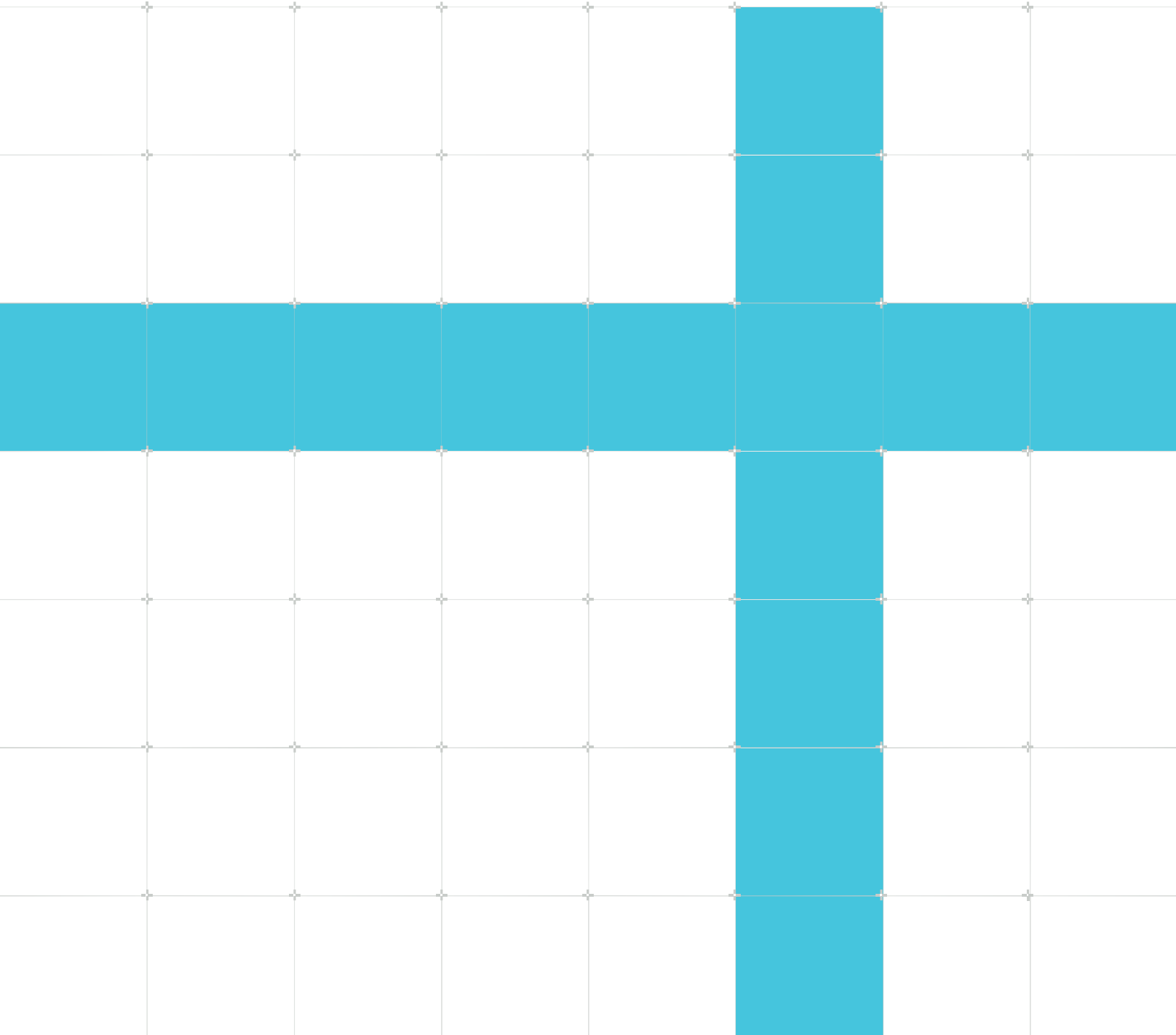
Arm® Development Studio

Version 2022.2

Getting Started Guide

Issue 00
101469_2022.2_00_en



Arm® Development Studio
Getting Started 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
2022.1-00	21 July 2022	Non-Confidential	Updated document for Arm Development Studio 2022.1
2022.2-00	17 November 2022	Non-Confidential	Updated document for Arm Development Studio 2022.2

Proprietary Notice

customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

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.

Previous issues of this document included language that can be offensive. We have replaced this language.

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

Contents

List of Tables

1. Introduction

This book describes how to get started with Arm® Development Studio. It takes you through the processes of installing and licensing Arm Development Studio, and guides you through some of the common tasks that you might encounter when using Arm Development Studio for the first time.

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. 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 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.
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
 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 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
µVision User's Guide	101407	Non-Confidential
Component Architecture Debug Interface User Guide	100963	Non-Confidential
Arm Development Studio Debugger Command Reference	101471	Non-Confidential
Arm Development Studio Heterogeneous system debug with Arm Development Studio	102021	Non-Confidential
Arm Development Studio User Guide	101470	Non-Confidential
Arm DSTREAM-HT Getting Started Guide	101760	Non-Confidential
Arm DSTREAM-HT System and Interface Design Reference Guide	101761	Non-Confidential
Arm DSTREAM-PT Getting Started Guide	101713	Non-Confidential
Arm DSTREAM-PT System and Interface Design Reference Guide	101714	Non-Confidential
Arm DSTREAM-ST Getting Started Guide	100892	Non-Confidential
Arm DSTREAM-ST System and Interface Design Reference Guide	100893	Non-Confidential
Arm DSTREAM-XT Getting Started Guide	102443	Non-Confidential
Arm DSTREAM-XT System and Interface Design Reference Guide	102444	Non-Confidential
CoreSight Components Technical Reference Manual	DDI0314	Non-Confidential

Arm product resources	Document ID	Confidentiality
CoreSight System Trace Macrocell Technical Reference Manual	DDI0444	Non-Confidential
CoreSight Trace Memory Controller Technical Reference Manual	DDI0461	Non-Confidential
Fast Models Fixed Virtual Platforms Reference Guide	100966	Non-Confidential
Iris User Guide	101196	Non-Confidential
User-based Licensing License Server Administration Guide	107573	Non-Confidential
User-based Licensing User Guide	102516	Non-Confidential

Arm® architecture and specifications	Document ID	Confidentiality
ARMv7-M Architecture Reference Manual	DDI0403	Non-Confidential
CoreSight Program Flow Trace Architecture Specification	IHI0035	Non-Confidential

Non-Arm resources	Documentation	Organization
Eclipse documentation	https://help.eclipse.org/	Eclipse Foundation
FTDI driver Installation Guide for Linux	https://www.ftdichip.com/Support/Documents/AppNotes/AN_220_FTDI_Drivers_Installation_Guide_for_Linux.pdf	Future Technology Devices International Limited (FTDI)

1.3 Other information

See the Arm website for other relevant information.

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

2. Introduction to Arm Development Studio

Arm® Development Studio is a professional software development solution for bare-metal embedded systems and Linux-based systems. It covers all stages in development from boot code and kernel porting to application and bare-metal debugging, including performance analysis.

It includes:

The Arm Compiler for Embedded 6 toolchain.
Build embedded and bare-metal embedded applications.

Arm Debugger
A graphical debugger supporting software development on Arm processor-based targets and Fixed Virtual Platform (FVP) targets.

Fixed Virtual Platform (FVP) targets
Single and multi-core simulation models for architectures Armv6-M, Armv7-A/R/M, Armv8-A/R/M, and Armv9-A. These enable you to develop software without any hardware.

Arm Streamline
A graphical performance analysis tool that enables you to transform sampling data and system trace into reports that present data in both visual and statistical forms.

Graphics Analyzer
Graphics Analyzer allows graphics developers to trace OpenGL ES, Vulkan and OpenCL API calls in their applications.

Dedicated examples, applications, and supporting documentation to help you get started with using Arm Development Studio tools.

Some third-party compilers are compatible with Arm Development Studio. For example, the GNU Compiler tools enable you to compile bare-metal, Linux kernel, and Linux applications for Arm targets.

2.1 Arm Compiler for Embedded

The Arm® Compiler for Embedded toolchains enable you to build applications and libraries that are suitable for bare-metal embedded systems.

As part of the download package, Arm Development Studio includes Arm Compiler for Embedded 6 for compiling embedded and bare-metal embedded applications. It supports the Armv6-M, Armv7, Armv8, and Armv9-A architectures.

There are two Arm Compiler toolchains that work with Arm Development Studio; the legacy Arm Compiler 5, and the latest Arm Compiler for Embedded 6. You can run these toolchains in the Arm Development Studio IDE, or from the command line.

- References to Arm Compiler for Embedded in the Arm Development Studio documentation refer to Arm Compiler for Embedded 6, unless otherwise specified.
- Arm Compiler 5 is not included in the Arm Development Studio download package. However, you can download the legacy toolchain from Arm Compiler 5 Downloads. To install, see Add a compiler to Arm Development Studio.


Note

- The features available to you in Arm Compiler for Embedded depend on your individual license type.

  For example, a license might:

  - Limit the use of Arm Compiler for Embedded to specific processors.
  - Place a maximum limit on the size of images that can be produced.

  You can enable additional features of Arm Compiler for Embedded by purchasing a license for the full Arm Development Studio suite. Contact your tools supplier for details.

Related information

Register a compiler toolchain on page 38

2.2 Arm Debugger

Arm® Debugger is accessible using either the Arm Development Studio IDE or command-line, and supports software development on Arm processor-based targets and Fixed Virtual Platform (FVP) targets.

Using Arm Debugger through the IDE allows you to debug bare-metal and Linux applications with comprehensive and intuitive views, including:

- Synchronized source and disassembly.
- Call stack.
- Memory.
- Registers.
- Expressions.
- Variables.
- Threads.
- Breakpoints.
- Trace.

The **Debug Control** view enables you to single-step through applications at source-level or instruction-level, and see other views update when the code is executed. Setting breakpoints or watchpoints stops the application and allows you to explore the behavior of the application. You

can also use the view to trace function executions in your application with a timeline showing the sequence of events, if supported by the target.

You can also debug using the **Arm DS Command Prompt** command-line console, which allows for automation of debug and trace activities through scripts.

Related information

Debug control view
Overview of Arm Debugger

2.3 Debug probes

Arm® Development Studio supports various debug adapters and connections.

Debug adapters

Debug adapters vary in complexity and capability. When you use them with Arm Development Studio, they provide high-level debug functionality, for example:

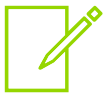
- Reading/writing registers
- Setting breakpoints
- Reading from memory
- Writing to memory

Supported Arm debug adapters include:

- Arm DSTREAM
- Arm DSTREAM-ST
- Arm DSTREAM-PT
- Arm DSTREAM-HT
- Arm DSTREAM-XT
- Keil® ULINK™2
- Keil ULINKpro™
- Keil ULINKpro D
- Keil ULINK-Plus

Supported third-party debug adapters include:

- 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.

Debug connections

Debug connections allow the debugger to debug a variety of targets.

Supported debug connections include:

- CADI (debug interface for models)
- Iris interface for models
- Ethernet to gdbserver
- CMSIS-DAP

Debug hardware configuration

Use the debug hardware configuration views in Arm Development Studio to update and configure the debug hardware adapter that provides the interface between your development target and your workstation.

Arm Development Studio provides the following views:

- **Debug Hardware Config IP view**
  Use this view to configure the IP address on a debug hardware adapter.
- **Debug Hardware Firmware Installer view**
  Use this view to update the firmware on a debug hardware adapter.


Note

These views only support the DSTREAM family of devices.

2.4 FVP models

Fixed Virtual Platforms (FVPs) are complete simulations of an Arm system, including processor, memory and peripherals. FVP targets give you a comprehensive model on which to build and test your software, from the view of a programmer.

When using an FVP, absolute timing accuracy is sacrificed to achieve fast simulated execution speed. This means that you can use a model for confirming software functionality, but you must not rely on the accuracy of cycle counts, low-level component interactions, or other hardware-specific behavior.

Arm® Development Studio provides several FVPs, covering a range of processors in the Cortex® family. You can also connect to a variety of other Arm and third-party simulation models that implement the Iris interface for debug and trace, or the deprecated Component Architecture Debug Interface (CADI).

Related information

The Iris User Guide
Introduction to the Component Architecture Debug Interface (CADI)

2.5 Arm Streamline

Arm® Streamline is a graphical performance analysis tool. It enables you to transform sampling data, instruction trace, and system trace into reports that present the data in both visual and statistical forms.

Arm Streamline uses hardware performance counters with kernel metrics to provide an accurate representation of system resources.

Related information

Streamline documentation

2.6 Graphics Analyzer

Graphics Analyzer is a tool to help OpenGL ES, EGL, OpenCL, and Vulkan developers get the best out of their applications through analysis at the API level.

Graphics Analyzer allows developers to trace OpenGL ES, Vulkan and OpenCL API calls in their application and understand frame-by-frame the effect on the application to help identify possible issues. Attempted misuse of the API is highlighted, as are recommendations for improvement on a Mali™-based system. Trace information may also be captured to a file on one system and be analyzed later. The state of the underlying GPU subsystem is observable at any point.

Related information

Graphics Analyzer

3. Installing and configuring Arm Development Studio

Arm® Development Studio is available for Windows and Linux operating systems. This chapter describes installation requirements, the installation process, and how to configure Arm Development Studio.

3.1 Hardware and host platform requirements

For the best experience with Arm® Development Studio, your hardware and host platform should meet the minimum requirements.

Hardware requirements

To install and use Arm Development Studio, your workstation must have at least:

- A dual core x86 2GHz processor (or equivalent).
- 2GB of RAM.
- Approximately 3GB of hard disk space.

To improve performance, Arm recommends a minimum of 4GB of RAM when you:

- Debug large images.
- Use models with large simulated memory maps.
- Use Arm Streamline.

Host platform requirements

Arm Development Studio supports the following host platforms:

- Windows 10
- Red Hat Enterprise Linux 7 Workstation
- Ubuntu Desktop Edition 18.04 LTS
- Ubuntu Desktop Edition 20.04 LTS

Arm Development Studio only supports 64-bit host platforms.

Arm Compiler for Embedded host platform requirements

Arm Development Studio contains the latest version of Arm Compiler for Embedded 6 that was available at the time your version of Arm Development Studio was released. The release note provides information on host platform compatibility:

- Arm Compiler for Embedded 6

For information on adding other versions of Arm Compiler to Arm Development Studio, including Arm Compiler 5, see register a compiler toolchain.

3.2 Debug system requirements

When debugging bare-metal and Linux targets, you need additional software and hardware.

Bare-metal requirements

You require a debug unit to connect bare-metal targets to Arm® Development Studio. For a list of supported debug units, see Debug Probes.

Linux application and Linux kernel requirements

Linux application debug requires gdbserver version 7.0 or later on your target.

In addition to gdbserver, certain architecture and debug features have minimum Linux kernel version requirements. This is shown in the following table:

Table 3-1: Linux kernel version requirements

Architecture or debug feature	Minimum Arm Linux kernel version
Debug with Arm Debugger	2.6.28
Application debug on Symmetric MultiProcessing (SMP) systems	2.6.36
Access VFP and Arm® Neon® registers	2.6.30
Arm Streamline	3.4

Managing firmware updates

- For DSTREAM, use the debug hardware firmware installer view to check the firmware and update it if necessary. Updated firmware is available in `<install_directory>/sw/debughw/firmware`.
- To use ULINK™2 debug probe with Arm Debugger, you must upgrade with CMSIS-DAP compatible firmware. On Windows, the `UL2_Upgrade.exe` program can upgrade your ULINK2 unit. The program and instructions are available in `<install_directory>/sw/debughw/ULINK2`.
- For ULINKpro™ and ULINKpro D, Arm Development Studio manages the firmware installation.

3.3 Installing on Windows

There are two ways to install Arm® Development Studio, you can use either the installation wizard, or the command line.

You can install multiple versions of Arm Development Studio on Windows platforms. To do this, you must use different root installation directories.

3.3.1 Using the installation wizard

To install Arm® Development Studio on Windows using the installation wizard, use the following procedure.

Before you begin

- Download the Arm Development Studio installation package.

Procedure

1. Unzip the downloaded `.zip` file.
2. Run **armds-<version>.exe** from this location. This opens the Arm Development Studio setup wizard.
3. Follow the on-screen instructions.

- During installation, you might be prompted to install device drivers. Arm recommends that you install these drivers. They allow USB connections to DSTREAM and Energy Probe hardware units. They also support networking for the simulation models. These drivers are required to use these features.
- When the drivers are installed, you might see some warnings about driver software. You can safely ignore these warnings.

3.3.2 Using the command line

To install Arm® Development Studio on Windows using the command line, use the following procedure.

Before you begin

- Download the Arm Development Studio installation package.
- You must have admin privileges on your machine to install from the command line.

Procedure

1. Open the command prompt, with administrative privileges.

2. Run the Microsoft installer, `msiexec.exe`.


Note

- You must provide the location of the `.msi` file as an argument to **msiexec**.
- To display a full list of **msiexec** options, run **msiexec /?** from the command line.

Example 3-1: Example using msiexec

An example of how to install Arm Development Studio using **msiexec** is:

```
msiexec.exe /i <installer_locationdatainstall.msi> EULA=1 /qn /l*v install.log
```

Table 3-2: Definitions of the commands in the msiexec example

Command	Definition
`/i`	Performs the installation.
`<installer_locationdatainstall.msi>`	Specifies the full path name of the `.msi` file to install.
`/EULA=1`	This is an Arm-specific option. Set EULA to 1 to accept the End User License Agreement (EULA). You must read the EULA before accepting it on the command line. This can be found in the GUI installer, the installation files, or on the Arm Development Studio downloads page.
`/qn`	Specifies quiet mode; installation does not require user interaction. **Note:** Device driver installation requires user interaction. If you do not require USB drivers, or if you want the installation to avoid user interaction for USB drivers, use the `SKIP_DRIVERS=1` option on the command line.
`/l*v<install.log>`	Specifies the log file to display all outputs from the installation.

3.4 Installing on Linux

Install Arm® Development Studio on Linux using the installation package provided on the Arm developer website.

Before you begin

Download the Linux installation package from the Arm Developer website.

About this task


Note

You can install multiple versions of Arm Development Studio on Linux platforms. To do this, you must use different root installation directories.

Procedure

Run `armds-<version>.sh` and follow the on-screen instructions.

During the installation, Arm Development Studio automatically runs a dependency check and produces a list of missing libraries. You can safely continue with the installation. Arm recommends that you install these libraries before using Arm Development Studio.

You can find more details and a full list of required libraries in Additional Linux libraries.

Arm recommends that you run the post install setup scripts during the installation process.

Next steps

To use the post install setup scripts after installation, with root privileges, run:

```
run_post_install_for_Arm_DS_IDE_<version>.sh
```

This script is in the install directory.

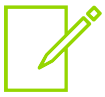
Device drivers and desktop shortcuts are optional features that are installed by this script. The device drivers allow USB connections to debug hardware units, for example, the DSTREAM family. The desktop menu is created using the http://www.freedesktop.org/ menu system on supported Linux platforms.

Use `suite_exec` to configure the environment variables correctly for Arm Development Studio. For example, run `<install_directory>/bin/suite_exec <shell>` to open a shell with the PATH and other environment variables correctly configured. Run `suite_exec` with no arguments for more help.

3.5 Additional Linux libraries

To install Arm® Development Studio on Linux, you need to install some additional libraries, which might not be installed on your system.

The specific libraries that require installation depend on the distribution of Linux that you are running. The `dependency_check_linux-x86_64.sh` script identifies libraries you must install. This script is in `<install_location>/sw/dependency_check`.


Note

If the required libraries are not installed, some of the Arm Development Studio tools might fail to run. You might encounter error messages, such as:

- armcc: No such file or directory
- arm-linux-gnueabihf-gcc: error while loading shared libraries: libstdc++.so.6: cannot open shared object file: No such file or directory

Required libraries

Arm Development Studio depends on the following libraries:

- `libasound.so.2`
- `libatk-1.0.so.0`
- `libc.so.6` *
- `libcairo.so.2`
- `libfontconfig.so.1`
- `libfreetype.so.6`
- `libgcc_s.so.1` *
- `libGL.so.1`
- `libGLU.so.1`
- `libgthread-2.0.so.0`
- `libgtk-x11-2.0.so.0`
- `libncurses.so.5`
- `libnsl.so.1`
- `libstdc++.so.6` *
- `libusb-0.1.so.4`
- `libX11.so.6`
- `libXext.so.6`
- `libXi.so.6`
- `libXrender.so.1`
- `libXt.so.6`
- `libXtst.so.6`
- `libz.so.1` *


Note

On a 64-bit installation, libraries marked with an asterisk require an additional 32-bit compatibility library. Tools installed by the 64-bit installer have dependencies on 32-bit system libraries. Arm Development Studio tools might fail to run, or might report errors about missing libraries if 32-bit compatibility libraries are not installed.

Some components also render using a browser library. Arm recommends that you install one of these libraries to ensure all components render correctly:

- `libwebkit-1.0.so.2`
- `libwebkitgtk-1.0.so.0`
- `libxpcom.so`

3.6 Uninstalling Arm Development Studio on Linux

Arm® Development Studio is not installed with a package manager. To uninstall Arm Development Studio on Linux, you must delete the installation directory. You might also need to delete additional configuration files manually.

Procedure

1. Locate your Arm Development Studio installation directory.
2. If you ran the optional post-install step during or after installation, up to three additional configuration files are created outside of the install directory. Delete the following files if they are present:
   - `/etc/udev/rules.d/ARM_debug_tools.rules`
   - `/etc/hotplug/usb/armdebugtools`
   - `/etc/hotplug/usb/armdebugtools.usermap`
3. If you installed the optional desktop shortcuts during or after installation, you can also remove them:
   a) Locate the Arm Development Studio installation directory.
   b) Run the following script: `remove_menus_for_Arm_DevelopmentStudio_<version>.sh`.
4. Delete the Arm Development Studio installation directory.

Related information

Installing on Linux on page 24

3.7 Licensing Arm Development Studio

Arm® Development Studio uses Arm user-based licensing or FlexNet license management software to enable features that correspond to specific editions.

To compare Arm Development Studio editions, see Compare editions

3.7.1 Add a license using Product Setup

When you first open Arm® Development Studio, the **Product Setup** dialog box opens and prompts you to add a license.

Before you begin

- If you or your company has purchased Arm Development Studio, you need one of the following:
  - Arm user-based licensing:

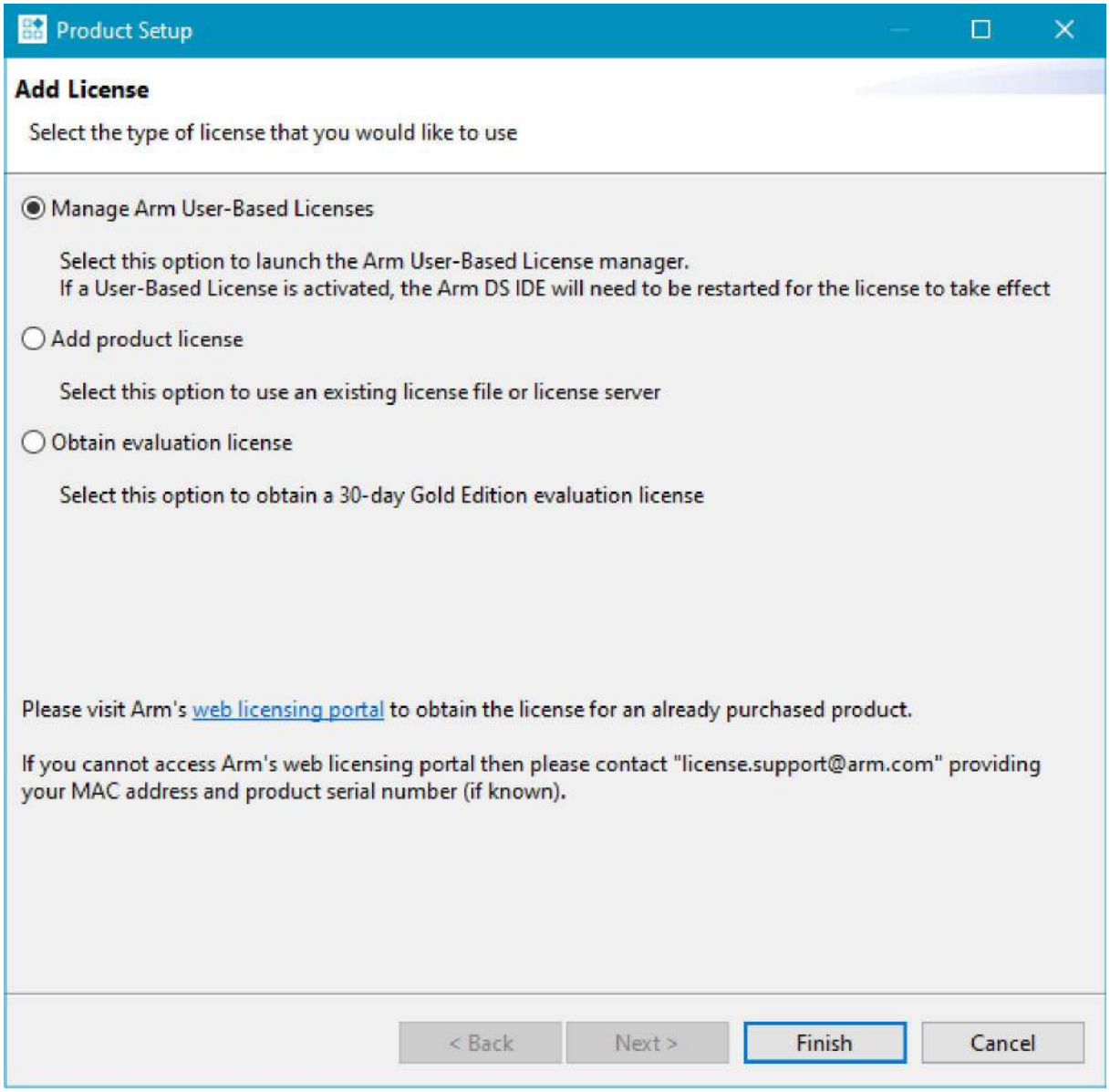
    The license server address or an activation code.
  - FlexNet license management:

    The license file or the license server address and port number.
- To obtain an evaluation license, you need an Arm account.

Procedure

1. Add your license:

Figure 3-1: Product Setup dialog box shown when you first open Arm Development Studio.



- For Arm user-based licensing, select **Manage Arm User-Based Licenses**, and then click **Finish** to open the **Arm License Management Utility** dialog box.

Note

Arm user-based licensing is only available to customers with a user-based licensing license. Documentation for user-based licensing is available at https://lm.arm.com. For assistance with user-based licensing issues, visit https://developer.arm.com/support and open a support case.

- For a FlexNet license server, select **Add product license**, and click **Next**. Enter the license server address and port number, in the form `<port number>` @ `<server address>`. Click **Next**.

- For a FlexNet license file, select **Add product license**, and click **Next**. Click **Browse...** and select the license file. Click **Next**.
- For an evaluation license:
    a. Select **Obtain evaluation license** and click **Next**.
    b. Log into your Arm account and click **Next**.
    c. Choose a network interface and click **Finish**. An evaluation license is generated.

2. Select a product to activate, and click **Finish**.

3.7.2 Viewing and managing licenses

To view license information in Arm® Development Studio, select **Help > Arm License Manager**.

3.7.2.1 Add a license using the Arm License Manager

Use the Arm License Manager to add a license to Arm® Development Studio.

