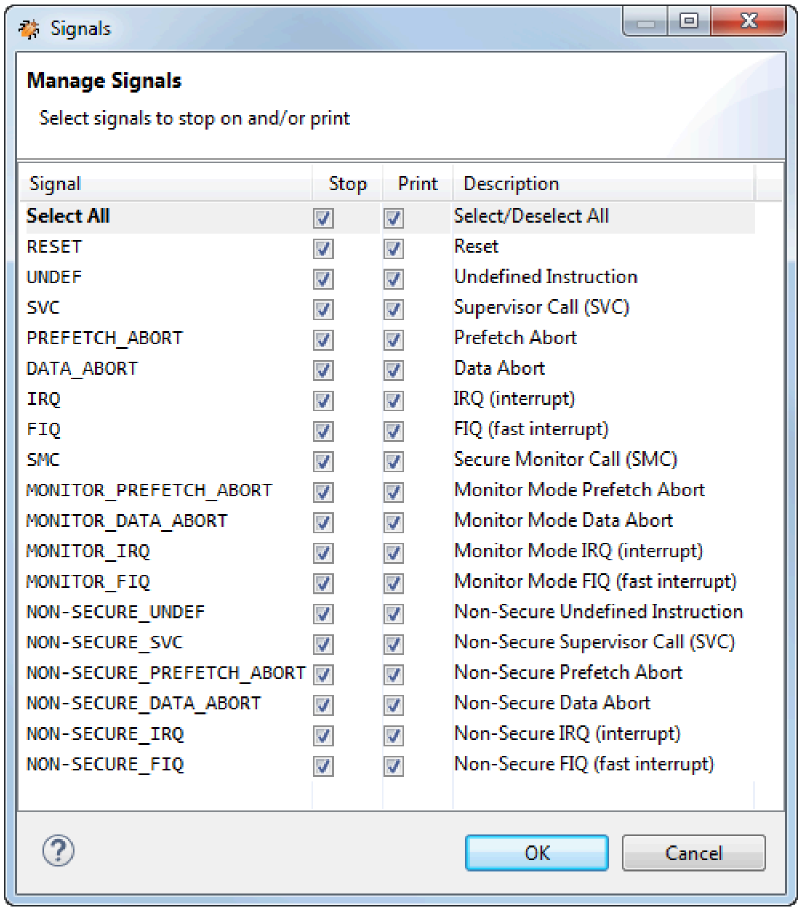
You can also use the info signals command to display the current signal handler settings.

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 11-64: Manage Signals dialog box



Manage Signals

Select signals to stop on and/or print

Signal	Stop	Print	Description
Select All	✓	✓	Select/Deselect All
RESET	✓	✓	Reset
UNDEF	✓	✓	Undefined Instruction
SVC	✓	✓	Supervisor Call (SVC)
PREFETCH_ABORT	✓	✓	Prefetch Abort
DATA_ABORT	✓	✓	Data Abort
IRQ	✓	✓	IRQ (interrupt)
FIQ	✓	✓	FIQ (fast interrupt)
SMC	✓	✓	Secure Monitor Call (SMC)
MONITOR_PREFETCH_ABORT	✓	✓	Monitor Mode Prefetch Abort
MONITOR_DATA_ABORT	✓	✓	Monitor Mode Data Abort
MONITOR_IRQ	✓	✓	Monitor Mode IRQ (interrupt)
MONITOR_FIQ	✓	✓	Monitor Mode FIQ (fast interrupt)
NON-SECURE_UNDEF	✓	✓	Non-Secure Undefined Instruction
NON-SECURE_SVC	✓	✓	Non-Secure Supervisor Call (SVC)
NON-SECURE_PREFETCH_ABORT	✓	✓	Non-Secure Prefetch Abort
NON-SECURE_DATA_ABORT	✓	✓	Non-Secure Data Abort
NON-SECURE_IRQ	✓	✓	Non-Secure IRQ (interrupt)
NON-SECURE_FIQ	✓	✓	Non-Secure FIQ (fast interrupt)

Related information

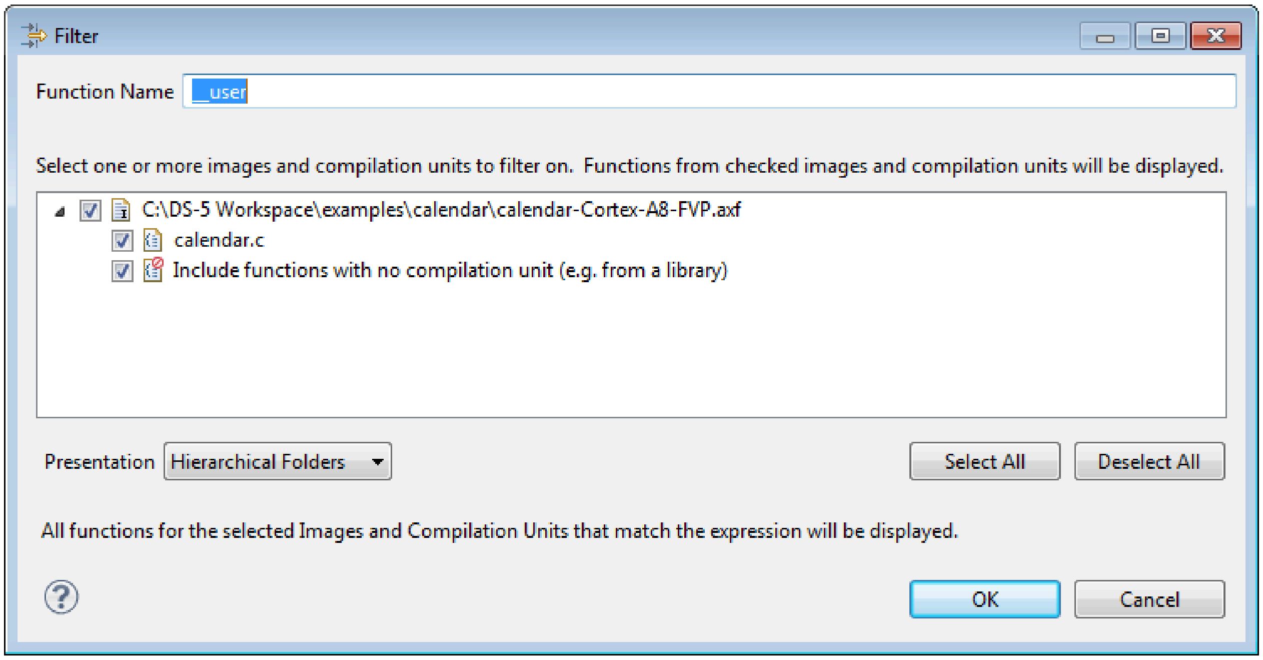
Handling UNIX signals on page 75
Handling processor exceptions on page 77
Breakpoints view on page 322

11.42 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 11-65: Function filter dialog box


Filter

Function Name: __user

Select one or more images and compilation units to filter on. Functions from checked images and compilation units will be displayed.

- [✓] C:\DS-5 Workspace\examples\calendar\calendar-Cortex-A8-FVP.axf
  - [✓] calendar.c
  - [✓] Include functions with no compilation unit (e.g. from a library)

Presentation: Hierarchical Folders

All functions for the selected Images and Compilation Units that match the expression will be displayed.

11.43 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.

Note

Arm® Development Studio resolves Eclipse variables when you export a debug configuration.

However, they are not resolved when you refer to them in a script, or when using the Commands view.

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.

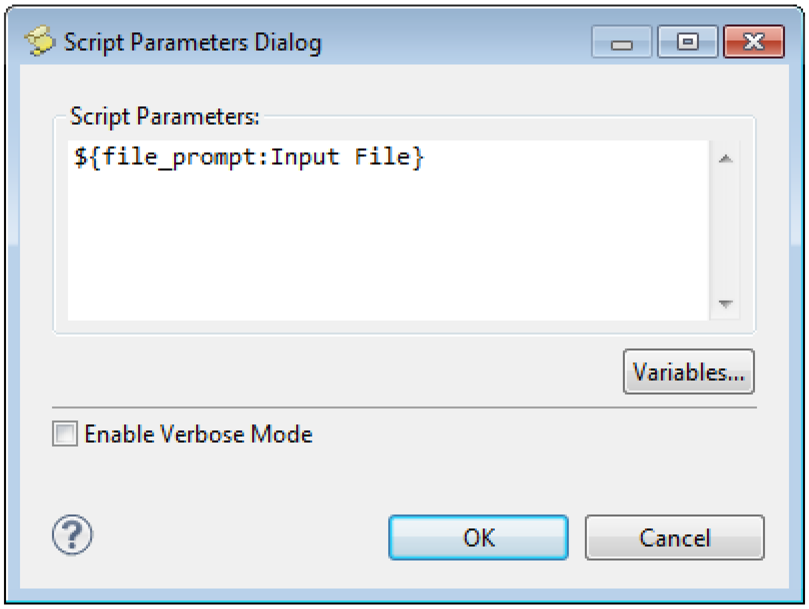
OK

Saves the current parameters and closes the Script Parameters dialog box.

Cancel

Closes the Script Parameters dialog box without saving the changes.

Figure 11-66: Script Parameters dialog box

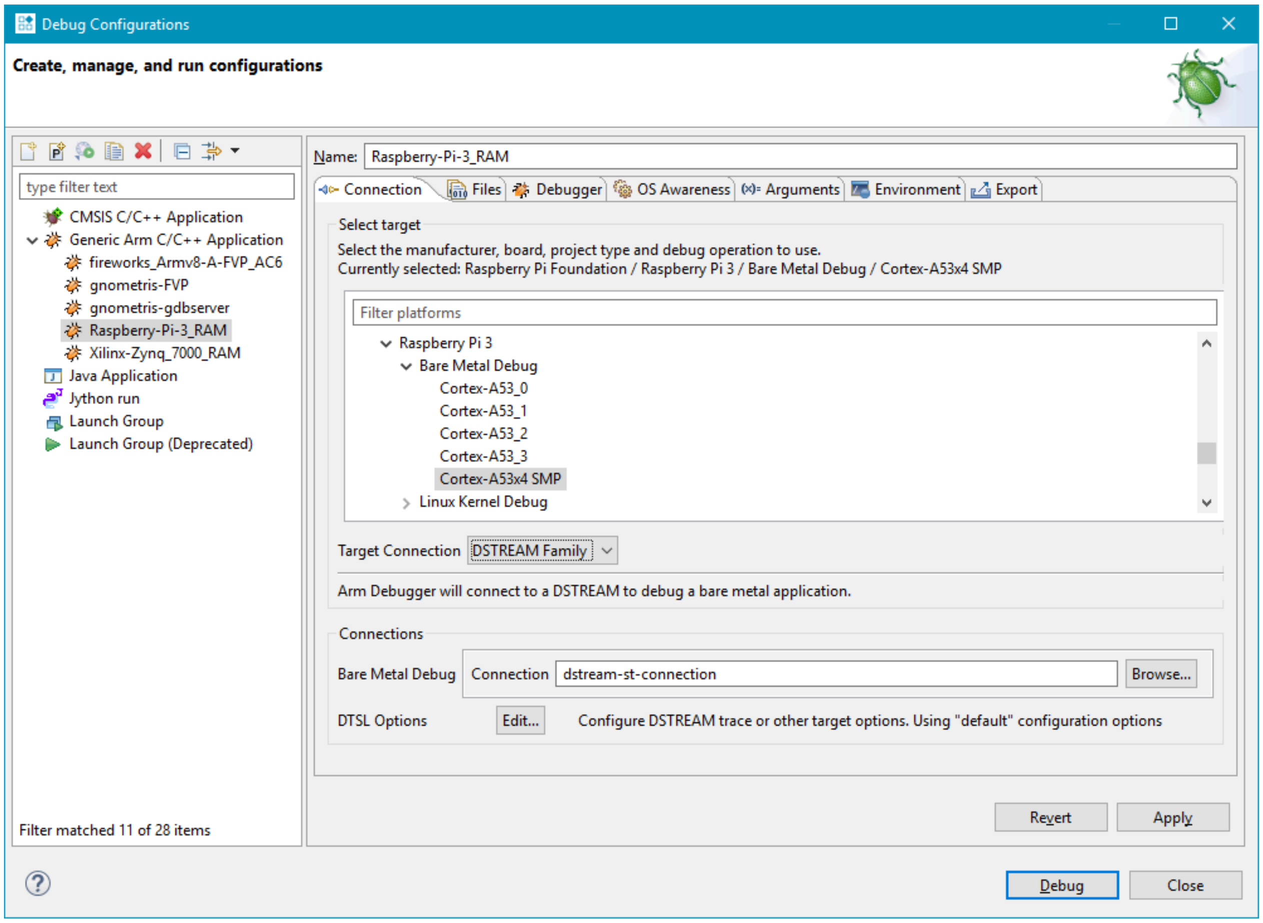


```
${file_prompt:Input File}
```

11.44 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 11-67: Connection tab



Name: Raspberry-Pi-3_RAM

Select target

Select the manufacturer, board, project type and debug operation to use.
Currently selected: Raspberry Pi Foundation / Raspberry Pi 3 / Bare Metal Debug / Cortex-A53x4 SMP

Target Connection: DSTREAM Family

Arm Debugger will connect to a DSTREAM to debug a bare metal application.

Connections

Bare Metal Debug — Connection: dstream-st-connection

DTSL Options — Configure DSTREAM trace or other target options. Using "default" configuration options

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

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.

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.

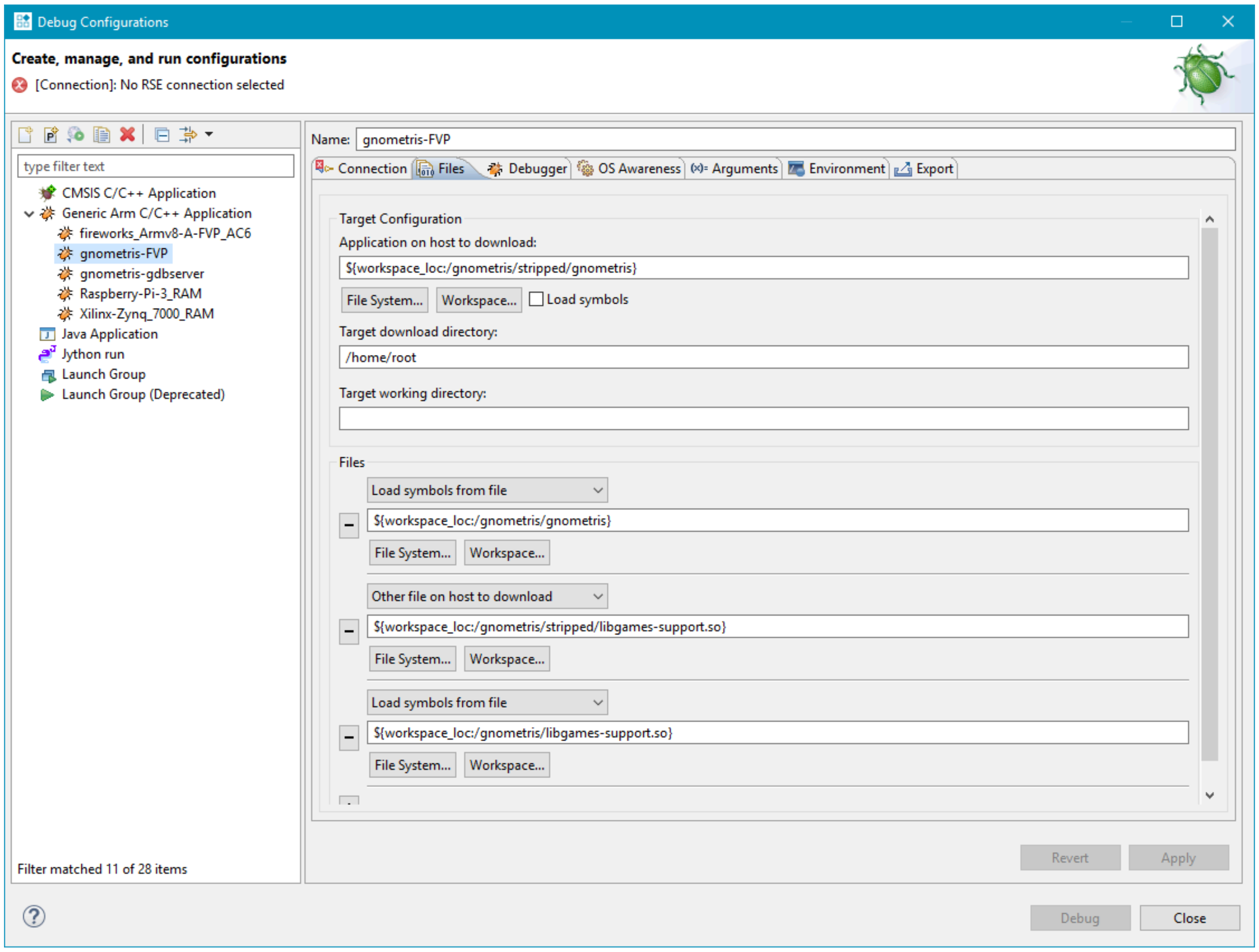
Close

Close the **Debug Configurations** dialog box.

11.45 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 11-68: Files tab (Shown with file system configuration for an application on a Fixed Virtual Platform)


Debug Configurations

Create, manage, and run configurations
[Connection]: No RSE connection selected

type filter text
- CMSIS C/C++ Application
- Generic Arm C/C++ Application
  - fireworks_Armv8-A-FVP_AC6
  - gnometris-FVP
  - gnometris-gdbserver
  - Raspberry-Pi-3_RAM
  - Xilinx-Zynq_7000_RAM
- Java Application
- Jython run
- Launch Group
- Launch Group (Deprecated)

Filter matched 11 of 28 items

Name: gnometris-FVP

Connection | Files | Debugger | OS Awareness | Arguments | Environment | Export

Target Configuration
Application on host to download:
${workspace_loc:/gnometris/stripped/gnometris}
File System... Workspace... Load symbols
Target download directory:
/home/root
Target working directory:

Files
Load symbols from file
${workspace_loc:/gnometris/gnometris}
File System... Workspace...
Other file on host to download
${workspace_loc:/gnometris/stripped/libgames-support.so}
File System... Workspace...
Load symbols from file
${workspace_loc:/gnometris/libgames-support.so}
File System... Workspace...

Revert Apply
Debug Close

Note

Options in the **Files** tab depend on the type of platform and debug operation that you select.

Files

These options enable you to configure the target file system and select files on the host that you want to download to the target or use by the debugger. The **Files** tab options available for each **Debug operation** are:

Table 11-3: 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.

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. You can use either a `.svd` or a `.tcf` configuration file type for defining peripherals.

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.

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.

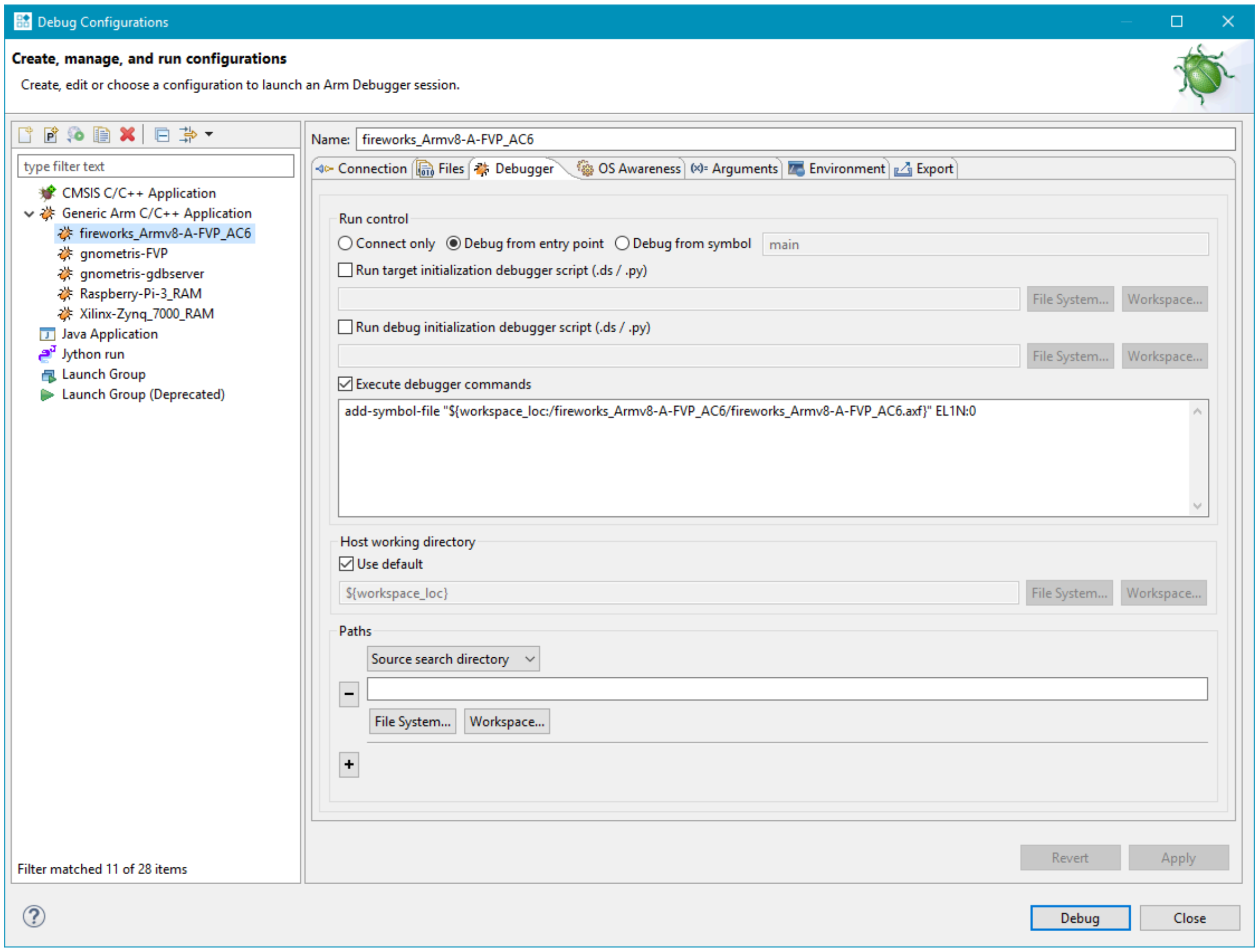
Add a new resource to the list

To add a new resource to the file settings, click this button and then configure the options as required.

11.46 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 11-69: Debugger tab (Shown with settings for application starting point and search paths)



Name: fireworks_Armv8-A-FVP_AC6

Run control
( ) Connect only (●) Debug from entry point ( ) Debug from symbol main
☐ Run target initialization debugger script (.ds / .py)
☐ Run debug initialization debugger script (.ds / .py)
☑ Execute debugger commands

```
add-symbol-file "${workspace_loc:/fireworks_Armv8-A-FVP_AC6/fireworks_Armv8-A-FVP_AC6.axf}" EL1N:0
```

Host working directory
☑ Use default
${workspace_loc}

Paths
Source search directory

Run Control

These options enable you to define the running state of the target when you connect:

Connect only

Connect to the target, but do not run the application.

Note

The PC register is not set and pending breakpoints or watchpoints are subsequently disabled when a connection is established.

Debug from entry point

Run the application when a connection is established, then stop at the image entry point.

Debug from symbol

Run the application when a connection is established, then stop at the address of the specified symbol. The debugger must be able to resolve the symbol. If you specify a C or C++ function name, then do not use the `()` suffix.

If the symbol can be resolved, execution stops at the address of that symbol.

If the symbol cannot be resolved, a message is displayed in the **Commands** view warning that the symbol cannot be found. The debugger then attempts to stop at the image entry point.

Run target initialization debugger script (.ds / .py)

Select this option to execute target initialization scripts (a file containing debugger commands) immediately after connection. To select a file:

- Enter the location and file name in the field provided.
- Click on **File System...** to locate the file in an external directory from the workspace.
- Click on **Workspace...** to locate the file in a project directory or sub-directory within the workspace.

Run debug initialization debugger script (.ds / .py)

Select this option to execute debug initialization scripts (a file containing debugger commands) after execution of any target initialization scripts and also running to an image entry point or symbol, if selected. To select a file:

- Enter the location and file name in the field provided.
- Click **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.

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.

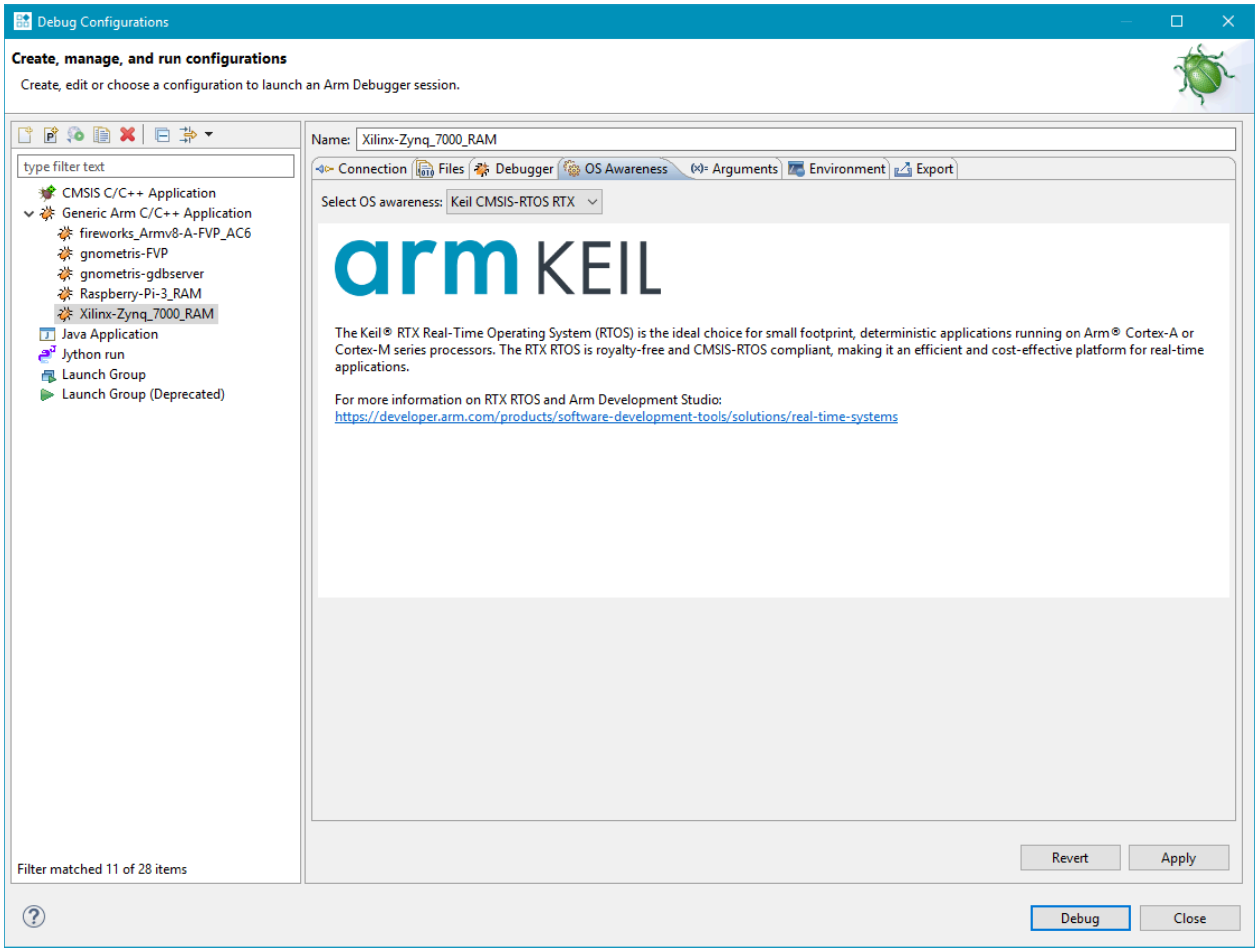
Close

Close the **Debug Configurations** dialog box.

11.47 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 11-70: OS Awareness tab


Debug Configurations
Create, manage, and run configurations
Create, edit or choose a configuration to launch an Arm Debugger session.
type filter text
CMSIS C/C++ Application
Generic Arm C/C++ Application
fireworks_Armv8-A-FVP_AC6
gnometris-FVP
gnometris-gdbserver
Raspberry-Pi-3_RAM
Xilinx-Zynq_7000_RAM
Java Application
Jython run
Launch Group
Launch Group (Deprecated)
Filter matched 11 of 28 items
Name: Xilinx-Zynq_7000_RAM
Connection
Files
Debugger
OS Awareness
Arguments
Environment
Export
Select OS awareness: Keil CMSIS-RTOS RTX
arm KEIL
Revert
Apply
Debug
Close

The Keil® RTX Real-Time Operating System (RTOS) is the ideal choice for small footprint, deterministic applications running on Arm® Cortex-A or Cortex-M series processors. The RTX RTOS is royalty-free and CMSIS-RTOS compliant, making it an efficient and cost-effective platform for real-time applications.

For more information on RTX RTOS and Arm Development Studio:
https://developer.arm.com/products/software-development-tools/solutions/real-time-systems

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.

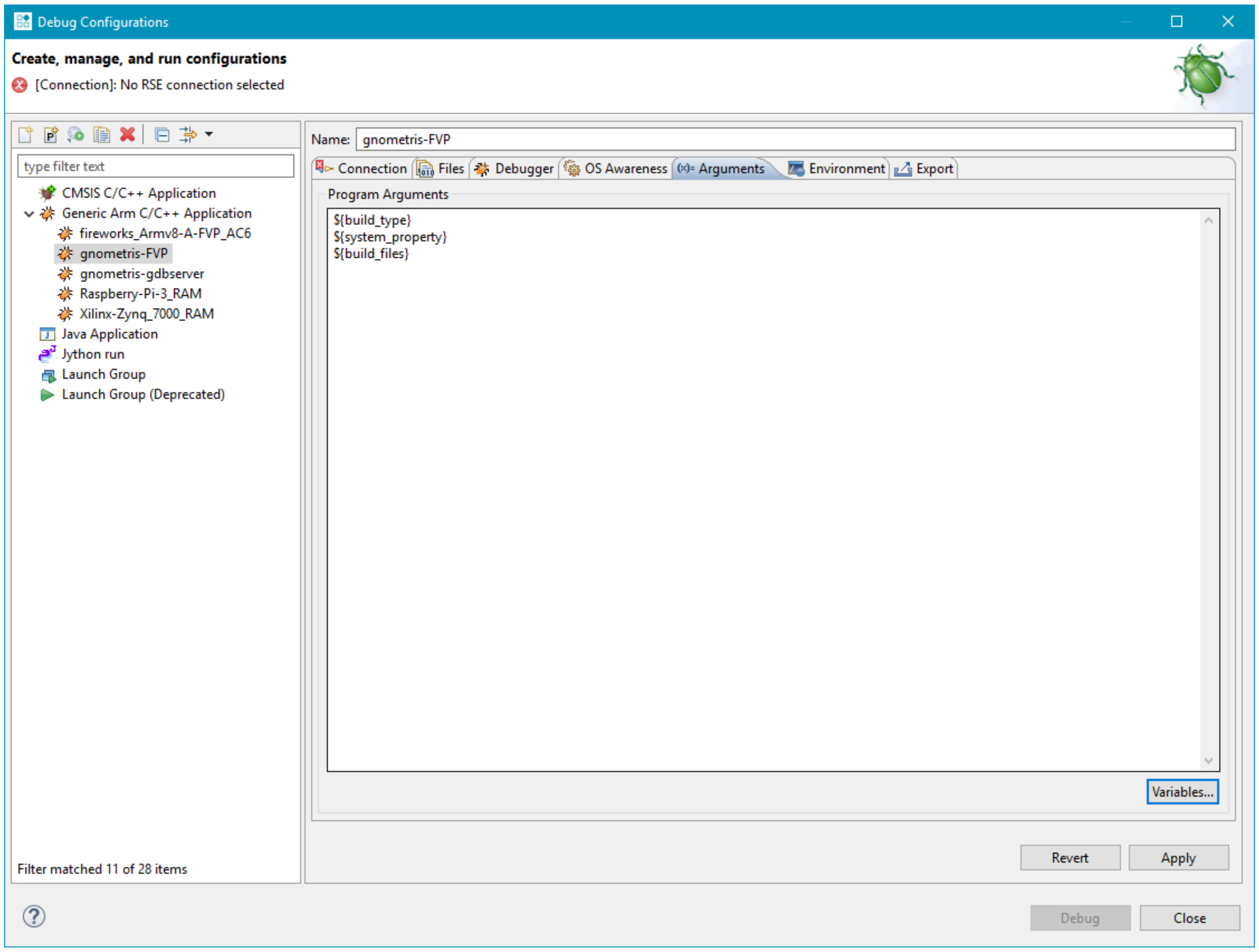
Note

Linux OS awareness is not currently available in this tab, and remains in the **Connection** tab as a separate debug operation.

11.48 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 11-71: Arguments tab



```
${build_type}
${system_property}
${build_files}
```

The **Arguments** tab contains the following elements:

Note

These settings only apply if the target supports semihosting and they cannot be changed while the target is running.

Program Arguments

This panel enables you to enter the arguments. Arguments are separated by spaces. They are passed to the target application unmodified except when the text is an Eclipse argument variable of the form `${<var_name>}` where Eclipse replaces it with the related value.

For a Linux target, you might have to escape some characters using a backslash (`\\`) character. For example, the `@`, `(`, `)`, `"`, and `#` characters must be escaped.

Variables...

This button opens the **Select Variable** dialog box where you can select variables that are passed to the application when the debug session starts. For more information on variables, use the dynamic help.

Apply

Save the current configuration. This does not connect to the target.

Revert

Undo any changes and revert to the last saved configuration.

Debug

Connect to the target and close the **Debug Configurations** dialog box.

Close

Close the **Debug Configurations** dialog box.

Related information

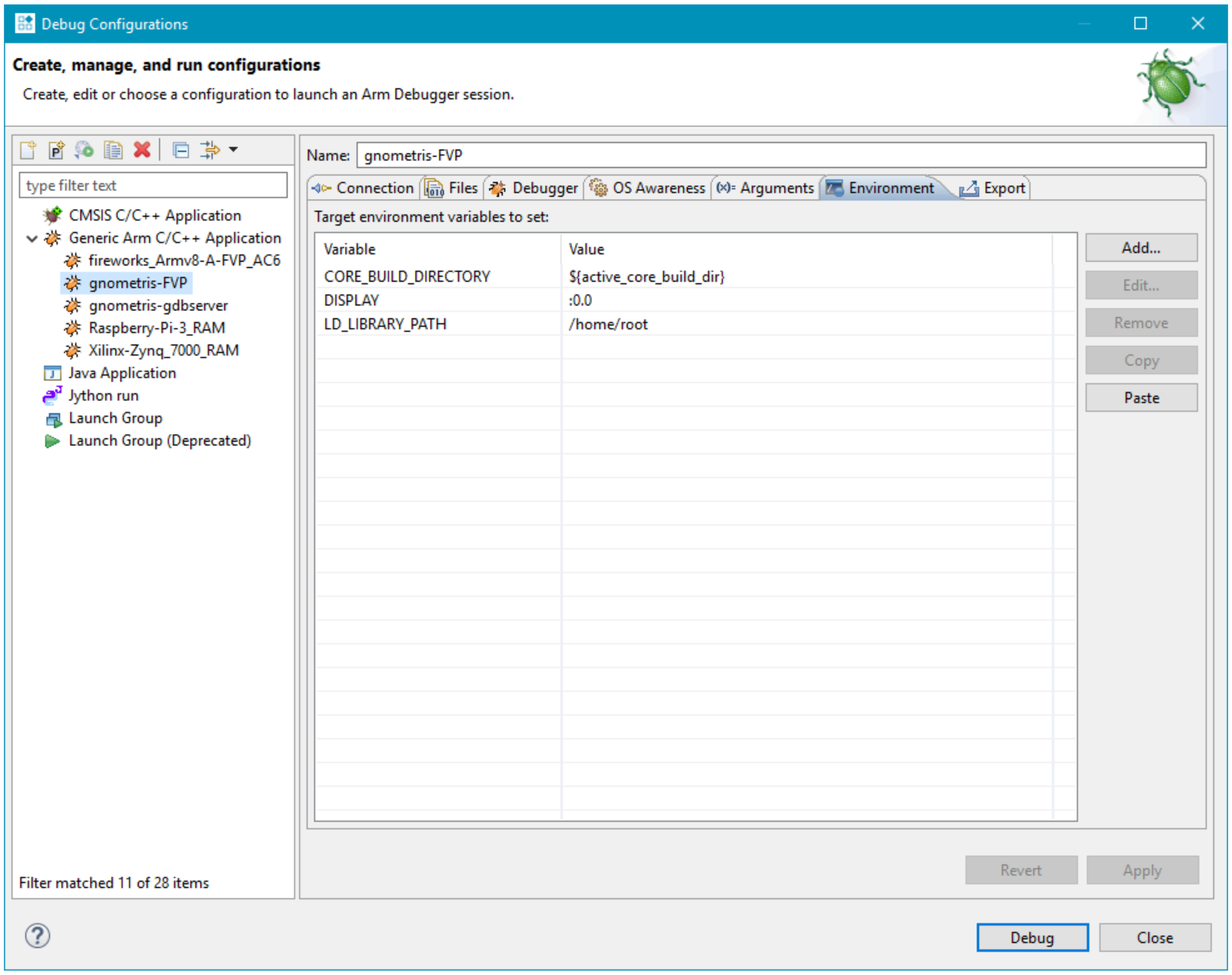
Using semihosting to access resources on the host computer on page 80

Working with semihosting on page 82

11.49 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 11-72: Environment tab (Shown with environment variables configured for a Fixed Virtual Platform)



Debug Configurations

Create, manage, and run configurations

Create, edit or choose a configuration to launch an Arm Debugger session.

type filter text

- CMSIS C/C++ Application
- Generic Arm C/C++ Application
  - fireworks_Armv8-A-FVP_AC6
  - gnometris-FVP
  - gnometris-gdbserver
  - Raspberry-Pi-3_RAM
  - Xilinx-Zynq_7000_RAM
- Java Application
- Jython run
- Launch Group
- Launch Group (Deprecated)

Filter matched 11 of 28 items

Name: gnometris-FVP

Connection | Files | Debugger | OS Awareness | Arguments | Environment | Export

Target environment variables to set:

Variable	Value
CORE_BUILD_DIRECTORY	${active_core_build_dir}
DISPLAY	:0.0
LD_LIBRARY_PATH	/home/root

The **Environment** tab contains the following elements:

Note

The settings in this tab are not used for connections that use the **Connect to already running gdbserver** debug operation.

Target environment variables to set

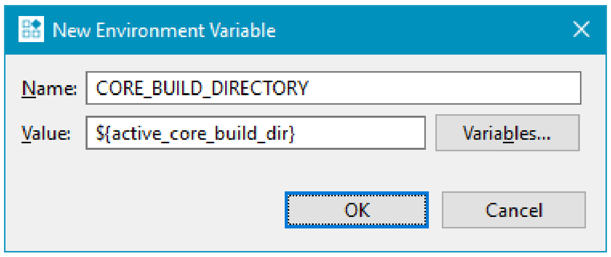
This panel displays the target environment variables in use by the debugger.

New...

Opens the **New Environment Variable** dialog box where you can create a new target environment variable.

For example, 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 11-73: New Environment Variable dialog box



New Environment Variable

Name: CORE_BUILD_DIRECTORY

Value: ${active_core_build_dir}

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.

11.50 Debug Configurations - Export tab

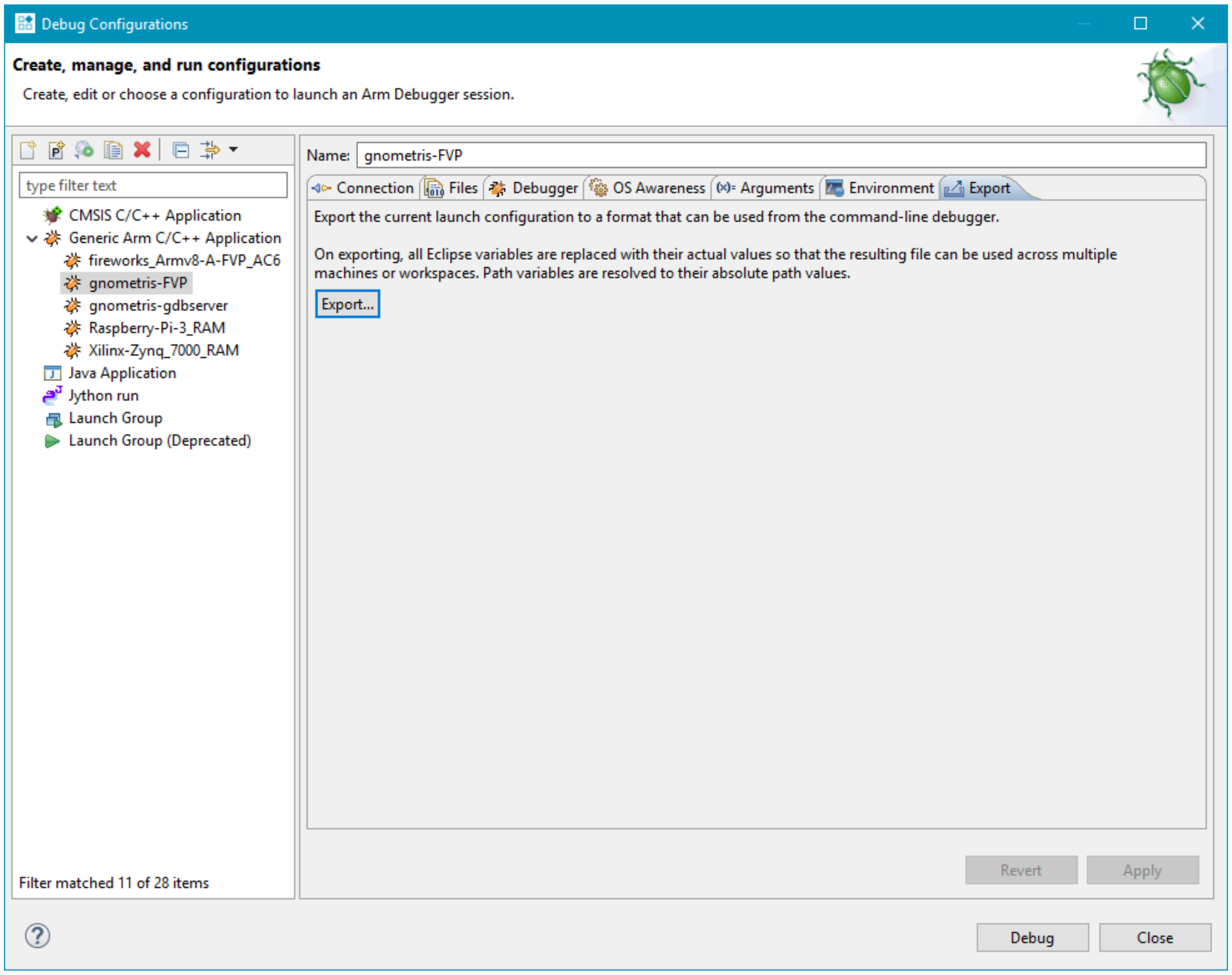
An Arm® Development Studio debug launch configuration typically describes the target to connect to, the communication protocol or probe to use, the application to load on the target, and debug information to load in the debugger. Use the **Export** tab in the **Debug Configurations** dialog box to export 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.