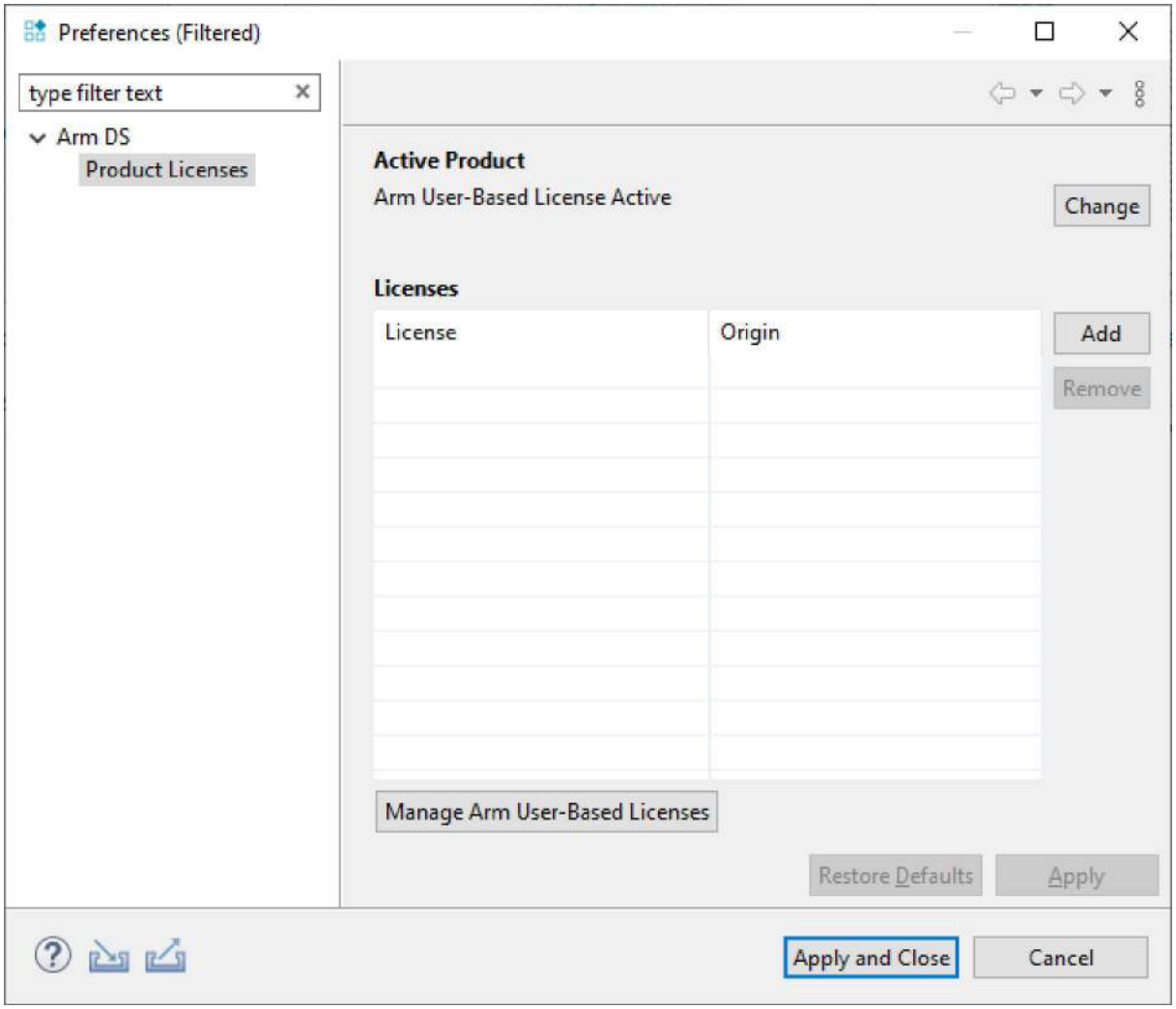
Before you begin

- If you are using Arm user-based licensing, you require the license server address or an activation code.
- If you are using FlexNet license management, you require the license file or the license server address and port number.

Procedure

1. Click **Help > Arm License Manager...** to view your license information.

Figure 3-2: Adding a license in preferences dialog box.



2. Use one of the following methods to add your license:
   - Click **Add** to open the **Product Setup** wizard, and follow the steps in Using Product Setup to add a license to add your license.
   - Click **Manage Arm User-Based Licenses**, to open the **Arm License Management Utility** dialog box.

> **Note**
> Arm user-based licensing is only available to customers with a user-based licensing license. Documentation for user-based licensing is available at https://lm.arm.com. For assistance with user-based licensing issues, visit https://developer.arm.com/support and open a support case.

3. Click **Apply and Close** to save.

Related information

Add a license using Product Setup on page 27

3.7.2.2 Delete a FlexNet license

You can use the Arm license manager to delete unwanted FlexNet licenses from Arm® Development Studio.

About this task

If you are using user-based licensing, there is generally no requirement to delete a license, as a user can use the license for multiple products on multiple devices. To delete a license, for example, if the local license cache is corrupt, follow the instructions in the User-based Licensing User Guide.

Procedure

1. Click **Help > Arm License Manager** to view your license information.
2. Select the license you want to delete, and click **Remove**.

3.8 Language settings

Only Japanese language packs are currently supported by Arm® Development Studio. These language packs are installed with Arm Development Studio.

Procedure

Launch the IDE in Japanese using one of the following methods:

- If your operating system locale is set as Japanese, the IDE automatically displays the translated features.
- If your operating system locale is not set as Japanese, you must specify the `-nl` command-line argument when launching the IDE:

```
armds_ide -nl ja
```

Arm Compiler for Embedded 6 does not support Japanese characters in source files.

3.9 Configuring an RSE connection to work with an Arm Linux target

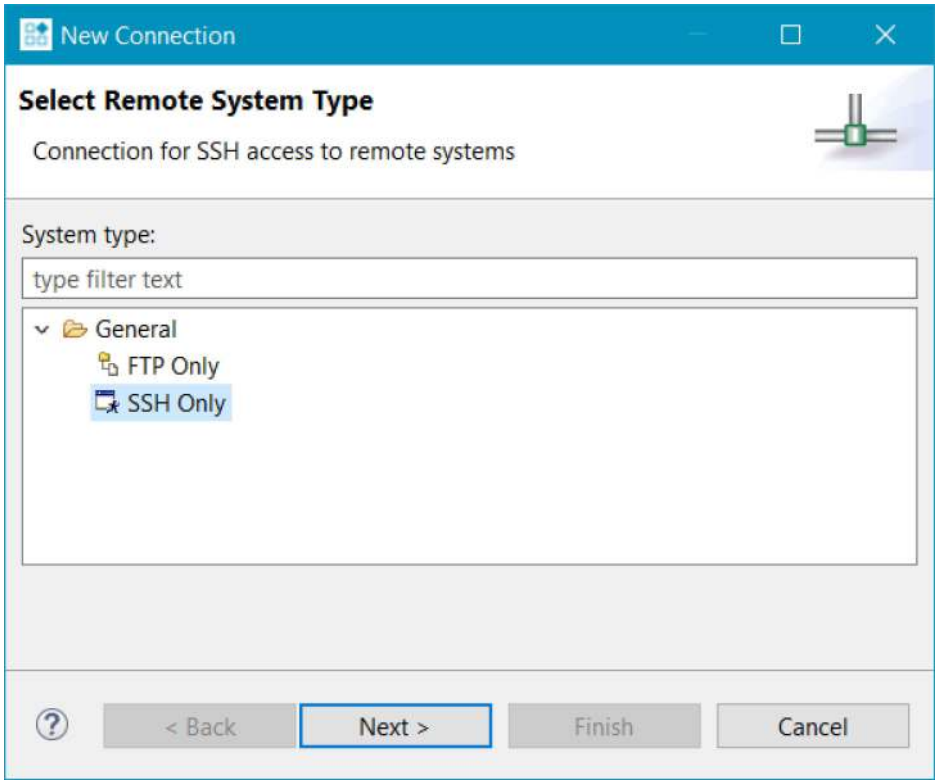
On some targets, you can use a SecureSHell (SSH) connection with the Remote System Explorer (RSE) provided with Arm® Development Studio.

Procedure

1. In the **Remote Systems** view, click the **Define a connection to remote system** option on the toolbar.

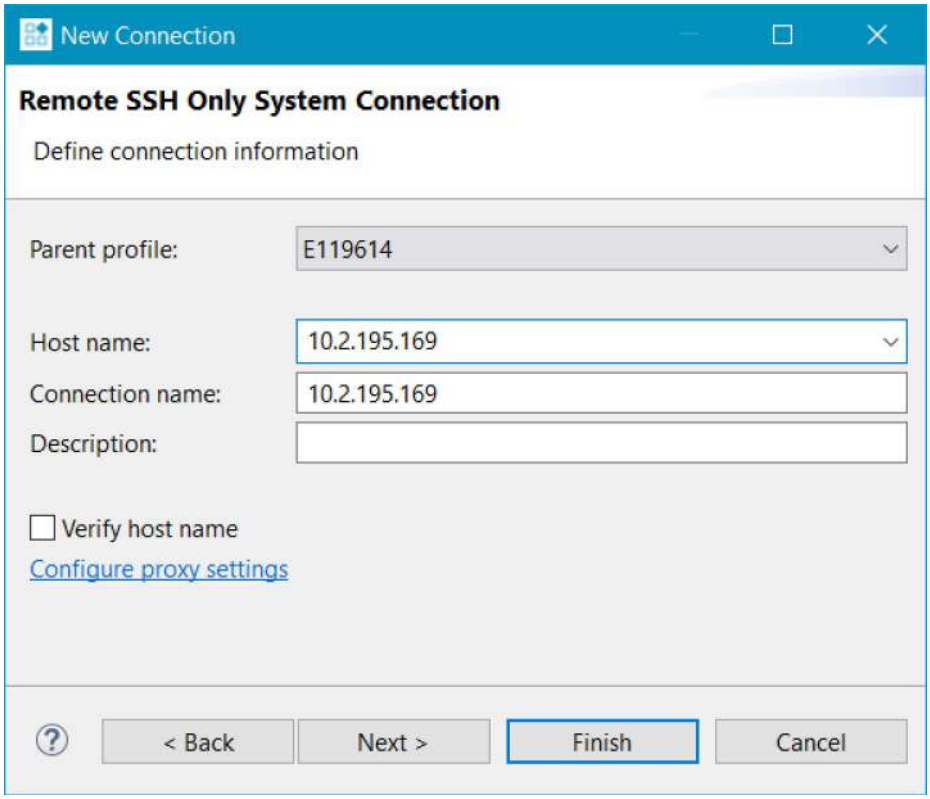
2. In the **Select Remote System Type** dialog box, expand the **General** group and select **SSH Only**.

Figure 3-3: Selecting a connection type



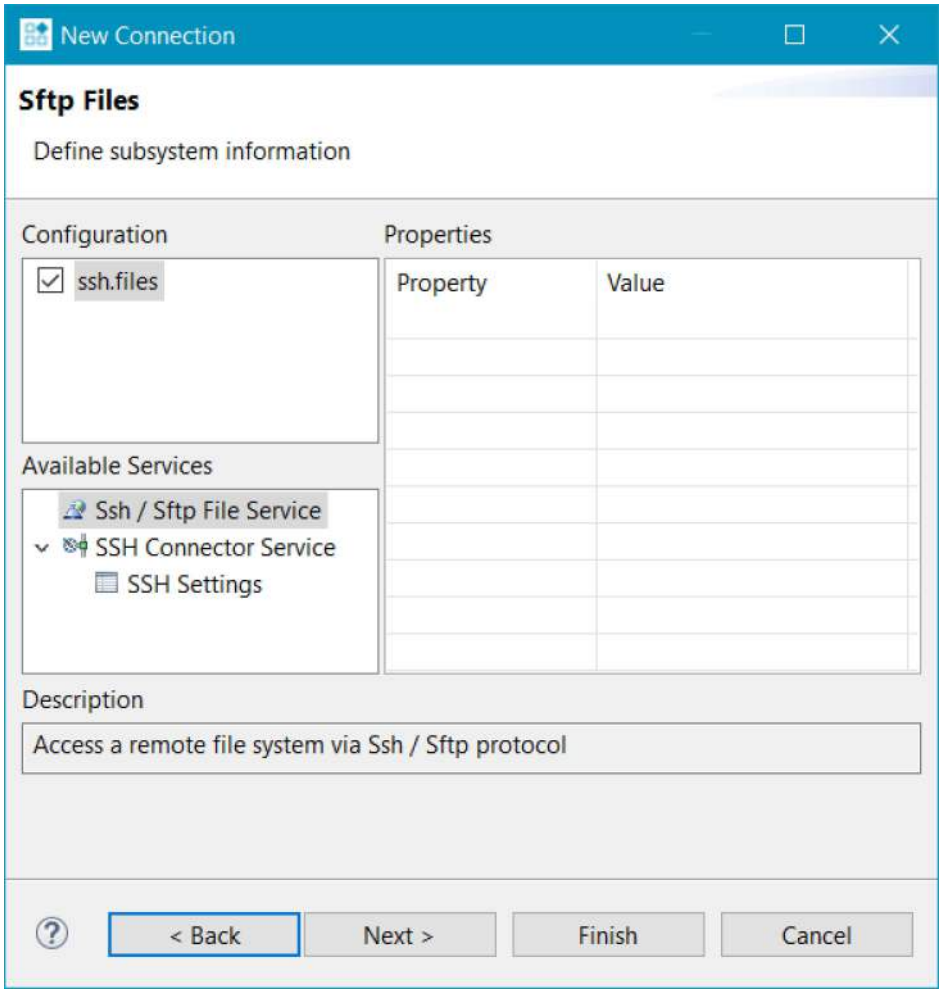
3. Click **Next**.
4. In **Remote SSH Only System Connection**, enter the remote target IP address or name in the **Host name** field.

Figure 3-4: Enter connection information



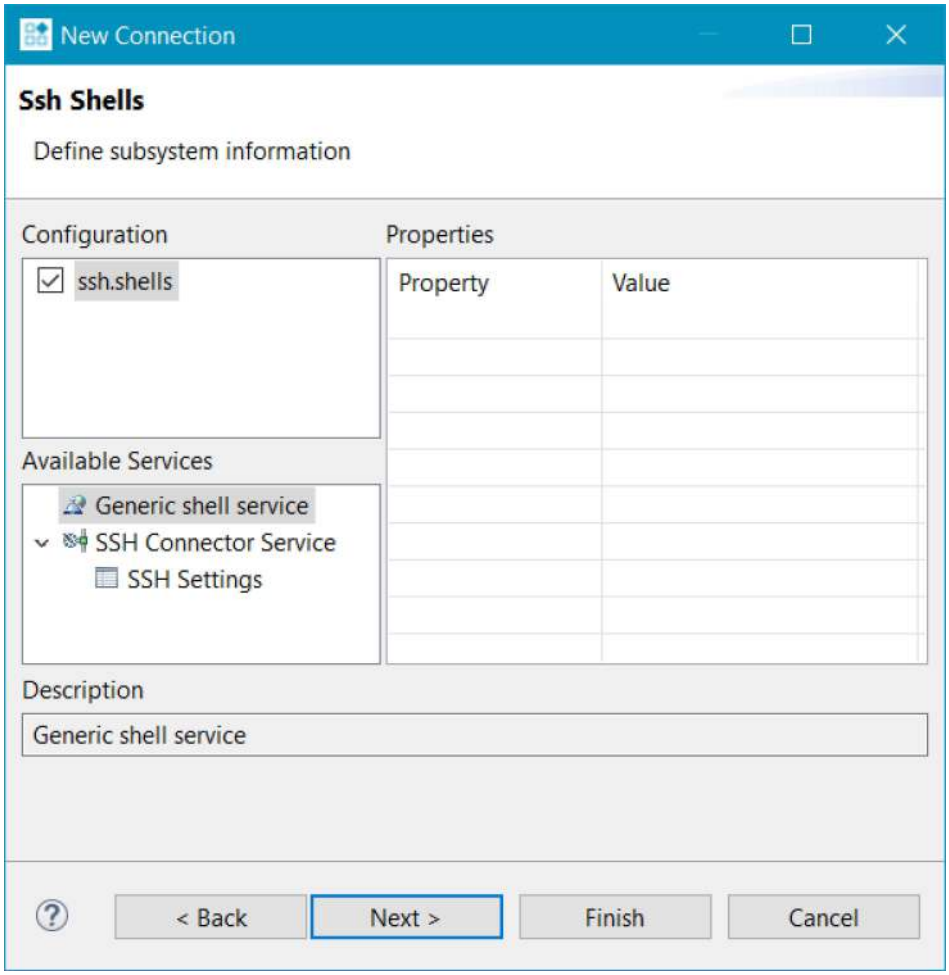
5. Click **Next**.
6. Verify if the **Sftp Files**, **Configuration**, and **Available Services** are what you require.

Figure 3-5: Sftp Files options



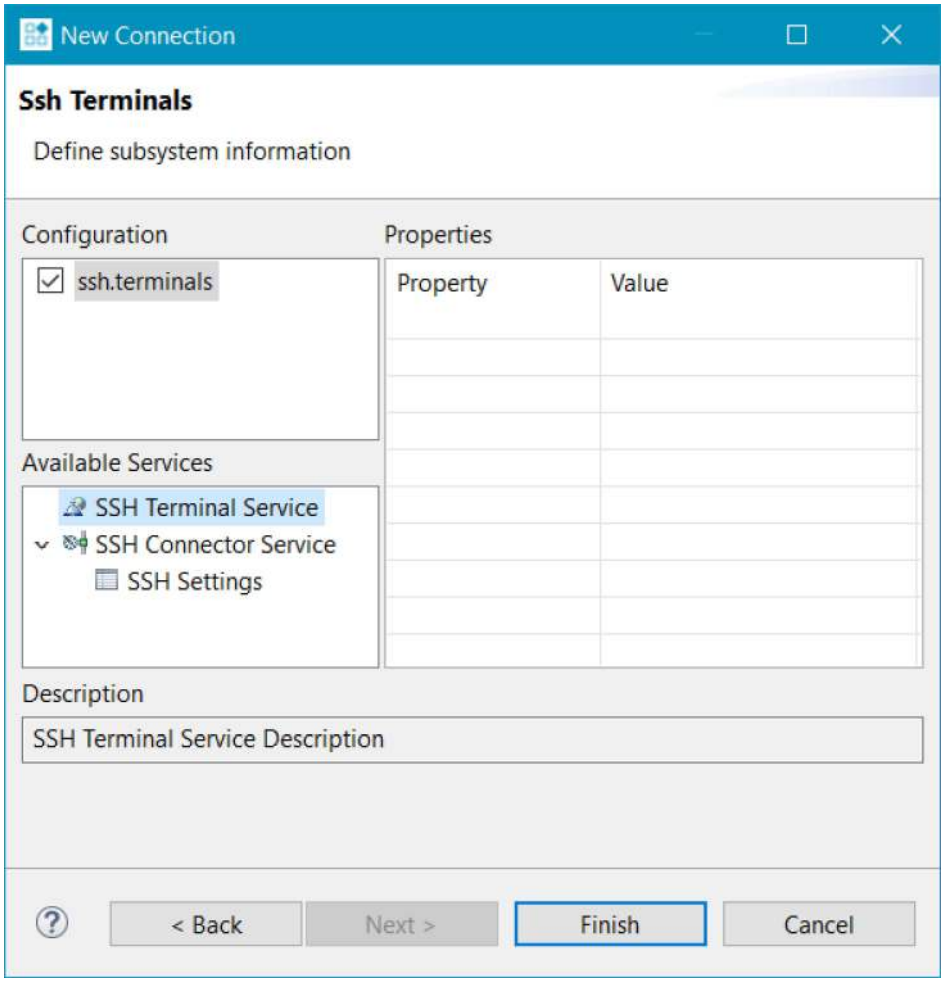
7. Click **Next**.
8. Verify if the **Ssh Shells**, **Configuration**, and **Available Services** are what you require.

Figure 3-6: Defining the shell services



9. Click **Next**.
10. Verify if the **Ssh Terminals**, **Configuration**, and **Available Services** are what you require.

Figure 3-7: Defining the terminal services



11. Click **Finish**.
12. In the **Remote Systems** view:
    a) Right-click on the target and select **Connect** from the context menu.
    b) In the **Enter Password** dialog box, enter a **UserID** and **Password** if required.
    c) Click **OK** to close the dialog box.

Results

Your SSH connection is now set up. You can copy any required files from the local file system on to the target file system. You can do this by dragging and dropping the relevant files into the **Remote Systems** view.

Related information

Import the example projects on page 98
Debug Configurations - Connection tab
Debug Configurations - Files tab
Debug Configurations - Debugger tab
Debug Configurations - Environment tab

Target management terminal for serial and SSH connections
Remote Systems view

3.10 Launching gdbserver with an application

Describes how to launch **gdbserver** with an application.

Procedure

1. Open a terminal shell that is connected to the target.
2. In the **Remote Systems** view, right-click on **Ssh Terminals**.
3. Select **Launch Terminal** to open a terminal shell.
4. In the terminal shell, navigate to the directory where you copied the application, then execute the required commands.

Example 3-2: Example: Launch Gnometris

The following example shows the commands used to launch the Gnometris application.

```
export DISPLAY=ip:0.0
gdbserver :port gnometris
```

Where:

ip
is the IP address of the host to display the Gnometris application.

port
is the connection port between `gdbserver` and the application, for example `5000`.

If the target has a display connected to it, you do not need to use the `export DISPLAY` command.

3.11 Register a compiler toolchain

You can use a different compiler toolchain other than the one installed with Arm® Development Studio.

If you want to build projects using a toolchain that is not installed with Arm Development Studio, you must first register the toolchain you want to use. You can register toolchains:

- Using the Preferences dialog box in Arm Development Studio.
- Using the **add_toolchain** utility from the Arm Development Studio Command Prompt.

You might want to register a compiler toolchain if:

- You want to use a GCC toolchain, or another Arm compiler such as Arm Compiler 5, that is not included in the Arm Development Studio installation.
- You upgrade your version of Arm Development Studio but you want to use an earlier version of the toolchain that was previously installed.
- You install a newer version or older version of the toolchain without re-installing Arm Development Studio.

A variety of other compiler toolchains are available. To find other compiler toolchains, you can do the following:

- Navigate to Arm Compiler for Embedded downloads for the latest Arm Compiler for Embedded toolchain.
- Download a GCC toolchain from Linaro.
- Download the GNU Arm Embedded toolchain for Arm processors.
- If you are using Arm Development Studio 2021.1 or later, and want to use Arm Compiler 5, you can download it from https://developer.arm.com/tools-and-software/embedded/arm-compiler/arm-compiler-5/downloads.

When you register a toolchain, the toolchain is available for new and existing projects in Arm Development Studio.

You can only register Arm or GCC toolchains.

3.11.1 Registering a compiler toolchain using the Arm Development Studio IDE

You can register compiler toolchains using the **Preferences** dialog box in Arm® Development Studio.

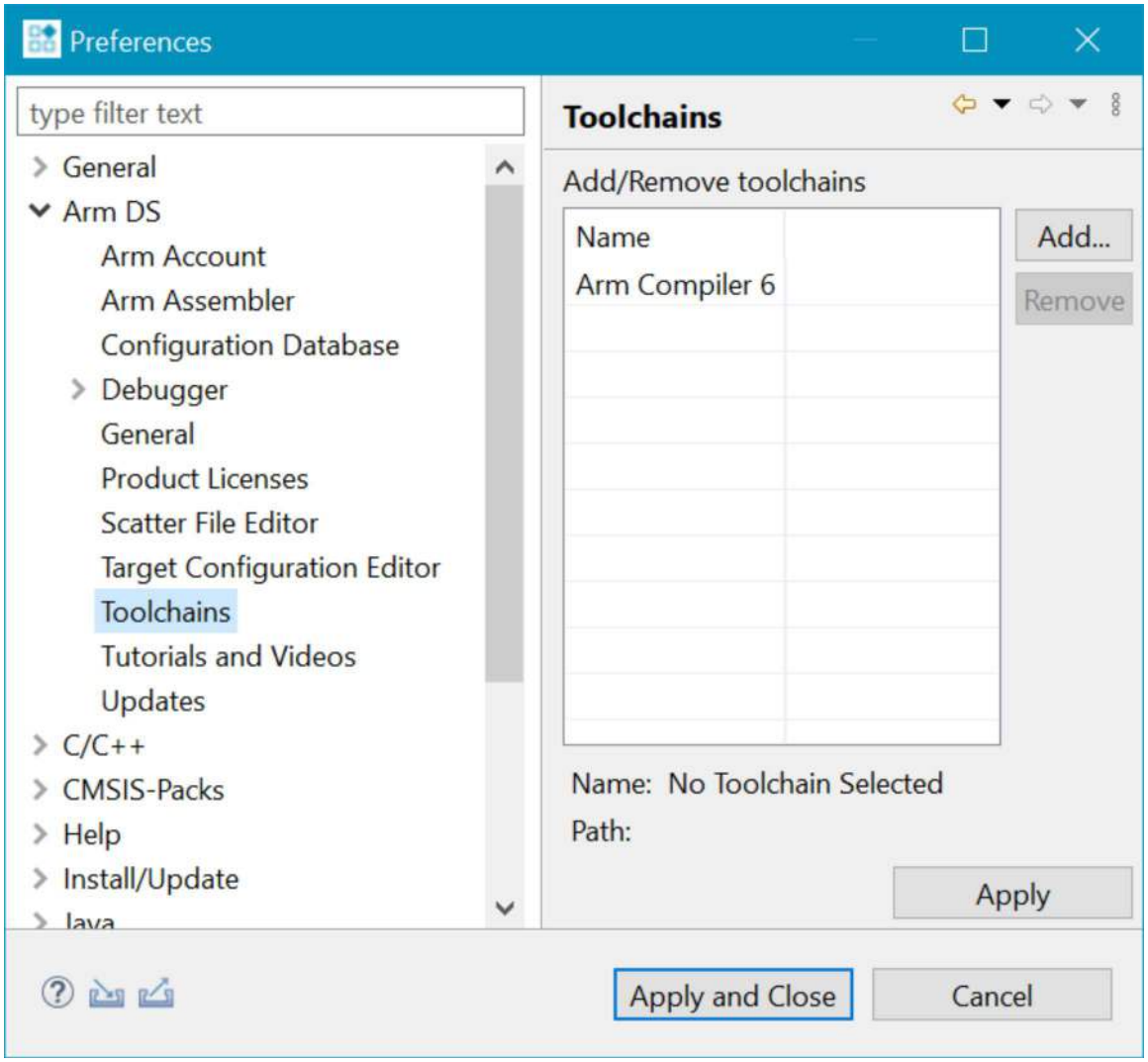
Before you begin

- Download an Arm Compiler for Embedded or GCC toolchain.

Procedure

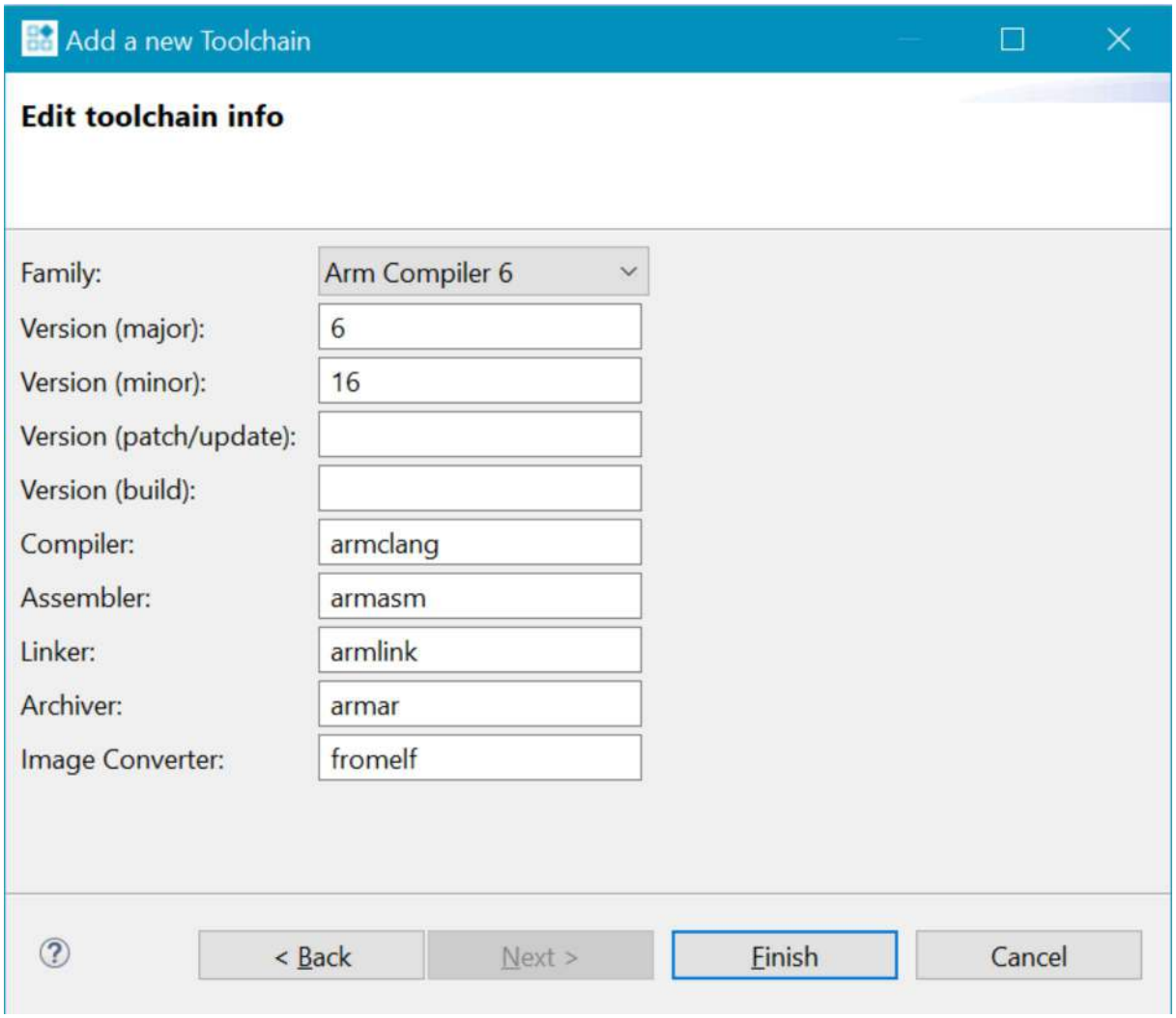
1. Open the Toolchains tab in the Preferences dialog box; **Windows > Preferences > Arm DS > Toolchains**. Here, you can see the compiler toolchains that Arm DS currently recognizes,

Figure 3-8: Toolchains Preferences dialog box



2. Click **Add** and enter the filepath to the toolchain binaries that you want to use. Then click **Next** to autodetect the toolchain properties.
3. After the toolchain properties are autodetected, click **Finish** to register the toolchain. Alternatively, click **Next** to manually enter or change the toolchain properties, and then click **Finish**.

Figure 3-9: Properties for the new toolchain



Note

You must manually enter the toolchain properties if:

- The toolchain properties were not autodetected.
- The family, major version, and minor version of the new toolchain are identical to a toolchain that Arm DS already knows about.

4. In the **Preferences** dialog box, click **Apply**.
5. Restart Arm Development Studio.

Results

- The new toolchain is registered with Arm Development Studio.
- When you create a new project, Arm DS shows the new toolchain in the available list of toolchains.

Related information

Reconfigure existing projects to use a newly registered compiler toolchain on page 43

3.11.2 Register a compiler toolchain using the Arm DS command prompt

Use the `add_toolchain` utility from the command prompt to register a new Arm® Compiler for Embedded or GCC toolchain.

Before you begin

- Download an Arm Compiler for Embedded or GCC toolchain.

Procedure

1. Open the Arm DS <version> Command Prompt, and enter `add_toolchain <path>`, where `<path>` is the directory containing the toolchain binaries. The utility automatically detects the toolchain properties.

   Note

   By default, the `add_toolchain` utility is an interactive tool. To use the `add_toochain` utility as a non-interactive tool, add the `--non-interactive` option to the command.

   For example, on Windows: `add_toolchain "C:\Program Files (x86)\ARM_Compiler_5.06u7\bin64" --non-interactive`

2. The utility prompts whether you want to register the toolchain with the details it has detected. If you want to change the details, the utility prompts for the details of the toolchain.
3. Restart Arm Development Studio. You must do this before you can use the toolchain in the Arm DS environment.

   Note

   - The toolchain target only applies to GCC toolchains. It indicates what target platform the GCC toolchain builds for. For example, if your compiler toolchain binary is named `arm-linux-gnueabihf-gcc`, then the target name is the prefix `arm-linux-gnueabihf`. The target field allows Arm DS to distinguish different toolchains that otherwise have the same version.
   - You must manually enter the toolchain properties if:
     - The toolchain properties were not autodetected.
     - The type, major version, and minor version of the new toolchain are identical to a toolchain that Arm DS already knows about.

Results

- The new toolchain is registered with Arm Development Studio.
- When you create a new project, Arm DS shows the new toolchain in the available list of toolchains.

Related information

Reconfigure existing projects to use a newly registered compiler toolchain on page 43

3.11.3 Reconfigure existing projects to use a newly registered compiler toolchain

When you register a new compiler toolchain in Arm® Development Studio, you can reconfigure existing projects to use the newly registered toolchain.

Before you begin

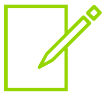
Register an Arm Compiler for Embedded or GCC toolchain. You can use the IDE or the Arm DS command prompt.

Procedure

1. Select a new compiler toolchain to use with your project.
   a) In the **Project Explorer** view, right-click your project and select **Properties > C/C++ Build > Tool Chain Editor**.
   b) Select the new toolchain under the **Current toolchain** drop-down menu.
   c) Click **Apply and Close**.
2. After you change the toolchain, clean and rebuild the project.
   a) In the **Project Explorer** view, select the project, right-click it and select **Clean Project**.
   b) In the **Project Explorer** view, select the project, right-click it and select **Build Project**.

3.11.4 Configure a compiler toolchain for the Arm DS command prompt

When you want to compile or build from the Arm DS command prompt, you must select the compiler toolchain you want to use. You can either specify a default toolchain, so that you do not need to select a toolchain every time you start the Arm DS command prompt, or you can specify a toolchain for the current session only.



By default, the Arm DS command prompt is not configured with a compiler toolchain.

3.11.4.1 Configure a compiler toolchain for the Arm DS command prompt on Linux

Describes how to specify a compiler toolchain using the Linux command-line utility.

Procedure

1. To set a default compiler toolchain, run `<install_directory>/bin/select_default_toolchain` and follow the instructions.
2. To specify a compiler toolchain for the current session, run `<install_directory>/bin/suite_exec --toolchain <toolchain_name>`

To list the available toolchains, run `suite_exec` with no arguments.

If you specify a toolchain using the `suite_exec --toolchain` command, it overwrites the default compiler toolchain for the current session.

Example 3-3: Example

To use the Arm® Compiler for Embedded toolchain in the current session, run:

```
<install_directory>/bin/suite_exec --toolchain "Arm Compiler for Embedded 6" bash --norc
```

3.11.4.2 Configure a compiler toolchain for the Arm DS command prompt on Windows

Describes how to specify a compiler toolchain using the Arm DS Command Prompt.

Procedure

1. To set a default compiler toolchain:
   a) Select **Start > All Programs > Arm DS Command Prompt**.
   b) To see the available compiler toolchains, enter `select_default_toolchain`.
   c) From the list of available toolchains, select your default compiler toolchain.
2. To specify a compiler toolchain for the current session:
   a) Select **Start > All Programs > Arm DS Command Prompt**.
   b) To see the available compiler toolchains, enter `select_toolchain`.

Using this command overwrites the default compiler toolchain for the current session.

   c) From the list of available toolchains, select the one that you want to use for this session.

3.12 Specify plug-in install location

By default, Arm® Development Studio installs plug-ins into the user's home area. You can override the default settings so that the plug-ins are installed into the Arm DS installation directory. Plug-ins available in the Arm DS installation directory are available to all users of the host workstation.

Before you begin

- Installation of Arm Development Studio with appropriate licenses applied.
- Access to your Arm Development Studio install.

About this task

You override the default Arm Development Studio configuration location using the Eclipse `vmargs` runtime option. The Eclipse `vmargs` runtime option allows you to customize the operation of the Java VM to run Eclipse. See the Eclipse runtime options documentation for more information about the Eclipse `vmargs` runtime option.

Procedure

1. At your operating system command prompt, enter: `<armds_install_directory>/bin/armds_ide -vmargs -Dosgi.configuration.area=<install_directory/sw/ide/configuration> -Dosgi.configuration.cascaded=false`.

   
   Note

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

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

2. Install your Eclipse plug-in using your preferred plug-in installation option, for example, the Eclipse Marketplace.
3. Restart Arm Development Studio when prompted to do so.

Results

Your plug-ins are now installed into the Arm Development Studio `<install_directory/sw/ide/configuration>` directory and are available to all users of the host workstation.

3.13 Development Studio perspective keyboard shortcuts

You can use various keyboard shortcuts in the Development Studio perspective.

You can access the dynamic help in any view or dialog box by using the following:

- On Windows, use the **F1** key
- On Linux, use the **Shift+F1** key combination.

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

Commands view

You can use:

Ctrl+Space

Access the content assist for autocompletion of commands.

Enter

Execute the command that is entered in the adjacent field.

DOWN arrow

Navigate down through the command history.

UP arrow

Navigate up through the command history.

Debug Control view

You can use:

F5

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

ALT+F5

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

F6

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

F7

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

F8

Continue running the target.

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

F9

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

Related information

Commands view

4. Introduction to Arm Debugger

Introduces Arm® Debugger and some important debugger concepts.

4.1 Overview: Arm Debugger and important concepts

Arm® Debugger is part of Arm Development Studio and helps you find the cause of software bugs on Arm processor-based targets and Fixed Virtual Platform (FVP) targets.

From device bring-up to application debug, it can be used to develop code on an RTL simulator, virtual platform, and hardware, to help get your products to market quickly.

Arm Debugger supports:

- Loading images and symbols.
- Running images.
- Breakpoints and watchpoints.
- Source and instruction level stepping.
- CoreSight™ and non-CoreSight trace (Embedded Trace Macrocell Architecture Specification v3.0 and above).
- Accessing variables and register values.
- Viewing the contents of memory.
- Navigating the call stack.
- Handling exceptions and Linux signals.
- Debugging bare-metal code.
- Debugging multi-threaded Linux applications.
- Debugging the Linux kernel and Linux kernel modules.
- Debugging multicore and multi-cluster systems, including big.LITTLE™.
- Debugging Real-Time Operating Systems (RTOSs).
- Debugging from the command-line.
- Performance analysis using Arm Streamline.
- A comprehensive set of debugger commands that can be executed in the Eclipse Integrated Development Environment (IDE), script files, or a command-line console.
- GDB debugger commands, making the transition from open source tools easier.
- A small subset of third party CMM-style commands sufficient for running target initialization scripts.

Using Arm Debugger, you can debug bare-metal and Linux applications with comprehensive and intuitive views, including synchronized source and disassembly, call stack, memory, registers, expressions, variables, threads, breakpoints, and trace.

4.2 Debugger concepts

Lists some of the useful concepts to be aware of when working with Arm® Debugger.

AMP

Asymmetric Multi-Processing (AMP) system has multiple processors that may be different architectures. See Debugging AMP Systems for more information.

Bare-metal

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

BBB

The old name for the MTB.

CADI

Component Architecture Debug Interface. This is the API used by debuggers to control models.

Configuration database

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

The database exists as a series of `.xml` files, python scripts, `.rvc` files, `.rcf` files, `.sdf` files, and other miscellaneous files in the `<installation_directory>/sw/debugger/configdb/` directory.

Arm Development Studio comes pre-configured with support for a wide variety of devices out-of-the-box, and you can view these in the **Debug Configuration** dialog box in the Arm Development Studio IDE.

You can also add support for your own devices using the Platform Configuration Editor (PCE) tool.

Contexts

Each processor in the target can run more than one process. However, the processor only executes one process at any given time. Each process uses values stored in variables, registers, and other memory locations. These values can change during the execution of the process.

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

The context changes when:

- A function is called.

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

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

CTI

The Cross Trigger Interface (CTI) combines and maps trigger requests, and broadcasts them to all other interfaces on the Embedded Cross Trigger (ECT) sub-system. See Cross-trigger configuration for more information.

DAP

The Debug Access Port (DAP) is a control and access component that enables debug access to the complete SoC through system ports. See About the Debug Access Port for more information.

Debugger

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

Debug agent

A debug agent is hardware or software, or both, that enables a host debugger to interact with a target. For example, a debug agent enables you to read from and write to registers, read from and write to memory, set breakpoints, download programs, run and single-step programs, program flash memory, and so on.

`gdbserver` is an example of a software debug agent.

Debug session

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

Debug target

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

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

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


Note

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

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

Debug adapter

A debug adapter is a physical interface between the host debugger and hardware target. It acts as a debug agent. A debug adapter is normally required for bare-metal start/stop debugging real target hardware, for example, using JTAG.

Examples include DSTREAM, DSTREAM-ST, and the ULINK family of debug and trace adapters.

DSTREAM

The Arm DSTREAM family of debug and trace units. For more information, see: DSTREAM family

Arm Development Studio supports the Arm DSTREAM debug unit, but it is discontinued and no longer available to purchase.

DTSL

Debug and Trace Services Layer (DTSL) is a software layer in the Arm Debugger stack. DTSL is implemented as a set of Java classes which are typically implemented (and possibly extended) by Jython scripts. A typical DTSL instance is a combination of Java and Jython. Arm has made DTSL available for your own use so that you can create programs (Java or Jython) to access/control the target platform.

DWARF

DWARF is a debugging format used to describe programs in C and other similar programming languages. It is most widely associated with the ELF object format but it has been used with other object file formats.

ELF

Executable and Linkable Format (ELF) is a common standard file format for executables, object code, shared libraries, and core dumps.

ETB

Embedded Trace Buffer (ETB) is an optional on-chip buffer that stores trace data from different trace sources. You can use a debugger to retrieve captured trace data.

ETF

Embedded Trace FIFO (ETF) is a trace buffer that uses a dedicated SRAM as either a circular capture buffer, or as a FIFO. The trace stream is captured by an ATB input that can then be output over an ATB output or the Debug APB interface. The ETF is a configuration option of the Trace Memory Controller (TMC).

ETM

Embedded Trace Macrocell (ETM) is an optional debug component that enables reconstruction of program execution. The ETM is designed to be a high-speed, low-power debug tool that supports trace.

ETR

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

FVP

Fixed Virtual Platform (FVP) enables development of software without the requirement for actual hardware. The functional behavior of the FVP is equivalent to real hardware from a programmers view.

ITM

Instruction Trace Macrocell (ITM) is a CoreSight component which delivers code instrumentation output and specific hardware data streams.

jRDDI

The Java API implementation of RDDI.

Jython

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

MTB

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

PTM

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

RDDI

Remote Device Debug Interface (RDDI) is a C-level API which allows access to target debug and trace functionality, typically through a DSTREAM box, or a CADI model.

Scope

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

Variables can have values that are relevant within:

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

SMP

A Symmetric Multi-Processing (SMP) system has multiple processors with the same architecture. See Debugging SMP systems for more information.

STM

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

TPIU

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

TMC

The Trace Memory Controller (TMC) enables you to capture trace using:

- The debug interface such as 2-pin serial wire debug.
- The system memory such as a dynamic Random Access Memory (RAM).
- The high-speed links that already exist in the System-on-Chip (SoC) peripheral.

4.3 Overview: Arm CoreSight debug and trace components

CoreSight™ defines a set of hardware components for Arm®-based SoCs. Arm Debugger uses the CoreSight components in your SoC to provide debug and performance analysis features.

Examples of common CoreSight components include:

- DAP: Debug Access Port
- ECT: Embedded Cross Trigger
- TMC: Trace Memory Controller
  - ETB: Embedded Trace Buffer
  - ETF: Embedded Trace FIFO
  - ETR: Embedded Trace Router
- ETM: Embedded Trace Macrocell
- PTM: Program Trace Macrocell
- ITM: Instrumentation Trace Macrocell
- STM: System Trace Macrocell


Note

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

Examples of how these components are used by Arm Debugger include:

- The **Trace** view displays data collected from PTM and ETM components.
- The **Events** view displays data collected from ITM and STM components.
- Debug connections can make use of the ECT to provide synchronized starting and stopping of groups of cores. For example, you can use the ECT to:
  - Stop all the cores in an SMP group simultaneously.
  - Halt heterogeneous cores simultaneously to allow whole system debug at a particular point in time.

If you are using an SoC that is supported out-of-the-box with Arm Debugger, select the correct platform (SoC) in the **Debug Configuration** dialog box to configure a debug connection. If you are using an SoC that is not supported by Arm Debugger by default, then you must first define a custom platform in Arm Debugger's configuration database using the **Platform Configuration Editor** tool.

For all platforms, whether built-in or manually created, you can use the **Platform Configuration Editor** (PCE) to easily define the debug topology between various components available on the platform. See the Platform Configuration Editor topic for details.

4.4 Overview: Debugging multi-core (SMP and AMP), big.LITTLE, and multi-cluster targets

Arm® Debugger is developed with multicore debug in mind for bare-metal, Linux kernel, or application-level software development.

Awareness for Symmetric Multi-Processing (SMP), Asymmetric Multi-Processing (AMP), and big.LITTLE™ configurations is embedded in Arm Debugger, allowing you to see which core, or cluster a thread is executing on.

When debugging applications in Arm Debugger, multicore configurations such as SMP or big.LITTLE require no special setup process. Arm Debugger includes predefined configurations, backed up by the Platform Configuration Editor which enables further customization. The nature of the connection determines how Arm Debugger behaves, for example stopping and starting all cores simultaneously in a SMP system.

4.4.1 Debugging SMP systems

From the point of view of Arm® Debugger, Symmetric Multi Processing (SMP) refers to a set of architecturally identical cores that are tightly coupled together and used as a single multi-core execution block. Also, from the point of view of the debugger, they must be started and halted together.

Arm Debugger expects an SMP system to meet the following requirements:

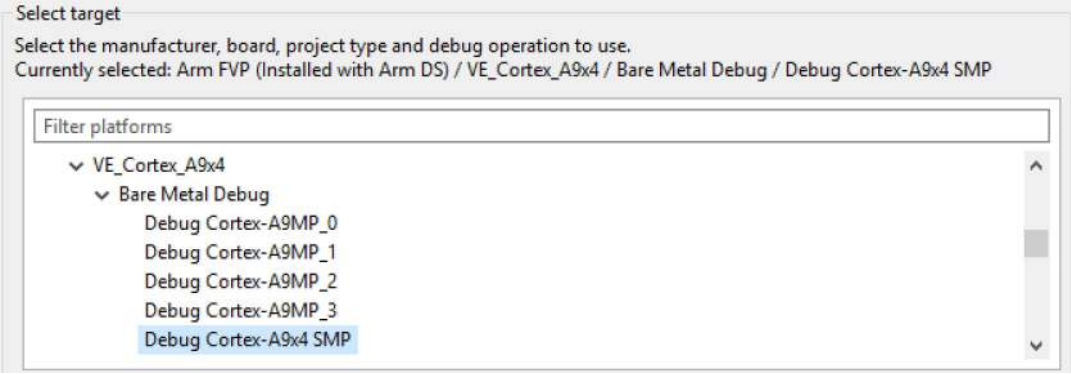
- The same ELF image running on all processors.
- All processors must have identical debug hardware. For example, the number of hardware breakpoint and watchpoint resources must be identical.
- Breakpoints and watchpoints must only be set in regions where all processors have identical physical and virtual memory maps. Processors with separate instances of identical peripherals mapped to the same address are considered to meet this requirement. Private peripherals of Arm multicore processors is a typical example.

Configuring and connecting

To enable SMP support in the debugger, you must first configure a debug session in the **Debug Configurations** dialog box. Configuring a single SMP connection is all that you require to enable SMP support in the debugger.

Targets that support SMP debugging have SMP mentioned against them.

Figure 4-1: Versatile Express A9x4 SMP configuration



Once connected to your target, use the **Debug Control** view to work with all the cores in your SMP system.

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.

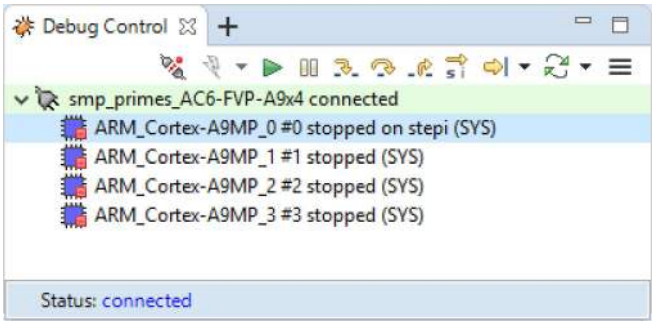
For symbol loading, this means that debug information is loaded once and is available when debugging any of the processors.

Running, stepping, and stopping

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 the other processors in the system stop.

For instruction level single-stepping commands, stepi and nexti, the currently selected processor steps one instruction.

Figure 4-2: Core 0 stopped on step i command



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 different cores running or stopping. This delay can be very large because the debugger must run and stop each core individually. However, hardware cross-trigger implementations in most SMP systems ensure that the delays are minimal and are limited to a few processor clock cycles.

In rare cases, one processor might stop, and one or more of the other processors might not respond. 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 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

If you are using a connection that enables trace support, you can view trace for each of the processors in your system using the **Trace** view.

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.


Note

The indexes in the different **Trace** views do not necessarily represent the same point in time for different processors.

4.4.2 Debugging AMP Systems

From the point of view of Arm® Debugger, Asymmetric Multi Processing (AMP) refers to a set of cores which operate in an uncoupled manner. The cores can be of different architectures or of the same architecture but not operating in an SMP configuration. Also, from the point of view of the debugger, it depends on the implementation whether the cores need to be started or halted together.

An example of this might be a Cortex®-A5 device coupled with a Cortex-M4, combining the benefits of an MCU running an RTOS which provides low-latency interrupt with an application processor running Linux. These are often found in industrial applications where a rich user-interface might need to interact closely with a safety-critical control system, combining multiple cores into an integrated SoC for efficiency gains.

Bare metal debug on AMP Systems

Arm Debugger supports simultaneous debug of the cores in AMP devices. In this case, you need to launch a debugger connection to each one of the cores and clusters in the system. Each one of these connections is treated independently, so images, debug symbols, and OS awareness are kept separate for each connection. For instance, you will normally load an image and its debug symbols for each AMP processor. With multiple debug sessions active, you can compare content in the **Registers**, **Disassembly**, and **Memory** views by opening multiple views and linking them to multiple connections, allowing you to view the state of each processor at the same time.

It is possible to connect to a system in which there is a cluster or big.LITTLE™ subsystem working in SMP mode, for example, running Linux, with extra processors working in AMP mode for example, running their own bare-metal software or an RTOS. Arm Debugger is capable of supporting these devices by just connecting the debugger to each core or subsystem separately.

4.4.3 Debugging big.LITTLE Systems

A big.LITTLE™ system optimizes for 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.

Awareness for big.LITTLE configurations is built into Arm® Debugger, allowing you to establish a bare-metal, Linux kernel, or Linux application debug connection, just as you would for a single core processor.

For the software required to enable big.LITTLE support in your own OS, visit the big.LITTLE Linaro git repository.

Bare-metal debug on big.LITTLE systems

For bare-metal debugging on big.LITTLE systems, you can establish a big.LITTLE connection in Arm Debugger. In this case, all the processors in the big.LITTLE 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. The remaining functionality of the debugger is equivalent to an SMP connection to a homogeneous cluster of cores.

Linux application debug on big.LITTLE systems

For Linux application debugging on big.LITTLE systems, you can establish a `gdbserver` connection in 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. Therefore, there is no difference when debugging a Linux application on a big.LITTLE system as compared to application debug on any other system.

4.5 Overview: Debugging Arm-based Linux applications

Arm® Debugger supports debugging Linux applications and libraries that are written in C, C++, and Arm assembly.

The integrated suite of tools in Arm Development Studio enables rapid development of optimal code for your target device.

For Linux applications, communication between the debugger and the debugged application is achieved using `gdbserver`. See Configuring a connection to a Linux application using gdbserver for more information.

Related information

Configuring a connection to a Linux kernel on page 125
Configuring a connection to a Linux application using gdbserver on page 122

About debugging shared libraries

5. Introduction to the IDE

The Arm® Development Studio Integrated Development Environment (IDE) is Eclipse-based, combining the Eclipse IDE from the Eclipse Foundation with the compilation and debug technology of Arm tools.

It includes:

Project Explorer

The project explorer enables you to perform various project tasks such as adding or removing files and dependencies to projects, importing, exporting, or creating projects, and managing build options.

Editors

Editors enable you to read, write, or modify C/C++ or Arm assembly language source files.

Perspectives and views

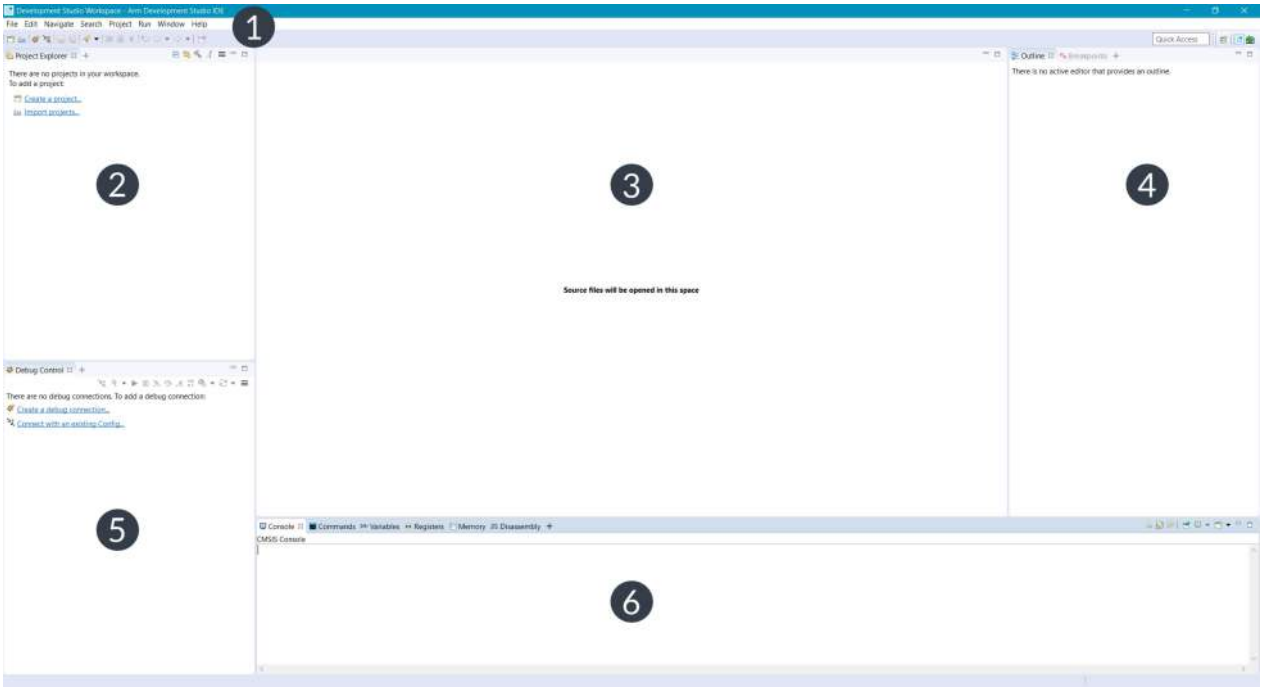
Perspectives provide customized views, menus, and toolbars to suit a particular type of environment. Arm Development Studio uses the **Development Studio** perspective by default. To switch perspectives, from the main menu, select **Window > Perspective > Open Perspective**.