Note

Arm Development Studio does not resolve Eclipse variables when scripting or using the Commands view.

Figure 11-74: Export tab



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.

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 to load the launch configuration from the command-line debugger.

For example, on Windows platforms: `debugger --launch-config "C:\Workspace \debugconfiguration.cli"`

11.51 DTSL Configuration Editor dialog box

Use the Debug and Trace Services Layer (DTSL) Configuration Editor to configure additional debug and trace settings. The configuration options available depend on the capabilities of the target. Typically, they enable configuration of the trace collection method and the trace that is generated.

A typical set of configuration options might include:

Trace Capture

Select the collection method that you want to use for this debug configuration. The available trace collection methods depend on the target and trace capture unit but can include Embedded Trace Buffer (ETB)/Micro Trace Buffer (MTB) (trace collected from an on-chip buffer) or DSTREAM (trace collected from the DSTREAM trace buffer). If no trace collection method is selected then no trace can be collected, even if the trace capture for processors and Instruction Trace Macrocell (ITM) are enabled.

Core Trace

Enable or disable trace collection. If enabled then the following options are available:

`<Enable core n trace>`

Specify trace capture for specific processors.

Cycle accurate trace

Enable or disable cycle accurate trace.

Trace capture range

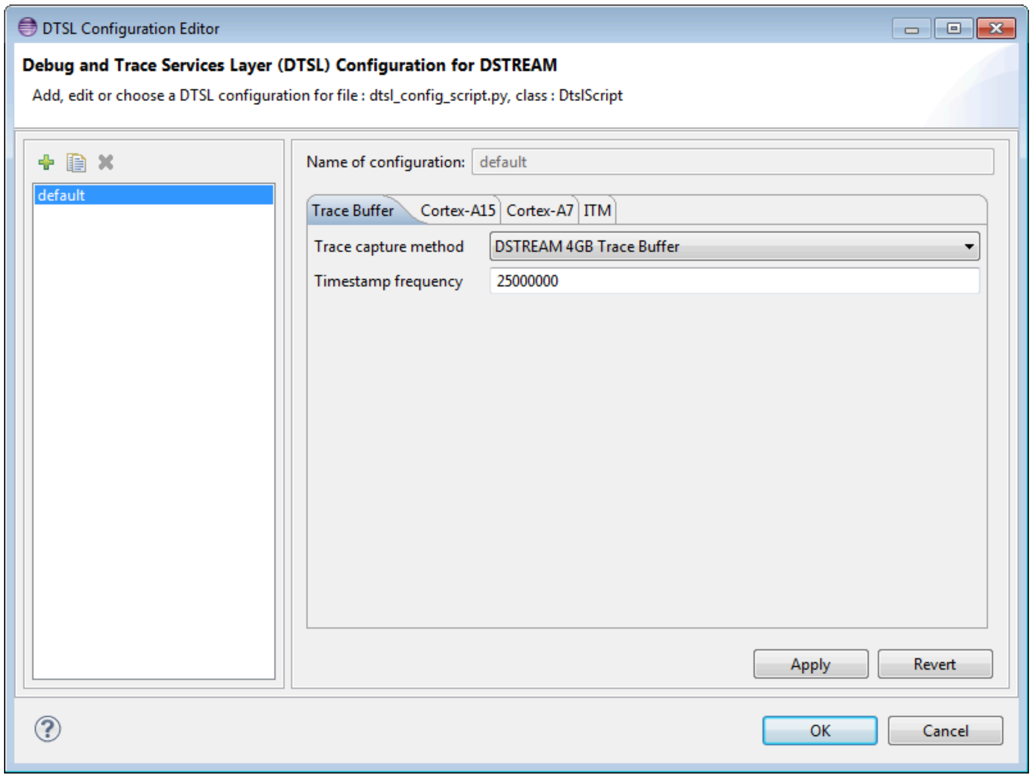
Specify an address range to limit the trace capture.

ITM

Enable or disable trace collection from the ITM unit.

Named DTSL configuration profiles can be saved for later use.

Figure 11-75: Configuration Editor (Shown with Trace capture method set to DSTREAM)



Debug and Trace Services Layer (DTSL) Configuration for DSTREAM

Add, edit or choose a DTSL configuration for file : dtsl_config_script.py, class : DtslScript

Name of configuration: default

Trace capture method: DSTREAM 4GB Trace Buffer

Timestamp frequency: 25000000

Related information

Cache Data view on page 387

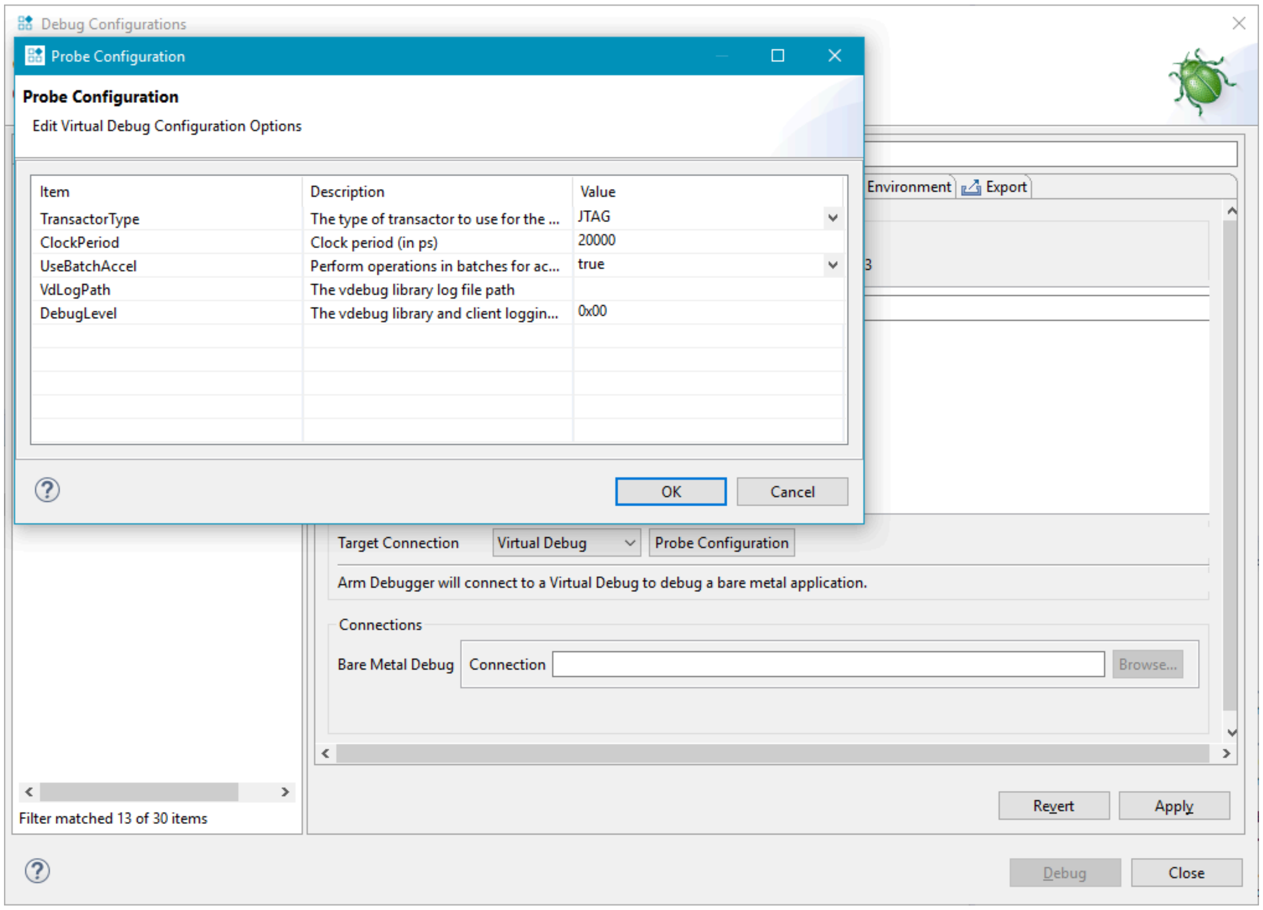
About debugging caches on page 45

cache commands

11.52 Probe Configuration dialog box

Edit and save configuration options for your probe using the **Probe Configuration** dialog box.

Figure 11-76: Probe Configuration dialog box



Probe Configuration

Edit Virtual Debug Configuration Options

Item	Description	Value
TransactorType	The type of transactor to use for the ...	JTAG
ClockPeriod	Clock period (in ps)	20000
UseBatchAccel	Perform operations in batches for ac...	true
VdLogPath	The vdebug library log file path	
DebugLevel	The vdebug library and client loggin...	0x00

Target Connection: Virtual Debug

Arm Debugger will connect to a Virtual Debug to debug a bare metal application.

Filter matched 13 of 30 items

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.

Related information

Add a debug connection over functional I/O on page 278

11.53 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**.

11.54 Remote Systems view

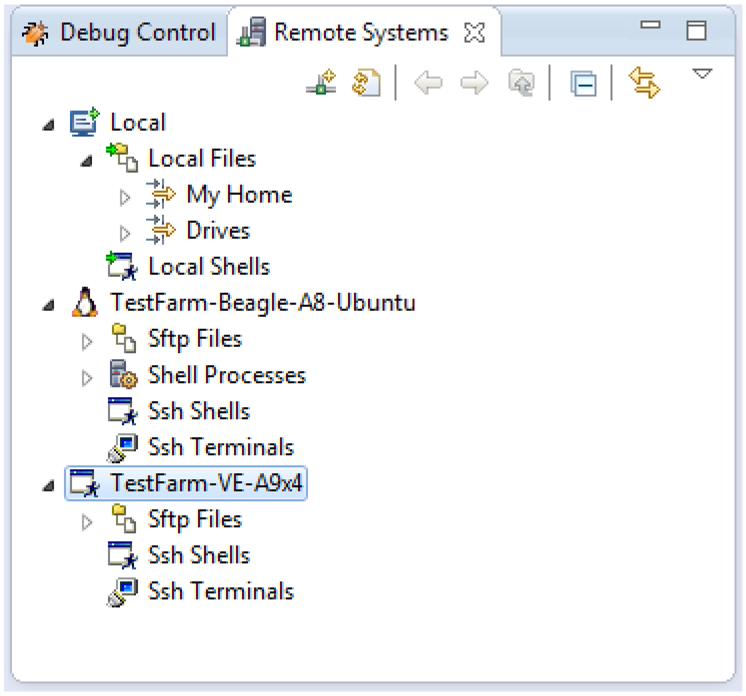
The **Remote Systems** view is a hierarchical tree view of local and remote systems.

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 11-77: Remote Systems view



11.55 Remote System Details view

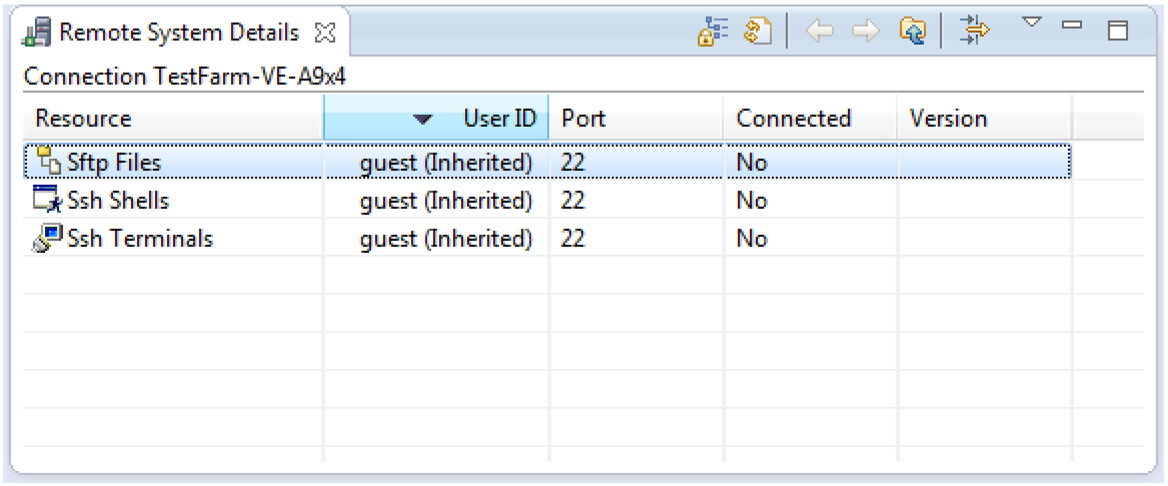
The **Remote System Details** view is a tabular view giving details about local and remote systems.

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 11-78: Remote System Details view



Connection TestFarm-VE-A9x4

Resource	User ID	Port	Connected	Version
Sftp Files	guest (Inherited)	22	No	
Ssh Shells	guest (Inherited)	22	No	
Ssh Terminals	guest (Inherited)	22	No	

The **Remote System Details** view is not visible by default. To add this view:

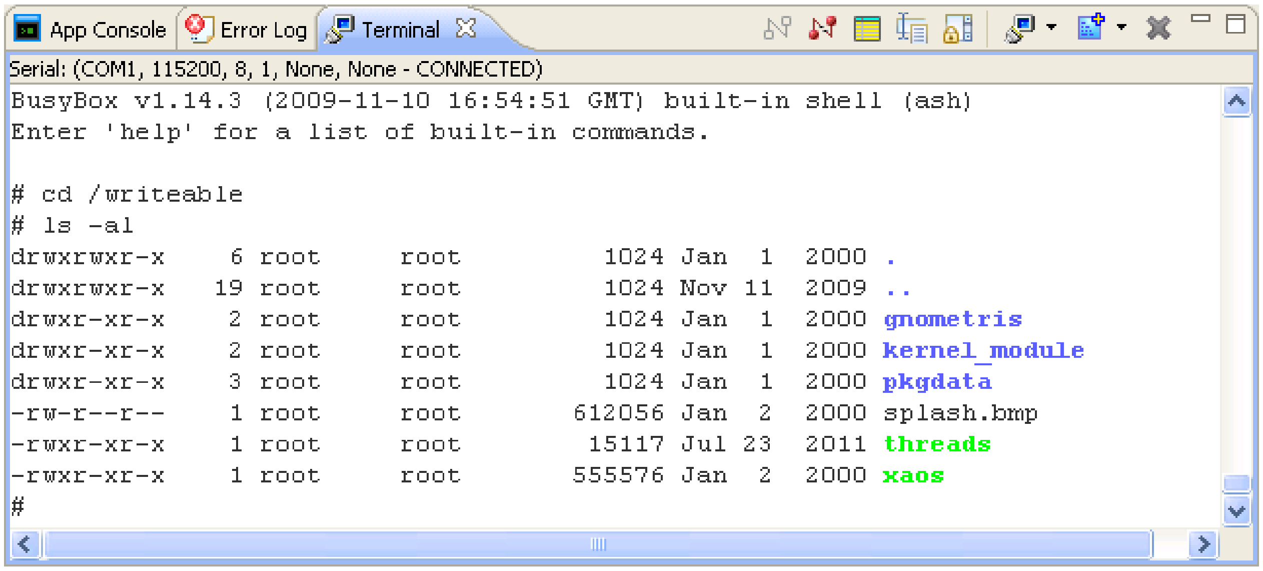
1. Select **Window** > **Show View** > **Other...** to open the **Show View** dialog box.
2. Expand the **Remote Systems** group and select **Remote System Details**.
3. Click **OK**.

11.56 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 11-79: Terminal view



```
Serial: (COM1, 115200, 8, 1, None, None - CONNECTED)
BusyBox v1.14.3 (2009-11-10 16:54:51 GMT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# cd /writeable
# ls -al
drwxrwxr-x    6 root     root          1024 Jan  1  2000 .
drwxrwxr-x   19 root     root          1024 Nov 11  2009 ..
drwxr-xr-x    2 root     root          1024 Jan  1  2000 gnometris
drwxr-xr-x    2 root     root          1024 Jan  1  2000 kernel_module
drwxr-xr-x    3 root     root          1024 Jan  1  2000 pkgdata
-rw-r--r--    1 root     root        612056 Jan  2  2000 splash.bmp
-rwxr-xr-x    1 root     root         15117 Jul 23  2011 threads
-rwxr-xr-x    1 root     root        555576 Jan  2  2000 xaos
#
```

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**.

Related information

Configuring a connection to a Linux application using gdbserver
Configuring a connection to a Linux kernel

11.57 Remote Scratchpad view

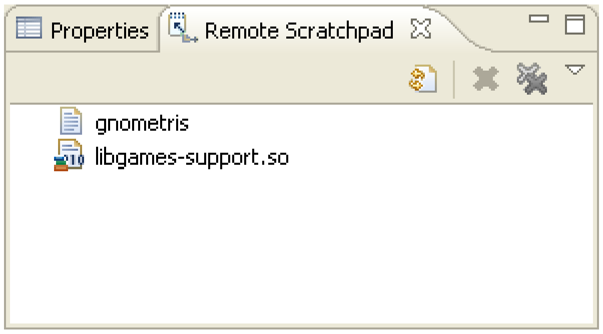
Use the **Remote Scratchpad** view as an electronic clipboard. You can copy and paste or drag and drop useful files and folders into it for later use.

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 11-80: Remote Scratchpad



The **Remote Scratchpad** view is not visible by default. To add this view:

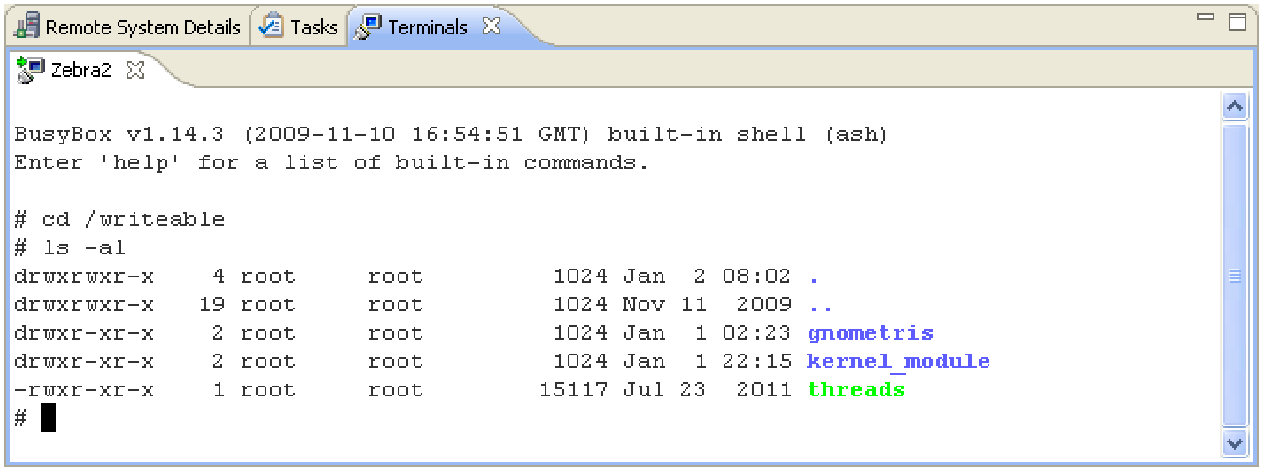
1. Select **Window** > **Show View** > **Other...** to open the Show View dialog box.
2. Expand the **Remote Systems** group and select **Remote Scratchpad**.
3. Click **OK**.

11.58 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 11-81: Remote Systems terminal



```
BusyBox v1.14.3 (2009-11-10 16:54:51 GMT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# cd /writeable
# ls -al
drwxrwxr-x    4 root     root          1024 Jan  2 08:02 .
drwxrwxr-x   19 root     root          1024 Nov 11  2009 ..
drwxr-xr-x    2 root     root          1024 Jan  1 02:23 gnometris
drwxr-xr-x    2 root     root          1024 Jan  1 22:15 kernel_module
-rwxr-xr-x    1 root     root         15117 Jul 23  2011 threads
#
```

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.

11.59 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

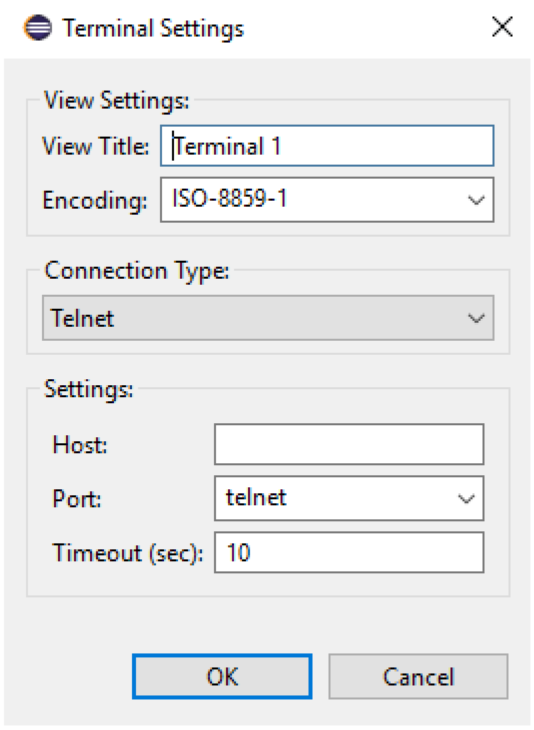
Enter a name for the **Terminal** view.

Encoding

Select the character set encoding for the terminal.

Terminal Settings for Telnet

Figure 11-82: Terminal Settings (Telnet) dialog box



View Title: Terminal 1
Encoding: ISO-8859-1
Telnet
Host:
Port: telnet
Timeout (sec): 10

Host

The host to connect to.

Port

The port that the target is connected to:

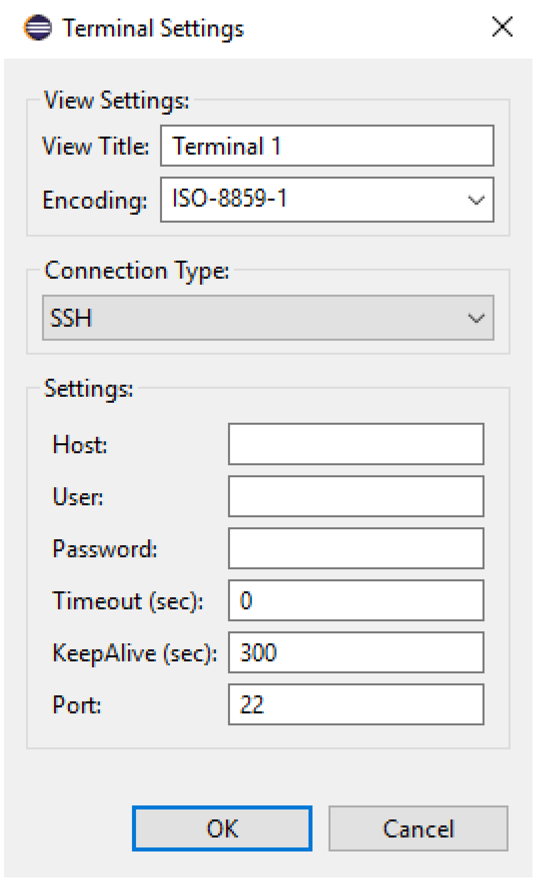
- `telnet`. This is the default.
- `tgtcons`.

Timeout (sec)

The connection's timeout in seconds.

Terminal Settings for SSH

Figure 11-83: Terminal Settings (SSH) dialog box


Terminal Settings

View Settings:

View Title: Terminal 1

Encoding: ISO-8859-1

Connection Type:

SSH

Settings:

Host:

User:

Password:

Timeout (sec): 0

KeepAlive (sec): 300

Port: 22

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)

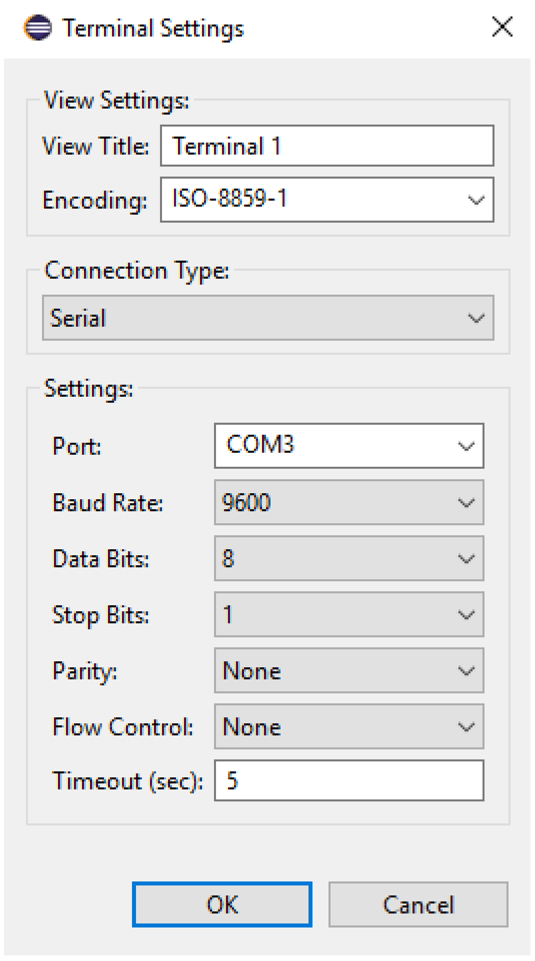
The connection's keep alive time in seconds. Defaults to 300.

Port

The port that the target is connected to. Defaults to 22.

Terminal Settings for Serial

Figure 11-84: Terminal Settings (Serial) dialog box



Terminal Settings

View Settings:

View Title: Terminal 1

Encoding: ISO-8859-1

Connection Type:

Serial

Settings:

Port: COM3

Baud Rate: 9600

Data Bits: 8

Stop Bits: 1

Parity: None

Flow Control: None

Timeout (sec): 5

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.

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.

11.60 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.

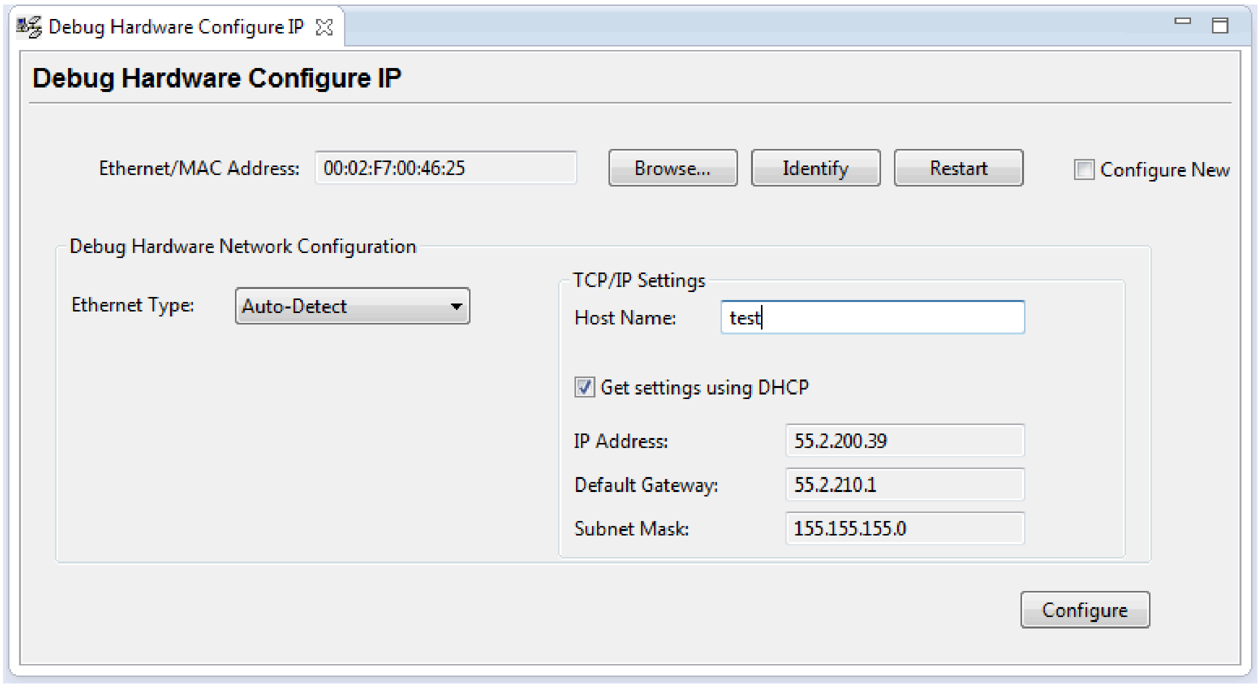
Note

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 11-85: Debug Hardware Configure IP view



Debug Hardware Configure IP

Ethernet/MAC Address: 00:02:F7:00:46:25

Configure New ☐

Debug Hardware Network Configuration

Ethernet Type: Auto-Detect

TCP/IP Settings

Host Name: test

☑ Get settings using DHCP

IP Address: 55.2.200.39

Default Gateway: 55.2.210.1

Subnet Mask: 155.155.155.0

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**.

Contents

Table 11-4: 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**.

Field	Description
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 information

Debug Hardware Firmware Installer view on page 462
Connection Browser dialog box on page 465
Debug Hardware configuration on page 267

11.61 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.

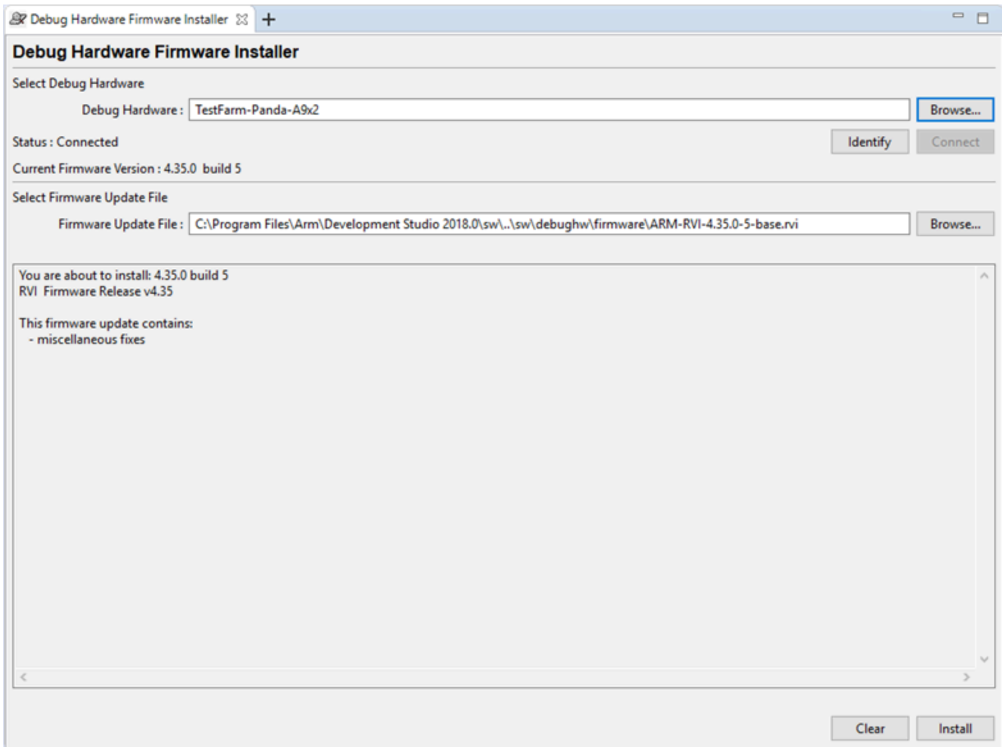
To access the **Debug Hardware Firmware Installer** view, from the main menu, select **Window** > **Show View** > **Other** > **Arm Debugger** > **Debug Hardware Firmware Installer**.

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 11-86: Debug Hardware Firmware Installer view



Debug Hardware Firmware Installer

Select Debug Hardware

Debug Hardware : TestFarm-Panda-A9x2

Status : Connected

Current Firmware Version : 4.35.0 build 5

Select Firmware Update File

Firmware Update File : C:\Program Files\Arm\Development Studio 2018.0\sw\..\sw\debughw\firmware\ARM-RVI-4.35.0-5-base.rvi

```
You are about to install: 4.35.0 build 5
RVI  Firmware Release v4.35

This firmware update contains:
  - miscellaneous fixes
```

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:

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 information

Debug Hardware Configure IP view on page 460
Connection Browser dialog box on page 465
Updating multiple debug hardware units on page 587

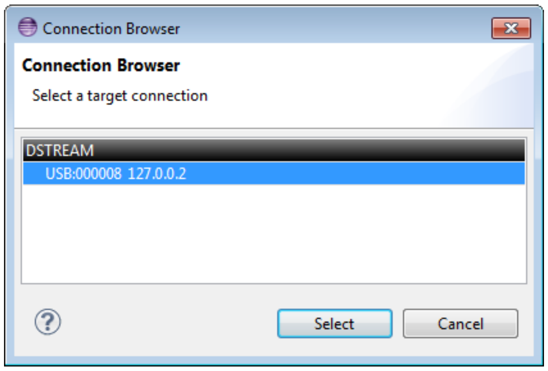
11.62 Connection Browser dialog box

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 11-87: Connection Browser (Showing a USB connected DSTREAM)


Connection Browser

Select a target connection

DSTREAM
USB:000008 127.0.0.2

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 information

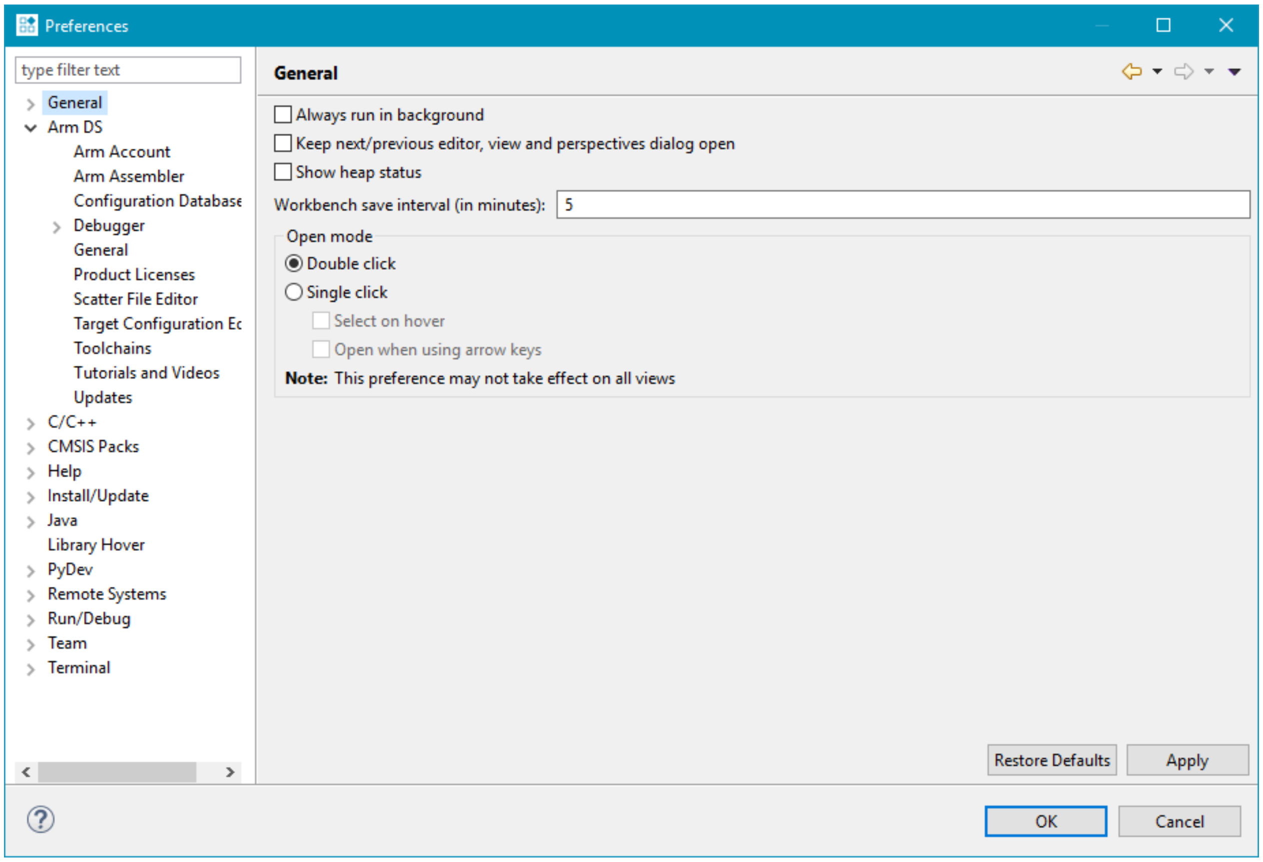
Debug Hardware Configure IP view on page 460

Debug Hardware Firmware Installer view on page 462

11.63 Preferences dialog box

You can customize your preferences using the **Preferences** dialog box.

Figure 11-88: Window preferences dialog box


Preferences

type filter text

- General
- Arm DS
  - Arm Account
  - Arm Assembler
  - Configuration Database
  - Debugger
  - General
  - Product Licenses
  - Scatter File Editor
  - Target Configuration Ec
  - Toolchains
  - Tutorials and Videos
  - Updates
- C/C++
- CMSIS Packs
- Help
- Install/Update
- Java
- Library Hover
- PyDev
- Remote Systems
- Run/Debug
- Team
- Terminal

General

☐ Always run in background
☐ Keep next/previous editor, view and perspectives dialog open
☐ Show heap status
Workbench save interval (in minutes): 5

Open mode
● Double click
○ Single click
☐ Select on hover
☐ Open when using arrow keys
Note: This preference may not take effect on all views

Restore Defaults | Apply
OK | Cancel

To access the **Preferences** dialog box, select **Preferences** from the **Window** menu. Changes to these preferences are only saved in the current workspace. If you want to copy your preferences to another workspace, select **File** > **Export...** to open the **Export** wizard. Then select **General** > **Preferences** and choose the location you want to export your preferences to.

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 and launch the **Preferences Wizard**.

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.

Install/Update

Controls the update history, scheduler, and policy.

Remote Systems

Controls the settings used by the **Remote System Explorer**.

Run/Debug

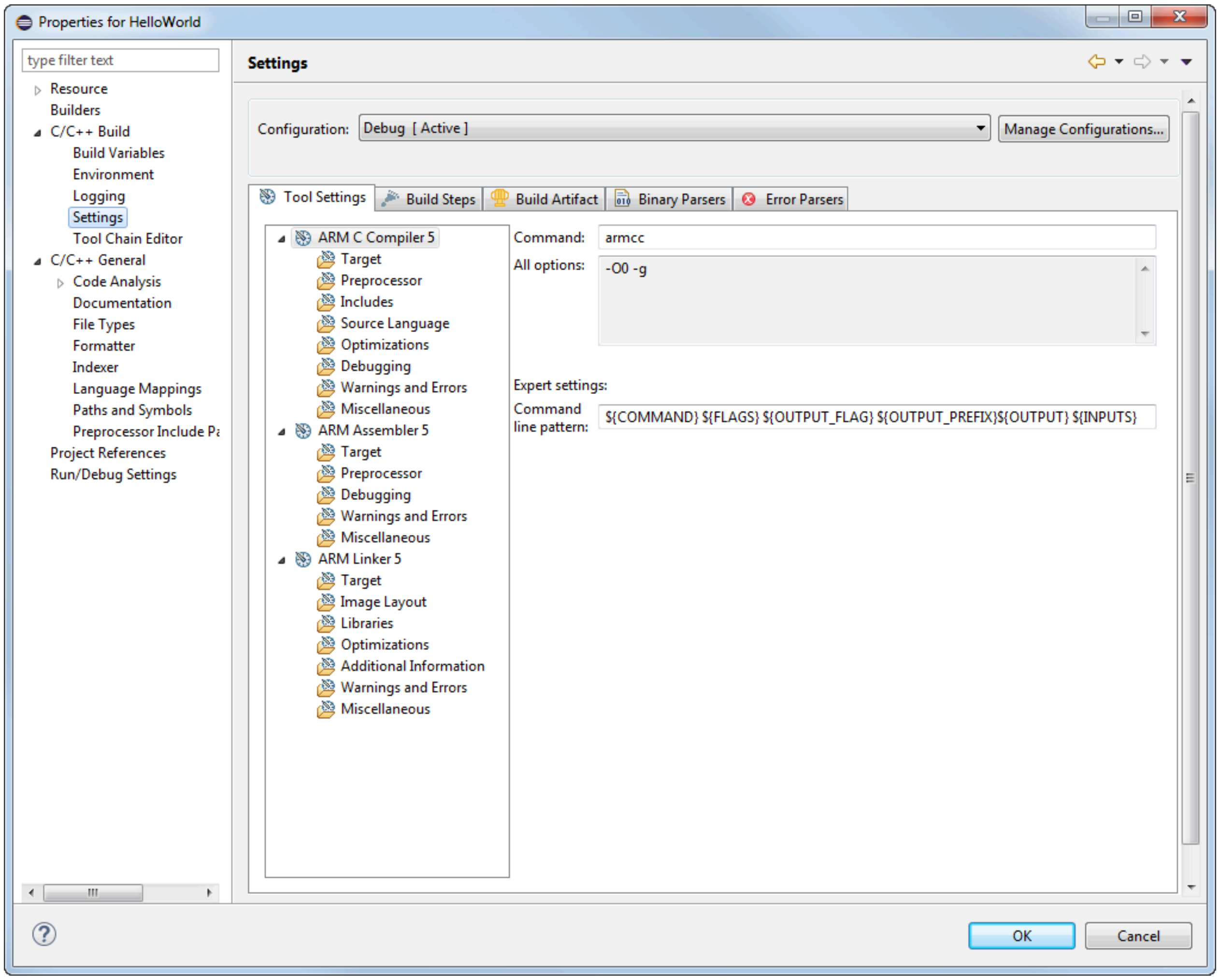
Controls the default perspectives, breakpoint, build, and launch settings before running and debugging.

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

11.64 Properties dialog box

You can customize project settings using the **Properties** dialog box.

Figure 11-89: Project properties dialog box



Configuration: Debug [Active]

Command: armcc

All options: -O0 -g

Expert settings:

Command line pattern: ${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}

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.

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.

12 File-based Flash Programming in Arm Development Studio

Describes the file-based flash programming options available in Arm® Development Studio.

12.1 About file-based flash programming in Arm Development Studio

The Arm® Development Studio configdb platform entry for a board can contain a flash definition section. This section defines one or more areas of flash, each with its own flash method and configuration parameters.

Flash methods are implemented in Jython and are typically located within the configdb. Each flash method is implemented with a specific technique of programming flash.

These techniques might involve:

- Running an external program supplied by a third party to program a file into flash.
- Copying a file to a file system mount point. For example, as implemented in the Arm Versatile Express (VE) designs.
- Download a code algorithm into the target system and to keep running that algorithm on a data set (typically a flash sector) until the entire flash device has been programmed.

Note

You can use the Arm Debugger `info flash` command to view the flash configuration for your board.

Examples of downloading a code algorithm into the target system are the Keil® flash programming algorithms which are fully supported by Arm Debugger. For the Keil flash method, one of the method configuration items is the algorithm to use to perform the flash programming. These algorithms all follow the same top level software interface and so the same Keil flash method can be used to program different types of flash. This means that Arm Debugger should be able to make direct use of any existing Keil flash algorithm.

Note

All flash methods which directly interact with the target should do so using the Arm Debugger's DTSL connection.

Flash programming supported features

The file flash programming operations support the following features:

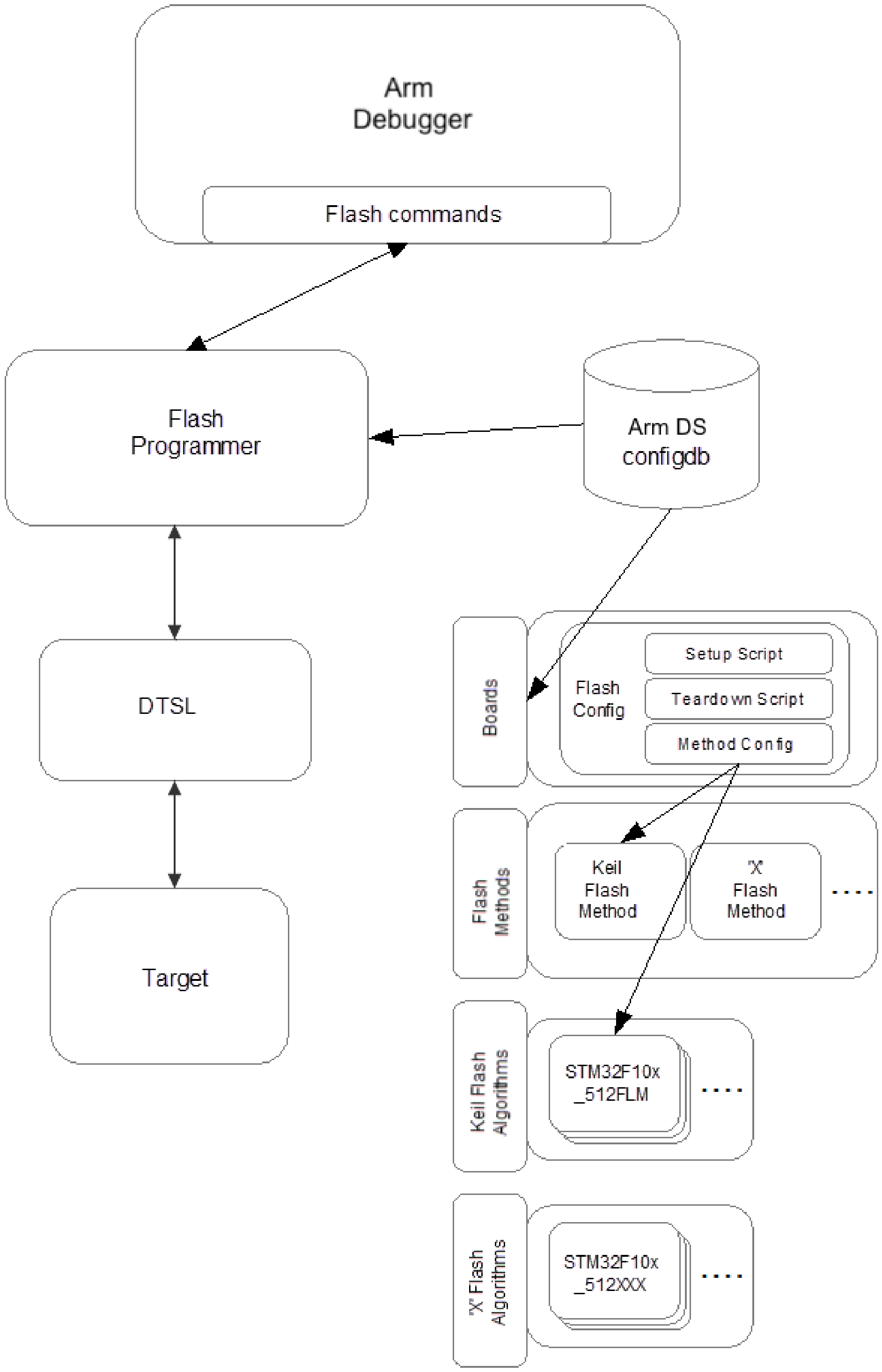
- ELF files (`.axf`) programming into flash.
- ELF files containing multiple flash areas which can each be programmed into a flash device or possible several different flash devices.
- Many and varied flash programming methods.
- All Keil flash programming algorithms.
- Target board setup and teardown to prepare it for flash programming.
- Arm Development Studio configuration database to learn about target flash devices and the options required for flash programming on a specific board or system on chip.
- Default flash options modification.
- Graphical progress reporting within the IDE and on a text only type console when used with the debugger outside the IDE, along with the ability to cancel the programming operation.
- A simple flash programming user interface where you can specify minimal configurations or options.
- Displaying warning and error messages to the user.

Note

An example, `flash_example-FVP-A9x4`, is provided with Arm Development Studio. This example shows two ways of programming flash devices using Arm Development Studio, one using a Keil Flash Method and the other using a Custom Flash Method written in Jython. For convenience, the `Cortex-A9x4 FVP` model supplied with Arm Development Studio is used as the target device. This example can be used as a template for creating new flash algorithms. The `readme.html` provided with the example contains basic information on how to use the example.

Arm Development Studio File Flash Architecture

Figure 12-1: File Flash Architecture



Related information

Flash commands

12.2 Flash programming configuration

Each target platform supported by Arm® Development Studio has an entry in the Arm Development Studio configuration database. To add support for flash programming, a target's platform entry in the database must define both the flash programming method and any required parameters.

Configuration files

The target's platform entry information is stored across two files in the configuration database:

- `project_types.xml` - This file describes the debug operations supported for the platform and may contain a reference to a flash configuration file. This is indicated by a tag such as `<flash_config>CDB://flash.xml</flash_config>`.
- The `CDB://` tag indicates a path relative to the target's platform directory which is usually the one that contains the `project_types.xml` file. You can define a relative path above the target platform directory using `../` . For example, a typical entry would be similar to `<flash_config>CDB://../../../Flash/STM32/flash.xml</flash_config>`.

  Using relative paths allows the flash configuration file to be shared between a number of targets with the same chip and same flash configuration.
- The `FDB://` tag indicates a path relative to where the Jython flash files (such as the `stm32_setup.py` and `keil_flash.py` used in the examples) are located. For Arm Development Studio installations, this is usually `<installation_directory>/sw/debugger/configdb/Flash`.
- A flash configuration `.xml` file. For example, `flash.xml`. This `.xml` file describes flash devices on a target, including which memory regions they are mapped to and what parameters need to be passed to the flash programming method.

A flash configuration must always specify the flash programming method to use, but can also optionally specify a setup script and a teardown script. Setup and teardown scripts are used to prepare the target platform for flash programming and to re-initialize it when flash programming is complete. These scripts might be very specific to the target platform, whereas the flash programming method might be generic.

Configuration file example

This example `flash.xml` is taken from the Keil® MCBSTM32E platform. It defines two flash devices even though there is only one built-in flash device in the MCBSTM32E. This is because the two flash sections, the main flash for program code and the option flash for device configuration, are viewed as separate devices when programming.

Note how the flash method is set to the `keil_flash.py` script and how the parameters for that method are subsequently defined.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--Copyright (C) 2012 ARM Limited. All rights reserved.-->
<flash_config
     xmlns:xi="http://www.w3.org/2001/XInclude"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="http://www.arm.com/flash_config"
```

```
      xsi:schemaLocation="http://www.arm.com/flash_config flash_config.xsd">
       <devices>
               <!-- STM32F1xx has 2 flash sections: main flash for program code and
 option
               flash for device configuration. These are viewed as separate devices
               when programming -->
               <!-- The main flash device -->
               <device name="MainFlash">
                       <programming_type type="FILE">
                               <!-- Use the standard method for running Keil
 algorithms -->
                               <method language="JYTHON" script="FDB://keil_flash.py"
 class="KeilFlash" method_config="Main"/>
                               <!-- Target specific script to get target in a
 suitable state for programming -->
                               <setup script="FDB://stm32_setup.py" method="setup"/>
                       </programming_type>
               </device>
               <!-- The option flash device -->
               <device name="OptionFlash">
                       <programming_type type="FILE">
                               <method language="JYTHON" script="FDB://keil_flash.py"
 class="KeilFlash" method_config="Option"/>
                               <setup script="FDB://stm32_setup.py" method="setup"/>
                       </programming_type>
               </device>
       </devices>
       <method_configs>
               <!-- Parameters for programming the main flash -->
               <method_config id="Main">
                       <params>
                               <!-- Programming algorithm binary to load to target --
>
                               <param name="algorithm" type="string" value="FDB://
algorithms/STM32F10x_512.FLM"/>
                               <!-- The core in the target to run the algorithm -->
                               <param name="coreName" type="string" value="Cortex-
M3"/>
                               <!-- RAM location & size for algorithm code and write
 buffers -->
                               <param name="ramAddress" type="integer"
 value="0x20000000"/>
                               <param name="ramSize" type="integer" value="0x10000"/>
                               <!-- Allow timeouts to be disabled -->
                               <param name="disableTimeouts" type="string"
 value="false"/>
                               <!-- Set to false to skip the verification stage -->
                               <param name="verify" type="string" value="true"/>
                       </params>
               </method_config>
               <!-- Parameters for programming the option flash -->
               <method_config id="Option">
                       <params>
                               <!-- Programming algorithm binary to load to target --
>
                               <param name="algorithm" type="string" value="FDB://
algorithms/STM32F10x_OPT.FLM"/>
                               <!-- The core in the target to run the algorithm -->
                               <param name="coreName" type="string" value="Cortex-
M3"/>
                               <!-- RAM location & size for algorithm code and write
 buffers -->
                               <param name="ramAddress" type="integer"
 value="0x20000000"/>
                               <param name="ramSize" type="integer" value="0x10000"/>
                               <!-- Allow timeouts to be disabled -->
                               <param name="disableTimeouts" type="string"
 value="false"/>
                               <!-- Set to false to skip the verification stage -->
                               <param name="verify" type="string" value="true"/>
                       </params>
```

```
            </method_config>
        </method_configs>
</flash_config>
```

Related information

Creating an extension database for flash programming on page 475

12.3 Creating an extension database for flash programming

In certain scenarios, it might not be desirable or possible to modify the default Arm® Development Studio configuration database. In this case, you can create your own configuration databases and use them to extend the default installed database.

Procedure

1. At your preferred location, create a new directory with the name of your choice for the extension database.
2. In your new directory, create two subdirectories and name them `Boards` and `Flash` respectively.
   a) In the `Boards` directory, create a subdirectory for the board manufacturer.
   b) In the board manufacturer subdirectory, create another directory for the board.
   c) In the `Flash` directory, create a subdirectory and name it `Algorithms`.
   For example, for a manufacturer `MegaSoc-Co` who makes `Acme-Board-2000`, the directory structure would look similar to this:

```
Boards
   \---> MegaSoc-Co
         \---> Acme-Board-2000
               project_types.xml
Flash
   \---> Algorithms
         Acme-Board-2000.flm
              Acme-Board-2000-Flash.py
```

3. From the main menu in Arm Development Studio, select **Window** > **Preferences** > **Arm DS** > **Configuration Database**.
   a) In the **User Configuration Databases** area, click **Add**.
   b) In the **Add configuration database location** dialog box, enter the **Name** and **Location** of your configuration database and click **OK**.
4. In the **Preferences** dialog box, click **OK** to confirm your changes.
   In the `project_types.xml` file for your platform, any reference to a `CDB://` location will resolve to the `Boards/<manufacturer>/<board>` directory and any reference to a `FDB://` location will resolve to the `Flash` directory.

12.4 About using or extending the supplied Arm Keil flash method

Arm® Debugger contains a full implementation of the Keil® flash programming method. This might be used to program any flash device supported by the Keil MDK product. It might also be used to support any future device for which a Keil flash programming algorithm can be created.

For details on creating new Keil Flash Programming algorithms (these links apply to the Keil µVision® product), see:

Algorithm Functions

Creating New Algorithms

To help with the creation of new Keil flash programming algorithms in Arm Development Studio, Arm Debugger contains a full platform flash example for the Keil MCBSTM32E board. This can be used as a template for new flash support.

Note

An example, `flash_example-FVP-A9x4`, is provided with Arm Development Studio. This example shows two ways of programming flash devices using Arm Development Studio, one using a Keil Flash Method and the other using a Custom Flash Method written in Jython. For convenience, the `Cortex-A9x4 FVP` model supplied with Arm Development Studio is used as the target device. This example can be used as a template for creating new flash algorithms. The `readme.html` provided with the example contains basic information on how to use the example.

This section describes how to add flash support to an existing platform using an existing Keil flash program, and how to add flash support to an existing platform using a new Keil flash algorithm.

12.4.1 Adding flash support to an existing platform using an existing Keil flash algorithm

To use the Keil® MDK flash algorithms within Arm® Development Studio, the algorithm binary needs to be imported into the target configuration database and the flash configuration files created to reference the `keil_flash.py` script.

About this task

This example uses the flash configuration for the Keil MCBSTM32E board example in Flash programming configuration as a template to add support to a board called the Acme-Board-2000 made by MegaSoc-Co.

Procedure

1. Copy the algorithm binary `.FLM` into your configuration database `Flash/Algorithms` directory.
2. Copy the flash configuration file from `Boards/Keil/MCBSTM32E/keil-mcbstm32e_flash.xml` to `Boards/MegaSoc-Co/Acme-Board-2000/flash.xml`.

3. Edit the platform's `project_types.xml` to reference the `flash.xml` file by inserting `<flash_config>CDB://flash.xml</flash_config>` below `platform_data` entry, for example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--Copyright (C) 2009-2012 ARM Limited. All rights reserved.-->
<platform_data xmlns="http://www.arm.com/project_type"
               xmlns:peripheral="http://com.arm.targetconfigurationeditor"
               xmlns:xi="http://www.w3.org/2001/XInclude"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"type="HARDWARE"
               xsi:schemaLocation="http://www.arm.com/project_type../../../
Schemas/platform_data-1.xsd">
               <flash_config>CDB://flash.xml</flash_config>
```

4. Edit the devices section, and create a `<device>` block for each flash device on the target.

Note

The `method_config` attribute should refer to a unique `<method_config>` block for that device in the `<method_configs>` section.

5. Optionally, create and then reference any setup or teardown script required for your board. If your board does not need these, then do not add these lines to your configuration.

```
<setup script="FDB://Acme-Board-2000-Flash.py" method="setup"/>
<teardown script="FDB://Acme-Board-2000-Flash.py" method="teardown"/>
```

6. Edit the `method_configs` section, creating a `<method_config>` block for each device.

Note

- The value for the algorithm parameter should be changed to the path to the algorithm copied in *Step 1*. The `FDB://` prefix is used to indicate the file can be found in the configuration database `Flash` directory.
- The `coreName` parameter must be the name of the core on the target that runs the algorithm. This must be the same name as used in the `<core>` definition within `project_types.xml`. For example, `<core connection_id ="Cortex-M3" core_definition ="Cortex-M3"/>`.
- The `ramAddress` and `ramSize` parameters should be set to an area of RAM that the algorithm can be downloaded in to and used as working RAM. It should be big enough to hold the algorithm, stack plus scratch areas required to run the algorithm, and a sufficiently big area to download image data. The other parameters do not normally need to be changed.

12.4.2 Adding flash support to an existing target platform using a new Keil flash algorithm

Arm® Development Studio ships with a complete Keil® flash algorithm example for the STM32 device family. You can use this as a template for creating and building your new flash algorithm.

Locate the `Bare-metal_examples_Armv7.zip` file within the `<installation_directory>/examples` directory. Extract it to your file system and then import the `examples/flash_algo-STM32F10x` project into your Arm Development Studio workspace.

Using this as your template, create a new project, copy the content from the example into your new project, and modify as needed.

When you have successfully built your `.FLM` file(s), follow the instructions in the *Adding flash support to an existing platform using an existing Keil flash algorithm* topic.

Related information

Adding flash support to an existing platform using an existing Keil flash algorithm on page 476

12.5 About creating a new flash method

If the Keil® flash method is inappropriate for your requirements, you need to create a new custom flash method for your use.

Programming methods are implemented in Jython (Python, utilizing the Jython runtime). The use of Jython allows access to the DTSL APIs used by Arm® Debugger. Arm Development Studio includes the PyDev tools to assist in writing Python scripts.

In an Arm Development Studio install, the `configdb\Flash\flashprogrammer` directory holds a number of Python files which contain utility methods used in the examples.

This section describes a default implementation of `com.arm.debug.flashprogrammer.FlashMethodv1` and creating a flash method using a Python script.

12.5.1 About using the default implementation FlashMethodv1

Flash programming methods are written as Python classes that are required to implement the `com.arm.debug.flashprogrammer.IFlashMethod` interface. This interface defines the methods the flash programming layer of Arm® Debugger might invoke.

See the `flash_method_v1.py` file in the `<installation_directory>\sw\debugger\configdb\Flash\flashprogrammer` for a default implementation of `com.arm.debug.flashprogrammer.FlashMethodv1`. This has empty implementations of all functions - this allows a Python class derived from this object to only implement the required functions.

Running a flash programming method is split into three phases:

1. Setup - the `setup()` function prepares the target for performing flash programming. This might involve:
   - Reading and validating parameters passed from the configuration file.
   - Opening a connection to the target.
   - Preparing the target state, for example, to initialize the flash controller.
   - Loading any flash programming algorithms to the target.

2. Programming - the `program()` function is called for each section of data to be written. Images might have multiple load regions, so the `program()` function might be called several times. The data to write is passed to this function and the method writes the data into flash at this stage.
3. Teardown - the `teardown()` function is called after all sections have been programmed. At this stage, the target state can be restored (for example, take the flash controller out of write mode or reset the target) and any debug connection closed.

Note

The `setup()` and `teardown()` functions are not to be confused with the target platform optional `setup()` and `teardown()` scripts. The `setup()` and `teardown()` functions defined in the flash method class are for the method itself and not the board.

12.5.2 About creating the flash method Python script

For the purposes of this example the Python script is called `example_flash.py`.

Start by importing the objects required in the script:

```
from flashprogrammer.flash_method_v1 import FlashMethodv1
from com.arm.debug.flashprogrammer import TargetStatus
```

Then, define the class implementing the method:

```
class ExampleFlashWriter(FlashMethodv1):
   def __init__(self, methodServices):
       FlashMethodv1.__init__(self, methodServices)
   def setup(self):
       # perform any setup for the method here
       pass
   def teardown(self):
       # perform any clean up for the method here
       # return the target status
       return TargetStatus.STATE_RETAINED
   def program(self, regionID, offset, data):
       # program a block of data to the flash
       # regionID indicates the region within the device (as defined in the flash
 configuration file)
       # offset is the byte offset within the region
       # perform programming here
       # return the target status
       return TargetStatus.STATE_RETAINED
```

Note

- The `__init__` function is the constructor and is called when the class instance is created.
- `methodServices` allows the method to make calls into the flash programmer - it must not be accessed directly.
- `FlashMethodv1` provides functions that the method can call while programming.
- The `program()` and `teardown()` methods must return a value that describes the state the target has been left in.

This can be one of:

- `STATE_RETAINED` - The target state has not been altered from the state when programming started. In this state, the register and memory contents have been preserved or restored.
- `STATE_LOST` - Register and memory contents have been altered, but a system reset is not required.
- `RESET_REQUIRED` - It is recommended or required that the target be reset.
- `POWER_CYCLE_REQUIRED` - It is required that the target be manually power cycled. For example, when a debugger-driven reset is not possible or not sufficient to reinitialize the target.

Creating the target platform setup and teardown scripts

If the hardware platform requires some setup (operations to be performed before flash programming) and/or teardown (operations performed after flash programming) functionality, you must create one or more scripts which contain `setup()` and `teardown()` functions. These can be in separate script files or you can combine them into a single file. This file must be placed into the configdb `Flash` directory so that it can be referenced using a `FDB://` prefix in the flash configuration file.

For example, the contents of a single file which contains both the `setup()` and `teardown()` functions would be similar to:

```
from com.arm.debug.flashprogrammer.IFlashClient import MessageLevel
from flashprogrammer.device import ensureDeviceOpen
from flashprogrammer.execution import ensureDeviceStopped
from flashprogrammer.device_memory import writeToTarget
def setup(client, services):
    # get a connection to the core
    conn = services.getConnection()
    dev = conn.getDeviceInterfaces().get("Cortex-M3")
    ensureDeviceOpen(dev)
    ensureDeviceStopped(dev)
    # Perform some target writes to enable flash programming
    writeToTarget(dev, FLASH_EN, intToBytes(0x81))
def teardown(client, services):
    # get a connection to the core
    conn = services.getConnection()
    dev = conn.getDeviceInterfaces().get("Cortex-M3")
    ensureDeviceOpen(dev)
    ensureDeviceStopped(dev)
    # Perform some target writes to disable flash programming
    writeToTarget(dev, FLASH_EN, intToBytes(0))
```

Creating the flash configuration file

To use the method to program flash, a configuration file must be created that describes the flash device, the method to use and any parameters or other information required. This is an `.xml` file and is typically stored in the same directory as the target's other configuration files (`Boards/<Manufacturer>/<Board name>`) as it contains target-specific information.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flash_config
```

```
    xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.arm.com/flash_config"
    xsi:schemaLocation="http://www.arm.com/flash_config flash_config.xsd">
    <devices>
        <device name="Example">
            <regions>
                <region address="0x8000" size="0x10000">
            </regions>
            <programming_type type="FILE">
                <method language="JYTHON" script="FDB://example_flash.py"
  class="ExampleFlashWriter" method_config="Default"/>
                <setup script="FDB://file_target.py" method="setup"/>
                <teardown script="FDB://file_target.py" method="teardown"/>
            </programming_type>
        </device>
    </devices>
    <method_configs>
        <method_config id="Default">
            <params>
                <!-- Use last 2K of RAM -->
                <param name="ramAddress" type="integer" value="0x00100000"/>
                <param name="ramSize" type="integer" value="0x800"/>
            </params>
        </method_config>
    </method_configs>
</flash_config>
```

- The `flash_config` tag defines used XML spaces and schema. This does not usually need to be changed. Under the `flash_config` tag, a devices tag is required. This contains a number of device tags, each representing one flash device on the target. The device tag defines the name of the device - this is the name reported by the `info flash` command and is used only when programming to a specific device. It also defines a number of regions where the flash device appears in the target's memory - the addresses of each region are matched against the address of each load region of the image being programmed.
- The `programming_type` tag defines the programming method and setup/teardown scripts to be used for a flash programming operation. Currently, only `FILE` is supported.
- The `method` tag defines the script which implements the programming method. Currently, only `JYTHON` is supported for the language attribute. The script and class attributes define which script file to load and the name of the class that implements the programming method within the script. The `method_config` attributes define which set of parameters are used by the device. This allows multiple devices to share a set of parameters.
- The `programming_type` may also have optional setup and teardown tags. These define a script and a method within that script to call before or after flash programming.
- Within the `method_configs` tag, the parameters for each device are contained within `method_config` tags.
- Parameters must have a unique name and a default value. You can override the value passed to the method. See the help for the `flash load` command in Arm® Debugger.
- Where the configuration file references another file, for example, the script files, the `FDB://` prefix indicates that the file is located in the `Flash` subdirectory of the configuration database. If there are multiple databases, then the `Flash` subdirectory of each database is searched until the file is found.
- The last file that needs to be changed is the `project_types.xml` file in the target's directory to tell Arm Development Studio that the flash configuration can be found in the file

created above. The following line must be added under the top-level `platform_data` tag: `<flash_config>CDB://flash.xml</flash_config>` The `CDB://` prefix tells Arm Development Studio that the `flash.xml` file is located in the same directory as the `project_types.xml` file.

12.6 About testing the flash configuration

With the files described in the previous sections in place, it should be possible to make a connection to the target in Arm® Development Studio and inspect the flash devices available and program an image. Although, with the files in their current form, no data will actually be written to flash.

Note

If Arm Development Studio is already open and `project_types.xml` is changed, it will be necessary to rebuild the configuration database.

Within Arm Debugger, connect to your target system and enter `info flash` into the **Commands** view. You should get an output similar to:

```
info flash
MainFlash
regions:    0x8000000-0x807FFFF
parameters: programPageTimeout: 100
            driverVersion: 257
            programPageSize: 0x400
            eraseSectorTimeout: 500
            sectorSizes: ((0x800, 0x00000000))
            valEmpty: 0xff
            type: 1
            size: 0x00080000
            name: STM32F10x High-density Flash
            address: 0x08000000
            algorithm: FDB://algorithms/STM32F10x_512.FLM
            coreName: Cortex-M3
            ramAddress: 0x20000000
            ramSize: 0x10000
            disableTimeouts: false
            verify: true
```

You can test the flash programming operation by attempting to program with a test ELF file.

```
flash load flashyprogram.axf
Writing segment 0x00008000 ~ 0x0000810C (size 0x10C)
Flash programming completed OK (target state has been preserved)
```

Note

You can use any ELF (`.axf`) file which contains data within the configured address range.

12.7 About flash method parameters

Programming methods can take parameters that serve to change the behavior of the flash programming operation.

Example parameters could be:

- The programming algorithm image to load, for example, the Keil® Flash Algorithm file.
- The location and size of RAM the method can use for running code, buffers, and similar items.
- Clock speeds.
- Timeouts.
- Programming and erase page sizes.

The default values of the parameters are taken from the flash configuration file.

Note

You can override the parameters from the Arm® Development Studio command line.

The programming method can obtain the value of the parameters with:

- `getParameter(name)` returns the value of a parameter as a string. The method can convert this to another type, such as integers, as required. `None` is returned if no value is set for this parameter.
- `getParameters()` returns a map of all parameters to values. Values can then be obtained with the `[]` operator.

For example:

```
def setup(self):
   # get the name of the core to connect to
   coreName = self.getParameter("coreName")
   # get parameters for working RAM
   self.ramAddr = int(self.getParameter("ramAddress"), 0)
   self.ramSize = int(self.getParameter("ramSize"), 0)
```

12.8 About getting data to the flash algorithm

Data is passed to the `program()` function by the data parameter.

A data parameter is an object that provides the following functions:

- `getSize()` returns the amount of data available in bytes.
- `getData(sz)` returns a buffer of up to `sz` data bytes. This may be less, for example, at the end of the data. The read position is advanced.

- `seek(pos)` move the read position.
- `getUnderlyingFile()` gets the file containing the data. (None, if not backed by a file). This allows the method to pass the file to an external tool.

The method can process the data with:

```
def program(self, regionID, offset, data):
   data.seek(0)
   bytesWritten = 0
   while bytesWritten < data.getSize():
     # get next block of data
     buf = data.getData(self.pageSize)
     # write buf to flash
     bytesWritten += len(buf)
```

12.9 About interacting with the target

To perform flash programming, the programming method might need to access the target.

The flash programmer provides access to the DTSL APIs for this and the programming method can then get a connection with the `getConnection()` function of class `FlashMethodv1`.

This is called from the `setup()` function of the programming method. If there is already an open connection, for example, from the Arm® Debugger, this will be re-used.

```
def setup(self):
   # connect to core
   self.conn = self.getConnection()
```

Note

An example, `flash_example-FVP-A9x4`, is provided with Arm Development Studio. This example shows two ways of programming flash devices using Arm Development Studio, one using a Keil® Flash Method and the other using a Custom Flash Method written in Jython. For convenience, the `Cortex-A9x4 FVP` model supplied with Arm Development Studio is used as the target device. This example can be used as a template for creating new flash algorithms. The `readme.html` provided with the example contains basic information on how to use the example.

Accessing the core

When interacting with the target, it might be necessary to open a connection to the core. If the debugger already has an open connection, a new connection might not be always possible. A utility function, `ensureDeviceOpen()`, is provided that will open the connection only if required. It will return `true` if the connection is open and so should be closed after programming in the `teardown()` function.

To access the core's registers and memory, the core has to be stopped. Use the `ensureDeviceStopped()` function to assist with this.

```
def setup(self):
   # connect to core & stop
   self.conn = self.getConnection()
   coreName = self.getParameter("coreName")
   self.dev = self.conn.getDeviceInterfaces().get(coreName)
   self.deviceOpened = ensureDeviceOpen(self.dev)
   ensureDeviceStopped(self.dev)
def teardown(self):
   if self.deviceOpened:
       # close device connection if opened by this script
       self.dev.closeConn()
```

Reading/writing memory

The core's memory can be accessed using the `memWrite()`, `memFill()`, and `memRead()` functions of the dev object (`IDevice`).

```
from com.arm.rddi import RDDI
from com.arm.rddi import RDDI_ACC_SIZE
from jarray import zeros
...
   def program(self):
       ...
       self.dev.memFill(0, addr, RDDI_ACC_SIZE.RDDI_ACC_WORD,
                        RDDI.RDDI_MRUL_NORMAL, False, words, 0)
       self.dev.memWrite(0, addr, RDDI_ACC_SIZE.RDDI_ACC_WORD,
                         RDDI.RDDI_MRUL_NORMAL, False, len(buf), buf)
       ...
   def verify(self):
       ...
       readBuf = zeros(len(buf), 'b')
       self.dev.memRead(0, addr, RDDI_ACC_SIZE.RDDI_ACC_WORD,
                        RDDI.RDDI_MRUL_NORMAL, len(readBuf), readBuf)
       ...
```

Utility routines to make the method code clearer are provided in `device_memory`:

```
from flashprogrammer.device_memory import writeToTarget, readFromTarget
...
   def program(self):
       ...
       writeToTarget(self.dev, address, buf)
       ...
   def verify(self):
       ...
       readBuf = readFromTarget(self.dev, addr, count)
       ...
```

Reading and writing registers

The core's registers can be read using the `regReadList()` and written using the `regWriteList()` functions of `Idevice`.

Note

You must be careful to only pass integer values and not long values.

These registers are accessed by using numeric IDs. These IDs are target specific. For example, `R0` is register 1 on a Cortex®-A device, but register 0 on a Cortex-M device.

`execution.py` provides functions that map register names to numbers and allow reading or writing by name.

- `writeRegs(device, regs)` writes a number of registers to a device. `regs` is a list of (name, value) pairs.

  For example:

  ```
  writeRegs (self.dev, [ ("R0", 0), ("R1", 1234), ("PC", 0x8000) ]
  ```

  sets `R0`, `R1`, and `PC` (R15).

- `readReg(device, reg)` reads a named register.

  For example:

  ```
  value = readReg ("R0")
  ```

  reads R0 and returns its value.

Running code on the core

The core can be started and stopped via the `go()` and `stop()` functions. Breakpoints can be set with the `setSWBreak()` or `setHWBreak()` functions and cleared with the `clearSWBreak()` or `clearHWBreak()` functions. As it may take some time to reach the breakpoint, before accessing the target further, the script should wait for the breakpoint to be hit and the core stopped.

`execution.py` provides utility methods to assist with running code on the target.

To request the core to stop and wait for the stop status event to be received, and raise an error if no event is received before timeout elapses `stopDevice(device, timeout=1.0)`

- To check the device's status and calls `stopDevice()` if it is not stopped.
  `ensureDeviceStopped(device, timeout=1.0):`
- To start the core and wait for it to stop, forces the core to stop and raise an error if it doesn't stop before timeout elapses. The caller must set the registers appropriately and have set a breakpoint or vector catch to cause the core to stop at the desired address.
  `runAndWaitForStop(device, timeout=1.0):`
- To set a software breakpoint at `addr`, start the core and wait for it to stop by calling `runAndWaitForStop()`. The caller must set the registers appropriately.
  `runToBreakpoint(device, addr, bpFlags = RDDI.RDDI_BRUL_STD, timeout=1.0):`

Flash programming algorithms are often implemented as functions that are run on the target itself. These functions may take parameters where the parameters are passed through registers.

`funcCall()` allows methods to call functions that follow AAPCS (with some restrictions):

- Up to the first four parameters are passed in registers `R0-R3`.
- Any parameters above this are passed via the stack.
- Only integers up to 32-bit or pointer parameters are supported. Floating point or 64-bit integers are not supported.
- The result is returned in `R0`.

We can use the above to simulate flash programming by writing the data to RAM. See `example_method_1.py`. This:

- Connects to the target on `setup()`.
- Fills the destination RAM with 0s to simulate erase.
- Writes data to a write buffer in working RAM.
- Runs a routine that copies the data from the write buffer to the destination RAM.
- Verifies the write by reading from the destination RAM.

Loading programming algorithm images onto the target

Programming algorithms are often compiled into `.elf` images.

`FlashMethodv1.locateFile()` locates a file for example, from a parameter, resolving any `FDB://` prefix to absolute paths.

`symfile.py` provides a class, `SymbolFileReader`, that allows the programming method to load an image file and get the locations of symbols. For example, to get the location of a function:

```
# load the algorithm image
algorithmFile = self.locateFile(self.getParameter('algorithm'))
algoReader = SymbolFileReader(algorithmFile)
# Find the address of the Program() function
funcInfo = algoReader.getFunctionInfo()['Program']
programAddr = funcInfo['address']
if funcInfo['thumb']:
   # set bit 0 if symbol is thumb
   programAddr |= 1
```

`image_loader.py` provides routines to load the image to the target:

```
# load algorithm into working RAM
algoAddr = self.ramAddr + 0x1000 # allow space for stack, buffers etc
loadAllCodeSegmentsToTarget(self.dev, algoReader, algoAddr)
```

If the algorithm binary was linked as position independent, the addresses of the symbols are relative to the load address and this offset should be applied when running the code on the target:

```
programAddr += algoAddr
```

```
args = [ writeBuffer, destAddr, pageSize ]
funcCall(self.dev, programAddr, args, self.stackTop)
```

Progress reporting

Flash programming can be a slow process, so it is desirable to have progress reporting features. The method can do this by calling `operationStarted()`. This returns an object with functions:

- `progress()` - update the reported progress.
- `complete()` - report the operation as completed, with a success or failure.

Progress reporting can be added to the `program()` function in the previous example:

```
def program(self, regionID, offset, data):
   # calculate the address to write to
   region = self.getRegion(regionID)
   addr = region.getAddress() + offset
   # Report progress, assuming erase takes 20% of the time, program 50%
   # and verify 30%
   progress = self.operationStarted(
      'Programming 0x%x bytes to 0x%08x' % (data.getSize(), addr),
      100)
   self.doErase(addr, data.getSize())
   progress.progress('Erasing completed', 20)
   self.doWrite(addr, data)
   progress.progress('Writing completed', 20+50)
   self.doVerify(addr, data)
   progress.progress('Verifying completed', 20+50+30)
   progress.completed(OperationResult.SUCCESS, 'All done')
   # register values have been changed
   return TargetStatus.STATE_LOST
```

The above example only has coarse progress reporting, only reporting at the end of each phase. Better resolution can be achieved by allowing each sub-task to have a progress monitor. `subOperation()` creates a child progress monitor.

Care should be taken to ensure `completed()` is called on the progress monitor when an error occurs. Arm recommends you place a `try: except:` block around the code after a progress monitor is created.

```
import java.lang.Exception
def program(self, regionID, offset, data):
      progress = self.operationStarted(
         'Programming 0x%x bytes to 0x%08x' % (data.getSize(), addr),
         100)
      try:
         # Do programming
      except (Exception, java.lang.Exception), e:
         # exceptions may be derived from Java Exception or Python Exception
         # report failure to progress monitor & rethrow
         progress.completed(OperationResult.FAILURE, 'Failed')
         raise
```

Note

`import java.lang.Exception` - If you omit import and a Java exception is thrown, you may get a confusing error report from Jython indicating that it cannot find the

Java namespace. Further, the python line location indicated as the source of the error will not be accurate.

Cancellation

If you wish to abort a long-running flash operation, programming methods can call `isCancelled()` to check if the operation is canceled. If this returns true, the method stops programming.

Note

The `teardown()` functions are still called.

Messages

The programming method can report messages by calling the following:

- `warning()` - reports a warning message.
- `info()` - reports an informational message.
- `debug()` - reports a debug message - not normally displayed.

Locating and resolving files

`FlashMethodv1.locateFile()` locates a file for example, from a parameter, resolving any `FDB://` prefix to absolute paths.

This searches paths of all flash subdirectories of every configuration database configured in Arm Development Studio.

For example:

`<installation_directory>/sw/debugger/configdb/Flash/`

`c:\MyDB\Flash`

Error handling

Exceptions are thrown when errors occur. Errors from the API calls made by the programming method will be `com.arm.debug.flashprogrammer.FlashProgrammerException` (or derived from this). Methods may also report errors using Python's `raise` keyword. For example, if verification fails:

```
# compare contents
res = compareBuffers(buf, readBuf)
if res != len(buf):
   raise FlashProgrammerRuntimeException, "Verify failed at address: %08x" %
(addr + res)
```

If a programming method needs to ensure that a cleanup occurs when an exception is thrown, the following code forms a template:

```
import java.lang.Exception
```

```
    ...
    try:
       # Do programming
    except (Exception, java.lang.Exception), e:
       # exceptions may be derived from Java Exception or Python Exception
       # report failure to progress monitor and rethrow
       # Handle errors here
       # Rethrow original exception
       raise
    finally:
       # This is always executed on success or failure
       # Close resources here
```

See the Progress handler section for example usage.

Note

`import java.lang.Exception` - If you omit import and a Java exception is thrown, you may get a confusing error report from Jython indicating that it cannot find the Java namespace. Further, the python line location indicated as the source of the error will not be accurate.

Running an external tool

Some targets may already have a standalone flash programming tool. It is possible to create a Arm Debugger programming method to call this tool, passing it to the path of the image to load. The following example shows how to do this, using the `fromelf` tool in place of a real flash programming tool.

```
from flashprogrammer.flash_method_v1 import FlashMethodv1
from com.arm.debug.flashprogrammer.IProgress import OperationResult
from com.arm.debug.flashprogrammer import TargetStatus
import java.lang.Exception
import subprocess
class RunProgrammer(FlashMethodv1):
   def __init__(self, methodServices):
       FlashMethodv1.__init__(self, methodServices)
   def program(self, regionID, offset, data):
       progress = self.operationStarted(
           'Programming 0x%x bytes with command %s' % (data.getSize(), '
 '.join(cmd)),
           100)
       try:
           # Get the path of the image file
           file = data.getUnderlyingFile().getCanonicalPath()
           cmd = [ 'fromelf', file ]
           self.info("Running %s" % ' '.join(cmd))
    # run command
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE)
    out, err = proc.communicate()
    # pass command output to user as info message
    self.info(out)
           progress.progress('Completed', 100)
           progress.completed(OperationResult.SUCCESS, 'All done')
       except (Exception, java.lang.Exception), e:
           # exceptions may be derived from Java Exception or Python Exception
           # report failure to progress monitor & rethrow
           progress.completed(OperationResult.FAILURE, 'Failed')
           raise
       return TargetStatus.STATE_RETAINED
```

`os.environ` can be used to lookup environment variables, for example, the location of a target's toolchain:

```
programmerTool = os.path.join(os.environ['TOOLCHAIN_INSTALL'], 'flashprogrammer')
```

Setup and teardown

The flash configuration file can specify scripts to be run before and after flash programming. These are termed setup and teardown scripts and are defined using setup and teardown tags. The setup script should put the target into a state ready for flash programming.

This might involve one or more of:

- Reset the target.
- Disable interrupts.
- Disable peripherals that might interfere with flash programming.
- Setup DRAM.
- Enable flash control.
- Setup clocks appropriately.

The teardown script should return the target to a usable state following flash programming.

In both cases, it may be necessary to reset the target. The following code can be used to stop the core on the reset vector.

Note

This example code assumes that the core supports the RSET vector catch feature.

```
def setup(client, services):
   # get a connection to the core
   conn = services.getConnection()
   dev = conn.getDeviceInterfaces().get("Cortex-M3")
   ensureDeviceOpen(dev)
   ensureDeviceStopped(dev)
   dev.setProcBreak("RSET")
   dev.systemReset(0)
   # TODO: wait for stop!
   dev.clearProcBreak("RSET")
```

Other ways of providing flash method parameters

The flash configuration file can provide flash region information and flash parameter information encoded into the XML. However, for some methods, this information may need to be extracted from the flash algorithm itself.

Programming methods can extend any information in the flash configuration file (if any) with address regions and parameters for the method by overriding a pair of class methods - `getDefaultRegions()` and `getDefaultParameters()`.