5.1 IDE Overview

The Integrated Development Environment (IDE) contains a collection of views that are associated with a specific perspective.

Arm® Development Studio uses the **Development Studio** perspective as default.

Figure 5-1: IDE in the Development Studio perspective.



1. The main menu and toolbar are both located at the top of the IDE window. Other toolbars, that are associated with specific features, are located at the top of each perspective or view.
2. **Project Explorer** view to create, build, and manage your projects.
3. **Editor** view to inspect and modify the content of your source code files. The tabs in the editor area show the files that are currently open for editing.
4. During a debug session this area typically shows the **Registers** and **Breakpoints** views. You can drag and drop other views into this area.
5. **Debug Control** view to create and control debug connections.
6. During a debug session this area typically shows views that are associated with debug inputs and outputs, such as the **Commands** and **Console** views.

On exit, your settings save automatically. When you next open Arm Development Studio, the window returns to the same perspective and views.

For further information on a view, click inside it and press F1 to open the **Help** view.

Customize the IDE

You can customize the IDE by changing the layout, key bindings, file associations, and color schemes. These settings can be found in **Window > Preferences**. Changes are saved in your workspace. If you select a different workspace, then these settings might be different.

Related information

Using the IDE on page 62
Perspectives in Arm Development Studio

5.2 Using the IDE

The Arm® Development Studio IDE can be customized. It is possible to choose the views you can see by following the instructions in this section.

5.2.1 Changing the default workspace

The workspace is an area on your file system to store files and folders related to your projects, and your IDE settings. When Arm® Development Studio launches for the first time, a default workspace is automatically created for you in `C:\Users\<user>\Development Studio Workspace`.

About this task

Note

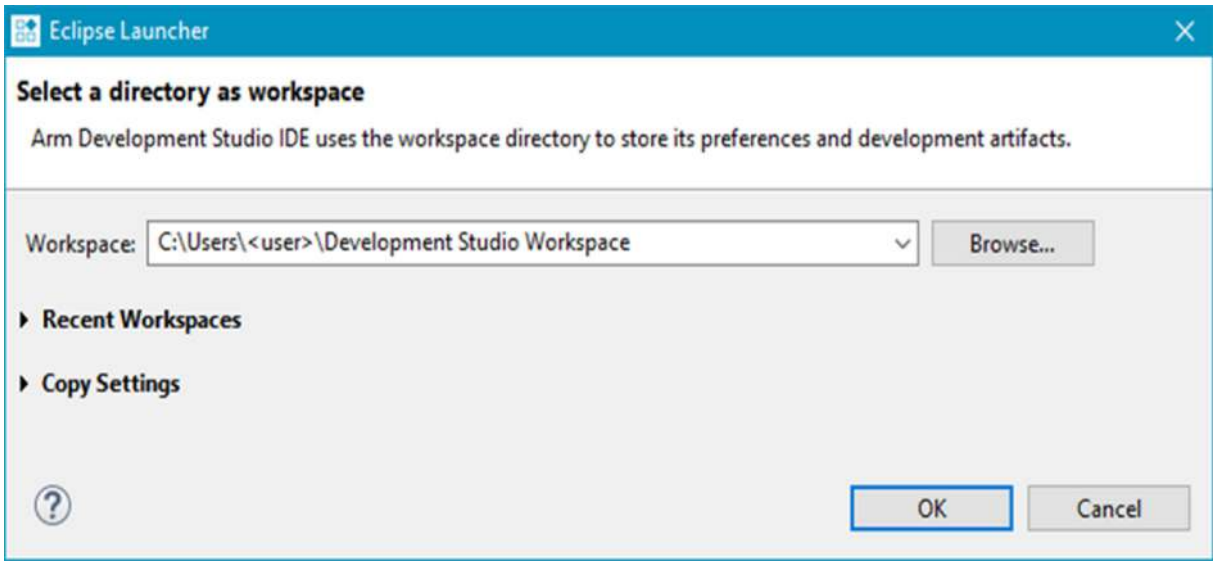
Arm recommends that you select a dedicated workspace folder for your projects. If you select an existing folder containing resources that are not related to your projects, you cannot access them in Arm Development Studio. These resources might also cause a conflict later when you create and build projects.

Arm Development Studio automatically opens in the last used workspace.

Procedure

1. Select **File > Switch Workspace > Other...**. The **Eclipse Launcher** dialog box opens.

   Figure 5-2: Workspace Launcher dialog box



2. Click **Browse...** to choose your workspace, and click **OK**.

Results

Arm Development Studio relaunches in the new workspace.

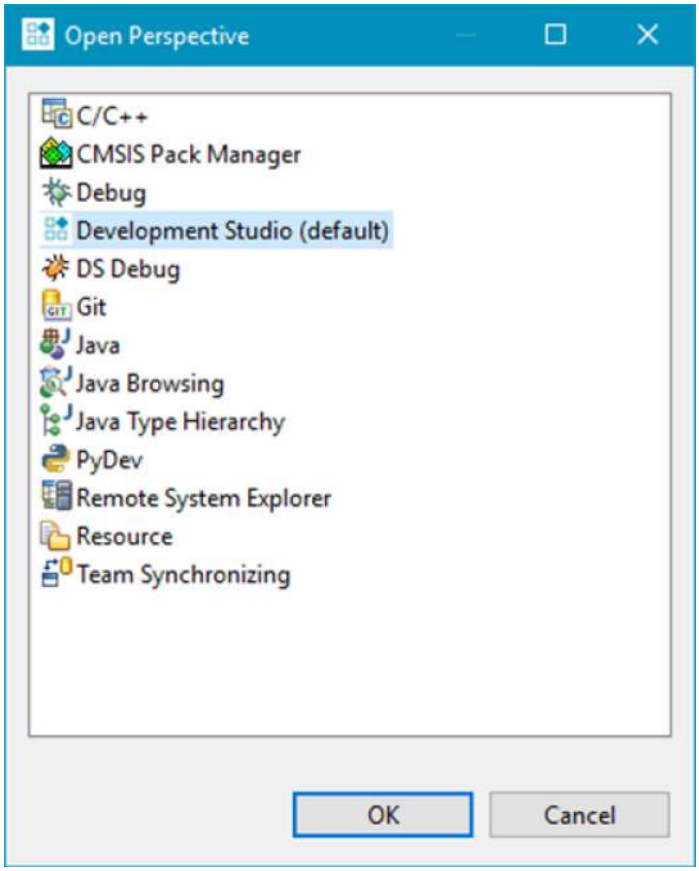
5.2.2 Switching perspectives

Perspectives define the layout of your selected views and editors in the Arm® Development Studio IDE. Each perspective has its own associated menus and toolbars.

Procedure

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 5-3: Open Perspective dialog box



Results

Your perspective opens in the workspace.

Related information

Arm Debugger perspectives and views

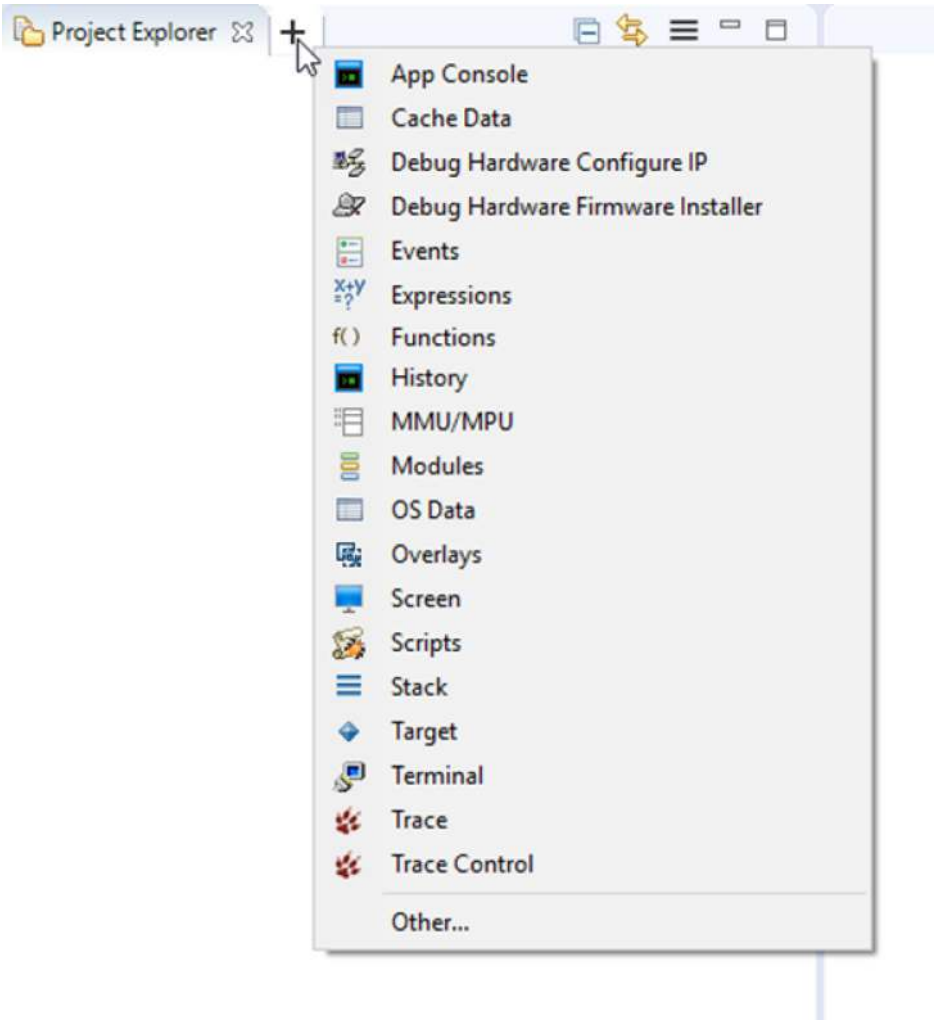
5.2.3 Adding views

Views provide information for a specific function, corresponding to the active debug connection. Each perspective has a set of default views. You can add, remove, or reposition the views to customize your workspace.

Procedure

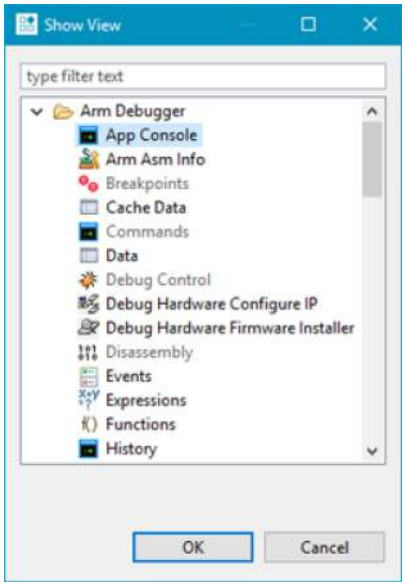
1. Click the **+** button in the area you want to add a view.

   Figure 5-4: Adding a view in an area



2. Choose a view to add, or click **Other...** to open the **Show View** dialog box to see a complete list of available views.

Figure 5-5: Adding a view in Arm Development Studio



3. Select the view you want to open, and click **OK**.

Results
The view opens in the selected area.

Related information
Arm Debugger perspectives and views

5.3 Personalize your development environment

Arm® Development Studio Integrated Development Environment (IDE) has many settings, called **Preferences**, that are available for you to adjust and change. Use these **Preferences** to adapt the IDE to best support your own personal development style.

When you launch Arm Development Studio for the first time, the **Preferences Wizard** takes you through the process of setting up the IDE.

This wizard presents the most commonly changed **Preferences** to customize for your requirements. These include specifying the start-up workspace location, selecting a theme, and tweaking the code editing format.

Note

- If you have upgraded from a previous version of Arm Development Studio and had your workspace preferences already set up, your preferences remain the same.
- These preferences are only saved in the current workspace. 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.

- You can click **Apply and Close** at any point during your wizard. The **Preferences Wizard** applies changes up to where you have modified the options and leaves the rest of the settings as default.
- There are more IDE configuration options in the **Preferences** dialog which allow you to make further in-depth changes to your IDE settings. For example, extra code formatting and syntax highlighting options. To open the **Preferences** dialog, from the main menu, select **Window > Preferences**.
- You can click **Skip** and ignore the **Preferences Wizard** and return to the wizard later to make changes. To restart the wizard later, in the **Preferences** dialog, select **Arm DS > General > Start Preferences Wizard**.
- To disable the **Preferences Wizard** when you launch Arm Development Studio, add `ARM_DS_DISABLE_PREFS_WIZARD` as an environment variable in your operating system.
- When switching Arm Development Studio between the light and dark themes, to apply your selection you must restart Arm Development Studio.

Related information

Installing and configuring Arm Development Studio on page 21
Licensing Arm Development Studio on page 27
Using the IDE on page 62
Language settings on page 32
Preferences dialogue box

5.4 Launch the Arm Development Studio command prompt

To configure the same features of Arm® Development Studio that you can configure through the GUI, you can use the Arm Development Studio command prompt.

About this task

The Arm Development Studio command prompt is useful when:

- You want to run scripts or automate tasks.
- You are more comfortable working with the command line.

You can use the Arm Development Studio command prompt to perform operations such as:

- Registering and configuring a compiler toolchain.
- Selecting and using a compiler.
- Running a model.
- Launching Graphics Analyzer or Arm Streamline.

- Importing, building, and cleaning Eclipse projects and µVision® projects.
- Configuring Arm Debugger.
- Configuring a connection to a built-in Fixed Virtual Platform (FVP).
- Batch updating firmware for the DSTREAM family of products.

Procedure

Launch the command prompt for your system:

On Windows:

- Select **Start > All Programs > Arm Development Studio > Arm Development Studio Command Prompt**.

On Linux:

1. Open a new terminal in your preferred shell.
2. Change directory to the `bin` directory inside your Arm Development Studio installation directory. For example: `cd /opt/arm/developmentstudio-2020.0/bin`.
3. Run `./suite_exec`.

Example 5-1: Example: Arm Development Studio command prompt usage scenarios

- Configure a compiler toolchain

  On Windows:

  - Set a default compiler toolchain:
    1. Follow the procedure to launch the command prompt.
    2. To see the available compiler toolchains, run `select_default_toolchain`.
    3. Select your preferred default compiler toolchain from the available list.
  - Specify a compiler toolchain for the current session:
    1. Follow the procedure to launch the command prompt.
    2. To see the available compiler toolchains, run `select_toolchain`.
    3. Select your preferred compiler toolchain for this session from the available list.

  On Linux:

  - Set a default compiler toolchain:
    1. Follow the procedure to launch the command prompt.
    2. Run `./select_default_toolchain`.
    3. Select your preferred default compiler toolchain from the available list.
  - Set a compiler toolchain for the current session:
    1. Follow the procedure to launch the command prompt.
    2. Run `./suite_exec --toolchain <toolchain_name> <preferred_shell>`.
- Set Arm Compiler for Embedded 6 as your compiler toolchain

On Windows:

1. Follow the procedure to launch the command prompt.
2. To see the available compiler toolchains, run `select_toolchain`.
3. Select Arm Compiler for Embedded 6 from the list.
4. To verify that the environment has been configured correctly, run `armclang --vsn` to see the version information and license details.

On Linux:

1. Follow steps 1 and 2 of the procedure to launch the command prompt.
2. Run `./suite_exec --toolchain "Arm Compiler 6" <preferred_shell>`.
3. To verify that the environment has been configured correctly, run `./armclang --vsn` to see the version information and license details.

- Connect to an Arm FVP Cortex-A9x4 model

  On Windows:

  1. Follow the procedure to launch the command prompt.
  2. Run `armdbg --cdb-entry "Arm FVP::VE_Cortex_A9x4::Bare Metal Debug::Bare Metal Debug::Cortex-A9x4 SMP"`.

  On Linux:

  1. Follow the procedure to launch the command prompt.
  2. Run `./armdbg --cdb-entry "Arm FVP::VE_Cortex_A9x4::Bare Metal Debug::Bare Metal Debug::Cortex-A9x4 SMP"`.

- Connect to an Arm FVP Cortex-A53x1 and specify an image to load

  On Windows:

  1. Follow the procedure to launch the command prompt.
  2. Run `armdbg --cdb-entry "Arm FVP (Installed with Arm DS)::Base_A53x1::Bare Metal Debug::Bare Metal Debug::Cortex-A53" --cdb-entry-param model_params="-C bp.secure_memory=false" --image "C:\<path_to_workspace_folder>\HelloWorld\Debug\HelloWorld.axf"`.

  On Linux:

  1. Follow the procedure to launch the command prompt.
  2. Run `./armdbg --cdb-entry "Arm FVP (Installed with Arm DS)::Base_A53x1::Bare Metal Debug::Bare Metal Debug::Cortex-A53" --cdb-entry-param model_params="-C bp.secure_memory=false" --image "<path_to_workspace_folder>/HelloWorld/Debug/HelloWorld.axf"`.

Related information

Run the Arm Development Studio IDE from the command-line to clean, build, and import projects on page 80
Configuring debug connections in Arm Debugger
Overview: Running Arm Debugger from the command-line or from a script
Configuring a connection from the command-line to a built-in FVP

Register a compiler toolchain using the Arm DS command prompt

5.5 Headless tools in the Arm Development Studio command prompt

Use the Arm® Development Studio command prompt to run features of the Arm Development Studio IDE without the GUI. You might want to do this to automate certain tasks.

The following commands run the Arm Development Studio IDE from the command prompt:

- On Windows: `armds_idec.exe`
- On Linux: `armds_ide`

You must specify a headless application as an argument to the `-application` option.

There are two headless applications provided with Arm Development Studio:

- Use `com.arm.cmsis.pack.project.headlessbuild` to clean, build, and import Eclipse projects.
- Use `com.arm.cmsis.pack.uv.headlessuvimport` to clean, build, and import µVision® projects.

You can also specify the following options as necessary:

Table 5-1: Arm DS IDE options

Option	Description
`-nosplash`	Disables the Arm Development Studio IDE splash screen.
`--launcher.suppressErrors`	Causes errors to be printed to the console instead of being reported in a graphical dialog box.
`-data <workspaceDir>`	Specify the location of your workspace.
`-import <projectDir>[/projectName.uvprojx]`	Import the project from the specified directory into your workspace. If you are using `com.arm.cmsis.pack.uv.headlessuvimport` to import a µVision project, you must specify the project file here. Use this option multiple times to import multiple projects.
`-build <projectName>[/<configName>] | all`	Build the project with the specified name, or all projects in your workspace. By default, this option builds all the configurations in each project. You can limit this action to a single configuration, such as `Debug` or `Release`, by specifying the configuration name immediately after your project name, separated with '/'. Use this option multiple times to build multiple projects.

Option	Description
`-cleanBuild <projectName>[/<configName>] | all`	Clean and build the project with the specified name, or all projects in your workspace. By default, this option cleans and builds all the configurations in each project. You can limit this action to a single configuration, such as `Debug` or `Release`, by specifying the configuration name immediately after your project name, separated with '/'. Use this option multiple times to clean and build multiple projects.
`-cmsisRoot <path>`	Set the path to the CMSIS Packs root directory
`-help`	Prints the list of available arguments.

Related information

Launch the Arm Development Studio command prompt on page 66
Run the Arm Development Studio IDE from the command-line to clean, build, and import projects on page 80

6. Projects and examples in Arm Development Studio

Describes how to work with projects in Arm® Development Studio. Also lists the example projects we provide, and how to import them into your workspace.

6.1 Working with projects

Projects are top level folders in your workspace that contain related files and sub-folders. A project must exist in your workspace before you add a new file or import an existing file.

6.1.1 Project types

Different project types are provided with Eclipse, depending on the requirements of your project.

Bare metal projects require a software license for Arm® Compiler for Embedded to successfully build an ELF image.

Bare-metal Executable

Uses Arm Compiler for Embedded to build a bare-metal executable ELF image.

Bare-metal Static library

Uses Arm Compiler for Embedded to build a library of ELF object format members for a bare-metal project.

It is not possible to debug or run a stand-alone library file until it is linked into an image.

Executable

Uses the GNU Compilation Tools to build a Linux executable ELF image.

Shared Library

Uses the GNU Compilation Tools to build a dynamic library for a Linux application.

Static library

Uses the GNU Compilation Tools to build a library of ELF object format members for a Linux application.


Note

It is not possible to debug or run a stand-alone library file until it is linked into an image.

Makefile project

Creates a project that requires a makefile to build the project. However, Eclipse does not automatically create a makefile for an empty Makefile project. You can write the makefile yourself or modify and use an existing makefile.


Note

Eclipse does not modify Makefile projects.

Build configurations

By default, the new project wizard provides two separate build configurations:

Debug

The debug target is configured to build output binaries that are fully debuggable, at the expense of optimization. It configures the compiler optimization setting to minimum (level 0), to provide an ideal debug view for code development.

Release

The release target is configured to build output binaries that are highly optimized, at the expense of a poorer debug view. It configures the compiler optimization setting to high (level 3).

In all new projects, the `Debug` configuration is automatically set as the active configuration. You can change this in the C/C++ **Build Settings** panel of the **Project Properties** dialog box.


Note

C project

This does not select a source language by default and leaves this decision up to the compiler. Both GCC and Arm Compiler for Embedded default to C for `.c` files and C++ for `.cpp` files.

C++ project

Selects C++ as the source language by default, regardless of file extension.

In both cases, the source language for the entire project a source directory, or individual source file can be configured in the build configuration settings.

6.1.2 Create a new C or C++ project

Create a new C or C++ project in Arm® Development Studio.

Procedure

1. Select **File > New > Project...** from the main menu.
2. Expand the **C/C++** group, select either **C Project** or **C++ Project**, and click **Next**.

C project

This does not select a source language by default and leaves this decision up to the compiler. Both GCC and Arm Compiler for Embedded default to C for `.c` files and C++ for `.cpp` files.

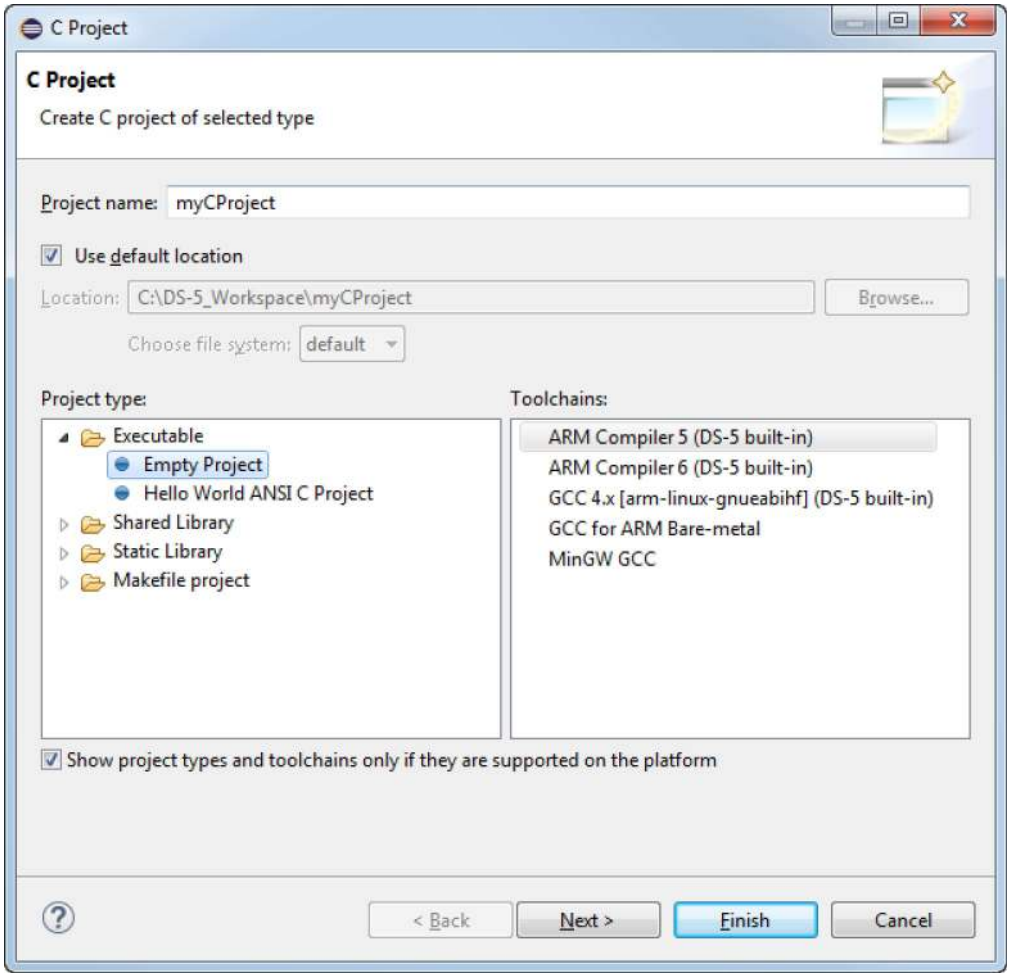
C++ project

Selects C++ as the source language by default, regardless of file extension.

In both cases, the source language for the entire project, a source directory or individual source file can be configured in the build configuration settings.

3. Enter a **Project name**.
4. Leave the **Use default location** option selected so that the project is created in the default folder shown. Alternatively, deselect this option and browse to your preferred project folder.
5. Select the type of project that you want to create.

Figure 6-1: Creating a new C project



6. Select a **Toolchain**.
7. Click **Finish** to create your new project.

Results

You can view the project in the **Project Explorer** view.

6.1.3 Creating an empty Makefile project

Describes how to create an empty C or C++ Makefile project for an Arm Linux target:

Procedure

1. Create a new project:
   a) Select **File > New > Project...** from the main menu.
   b) Expand the **C/C++** group, select either **C Project** or **C++ Project**, and click **Next**.
   c) Enter a project name.

d) Leave the **Use default location** option selected so that the project is created in the default folder shown. Alternatively, deselect this option and browse to your preferred project folder.
e) Expand the **Makefile project** group.
f) Select **Empty project** in the **Project type** panel.
g) Select the toolchain that you want to use when building your project. If your project is for an Arm Linux target, select the appropriate GCC toolchain. You might need to download a GCC toolchain if you have not done so already.
h) Click **Finish** to create your new project. The project is visible in the **Project Explorer** view.

2. Create a Makefile, and then edit:
   a) Before you can build the project, you must have a `Makefile` that contains the compilation tool settings. The easiest way to create one is to copy the `Makefile` from the example project, `hello` and paste it into your new project. The `hello` project is in the Linux examples provided with Arm® Development Studio.
   b) Locate the line that contains `OBJS = hello.o`.
   c) Replace `hello.o` with the names of the object files corresponding to your source files.
   d) Locate the line that contains `TARGET =hello`.
   e) Replace `hello` with the name of the target image file corresponding to your source files.
   f) Save the file.
   g) Right-click the project and then select **Properties > C/C++ Build**. In the **Builder Settings** tab, ensure that the **Build directory** points to the location of the `Makefile`.
3. Add your C/C++ files to the project.