```
getDefaultParameters().
from com.arm.debug.flashprogrammer import FlashRegion
...
class ProgrammingMethod(FlashMethodv1):
...
   def getDefaultRegions(self):
       return [ FlashRegion(0x00100000, 0x10000), FlashRegion(0x00200000, 0x10000) ]
   def getDefaultParameters(self):
       params = {}
       params['param1'] = "DefaultValue1"
       params['param2'] = "DefaultValue2"
       return params
```

The above code defines two 64kB regions at `0x00100000` and `0x00200000` . Regions supplied by this method are only used if no regions are specified for the device in the configuration file. The above code defines 2 extra parameters. These parameters are added to the parameters in the flash configuration. If a parameter is defined in both, the default value in the flash configuration file is used. This region and parameter information can be extracted from the algorithm binary itself (rather than being hard-coded as in the above example). The Keil algorithm images contain a data structure defining regions covered by the device and the programming parameters for the device. The Keil programming method loads the algorithm binary (specified by a parameter in the configuration file) and extracts this information to return in these calls.

12.10 Flash programming CMSIS pack-based projects

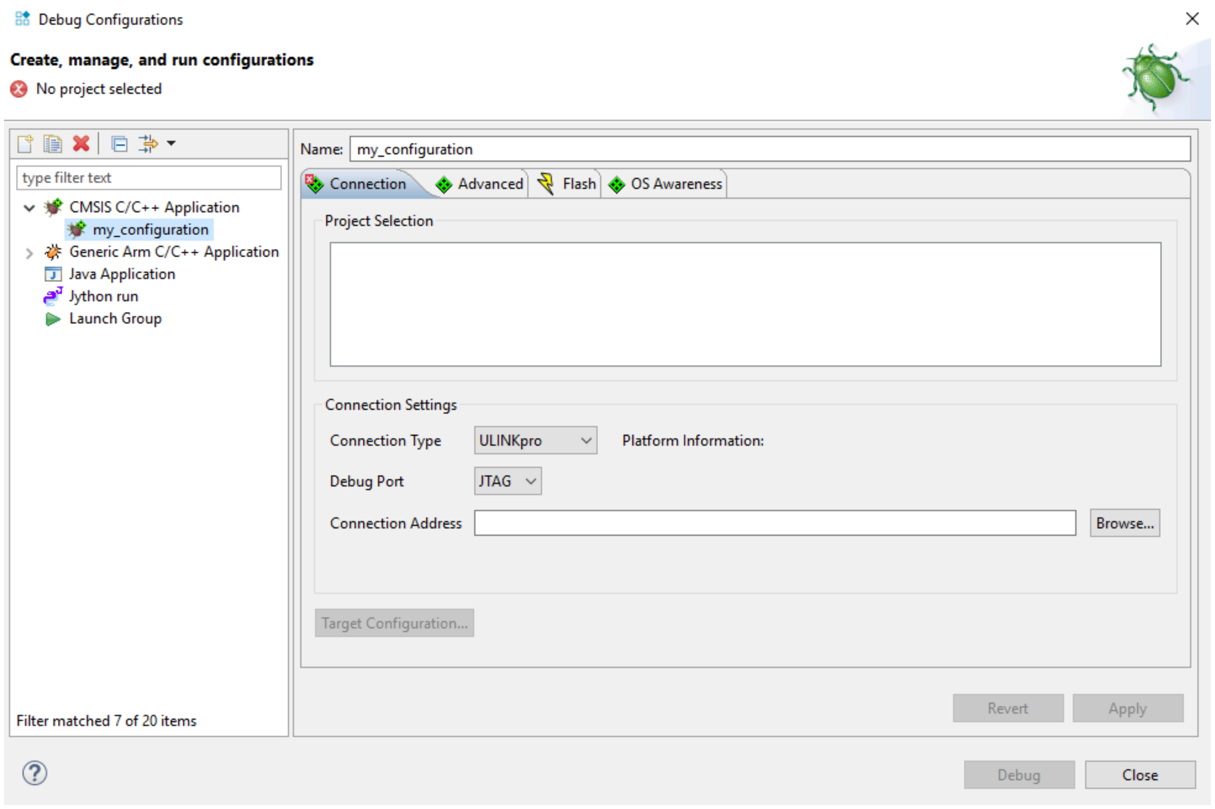
Arm® Development Studio supports flash programming for CMSIS pack-based projects, including the ability to flash both single and multiple images onto your target.

Before you begin

You require:

- A target with writable flash regions.
- A CMSIS pack for your target.
- One or more images that you want to load on to your target.
- An existing CMSIS C/C++ Application configuration. The **Debug Configurations** dialog box for a CMSIS C/C++ Application configuration contains four tabs:
    - **Connection** - Specify the probe type and connection address.
    - **Advanced** - Specify the image(s) to flash to the target, including the reset options.
    - **Flash** - Specify the flash algorithms and download options.
    - **OS Awareness** - target-dependent. Arm Debugger has debug symbols that are loaded for certain OSs. See About OS awareness for details.

Figure 12-2: CMSIS configuration dialog box



For this task, we are going to use the **Advanced** and **Flash** tabs.

About this task

When you create a CMSIS pack-based project, you can modify the debug configuration to:

- Specify one or more images to flash to the device.
- When flashing multiple images, specify which image provides the entry point.
- Specify how the target is reset during connection.
- Configure the flash algorithm.

Flashing single and multiple images follow the same process. This topic describes how to flash multiple images using the IDE. You can also use the command-line to flash images using Arm Debugger commands. To flash single images use `flash load`, and to flash multiple images use `flash load-multiple`.

Procedure

1. Load your CMSIS pack-based project into Arm Development Studio, either by importing an existing project , or by creating a new project.
2. Open the **Debug Configurations** dialog box.
3. Specify the images to flash to the target:

a) Select the **Advanced** tab
b) Add a new image by clicking the green + button. The **Add new image** dialog box opens, where you can select an `.axf` image from your filesystem or from your workspace.

Note

Any image found within the CMSIS C/C++ application project is automatically added to the table when you first open the dialog box.

---

c) When the image is added to the table, use the checkboxes to:

- Select **Download** to tell Arm Debugger to load the image on to the target. If you do not select this option, only the debug symbols are loaded.

Note

Arm recommends that you select **Download** for all or none of your images. If some images are set to download and some are not, previously loaded image sections might get overwritten or cleared when you flash the target. If there is a danger of this happening, a warning message is shown at the top of the dialog box.

- Select the primary image. The primary image tells the debugger which of the images in the table contains the entry point. This is used if you have selected **Debug from entry point**.

Note

You must only have one primary image.

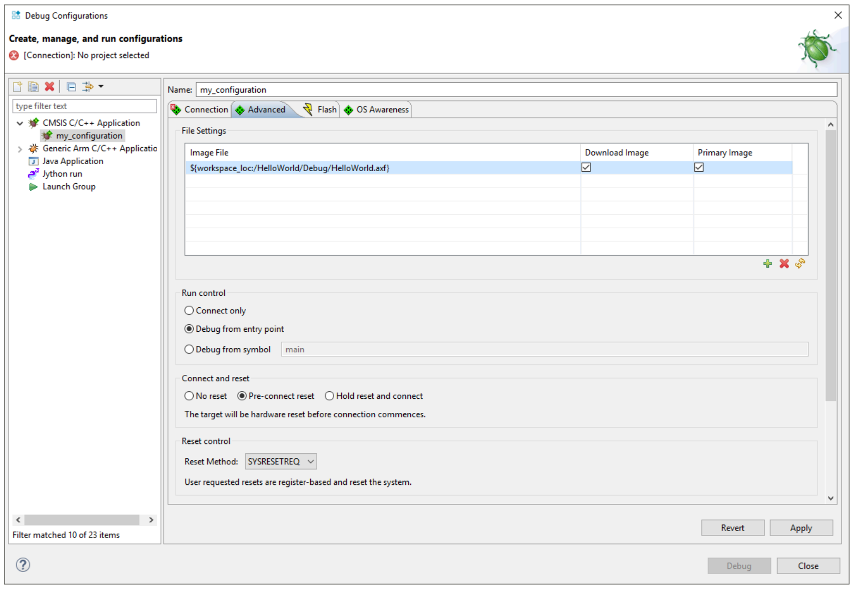
---

d) Remove an image by selecting it and clicking the red x button.
e) Refresh the list of images by clicking the yellow arrow button. You need to do this if you have renamed or deleted any of the images, or if you have rebuilt any of the images since adding them to the list.

Note

- When you flash the target, if some images are set to download and some are not, previously loaded image sections might get cleared or overwritten.
- Images are validated to check if they exist. If the debugger cannot find the image, it is highlighted red in the table.
- The target is reset before flashing the image(s). The **Connect and reset** option specifies which type of reset occurs when it reconnects.

Figure 12-3: Screenshot of the CMSIS configuration Advanced tab dialog box



Debug Configurations

Create, manage, and run configurations

[Connection]: No project selected

Name: my_configuration

File Settings

Image File	Download Image	Primary Image
${workspace_loc:/HelloWorld/Debug/HelloWorld.axf}	✓	✓

Run control

- Connect only
- [✓ Debug from entry point]
- Debug from symbol: main

Connect and reset

No reset [✓ Pre-connect reset] Hold reset and connect

The target will be hardware reset before connection commences.

Reset control

Reset Method: SYSRESETREQ

User requested resets are register-based and reset the system.

4. Specify the flash algorithms:
   a) Select the **Flash** tab.
      The **Flash** tab shows the flash algorithms that are used by Arm Debugger on connection when flashing the target. Flash algorithms describe the flash devices on the target, and how to download or erase applications in the flash sectors. Each algorithm has:
      - A region start address
      - A region size
      - A RAM start address
      - A RAM size

      Flash algorithms are loaded directly from the CMSIS pack you are using. Packs can have default and non-default algorithms. Default algorithms are loaded in the table when you first open the configuration.
   b) Add more algorithms by clicking the green + button below the table. The available algorithms are displayed. To add your own algorithm, browse to the location in your file system, or search for the algorithm using the text box at the bottom of the dialog box.
   c) Remove an algorithm by selecting it and clicking the red x button,
   d) Restore the default algorithm from the CMSIS pack by clicking the yellow arrow button.

   Like images, flash algorithms are validated to check if they exist. If the debugger cannot find the algorithm, it is highlighted red in the table. Additionally, flash algorithm ranges cannot overlap. If this happens, an error is displayed.
5. Click **Debug** to close the dialog box, and start the debug session.

Results

Arm Debugger connects to the selected target. On connection, Arm Debugger performs these steps:

- Resets the target using the method you selected.
- For new or changed images that are set to `Download`, runs the following command, where each `<image.axf>` is an image and its parameters:

  ```
  flash load-multiple <image1.axf> <image2.axf> ..... <imageN.axf>
  ```

- Loads the symbols for all of the images sequentially.
- Resets the target again using the selected method.
- For any image regions that were not loaded by the `flash load-multiple` command:
  - Runs the `restore` command, on the non-primary images, and sends them to RAM.
  - Runs the `load` command on the primary image, sends it to RAM, and sets the entry point.
- Starts debugging from the entry-point or main method, depending on which option you selected in the configuration.

Related information

Flash load command
Flash load-multiple command
Flash programming with Arm Debugger (video)

13 Writing OS Awareness for Arm Debugger

Describes the OS awareness feature available in Arm® Development Studio.

13.1 About Writing operating system awareness for Arm Debugger

Arm® Debugger offers an Application Programming Interface (API) for third parties to contribute awareness for their operating systems (OS).

The OS awareness extends the debugger to provide a representation of the OS threads - or tasks - and other relevant data structures, typically semaphores, mutexes, or queues.

Thread-awareness, in particular, enables the following features in the debugger:

- Setting breakpoints for a particular thread, or a group of threads.
- Displaying the call stack for a specific thread.
- For any given thread, inspecting local variables and register values at a selected stack frame.

To illustrate different stages of the implementation, this chapter explains how to add support for a fictional OS named `myos`.

The steps can be summarized as follows:

1. Create a new configuration database folder on your workstation to host the OS awareness extension and add it to the Arm DS **Preferences** in **Window > Preferences> Arm DS > Configuration Database**.
2. Create the files `extension.xml` and `messages.properties` so that the extension appears on the **OS Awareness** tab in the **Debug configuration** dialog box.
3. Add `provider.py` and implement the awareness enablement logic.
4. Add `contexts.py` and implement the thread awareness.
5. Add `tasks.py` to contribute a table to the **OS Data** view, showing detailed information about tasks.

13.2 Creating an OS awareness extension

A debugger operating systems (OS) awareness extension enables the debugger to provide a representation of the OS threads and other relevant data structures.

About this task

OS awareness extensions are created in the debugger configuration database. This procedure describes how to create the folder structure and definition files for an OS awareness extension. All files for an extension must be located in the configuration database `OS/` folder, either in a folder or at the root of a Java ARchive (JAR) file.

Procedure

1. Create a new configuration database folder containing an empty folder named `OS` (uppercase).
2. In the `OS` folder, do one of the following:
   - Create a new folder for the OS awareness extension
   - Create a JAR file
3. Add the following OS awareness extension files into the OS awareness extension folder or the root of the JAR file:
   - An `extension.xml` file to declare the OS awareness extension containing the following information:
     - The OS name, description, and, optionally, a logo to display in the **OS Awareness** selection pane
     - The root Python script or Java class providing the actual implementation
     - The details of cores, architectures, or platforms this implementation applies to

     The schema describing the structure of the `extension.xml` file can be found in the Arm DS installation folder at `sw/debugger/configdb/Schemas/os_extension.xsd`.
   - One or more message properties files that contain all user-visible strings and, optionally, the translations of those strings. The file format is documented in messages.properties format. The debugger searches for translations in the following order of language file names:
     a. `messages_<language code>_<country_code>.properties`
     b. `messages_<language code>.properties`
     c. `messages.properties`

     - Language codes: https://www.loc.gov/standards/iso639-2/php/English_list.php
     - Country codes: https://www.iso.org/iso-3166-country-codes.html
4. In Arm® Development Studio:
   a) Select **Window > Preferences**.
   b) In the **Preferences** dialog, expand **Arm DS** and select **Configuration Database**.
   c) Click **Add** and, in the **Add configuration database location** dialog box, enter:
      - A name for the new configuration database in the **Name** field
      - The path to the new configuration database folder in the **Location** field

Example 13-1: Example: Create an extension

This example implementation creates a `myos` OS awareness extension that appears in the Arm Development Studio Debug Configuration:

1. Create a `mydb` configuration database folder containing an `os` folder and, in this folder, create a new folder for the OS awareness extension:

```
<some folder>
   /mydb
      /OS
         /myos
```

2. Add the following files into the OS awareness extension folder:
   - An `extension.xml` file with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
   <os id="myos" version="5.15" xmlns="http://www.arm.com/os_extension">
   <name>myos.title</name>
   <description>myos.desc</description>
   <provider>provider.py</provider>
</os>
```

Note

The `version` attribute in the `os` element refers to the API version, which is aligned with the version of Arm DS that the API was released with. You must set the `version` attribute to the Arm DS version that the OS awareness extension was developed in, or the lowest version that it was tested with. The debugger does not display any extensions that require a higher version number. However, as the API is backwards compatible, the debugger displays extensions with earlier API versions. The OS awareness API in Arm DS is backwards compatible with versions of the OS awareness API in DS-5. The OS Awareness API versioning scheme in DS-5 matches the DS-5 product versions. Therefore, the earliest API version is 5.15.

   - A `messages.properties` file with the English language versions of all user-visible strings in the `myos` OS awareness extension:

```
myos.title=My OS
myos.desc=This is My OS.
myos.help=Displays information about My OS.
```

3. In Arm Development Studio:
   a. Select **Window > Preferences**.
   b. In the **Preferences** dialog, expand **Arm DS** and select **Configuration Database**.
   c. Click **Add**. In the **Add configuration database location** dialog box:
      - In the **Name** field, enter `My OS awareness extension`
      - in the **Location** field, enter the path to the `mydb` folder

Caution

This implementation is not complete and would cause errors if used for a debugger connection.

13.3 Implementing the OS awareness API

The OS awareness API consists of callbacks that the debugger makes at specific times. For each callback, the debugger provides a means for the implementation to retrieve information about the target and resolve variables and pointers, through an expression evaluator.

The API exists primarily as a set of Java interfaces since the debugger itself is written in Java. However, the debugger provides a Python interpreter and bindings to translate calls between Python and Java, allowing the Java interfaces to be implemented by Python scripts. This section and the next ones refer to the Java interfaces but explain how to implement the extension in Python.

Note

A Python implementation does not require any particular build or compilation environment, as opposed to a Java implementation. However, investigating problems within Python code is more difficult. You are advised to read the Programming advice and noteworthy information section before starting to write your own Python implementation.

The detailed Java interfaces to implement are available in the Arm DS installation folder under `sw/ide/plugins`, within the `com.arm.debug.extension.source_<version>.jar` file.

Note

You are encouraged to read the Javadoc documentation on Java interfaces as it contains essential information that is not presented here.

The Java interface of immediate interest at this point is `IOSProvider`, in the package `com.arm.debug.extension.os`. This interface must be implemented by the provider instance that was left out with a `todo` comment in `extension.xml`.

First, add the simplest implementation to the configuration database entry:

```
<some folder>
    /mydb
       /OS
           /myos
               /extension.xml
               /messages.properties
               /provider.py
```

- `extension.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<os id="myos" version="5.15" xmlns="http://www.arm.com/os_extension">
    <name>myos.title</name>
    <description>myos.desc</description>
    <provider>provider.py</provider>
</os>
```

- `provider.py`

```
# this script implements the Java interface IOSProvider
def areOSSymbolsLoaded(debugger):
    return False
def isOSInitialised(debugger):
    return False
def getOSContextProvider():
    return None
def getDataModel():
    return None
```

This is enough to make the OS awareness implementation valid. A debug configuration with this OS awareness selected works, although this does not add anything on top of a plain bare-metal connection. However, this illustrates the logical lifecycle of the OS awareness:

1. Ensure debug information for the OS is available. On loading symbols, the debugger calls `areOSSymbolsLoaded();` the implementation returns true if it recognizes symbols as belonging to the OS, enabling the next callback.
2. Ensure the OS is initialized. Once the symbols for the OS are available, the debugger calls `isOSInitialised()`, immediately if the target is stopped or whenever the target stops next. This is an opportunity for the awareness implementation to check that the OS has reached a state where threads and other data structures are ready to be read, enabling the next two callbacks.
3. Retrieve information about threads and other data structures. Once the OS is initialized, the debugger calls out to `getOSContextProvider()` and `getDataModel()` to read information from the target. In reality, the debugger may call out to `getOSContextProvider()` and `getDataModel()` earlier on, but does not use the returned objects to read from the target until `areOSSymbolsLoaded()` and `isOSInitialised()` both returned `true`.

13.4 Enabling the OS awareness

The below implementation in `provider.py`, assumes `myos` has a global variable called `tasks` listing the OS tasks in an array and another global variable `scheduler_running` indicating that the OS has started scheduling tasks.

```
# this script implements the Java interface IOSProvider
from osapi import DebugSessionException
def areOSSymbolsLoaded(debugger):
    return debugger.symbolExists("tasks") \
        and debugger.symbolExists("scheduler_running")
def isOSInitialised(debugger):
    try:
        result = debugger.evaluateExpression("scheduler_running")
        return result.readAsNumber() == 1
    except DebugSessionException:
        return False
def getOSContextProvider():
    return None
def getDataModel():
    return None
```

The `osapi` module in the import statement at the top of `provider.py` is a collection of wrappers around Java objects and utility functions. The file `osapi.py` itself can be found in JAR file `com.arm.debug.extension_<version>.jar`.

Connecting to a running target and loading symbols manually for the OS shows both `areOSSymbolsLoaded()` and `isOSInitialised()` stages distinctly.

On connecting to the target running the OS, without loading symbols, the **Debug Control** view displays **Waiting for symbols to be loaded**.

- After loading symbols for the OS, with the target still running, the **Debug Control** view now displays **Waiting for the target to stop**. At this point, `areOSSymbolsLoaded()` has been called and returned true, and the debugger is now waiting for the target to stop to call `isOSInitialised()`.
- As soon as the target is stopped, the **Debug Control** view updates to show the OS awareness is enabled. At this point, `isOSInitialised()` has been called and returned `true`.

  Both the `Active Threads` and `All Threads` folders are always empty until you implement thread awareness using `getOSContextProvider()`.

  Note

  You can show the `cores` folder by enabling the **Always Show Cores** option in the View Menu of the **Debug Control** view.

- Another case is where `areOSSymbolsLoaded()` returns `true` but `isOSInitialised()` returns `false`. This can happen, for instance, when connecting to a stopped target, loading both the kernel image to the target and associated symbols in the debugger and starting debugging from a point in time earlier than the OS initialization, for example, debugging from the image entry point.

  In this case, the **Debug Control** view shows **Waiting for the OS to be initialised** as `scheduler_running` is not set to `1` yet, but symbols are loaded.

  Without the call to `isOSInitialised()` the debugger lets the awareness implementation start reading potentially uninitialized memory, which is why this callback exists. The debugger keeps calling back to `isOSInitialised()` on subsequent stops until it returns true, and the OS awareness can finally be enabled.

13.5 Implementing thread awareness

Thread awareness is probably the most significant part of the implementation.

The corresponding call on the API is `getOSContextProvider()`, where `context` here means `execution context`, as in a thread or a task. The API expects an instance of the Java interface `IOSContextProvider` to be returned by `getOSContextProvider()`. This interface can be found in package `com.arm.debug.extension.os.context` within the same JAR file as `IOSProvider` mentioned earlier.

Given the following C types for myos tasks:

```
typedef enum {
   UNINITIALIZED = 0,
   READY
} tstatus_t ;
typedef struct {
   uint32_t            id;
   char                *name;
   volatile tstatus_t  status;
   uint32_t            stack[STACK_SIZE];
   uint32_t            *sp;
} task_t;
```

And assuming the OS always stores the currently running task at the first element of the tasks array, further callbacks can be implemented to return the currently running (or scheduled) task and all the tasks (both scheduled and unscheduled) in a new `contexts.py` file:

```
<some folder>
   /mydb
       /OS
           /myos
               /extension.xml
               /messages.properties
               /provider.py
               /contexts.py
```

- `provider.py`

```
# this script implements the Java interface IOSProvider
from osapi import DebugSessionException
from contexts import ContextsProvider
def areOSSymbolsLoaded(debugger):
    [...]
def isOSInitialised(debugger):
    [...]
def getOSContextProvider():
    # returns an instance of the Java interface IOSContextProvider
    return ContextsProvider()
def getDataModel():
    [...]
```

- `contexts.py`

```
from osapi import ExecutionContext
from osapi import ExecutionContextsProvider
# this class implements the Java interface IOSContextProvider
```

```
class ContextsProvider(ExecutionContextsProvider):
   def getCurrentOSContext(self, debugger):
     task = debugger.evaluateExpression("tasks[0]")
     return self.createContext(debugger, task)
   def getAllOSContexts(self, debugger):
     tasks = debugger.evaluateExpression("tasks").getArrayElements()
     contexts = []
     for task in tasks:
       if task.getStructureMembers()["status"].readAsNumber() > 0:
           contexts.append(self.createContext(debugger, task))
       return contexts
   def getOSContextSavedRegister(self, debugger, context, name):
       return None
   def createContext(self, debugger, task):
     members = task.getStructureMembers()
     id = members["id"].readAsNumber()
     name = members["name"].readAsNullTerminatedString()
     context = ExecutionContext(id, name, None)
     return context
```

Although `getOSContextSavedRegister()` is not yet implemented, this is enough for the debugger to now populate the **Debug Control** view with the OS tasks as soon as the OS awareness is enabled.

Decoding the call stack of the currently running task and inspecting local variables at specific stack frames for that task works without further changes since the task's registers values are read straight from the core's registers. For unscheduled tasks, however, `getOSContextSavedRegister()` must be implemented to read the registers values saved by the OS on switching contexts. How to read those values depends entirely on the OS context switching logic.

Here is the implementation for `myos`, based on a typical context switching routine for M-class Arm processors where registers are pushed onto the stack when a task is switched out by the OS scheduler:

```
from osapi import ExecutionContext
from osapi import ExecutionContextsProvider
STACK_POINTER = "stack pointer"
REGISTER_OFFSET_MAP = {"R4":0L,    "R5":4L,  "R6":8L,   "R7":12L,
                       "R8":16L,   "R9":20L, "R10":24L, "R11":28L,
                       "R0":32L,   "R1":36L, "R2":40L,  "R3":44L,
                       "R12":48L,  "LR":52L, "PC":56L,  "XPSR":60L,
                       "SP":64L}
# this class implements the Java interface IOSContextProvider
class ContextsProvider(ExecutionContextsProvider):
   def getCurrentOSContext(self, debugger):
       [...]
   def getAllOSContexts(self, debugger):
       [...]
   def getOSContextSavedRegister(self, debugger, context, name):
       offset = REGISTER_OFFSET_MAP.get(name)
       base = context.getAdditionalData()[STACK_POINTER]
       addr = base.addOffset(offset)
       if name == "SP":
           # SP itself isn't pushed onto the stack: return its computed value
           return debugger.evaluateExpression("(long)" + str(addr))
       else:
           # for any other register, return the value at the computed address
           return debugger.evaluateExpression("(long*)" + str(addr))
   def createContext(self, debugger, task):
       members = task.getStructureMembers()
       id = members["id"].readAsNumber()
       name = members["name"].readAsNullTerminatedString()
       context = ExecutionContext(id, name, None)
       # record the stack address for this task in the context's
```

```
        # additional data; this saves having to look it up later in
        # getOSContextSavedRegister()
        stackPointer = members["sp"].readAsAddress()
        context.getAdditionalData()[STACK_POINTER] = stackPointer
        return context
```

The debugger can now get the values of saved registers, allowing unwinding the stack of unscheduled tasks.

Note

Enter `info threads` in the **Commands** view to display similar thread information as displayed in the **Debug Control** view.

13.6 Implementing data views

Along with threads, OS awareness can provide arbitrary tabular data, which the debugger shows in the **OS Data** view.

The corresponding callback on the API is `getDataModel()`. It must return an instance of the Java interface `com.arm.debug.extension.datamodel.IDataModel`, which sources can be found in `com.arm.debug.extension.source_<version>.jar`.

This section demonstrates how to implement a view, listing the tasks, including all available information. The following additions to the implementation creates an empty **Tasks** table in the **OS Data** view:

```
<some folder>
   /mydb
       /OS
           /myos
               /extension.xml
               /messages.properties
               /provider.py
               /contexts.py
               /tasks.py
```

- `provider.py`

```
# this script implements the Java interface IOSProvider
from osapi import DebugSessionException
from osapi import Model
from contexts import ContextsProvider
from tasks import Tasks
def areOSSymbolsLoaded(debugger):
   [...]
def isOSInitialised(debugger):
   [...]
def getOSContextProvider():
   [...]
def getDataModel():
   # returns an instance of the Java interface IDataModel
   return Model("myos", [Tasks()])
```

- `messages.properties`

```
myos.title=My OS
myos.desc=This is My OS.
myos.help=Displays information about My OS.
tasks.title=Tasks
tasks.desc=This table shows all tasks, including uninitialized ones
tasks.help=Displays tasks defined within the OS and their current status.
tasks.id.title=Task
tasks.id.desc=The task identifier
tasks.name.title=Name
tasks.name.desc=The task name
tasks.status.title=Status
tasks.status.desc=The task status
tasks.priority.title=Priority
tasks.priority.desc=The task priority
tasks.sp.title=Stack pointer
tasks.sp.desc=This task's stack address
```

- `tasks.py`

```
from osapi import Table
from osapi import createField
from osapi import DECIMAL, TEXT, ADDRESS
# this class implements the Java interface IDataModelTable
class Tasks(Table):
    def __init__(self):
        id = "tasks"
        fields = [createField(id, "id", DECIMAL),
                  createField(id, "name", TEXT),
                  createField(id, "status", TEXT),
                  createField(id, "priority", DECIMAL),
                  createField(id, "sp", ADDRESS)]
        Table.__init__(self, id, fields)
    def getRecords(self, debugger):
        records = [] # todo
```

The `createField` and `Table.__init__()` functions automatically build up the keys to look for at run-time in the `messages.properties` file. Any key that is not found in `messages.properties` is printed as is.

The above modifications create a new empty `Tasks` table.

To populate the table, `getRecords()` in `tasks.py` must be implemented:

```
from osapi import Table
from osapi import createField
from osapi import createNumberCell, createTextCell, createAddressCell
from osapi import DECIMAL, TEXT, ADDRESS
# this class implements the Java interface IDataModelTable
 class Tasks(Table):
    def __init__(self):
        [...]
    def readTask(self, task, first):
        members = task.getStructureMembers()
        id = members["id"].readAsNumber()
        if (members["status"].readAsNumber() == 0):
            status = "Uninitialized"
            name = None
            sp = None
            priority = None
        else:
            if (first):
               status = "Running"
            else:
                status = "Ready"
            name = members["name"].readAsNullTerminatedString()
            sp = members["sp"].readAsAddress()
            priority = members["priority"].readAsNumber()
        cells = [createNumberCell(id),
                 createTextCell(name),
                 createTextCell(status),
                 createNumberCell(priority),
                 createAddressCell(sp)]
        return self.createRecord(cells)
    def getRecords(self, debugger):
        records = []
        tasks = debugger.evaluateExpression("tasks").getArrayElements()
        first = True
        for task in tasks:
            records.append(self.readTask(task, first))
            first = False
        return records
```

Note

The debugger command `info myos tasks` prints the same information in the **Commands** view.

13.7 Advanced OS awareness extension

You can extend the OS awareness features by defining more parameters.

The OS awareness extension might sometimes be unable to determine all the necessary information about the OS at runtime. There might be a compile option that controls the presence of a feature which fundamentally changes how a section of the OS works, and this might be undetectable at runtime. An example is whether the OS is compiled with hard or soft floating point support.

In the majority of cases, these features can be detected at runtime. However in rare instances this might not be possible. To deal with these cases, you can define parameters for the OS awareness extension, which you can then specify at runtime. For example, you can inform the OS awareness extension of the state of a compile flag. These parameters are then revealed to the debugger as operating system settings which can be queried from within the OS awareness Python scripts.

Note

This API feature is only available in API version `5.23` and later. You must change the `version` attribute of the `os` element to prevent running in incompatible versions.

The `extension.xml` file declares the OS awareness extension, as described in Creating an OS awareness extension. You can define the additional parameters by adding to the `os` element in `extension.xml` using the syntax that this example shows:

```
<parameter name="my-setting" description="myos.param.my_setting.desc" type="enum"
 default="enabled" help="myos.param.my_setting.help">
   <value name="enabled"  description="myos.param.my_setting.enabled"/>
   <value name="disabled" description="myos.param.my_setting.disabled"/>
</parameter>
```

For this example, you must also add the following to `message.properties`:

```
myos.param.my_setting.desc=My setting
myos.param.my_setting.help=This is my setting
myos.param.my_setting.enabled=Enabled
myos.param.my_setting.disabled=Disabled
```

Each parameter has:

type

The only supported type is `enum`.

name

The string used to identify the parameter within the debugger. The string can contain alphanumeric characters `a` - `z`, `A` - `Z` and `0` - `9`. The string can also contain – and _ but must not contain whitespace or any other special characters. The string must be unique among other parameters.

description

The localizable string shown in the GUI.

help

The localizable string shown in the parameter tooltip.

default

The `name` string corresponding to the default value of the parameter.

Each value has:

name

The string used to identify it within the debugger. The string can contain alphanumeric characters `a` - `z`, `A` - `Z` and `0` - `9`. The string can also contain – and _ but must not contain whitespace or any other special characters. The string must be unique among other parameters.

description

The localizable string shown in the GUI.

How to set the parameter value in Arm Development Studio

When you define these parameter settings in the OS awareness extension, you can set the parameter as an Operating System setting, either from the GUI or using the command-line. You can change or view these parameters using the `set os` and `showos` commands respectively, for example `set os my-setting enabled`.

Arm® Debugger shows these settings in the **OS Awareness** tab in the **Debug Configurations** dialog box

Note

When using the command-line debugger the parameter is set to its default value initially. It is then possible to manually change it after connecting to the target.

The parameters can be read from the OS awareness API using the `getConnectionSetting` method, for example `debugger.getConnectionSetting("os my-setting")`. This returns the `name` string of the selected value, or throws a `DebugSessionException` if the parameter does not exist.

13.8 Programming advice and noteworthy information

Investigating issues in Python code for an OS awareness extension can sometimes be difficult.

Here are a few recommendations to make debugging easier:

- Start Arm® Development Studio IDE from a console.

  Python `print` statements go to the Eclipse process standard output/error streams, which are not visible unless Arm Development Studio IDE is started from a console.

  - On Linux, open a new terminal and run: `<Arm DS installation folder>/bin/armds_ide`
  - On Windows, open command prompt and run: `<Arm DS installation folder>\bin\armds_idec`

    Note the trailing `c` in `armds_idec`.
- Turn on `verbose error` logging in the debugger.

  Any errors in the parts of the OS awareness logic which provide information to the debugger results in debugger errors being logged to the **Commands** view. Turning on verbose error logging prints the full stack trace when such errors occur, including source file locations, which can help identify the origin of the fault.

  To turn on verbose error logging, execute the following command early in the debug session:

  ```
  log config infoex
  ```

  Note

  - An OS awareness implementation interacts at the deepest level with the debugger, and some errors may cause the debugger to lose control of the target.
  - Semihosting is not available when OS awareness is in use.
- Any errors in the parts of the OS awareness logic which provide information to the debugger's user interface results in errors being logged to the Eclipse log. The log can be found in your

workspace as `.metadata/.log`. It contains full stack traces and timestamps for all such errors, including source file locations.

14 Debug and Trace Services Layer (DTSL)

Describes the Arm® Debugger Debug and Trace Services Layer (DTSL).

DTSL is a software layer that sits between the debugger and the RDDI target access API. Arm Debugger uses DTSL to:

- Create target connections.
- Configure the target platform to be ready for debug operations.
- Communicate with the debug components on the target.

As a power user of Arm Debugger, you might need to use DTSL:

- As part of new platform support.
- To extend the capabilities of Arm Debugger.
- To add support for custom debug components.
- To create your own Java or Jython programs which interact with your target.

14.1 Additional DTSL documentation and files

Additional DTSL documents and files are provided in `<installation_directory>\sw\DTSL`.

The following documents are useful for understanding DTSL. Make sure you have access to them.

DTSL object level documentation

DTSL is mainly written in Java, so the documentation takes the form of Javadoc files. The DTSL Javadoc is provided as HTML files (inside `com.arm.debug.dtsl.docs.zip`). You can view the HTML files directly in a browser, or use them from within the IDE.

Certain classes in the DTSL Javadocs are marked as Deprecated. These classes must not be used in new DTSL code. They are only provided in the documentation in case you encounter them when inspecting older DTSL code.

RDDI API documentation

DTSL is designed to use RDDI-DEBUG as its native target connection API. Some of the RDDI-DEBUG API is therefore referred to from within DTSL. For completeness, the RDDI documentation is included with the DTSL documentation.

The RDDI documentation is provided in HTML format in `<installation_directory>\sw\debugger\RDDI\docs\html`. To access the documentation, open `index.html`.

Also, make sure you have access to `DTSLExamples.zip`. This contains example DTSL code, in addition to the Arm® Development Studio configdb entries discussed in this document. This document assumes that you have added the examples to your Arm Development Studio IDE by importing the projects contained in this file.

Related information

Arm Debug Interface Architecture Specification

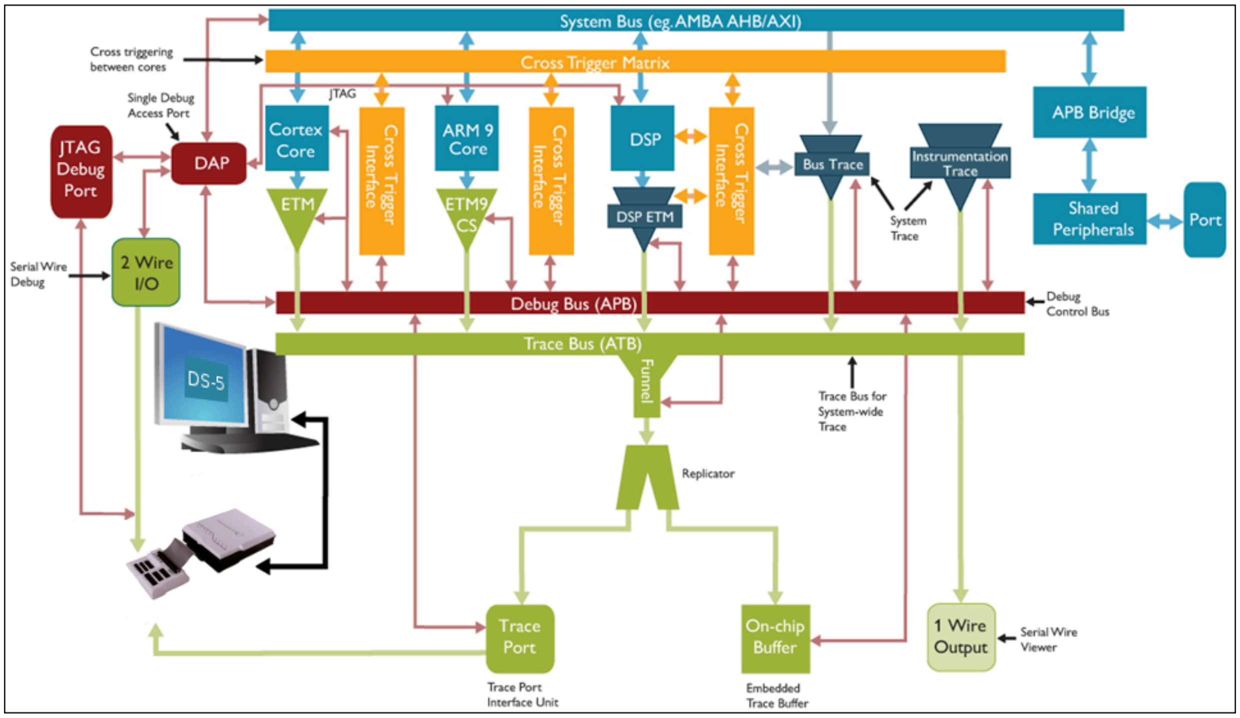
14.2 Need for DTSL

DTSL addresses the growing complexity and customization of Arm-based SoCs using the CoreSight™ Architecture. Before the creation of DTSL, most debug tools were designed at a time when SoC debug architecture was much simpler. SoCs typically contained only one core, and if multiple cores were used, they were of different types and were accessed by dedicated debug channels. Debug tools designed during that time, including Arm debuggers, cannot easily be scaled to more modern and complex debug architectures. DTSL is therefore designed to address several problems which older debug tools cannot easily address.

14.2.1 SoC design complexity

The debugger must be able to handle complex SoC designs which contain many cores and many debug components. For example, the following figure shows a relatively simple SoC design containing many debug components:

Figure 14-1: A simple CoreSight Design



Such systems are continuing to become more complicated as time goes on. For example, SoC designers might want to use multiple sub-systems which are accessed through multiple DAPs, but which are linked by multiple Cross Trigger Interfaces (CTIs) so that they can still be synchronized.

Each sub-system would have a similar design to that shown in the figure, but with shared CTIs and possibly shared TPIU.

Because system designs are so complicated, and vary so greatly, DTSL is designed to provide a layer of abstraction between the details of a particular system and the tools which provide debugging functionality to the user. For example, a debug tool using DTSL knows that there is a source of trace data for a particular core, and can access that data, but does not have to handle the complexities of system configuration and tool set-up in order to get that data. It does not have to know how to, for example, program up CoreSight Funnels, collect trace data from a DSTREAM, or demultiplex the TPIU trace protocol.

14.2.2 Debug flexibility

DTSL is designed to address the problems associated with the following:

- SoC designers sometimes add their own components, which are not part of any Arm standard. Debug tools might need to interact with these components to access the target.
- CoreSight™ designs can be very flexible, and early implementations might have design issues that the debug tool needs to work around.
- CoreSight designs can contain components which can be interconnected in many ways.

14.2.3 Integrated tool solutions

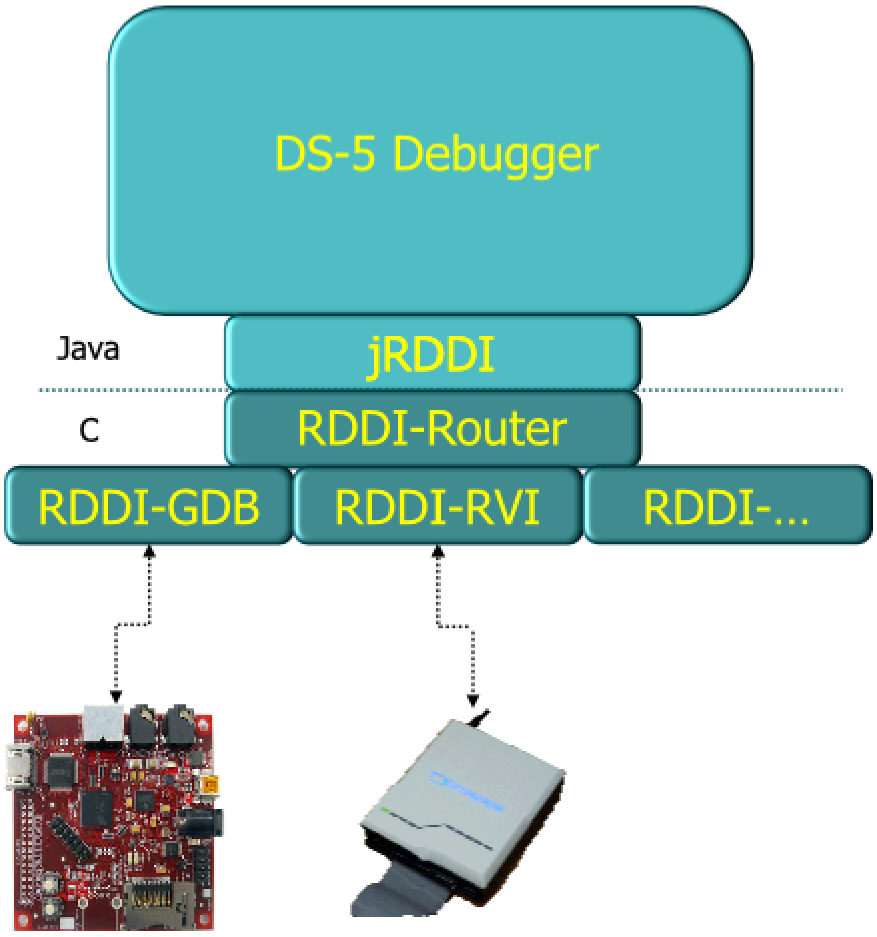
CoreSight™ designs can contain shared debug resources which need to be managed and used from multiple tools. For example, the system might be able to generate trace from several trace sources, such as Arm cores + DSP. In legacy designs, the trace paths would be independent and each debug tool would have its own connection to the respective sub-system. In a typical CoreSight system, the trace data is merged by a `Funnel` component and delivered to a single trace storage device though a single interface. The trace data is then uploaded and de-multiplexed. The trace data might need to be delivered to several different debug tools, such as Arm Development Studio and DSP Debug Tool.

DTSL addresses the tool integration problem that this situation raises.

14.2.4 Arm Debugger architecture before DTSL

Before DTSL first became available, the early Arm® Debugger Software stack was as shown in the following figure:

Figure 14-2: Debugger software stack before DTSL



From the bottom upwards, the components of the debug stack are:

RDDI-DEBUG API

The debugger uses this API as its standard native connection to a debug controller such as DSTREAM, CADI Model, or gdbserver. There is an implementation of `RDDI-DEBUG` for each of the supported types of debug controller.

RDDI-Router API

This API is identical to `RDDI-DEBUG`, but it is used to 'vector' the API calls to the appropriate implementation. This is necessary because the debugger can support multiple connections and connection types simultaneously.

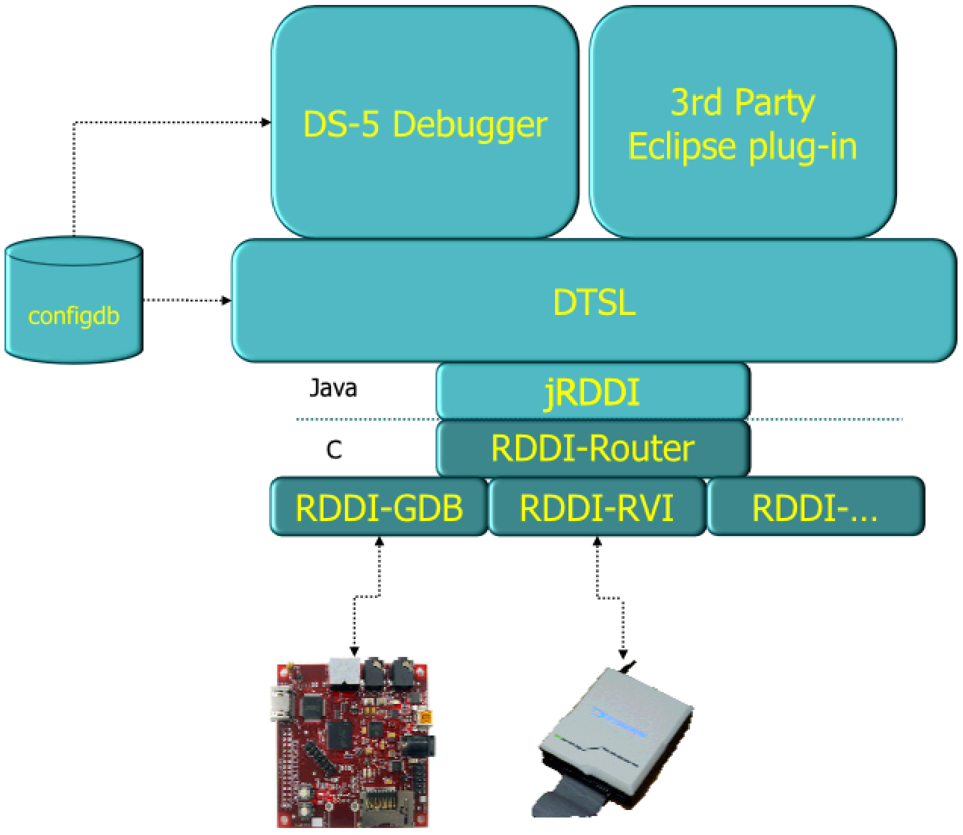
jRDDI

This is a Java wrapper for the C `RDDI-DEBUG` API. It is not a true Java layer, but nominally it is the lowest Java layer in the stack.

14.2.5 Arm Debugger architecture after DTSL

After DTSL was introduced, the Arm® Debugger Software stack changed. It is now as shown in the following figure:

Figure 14-3: Post DTSL



In addition to the layers that existed before DTSL, the stack now contains a DTSL layer which does the following:

- Handles system initialization and DTSL-level component creation. This is controlled by DTSL Jython scripts, which are typically contained in a platform configuration database (configdb).

Note

Do not confuse DTSL Jython Scripting with Arm Debugger Jython Scripting. Both of them use Jython, but they operate at different levels in the software stack. However, a Debugger Jython Script can use DTSL functionality.

- Provides a common connection interface for several client programs.
- Delivers trace streams to several trace clients.
- Uses the existing native RDDI infrastructure.

Arm Debugger uses DTSL to communicate with the lower layers of the stack. DTSL provides a set of named objects for the debugger (or another tool) to use. The object set consists of the following:

- Debug objects, which control core execution. Their interface looks very similar to the `jRDDI` and `RDDI-DEBUG` interfaces.
- Trace source interfaces, which represent target components which can generate trace data.
- Trace capture interfaces, which are used to start and stop trace collection and to provide notification events to clients.
- Other target components, such as other CoreSight™ devices or other third-party target devices.
- A Configuration and Connection interface, which instantiates and configures the DTSL objects and queries the configuration to allow clients to discover which top level interfaces are present.

Related information

DTSL access from Debugger Jython scripts on page 524

14.2.6 Arm Debugger connection sequence showing where DTSL fits in

The sequence below outlines the Arm® Debugger connection sequence when connecting to a target, and where DTSL fits in.

1. The user creates an Arm Development Studio launch configuration by selecting a platform (board) and a debug operation from the Arm Development Studio `configdb`. The user also specifies other debugger configuration parameters such as which file (`.axf`) to debug.
2. The user activates a launch configuration, either from the launch configuration editor or by selecting a previously prepared launch configuration.
3. The debugger launcher code locates the platform (board) entry in the Arm Debugger `configdb` and locates the DTSL configuration script. This script is run, and it creates the DTSL configuration.
4. The debugger connects to the DTSL configuration created by the script. It locates, by name, the object or objects identified by the debug operation specified in the `configdb` platform entry. It uses these objects to access the target device, including access to memory, registers, and execution state.

14.3 Arm Development Studio configuration database

All use cases of DTSL potentially require the use of the Arm® Development Studio configuration database (`configdb`). Arm therefore recommends that you have a basic understanding of the configuration database.

Arm Debugger uses a configuration database, called `configdb`, to store information on how to connect to platforms. This information is split across several top-level locations in the database, each of which can contain the following:

Board information
Manufacturer, name, list of SoCs, list of Flash, DTSL initialization Jython scripts.

SoC information
Core information. For example, a SoC may contain Cortex®-A9 + Cortex-M3.

Core information
Register sets for the core, and other information such as TrustZone support.

Flash information
Information on flash types and programming algorithms.

Common scripts (Jython)
Jython scripts which might be of use to several database entries.

This information is mainly stored as XML files, located in sub-directories of the top-level locations.

The configuration database is located at `<installation_directory>\sw\debugger\configdb`.

Arm Development Studio allows you to configure one or more extension configdb locations, which are typically used to add more board definitions or flash support to Arm Development Studio.

14.3.1 Modifying Arm Development Studio configdb

The Arm® Development Studio `configdb` is usually installed into a read-only location, to prevent accidental modification of the installed files. However, Arm Development Studio allows the user to install `configdb` extensions, which can be in a writeable location. The `configdb` extensions can also override the entries in the installed directories. To modify an installed `configdb` board entry (directory), you need to copy the installed entry into your `Documents` folder or `home` directory, modify it, and tell Arm Development Studio to add it as a `configdb` extension.

For example, to modify the Keil® MCBSTM32E platform files:

1. Create a `configdb` directory in your `Documents` folder or in another writeable location.
2. Create a `Boards` directory inside the `configdb` directory.
3. Copy the `Keil/MSCSTM32E` directory into the `Boards` directory.
4. Modify the copied `configdb` files.
5. Tell Arm Development Studio about the new `configdb` extension. To do this:
    a. Select **Window** > **Preferences** .
    b. Expand the **Arm DS** entry in the **Preferences** window.
    c. Select the **Configuration Database** entry.
    d. Click the **Add...** button to add the location of the new configuration database to the **User Configuration Databases** list.
    e. If you have modified any of the XML files in a `configdb` directory, you must tell Arm Development Studio to rebuild the database by clicking the **Rebuild database...** button on the Arm Development Studio **Configuration Database** preferences panel.

14.3.2 Configdb board files

Within the `configdb` is the `Boards` directory. It contains one sub-directory for each board manufacturer, and a `boards.xml` file which optionally provides human-readable display names for boards.

For example, the Keil® MCBSTM32E platform is a simple STM32E (Cortex®-M3) MCU board. The main `configdb` files are located in `<installation-directory>\sw\debugger\configdb\Boards\Keil\MCBSTM32E`, and are as follows:

project_types.xml

This is the main XML file which describes the platform entry to Arm® Development Studio.

keil-mcbstm32e.py

This is the DTSL platform configuration and setup script, implemented in Jython.

keil-mcbstm32e.rvc

This is the DSTREAM RDDI configuration file for the platform. This file can have an extension of either `.rcf` or `.rvc`. The Arm Development Studio **Platform Configuration Editor** usually creates this file.

keil-mcbstm32e_flash.xml

This contains information on flash devices and algorithms, and their configuration parameters.

14.3.3 About project_types.xml

The `project_types.xml` file defines the project types supported for the platform. Debug operations and activities, which refer to the other files in the platform directory, are defined for each project type.

The following code is part of the `project_types.xml` file for the Keil® MCBSTM32E platform.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!--Copyright (C) 2009-2013 ARM Limited. All rights reserved.-->
 <platform_data xmlns="http://www.arm.com/project_type" xmlns:peripheral="http://
com.arm.targetconfigurationeditor" xmlns:xi="http://www.w3.org/2001/XInclude"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" type="HARDWARE"
 xsi:schemaLocation="http://www.arm.com/project_type ../../../Schemas/
platform_data-1.xsd">
 <flash_config>CDB://keil-mcbstm32e_flash.xml</flash_config>
 <project_type_list>
  <project_type type="BARE_METAL">
   <name language="en">Bare Metal Debug</name>
   <description language="en">This allows a bare-metal debug connection.</
description>
   <execution_environment id="BARE_METAL">
    <name language="en">Bare Metal Debug</name>
    <description language="en">This allows a bare-metal debug connection.</
description>
    <param default="CDB://mcbstm32e.rvc" id="config_file" type="string"
 visible="false"/>
    <param default="CDB://mcbstm32e.py" id="dtsl_config_script" type="string"
             visible="false"/>
    <xi:include href="../../../Include/hardware_address.xml"/>
    <activity id="ICE_DEBUG" type="Debug">
```

```
      <name language="en">Debug Cortex-M3</name>
      <xi:include href="../../../Include/ulinkpro_activity_description.xml"/>
      <xi:include href="../../../Include/ulinkpro_connection_type.xml"/>
      <core connection_id="Cortex-M3" core_ref="Cortex-M3" soc="st/stm32f103xx"/>
      <param default="DebugOnly" id="dtsl_config" type="string" visible="false"/>
      <xi:include href="../../../Include/ulinkpro_browse_script.xml"/>
      <xi:include href="../../../Include/ulinkpro_setup_script.xml"/>
     </activity>
     <activity id="ICE_DEBUG" type="Debug">
      <name language="en">Debug Cortex-M3</name>
      <xi:include href="../../../Include/dstream_activity_description_bm.xml"/>
      <xi:include href="../../../Include/dstream_connection_type.xml"/>
      <core connection_id="Cortex-M3" core_ref="Cortex-M3" soc="st/stm32f103xx"/>
      <param default="DSTREAMDebugAndTrace" id="dtsl_config" type="string"
 visible="false"/>
      <param default="options.traceBuffer.traceCaptureDevice"
 id="dtsl_tracecapture_option"
             type="string" visible="false"/>
      <param default="ITM" id="eventviewer_tracesource" type="string"
 visible="false"/>
     </activity>
    </execution_environment>
   </project_type>
  </project_type_list>
 </platform_data>
```

The XML file declares a `BARE_METAL` project type. `BARE_METAL` is a term which describes a system not running an OS, where the debug connection takes full control of the core. The file declares an execution environment within the project type, and declares debug activities within that execution environment. The code here shows only one debug activity, but each execution environment can declare several debug activities. The debug activity shown here is a debug and trace session using a DSTREAM target connection.

When Arm® Development Studio displays the debug session launcher dialog box, it scans the entire configdb and builds a list of supported manufacturers and boards, and the supported project types and debug activities, and lets the user choose which one they want to use. In the following example, the user is assumed to have chosen the highlighted debug activity. When Arm Debugger launches the debug session, it creates a DTSL configuration and passes it the `{config_file, dtsl_config_script, dtsl_config}` property set. These parameters are used as follows:

`<config_file>`

This value is passed to the RDDI-DEBUG connection DLL or so (Arm® Development Studio uses RDDI-DEBUG as its target connection interface, and RDDI-DEBUG needs this file to tell it which devices are in the target system).

`<dtsl_config_script>`

This value tells DTSL which Jython script to use to create the DTSL configuration used by the debugger.

`<dtsl_config>`

The DTSL Jython script can contain several system configurations, defined by Jython class names which in turn are derived from the DTSLv1 object. This value tells DTSL which class to create. The MCBSTM32E Jython script contains two such classes, one for a debug and trace configuration and one for a debug-only configuration. The class name used for the highlighted example is `DSTREAMDebugAndTrace`, so in this example a Jython class named `DSTREAMDebugAndTrace` must exist in the `dtsl_config_script`.

Some of these entries have a file location prefix of `CDB://`. This indicates that the location is within the platform directory in the configuration database.

DTSL creates an instance of the referenced Jython class, which causes the `def __init__(self, root)` constructor to be run. After this constructor is run, the debugger expects to find a DTSL Device object whose name is the same as the name given in the core setting in the debug activity. In this example, therefore, the debugger expects to find a DTSL object named 'Cortex-M3', and it directs all debug operation requests to this object.

14.3.4 About the keil-mcbstm32e.py script

The complete content of the `keil-mcbstm32e.py` file for the Keil® MCBSTM32E platform is included in the `DTSLExampleConfigdb` project in `DTSLExamples.zip`. The important aspects of the script are as follows:

- The script is written in Jython. Jython is an implementation of the Python language which integrates tightly with Java. The integration is tight enough to allow the following:
    - A Jython script can contain Python classes which Java can use, and which appear to Java as though they were Java classes.
    - A Jython script can contain Python classes which can sub-class Java classes.
    - A Jython script can create Java class instances and use them. This is why the script contains some `import` statements which import Java classes. Many of these classes are from the `com.arm.debug.dtsl` package.
- DTSL creates an instance of a class named `DSTREAMDebugAndTrace`.
- The constructor `__init__` creates all the DTSL objects required for the connection.
- The RDDI-DEBUG API, which is the native API used by the debugger for target access, assigns each device a unique device index number. The script contains lines which find the index number for a named device and assign that number to a variable. The following is an example of such a line: `devID = self.findDevice("Cortex-M3")` This line assigns the RDDI device index number for the named device 'Cortex-M3' to the variable `devID`.
- The script creates a `ResetHookedDevice` object, derived from `Device`, with the name 'Cortex-M3'. This is an example of how Jython can extend the standard DTSL Java classes by sub-classing them.
- The script creates an `AHBCortexMMemAPAccessor` and installs it into the Cortex®-M3 object as a memory filter. This is how a custom named memory space is added to the core. When a memory access is requested with an address prefixed by 'AHB', the access is redirected to the `AHBCortexMMemAPAccessor` object which, in this case, uses the CoreSight™ AHB-AP to access the memory.
- The script creates DTSL objects for the CoreSight components in the SoC.
- The script creates a `DSTREAMTraceCapture` object, which the debugger uses to read trace data.
- The script declares a set of options which provide user configuration data for the script. The debug session launcher panel displays these options so that they can be set before making a

target connection. After the constructor is called, DTSL passes the option values to the class by calling its `optionValuesChanged()` method.

```
from com.arm.debug.dtsl.configurations import DTSLv1
[snip]
class ResetHookedDevice(Device):
    def __init__(self, root, devNo, name):
        Device.__init__(self, root, devNo, name)
        self.parent = root
    def systemReset(self, resetType):
        Device.systemReset(self, resetType)
        # Notify root configuration
        self.parent.postReset()
class DSTREAMDebugAndTrace(DTSLv1):
    '''A top level configuration class which supports debug and trace'''
    @staticmethod
    def getOptionList():
        '''The method which specifies the configuration options which
           the user can edit via the launcher panel |Edit...| button
        '''
        return [
            DTSLv1.tabSet(
                name='options',
                displayName='Options',
                childOptions=[
                    DSTREAMDebugAndTrace.getTraceBufferOptionsPage(),
                    DSTREAMDebugAndTrace.getETMOptionsPage(),
                    DSTREAMDebugAndTrace.getITMOptionsPage()
                ]
            )
        ]
    @staticmethod
    def getTraceBufferOptionsPage():
        # If you change the position or name of the traceCapture
        # device option you MUST modify the project_types.xml to
        # tell the debugger about the new location/name
        return DTSLv1.tabPage(
            name='traceBuffer',
            displayName='Trace Buffer',
            childOptions=[
                DTSLv1.enumOption(
                    name='traceCaptureDevice',
                    displayName='Trace capture method',
                    defaultValue='DSTREAM',
                    values=[
                        ('none', 'No trace capture device'),
                        ('DSTREAM', 'DSTREAM 4GB Trace Buffer')
                    ]
                ),
                DTSLv1.booleanOption(
                    name='clearTraceOnConnect',
                    displayName='Clear Trace Buffer on connect',
                    defaultValue=True
                ),
                DTSLv1.booleanOption(
                    name='startTraceOnConnect',
                    displayName='Start Trace Buffer on connect',
                    defaultValue=True
                ),
                DTSLv1.enumOption(
                    name='traceWrapMode',
                    displayName='Trace full action',
                    defaultValue='wrap',
                    values=[
                        ('wrap', 'Trace wraps on full and continues to store data'),
                        ('stop', 'Trace halts on full')
                    ]
                )
            ]
        )
[snip]
    def __init__(self, root):
        '''The class constructor'''
        # base class construction
        DTSLv1.__init__(self, root)
        # create the devices in the platform
        self.cores = []
        self.traceSources = []
        self.reservedATBIDs = {}
        self.createDevices()
        self.setupDSTREAMTrace()
        for core in self.cores:
            self.addDeviceInterface(core)
    def createDevices(self):
# create MEMAP
    devID = self.findDevice("CSMEMAP")
        self.AHB = CortexM_AHBAP(self, devID, "CSMEMAP")
        # create core
        devID = self.findDevice("Cortex-M3")
        self.cortexM3 = ResetHookedDevice(self, devID, "Cortex-M3")
        self.cortexM3.registerAddressFilters(
                [AHBCortexMMemAPAccessor("AHB", self.AHB, "AHB bus accessed via
 AP_0")])
        self.cores.append(self.cortexM3)
        # create the ETM disabled by default - will enable with option
        devID = self.findDevice("CSETM")
        self.ETM = V7M_ETMTraceSource(self, devID, 1, "ETM")
        self.ETM.setEnabled(False)
        self.traceSources.append(self.ETM)
        # ITM disabled by default - will enable with option
        devID = self.findDevice("CSITM")
        self.ITM = V7M_ITMTraceSource(self, devID, 2, "ITM")
        #self.ITM = M3_ITM(self, devID, 2, "ITM")
        self.ITM.setEnabled(False)
        self.traceSources.append(self.ITM)
        # TPIU
        devID = self.findDevice("CSTPIU")
        self.TPIU = V7M_CSTPIU(self, devID, "TPIU", self.AHB)
        # DSTREAM
        self.DSTREAM = DSTREAMTraceCapture(self, "DSTREAM")
        self.DSTREAM.setTraceMode(DSTREAMTraceCapture.TraceMode.Continuous)
```

14.3.5 DTSL script

The DTSL script defines the DTSL options using a set of static methods. The option definitions must be available before creating an instance of the configuration class.

To display and modify the DTSL options before connecting, use the IDE launcher panel. To display and modify the DTSL options during an Arm® Development Studio debug session, use the command line or the Debug Control view.

In Windows 10, the DTSL options values are persisted in your workspace under the directory `C:\Users\<user>\Documents\ArmDS_Workspace\.metadata\.plugins\com.arm.ds\DTSL`. In this directory there is a sub-directory for the platform, in which there is another sub-directory for the debug operation. Within the debug operation directory there are one or more `.dtslprops` files, whose names match the names option sets in the DTSL Options dialog box. These files are standard Java properties files. The following is the default properties file for the Keil® MCBSTM32E Platform, Bare Metal Project, Debug and Trace Debug operation:

```
options.ETM.cortexM3coreTraceEnabled=true
options.ITM.itmTraceEnabled=true
options.ITM.itmTraceEnabled.itmowner=Target
options.ITM.itmTraceEnabled.itmowner.target.targetITMATBID=2
options.ITM.itmTraceEnabled.itmowner.debugger.DWTENA=true
options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[15\:8]=true
options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[23\:16]=true
options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[31\:24]=true
options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[7\:0]=true
options.ITM.itmTraceEnabled.itmowner.debugger.STIMENA=0xFFFFFFFF
options.ITM.itmTraceEnabled.itmowner.debugger.TSENA=true
options.ITM.itmTraceEnabled.itmowner.debugger.TSPrescale=none
options.traceBuffer.traceCaptureDevice.clearTraceOnConnect=true
options.traceBuffer.traceCaptureDevice.startTraceOnConnect=true
options.traceBuffer.traceCaptureDevice.traceWrapMode=wrap
options.traceBuffer.traceCaptureDevice=DSTREAM
```

The names of the options exactly match the name hierarchy defined in the DTSL script (see the full DTSL script source code to create the configuration options).

When Arm Debugger displays the options, it calls the `getOptionList()` method in the DTSL configuration class to retrieve a data description of the options. It matches these options with the persisted values from the `.dtslprops` file and transforms this data into an interactive dialog type display for the user. When the user saves the options, the `.dtslprops` file is updated. After the DTSL configuration instance is created, DTSL calls the `optionValuesChanged()` method to inform the instance of the configuration settings values. During the debug session, the user can change any option which is marked with an `isDynamic=True` property.

Related information

DTSL options on page 535

14.4 DTSL as used by Arm Debugger

14.4.1 Arm Development Studio debug session launcher

After you have created a new debug connection, the debug session begins.

When the session starts running, Arm® Debugger first scans the entire Arm Development Studio configdb, including the extension directories. It dynamically builds a list of supported manufacturers and boards, along with the supported project types and debug activities. To build this list, Arm Debugger refers to the `project_types.xml` files in each `/Boards` directory.

When built, the list is displayed in the **Edit Configuration** dialog box. You can then choose the combination of manufacturer, board, project type, and debug activity you require.

After you select the debug activity, Arm Debugger inspects the DTSL script `dtsl_config_script`, and configuration class `dtsl_config`, for any DTSL options. If any DTSL options are specified, Arm Debugger activates the **Edit...** button so that you can change the values for the DTSL options.

14.4.2 Connecting to DTSL

To use DTSL, a client must create a `DTSLConnection` object using the DTSL `ConnectionManager` class (`com.arm.debug.dtsl.ConnectionManager`). `ConnectionManager` has static methods that allow the `DTSLConnection` object to be created from a set of connection parameters. After a `ConnectionManager` object is obtained, calling its `connect()` method creates the `DTSLConfiguration` object which contains all the target component objects.

When the DTSL `ConnectionManager` class creates a new `DTSLConnection`, it assigns a unique key to it. It constructs this key from the connection properties:

- `dtsl_config_script`: the absolute path to the DTSL Jython script.
- `dtsl_config`: the Jython DTSL class name.
- `config_file`: the absolute path to the RDDI configuration file.
- `dtsl_config_options`: optional DTSL options (a hash of the content of the DTSL options file).
- `rddi_retarget_address`: optional re-target address for the RDDI configuration.
- possibly other input.

If the DTSL `ConnectionManager` detects an attempt to connect to an already existing `DTSLConnection` (that is, the connection key matches an existing `DTSLConnection` instance) then DTSL returns the already existing instance. There can only be one `DTSLConnection` with any given key.

A `DTSLConnection` can also be created by obtaining an existing DTSL instance key and requesting a connection to that instance. Both Arm® Debugger and third-party Eclipse plugins can therefore connect to an existing `DTSLConnection` instance. If Arm Debugger creates the `DTSLConnection` instance for a platform, then a third-party plugin can connect to the same instance by one of two methods:

- Use an identical set of connection properties.
- Arrange to get the `DTSLConnection` instance key from the debugger, and use that key to make the connection.

DTSL reference-counts connections to a platform instance and only closes the `DTSLConnection` instance when all clients have disconnected.

14.4.3 DTSL access from Debugger Jython scripts

DTSL uses Jython scripting to create the DTSL configuration. The configuration typically stores objects for each debug component in the target system.

Arm® Debugger also uses Jython scripting, but at a different level, to DTSL, in the debugger software stack. In debugger scripting, the debugger provides an object interface to the debugger features. For example, a debugger script can:

- load `.axf` files
- determine the current execution context
- read registers
- set breakpoints
- control execution

These operations cause operations on the objects in the DTSL configuration, but there is not always a direct mapping from debugger operations to DTSL object operations. This is especially true for SMP systems.

Sometimes, however, it makes sense for a debugger script to access low level DTSL objects. For example, a user with in-depth CoreSight™ experience might want to manually program up a PTM sequencer, or directly drive CTI inputs. In such cases, the debugger script can get the DTSL configuration, locate the objects of interest and call their methods directly. Although this is a very powerful feature, it must be used with care, because the debugger has no way of knowing that such operations have taken place. In many cases this does not matter, especially if the DTSL objects being used are not directly used by the debugger. However, more care is required when directly accessing core objects used by the debugger.

The following is an example of how a debugger Jython script might get access to a DTSL object called 'PTM':

```
from arm_ds.debugger_v1 import Debugger
from com.arm.debug.dtsl import ConnectionManager
from com.arm.debug.dtsl.interfaces import IConfiguration
# Connect to Arm Debugger
debugger = Debugger()
assert isinstance(debugger, Debugger)
if not debugger.isConnected():
   return
# Access the underlying DTSL configuration
dtslConnectionConfigurationKey = debugger.getConnectionConfigurationKey()
dtslConnection = ConnectionManager.openConnection(dtslConnectionConfigurationKey)
dtslConfiguration = dtslConnection.getConfiguration()
assert isinstance(dtslConfiguration, IConfiguration)
deviceList = dtslConfiguration.getDevices()
for device in deviceList:
   assert isinstance(device, IDevice)
   if device.getName() == "PTM":
      ...
```

14.5 Main DTSL classes and hierarchy

There are four basic types of object that DTSL exposes to the Debugger or third-party plugin:

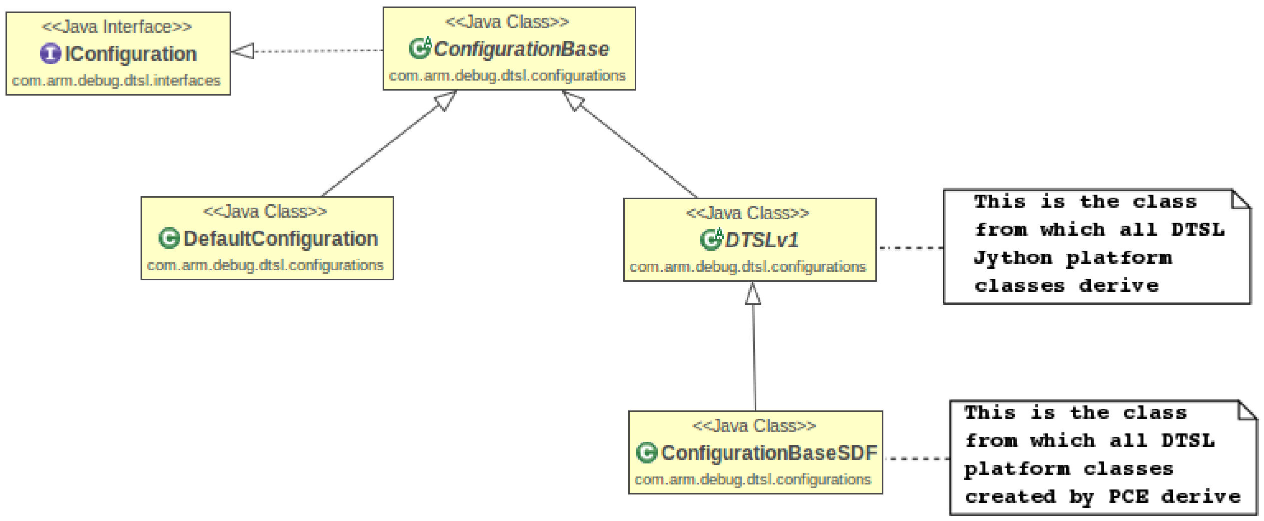
- Connection and Configuration objects, which implement the `IConnection` and `IConfiguration` interfaces respectively.
- Device objects, which implement the `IDevice` interface. Cores, and most CoreSight™ components, are of this type. If a device needs a connection type operation, which most devices do, then it also implements `IDeviceConnection` (see the `ConnectableDevice` object).
- TraceSource objects, which typically implement both the `IDevice` and `ITraceSource` interfaces. ETM and PTM objects are of this type.
- Trace capture devices, which typically implement the `ITraceCapture` interface. These objects give access to a trace capture device such as a DSTREAM or an ETB.

14.5.1 DTSL configuration objects

The `DTSLConnection` object is the top-level DTSL object that allows access to all the other DTSL objects using the platform configuration.

Specifically, the `DTSLConnection` allows access to the `ConfigurationBase` instance, for example DTSLv1, which allows access to the rest of the DTSL objects. The content of the platform configuration depends on the associated `ConnectionParameters` set.

Figure 14-4: DTSL Configuration class hierarchy


<<Java Interface>>
IConfiguration
com.arm.debug.dtsl.interfaces
<<Java Class>>
ConfigurationBase
com.arm.debug.dtsl.configurations
<<Java Class>>
DefaultConfiguration
com.arm.debug.dtsl.configurations
<<Java Class>>
DTSLv1
com.arm.debug.dtsl.configurations
This is the class from which all DTSL Jython platform classes derive
<<Java Class>>
ConfigurationBaseSDF
com.arm.debug.dtsl.configurations
This is the class from which all DTSL platform classes created by PCE derive

If the `ConnectionParameters` instance does not specify a DTSL configuration script, then an object of type `DefaultConfiguration` is created. The configuration content is constructed by creating a `Device` object for each device known to RDDI-DEBUG. For DSTREAM, this means that a `Device` object is created for each device declared in the `.rcf`, `.rvc`, or `.sdf` files, but for other kinds of RDDI this might come from a different data set. This allows for a simple connection to a platform with direct connections to any target devices specified in the RDDI configuration file.

If the `ConnectionParameters` instance does specify a DTSL configuration script, then that script is run to create an instance of a configuration object derived from DTSLv1. When the configuration script is run, it is expected to populate the configuration with the set of known device objects, trace sources, and trace capture devices.

Note

- Arm recommends using a configuration script to create a DTSL configuration, because it allows much greater flexibility when creating devices.
- DTSLv1 is named as such to show that the configuration is using the V1 interface and object set. This is the current set. If Arm changes the interface and object set, then it might start using DTSLv2. This allows Arm to maintain backwards compatibility, but also to move forward with new or modified interfaces.

There are two object hierarchies in use for DTSL configurations, that can be split into:

1. Configurations which use `.rcf` or `.rvc` files.

   For these types of configurations, all information related to the topology of the system is contained within the Jython configuration script, where managed devices, trace component orders, and device configuration (For example, funnel port configuration) are all performed explicitly within the script. The Arm® Development Studio Platform Configuration Editor previously created all configurations which behaved in this way. For more details, see Arm DS configuration database.
2. Configurations which use a `.sdf` file directly.

   SDF files can contain all required information related to target device topology, and so configurations which use them directly do not have large amounts of python configuration code, and all this work is performed internally by the `ConfigurationBaseSDF` class. Configurations created by the Arm Development Studio Platform Configuration Editor use a `.sdf` file as an input. For more details, see Platform Configuration .

14.5.2 DTSL device objects

`Device` objects are used to interface to any target component that has an RDDI-DEBUG interface. Such components are typically cores or CoreSight™ devices. All `Device` objects implement the `IDevice` interface, which closely matches the RDDI-DEBUG native interface.

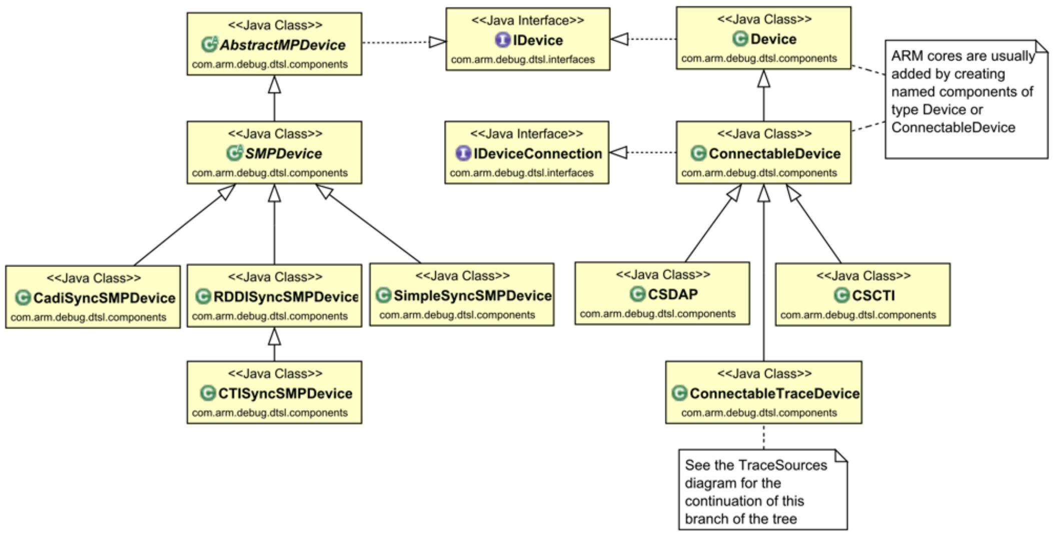
The following is a code sequence from a DTSL Jython script to create the `Device` object for a Cortex®-A8 core:

```
1. devID = self.findDevice("Cortex-A8")
2. self.cortexA8 = ConnectableDevice(self, devID, "Cortex-A8")
3. self.addDeviceInterface(self.cortexA8)
```

Line 1 locates the device ID (RDDI-DEBUG device index number) for the named device from the RDDI configuration. Line 2 creates the DTSL `ConnectableDevice` object. Line 3 adds the device object to the DTSL configuration.

The following figure shows part of the Device class hierarchy:

Figure 14-5: DTSL Device object hierarchy


<<Java Class>>
AbstractMPDevice
com.arm.debug.dtsl.components
<<Java Interface>>
IDevice
com.arm.debug.dtsl.interfaces
<<Java Class>>
Device
com.arm.debug.dtsl.components
ARM cores are usually added by creating named components of type Device or ConnectableDevice
<<Java Class>>
SMPDevice
com.arm.debug.dtsl.components
<<Java Interface>>
IDeviceConnection
com.arm.debug.dtsl.interfaces
<<Java Class>>
ConnectableDevice
com.arm.debug.dtsl.components
<<Java Class>>
CadiSyncSMPDevice
com.arm.debug.dtsl.components
<<Java Class>>
RDDISyncSMPDevice
com.arm.debug.dtsl.components
<<Java Class>>
SimpleSyncSMPDevice
com.arm.debug.dtsl.components
<<Java Class>>
CSDAP
com.arm.debug.dtsl.components
<<Java Class>>
CSCTI
com.arm.debug.dtsl.components
<<Java Class>>
CTISyncSMPDevice
com.arm.debug.dtsl.components
<<Java Class>>
ConnectableTraceDevice
com.arm.debug.dtsl.components
See the TraceSources diagram for the continuation of this branch of the tree

Note

The figure shows the main components used for cores and core clusters.

14.5.3 CoreSight device component register IDs

The documentation for a CoreSight component lists its component registers and their address offsets. For example, the CoreSight™ STM component has a Trace Control and Status Register called `STMTCSR` which has an offset of `0xE80`. To access this register through the `IDevice` interface, you need to know its register ID. To determine the ID, divide the documented offset by four. For example, the register ID for the `STMTCSR` register is `0x3A0`, which is `0xE80`/4.

14.5.4 DTSL trace source objects

These objects represent sources of trace data within the platform. These could be Arm devices such as:

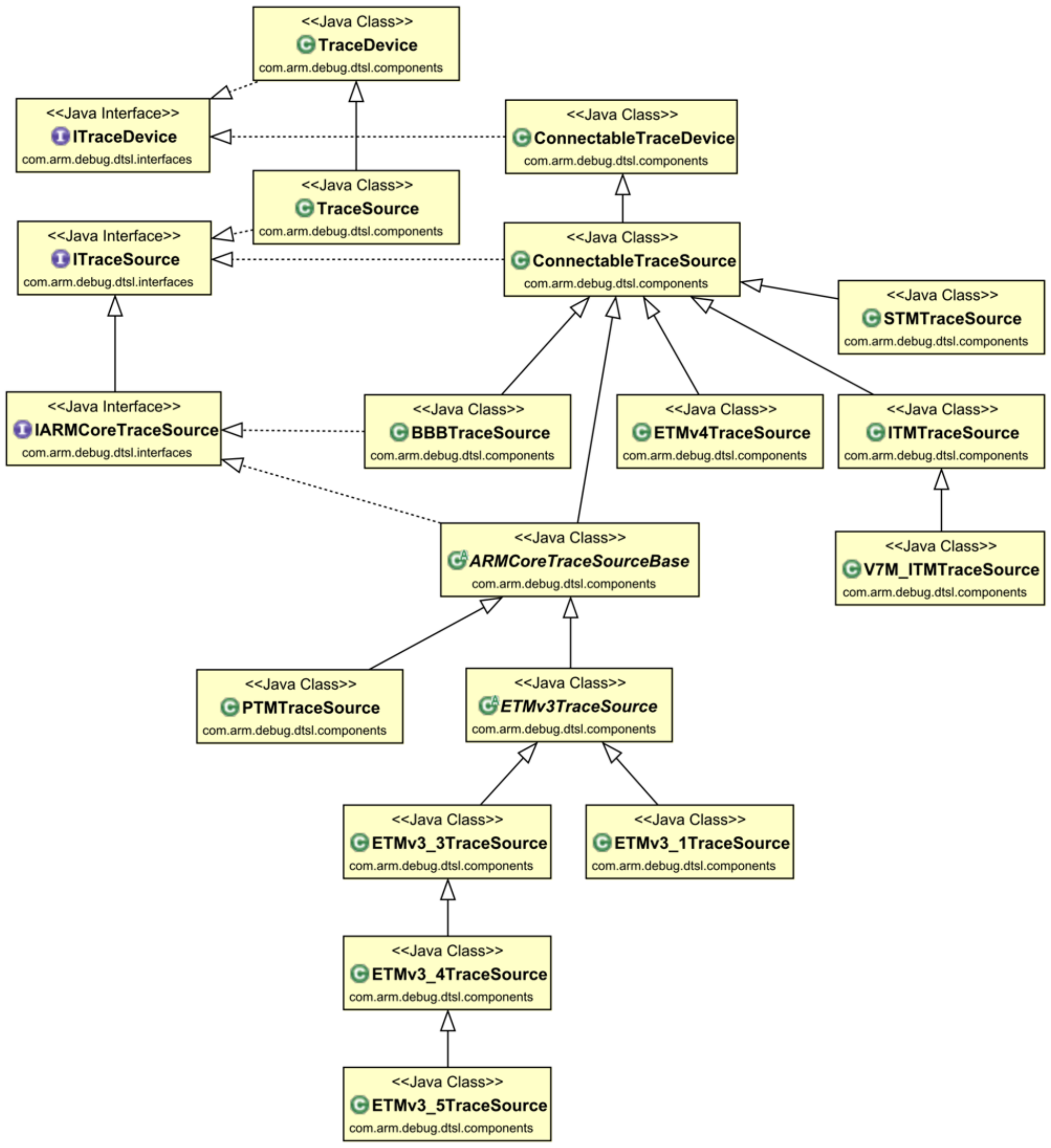
- ETM
- PTM
- ITM
- STM
- MTB (previously known as BBB)

- Custom trace components that output data onto the CoreSight™ ATB

These devices must implement the `ITraceSource` interface to be recognized as a trace source and to provide ATB ID information. They typically also implement `IDevice`. Most of these types of device only implement the register access methods from `IDevice` to allow configuration and control of the device, and they usually have a partner class which defines the names of the registers supported. For example, the `STMTraceSource` class has a partner class called `STMRegisters` which, for convenience, defines the STM register set IDs and many of their bit fields.

The class hierarchy for trace source objects is shown in the following figure:

Figure 14-6: DTSL Trace Source class hierarchy


<<Java Class>> TraceDevice com.arm.debug.dtsl.components
<<Java Interface>> ITraceDevice com.arm.debug.dtsl.interfaces
<<Java Class>> ConnectableTraceDevice com.arm.debug.dtsl.components
<<Java Class>> TraceSource com.arm.debug.dtsl.components
<<Java Interface>> ITraceSource com.arm.debug.dtsl.interfaces
<<Java Class>> ConnectableTraceSource com.arm.debug.dtsl.components
<<Java Class>> STMTraceSource com.arm.debug.dtsl.components
<<Java Interface>> IARMCoreTraceSource com.arm.debug.dtsl.interfaces
<<Java Class>> BBBTraceSource com.arm.debug.dtsl.components
<<Java Class>> ETMv4TraceSource com.arm.debug.dtsl.components
<<Java Class>> ITMTraceSource com.arm.debug.dtsl.components
<<Java Class>> ARMCoreTraceSourceBase com.arm.debug.dtsl.components
<<Java Class>> V7M_ITMTraceSource com.arm.debug.dtsl.components
<<Java Class>> PTMTraceSource com.arm.debug.dtsl.components
<<Java Class>> ETMv3TraceSource com.arm.debug.dtsl.components
<<Java Class>> ETMv3_3TraceSource com.arm.debug.dtsl.components
<<Java Class>> ETMv3_1TraceSource com.arm.debug.dtsl.components
<<Java Class>> ETMv3_4TraceSource com.arm.debug.dtsl.components
<<Java Class>> ETMv3_5TraceSource com.arm.debug.dtsl.components

When implementing new trace source objects, you can choose to base them on `TraceDevice`, `ConnectableTraceDevice`, `TraceSource`, or `ConnectableTraceSource`. The choice depends on whether the source needs a connection, and whether it can identify itself in the trace stream with a source ID. As shown in the figure, all the standard Arm trace sources are derived from `ConnectableTraceSource`. This is because they are real devices which can be connected to for configuration, and which have ATB IDs to identify themselves in the received trace stream.

The following is a typical code sequence from a DTSL Jython script to create an ETM trace source:

```
1. devID = self.findDevice("CSETM")
2. etmATBID = 1
3. self.ETM = ETMv3_3TraceSource(self, devID, etmATBID, "ETM")
```

Line 1 locates the CSETM device ID (RDDI-DEBUG device index number) from the RDDI configuration. Line 2 assigns the ATB ID to be used for the ETM. Line 3 creates the DTSL `ETMv3_3TraceSource` object and names it 'ETM'. If there are multiple ETMs in the platform, they should have different names, such as 'ETM_1' and 'ETM_2', or 'ETM_Cortex-A8' and 'ETM_Cortex-M3'.

After creating the trace source objects, you must inform any trace capture device about the set of trace source objects to associate with it. This allows the client program to locate the ATB ID for the source of interest and request delivery of trace data for that source.