Next steps
Build the project. In the **Project Explorer** view, right-click the project and select **Build Project**.

Related information
Create a new Makefile project with existing code on page 75

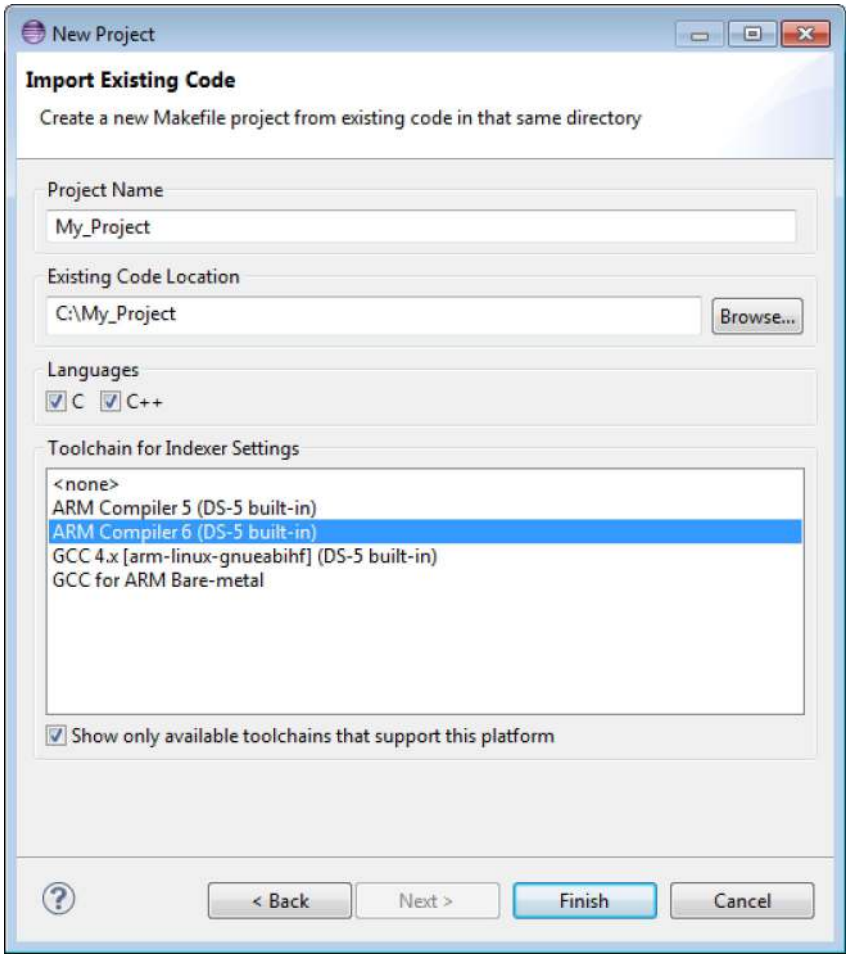
6.1.4 Create a new Makefile project with existing code

You can create a new Makefile project in Arm® Development Studio with your existing source code.

About this task
The following procedure describes how to create a new Makefile project in the same directory as your source code.

Procedure
1. Create a Makefile project:
   a) Select **File > New > Project...** from the main menu.
   b) Expand the **C/C++** group, select **Makefile Project with Existing Code**, and click **Next**.
   c) Enter a project name and enter the location of your existing source code.
   d) Select the toolchain that you want to use for Indexer Settings. Indexer Settings provide source code navigation in the Arm Development Studio IDE.

Figure 6-2: Creating a new Makefile project with existing code



e) Click **Finish** to create your new project. The project and source files are visible in the **Project Explorer** view.

2. Create a Makefile:
   a) Before you can build the project, you need to have a `Makefile` that contains the compilation tool settings. The easiest way to create one is to copy the `Makefile` from an example project, and paste it into your new project.
   b) Edit the `Makefile` for your new project.
   c) Right-click the project and then select **Properties > C/C++ Build** to access the build settings. In the **Builder Settings** tab, check that the **Build directory** points to the location of the `Makefile`.
3. Add any other source files you need to the project.
4. Build the project. In the **Project Explorer** view, right-click the project and select **Build Project**.

Related information

Creating an empty Makefile project on page 74

6.1.5 Setting up the compilation tools for a specific build configuration

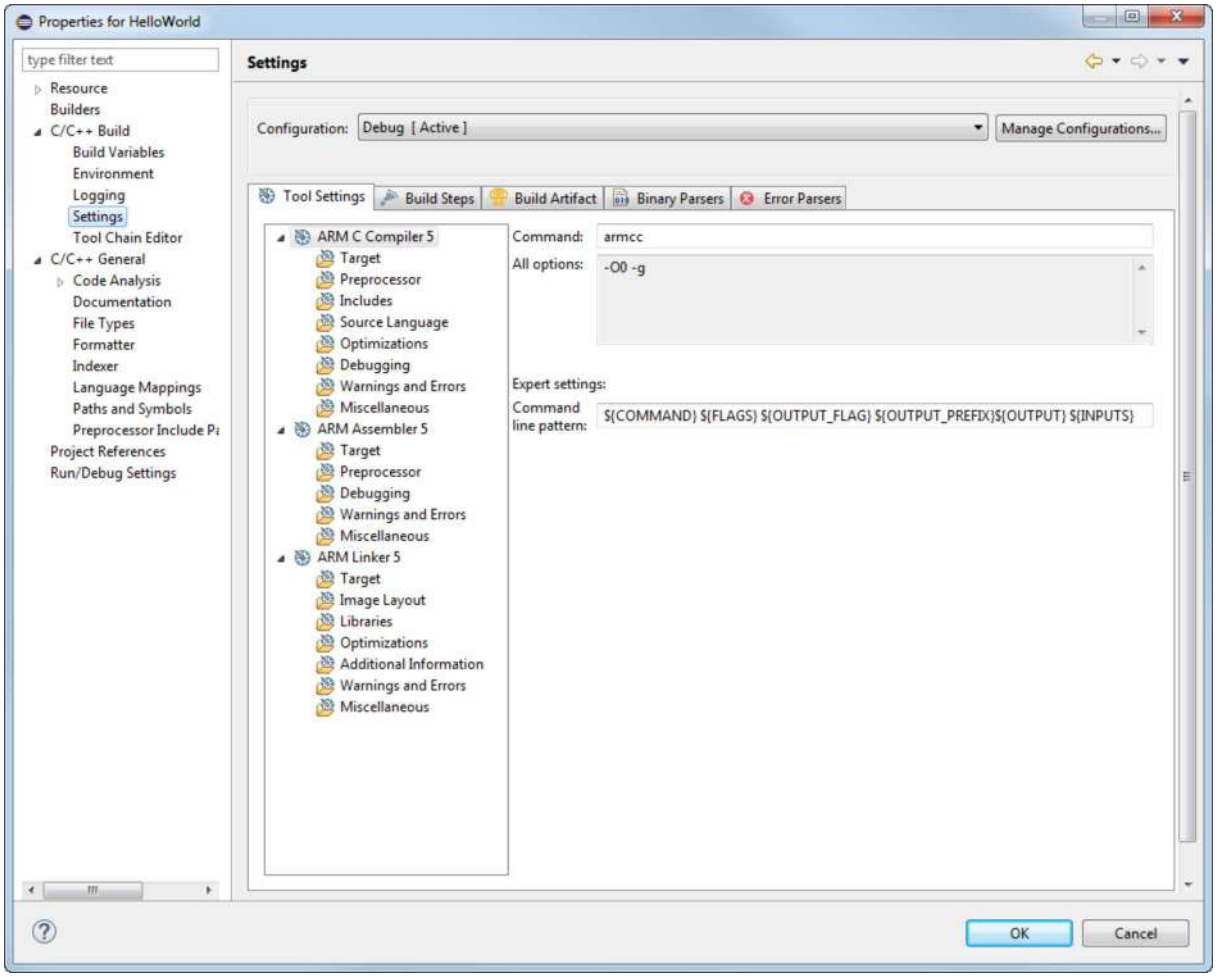
The C/C++ Build configuration panels enable you to set up the compilation tools for a specific build configuration. These settings determine how the compilation tools build an Arm executable image or library.

Procedure

1. In the **Project Explorer** view, right-click the source file or project and select **Properties**.
2. Expand **C/C++ Build** and select **Settings**.
3. The **Configuration** panel shows the active configuration. To create a new build configuration or change the active setting, click **Manage Configurations...**.
4. The compilation tools available for the current project, and their respective build configuration panels, are displayed in the **Tool Settings** tab. Click on this tab and configure the build as required.

Makefile projects do not use these configuration panels. The Makefile must contain all the required compilation tool settings.

Figure 6-3: Typical build settings dialog box for a C project



5. Click **OK**.

Results

The updated settings for your build configuration are saved.

6.1.6 Configuring the C/C++ build behavior

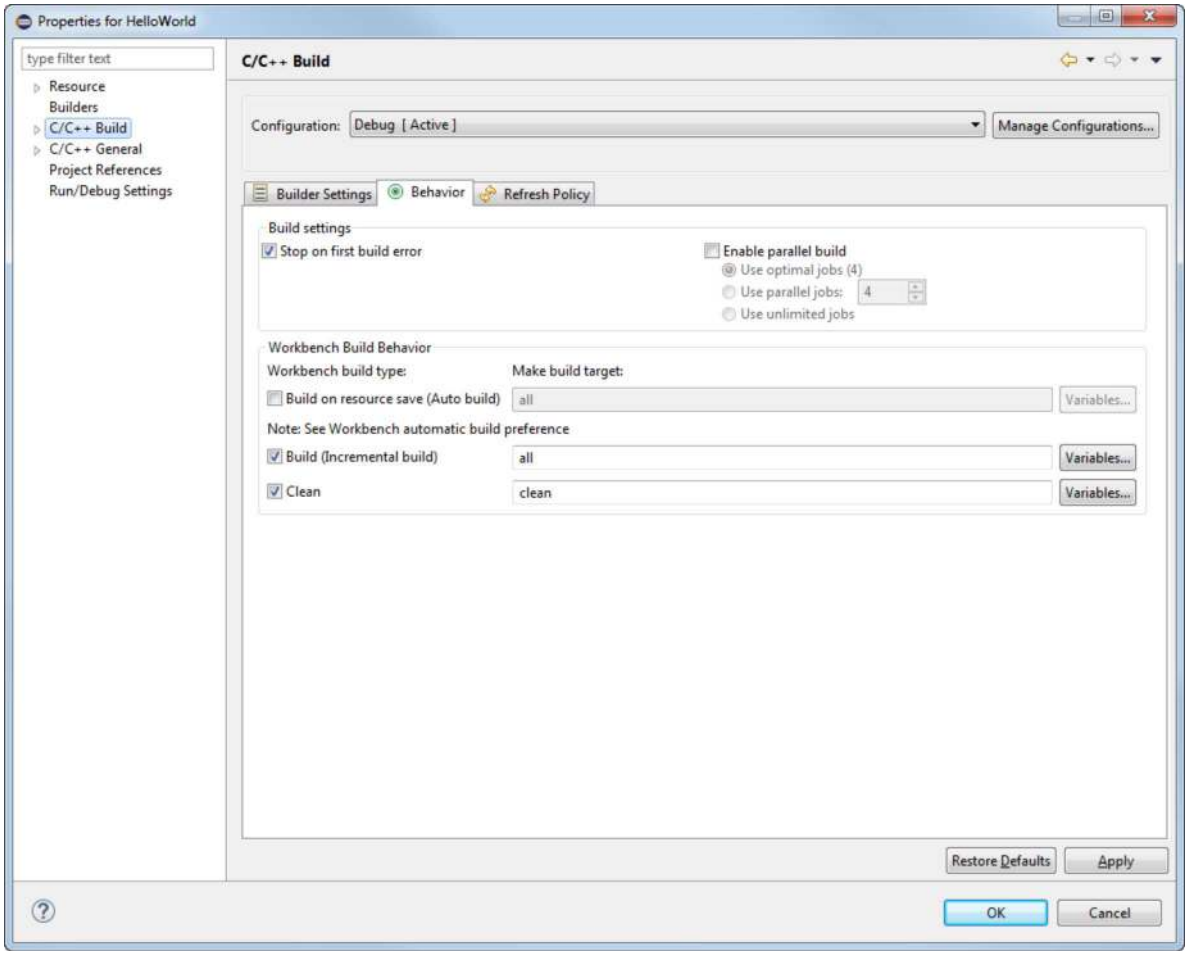
A build is the process of compiling and linking source files to generate an output file. A build can be applied to either a specific set of projects or the entire workspace. It is not possible to build an individual file or sub-folder.

Arm® Development Studio IDE provides an incremental build that applies the selected build configuration to resources that have changed since the last build. Another type of build is the **Clean build** that applies the selected build configuration to all resources, discarding any previous build states.

Automatic

This is an incremental build that operates over the entire workspace and can run automatically when a resource is saved. This setting must be enabled for each project by selecting **Build on resource save (Auto build)** in the **Behaviour** tab. By default, this behavior is not selected for any project.

Figure 6-4: Workbench build behavior



You must also ensure that **Build Automatically** is selected from the **Project** menu. By default, this menu option is selected.

Manual

This is an incremental build that operates over the entire workspace on projects with **Build (Incremental build)** selected. By default, this behavior is selected for all projects.

You can run an incremental build by selecting **Build All** or **Build Project** from the **Project** menu.


Note

Manual builds do not save before running so you must save all related files before selecting this option! To save automatically before building, you can change your default settings by selecting **Preferences... > General > Workspace** from the **Window** menu.

Clean

This option discards any previous build states including object files and images from the selected projects. The next automatic or manual build after a clean, applies the selected build configuration to all resources.

You can run a clean build on either the entire workspace or specific projects by selecting **Clean...** from the **Project** menu. You must also ensure that **Clean** is selected in the **C/C++ Build > Behaviour** tab of the **Preferences** dialog box. By default, this behavior is selected for all projects.

Build order is a feature where inter-project dependencies are created and a specific build order is defined. For example, an image might require several object files to be built in a specific order. To do this, you must split your object files into separate smaller projects, reference them within a larger project to ensure they are built before the larger project. Build order can also be applied to the referenced projects.

6.1.7 Run the Arm Development Studio IDE from the command-line to clean, build, and import projects

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

Before you begin

- Ensure that the Arm Development Studio IDE is closed.

Procedure

1. Launch the command-line console.
   - On Windows, select **Start > All Programs > Arm Development Studio > Arm DS Command Prompt**.
   - On Linux, run `<install_directory>/bin/suite_exec <shell>` to open a shell.
2. Run `armds_idec.exe` (on Windows) or `armds_ide` (on Linux) with the necessary options.


Note

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

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

For example:

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

---

a) You must specify one of the following application options:

- Specify `-application com.arm.cmsis.pack.project.headlessbuild` to build, clean, and import existing Eclipse projects.
- Specify `-application com.arm.cmsis.pack.uv.headlessuvimport` to build, clean, and import existing µVision projects.

b) Specify additional options as required. See Headless tools in the Arm Development Studio command prompt for more information on the available options.

Example 6-1: Example: Build and clean projects with the Arm Development Studio command-line

- On Windows, import, clean, and build an Eclipse project.

```
armds_idec.exe -nosplash -application com.arm.cmssis.pack.project.headlessbuild
-data C:<\path\to\workspace> -import C:<\path\to\project\directory> -cleanBuild
<projectName>
```

- On Windows, clean and build all the projects in a specified workplace.

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

- On Linux, import and build multiple Eclipse projects.

```
armds_ide -nosplash -application com.arm.cmsis.pack.project.headlessbuild -data </
path/to/workspace> -import <path/to/project1> -import <path/to/project2> -build
<project1> -build <project2>
```

- On Windows, build an Eclipse project's Release configuration.

```
armds_idec.exe -nosplash -application com.arm.cmsis.pack.project.headlessbuild -
build <project/Release>
```

- On Linux, clean and build the Debug configurations of multiple Eclipse projects in your workspace.

```
armds_ide -nosplash -application com.arm.cmsis.pack.project.headlessbuild -data </
path/to/workspace> -cleanBuild <project1/Debug> -cleanBuild <project2/Debug>
```

- On Windows, import a µVision project.

```
armds_idec.exe -nosplash -application com.arm.cmsis.pack.uv.headlessuvimport -data
C:<\path\to\workspace> -import <path/to/projectName.uvprojx>
```

- On Linux, import multiple µVision projects, then clean and build all projects in your workspace.

```
armds_ide -nosplash -application com.arm.cmsis.pack.uv.headlessuvimport -
data </path/to/workspace> -import <path/to/project1.uvprojx> -import <path/to/
project2.uvprojx> -cleanBuild all
```

Related information

Launch the Arm Development Studio command prompt on page 66
Headless tools in the Arm Development Studio command prompt on page 69

6.1.8 Updating a project to a new toolchain

If you have several products installed, only the latest toolchain is listed in the **New Project** wizard. Therefore, if you have projects that use an older toolchain, you must update them to the latest toolchain.

Procedure

1. Right-click on the project in the **Project Explorer** view, and select **Properties**.
2. Expand **C/C++ Build** and select **Tool Chain Editor**.
3. Select the toolchain from the **Current toolchain** drop-down list and click **OK**.

6.1.9 Add a source file to your project

You can add existing source files to your Arm® Development Studio project. If you want to add a new source file to your project see Add a new source file to your project.

Procedure

You can add existing source files to your project using one of the following methods:

- You can create source files and then add them to the project:
  1. Create files on your local system.
  2. Drag and drop the files into the project folder structure in the **Project Explorer** view of Arm Development Studio.

     In the **File Operation** dialog box, select whether you want to ** Copy files** or **Link to Files**. Click **OK**.
- Import existing source files:
  1. Select **File > Import > General > File System**.
  2. In the **File system** dialog box:
     a. In **From directory**, enter directory containing the existing source files.

b. Select the required files in the list of files for the selected directory.
c. In **Into Folder**, click **Browse...** and select the required project folder.
d. Ensure the **Options** are set as you require.
e. Click **Finish**.

Results

The source file is visible in the **Project Explorer** view. If the files do not show in the project, update the views in Arm Development Studio by selecting **File > Refresh** from the main menu.

Related information

Perspectives and Views
Eclipse online documentation: Code templates

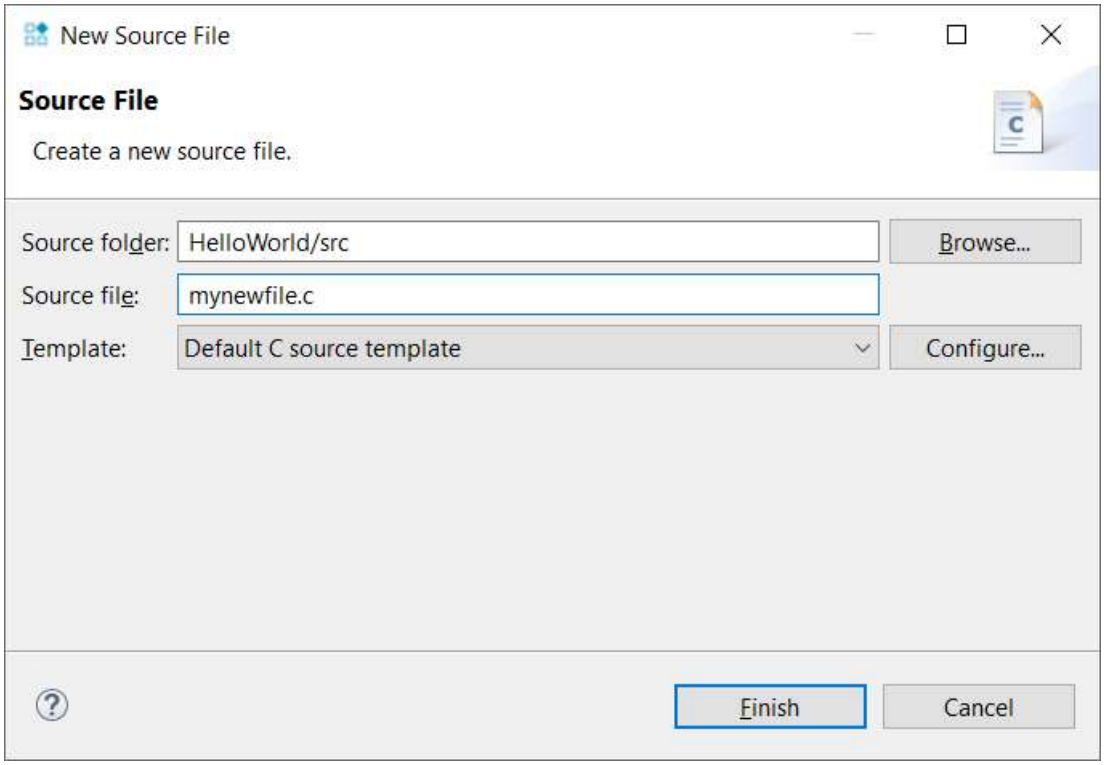
6.1.10 Add a new source file to your project

You can add new source files to your Arm® Development Studio project. If you want to add an existing source file to your project see Add a source file to your project.

Procedure

1. Access the **New Source File** wizard using one of the following methods:
   - In the **Project Explorer**, right-click on the project and select **New > Source File** to open the **New Source File** wizard.
   - From the main menu bar, select **File > New > Other > C/C++ > Source File**. Then click **Next**.

Figure 6-5: Adding a new source file to your project



2. Update the fields in the **New Source File** dialog box as required:

- **Source folder**

  Enter the source folder where the new source file will be saved. If this field is not already populated with the required source folder, click **Browse...** and select a source folder from the required project.

- **Source file**

  Enter a name for the new source file including the file extension.

- **Template**

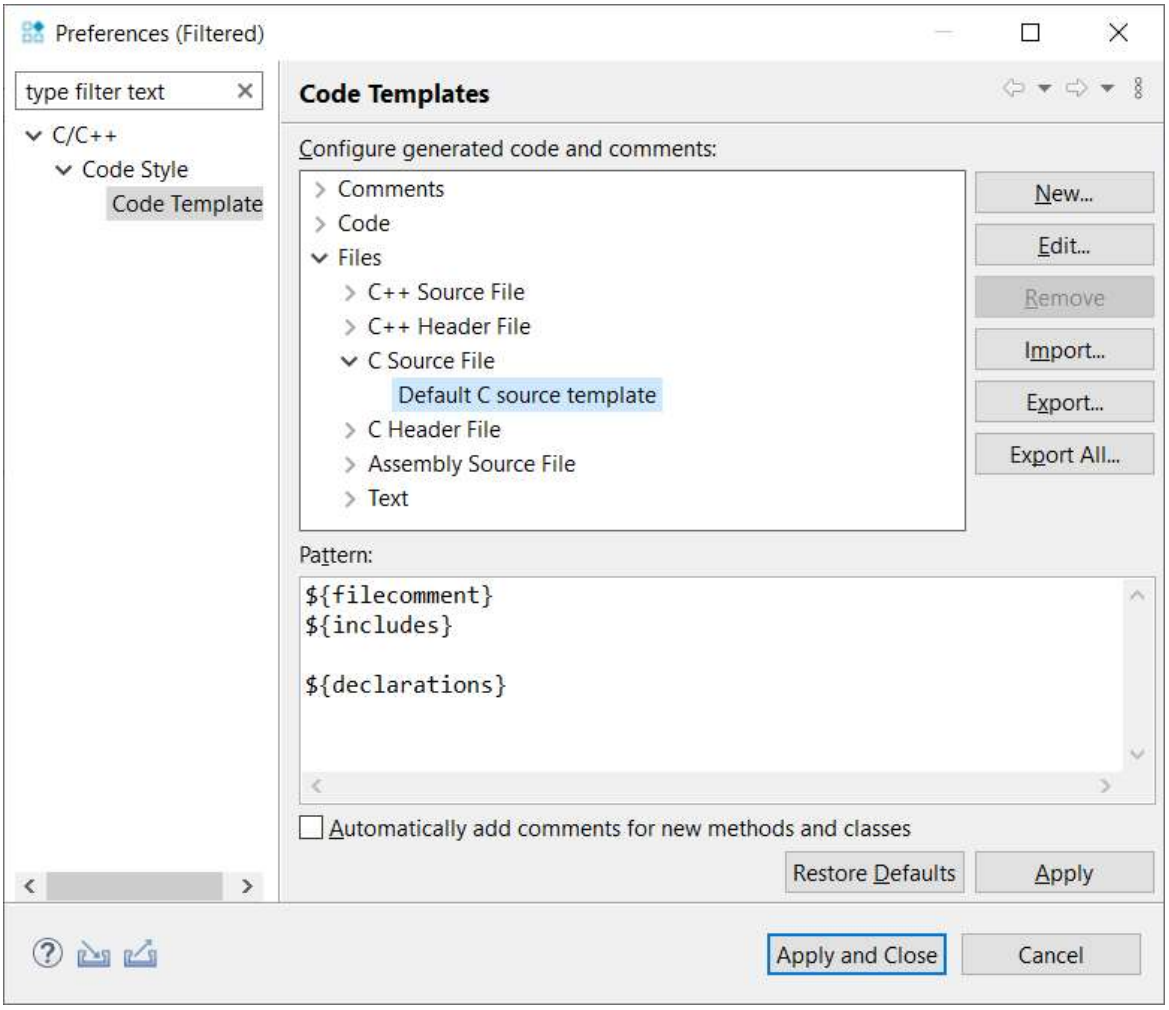
  Select a source file template from the drop-down list. The default options are:

  - <None>
  - Default C++ source template
  - Default C++ test template
  - Default C source template

  The default templates only provide basic metadata about the newly created file, that is, the author and the date it was created.

  To use your own source file template, click **Configure** and the **Code Templates** preference panel opens, where you can add or configure your own templates.

Figure 6-6: Code template configuration



3. Click **Finish**.

Results

The new source file is visible in the **Project Explorer** view.

Related information

Perspectives and Views
Eclipse online documentation: Code templates

6.1.11 Sharing Arm Development Studio projects

You can share Arm® Development Studio projects between users if necessary.

- There are many different ways to share projects and files, for example, using a source control tool. This topic covers the general principles of sharing projects and files using Arm Development Studio, and not the specifics of any particular tool.
- To share files, it is recommended to do so at the level of the project and not the workspace. Your source files in Arm Development Studio are organized into projects, and projects exist in your workspace. A workspace contains many files, including files in the `.metadata` directory, that are specific to an individual user or installation.

In each project, the files that must be shared beyond just your source code are:

- `.project` - Contains general information about the project type, and the Arm Development Studio plug-ins to use to edit and build the project.
- `.cproject` - Contains C/C++ specific information, including compiler settings.

Arm Development Studio places built files into the project directory, including auto-generated makefiles, object files, and image files. Not all files have to be shared. For example, sharing an auto-generated makefile might be useful to allow building the project outside of Arm Development Studio, but if projects are only built in Arm Development Studio then this is not necessary.

You must be careful when creating and configuring projects to avoid hard-coded references to tools and files outside of Arm Development Studio that might differ between users.

To ensure that files outside of Arm Development Studio can be referenced in a user agnostic way, use the *${workspace_loc}* built-in variable or custom environment variables.

6.1.12 Working sets

Describes what working sets are, and how to use them in Arm® Development Studio.

6.1.12.1 About working sets

A working set enables you to group projects together and display a smaller subset of projects.

The **Project Explorer** view usually displays a full list of all your projects associated with the current workspace. If you have a lot of projects it can be difficult to navigate through the list to find the project that you want to use.