Related information

DTSL trace capture objects on page 530

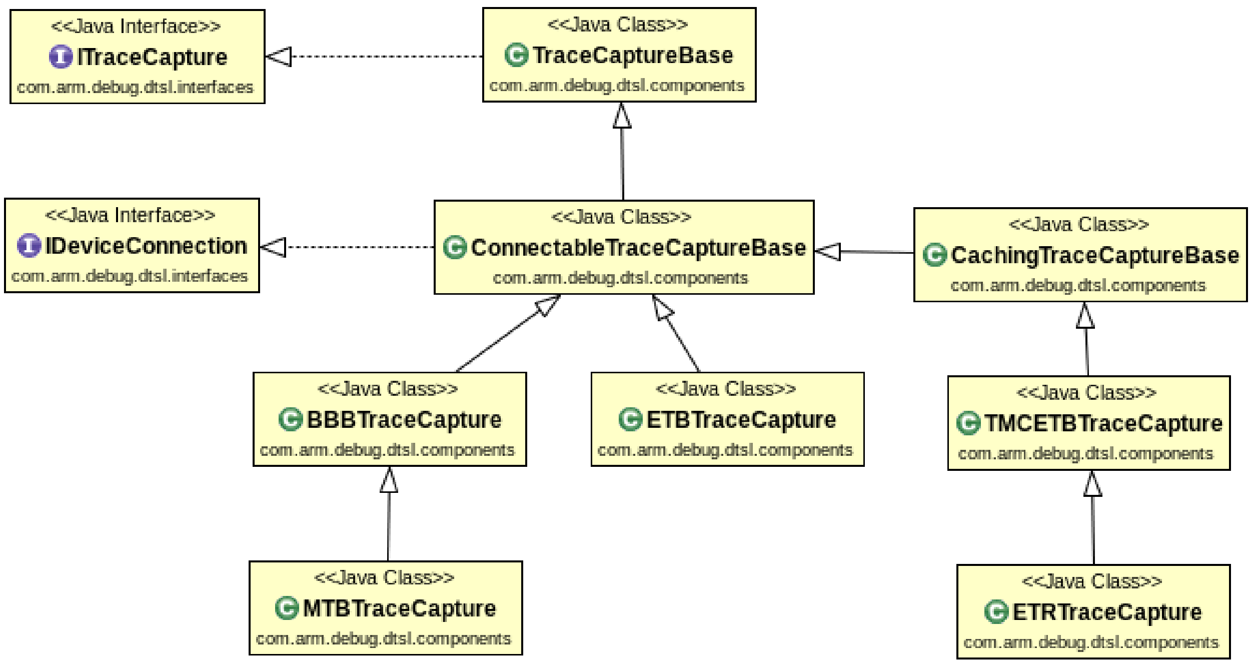
14.5.5 DTSL trace capture objects

Trace capture objects are responsible for storing and delivering trace data.

Some trace capture devices reside on the platform itself, such as CoreSight™ ETB, TMC/ETB, and TMC/ETR. In other cases, trace capture devices capture data into an off-platform storage, such as DSTREAM with its 4GB trace buffer.

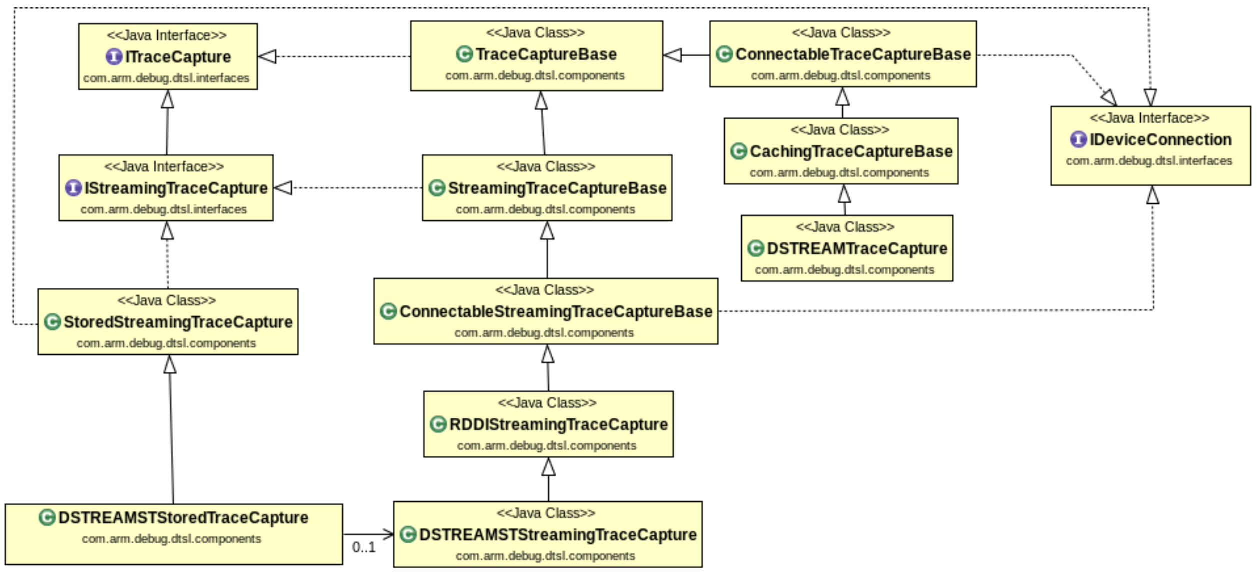
The following image shows the on-chip trace class hierarchy and interfaces:

Figure 14-7: On-chip trace class hierarchy


<<Java Interface>>
ITraceCapture
com.arm.debug.dtsl.interfaces
<<Java Class>>
TraceCaptureBase
com.arm.debug.dtsl.components
<<Java Interface>>
IDeviceConnection
com.arm.debug.dtsl.interfaces
<<Java Class>>
ConnectableTraceCaptureBase
com.arm.debug.dtsl.components
<<Java Class>>
CachingTraceCaptureBase
com.arm.debug.dtsl.components
<<Java Class>>
BBBTraceCapture
com.arm.debug.dtsl.components
<<Java Class>>
ETBTraceCapture
com.arm.debug.dtsl.components
<<Java Class>>
TMCETBTraceCapture
com.arm.debug.dtsl.components
<<Java Class>>
MTBTraceCapture
com.arm.debug.dtsl.components
<<Java Class>>
ETRTraceCapture
com.arm.debug.dtsl.components

The following image shows the off-chip trace class hierarchy and interfaces:

Figure 14-8: Off-chip trace class hierarchy


<<Java Interface>>
ITraceCapture
com.arm.debug.dtsl.interfaces
<<Java Class>>
TraceCaptureBase
com.arm.debug.dtsl.components
<<Java Class>>
ConnectableTraceCaptureBase
com.arm.debug.dtsl.components
<<Java Interface>>
IDeviceConnection
com.arm.debug.dtsl.interfaces
<<Java Class>>
CachingTraceCaptureBase
com.arm.debug.dtsl.components
<<Java Interface>>
IStreamingTraceCapture
com.arm.debug.dtsl.interfaces
<<Java Class>>
StreamingTraceCaptureBase
com.arm.debug.dtsl.components
<<Java Class>>
DSTREAMTraceCapture
com.arm.debug.dtsl.components
<<Java Class>>
StoredStreamingTraceCapture
com.arm.debug.dtsl.components
<<Java Class>>
ConnectableStreamingTraceCaptureBase
com.arm.debug.dtsl.components
<<Java Class>>
RDDIStreamingTraceCapture
com.arm.debug.dtsl.components
DSTREAMSTStoredTraceCapture
com.arm.debug.dtsl.components
0..1
<<Java Class>>
DSTREAMSTStreamingTraceCapture
com.arm.debug.dtsl.components

The following is a typical code sequence from a DTSL Jython script to create an ETB trace capture device:

```
devID = self.findDevice("CSETB")
self.ETB = ETBTraceCapture(self, devID, "ETB")
self.ETB.setFormatterMode(FormatterMode.BYPASS)
```

```
4. self.ETB.addTraceSource(self.ETM, self.coretexA8.getID())
5. self.addTraceCaptureInterface(self.ETB)
6. self.setManagedDevices([self.ETM, self.ETB])
```

Line 1 locates the ETB device ID (number) from the RDDI configuration (`.rcf` file or `.rvc` file). Line 2 creates the `ETBTraceCapture` object with the name 'ETB'. Line 3 configures the formatter mode of the ETB. Line 4 adds an ETM object, such as that created by the code sequence in DTSL trace source objects , to the set of trace sources to associate with the trace capturedevice. This should be done for all trace source objects which deliver trace to the trace capture device. To associate the ETM with a core, the code uses a version of the `addTraceSource()` method which allows it to associate the core by its ID. Line 5 adds the trace capture device to the DTSL configuration. Line 6 tells DTSL to automatically manage connection and disconnection to and from the ETM and ETB devices.

When a client program has a reference to the DTSL configuration object, it can query it for its set of trace capture devices. For each trace capture device, it can find out which trace sources feed into the trace capture device.

14.5.6 Memory as seen by a core device

When a DTSL configuration creates DTSL device objects for Arm cores, target memory can be accessed by performing memory operations on the device objects. This is how Arm® Debugger typically accesses memory during a debug session. However, such memory accesses have certain characteristics and are restricted in certain ways:

- For most Arm cores, memory cannot be accessed through the core when the core is executing.
- For cores with an MMU, the address used to access memory through the memory access methods of a device is the address as seen from the point of view of the core. This means that if the MMU is enabled, then the address is a virtual address, and it undergoes the same address translation as if it had been accessed by an instruction executed by the core. This is usually what a DTSL client, such as a debugger, wants to happen, so that it can present the same view of memory as that which the core sees when executing instructions.
- For cores with enabled caches, the data returned by the memory access methods of a device is the same as would be returned by a memory access by an instruction executed on the core. This means that if the data for the accessed address is currently in a cache, then the cached data value is returned. This value might be different from the value in physical memory. This is usually what a DTSL client, such as a debugger, wants to happen, so that it can present the same view of memory as that which the core sees when executing instructions.

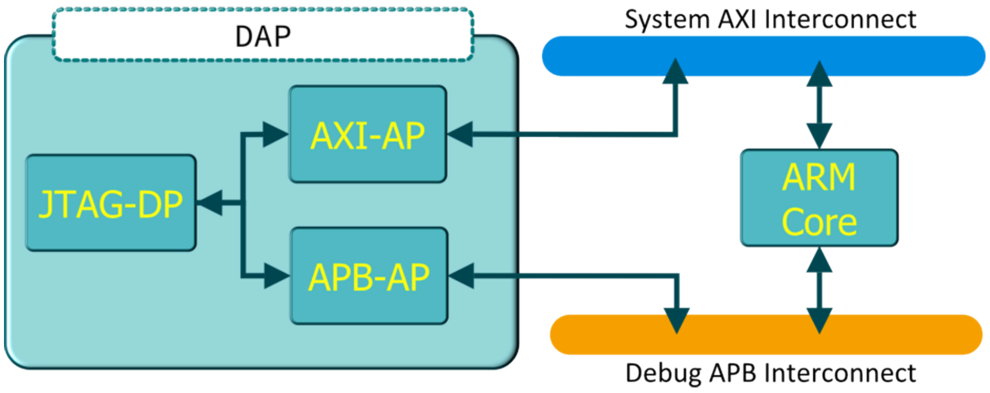
14.5.7 Physical memory access via CoreSight

Although CoreSight™ does not require it, most CoreSight implementations provide a direct way to access the bus or buses of the target system. They do this by providing a Memory Access Port (MEM-AP) which is accessed through the CoreSight DAP. There are several types of MEM-AP depending on the type of the system bus. The three main types are APB-AP, AHB-AP, and AXI-

AP, which provide access to APB, AHB, and AXI bus types respectively. Each of these access ports implements the CoreSight MEM-AP interface.

The following figure shows a simple, but typical, arrangement of MEM-APs:

Figure 14-9: MEM-AP Access Ports



To allow direct memory access through one of the MEM-APs, a DTSL configuration can create device objects for the MEM-APs themselves. When the memory access methods are called on such devices, the memory access is directed straight onto the system bus, completely bypassing the core or cores.

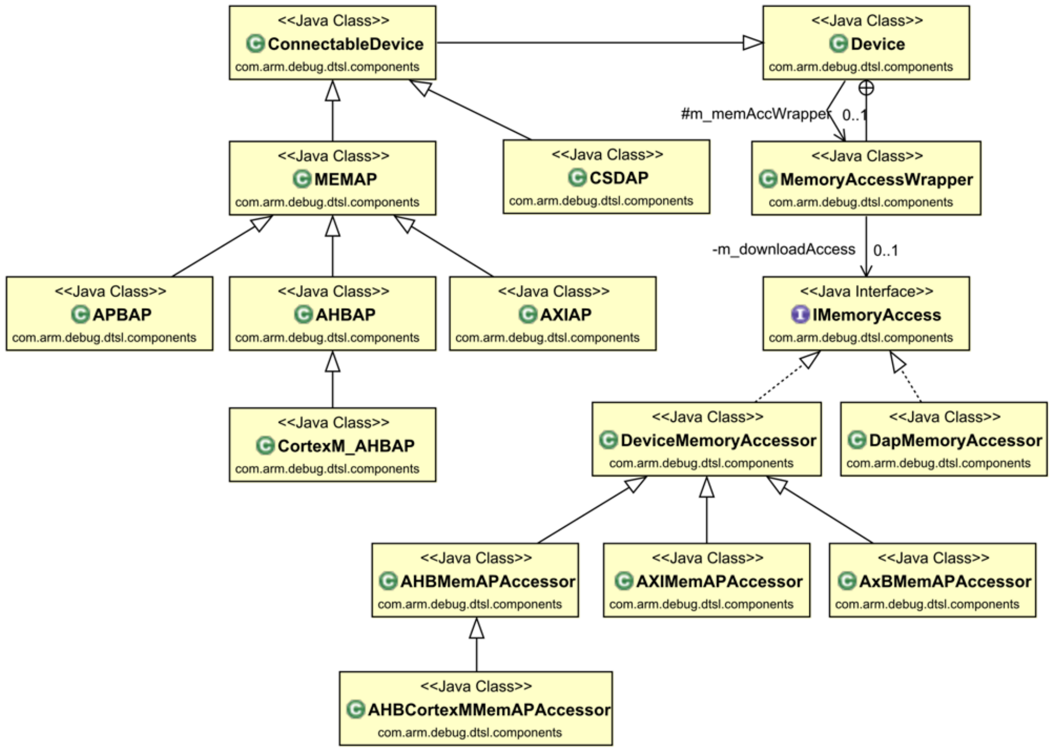
Note

The memory access is not processed by the core MMU (so there is no core MMU address translation), and bypasses any cache in the core, which might result in a different value being observed to that observed by the core.

14.5.8 DTSL MEM-AP support

DTSL provides special classes for MEM-AP support. The following figure shows the class hierarchy:

Figure 14-10: MEM-AP Class Hierarchy



The image shows two main class trees. These are the MEM-AP tree and the `DeviceMemoryAccessor` tree. The DTSL configuration typically creates objects for one or more of the MEM-AP class types, suitably specialized for the actual bus type.

In the MCBSTM32E example, there is an AHB-AP which can be used to access memory directly. In the case of Cortex®-M3, this bus is also used to access the CoreSight™ debug components, but for non-Cortex-M cores it is more typical for there to be a separate APB-AP for debug component access. The significant lines of the DTSL configuration script are similar to the following:

```
devID = self.findDevice("CSMEMAP")
self.AHB = CortexM_AHBAP(self, devID, "CSMEMAP")
```

In this case, the RDDI-DEBUG configuration has a device called CSMEMAP, which associates with a `CortexM_AHBAP` DTSL object. This object is derived from a DTSL `Device`, and so has memory access methods available.

If a client is aware of such DTSL devices, then it can use them to access memory directly.

14.5.9 Linking MEM-AP access to a core device

Not all clients are directly aware of MEM-AP type devices. Arm® Debugger is an example of such a client. To allow such clients to make use of MEM-AP devices, named address space filters can be added to any DTSL Device object. The purpose of the address space filter is to tell the Device object that, if it sees a memory access with a known address space name, it should carry out the access through another DTSL device, rather than through the core. For example, we can add an address space filter to the Cortex®-M3 DTSL Device which detects memory accesses to an address with an address space of 'AHB'. When it detects such an access, it performs the access using the AHB device, instead of going through the Cortex-M3. For Arm Debugger, this means that the user can prefix an address with `AHB:` (for example, `AHB:0x20000000`), and the access is performed using the AHB-AP.

The following code shows how the address space filter is added to the Cortex-M3 object:

```
devID = self.findDevice("CSMEMAP")
self.AHB = CortexM_AHBAP(self, devID, "CSMEMAP")
devID = self.findDevice("Cortex-M3")
self.cortexM3 = ResetHookedDevice(self, devID, "Cortex-M3")
self.cortexM3.registerAddressFilters(
    [AHBCortexMMemAPAccessor("AHB", self.AHB, "AHB bus accessed via AP_0")])
```

Any number of address filters can be added, but each filter name (Arm® Debugger address prefix) must be unique.

To determine the supported address spaces for an object which implements `IDevice`, call the `getAddressSpaces()` method. When a client matches against an address space, it can map the address space to a `rule` parameter which is passed into the `IDevice` memory access methods. The `rule` parameter is then used to direct the memory access to the appropriate device.

14.6 DTSL options

On many platforms, the debug components allow configuration of their properties. For example, in some CoreSight™ PTM components, the generation of timestamps within the trace data stream can be turned on or off. Such options are typically accessed and changed by using the DTSL objects that were created as part of the DTSL configuration. For example, the DTSL `PTMTraceSource` object has a `setTimestampingEnabled()` method to control timestamping. In this way, the DTSL objects that a DTSL configuration holds can expose a set of configuration options that you might want to modify. You might also want to create an initial option set to be applied at platform connection time, and then change some of those options after connecting, during a debug session. For example, this allows the PTM timestamp option to have a user setting applied at connection time, while also allowing you to turn the timestamps on and off during a debug session.

14.6.1 DTSL option classes

To support the concept of DTSL options, a DTSL configuration can expose a set of option objects. These objects allow a client to query the option set and their default values, and to modify the option values before and after connecting. The option objects are arranged hierarchically, and grouped in a way that allows them to be presented in a GUI. The option set must be available before connecting, so the options are exposed by a static method `getOptionList()` on the DTSLv1 derived class within a Jython script.

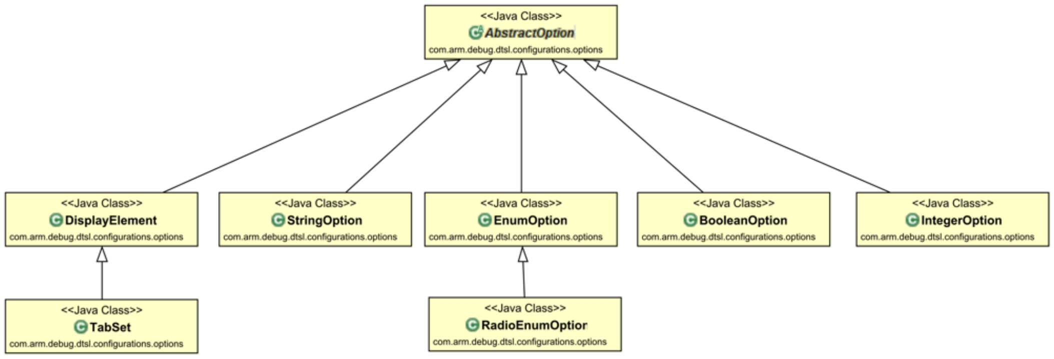
Note

The `getOptionList()` static method is not part of any defined Java interface. The DTSL configuration script manager uses Jython introspection at run time to determine whether the method exists.

The object set returned from `getOptionList()` should be an array of option objects. It is very common to partition the set of options into logical groups, each of which has its own tab page within a `TabSet`. Each tab page contains the options for its associated group.

The following image shows the supported options types and class hierarchy:

Figure 14-11: DTSL Option Classes


<<Java Class>>
AbstractOption
com.arm.debug.dtsl.configurations.options
<<Java Class>>
DisplayElement
com.arm.debug.dtsl.configurations.options
<<Java Class>>
StringOption
com.arm.debug.dtsl.configurations.options
<<Java Class>>
EnumOption
com.arm.debug.dtsl.configurations.options
<<Java Class>>
BooleanOption
com.arm.debug.dtsl.configurations.options
<<Java Class>>
IntegerOption
com.arm.debug.dtsl.configurations.options
<<Java Class>>
TabSet
com.arm.debug.dtsl.configurations.options
<<Java Class>>
RadioEnumOptior
com.arm.debug.dtsl.configurations.options

The DTSLv1 class provides many static helper methods for creating the options, and it is more usual for these methods to be used rather than directly creating the objects.

14.6.2 DTSL option example walk-through

The following is a simplified example from the Keil® MCBSTM32E platform Jython script:

```
    class DSTREAMDebugAndTrace(DTSLv1):
        '''A top level configuration class which supports debug and trace'''

        @staticmethod
        def getOptionList():
            '''The method which specifies the configuration options which
               the user can edit via the launcher panel |Edit...| button
            '''
            return [
                DTSLv1.tabSet(
                    name='options',
                    displayName='Options',
                    childOptions=[
                        DSTREAMDebugAndTrace.getTraceBufferOptionsPage(),
                        DSTREAMDebugAndTrace.getETMOptionsPage(),
16.                         DSTREAMDebugAndTrace.getITMOptionsPage()
17.                     ]
18.                 )
19.             ]
```

Line 4 marks the method as a static method of the containing class. This allows it to be called before an instance of the class exists. It also implies that any methods that are called are also static methods, because there is no self (this) associated with an instance of the class. Line 5 defines the static method with the name `getOptionList`. If this static method is present, then the configuration has options, otherwise it does not. Line 10 creates a `TabSet` object with name `options`, display name 'Options', and an array of child options, which in this example are each created by calling another static method.

Note

You might find it helpful to provide child options using several static methods. This prevents the nesting level of brackets from becoming too deep and difficult to understand, and makes it easier for you to avoid using the wrong type of bracket in the wrong place.

The following code extract shows the `getTraceBufferOptionsPage` method:

```
@staticmethod
def getTraceBufferOptionsPage():
    return DTSLv1.tabPage(
        name='traceBuffer',
        displayName='Trace Buffer',
        childOptions=[
            DTSLv1.enumOption(
                name='traceCaptureDevice',
                displayName='Trace capture method',
                defaultValue='none',
                values=[
                    ('none', 'No trace capture device'),
                    ('DSTREAM', 'DSTREAM 4GB Trace Buffer')
                ]
            ),
            DTSLv1.booleanOption(
                name='clearTraceOnConnect',
                displayName='Clear Trace Buffer on connect',
                defaultValue=True
            ),
            DTSLv1.booleanOption(
                name='startTraceOnConnect',
                displayName='Start Trace Buffer on connect',
                defaultValue=True
            ),
            DTSLv1.enumOption(
                name='traceWrapMode',
                displayName='Trace full action',
                defaultValue='wrap',
                values=[
                    ('wrap', 'Trace wraps on full and continues to store data'),
                    ('stop', 'Trace halts on full')
                ]
            )
        ]
)
```

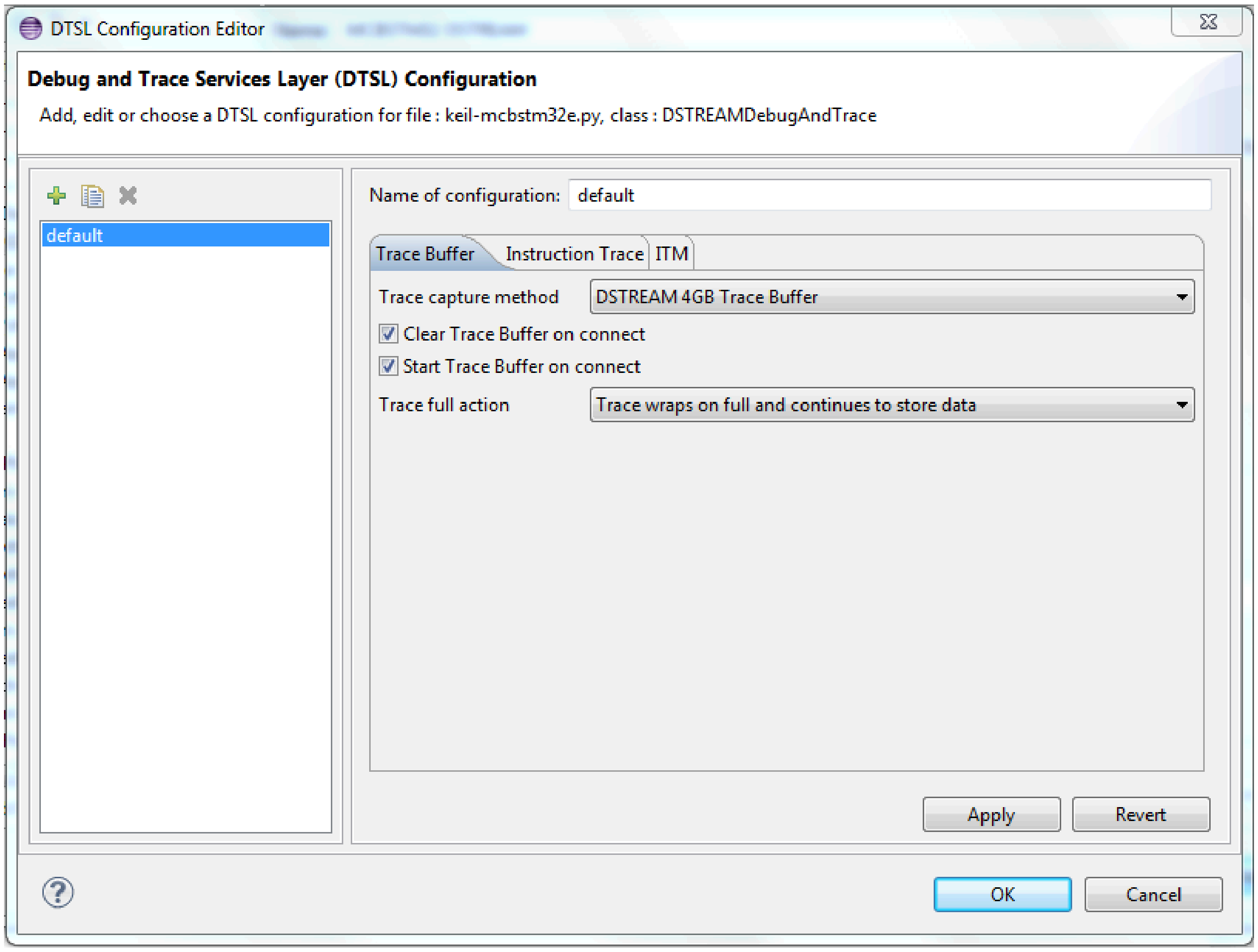
Note

The code uses nesting and indentation to help keep track of closing bracket types.

Line 3 creates a tab page named `traceBuffer`, which has an array of child options. These child options are displayed on the tab page within a GUI. Working through the child options might help you understand how they are displayed to the user. Line 7 creates an enum option. This is an option whose value is one of a set of pre-defined values, and which is typically presented to the user as a drop down list box. The list box shows the pre-defined values, and the user selects one of them. The values are given as pairs of strings. The first string is the internal value, and the second string is the text displayed to the user. Lines 16 to 21 create boolean options. These are options which are true or false, or on or off, and are usually shown to the user as a check box GUI element.

The following image shows how Arm® Development Studio renders the tab set and tab page:

Figure 14-12: DSTREAM Trace Options


DTSL Configuration Editor
Debug and Trace Services Layer (DTSL) Configuration

Add, edit or choose a DTSL configuration for file : keil-mcbstm32e.py, class : DSTREAMDebugAndTrace

Name of configuration: default

default
Trace Buffer
Instruction Trace
ITM

Trace capture method: DSTREAM 4GB Trace Buffer

[✓] Clear Trace Buffer on connect

[✓] Start Trace Buffer on connect

Trace full action: Trace wraps on full and continues to store data

For more examples, see the full source code for the Keil example in the DTSLExampleConfigdb project.

14.6.3 Option names and hierarchy

All options are part of an option hierarchy. Starting at the outermost level, the `TabSet` object is usually named 'options'. All other options are then created in a `childOptions` path, starting from this outermost level. The `name path` for an option consists of all the internal names (not the display names) in the hierarchy between the outermost level and the option in question, joined by the `.` character. For example, in the previous code samples, the option which indicates the currently selected trace capture device has the name path `options.traceBuffer.traceCaptureDevice`. The components of this name path, joined by `.`, are as follows:

options

The internal name of the outermost `TabSet`.

traceBuffer

The internal name of the child option for the trace buffer tab page object.

traceCaptureDevice

The internal name of the `EnumOption` for the currently selected trace capture device.

The full path name is important for at least three reasons:

- It can be used from the Arm® Debugger command line, to read or modify the option value, using the commands `show dtsl-options` or `set dtsl-options`.
- It can be used in the `project_types.xml` file to direct the Arm Debugger to relevant options, such as which trace capture device to use (if any).
- It can be used in the `getOptionValue` and `setOptionValue` methods of the configuration, to read or modify an option's value.

Note

The full path option name is case sensitive.

Here is an example output from the `show dtsl-options` command to see the list of available DTSL options and their current values.

```
Command: show dtsl-options
dtsl-options options.ETM.cortexM3coreTraceEnabled:                      value is
 "true"
dtsl-options options.ITM.itmTraceEnabled:                               value is
 "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner:                      value is
 "Target"
                                                                          (read
 only)
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.DWTENA:            value
 is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[15:8]:   value
 is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[23:16]:  value
 is "true"
```

```
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[31:24]:  value
 is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.PRIVMASK.[7:0]:    value
 is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.STIMENA:     value is
 "0xFFFFFFFF"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.TSENA:             value
 is "true"
dtsl-options options.ITM.itmTraceEnabled.itmowner.debugger.TSPrescale:        value
 is "none"
dtsl-options options.ITM.itmTraceEnabled.itmowner.target.targetITMATBID:value is "2"
                                                                          (read
 only)
dtsl-options options.traceBuffer.clearTraceOnConnect:                   value is
 "true"
                                                                          (read
 only)
dtsl-options options.traceBuffer.startTraceOnConnect:                   value is
 "true"
                                                                          (read
 only)
dtsl-options options.traceBuffer.traceCaptureDevice:                    value is
 "DSTREAM"
                                                                          (read
 only)
dtsl-options options.traceBuffer.traceWrapMode:                         value is
 "wrap"
                                                                          (read
 only)
dtsl-options options.ucProbe.ucProbeEnabled:                            value is
 "false"
dtsl-options options.ucProbe.ucProbeEnabled.PORT:                       value is
 "9930"
                                                                          (read
 only)
```

Here is an example of the `set dtsl-options` command to change the current value of any non read-only option:

```
Command: set dtsl-options options.ITM.itmTraceEnabled false
DTSL Configuration Option "options.ITM.itmTraceEnabled" set to false
```

14.6.4 Dynamic options

Some option values can be modified dynamically, after connecting to the platform. For an Arm® Development Studio Debug session, this means the option can be changed during the debug session, using either the Arm Debugger command line or the **DTSL Options...** menu selection with the Debug Control View.

Not all options can be modified after connecting. For example, the trace capture device cannot typically change during the debug session, although the option to enable ITM trace can change. Even if an option can be changed, it might not apply the change immediately. For example, most trace-related dynamic options apply changes only when tracing is started or restarted.

To mark an option as dynamic, add the `isDynamic=True` parameter to the option constructor. For example, the ITM option to generate timestamps could be created as follows:

```
DTSLv1.booleanOption(
     name='TSENA',
     displayName = 'Enable differential timestamps',
     defaultValue=True,
     isDynamic=True
)
```

When Arm Debugger displays the options during a debug session, it only allows the dynamic options to be changed. All the options are shown to the user, but the non-dynamic ones are grayed out and cannot be changed.

14.6.5 Option change notification

Shortly after the DTSL configuration instance (the object derived from DTSLv1) is created, the option values are given to the instance by calling its `optionValuesChanged` method. This method inspects the current option values and configures the platform components accordingly.

Note

The `optionValuesChanged` method is called after the constructor is called, but before the DTSL components are connected to the target platform. This means that the DTSL objects can be configured with their settings, but cannot send the settings to the target components.

If the options are changed during a debug session, then the `optionValuesChanged` method is called again, to inform the DTSL components that the options have changed.

Note

Currently, the call to the `optionValuesChanged` method does not indicate which options have changed. A future version of DTSL will address this.

14.6.6 Option change notification example walk-through

These Jython code snippets are from the Keil® MCBSTM32E platform Jython script and the `DSTREAMDebugAndTrace` class:

```
def optionValuesChanged(self):
    '''Callback to update the configuration state after options are changed.
        This will be called:
           * after construction but before device connection
           * during a debug session should the user change the DTSL options
     '''
     obase = "options"
     if self.isConnected():
         self.updateDynamicOptions(obase)
     else:
         self.setInitialOptions(obase)
```

Line 1 declares the `optionValuesChanged` method, which is called to tell the DTSL components that the options have changed. Line 7 assigns the top level options name path value. Lines 8 to 11 call one of two methods depending on whether the configuration is connected yet.

```
def setInitialOptions(self, obase):
   '''Takes the configuration options and configures the
       DTSL objects prior to target connection
       Param: obase the option path string to top level options
    '''
    if self.traceDeviceIsDSTREAM(obase):
        self.setDSTREAMTraceEnabled(True)
        self.setDSTREAMOptions(obase+".traceBuffer")
        obaseETM = obase+".ETM"
        obaseITM = obase+".ITM"
        self.setETMEnabled(self.getOptionValue(
                             obaseETM+".cortexM3coreTraceEnabled"))
        self.reservedATBIDs = {}
        self.setITMEnabled(self.getOptionValue(obaseITM+".itmTraceEnabled"))
        obaseITMOwner = obaseITM+".itmTraceEnabled.itmowner"
        if self.debuggerOwnsITM(obaseITMOwner):
            self.setITMOwnedByDebugger(True);
            self.setITMOptions(obaseITMOwner+".debugger")
        else:
            self.setITMOwnedByDebugger(False);
            self.reservedATBIDs["ITM"] =
            self.getOptionValue(obaseITMOwner+".target.targetITMATBID")
        self.updateATBIDAssignments()
    else:
        self.setDSTREAMTraceEnabled(False)
        self.setETMEnabled(False)
        self.setITMEnabled(False)
```

In this code example, note the following:

- The value for an option is retrieved using the `self.getOptionValue` method, which takes the full option path name to the option value.
- The code builds up the full option path names, which allows the options to be moved more easily. This can be seen in the way that the `obaseITMOwner` value is constructed and passed to the `self.setITMOptions` method. This allows `self.setITMOptions` to be written without having to hard code the full option name path into it. Instead, it only needs to know the path extensions from the passed base to determine its option values.

For completeness, the following shows the dynamic option update method:

```
def updateDynamicOptions(self, obase):
  '''Takes any changes to the dynamic options and
       applies them. Note that some trace options may
       not take effect until trace is (re)started
       Param: obase the option path string to top level options
    '''
    if self.traceDeviceIsDSTREAM(obase):
        obaseETM = obase+".ETM"
        self.setETMEnabled(self.getOptionValue(
            obaseETM+".cortexM3coreTraceEnabled"))
        obaseITM = obase+".ITM"
        if self.getOptionValue(obaseITM+".itmTraceEnabled"):
            self.setITMEnabled(True)
            obaseITMOwner = obaseITM+".itmTraceEnabled.itmowner"
            if self.debuggerOwnsITM(obaseITMOwner):
                self.setITMOptions(obaseITMOwner+".debugger")
        else:
            self.setITMEnabled(False)
```

For the dynamic option changes, only the options marked as dynamic need inspecting.

Note

The option values are passed on to the corresponding DTSL objects, but the option changes might not be applied immediately. In many cases, the change only applies when execution or trace is next started. Whether the option change is applied immediately is determined by the implementation of the DTSL object.

14.7 DTSL support for SMP and AMP configurations

From the point of view of Arm® Debugger, Symmetric Multi Processing (SMP) refers to a set of architecturally identical cores which are tightly coupled together and used as a single multi-core execution block. From the point of view of the debugger, they must be started and halted together.

In larger systems, there may be several SMP sets, each of which is referred to as a cluster. Typically, a cluster is a set of 4 cores in an SMP configuration. All cores in the SMP cluster also have the same view of memory and run the same image.

From the point of view of Arm Debugger, Asymmetric Multi Processing (AMP) refers to a set of cores which are operating in an uncoupled manner. The cores can be of different architectures {Cortex®-A8, Cortex-M3}, or of the same architecture but not operating in an SMP configuration. From the point of view of the debugger, it depends on the application whether the cores need to be started or halted together.

From the point of view of DTSL, the cores in the set (SMP or AMP) are part of the same configdb platform. Using `project_types.xml`, the platform exposes a set of debug operations which cover the supported use cases. All of these use cases must be provided for by the same Jython DTSL configuration class. This is because, although there can be multiple clients using DTSL (for example, one debugger controlling a Cortex-A8 and another controlling a Cortex-M3), there is only one set of target debug hardware (for example, only one TPIU). There must therefore be a single DTSL instance in control of the debug hardware.

In SMP systems, there is usually a hardware mechanism which keeps the set of cores at the same execution state. Some AMP systems must also have synchronized execution state, and the multi-client, single DTSL instance architecture supports this. The single DTSL instance is always aware of the target execution state, and can typically arrange for a single execution state between all AMP cores.

14.7.1 AMP systems and synchronized execution

If a platform contains multiple cores, then when the first DTSL client connects, the DTSL configuration creates devices for all of the cores. The client uses the devices for the cores it wants to control. When a second client connects to the same platform, it must present an identical set of connection parameters. The DTSL connection manager therefore returns the same DTSL configuration instance that was created by the first client connection. The second client can use the devices for the cores it wants to control. In this way, two clients can use the same DTSL configuration instance, including any DTSL options.

If execution synchronization is not required, a simple DTSL configuration is enough, with core execution state being independent. However, if synchronized execution state is required, then the created object model must provide this. The execution synchronization can be implemented with features in the hardware, or by creating a software object model hierarchy which arranges for a shared execution state.

14.7.2 Execution synchronization levels

The level at which DTSL can perform synchronized execution status depends heavily on both the execution controller (for example, the JTAG control box) and the on-chip debug hardware. However, there are roughly three different levels (or qualities) of synchronization:

- Software synchronization
- Tight synchronization
- Hardware synchronization

14.7.3 Software synchronization

This is the lowest level or quality of synchronization. 'Software' refers to the DTSL software running on the host debugger computer. At this level, the synchronization is of the following form:

Synchronized start

This is achieved by asking each device to start executing, by calling the `go()` method on each device in turn.

Synchronized stop

This is achieved by asking each device to stop executing, by calling the `stop()` method on each device in turn. If one device is seen to be stopped (by DTSL receiving a `RDDI_EVENT_TYPE.RDDI_PROC_STATE_STOPPED` stopped event), then the DTSL configuration must request all other devices to stop.

This synchronization is done on the host computer, so there can be hundreds of milliseconds between each core actually stopping. Whether this is a problem depends on how the target application handles other cores not responding (if they communicate with each other at all).

14.7.4 Tight synchronization

With tight synchronization, the execution controller (JTAG box such as DSTREAM) can manage the synchronization. This can typically be further divided into several sub-levels of support:

- The execution controller supports the RDDI `Debug_Synchronize()` call. In this case, the synchronized start and stop functionality is implemented in the execution controller. The controller is much 'closer' to the target system, so it can typically synchronize down to sub-millisecond intervals.
- The execution controller can define one or more sets of cores which form a cluster. When any one of the cores in a set is seen to halt execution, the others are automatically halted. This typically provides synchronized stop down to a sub-millisecond interval. DSTREAM supports this technique.
- The execution controller supports `Debug_Synchronize()`, but cannot define clusters. In this case, the DTSL configuration must be written so that if it sees any core in a synchronized group as halted, it issues the RDDI `Debug_Synchronize()` call to halt the others in the group. In a group of several devices, the time interval between the first halting and the others halting may be hundreds of milliseconds, but the interval between the others halting is typically sub-millisecond.

14.7.5 Hardware synchronization

With hardware synchronization, the target provides synchronization features on-chip. This is typically the case for Arm® CoreSight™ systems that use the Cross Trigger Interface (CTI) to propagate debug halt and go requests between cores. This ability relies on the hardware design implementing this feature, and so might not be available on all CoreSight designs.

14.7.6 SMP states

For SMP systems, DTSL presents a single device to the client (see `SMPDevice` and its relations in the DTSL Java docs), and the client controls execution state through this device. This `SMPDevice` is a 'front' for the set of real devices which form an SMP group. When the `SMPDevice` reports the execution state to the client, there is the possibility of inconsistent states. Ideally, for an SMP group, all the cores have the same state, either executing or halted. In practice, this might not be the case. To allow for this possibility, the `SMPDevice` can report an inconsistent state to the client (debugger). This represents the case when not all cores are in the same state. Normally, DTSL provides a time window within which it expects all cores to get into the same state. If all cores become consistent within this time window, then DTSL reports a consistent state to the client, otherwise it reports an inconsistent state. This allows the client to reflect the true state of the system to the user, but still allows the state to be reported as consistent if consistency is achieved at some future time.

14.7.7 Use of CTI for SMP execution synchronization

Cross Trigger Interface (CTI) is part of the Arm Embedded Cross Trigger (ECT) system. Each component in the system can have a CTI which allows inputs and outputs to be routed (or channeled) between the components. The channeling is done by the CrossTrigger Matrix (CTM), which is part of the ECT. The CTM supports a fixed number of channels onto which the CTIs can output or input signals. There might be many signals in the system which can be routed between components, and the CTIs can be told which signals to route by assigning them to a channel.

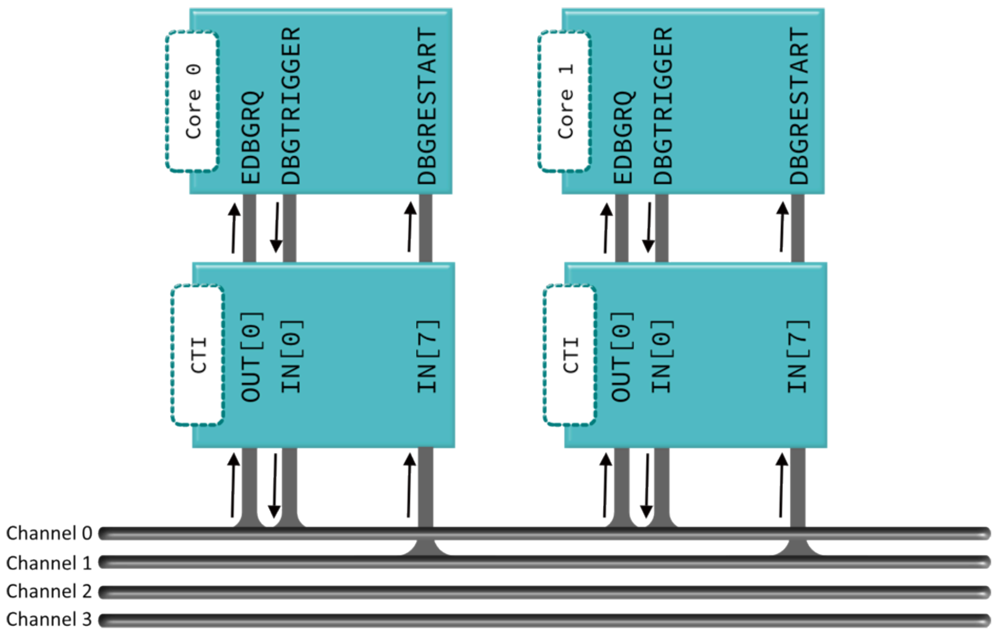
For example, in many systems, each core in the SMP group has a CTI connected to the following signals:

Table 14-1: CTI Signal Connections

Name	Direction	Purpose
DBGTRIGGER	Output from core to CTI	Indicates that the core is going to enter debug state (is going to stop executing)
EDBGRQ	Input to core from CTI	An external request for the core to enter debug state (stop executing)
DBGRESTART	Input to core from CTI	An external request for the core to exit debug state (start executing)

For synchronized execution to work, the DTSL configuration assigns two channels, one of which is for stop requests and the other of which is for start requests. The CTI or CTIs are configured to connect the above signals onto these channels.