To make navigation easier, group your projects into working sets. You can select one or more working sets at the same time, or you can use the **Project Explorer View Menu** to switch between

one set and another. To return to the original view, select the **Deselect Working Sets** options in the **View Menu**.

Working sets are also useful to refine the scope of a search or build projects in a specific working set.

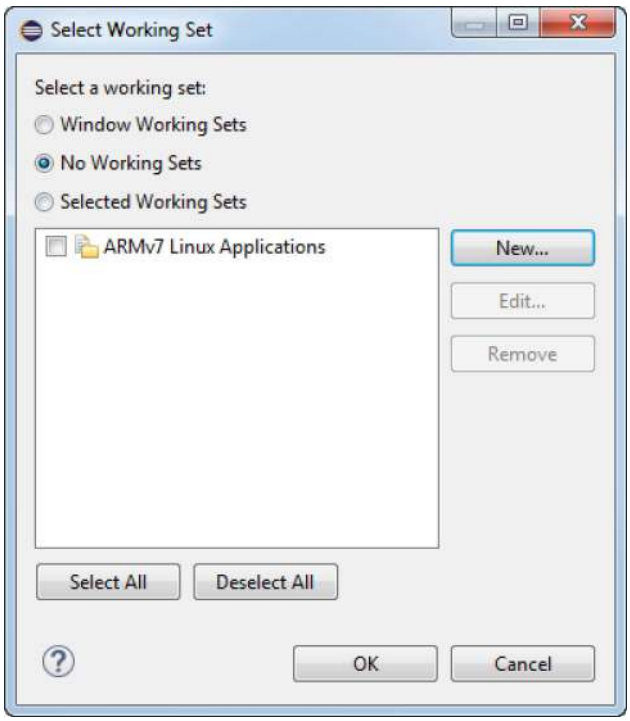
6.1.12.2 Creating a working set

Create a working set to group related projects together.

Procedure

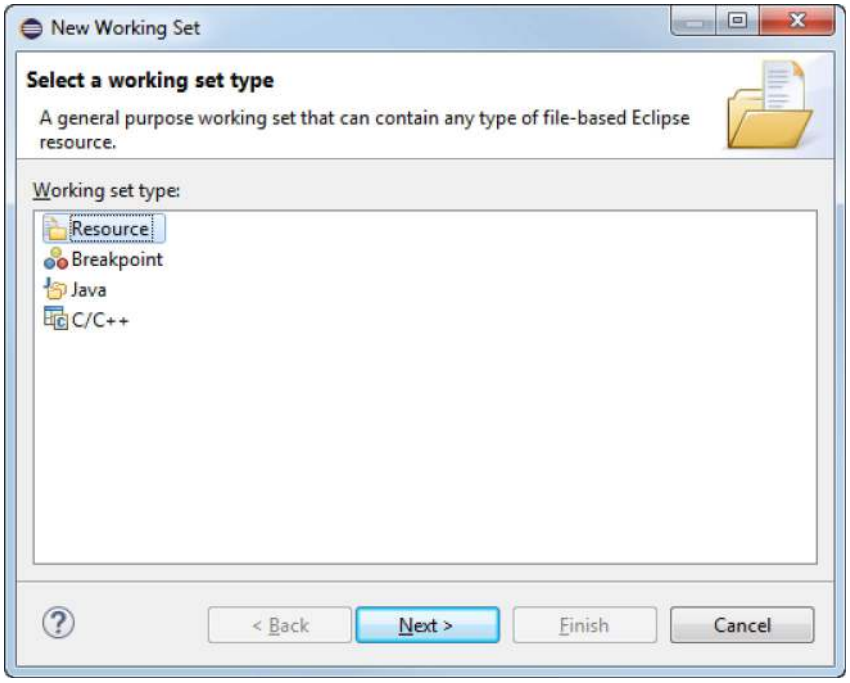
1. Click the **View Menu** hamburger icon in the **Project Explorer** view toolbar.
2. Select the **Select Working Set...** option.
3. In the **Select Working Set** dialog box, click **New...**.

   Figure 6-7: Creating a new working set



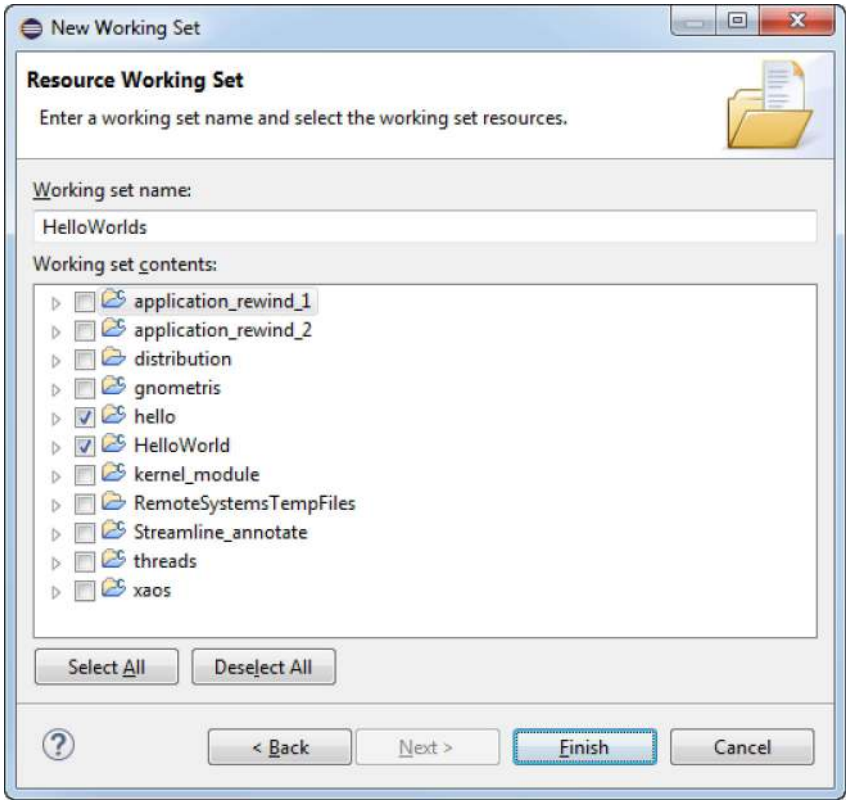
4. Under **Working set type**, select **Resource** and click **Next**.

Figure 6-8: Selecting the resource type for the new working set



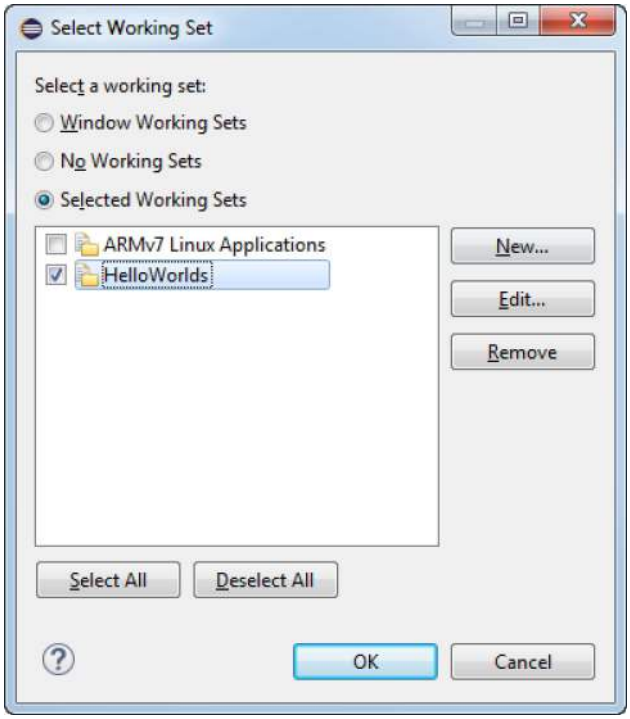
5. In the **Working set name** field, enter a suitable name.
6. In the **Working set contents** panel, you can select existing projects that you want to associate with this working set, or you can return to the wizard later to add projects.

Figure 6-9: Adding new resources to a working set



7. Click **Finish**.
8. If required, repeat these steps to create more working sets.
9. In the **Select Working Set** dialog box, select the working sets that you want to display in the **Project Explorer** view.

Figure 6-10: Select the required working set



10. Click **OK**.

Results

The filtered list of projects are displayed in the **Project Explorer** view. Another feature of working sets that can help with navigation is the option to change the top level element in the **Project Explorer** view.

6.1.12.3 Changing the top-level element when displaying working sets

In the **Project Explorer** view, if you have more than one working set then you might want to display the projects in a hierarchical tree with the working set names as the top level element. This is not selected by default.

Procedure

1. In the **Project Explorer** view toolbar, click the **View Menu** hamburger icon.
2. Select **Top Level Elements** from the context menu.
3. Select either **Projects** or **Working Sets**.

6.1.12.4 Deselecting a working set

You can change the display of projects in the **Project Explorer** view and return to the full listing of all the projects in the workspace.

Procedure

1. Click on the **View Menu** icon in the **Project Explorer** view toolbar.
2. Select **Deselect Working Set** from the context menu.

6.2 Importing and exporting projects

Describes how to import resources from existing projects and how to export resources to use with tools external to Arm® Development Studio.

6.2.1 Importing and exporting options

A resource must exist in a project in Arm® Development Studio before you can use it in a build.

If you want to use an existing resource from your file system in one of your projects, the recommended method is to use the **Import** wizard. To do this, select **Import...** from the **File** menu.

If you want to use a resource externally, the recommended method is to use the **Export** wizard. To do this, select **Export...** from the **File** menu.

There are several options available in the import and export wizards:

General

This option enables you to import and export the following:

- Files from an archive zip file.
- Complete projects.
- Selected source files and project sub-folders.
- Preference settings.

C/C++

This option enables you to import the following:

- C/C++ executable files.
- C/C++ project settings.
- Existing code as Makefile project.

You can also export C/C++ project settings and indexes.

Remote Systems

This option enables you to transfer files between the local host and the remote target.

Run/Debug

This option enables you to import and export the following:

- Breakpoint settings.
- Launch configurations.

Scatter File Editor

This option enables you to import the memory map from a BCD file and convert it into a scatter file for use in an existing project.

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

6.2.2 Using the Import wizard

In addition to breakpoint and preference settings, you can use the **Import** wizard to import complete projects, source files, and project sub-folders.

Select **Import...** from the **File** menu to display the **Import** wizard.

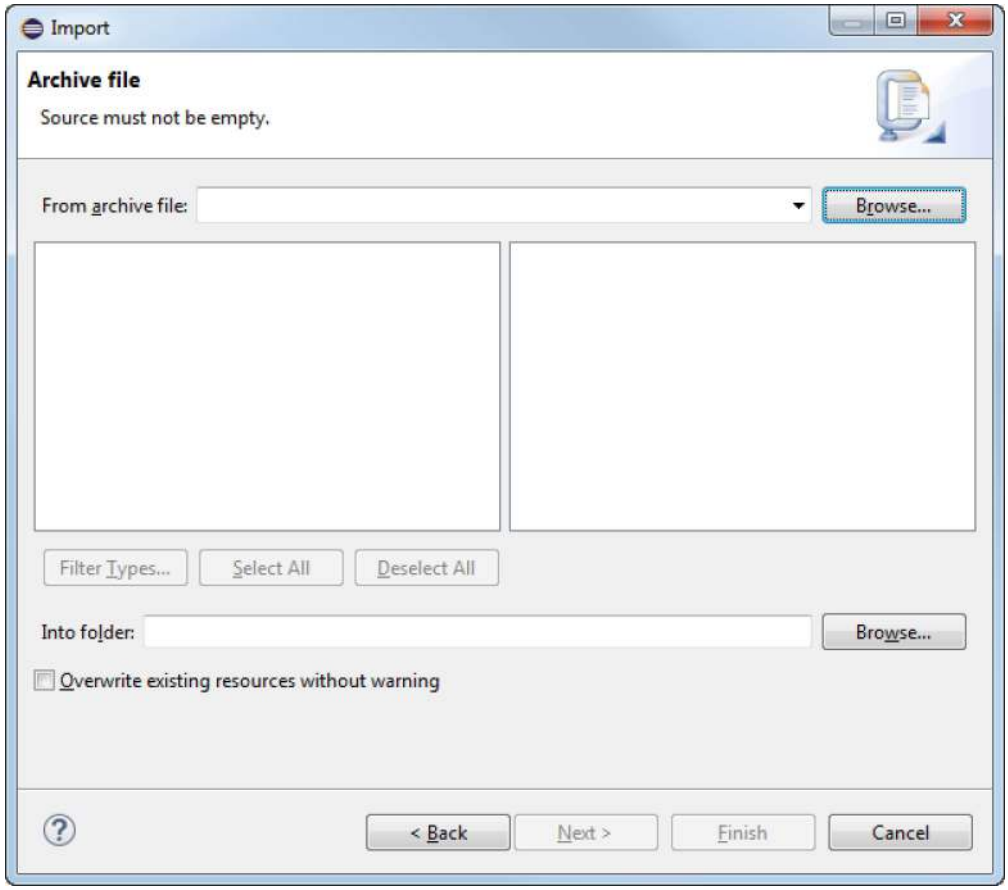
Importing complete projects

To import a complete project either from an archive zip file or an external folder from your file system, you must use the **Existing Projects into Workspace** wizard. This ensures that the relevant project files are also imported into your workspace.

Importing source files and project sub-folders

Individual source files and project sub-folders can be imported using either the **Archive File** or **File System** wizard. Both options produce a dialog box similar to the following example. Using the options provided you can select the required resources and specify the relevant options, filename, and destination path.

Figure 6-11: Typical example of the import wizard



With the exception of the **Existing Projects** into **Workspace** wizard, files and folders are copied into your workspace when you use the **Import** wizard. To create a link to an external file or project sub-folder you must use the **New File** or **New Folder** wizard.

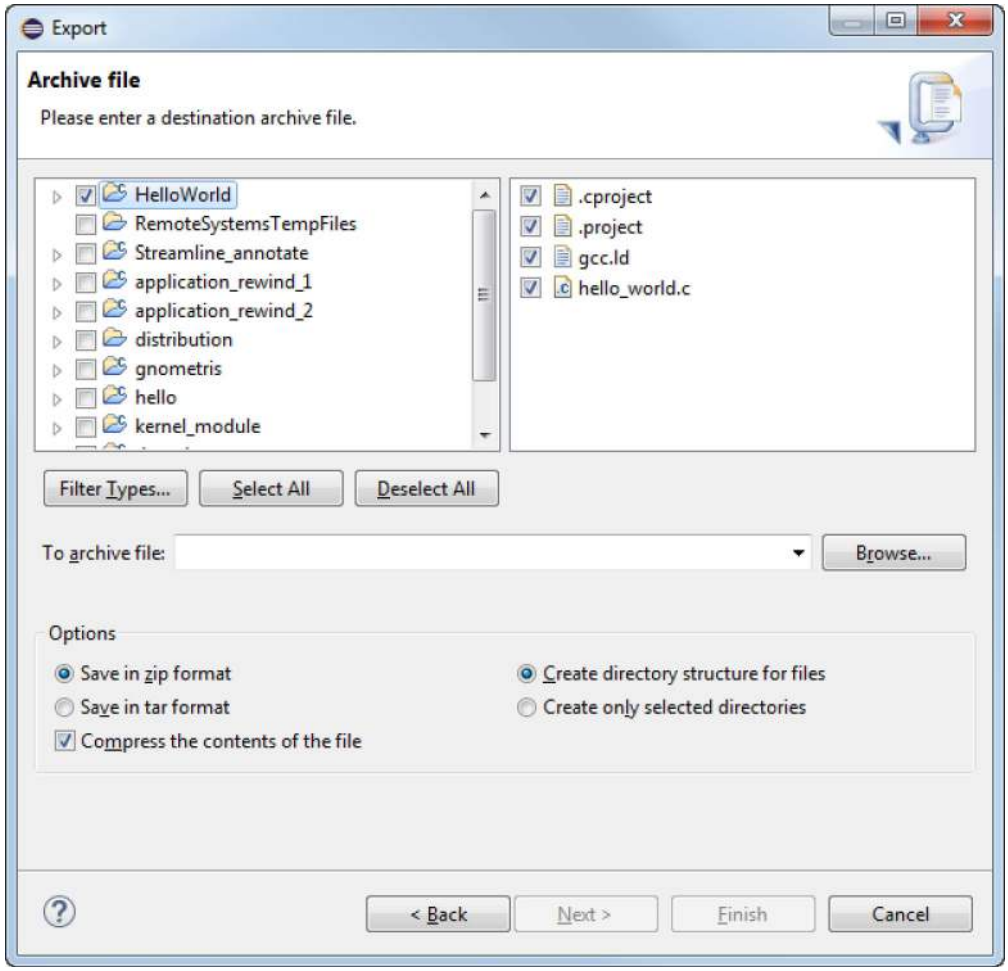
6.2.3 Using the Export wizard

You can use the **Export** wizard to export complete projects, source files and, project sub-folders in addition to breakpoint and preference settings.

Select **Export...** from the **File** menu to display the **Export** wizard.

The procedure is the same for exporting a complete project, a source file, and a project sub-folder. If you want to create a zip file you can use the **Archive File** wizard, or alternatively you can use the **File System** wizard. Both options produce a dialog box similar to the example shown here. Using the options provided you can select the required resources and specify the relevant options, filename, and destination path.

Figure 6-12: Typical example of the export wizard



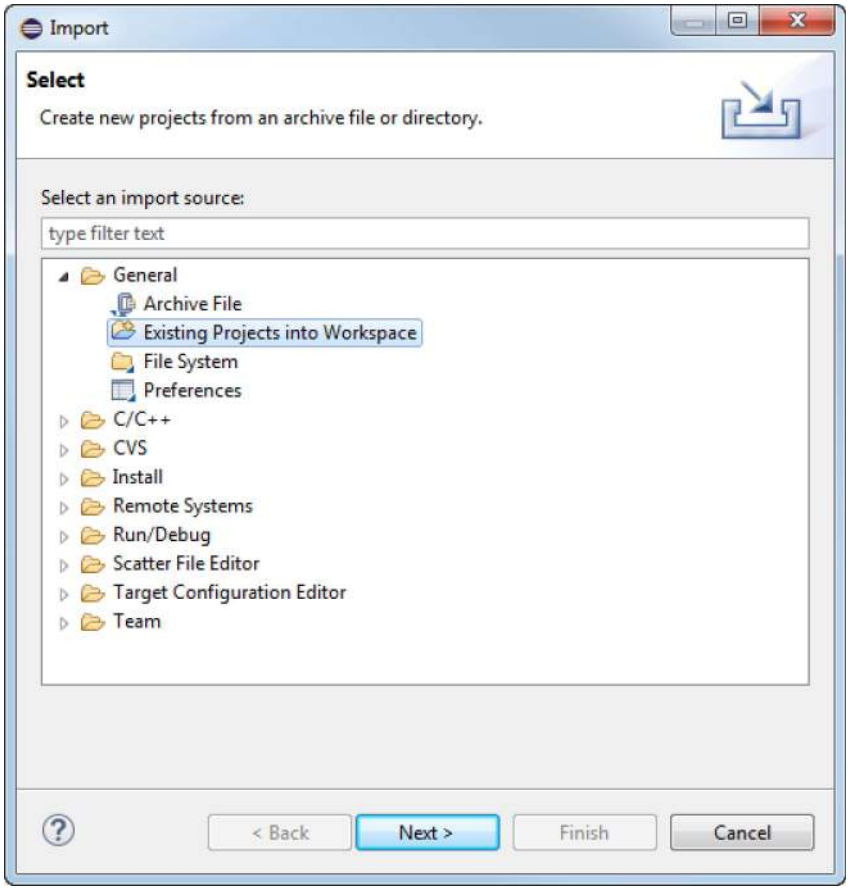
6.2.4 Import an existing Eclipse project

If you have an existing Eclipse project, you can import it into your workspace.

Procedure

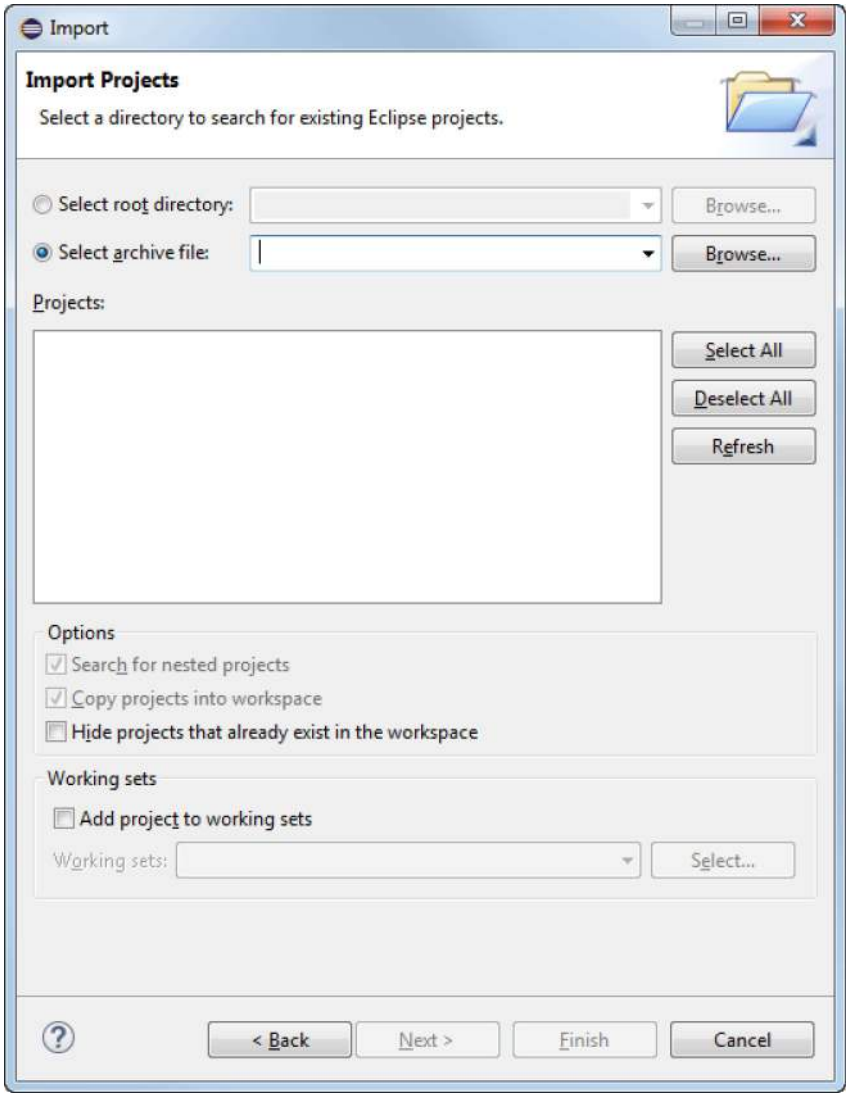
1. Select **File > Import... > Existing Project into Workspace**. Click **Next**

Figure 6-13: Selecting the import source type



2. Click **Browse** and navigate to the folder that contains the project that you want to import.
3. In the **Projects** panel, select the project that you want to import.
4. Select **Copy projects into workspace** if required, or deselect to create links to your existing project(s) and associated files.
5. If you are not using working sets to group your projects then you can skip this step.
   a) Select **Add project to working sets**.
   b) Click **Select...**.
   c) Select an existing working set or create a new one and then select it.
   d) Click **OK**.
6. Click **Finish**.

Figure 6-14: Selecting an existing Eclipse projects for import



Note

If your existing project contains project settings from an older version of the build system, you are given the option to update your project. Using the latest version means that you can access all the latest toolchain features.

Results

The imported project is visible in the **Project Explorer** view.

6.3 Examples provided with Arm Development Studio

Arm® Development Studio provides a selection of examples to help you get started:

- Bare-metal software development examples for Armv7 that show:
  - Compilation with Arm Compiler for Embedded 6.
  - Compilation with GCC bare-metal compiler.
  - Armv7 bare-metal debug.

  The code is located in the `<examples_directory>\Bare-metal_examples_Armv7.zip` archive file.
- Bare-metal software development examples for Armv8 that show:
  - Compilation with Arm Compiler for Embedded 6.
  - Compilation with GCC bare-metal compiler.
  - Armv8 bare-metal debug.

  The code is located in the `<examples_directory>\Bare-metal_examples_Armv8.zip` archive file.
- Bare-metal software development examples for Armv9 that show:
  - Compilation with Arm Compiler for Embedded 6.
  - Armv9 bare-metal debug.

  The code is located in the `<examples_directory>\Bare-metal_examples_Armv9.zip` archive file.

The debug and compilation features that are available to you depends on which version of Arm Development Studio you have installed. For detailed information on which features are available in the different Arm Development Studio editions, see: Arm Development editions

---

- Bare-metal software development examples for *Scalable Vector Extension* (SVE) and SVE2 using Arm Compiler for Embedded 6.

  The code is located in the `<examples_directory>\SVE2_examples.zip` archive file.
- Arm Linux examples built with GCC Linux compiler that show build, debug, and performance analysis of simple C/C++ console applications, shared libraries, and multi-threaded applications. The files are located in the `<examples_directory>\Linux_examples.zip` archive file.
- Examples for Keil® RTX version 5 RTX Real-Time Operating System (RTX-RTOS) are located in the `<examples_directory>\RTX5_examples.zip` archive file.
- Software examples for Arm Debugger's Debug and Trace Services Layer (DTSL). The examples are located in the `<examples_directory>\DTSL_examples.zip` archive file.
- Jython examples for Arm Debugger. The examples are located in the `<examples_directory>\Jython_examples.zip` archive file.
- The CoreSight Access Library is available as a github project at https://github.com/ARM-software/CSAL. A recent snapshot of the library from github is located in the archive file, `<examples_directory>\CoreSight_Access_Library.zip`.

- Optional packages with source files, libraries, and pre-built images for running the examples can be downloaded from the Arm Development Studio downloads page . You can also download the Linux distribution project with header files and libraries for the purpose of rebuilding the Arm Linux examples from the Arm Development Studio downloads page.

You can extract these examples to a working directory and build them from the command-line, or you can import them into Arm Development Studio IDE using the import wizard. All examples provided with Arm Development Studio contain a pre-configured IDE launch script that enables you to easily load and debug example code on a target.

Each example provides instructions on how to build, run, and debug the example code. You can access the instructions from the main index, `<examples_directory>\docs\index.html`.

Related information

Import the example projects on page 98

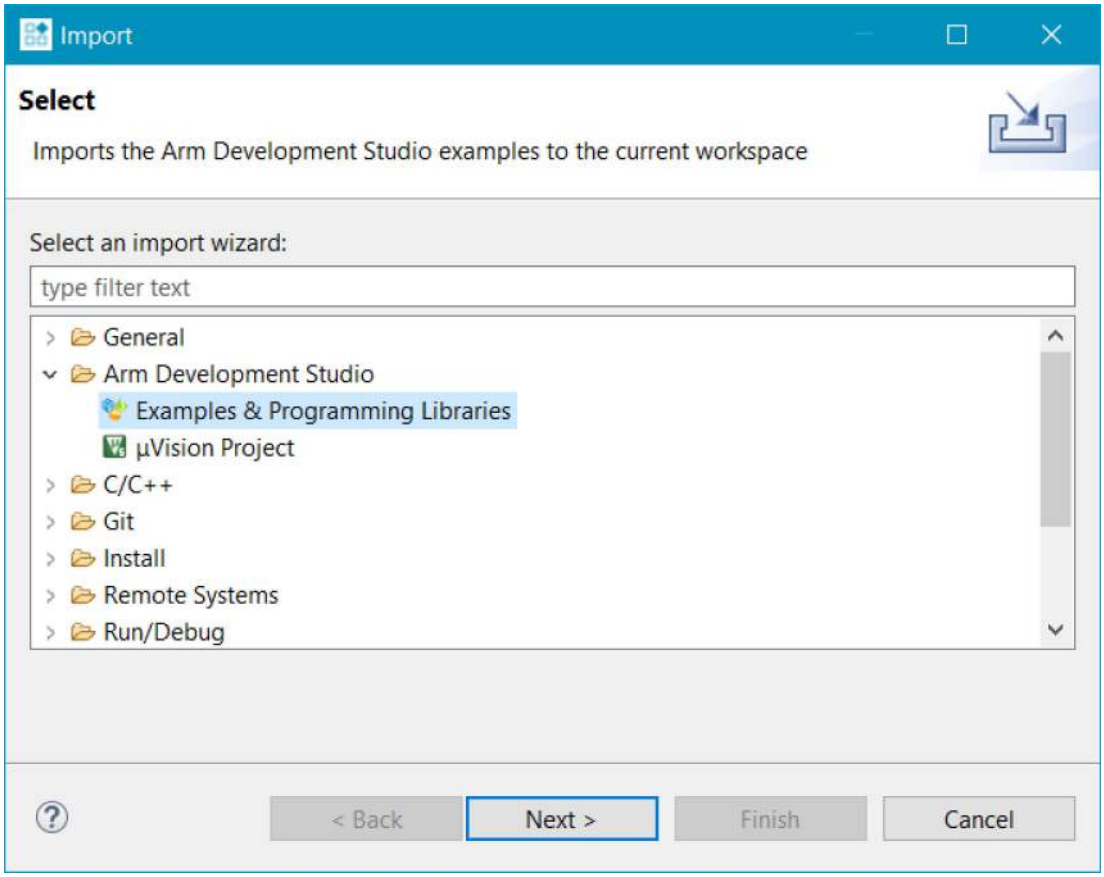
6.4 Import the example projects

To use the example projects provided with Arm® Development Studio, you must first import them.

Procedure

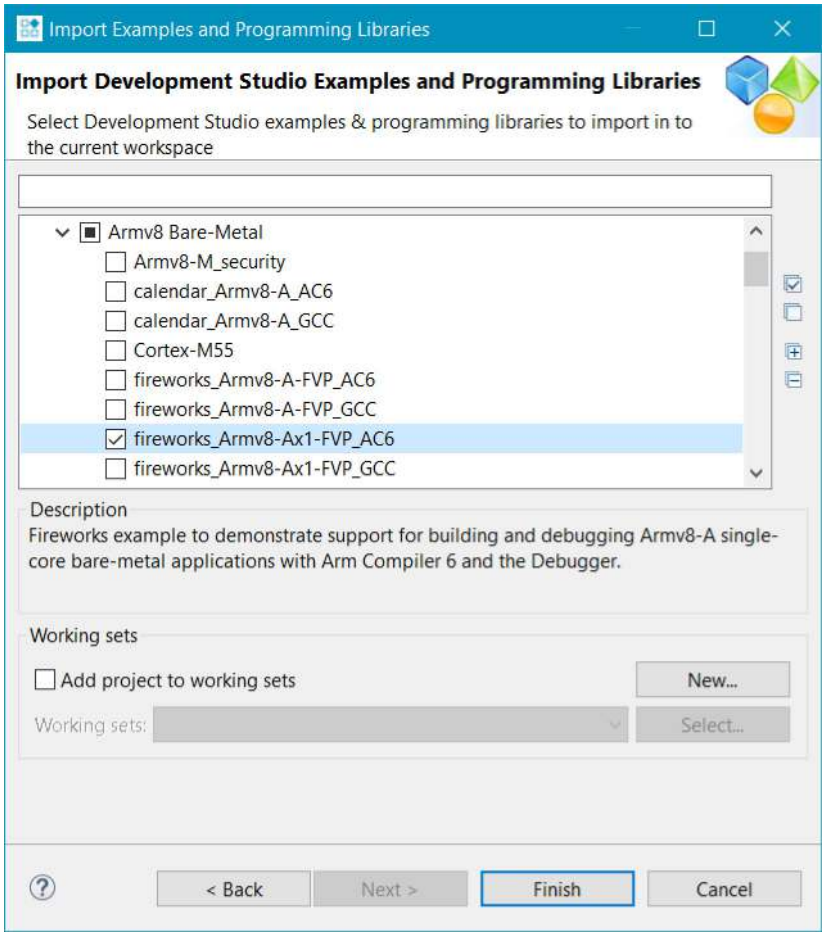
1. Launch **Arm Development Studio IDE**.
2. Arm recommends that you create another workspace for example projects, so that they remain separate from your own projects. To do this, select **File > Switch Workspace > Other > Browse > Make new folder**, and enter a suitable name.
   Result: Arm Development Studio IDE relaunches.
3. In the main menu, select **File > Import...**.
4. Expand the **Arm Development Studio** group, select **Examples and Programming Libraries** and click **Next**.

Figure 6-15: Import dialog box



5. Select the examples and programming libraries you want to import. If a description for the selected example exists, you can view it in the **Description** pane.

Figure 6-16: Select items to import



6. Click **Finish**.

Results

You can browse the imported examples in the **Project Explorer**.

Each example contains a `readme.html` which explains how you can work with the example.

Related information

About working sets on page 86

7. Writing code

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

7.1 Editing source code

You can use the editors provided with Arm® Development Studio to edit your source code or you can use an external editor. If you work with an external editor you must refresh Development Studio to synchronize the views with the latest updates.

To do this, in the **Project Explorer** view, select the updated project, sub-folder, or file and click **File > Refresh**. Alternatively, enable automatic refresh options under **General > Workspace** in the **Preferences** dialog box. Configure your automatic refresh settings by selecting either **Refresh using native hooks or polling** or **Refresh on access** options.

When you open a file in Development Studio, a new editor tab appears with the name of the file. An edited file displays an asterisk (*) in the tab name to show that it has unsaved changes.

To view two or more editor tabs side-by-side, click on one of the tabs and drag it over an editor border.

In the left-hand margin of the editor tab you can find a vertical bar that displays markers relating to the active file.

Navigating

There are several ways to navigate to a specific resource in Development Studio. You can use the **Project Explorer** view to open a resource by browsing through the resource tree and double-clicking on a file. An alternative is to use the keyboard shortcuts or use the options from the **Navigate** menu.

Searching

To locate information or specific code contained in one or more files in Development Studio, you can use the options from the **Search** menu. Textual searching with pattern matching and filters to refine the search fields are provided in a customizable **Search** dialog box. You can also open this dialog box from the main workbench toolbar.

Content assist

The C/C++ editor, Arm assembler editor, and the Arm Debugger **Commands** view provide content assistance at the cursor position to auto-complete the selected item. Using the Ctrl+Space keyboard shortcut produces a small dialog box with a list of valid options to choose from. You can filter the list by partially typing a few characters before using the keyboard shortcut. From the list you can use the Arrow Keys to select the required item and then press the Enter key to insert it.

Bookmarks

You can use bookmarks to mark a specific position in a file or mark an entire file so that you can return to it quickly. To create a bookmark, select a file or line of code that you want to mark and select **Add Bookmark** from the **Edit** menu. The Bookmarks view displays all the user defined bookmarks. To access the bookmarks, select **Window > Show View > Bookmarks** from the main menu. If the **Bookmarks** view is not listed then select **Others...** for an extended list.

To delete a bookmark, open the **Bookmarks** view, click on the bookmark that you want to delete, and select **Delete** from the **Edit** menu.

7.2 About the C/C++ editor

The standard C/C++ editor is provided by the CDT plug-in that provides C and C++ extensions to Eclipse. It provides syntax highlighting, formatting of code and content assistance when editing C/C ++ code.

Embedded assembler in C/C++ files is supported by Arm® Compiler for Embedded but this editor does not support it and so an error is displayed. This type of code is Arm-specific and accepted Eclipse behavior so you can ignore the syntax error.

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

See the *C/C++ Development User Guide* for more information. Select **Help > Help Contents** from the main menu.

7.3 About the Arm assembler editor

The Arm assembler editor provides syntax highlighting, formatting of code and content assistance for labels in Arm assembly language source files. You can change the default settings in the dialog box.

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

The following shortcuts are also available for use:

Table 7-1: Arm assembler editor shortcuts

Shortcut	Description
Content assist	Content assist provides auto-completion on labels existing in the active file. When entering a label for a branch instruction, Partially type the label and then use the keyboard shortcut Ctrl+Space to display a list of valid auto-complete options. Use the Arrow Keys to select the required label and press Enter to complete the term. Continue typing to ignore the auto-complete list.

Shortcut	Description
Editor focus	The following options change the editor focus: • Outline View provides a list of all areas and labels in the active file. Click on an area or label to move the focus of the editor to the position of the selected item. • Select a label from a branch instruction and press F3 to move the focus of the editor to the position of the selected label.
Formatter activation	Press Ctrl+Shift+F to activate the formatter settings.
Block comments	Block comments are enabled or disabled by using Ctrl+Semicolon. Select a block of code and apply the keyboard shortcut to change the commenting state.

7.4 About the ELF content editor

The ELF content editor creates forms for the selected ELF file. You can use this editor to view the contents of image files and object files. The editor is read-only and cannot be used to modify the contents of any files.

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

The ELF content editor displays one or more of the following tabs depending on the selected file type:

Header
: Form view showing the header information.

Sections
: Tabular view showing the breakdown of all section information.

Segments
: Tabular view showing the breakdown of all segment information.

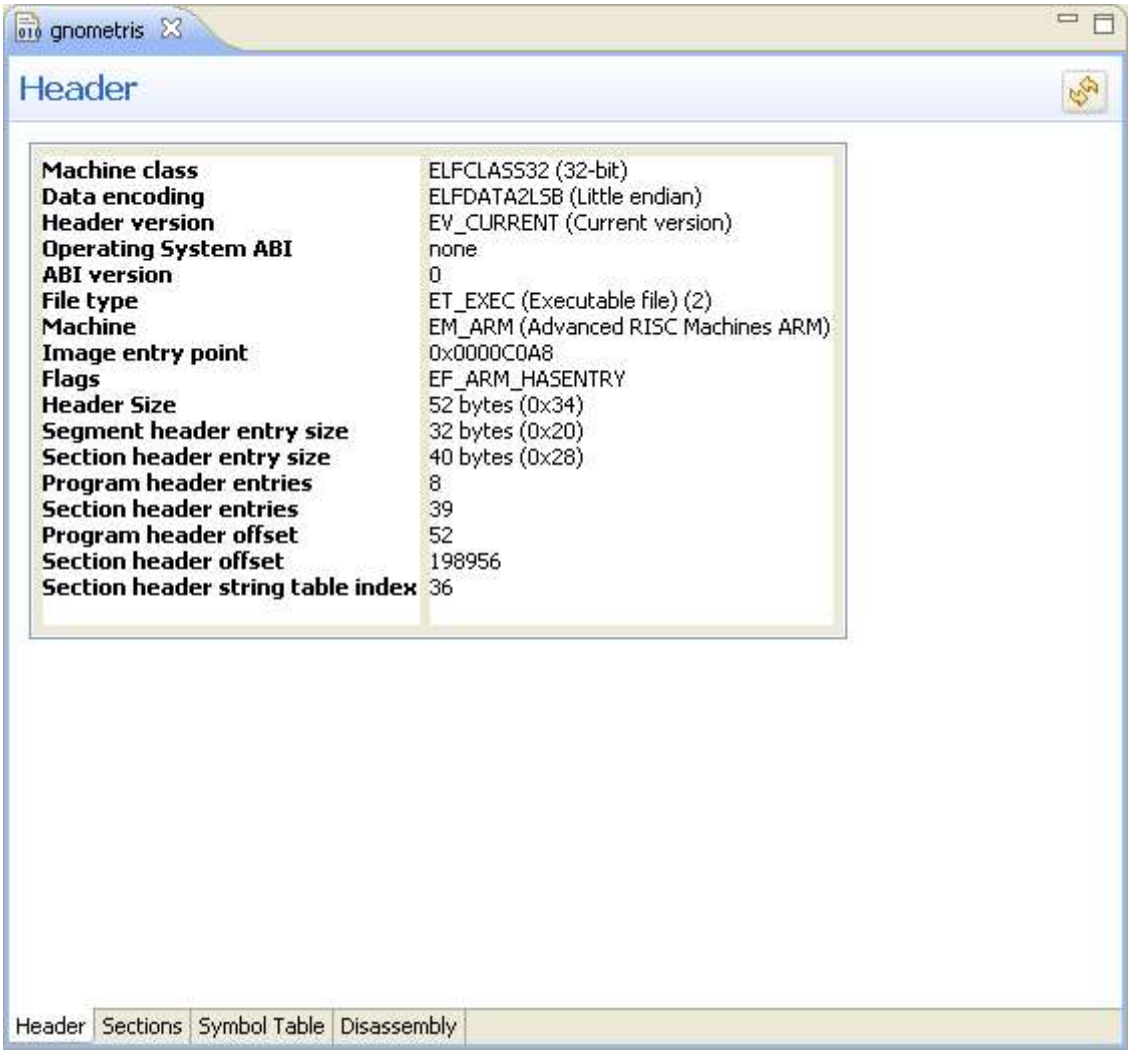
Symbol Table
: Tabular view showing the breakdown of all symbols.

Disassembly
: Textual view of the disassembly with syntax highlighting.

7.5 ELF content editor - Header tab

The **Header** tab provides a form view of the ELF header information.

Figure 7-1: Header tab

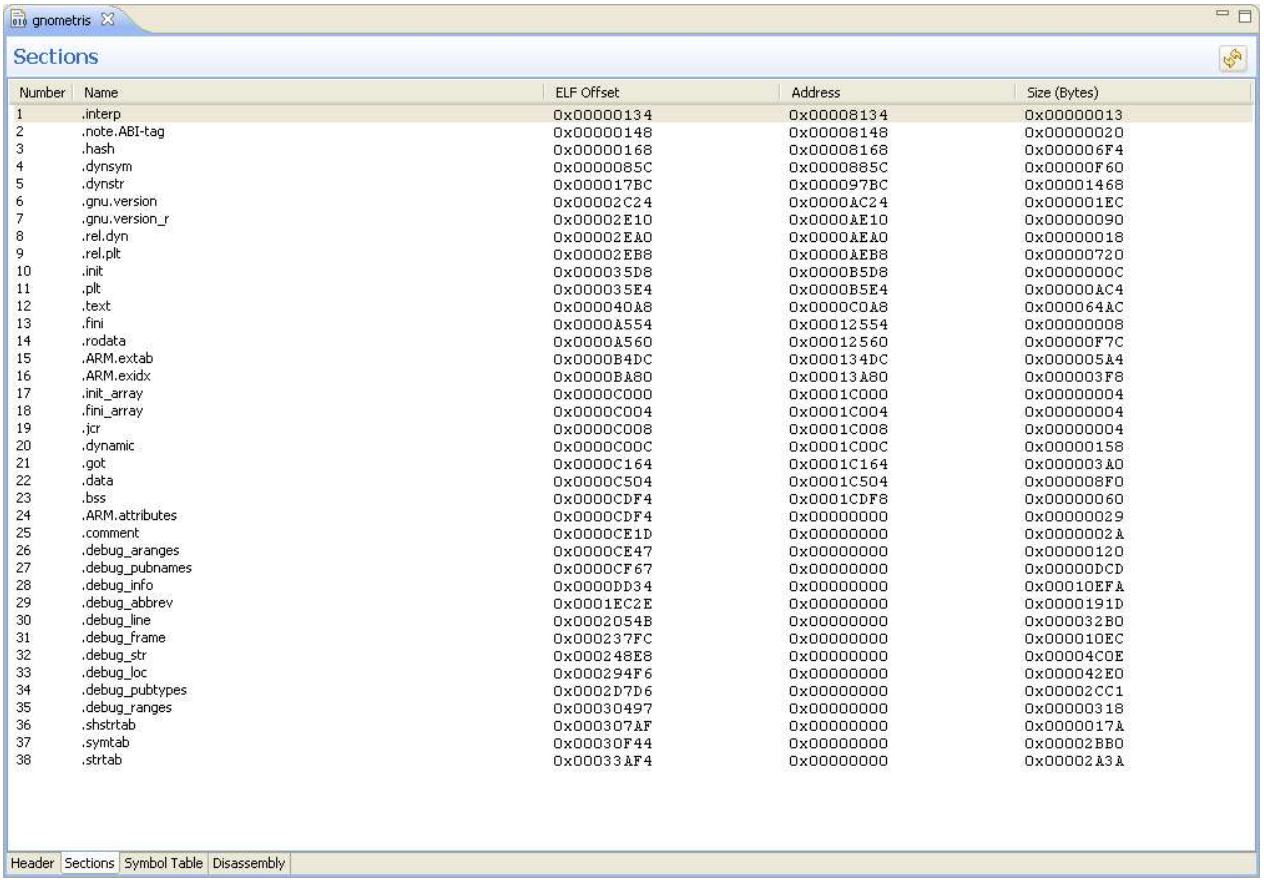


7.6 ELF content editor - Sections tab

The **Sections** tab provides a tabular view of the ELF section information.

To sort the columns, click on the column headers.

Figure 7-2: Sections tab

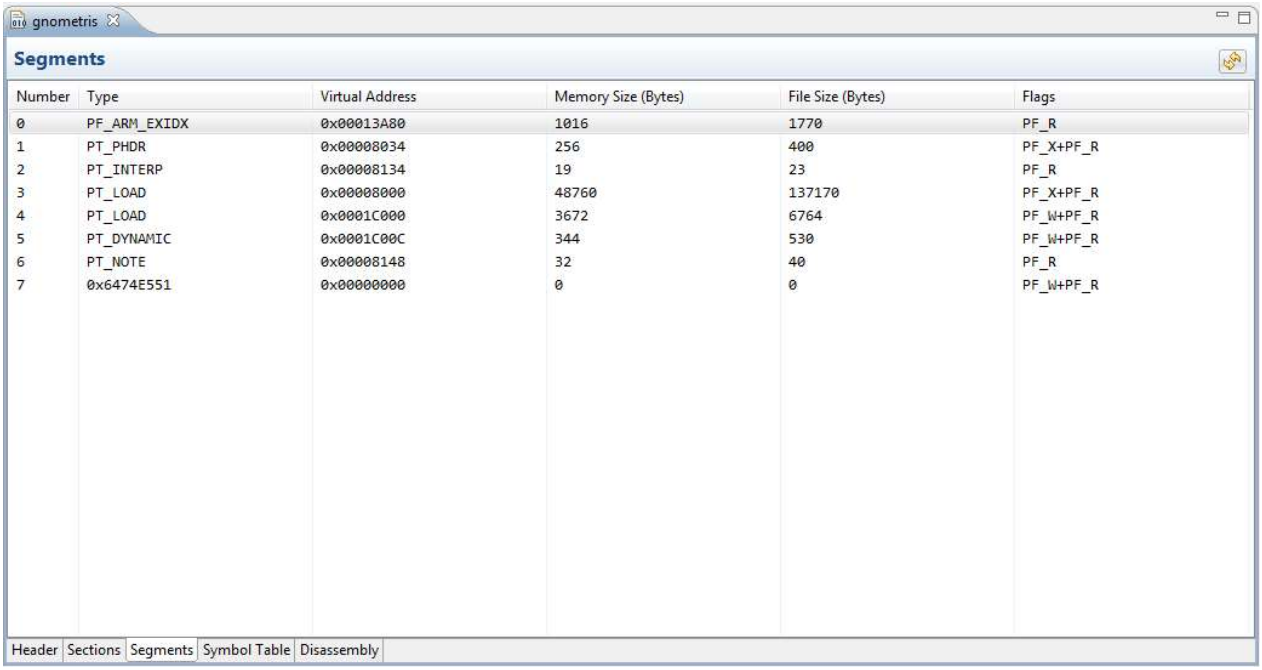


7.7 ELF content editor - Segments tab

The **Segments** tab provides a tabular view of the ELF segment information.

To sort the columns, click on the column headers.

Figure 7-3: Segments tab

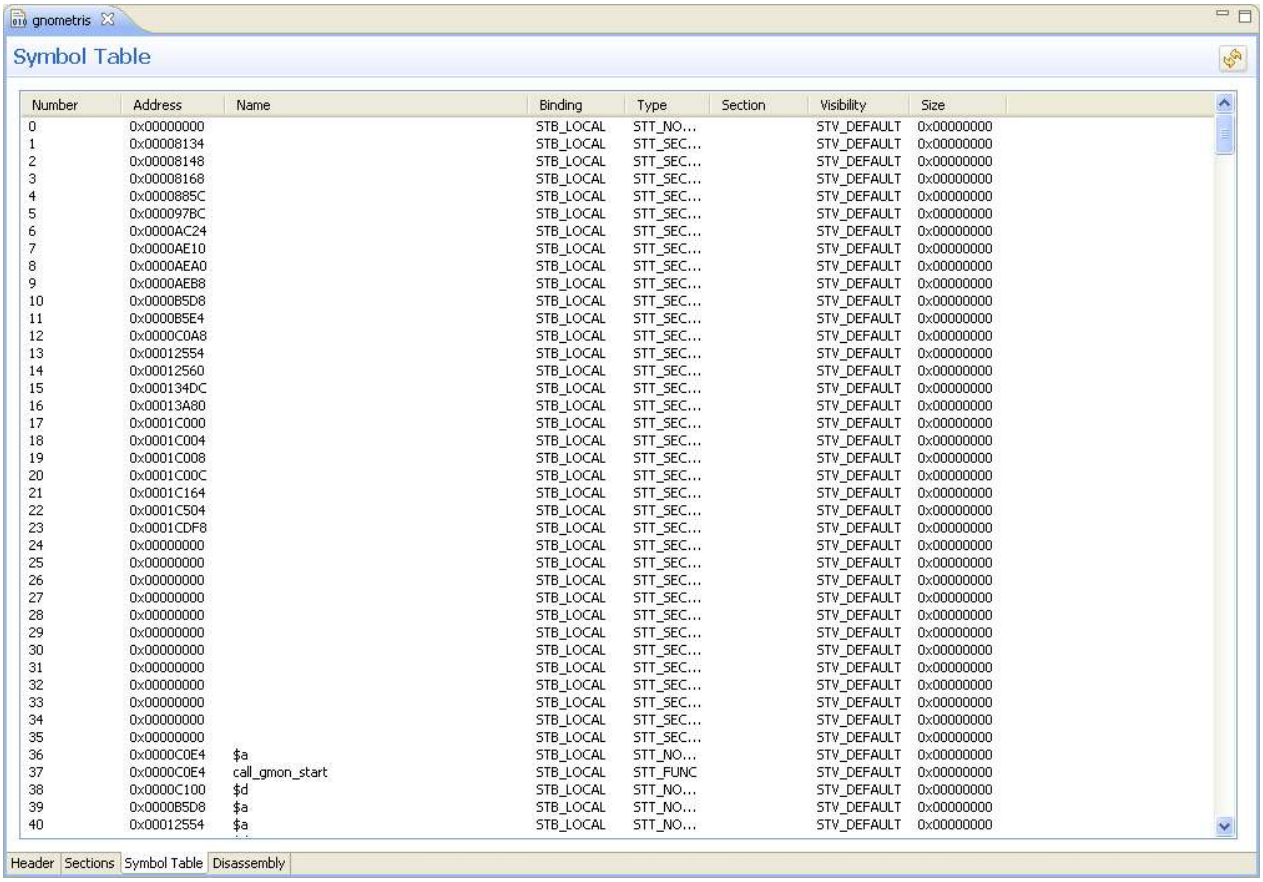


7.8 ELF content editor - Symbol Table tab

The **Symbol Table** tab provides a tabular view of the symbols.

To sort the columns, click on the column headers.

Figure 7-4: Symbol Table tab



Number	Address	Name	Binding	Type	Section	Visibility	Size
0	0x00000000		STB_LOCAL	STT_NO...		STV_DEFAULT	0x00000000
1	0x00008134		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
2	0x00008148		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
3	0x00008168		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
4	0x0000885C		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
5	0x000097BC		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
6	0x0000AC24		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
7	0x0000AE10		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
8	0x0000AEA0		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
9	0x0000AEB8		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
10	0x0000B5D8		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
11	0x0000B5E4		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
12	0x0000C0A8		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
13	0x00012554		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
14	0x00012560		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
15	0x000134DC		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
16	0x00013A80		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
17	0x0001C000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
18	0x0001C004		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
19	0x0001C008		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
20	0x0001C00C		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
21	0x0001C164		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
22	0x0001C504		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
23	0x0001CDF8		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
24	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
25	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
26	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
27	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
28	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
29	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
30	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
31	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
32	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
33	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
34	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
35	0x00000000		STB_LOCAL	STT_SEC...		STV_DEFAULT	0x00000000
36	0x0000C0E4	$a	STB_LOCAL	STT_NO...		STV_DEFAULT	0x00000000
37	0x0000C0E4	call_gmon_start	STB_LOCAL	STT_FUNC		STV_DEFAULT	0x00000000
38	0x0000C100	$d	STB_LOCAL	STT_NO...		STV_DEFAULT	0x00000000
39	0x0000B5D8	$a	STB_LOCAL	STT_NO...		STV_DEFAULT	0x00000000
40	0x00012554	$a	STB_LOCAL	STT_NO...		STV_DEFAULT	0x00000000

7.9 ELF content editor - Disassembly tab

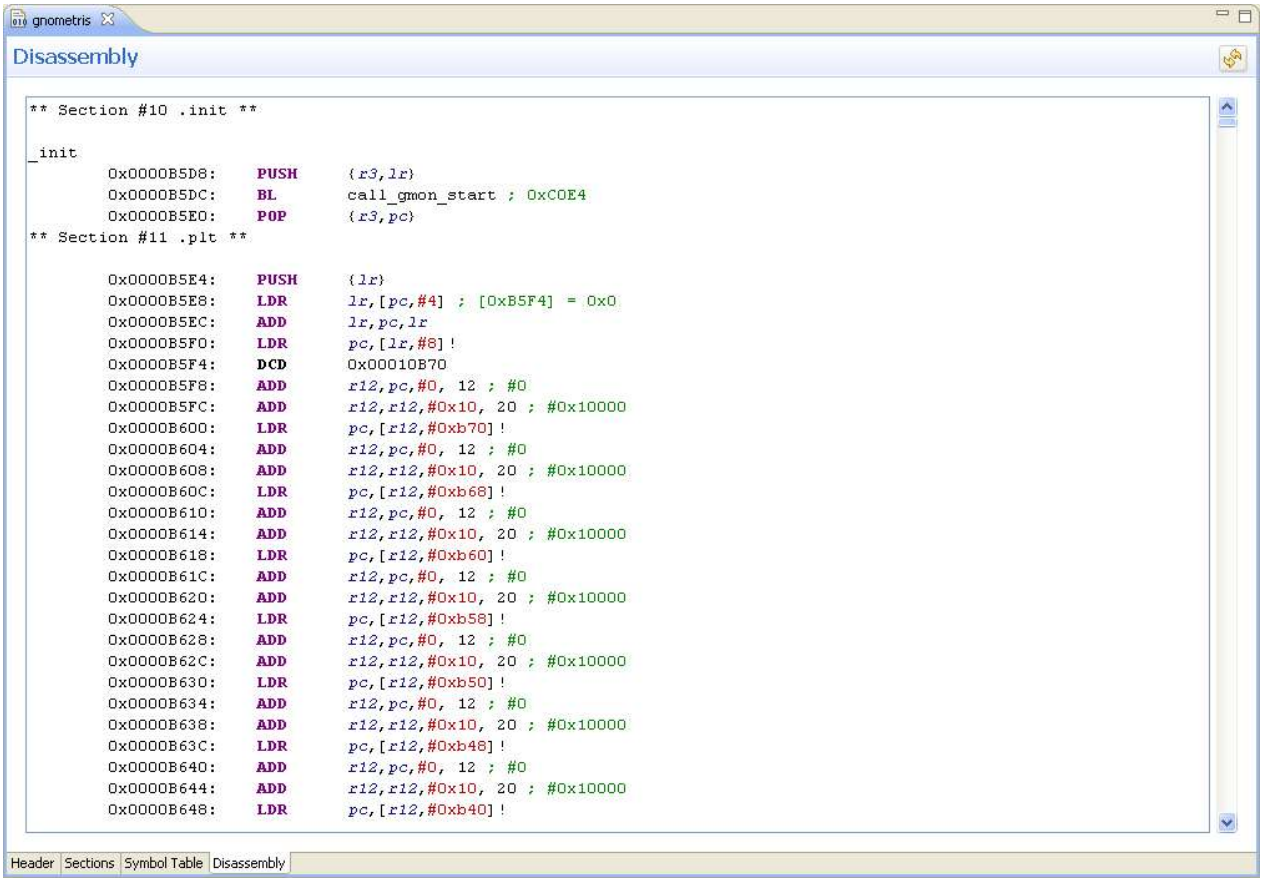
The **Disassembly** tab displays the output with syntax highlighting. The color schemes and syntax preferences use the same settings as the Arm assembler editor.