Figure 14-13: Example use of CTI for H/W execution synchronization



With this configuration:

- When a debug tool wants to halt all cores, it sends a CTI pulse. Sending a pulse from any CTI onto the stop channel sends a EDBGRQ to all cores, which causes them to halt. This provides the synchronized stop functionality for stop requests instigated by the debug tool.
- When any core halts (hits a breakpoint), the DBGTRIGGER signal outputs onto the stop channel and sends a EDBGRQ signal to all the other cores, which causes them to halt. This provides the synchronized stop functionality for breakpoints and watchpoints, for example.
- When all cores are ready to restart, sending a pulse from any CTI onto the start channel sends a DBGRESTART signal to all cores. This provides the synchronized start functionality.

The convention for DTSL configurations is that channel 0 is used for the stop channel and channel 1 is used for the start channel. DTSL configuration scripts usually allow this to be modified by changing the following constants, which are assigned near the top of the configuration script:

```
CTM_CHANNEL_SYNC_STOP = 0 # use channel 0 for sync stop
CTM_CHANNEL_SYNC_START = 1 # use channel 1 for sync start
```

14.8 DTSL Trace

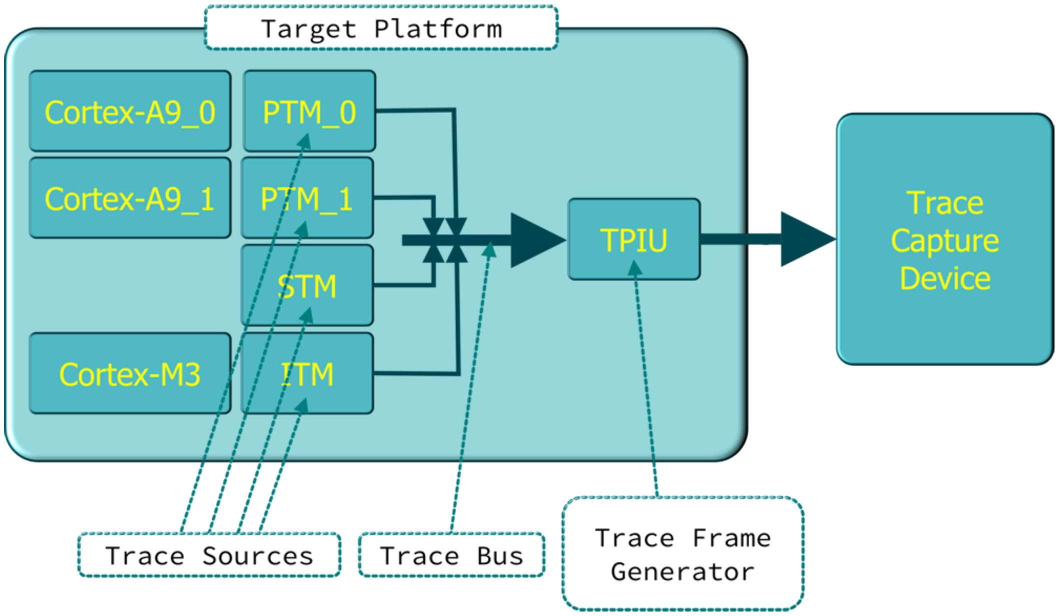
DTSL is designed to support many different trace capture devices, such as DSTREAM, ETB, TMC/ETB and TMC/ETR. It is also possible to extend DTSL to support other trace capture devices. Each of these capture devices can present its data to DTSL in a different format.

Within a platform, trace data can originate from several trace sources. This data is mixed together into the data stream which the trace capture device collects. For simplicity, trace clients (software packages which receive or read trace data from DTSL) are usually designed based on the assumption that the only trace data they receive from the trace source is data which they know how to decode. For example, if a trace client knows how to decode PTM data, then it only expects to receive PTM data when it reads trace data from DTSL.

14.8.1 Platform trace generation

The following figure shows a simplified diagram of trace generation within a platform. There are several trace sources outputting trace data onto a trace bus. The bus takes the data through a frame generator and outputs it to a trace capture device.

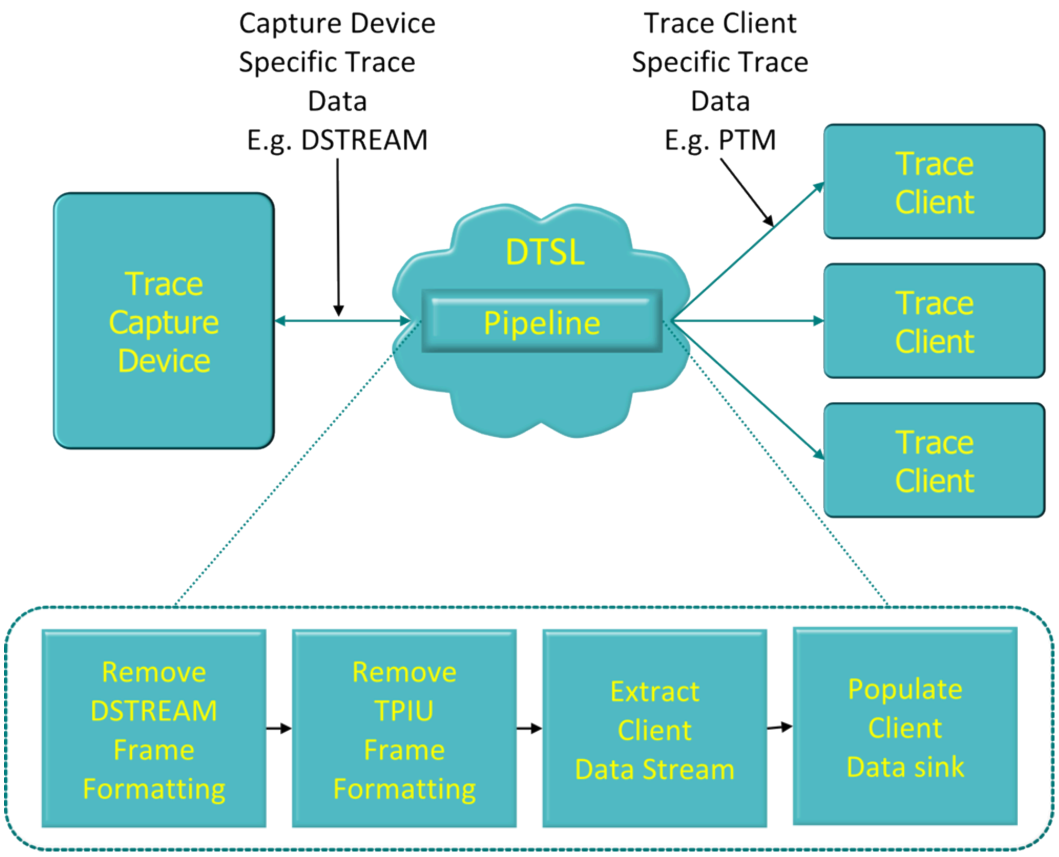
Figure 14-14: Trace Generation



14.8.2 DTSL trace decoding

To process the raw trace data from the trace capture device into a format which is suitable for trace clients to consume, DTSL pushes the raw trace data through a pipeline of trace decoders. The following image shows an example of this flow for DSTREAM trace data:

Figure 14-15: DTSL Trace Decoding Stages for DSTREAM


Capture Device Specific Trace Data E.g. DSTREAM
Trace Client Specific Trace Data E.g. PTM
Trace Capture Device
DTSL
Pipeline
Trace Client
Trace Client
Trace Client
Remove DSTREAM Frame Formatting
Remove TPIU Frame Formatting
Extract Client Data Stream
Populate Client Data sink

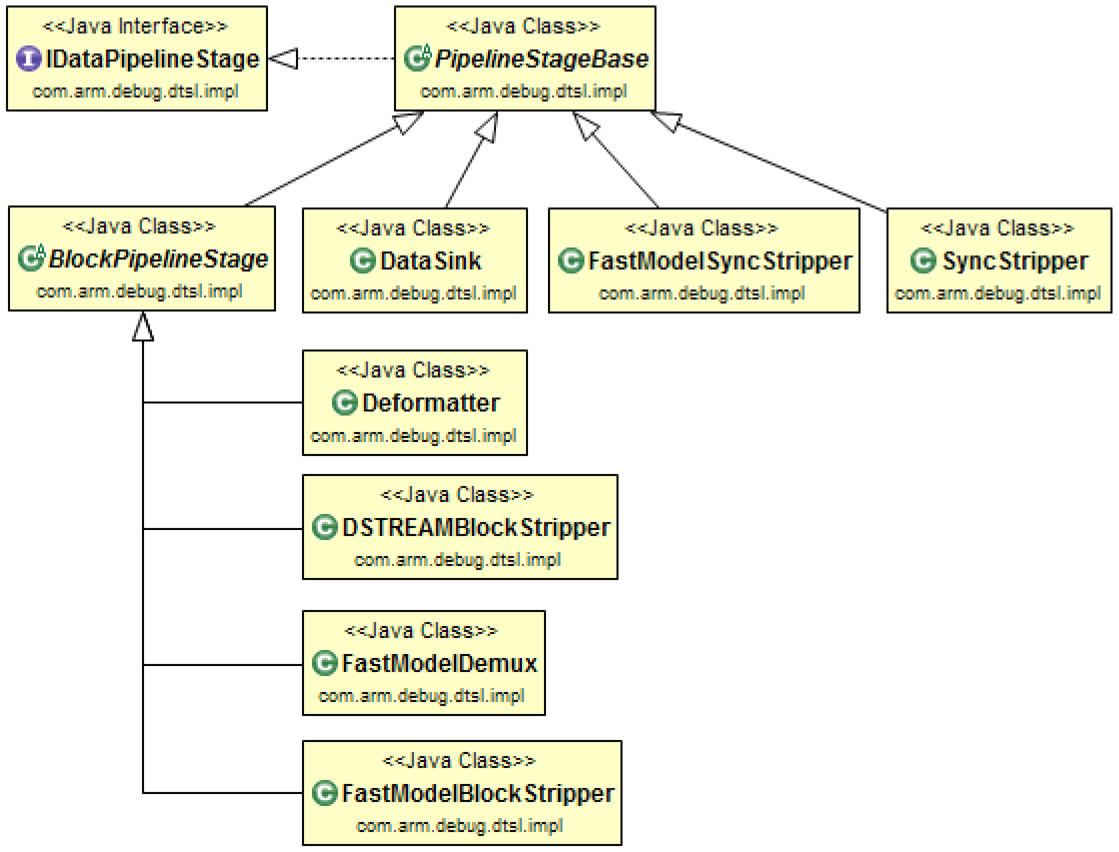
The number and type of the pipeline decoding blocks depends on the format of the trace capture device and the trace data format. However, the final stage should always be to place client compatible data (raw trace source data) into a `DataSink`, ready for the trace client to consume.

14.8.3 DTSL decoding stages

The minimal pipeline decoder does nothing to the data from the trace capture device, except to write it into a `DataSink` for the trace client to read. You can use this pipeline when you know that the data from the trace capture device is already in the format required by a trace client. For example, if you have a disk file which contains raw PTM trace data (that is, the data exactly as output from the PTM onto the system ATB, which you might have captured from a simulation), then you can create a PTM-file-based trace capture device. The decoding pipeline would contain only a `DataSink`, into which you would write the content of the file. The PTM trace client could then read the PTM data directly from the `DataSink`.

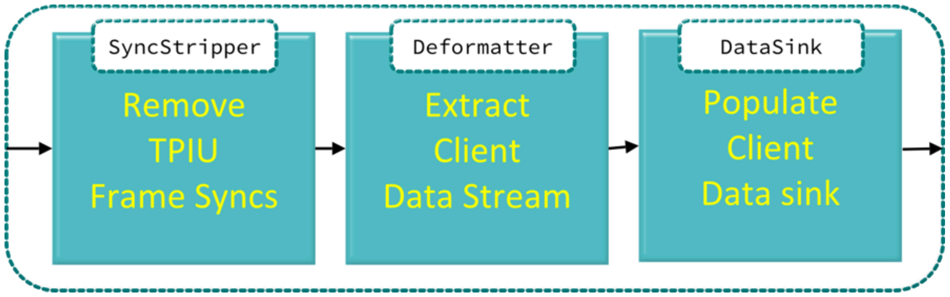
For less straightforward pipelines, a chain of objects must be constructed, each of which must implement the `IDataPipelineStage` interface.

Figure 14-16: DTSL Trace Pipeline Hierarchy



For Arm-based trace systems which use a TPIU Formatter (CoreSight™ ETB, TMC/ETB and TMC/ETR), two further pipeline stages must be added. These are the `SyncStripper` and `Deformatter` stages, which remove the TPIU sync frames and extract data for a particular trace source ID respectively.

Figure 14-17: ETB Trace Decode Pipeline Stages



By implementing new pipeline stages, it is possible to provide trace support for any trace capture device, as long as the final output stage can be reduced to data which is compatible with the expectations of a trace client.

14.8.4 DTSL trace client read interface

Before a trace client can read trace data, it must get a `TraceSourceReader` object from the trace capture device. In practice, this means querying the DTSL configuration for the correct trace capture device, several of which might be available within a configuration, and calling the `borrowSourceReader()` method to get an appropriate `TraceSourceReader`. Trace data can then be retrieved from the `TraceSourceReader`. When it finishes reading trace data, the client must then return the `TraceSourceReader` object to the trace capture device. This is so that the trace capture device knows when there are no clients reading trace, and therefore when it is free to start, or restart, trace collection.

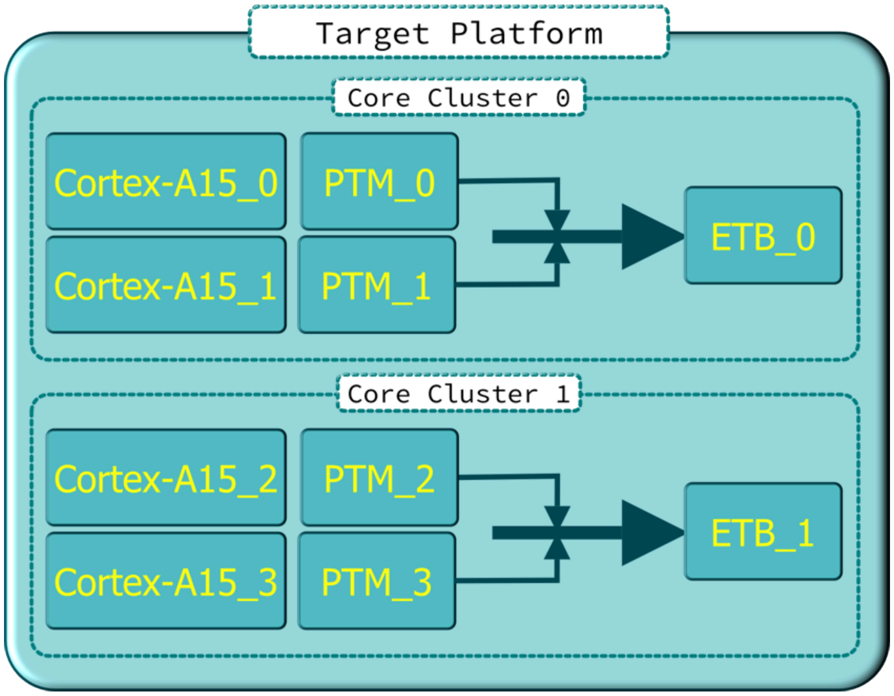
14.8.5 Supporting multiple trace capture devices

A DTSL configuration can contain several trace capture devices. The following are some possible reasons for this:

- The target platform contains several CoreSight™ ETB components.
- The target platform can output to an external DSTREAM device, in addition to an internal CoreSight ETB.

In some cases, there can only be one active trace capture device. In this case, you can choose whether to use the DSTREAM or the ETB. In other cases, there can be several trace capture devices active at the same time. This is common when the platform contains multiple clusters of cores, each of which outputs trace to its own CoreSight ETB.

Figure 14-18: Example of Multiple Trace Capture Devices



To allow trace clients to receive trace data from a trace source, the DTSL configuration can be told about the association between a trace source and one or more trace capture devices. In the figure, for example, the trace source named PTM_2 is associated with the trace capture device named ETB_1. If a client wants to display the trace data for PTM_2, it can ask DTSL which trace capture devices are associated with that trace source, and direct its trace read requests to the correct trace capture device.

Note

It is possible for a trace source to be associated with multiple trace capture devices, such as an internal ETB and an external DSTREAM. In such cases, you might need to provide more information to the client about which trace capture device to use when reading trace data.

14.8.6 Decoding STM STPv2 output

The Arm STM CoreSight™ component generates a STPv2 compatible data stream as its output. The STPv2 specification is a MIPI Alliance specification which is not publicly available. To allow clients to consume STM output, DTSL has a built-in decoder which turns STPv2 into an STM object stream.

To consume STM output, a client should do the following:

- Create an object which implements the `ISTMSourceMatcher` interface. This object tells the decoder which STM master IDs and channel IDs to decode. The STM package includes three implementations of the `ISTMSourceMatcher` interface. These are `STMSourceMatcherRange`, STMSourceMatcherSet, and STMSourceMatcherSingle. If none of these implementations covers your needs, you can also create a new class which implements the ISTMSourceMatcher interface.

- Create an STMChannelReader object, specifying the trace capture device object and the source matcher object.
- Create an object which implements the ISTMObjectReceiver interface, to receive the STM objects.
- When trace data is available, get hold of an ISourceReader object. Pass this, along with the ISTMObjectReceiver object, to the read method on the STMChannelReader object. The read method decodes the trace into an STM object stream, and passes these objects to the ISTMObjectReceiver.

Related information

DTSL trace client read interface on page 551

14.8.7 Example STM reading code

The following is some Java code which shows an example of STM Object reading. This could also be implemented in Jython.

In this case, the STMTraceReader object implements the ISTMObjectReceiver interface itself. This means that the code can pass the object to the STMChannelReader read method as the class to receive the STMObject.

The example code creates an STMSourceMatcherRange object with a parameter set which matches against all Masters and Channels.

The STPv2 packet protocol outputs a SYNC packet which allows the decoder to synchronize the binary data stream to the start of a packet. When decoding arbitrary data streams, the decoder needs to synchronize to the stream before starting to decode the STPv2 packets. Once the stream is synchronized, there is no need to resynchronize, as long as contiguous data is being decoded.

The decoder has two ways to handle errors in the STPv2 packets stream:

- Throw an STMDecodeException or an STMParseException. The advantage of this method is that the errors are detected immediately, but the disadvantage is that you cannot continue processing STPv2 packets (there is no way to resume decoding after the last error position).
- Insert an STMDecodeError object into the generated STMObject set. The advantage of this method is that the full data stream is decoded, but the disadvantage is that the error is not processed by the client until the generated STMDecodeError is processed.

```
/**
 * Class to read STM trace data and to get it processed into a
 * text stream.
 */
public class STMTraceReader implements ISTMObjectReceiver {
     /**
      * The trace device - ETB or DSTREAM or .....
      */
     private ITraceCapture traceDevice;
```

```
    /**
     * A list of STMObjects that gets generated for us
     */
    private List<STMObject> stmObjects;
[snip - other attribute declarations]
    public void decodeSTMTrace() {
        STMSourceMatcherRange stmSourceMatcher = new STMSourceMatcherRange(0, 128,
 0, 65535);
        STMChannelReader stmChannelReader = new STMChannelReader(
            "STM Example",
            this.traceDevice,
            stmSourceMatcher);
        ISourceReader reader = this.traceDevice.borrowSourceReader(
            "STM Reader", this.stmStreamID);
        if (reader != null) {
            try {
[snip - code to figure out if trace is contiguous from last read.]
                if (!traceDataIsContiguous) {
                    stmChannelReader.reSync();
                }
[snip - code to figure out how much trace to read and from where. Also assign values
 to nextPos[] and readSize.]
                this.stmObjects.clear();
                try {
                    stmChannelReader.read(nextPos[0], readSize, this, nextPos,
 reader);
                } catch (DTSLException e) {
                    System.out.println("Caught DTSLException during STPv2 decode:");
                    System.out.println(e.getLocalizedMessage());
                    stmChannelReader.reSync();
                }
            }
            catch (DTSLException e) {
                System.out.println("DTSLException:");
                e.printStackTrace();
            }
            finally {
                /* Must return the trace reader to DTSL so that it knows we have
 finished reading
                 */
                this.traceDevice.returnSourceReader(reader);
            }
        }
    }
    /* (non-Javadoc)
     * @see
 com.arm.debug.dtsl.decoders.stm.stmobjects.ISTMObjectReceiver#write(com.arm.debug.dtsl.decoders.
     */
    @Override
    public boolean write(STMObject stmObject) {
        this.stmObjects.add(stmObject);
        return true;
    }
```

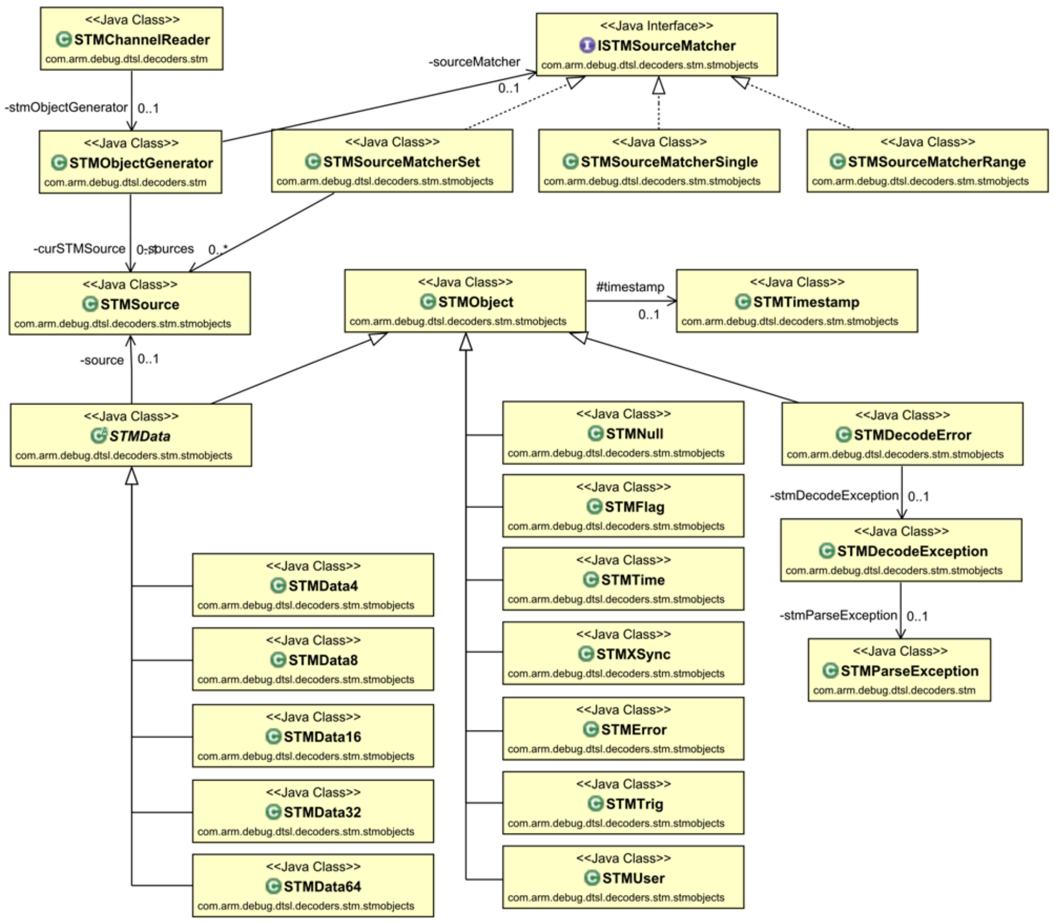
14.8.8 STM objects

The following figure shows the STM object model. All objects generated by the decoder are derived from `STMObject`. All `STMObject` s can contain a timestamp (`STMTimestamp`) if one was generated, otherwise the timestamp attribute is `null`.

The most common form of object generated is the `STMData` objects, which can hold multiple 4-bit, 8-bit, 16-bit, 32-bit, or 64-bit data payloads. Each data packet can also have a marker attribute, in which case it holds only one data payload.

Not all STM object types can be generated from an Arm STM component. {`STMTime`, `STMXSync`, `STMTrig`, `STMUser`} are not generated in Arm STM output.

Figure 14-19: STM Object Model



14.8.9 DTSL client time-stamp synchronization support

Some trace sources have the concept of time-stamping certain trace records or events. If all trace sources are using a common, synchronized time system, then is possible to synchronize all the client trace displays about the same real-time location. To support this, DTSL allows trace clients to request view synchronization at a particular time value. When DTSL receives such a request, it passes it on to all registered trace clients. The trace clients can receive the request and reposition their displays to show data at or around the requested time.

For a client to use the time-stamp synchronization facility, it must register an observer with the DTSL configuration. An observer is an object which implements the `ITraceSyncObserver` interface. See `ConfigurationBase.registerTraceSyncObserver` for details of how to register an observer. If, after registering an observer, the trace client requests time-stamp synchronization, then the observer receives an object. This object implements either the `ITraceSyncEvent` interface or the `ITraceSyncDetailedEvent` interface. The `ITraceSyncEvent` interface only allows reading the requested time value. The `ITraceSyncDetailedEvent` interface, however, extends this, by identifying the trace capture device and buffer location which contained the time position from the point of view of the requesting client. This might be useful to the receiving client as a hint to where they can start searching for the time value in their own trace stream.

If a client wants to request other clients to synchronize to a time value, it must use one of the `ConfigurationBase.reportTraceSyncEvent` methods.

14.9 Embedded Logic Analyzer (ELA)

This topic provides an overview the Embedded Logic Analyzer in the context of Arm® Development Studio, and explains how to use it to capture trace data in Arm Development Studio.

Introduction to the ELA

The ELA is a component of CoreSight™ which provides low-level signal visibility into Arm IP and third party IP.

The ELA allows you to monitor any signal that is part of an implementation-defined Signal Group. This can include loads, stores, speculative fetches, cache activity, and transaction life cycles. You can have multiple ELAs that monitor the various components of your SoC, providing comprehensive insight into the workings of your system. In Arm Development Studio, you can configure the ELA to trigger signal capture in response to an event, or you can configure it to cause triggers elsewhere in your SoC to further your debug process.

Instruction tracing and the ELA offer similar functionality, however the way they work is different. Instruction tracing offers a wide and shallow view of all the instructions that are executed, whereas the ELA offers a narrow yet detailed view of the signals that it detects.

In terms of usage. you might want to use instruction tracing when there is a problem and you are not sure of what is causing it, so you need to collect a large amount of data first. You might want to use the ELA when you have a good guess as to what is causing a problem, and you want to monitor specific signals in your IP.

There are two versions; ELA-500 and ELA-600.

Differences between ELA-500 and ELA-600

The ELA-600 features are a superset of the ELA-500 features. There are more Trigger States, and in addition to SRAM, it also has an ATB trace interface. This means that, at hardware design time, you have the option to either collect data on the ELA itself, or you can push it to your computer. The table below highlights some of the key features of the ELA-500 and ELA-600. For a full list of the features, see the documentation for each product.

Table 14-2: Comparison of features between ELA-500 and ELA-600

Feature	ELA-500	ELA-600
Trigger states	5	8
Embedded RAM configuration	Yes	Yes

Feature	ELA-500	ELA-600
Data compression	No	Yes
ATB interface	No	Yes

Requirements for use

If you are using the ELA-500, or your ELA-600 is configured to capture trace data using the SRAM, check that you have the following:

- A platform configuration that:
  - Lists the relevant ELA trace components.
  - If applicable, lists the CTIs and their connections.
- A JSON file that contains a list of the components of your IP and their corresponding signal group connections. This file is available from the IP designer.

If your ELA-600 is configured to capture trace data using the ATB interface, check that you have the following:

- A platform configuration that:
  - Lists the relevant ELA trace components.
  - Lists the component connections and their mapping between the ELA and its trace sink.
  - If applicable, lists the CTIs and their connections.
- In the DTSL configuration view, you need to enable ELA trace, and setup and enable the trace source and sink.
- A JSON file that contains a list of the components of your IP and their corresponding signal group connections. This file is available from the IP designer.

ELA Scripts

Arm provides several python scripts with your Arm Development Studio installation. These allow you to use and configure the ELA.

There are different scripts for ELA-500 and ELA-600:

Table 14-3: ELA Scripts

Script name	Usage in ELA-500	Usage in ELA-600
ela_control.py	Use this script to: • Run and stop the ELA • Print a summary of the ELA status registers. • Read trace data from the ELA buffer. • [ELA-600 only] Trace until the core stops.	

Script name	Usage in ELA-500	Usage in ELA-600
ela_example.py	This script provides some usecase examples that you can use in your own implementation. These are: • Periodic trace. • Configure the ELA using signal groups. • Decode the trace data. This script works with the `example_ela_connections.json` file, and shows how the signal group descriptions provided in the JSON file are used for configuration and decoding registers.	-
ela_lowlevel.py	Use this script to configure the trigger state registers and the control registers.	-
ela_test.py	Use this script to generate some random trace data in the ELA buffer for testing purposes.	-
process_trace.py	-	Use this script to: • Decompress and decode trace data and decode trace data from the buffer. • Decompress and decode trace data and decode trace data from a file. • Dump trace data from the buffer into a file.
ela_setup.py	-	Use this script to: • Configure the trigger state registers and the control registers. • Run a Periodic trace. You can use this example function to check that your ELA-600 is working properly with Arm Development Studio.

Related information

Using the ELA-500 on page 558
Using the ELA-600 on page 561
Use case scripts on page 140
ELA-500 Product page
ELA-600 Product page
Using the CoreSight ELA-500 Embedded Logic Analyzer with Arm DS-5 tutorial

14.10 Using the ELA-500

Start and stop a capture using the ELA-500, and decode the captured data. There is also an example of the ELA-500 output, both before and after decoding.

Before you begin

For the ELA to correctly decode your captured data, you need:

- A JSON file that lists all the components of your SoC, and their addresses.
- To save the JSON file in the same location as your scripts.
- To update the JSON file name in `ela_example.py`.

Procedure

The designer of your target provides the JSON file. Alternatively you can use the provided `example_ela_connections.json` file as a starting point, and manually add the information for your target. You can find the information you need, such as the component IDs and their addresses, in the ConfigDB entry for your target.

14.10.1 Configure the ELA-500

Import the scripts and configure the ELA-500 for use with Arm® Development Studio. This guide provides a generic overview of the process, because each configuration is target-dependent. To configure the ELA to work with your target, see the TRMs for both your target and the ELA-500.

Procedure

1. Import the scripts into Arm Development Studio as usecase scripts.
   a) Open Arm Development Studio and import the ELA-500 examples file: **File** > **Import** > **Arm DS** > **Examples & Programming Libraries** > **Next**.
   b) Expand **Examples** and **Debug and Trace Services Layer (DTSL)**, and select **ELA-500**.
   c) Click **Finish**.
   d) Open the **Scripts** view, right-click **Use-case** and select **Add use case script directory**.
   If this option is inactive, connect to your target and the option will become active
   e) Browse to your workspace, select the **DTSLELA-500** folder, and click **OK**.
   Arm Development Studio finds all the ELA scripts in that folder, and displays them under the Use case list item.
2. Configure the ELA-500 using the Configuration Utility.
   a) Expand **ela_lowlevel.py**, right-click **Configure ELA** and select **Configure**.
   b) Under the **Common** tab, check **Enable trace** and click **Apply**.
   c) Configure the Trigger States using the **Trigger State {n}** tabs. This configuration is target-dependent. See the TRMs for the ELA-500 and for your target. For an example on what this might look like, see the linked ELA-500 tutorial at the end of the introduction.

14.10.2 Start and stop an ELA-500 trace capture

This topic describes how to start and stop a trace capture using the ELA-500 in Arm® Development Studio.

Procedure

1. To start an ELA-500 trace capture:
   a) Under the **ela_control.py** sub-menu, right-click **Run ELA-500**, and select **Run ela_control.py::Run ELA-500**.
   b) Run the target. The ELA-500 starts monitoring the specified signal groups running on the target, waiting to respond to the specified trigger conditions.
2. To stop an ELA-500 trace capture:
   a) Under the **ela_control.py** sub-menu, right-click **Stop ELA-500**, and select **Run ela_control.py::Stop ELA-500**.

14.10.3 Decode the trace capture

Describes how to decode a trace capture when using the ELA-500 in Arm® Development Studio, and provides some example output.

Before you begin

Check that your JSON file is specified in the `ela_example.py` file, as this is used by the Decode trace data function. You must place your JSON file in the `DTSLELA-500` directory.

Procedure

1. Under the **ela_example.py** sub-menu, right-click **Decode trace data** and select **Configure**.
2. Configure the Signal Groups. To see what each signal group refers to, refer to your target's documentation.
3. Right-click **Decode trace data** and click **Run**.

Example 14-1: Example

The raw data captured by the ELA looks like this:

```
Data: state = 0, overwrite = 1, counter=1, data = 91930905BEA4C03504A897513488810B
Data: state = 2, overwrite = 0, counter=0, data = E8811839A529D159A318B9330FFC31D3
Data: state = 5, overwrite = 0, counter=0, data = 70D1B1DBACDA8AA69CECBFECD89EDAF
Data: state = 6, overwrite = 0, counter=0, data = 501BA4E34421DABAA1443FEF04814076
Data: state = 1, overwrite = 0, counter=2, data = C990BF4889DA7876E0A3178C9A80EEDC
Data: state = 6, overwrite = 1, counter=2, data = 7019FA1873659F1B600EF7BD72B58501
```

Decoding the data, based on the configured signal groups, turns it into something like this:

```
read 180 words
Data: state = 0, overwrite = 1, counter=1, data = 91930905BEA4C03504A897513488810B
  timestamp[48:0]                  = 0x12ea26911021L
  sleep                            = Running
  reset                            = In reset
  power_up                         = Power up
Data: state = 2, overwrite = 0, counter=0, data = E8811839A529D159A318B9330FFC31D3
Data: state = 5, overwrite = 0, counter=0, data = 70D1B1DBACDA8AA69CECBFECD89EDAF
```

```
Data: state = 6, overwrite = 0, counter=0, data = 501BA4E34421DABAA1443FEF04814076
Data: state = 1, overwrite = 0, counter=2, data = C990BF4889DA7876E0A3178C9A80EEDC
  id                                = 0x17e9L
  status                            = Success
  data[31:0]                        = 0x44ed3c3bL
  r_w                               = Read
  address[63:0]                     = 0xe0a3178c9a80eedcL
Data: state = 6, overwrite = 1, counter=2, data = 7019FA1873659F1B600EF7BD72B58501
```

These code examples are for illustrative purposes only, to show the type of output you might expect when using the ELA-500.

14.11 Using the ELA-600

Start and stop a capture using the ELA-600, and decode the captured data. There is also an example of the ELA-600 output, both before and after decoding.

Before you begin

For the ELA to correctly decode your captured data, you need a JSON file that lists the Signal Groups of your SoC, and their connections.

Procedure

The designer of your target provides the JSON file. Alternatively you can use the provided `axi_interconnect_mapping.json` file as a starting point, and manually add the information for your target. You can find the information you need in the ConfigDB entry for your target.

Note

The ETR is enabled by default. If your ELA-600 is connected to a different trace sink, you need to disable the ETR; see the Start and stop an ELA-600 trace capture section for details on how to do this. You also need to configure your trace sink in the Configuration Utility. See Configure the ELA-600 for details on how to access the Configuration Utility.

14.11.1 Configure the ELA-600

Import the scripts and configure the ELA-600 for use with Arm® Development Studio. This guide provides a generic overview of the process, because each configuration is target-dependent. To configure the ELA to work with your target, see the TRMs for both your target and the ELA-600.

Procedure

1. Import the scripts into Arm Development Studio as usecase scripts:
   a) Open Arm Development Studio and import the DTSL zip file: **File** > **Import** > **Arm DS** > **Examples & Programming Libraries** > **Next**.
   b) Expand **Examples** and **Debug and Trace Services Layer (DTSL)**, and select **ELA-600**.
   c) Click **Finish**.
   d) Open the **Scripts** view, right-click **Use-case** and select **Add use case script directory**.
   If this option is inactive, connect to your target and the option will become active.
   e) Browse to your workspace, select the **DTSLELA-600** folder, and click **OK**.

Arm Development Studio finds all the ELA scripts in that folder, and displays them under the Use case list item.

2. Configure the ELA-600 using the Configuration Utility:
   a) Expand **ela_setup.py**, right-click **Configure ELA** and select **Configure**.
   b) Under the **Common** tab, configure the common registers of the ELA. One of the usual settings is Enable trace. When you have done this, click **Apply**.
   c) Configure the Trigger States using the **Trigger State {n}** tabs. This configuration is target-dependent. See the TRMs for the ELA-600 and for your target. For an example on what this might look like, see the linked ELA-600 tutorial at the end of the introduction.

14.11.2 Start and stop an ELA-600 trace capture

This topic describes how to start and stop a trace capture using the ELA-600 in Arm® Development Studio.

Before you begin

The ETR is enabled by default. If you are not using the ETR as your trace sink, you must disable it in two places as described in the following procedure.

Procedure

1. To start an ELA-600 trace capture:
   a) Make sure your target is connected.
   b) Under the **ela_control.py** sub-menu, right-click **Run ELA-600**, and select **Run ela_control.py::Run ELA-600**.
   c) Run the target. The ELA-600 starts monitoring the specified signal groups that are running on the target, waiting to respond to the specified trigger conditions.
2. If your ELA-600 is not using the ETR as the trace sink, disable the ETR:
   a) Right-click the **Run ELA-600** script, select **Configure**, and deselect the **Start the ETR when the ELA-600 starts** option.
   b) Configure and enable your trace sink using the **Configuration Utility**. For configuration, see Configure the ELA-600.
3. To stop an ELA-600 trace capture:
   a) Under the **ela_control.py** sub-menu, right-click **Stop ELA-600**, and select **Run ela_control.py::Stop ELA-600**.
4. If your ELA-600 is not using the ETR as the trace sink, disable the ETR:
   a) Right-click the **Stop ELA-600** script, select **Configure**, and deselect the **Stop the ETR when the ELA-600 stops** option.

14.11.3 Decompress and decode an ELA-600 trace

This section describes the various ways to configure the ELA-600 to correctly decompress and decode your trace capture, based on either an input source file or data coming from the buffer.

14.11.3.1 Decompress and decode an ELA-600 trace from buffer source

Describes how to configure the ELA-600 to generate either the raw output of your trace capture, or data that has been decoded and mapped to the components of your target, where the data is coming from the buffer source.

Procedure

1. To generate the raw output of your capture:
    a) In the **Scripts** view, expand **ela_process_trace.py**, right-click **Decompress and decode ELA trace** and select **Configure**.
    b) Under the **General** tab, make sure that **Decompress trace** is selected, and choose your preferred output option.
    c) If your trace data was processed with delta compression enabled, you will also need to go to the **Decompress** tab and select the **ELA trace captured with delta compression enabled** checkbox.
    d) Click **OK** to save these settings.
    e) Right-click **Decompress and decode ELA trace**, and select **Run ela_process_trace.py::Decompress and decode ELA trace**.

    The decompressed data captured by the ELA looks like this:

```
Trace data: trigger state = 0, overrun = 0,
 data=0x80300162000003481C00400028082D07
Trace data: trigger state = 0, overrun = 0,
 data=0xA030016200000 2C81C00400000602675
Trace data: trigger state = 0, overrun = 0,
 data=0x80400162000006506C005881F7C02C74
```

2. To generate data that has been mapped to your target's components:
    a) In the **Scripts** view, expand **ela_process_trace.py**, right-click **Decompress and decode ELA trace** and select **Configure**.
    b) Under the **General** tab, make sure that **Decompress and decode trace** is selected, and choose your preferred output option.
    c) If delta compression was enabled during the trace capture, under the **Decompress** tab, check the **ELA trace captured with delta compression enabled** checkbox.
    d) Under the **Decode** tab, specify your JSON file in the **ELA trace mapping file** field, and set the **State** for each monitored signal group by using the drop-down menus.
    e) Click **OK** to save these settings.
    f) Right-click **Decompress and decode ELA trace**, and select **Run ela_process_trace.py::Decompress and decode ELA trace**.

    Decoding the data, based on the configured signal groups, turns it into something like this:

```
Trace type: Data, Trace Stream: 0, Overrun: 0, Data:
 0x80300162000003481C00400028082D07
P1_VALID : 1'h1
P1_AXID : 12'h6
P1_addr : 42'hB1000000
P1non-secure : 1'h0 => secure
Type_P1 : 4'hD => Exclusive Read
P0_VALID : 1'h0
P0_AXID : 12'h40E
P0_addr : 42'h80005010
P0non-secure : 1'h0 => secure
Type_P0 : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1 : 6'h34
TTID_P0 : 6'h7
```

```
Trace type: Data, Trace Stream: 0, Overrun: 0, Data:
 0xA0300162000002C81C00400000602675
P1_VALID : 1'h1
P1_AXID : 12'h406
P1_addr : 42'hB1000000
P1non-secure : 1'h0 => secure
Type_P1 : 4'hB => Write Back, Writes Clean
P0_VALID : 1'h0
P0_AXID : 12'h40E
P0_addr : 42'h800000C0
P0non-secure : 1'h0 => secure
Type_P0 : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1 : 6'h19
TTID_P0 : 6'h35
Trace type: Data, Trace Stream: 0, Overrun: 0, Data:
 0x80400162000006506C005881F7C02C74
P1_VALID : 1'h1
P1_AXID : 12'h8
P1_addr : 42'hB1000000
P1non-secure : 1'h1 => non-secure
Type_P1 : 4'h9 => Write No Snoop
P0_VALID : 1'h0
P0_AXID : 12'h836
P0_addr : 42'hB103EF80
P0non-secure : 1'h0 => secure
Type_P0 : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1 : 6'h31
TTID_P0 : 6'h34
```

14.11.3.2 Decompress and decode an ELA-600 trace from binary source file

Describes how to configure the ELA-600 to generate either the raw output of your trace capture, or data that has been decoded and mapped to the components of your target, where the data is coming from the binary source file.

Procedure

1. To generate the raw output from your source file:
   a) In the **Scripts** view, expand **ela_process_trace.py**, right-click **Decompress and decode stored binary data** and select **Configure**.
   b) Under the **General** tab, select **Decompress trace** and specify the source file in the **Binary input file...** field.
   c) If your trace data was processed with delta compression enabled, you will also need to go to the **Decompress** tab and select the **ELA trace captured with delta compression enabled** checkbox.
   d) Click **OK** to save these settings.
   e) Right-click **Decompress and decode stored binary data**, and select **Run ela_process_trace.py::Decompress and decode stored binary data**.
   The decompressed data captured by the ELA looks like this:

```
Trace data: trigger state = 0, overrun = 0,
 data=0x80300162000003481C00400028082D07
Trace data: trigger state = 0, overrun = 0,
 data=0xA0300162000002C81C00400000602675
Trace data: trigger state = 0, overrun = 0,
 data=0x80400162000006506C005881F7C02C74
```

2. To decompress and decode the data from the source file:

a) In the **Scripts** view, expand **ela_process_trace.py**, right-click **Decompress and decode stored binary data** and select **Configure**.
b) Under the **General** tab, select **Decompress and decode trace** and specify the source file in the **Binary input file...** field.
c) If delta compression was enabled during the trace capture, under the **Decompress** tab, check the **ELA trace captured with delta compression enabled** checkbox.
d) Under the **Decode** tab, specify your JSON file in the **ELA trace mapping file** field, and set the **State** for each monitored signal group by using the drop-down menus.
e) Click **OK** to save these settings
f) Right-click **Decompress and decode stored binary data**, and select **Run ela_process_trace.py::Decompress and decode stored binary data**.

Decoding the data, based on the configured signal groups, turns it into something like this:

```
Trace type: Data, Trace Stream: 0, Overrun: 0, Data:
 0x80300162000003481C00400028082D07
P1_VALID : 1'h1
P1_AXID : 12'h6
P1_addr : 42'hB1000000
P1non-secure : 1'h0 => secure
Type_P1 : 4'hD => Exclusive Read
P0_VALID : 1'h0
P0_AXID : 12'h40E
P0_addr : 42'h80005010
P0non-secure : 1'h0 => secure
Type_P0 : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1 : 6'h34
TTID_P0 : 6'h7
Trace type: Data, Trace Stream: 0, Overrun: 0, Data:
 0xA0300162000002C81C00400000602675
P1_VALID : 1'h1
P1_AXID : 12'h406
P1_addr : 42'hB1000000
P1non-secure : 1'h0 => secure
Type_P1 : 4'hB => Write Back, Writes Clean
P0_VALID : 1'h0
P0_AXID : 12'h40E
P0_addr : 42'h800000C0
P0non-secure : 1'h0 => secure
Type_P0 : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1 : 6'h19
TTID_P0 : 6'h35
Trace type: Data, Trace Stream: 0, Overrun: 0, Data:
 0x80400162000006506C005881F7C02C74
P1_VALID : 1'h1
P1_AXID : 12'h8
P1_addr : 42'hB1000000
P1non-secure : 1'h1 => non-secure
Type_P1 : 4'h9 => Write No Snoop
P0_VALID : 1'h0
P0_AXID : 12'h836
P0_addr : 42'hB103EF80
P0non-secure : 1'h0 => secure
Type_P0 : 4'h2 => Read Shared, Read Clean, Read No Snoop Dirty
TTID_P1 : 6'h31
TTID_P0 : 6'h34
```

14.11.3.3 Dump ELA-600 trace data to binary source file

Describes how to transfer data from the ELA-600 buffer to a binary source file.

Procedure

1. In the **Scripts** view, expand **ela_process_trace.py**, right-click **Dump ELA trace** and select **Configure**.
2. Specify a name for the **Output file** and click **OK**.
3. To generate the file, right-click **Dump ELA trace** and select **Run ela_process_trace.py::Dump ELA trace**.

14.12 Extending the DTSL object model

For most platform configurations, the DTSL configuration class creates standard Java DTSL components, such as CoreSight™ devices or Arm cores, represented as Device objects. Sometimes, the behavior of these standard components needs to be changed, or new DTSL components need to be created.

Arm® Debugger uses the Java components that the DTSL configuration script creates. Since there is a high level of integration between Java and Jython, the DTSL configuration can create new Jython objects which extend the standard Java DTSL objects. And Arm Debugger can also use these Jython objects to access the target platform. This is because of the very tight integration between Java and Jython. This way of modifying behavior is straightforward if you are familiar with object oriented techniques, especially in Java. The only new technique might be the way in which a Java object can by modified by extending it in Jython. This is possible because Jython code is compiled down to Java byte code, so the system does not know whether the code was written in Java or Jython.

14.12.1 Performing custom actions on connect

On some platforms, it might be necessary to configure the system to enable access by a debugger. For example, some platforms have a scan chain controller that controls which devices are visible on the JTAG scan chain.

On other platforms, it might be necessary to power up subsystems by writing control registers. The DTSL configuration provides several hooks that can be overridden to perform such actions.

Each DTSL configuration class is derived from a parent class, usually `DTSLv1` . The derived class gets all the methods the parent class implements and can replace the methods of the parent class to modify the behavior. When replacing a method, the original implementation can be called by `DTSLv1.<method_name>()`.

postRDDIConnect

This is called immediately after the RDDI interface has been opened. At this point, the RDDI interface has been opened, but no connection to the debug server has been made. This method should be implemented to perform low-level configuration, for example, using the JTAG interface to configure a TAP controller to make debug devices visible on the JTAG scan chain.

```
class DtslScript(DTSLv1):
   '''A top-level configuration class which supports debug and trace'''

 [snip]

   def postRDDIConnect(self):
       DTSLv1.postRDDIConnect(self)
       self.jtag_config()

   def jtag_config(self):
       jtag = self.getJTAG()
       pVer = zeros(1, 'i')
       jtag.connect(pVer)
       try:
           jtag.setUseRTCLK(0)
           jtag.setJTAGClock(1000000)
           # perform target configuration JTAG scans here
       finally:
           jtag.disconnect()
```

postDebugConnect

This is called after the RDDI debug interface has been opened. At this point, the RDDI debug interface has been opened, but no connection to any device has been made. This method should be implemented to perform any configuration required to access devices. For example, writes using a DAP to power on other components could be performed here.

```
class DtslScript(DTSLv1):
   '''A top-level configuration class which supports debug and trace'''

 [snip]

   def postDebugConnect(self):
       DTSLv1.postDebugConnect(self)
       self.power_config()

   def power_config(self):
       self.ahb.connect()
       # power up cores
       self.ahb.writeMem(0xA0001000, 1)
       self.ahb.disconnect()
```

postConnect

This is called after the connection and all devices in the managed device list have been opened. This method should be implemented to perform any other configuration that isn't required to be done at an earlier stage, for example, trace pin muxing.

```
class DtslScript(DTSLv1):
   '''A top-level configuration class which supports debug and trace'''

 [snip]

   def postConnect(self):
       DTSLv1.postConnect(self)
       self.tpiu_config()

   def tpiu_config(self):
       # select trace pins
       self.ahb.writeMem(0xB0001100, 0xAA)
```

Related information

DTSL device objects on page 527

14.12.2 Overriding device reset behavior

For a DSTREAM class device, the default operation for a System Reset request is to drive `nSRST` on the JTAG connector. On some platforms, this pin is not present on the JTAG connector. So, some other method must be used to perform the reset.

Sometimes, the reset is performed by writing to another system component, such as a System Reset Controller device. If this is not available, another approach is to cause a system watchdog timeout, which in turn causes a system reset. Whichever approach is taken, the default reset behavior must be modified. To override the default reset behavior, the `resetTarget` method can be overridden to perform the necessary actions.

The following code sequence is an example of this:

```
from com.arm.debug.dtsl.components import ConnectableDevice
[snip]

 class DtslScript(DTSLv1):
    '''A top-level configuration class which supports debug and trace'''

  [snip]

     def setupPinMUXForTrace(self):
         '''Sets up the IO Pin MUX to select 4 bit TPIU trace'''
         addrDBGMCU_CR = 0xE0042004
         value = self.readMem(addrDBGMCU_CR)
         value |=  0xE0 # TRACE_MODE=11 (4 bit port), TRACE_IOEN=1
         self.writeMem(addrDBGMCU_CR, value)

     def enableSystemTrace(self):
         '''Sets up the system to enable trace
            For a Cortex-M3 system we must make sure that the
            TRCENA bit (24) in the DEMCR registers is set.
            NOTE: This bit is normally set by the DSTREAM Cortex-M3
                  template - but we set it ourselves here in case
                  no one connects to the Cortex-M3 device.
         '''
         addrDEMCR = 0xE000EDFC
         bitTRCENA = 0x01000000
         value = self.readMem(addrDEMCR)
         value |= bitTRCENA
         self.writeMem(addrDEMCR, value)

     def postReset(self):
         '''Makes sure the debug configuration is re-instated
            following a reset event
         '''
         if self.getOptionValue("options.traceBuffer.traceCaptureDevice") ==
 "DSTREAM":
             self.setupPinMUXForTrace()
         self.enableSystemTrace()

     def resetTarget(self, resetType, targetDevice):
         # perform the reset
         DTSLv1.resetTarget(self, resetType, targetDevice)
         # perform the post-reset actions
         Self.postReset()
```

Line `38` declares the `resetTarget` method. This calls the normal reset method to perform the reset and then calls the custom `postReset` method to perform the actions required after a reset.

The implementation of `resetTarget` in `DTSLv1` is to call the `systemReset` method of the `targetDevice`.

14.12.3 Adding a new trace capture device

Arm® Debugger has built in support for reading trace data from DSTREAM, ETB, TMC/ETM and TMC/ETR devices. Adding support for a new trace capture device is not very difficult, however, and can be done entirely with DTSL Jython scripts.

The DTSL trace capture objects class hierarchy shows that all DTSL trace capture objects are derived from the `ConnectableTraceCaptureBase` class. This base class implements two interfaces, `ITraceCapture` and `IDeviceConnection`. `ITraceCapture` defines all the methods that relate to controlling and reading trace data from a capture device, and `IDeviceConnection` defines the methods for a component that needs to be connected to. The `ConnectableTraceCaptureBase` class contains stub implementations for all the methods in both interfaces.

To create a new trace capture class:

1. Create a new class derived from the `ConnectableTraceCaptureBase` class, or the `TraceCaptureBase` class if appropriate.
2. Implement the class constructor, making sure to call the base class constructor in your implementation.
3. Override the `startTraceCapture()` and `stopTraceCapture()` methods. The default implementations of these methods throw an exception when DTSL calls them, so you must override them to avoid this.
4. Override the `getCaptureSize()` method to return the size of raw trace data in the device.
5. Override the `getSourceData()` method to return trace data for a specified trace source.
6. If your trace device requires a connection, override the `connect()`, `disconnect()`, and `isConnected()` methods.
7. In your platform DTSL Jython script, create an instance of your new trace capture device class and add it to the DTSL configuration.

The following example Jython code implements a new trace capture device which reads its trace data from an ETB dump file (the raw content of an ETB buffer). It is assumed that this code is in `FileBasedTraceCapture.py`.

```
from java.lang import Math
from com.arm.debug.dtsl.impl import DataSink
from com.arm.debug.dtsl.impl import Deformatter
from com.arm.debug.dtsl.impl import SyncStripper
from com.arm.debug.dtsl.components import ConnectableTraceCaptureBase
from com.arm.debug.dtsl.configurations import ConfigurationBase
import sys
import os
import jarray
class FileBasedTraceCaptureDevice(ConnectableTraceCaptureBase):
    '''
    Base class for a trace capture device which just returns
    a fixed data set from a file. The amount of trace data captured
    is just the size of the file.
    '''
    def __init__(self, configuration, name):
        '''Construction
        Params: configuration
                    the top level DTSL configuration (the
                    class you derived from DTSLv1)
                name
                    the name for the trace capture device
        '''
        ConnectableTraceCaptureBase.__init__(self, configuration, name)
        self.filename = None
        self.fileOpened = False
        self.hasStarted = False
        self.trcFile = None
    def setTraceFile(self, filename):
        '''Sets the file to use as the trace data source
        Params: filename
                    the file containing the trace data
        '''
        self.filename = filename
    def connect(self):
        '''We interpret connect() as an opening of the trace data file
        '''
        self.trcFile = file(self.filename, 'rb')
        self.fileOpened = True
        self.fileSize = os.path.getsize(self.filename)
    def disconnect(self):
        '''We interpret disconnect() as a closing of the trace data file
        '''
        if self.trcFile != None:
            self.trcFile.close()
        self.fileOpened = False
        self.fileSize = 0
    def isConnected(self):
        return self.fileOpened
    def startTraceCapture(self):
        self.hasStarted = True
    def stopTraceCapture(self):
        self.hasStarted = False
    def getMaxCaptureSize(self):
        return self.fileSize
    def setMaxCaptureSize(self, size):
        return self.getMaxCaptureSize()
    def getCaptureSize(self):
        return self.fileSize
    def getNewCaptureSize(self):
        return self.getCaptureSize()
    def hasWrapped(self):
        return True
class ETBFileBasedTraceCaptureDevice(FileBasedTraceCaptureDevice):
    '''
    Creates a trace capture device which returns ETB trace
    data from a file.
    '''
    def __init__(self, configuration, name):
        '''Construction
        Params: configuration
                    the top level DTSL configuration (the
                    class you derived from DTSLv1)
                name
                    the name for the trace capture device
        '''
        FileBasedTraceCaptureDevice.__init__(self, configuration, name)
    def getSourceData(self, streamID, position, size, data, nextPos):
        '''Reads the ETB trace data from the file
        Params: streamID
                    for file formats which contain multiple
```

```
                     streams, this identifies the stream for which
                     data should be returned from
                 position
                     the byte index position to read from
                 size
                     the max size of data (in bytes) we should return
                 data
                     where to write the extracted data
                 nextPos
                     an array into which we set entry [0] to the
                     next position to read from i.e. the position parameter
                     value which will return data that immediately follows
                     the last entry written into data
         '''
         # We assume that size is small enough to allow to read an entire
         # data block in one operation
         self.trcFile.seek(position)
         rawdata = jarray.array(self.trcFile.read(size), 'b')
         nextPos[0] = position+size
         dest = DataSink(0, 0, size, data)
         # We assume the file contains TPIU frames with sync sequences
         # Se we set up a processing chain as follows:
         # file data -> strip syncs -> de formatter -> to our caller
         deformatter = Deformatter(dest, streamID)
         syncStripper = SyncStripper(deformatter)
         syncStripper.forceSync(True)
         syncStripper.push(rawdata)
         syncStripper.flush()
         return dest.size()
```

We can use the new trace capture device in the platform DTSL Jython code:

```
from FileBasedTraceCapture import ETBFileBasedTraceCaptureDevice
[snip]
   self.etbFileCaptureDevice = ETBFileBasedTraceCaptureDevice(self, 'ETB(FILE)')
   self.etbFileCaptureDevice.setTraceFile('c:\\etbdump.bin')
   self.addTraceCaptureInterface(self.etbFileCaptureDevice)
```

We can add it to the configuration as though it were an ETB or DSTREAM device.

Related information

DTSL trace capture objects on page 530

14.13 Debugging DTSL Jython code within Arm Debugger

When Arm® Development Studio connects to a platform, it automatically loads the platform Jython script and creates an instance of the configuration class. The Jython scripts which are shipped with Arm Development Studio should not contain any errors, but if you create your own scripts, or make modifications to the installed scripts, then you might introduce errors. These errors have two common forms:

- Syntax or import errors
- Functional errors.

14.13.1 DTSL Jython syntax errors

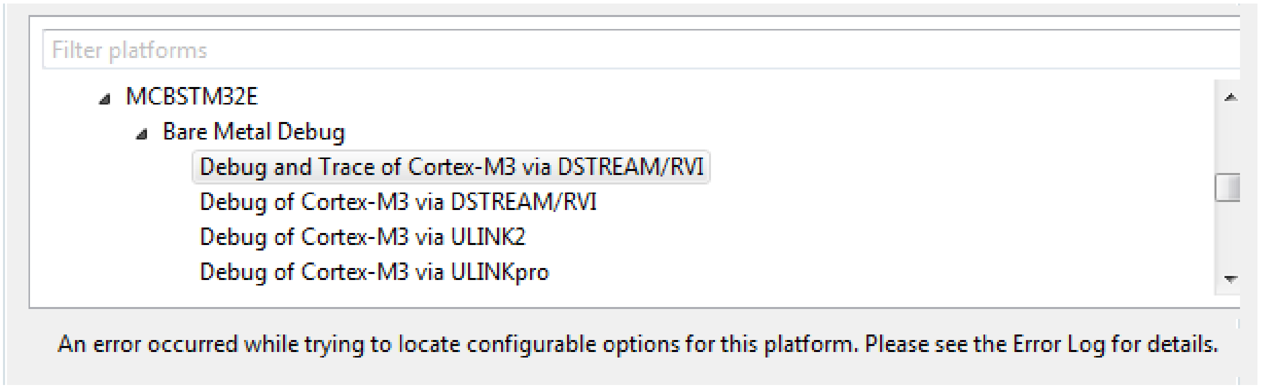
These can occur in two situations:

1. Attempting to change the DTSL options from within the Launcher Panel.
2. Attempting to connect Arm® Debugger to the platform.

14.13.2 Errors reported by the launcher panel

These errors usually appear in the area where the **Edit...** button for the DTSL options would normally appear, replacing it with a message:

Figure 14-20: Launcher panel reporting DTSL Jython script error


Filter platforms
MCBSTM32E
Bare Metal Debug
Debug and Trace of Cortex-M3 via DSTREAM/RVI
Debug of Cortex-M3 via DSTREAM/RVI
Debug of Cortex-M3 via ULINK2
Debug of Cortex-M3 via ULINKpro

An error occurred while trying to locate configurable options for this platform. Please see the Error Log for details.

To find the cause of the error, try inspecting the Error Log. If the Error Log is not visible, select **Window** > **Show View** > **Error Log** to show it.

The following is an example of some Error Log text:

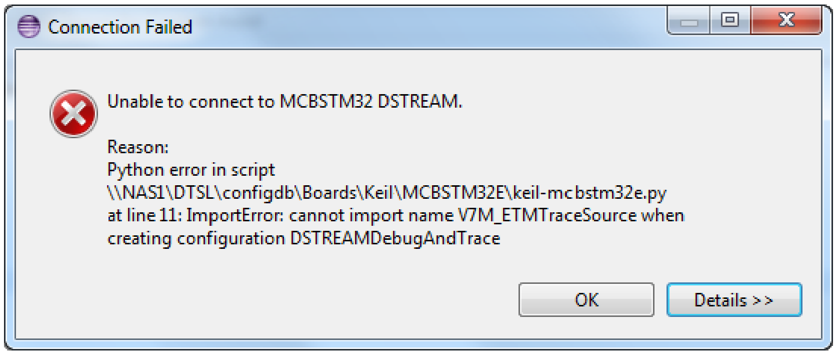
```
Python error in script \\\\NAS1\\DTSL\\configdb\\Boards\\Keil\\MCBSTM32E\\keil-
mcbstm32e.py at line 11: ImportError: cannot import name V7M_ETMTraceSource when
 creating configuration DSTREAMDebugAndTrace
```

After resolving any issues, close and reopen the Launcher Panel to make Arm® Development Studio reinspect the Jython script. If an error still occurs, you get more entries in the Error Log. If the error is resolved, then the **Edit...** button for the DTSL options will appear as normal.

14.13.3 Errors reported at connection time

If you try to connect to a platform which contains an error in its Jython script, Arm® Development Studio displays an error dialog box indicating the cause of the error:

Figure 14-21: Connection Error dialog box



```
Unable to connect to MCBSTM32 DSTREAM.

Reason:
Python error in script
\\NAS1\DTSL\configdb\Boards\Keil\MCBSTM32E\keil-mcbstm32e.py
at line 11: ImportError: cannot import name V7M_ETMTraceSource when
creating configuration DSTREAMDebugAndTrace
```

Note

Sometimes, the error message shown in the dialog box might not be helpful, especially for run-time errors rather than syntax or import errors. Arm Development Studio also places an entry in the Error Log, so that you can inspect the error after dismissing the error dialog box. This error log entry might contain more information. You can typically find this information by scrolling down the Exception Stack Trace until you see the error reported at the point the Jython code was run.

After editing the Jython script to resolve any issues, try connecting again.

Note

You do not need to tell Arm Development Studio that the configdb has changed when you make changes only to Jython scripts.

14.13.4 DTSL Jython functional errors

If the Jython script error you are tracking down cannot be resolved by code inspection, then you might need to use a Jython debugger. For some use cases, you can use the debugger which is built in to Arm® Development Studio as part of PyDev. Other use cases, however, display modal dialog boxes within Arm Development Studio, preventing the use of the same instance of Arm Development Studio to debug the scripts. Arm therefore recommends that you use another instance of Arm Development Studio, or another Eclipse installation which also contains the PyDev plugin or plugins.

Note

Although you can run multiple instances of the Arm Development Studio IDE at the same time, the instances cannot use the same Workspace.

14.13.5 Walk-through of a DTSL debug session

Make sure that Arm® Development Studio is using your intended workspace.

The debug session involves modifying a DTSL Jython script, so make sure that you are using a writeable copy of the Arm Development Studio configdb.

Related information

Modifying Arm Development Studio configdb on page 517

14.13.6 Starting a second instance of Arm Development Studio for Jython debug

When you start a second instance of Arm® Development Studio, with the first instance still running, you are asked to use a different workspace. Choose a suitable location for this second workspace.

In this second instance of Arm Development Studio, switch to the PyDev perspective. To enable the toolbar buttons that allow you to start and stop the PyDev debug server:

- Select **Window** > **Customize Perspective...** .
- Click the **Command Groups Availability** tab.
- Scroll down through the Available Command Groups and select the PyDev Debug entry.
- Click the **Tool Bar Visibility** tab.
- Make sure that the PyDev Debug entry, and the two End Debug Server and Start Debug Server entries, are selected.

On the toolbar, you should see two new icons **PyDev debug server start and stop icons** to stop and start the debug server.

Click the green P-bug icon to start the PyDev debug server. You should see a console view reporting the port number on which the debugger is listening (5678 by default). The Arm Development Studio instance is ready to accept remote debug connections.

Switch to the **Development Studio** perspective. Arm Development Studio IDE does not automatically switch to the Development Studio perspective when a connection is made to the PyDev debugger. So if you do not switch to the Development Studio perspective yourself, then you cannot notice the connection.

14.13.7 Preparing the DTSL script for debug

When a Jython script is being debugged, it is normally launched by PyDev, and PyDev can optionally create a debug session for the script. When Arm® Development Studio launches the Jython script, however, this does not happen. This is not a problem, however, because the script itself can register with the PyDev debugger after it is launched. To do this in your script:

- Extend the import list for the script to import pydevd. If you are using a second Arm Development Studio instance to host the PyDev debugger, then add the following to the top of the DTSL script: `import pydevd` If you are using another Eclipse (non-Arm Development Studio) to host the PyDev debugger, then import the pydevd from that Eclipse instance. Locate the pydev plugin `pysrc` directory and add its path to the import path before importing pydevd.

  For example, if the Eclipse is installed in `C:\Users\<username>\eclipse` , then the code would be as follows:

  ```
  import sys;
  sys.path.append(r'C:\Users\<username>\eclipse\plugins
  \org.python.pydev_2.7.4.2013051601\pysrc')
  import pydevd
  ```

  Where `pydev_<xyz>` depends on the version of pydev installed within Eclipse.
- Insert the following line at the location where you want the PyDev debugger to gain control of the script: `pydevd.settrace(stdoutToServer=True, stderrToServer=True)` This causes a break into the debugger at that location, and redirects all standard output from the script to the debugger console. This allows you to place print statements into the script and see them in the debugger, whereas normally Arm Development Studio would discard any such print output. Good places to insert this statement are:
  - In the constructor (`__init__`) for the DTSL configuration class.
  - In the `optionValuesChanged` method.

The function documentation for the `settrace` call in pydev 2.7.4 is as follows:

```
def settrace(host=None, stdoutToServer=False, stderrToServer=False, port=5678,
 suspend=True, trace_only_current_thread=True):
    '''Sets the tracing function with the pydev debug function and initializes
 needed facilities.
    @param host: the user may specify another host, if the debug server is not in
 the
          same machine (default is the local host)
    @param stdoutToServer: when this is true, the stdout is passed to the debug
 server
    @param stderrToServer: when this is true, the stderr is passed to the debug
 server
          so that they are printed in its console and not in this process console.
```

```
    @param port: specifies which port to use for communicating with the server (note
that
        the server must be started in the same port).
        @note: currently it's hard-coded at 5678 in the client
    @param suspend: whether a breakpoint should be emulated as soon as this function
        is called.
    @param trace_only_current_thread: determines if only the current thread will be
        traced or all future threads will also have the tracing enabled.
    '''
```

Note

Calls to the DTSL `.settrace()` function without an active debug server produces errors. For example, you might see errors similar to `Python error in script pyclasspath/Lib/socket.py at line 1,159: error: when creating configuration DtslScript`

In this situation:

- Check if the debug server has crashed during your debug session. Restart debug server if required.
- Check if you have removed the debug code from your script. It is good practice to remove debug code from your script when you have finished debugging. Run your script again after removing the debug code.

14.13.8 Debugging the DTSL code

In your main instance of Arm® Development Studio (not the PyDev debug instance), launch the connection to the platform. When the DTSL script reaches the `settrace` call, the second Arm Development Studio instance halts the execution of the script immediately after the call. This allows you to use the PyDev debugger for tasks such as stepping through the code, examining variables, and setting breakpoints. While you are debugging, your main Arm Development Studio instance waits for the Jython script to complete.

14.14 DTSL in stand-alone mode

DTSL is commonly used by Arm® Debugger, both within the IDE and in the console version of the debugger. However, it can also be used in 'stand-alone' mode, completely outside of the IDE. This allows you to use the DTSL API to take care of the target connection and configuration when writing your program. The rest of your program can concentrate on the main function of your application.

DTSL is mainly written in Java and Jython. There are therefore two kinds of stand-alone program, those written in Jython and those in Java. The `DTSLExamples.zip` file contains examples of both kinds of program, which you can look at to help you decide the best route for your application. The programs are easy to compare because they both do essentially the same things.

14.14.1 Comparing Java with Jython for DTSL development

The advantages of Java are:

- The Javadoc for DTSL is directly available, which helps greatly when writing DTSL Java programs in environments such as the Arm® Development Studio IDE, which is based on Eclipse.
- Java programs seem to have faster start-up times than Jython programs.
- The Eclipse Java development environment might be considered more mature than the Python PyDev Eclipse development environment.

The advantages of Jython are:

- There are probably more people familiar with Python than with Java.
- Python is not a statically-typed language. So it is easier to write Python code without always having to create variables of specific types.

The disadvantages of Jython are:

- There is no DTSL Javadoc support, because the PyDev editor does not understand how to extract the Javadoc information from the Java `.jar` files.
- Python is not a statically-typed language, so it is hard for the PyDev editor to know the type of a variable. Using "`assert isinstance(<variable>,<type>)`" works around this to an extent, and this code appears many times in the example. After the PyDev editor sees it, it knows the type of the variable and so can provide code completion facilities. However, you still do not get access to the Javadoc within the editor. If you want to access the Javadoc, you must do it by some other method, such as through a web browser.
- Jython can be slower than Java. For example, if Jython is used as part of a trace decoding pipeline, it can significantly slow down trace processing.

14.14.2 DTSL as used by a stand-alone Jython program

The example Jython application demonstrates how to do the following:

- Create a DTSL configuration instance for the requested platform.
- Connect to a core device, such as a Cortex®-M3 or other such Arm core.
- Perform the following operations:
    - Get control of the core following a reset.
    - Read and write registers on the core.
    - Read and write memory through the core.
    - Single step instructions on the core.
    - Start and stop core execution.

The example application connects to and controls the Arm core only. However, it can just as easily connect to any of the devices in the configuration, such as CoreSight™ components (PTM or ETB), and configure and control those devices as well.

Note

The example is a complete stand-alone application. It cannot be run when a Arm Debugger connection is made to the same target. However, a Arm Debugger Jython script can access the DTSL configuration. If you do this, take care not to interfere with the debugger.

14.14.3 Installing the Jython example within the Arm Development Studio IDE

The DTSL Python example project requires that you have Jython and the PyDev plugin installed.

To download Jython, and for installation instructions, go to https://www.jython.org/. This document is written with reference to Jython 2.5.3, but later versions should also work.

To download PyDev, and for installation instructions, go to https://pydev.org/. Make sure you configure PyDev to know about the Jython version you have installed.

The example project DTSLPythonExample is in the `DTSLExamples.zip` file. You can import `DTSLPythonExample` directly into your Arm® Development Studio workspace. You must also import `DTSL.zip` into your workspace.

The example project also contains two launch configurations for running the program. One configuration uses the Arm Development Studio configdb board specification, and the other refers directly to the files in the configdb. The project contains a configdb extension, which contains the Keil® MCBSTM32 entries compatible with this project.

Note

If you use your own Eclipse (non Arm Development Studio) installation, then you must set the Arm Development Studio installation location within the Arm Development Studio preferences. This value is used within the provided launch configurations.

The `readme.txt` file contained within the project has more information.

Related information

Additional DTSL documentation and files on page 511

14.14.4 Running the Jython program

To run the example in the IDE:

1. Import the supplied launch configurations.

2. Modify the program arguments to refer to your installed Arm® Development Studio location.

1. Run or debug the application.

To run the example use:

- `dtslexample.bat` from Windows
- `dtslexample` from Linux.

Before running the file, edit it and change the program parameters to suit the target system you are connecting to. You might need to make the following changes:

- Change the location of `jython.bat` to match your Jython installation. Arm Development Studio does contain part of a Jython installation, but it lacks the main `jython.bat` executable, so you must install your own.
- Change the defined location of the Arm Development Studio workspace.
- Change the location of the Arm Development Studio configuration database to include the database installed by Arm Development Studio and any further extensions you require (the location within a workspace of `DTSLExampleConfigdb\configdb` is an extension required to run the example).
- Change the connection address for the DSTREAM box to match your box. If you are using a USB connection then the code `--connectionAddress "USB"` can be left unchanged, but if you are using a TCP connection then you must change it to be of the form `--connectionAddress "TCP:<host-name|ip-address>"`, for example `--connectionAddress "TCP:DS-Tony"` or `--connectionAddress "TCP:192.168.1.32"`.
- Change the manufacturer to match the directory name of your platform in the `Boards` sub-directory of the Arm Development Studio config database.
- Change the board name to match the name of the board directory within the manufacturer directory.
- Change the debug operation to match one of the activity names contained in a bare metal debug section of the `project_types.xml` file. For example:

```
<activity id="ICE_DEBUG" type="Debug">

<name language="en">Debug Cortex-M3</name>
```

When you run the `dtslexample.py` script, it connects to the target and runs through a series of register, memory, and execution operations. By default, the script assumes that there is RAM at `0x20000000`, and that there is 64KB of it. This is correct for the Keil® MCBSTM32 board. To change these values, use the `--ramStart` and `--ramSize` options.

14.14.5 Invoking the Jython program

For information on the full set of program arguments, run the program with the `--help` parameter.

There are two ways to invoke the program:

- Specify the DTSL connection properties directly, using the { `--rddiConfigFile` , `--dtslScript`, `--dtslClass`, `--connectionType`, `--connectionAddress`, `--device` } parameters.
- Specify the Arm® Development Studio configdb parameters (equivalent to using the Eclipse launcher) using the {`--configdb`, `--manufacturer`, `--board`, `--debugOperation`, `--connectionType`, `--connectionAddress` } parameters, and let the program extract the DTSL connection properties from the Arm Development Studio configdb.

14.14.6 About the Jython program

We provide a DTSL Eclipse project that contains an example Jython program.

- The main program is in the `dtslexample.py` source file.
- The project is set up to use the DTSL libraries from the DTSL Eclipse project.
- The DTSL interaction flow is as follows:
  1. Connecting to DTSL. This involves forming the `ConnectionParameters` set and passing it to the DTSL static `ConnectionManager.openConnection()` method. See the Python method `connectToDTSL()` for details.
  2. Accessing the DTSL connection configuration and locating the DTSL object with the name requested in either:
     - the `--device` parameter
     - the core specified in the Arm® Development Studio configdb platform debug operation.
  3. Connecting to the core located in step 2.
  4. Performing the operations on the core, which is represented by a DTSL object that implements the `IDevice` interface. The DTSL Javadoc lists the full set of operations available on such an object. The example uses some of the more common operations, but does not cover all of them.
  5. Disconnecting from the core.
  6. Disconnecting from DTSL.
- The `IDevice` interface is a Java interface, so there are some operations which take Java parameters such as `StringBuilder` objects. This is not a problem for Jython because you can create such Java objects within your Jython program. Most of the memory operations use Java `byte[]` arrays to transport the data. Interfacing these between Jython and Java is relatively simple, but be sure to inspect the example code carefully if you want to understand how to do this.
- The `IDevice` Java interface wraps the RDDI-DEBUG C interface thinly, which means that many of the RDDI constants are used directly rather than being wrapped. This is why the example uses constants such as `RDDI_ACC_SIZE.RDDI_ACC_DEF`.

14.14.7 DTSL as used by a stand-alone Java program

The example Java application shows you how to do the following:

- Create a DTSL configuration instance for the requested platform.
- Connect to a core device, such as a Cortex®-M3 or other such Arm core.
- Perform the following operations:
  - Get control of the core following a reset.
  - Read and write registers on the core.
  - Read and write memory through the core.
  - Single step instructions on the core.
  - Start and stop core execution.

The example application connects to and controls the Arm core only. However, it can just as easily connect to any of the devices in the configuration, such as CoreSight™ components (PTM or ETB), and configure and control those devices as well.

Note

The example is a complete stand-alone application. It cannot be run when a Arm Debugger connection is made to the same target. However, a Arm Debugger Jython script can access the DTSL configuration. If you do this, take care not to interfere with the debugger.

14.14.8 Installing the Java example within the Arm Development Studio IDE

The example project DTSLJavaExample is in the `DTSLExamples.zip` file. You can import `DTSLJavaExample` directly into your Arm® Development Studio workspace. You must also import `DTSL.zip` into your workspace. After importing it, change the project configuration to refer to your DTSL library location:

1. Select the DTSLJavaExample within the Project Explorer, right click it, and select **Properties**.
2. Select 'Java Build Path' from the properties list.
3. Click the **Libraries** tab.
4. Replace all the referenced `DTSL\libs.jar` files with new entries which have the correct paths.

The example project also contains two launch configurations for running the program. One configuration uses the Arm Development Studio configdb board specification, and the other refers directly to the files in the configdb. The project contains a configdb extension, which contains the Keil® MCBSTM32 entries compatible with this project.

Note

If you use your own Eclipse (non Arm Development Studio) installation, then you must set the Arm Development Studio installation location within the Arm Development Studio preferences. This value is used within the provided launch configurations.

The `readme.txt` file contained within the project has more information.

Related information

Additional DTSL documentation and files on page 511

14.14.9 Running the Java program

To run the example in the IDE:

1. Import the supplied launch configurations.
2. Modify the program arguments to refer to your installed Arm® Development Studio location.
3. Run or debug the application.

To run the example use:

- `dtslexample.bat` from Windows
- `dtslexample` from Linux.

Before running the batch file, edit it and change the program parameters to suit the target system you are connecting to. You might need to make the following changes:

- Change the defined location of the Arm Development Studio workspace.
- Change the location of the Arm Development Studio configuration database to include the database installed by Arm Development Studio.
- Change the connection address for the DSTREAM box to match your box. If you are using a USB connection then the code `--connectionAddress "USB"` can be left unchanged, but if you are using a TCP connection then you must change it to be of the form `--connectionAddress "TCP:<host-name|ip-address>"`, for example `--connectionAddress "TCP:DS-Tony"` or `--connectionAddress "TCP:192.168.1.32"`.
- Change the manufacturer to match the directory name of your platform in the `Boards` sub-directory of the Arm Development Studio config database.
- Change the board name to match the name of the board directory within the manufacturer directory.
- Change the debug operation to match one of the activity names contained in a bare metal debug section of the `project_types.xml` file. For example:

```
<activity id="ICE_DEBUG" type="Debug">

<name language="en">Debug Cortex-M3<name>
```

When you run the `DTSLExample.java` program, it connects to the target and runs through a series of register, memory, and execution operations. By default, the program assumes that there is RAM at `0x20000000`, and that there is 64KB of it. This is correct for the Keil® MCBSTM32 board. To change these values, use the `--ramStart` and `--ramSize` options.

14.14.10 Invoking the Java program

For information on the full set of program arguments, run the program with the `--help` parameter.

There are two ways to invoke the program:

- Specify the DTSL connection properties directly, using the { `--rddiConfigFile` , `--dtslScript`, `--dtslClass`, `--connectionType`, `--connectionAddress`, `--device` } parameters.
- Specify the Arm® Development Studio configdb parameters (equivalent to using the IDE launcher) using the {`--configdb`, `--manufacturer`, `--board`, `--debugOperation`, `--connectionType`, `--connectionAddress` } parameters, and let the program extract the DTSL connection properties from the Arm Development Studio configdb.

14.14.11 About the Java program

We provide a DTSL Eclipse project that contains an example Java program.

- The main program is in the `DTSLExample.java` source file.
- The project is set up to use the DTSL libraries from the DTSL Eclipse project.
- The DTSL interaction flow is as follows:
    1. Connecting to DTSL. This involves forming the *ConnectionParameters* set and passing it to the DTSL static `ConnectionManager.openConnection()` method. See the *connectToDTSL()* method for details.
    2. **Accessing the DTSL connection configuration and locating the DTSL object with the name requested in either:**
        - the `--device` parameter
        - the core specified in the Arm® Development Studio configdb platform debug operation.
    3. Connecting to the core located in step 2.
    4. Performing the operations on the core, which is represented by a DTSL object that implements the *IDevice* interface. The DTSL Javadoc lists the full set of operations available on such an object. The example uses some of the more common operations, but does not cover all of them.
    5. Disconnecting from the core.
    6. Disconnecting from DTSL.
- The *IDevice* Java interface (used for all target devices) wraps the RDDI-DEBUG C interface thinly, which means that many of the RDDI constants are used directly rather than being wrapped. That is why the example uses constants such as `RDDI_ACC_SIZE.RDDI_ACC_DEF`.

15 Reference

Lists other information that might be useful when working with Arm® Debugger.

15.1 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.

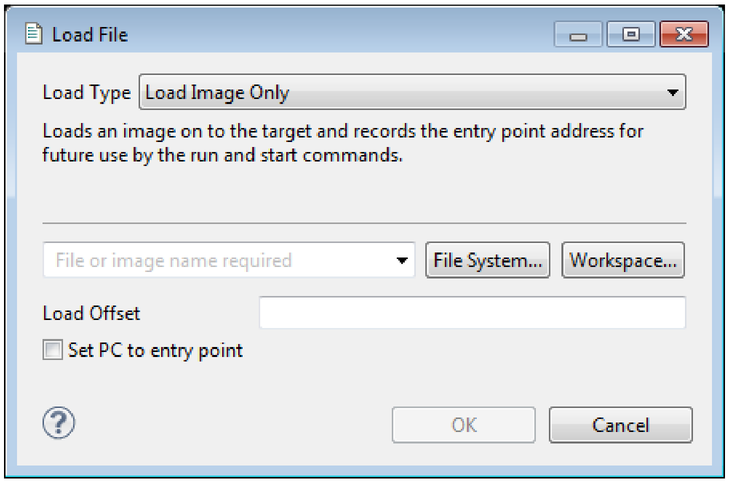
Load Offset

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 15-1: Load File dialog box



Load File

Load Type: Load Image Only

Loads an image on to the target and records the entry point address for future use by the run and start commands.

File or image name required

Load Offset

Set PC to entry point

Related information

About loading debug information into the debugger on page 585
Commands view on page 329
Configuring a connection to an external Fixed Virtual Platform (FVP) for bare-metal application debug
Configuring a connection to a Linux application using gdbserver
Configuring a connection to a Linux kernel
Configuring a connection to a bare-metal hardware target
Configuring an events view connection to a bare metal target
Arm Debugger commands

15.2 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.

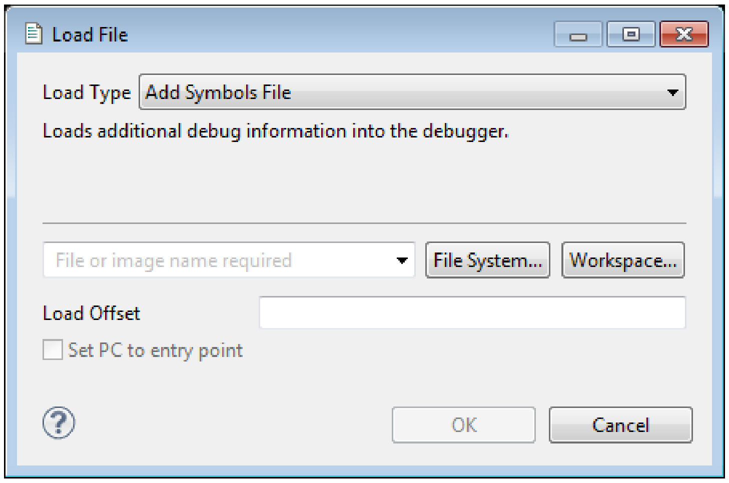
Load Offset
Specifies a decimal or hexadecimal offset that is added to all addresses within the image. A hexadecimal offset must be prefixed with `0x`.

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.

Figure 15-2: Load additional debug information dialog box



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

Related information

About loading an image on to the target on page 584
Commands view on page 329
Configuring the debugger path substitution rules on page 83
Configuring a connection to an external Fixed Virtual Platform (FVP) for bare-metal application debug
Configuring a connection to a Linux application using gdbserver
Configuring a connection to a Linux kernel
Configuring a connection to a bare-metal hardware target
Configuring an events view connection to a bare metal target
Arm Debugger commands

15.3 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 information

Using semihosting to access resources on the host computer on page 80
Working with semihosting on page 82
Debug Configurations - Arguments tab on page 442
Arm Debugger commands

15.4 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 information

Debug Hardware Configure IP view on page 460
Debug Hardware Firmware Installer view on page 462

15.5 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