There are several keyboard combinations that you can use to navigate around the output:

- Use Ctrl+F to open the **Find** dialog box to search the output.
- Use Ctrl+Home to move the focus to the beginning of the output.
- Use Ctrl+End to move the focus to the end of the output.
- Use Page Up and Page Down to navigate through the output one page at a time.

You can also right-click in the **Disassembly** view and select the **Copy** and **Find** options in the context menu.

Figure 7-5: Disassembly tab



7.10 About the scatter file editor

The scatter file editor enables you to easily create and edit scatter files for use with the Arm linker to construct the memory map of an image.

It provides a text editor, a hierarchical tree, and a graphical view of the regions and output sections of an image. You can change the default syntax formatting and color schemes in the **Preferences** dialog box.

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

The scatter file editor displays the following tabs:

Source

Textual view of the source code with syntax highlighting and formatting.

Memory Map

A graphical view showing load and execute memory maps. Although these maps are not editable, you can select a load region to show the related memory blocks in the execution regions.

The scatter file editor also provides a hierarchical tree with associated toolbar and context menus using the **Outline** view. Clicking on a region or section in the **Outline** view moves the focus of the editor to the relevant position in your code. If this view is not visible, select **Show View > Outline** from the **Window** menu.


Note

The linker documentation for Arm® Compiler for Embedded describes in more detail how to use scatter files.

Before you can use a scatter file you must add the `--scatter=file` option to the project in the **C/C++ Build > Settings > Tool settings > Arm Linker > Image Layout** panel of the **Properties** dialog box.

7.11 Creating a scatter file

Create a scatter file to specify more complex memory maps that cannot be specified using compiler command-line memory map options.

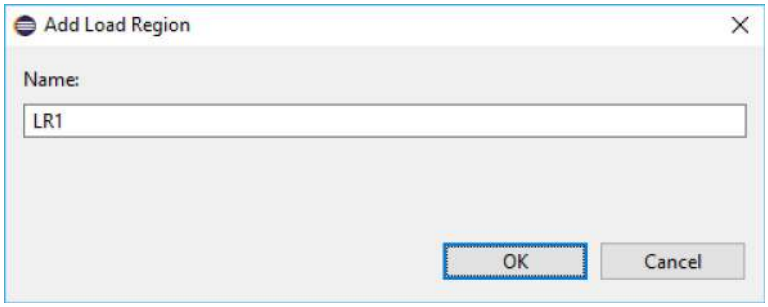
Before you begin

Before you can use a scatter file, you must add the `--scatter=file` option to the project in the **C/C++ Build > Settings > Tool settings > Arm Linker > Image Layout** panel of the **Properties** dialog box.

Procedure

1. Open an existing project, or create a new project.
2. In your project, add a new empty text file with the extension `.scat`. For example `scatter.scat`.
3. In the **Outline** view, click the **Add load region** toolbar icon, or right-click and select **Add load region** from the context menu.
4. Enter a load region name, for example, **LR1**.

Figure 7-6: Add load region dialog box

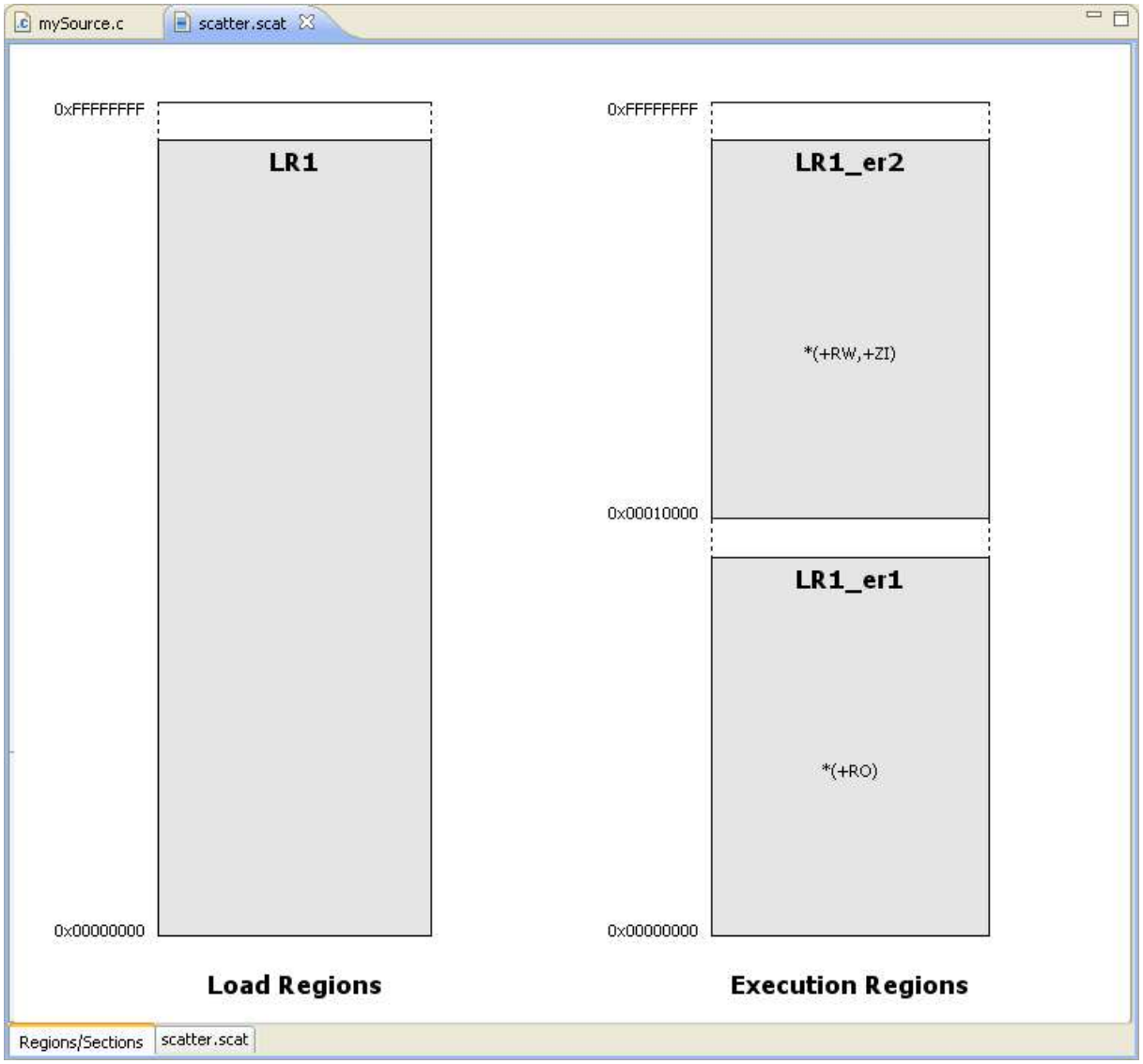


5. Click **OK**.
6. Modify the load region as shown in the following example.

```
LR1 0x0000 0x8000
{
   LR1_er1 0x0000 0x8000
   {
       * (+RO)
   }
   LR1_er2 0x10000 0x6000
   {
       * (+RW,+ZI)
   }
}
```

7. Select the **Regions/Sections** tab to view a graphical representation.

Figure 7-7: Graphical view of a simple scatter file



8. Save your changes.

7.12 Importing a memory map from a BCD file

If you have a BCD file that defines a memory map, you can import this into the **Scatter File Editor**.

Before you begin

Before you can use a scatter file, you must add the `--scatter=file` option to the project in the **C/C++ Build > Settings > Tool settings > Arm Linker > Image Layout** panel of the **Properties** dialog box.

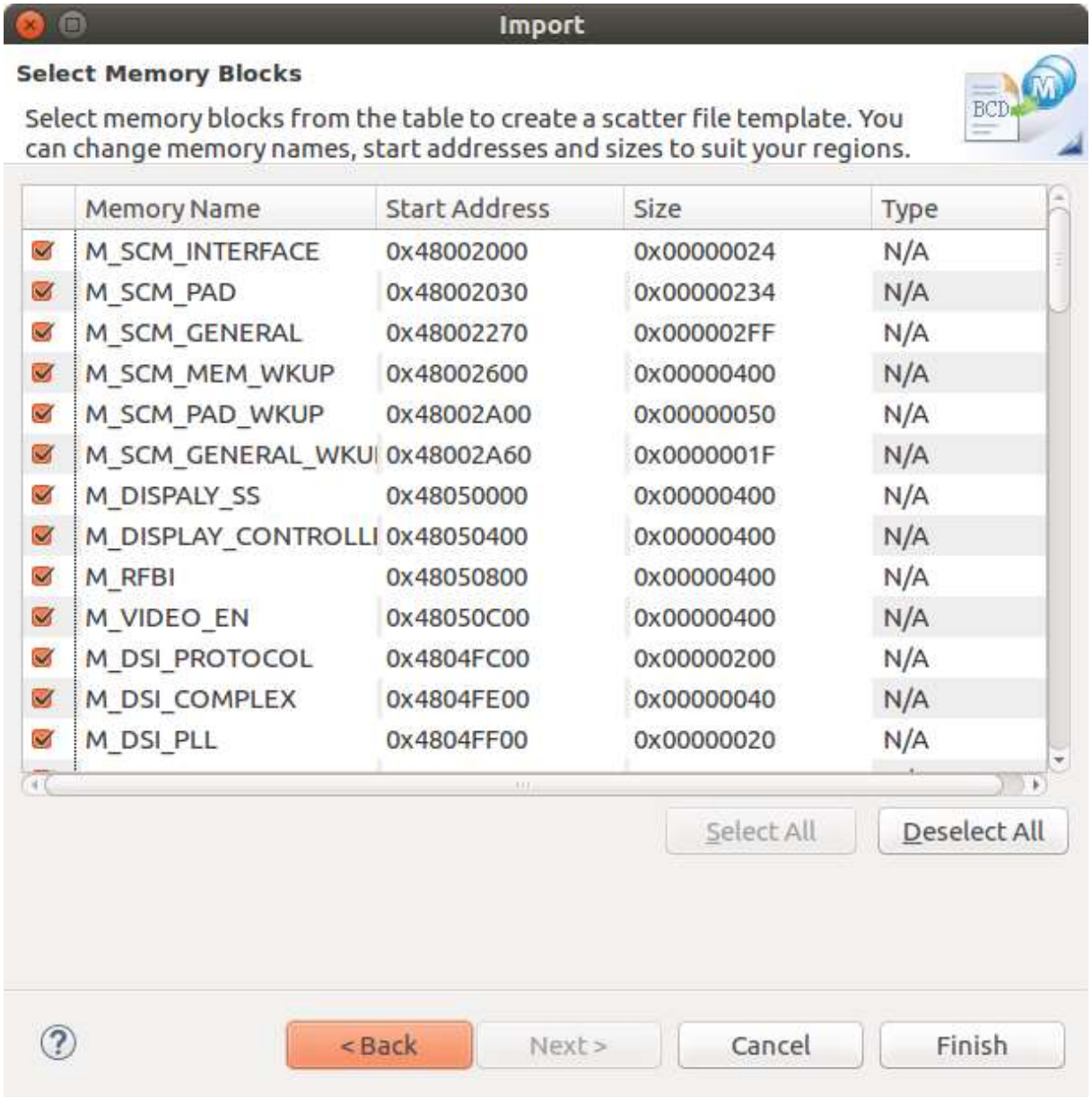
Procedure

1. Select **File > Import > Scatter File Editor > Memory from a BCD File**.
2. Enter the location of the BCD file, or click **Browse...** to select the folder.
3. Select the file that contains the memory map that you want to import.
4. If you want to add specific memory regions to an existing scatter file, select **Add to current scatter file**.

The scatter file must be open and active in the editor view before you can use this option.

5. If you want the wizard to create a new file with the same name as the BCD file but with a `.scat` file extension, select **Create new scatter file template**.
6. Select the destination project folder.
7. By default, all the memory regions are selected. Edit the selections and table content as required.

Figure 7-8: Memory block selection for the scatter file editor



8. Click **Finish**.

8. Debugging code

Describes how to configure and connect to a debug target using Arm® Debugger.

8.1 Overview: Debug connections in Arm Debugger

You can set up connections to debug bare-metal targets, Linux kernel, and Linux applications. You can also use the **Snapshot View** feature to view previously captured application states.

Bare-metal debug connections

Bare-metal targets run without an underlying operating system. To debug bare-metal targets using Arm® Debugger:

- If debugging on hardware, use a debug hardware adapter that is connected to the host workstation and the debug target.
- If debugging on a model, use a CADI-compliant connection between the debugger and a model.

Linux kernel debug connections

Arm Debugger supports source-level debugging of a Linux kernel or a Linux kernel model. For example, you can set breakpoints in the kernel code, step through the source, inspect the call stack, and watch variables. The connection method is similar to bare-metal debug connections.

Linux application debug connections

For Linux application debugging in Arm Debugger, you can connect to your target with a TCP/IP connection.

Before you attempt to connect to your target, ensure that:

- `gdbserver` is present on the target. If `gdbserver` is not installed on the target, either see the documentation for your Linux distribution or check with your provider.
- For AArch64 (Arm®v8-A, Armv8-R AArch64, or Armv9-A) targets, you need to use the AArch64 gdbserver.
- `ssh daemon` (`sshd`) must be running on the target to use the **Remote System Explorer** (RSE) in Development Studio.
- `sftp-server` must be present on the target to use RSE for file transfers.

Snapshot Viewer

Use the **Snapshot Viewer** to analyze and debug a read-only representation of the application state of your processor using previously captured data. This is useful in scenarios where interactive debugging with a target is not possible. For more information, see Working with the Snapshot Viewer.

Related information

Configuring a connection to a bare-metal hardware target on page 119
Configuring a connection to a Linux application using gdbserver on page 122
Configuring a connection to a Linux kernel on page 125
Configuring a connection to an external FVP for bare-metal application debug on page 116
Working with the Snapshot Viewer
About the Snapshot Viewer

8.2 Using FVPs with Arm Development Studio

A Fixed Virtual Platform (FVP) is a software model of a development platform, including processors and peripherals. FVPs are provided as executables, and some are included in Development Studio.

Depending on your requirements, you can:

- Create a new model configuration
- Configure a connection to an FVP for bare-metal application debug
- From the command-line, configure a connection to an FVP for bare-metal application debug
- Configure a connection to an FVP for Linux application debug
- Configure a connection to an FVP for Linux kernel debug

8.3 Configuring a connection from the command-line to a built-in FVP

You can configure a connection to a Fixed Virtual Platform (FVP) using the command-line only mode available in Arm® Development Studio.

Before you begin

- The FVP model that you connect to must be available in the Development Studio configuration database so that you can select it. If the FVP model is not available, you must first import it and create a new model configuration. See Create a new model configuration for information.
- To load and execute the application on your FVP model using Development Studio, your application must first be built with the appropriate compiler and linker options so that it can run on your model. To locate the options and parameters required to build your application, see the documentation for your compiler and linker.
- You must have the appropriate licenses installed to run your FVP model from the command line.
- If you use the command-line only mode, you can automate 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.

Procedure

1. Open the Arm Development Studio command prompt:
   - On Windows, select **Start > All Programs > Arm Development Studio > Arm Development Studio Command Prompt**.
   - On Linux, add the `<install_directory/bin>` location to your *PATH* environment variable and then open a UNIX bash shell.
2. To connect to the Arm FVP Cortex®-A9x4 FVP model and specify an image to load from your workspace, at the command prompt, enter:
   - On Windows: `debugger --cdb-entry "Arm FVP::VE_Cortex_A9x4::Bare Metal Debug::Bare Metal Debug::Debug Cortex-A9x4 SMP" --image "C:\Users\<user>\developmentstudio-workspace\HelloWorld\Debug\HelloWorld.axf"`
   - On Linux: `debugger --cdb-entry "Arm FVP::VE_Cortex_A9x4::Bare Metal Debug::Bare Metal Debug::Debug Cortex-A9x4 SMP" --image "/home/<user>/developmentstudio-workspace/HelloWorld/Debug/HelloWorld.axf"`

Results

Development Studio starts the Arm FVP Cortex-A9x4 FVP and loads the image. When you are 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.

See Running Arm Debugger from the operating system command-line or from a script for more information about how to use Arm Debugger from the operating system command-line or from a script.

8.4 Configuring a connection to an external FVP for bare-metal application debug

You can use Arm® Development Studio to connect to an external Fixed Virtual Platform (FVP) model for bare-metal debugging.

Before you begin

- The FVP model that you are connecting to must be available in the Development Studio configuration database so that you can select it in the **Model Connection** dialog box. If the FVP model is not available, you must first import it and create a new model configuration. See Create a new model configuration for information.
- To load and execute the application on your FVP model using Development Studio, your application must first be built with the appropriate compiler and linker options so that it can run on your model. To locate the options and parameters required to build your application, check the documentation for your compiler and linker.
- You must have the appropriate licenses installed to run your FVP model from the command line.

About this task

This task explains how to:

- Create a model connection to connect to the `Base_AEMv8A_AEMv8A` FVP model and load your application on the model.
- Start up the `Base_AEMv8A_AEMv8A` FVP model separately with the appropriate settings.
- Start a debug session in Development Studio to connect and attach to the running `Base_AEMv8A_AEMv8A` FVP model.

- Configuring a connection to a built-in FVP model follows a similar sequence of steps. Development Studio launches built-in FVPs automatically when you start up a debug connection.
- FVPs available with your edition of Development Studio are listed under the **Arm FVP (Installed with Arm DS)** tree. To see which FVPs are available with your license, compare Arm Development Studio editions.

Procedure

1. From the Arm Development Studio main menu, select **File > New > Model Connection**.
2. In the **Model Connection** dialog box, specify the details of the connection:
   a) Give the connection a name in **Debug connection name**, for example: **my_external_fvp_connection**.
   b) If you want to associate the connection to an existing project, select **Associate debug connection with an existing project** and click **Next**.
   c) In **Target Selection** browse and select `Base_AEMv8A_AEMv8A` and click **Finish** to complete the initial configuration of the connection.
3. In the displayed **Edit Configuration** dialog box, use the **Connection** tab to select the target and connection settings:
   a) In the **Select target** panel confirm the target selected.
   b) If required, specify **Model parameters** under **Connections**.
   c) If required, **Edit** the Debug and Trace Services Layer (DTSL) settings in the **DTSL Configuration** dialog box to configure additional debug and trace settings for your target.
4. Use the **Files** tab to specify your application and additional resources to download to the target:
   a) In **Target Configuration > Application on host to download**, specify the application that you want to load on the model.
   b) If you want to debug your application at source level, select **Load symbols**.
   c) If you want to load additional resources, for example, additional symbols or peripheral description files from a directory, use the **Files** area to add them. Click **+** to add resources, click **-** to remove resources.
5. Use the **Debugger** tab to configure debugger settings.
   a) In the **Run control** area:
   - Choose 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**.
   - If you need to run target or debugger initialization scripts, select the relevant options and specify the script paths.

- If you need 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. If you want to change the default location, deselect the **Use default** option under **Host working directory** and specify a new location.
c) In the **Paths** area, use the **Source search directory** field to enter any directions on the host to search for your application files.
d) If you need to use additional resources, click **Add resource (+)** to add resources, click **Remove resources (-)** to remove resources.

6. If required, specify arguments to pass to the `main()` function. The methods of passing arguments are described in About passing arguments to main().
7. If required, 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 **Close** to save the configuration settings and close the **Debug Configurations** dialog box.
   You have now created a debug configuration to connect to the `Base_AEMv8A_AEMv8A` FVP target. You can view this debug configuration in the **Debug Control** view.
9. The next step is to start up the `Base_AEMv8A_AEMv8A` FVP with the appropriate settings so that Development Studio can connect to it when you start your debugging session.
   a) Open a terminal window and navigate to the installation directory of the `Base_AEMv8A_AEMv8A` FVP.
   b) Start up the `Base_AEMv8A_AEMv8A` separately with the appropriate options and parameters.
   For example, to run the `FVP_Base_AEMv8A-AEMv8A.exe` FVP model on Windows platforms, at the command prompt enter:

```
FVP_Base_AEMv8A-AEMv8A.exe -S -C cluster0.NUM_CORES=0x1 -C bp.secure_memory=false -C cache_state_modelled=0
```

   Where:

   - `FVP_Base_AEMv8A-AEMv8A.exe` - The executable for the FVP model on Windows platforms.
   - `-S` or `--cadi-server` - Starts the CADI server so that Arm Debugger can connect to the FVP model.
   - `-C` or `--parameter` - Sets the parameter you want to use when running the FVP model.
   - `cluster0.NUM_CORES=0x1` - Specifies the number of cores to activate on the cluster in this instance.
   - `bp.secure_memory=false` - Sets the security state for memory access. In this example, memory access is disabled.
   - `cache_state_modelled=0` - Sets the core cache state. In this example, it is disabled.


Note

The parameters and options that are required depend on your specific requirements. Check the documentation for your FVP to locate the appropriate parameters.

You can find the options and parameters that are used in this example in the Fixed Virtual Platforms FVP Reference Guide. You can also enter `--list-params` after the FVP executable name to print available platform parameters.

The FVP is now running in the background awaiting incoming CADI connection requests from Arm Debugger.

10. In the **Debug Control** view, double-click the debug configuration that you created.
    This action starts the debug connection, loads the application on the model, and loads the debug information into the debugger.
11. Click **Continue running application** to continue running your application.

8.5 Configuring a connection to a bare-metal hardware target

To configure a connection to a bare-metal hardware target, create a hardware debug connection. Then, connect to your hardware target using JTAG or Serial Wire Debug (SWD) using DSTREAM-ST or a similar debug hardware adapter.

Before you begin

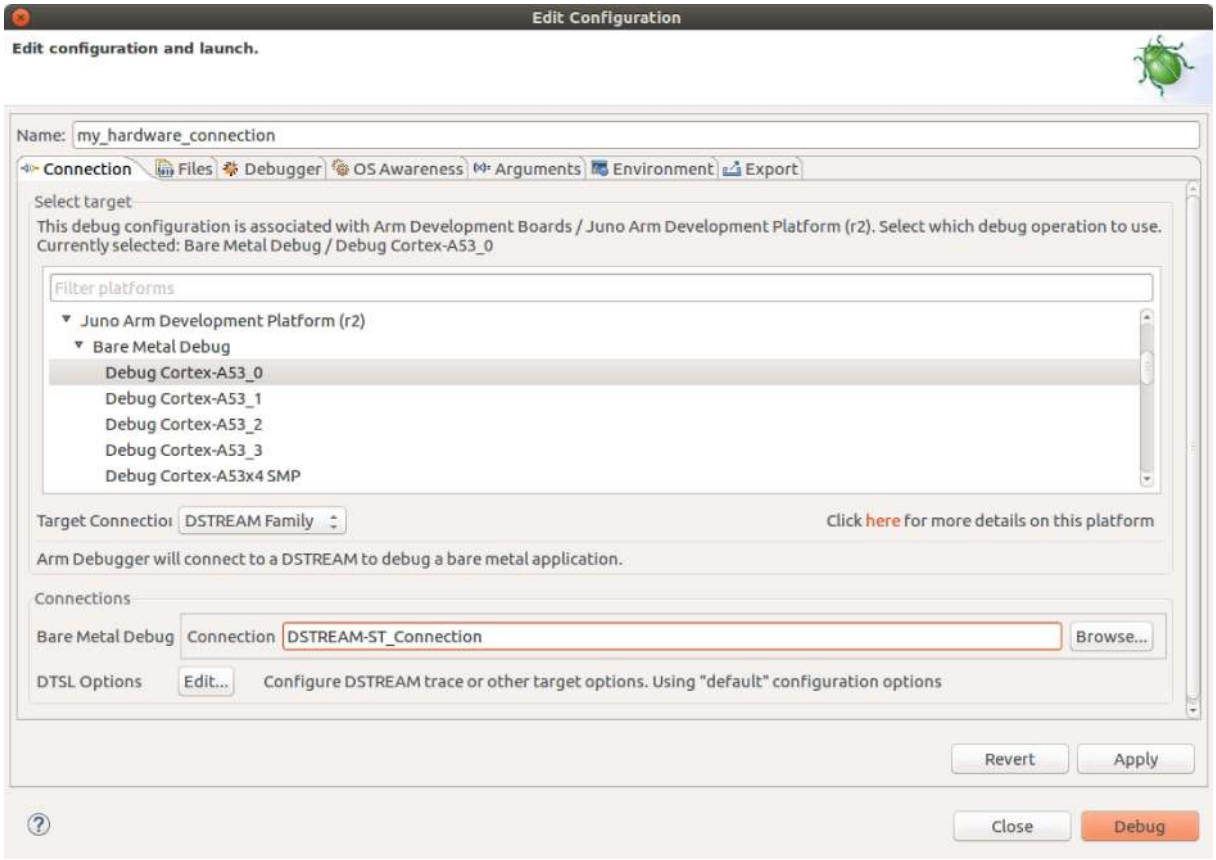
- Ensure that your target is powered on. Refer to the documentation supplied with the target for more information.
- Ensure that the debug hardware adapter connecting your target with your workstation is powered on and working.
- If using DSTREAM-ST, ensure that your target is connected correctly to the DSTREAM-ST unit. If the target is connected and powered on, the **TARGET** LED illuminates green.
- If using DSTREAM, ensure that your target is connected correctly to the DSTREAM unit. If the target is connected and powered on, the **TARGET** LED illuminates green, and the **VTREF** LED on the DSTREAM probe illuminates.

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** enter a debug connection name, for example `my_hardware_connection` and click **Next**.
   b) In **Target Selection** select a target, for example `Juno Arm Development Platform (r2)` and click **Finish** to complete the initial connection configuration.
3. In the displayed **Edit Configuration** dialog box, click the **Connection** tab to specify the target and connection settings:
   a) In the **Select target** panel confirm the target selected.
   b) Select your debug hardware unit in the **Target Connection** list. For example, **DSTREAM Family**.
   c) If required, **Edit** the Debug and Trace Services Layer (DTSL) settings in the **DTSL Configuration Configuration** dialog box to configure additional debug and trace settings for your target.

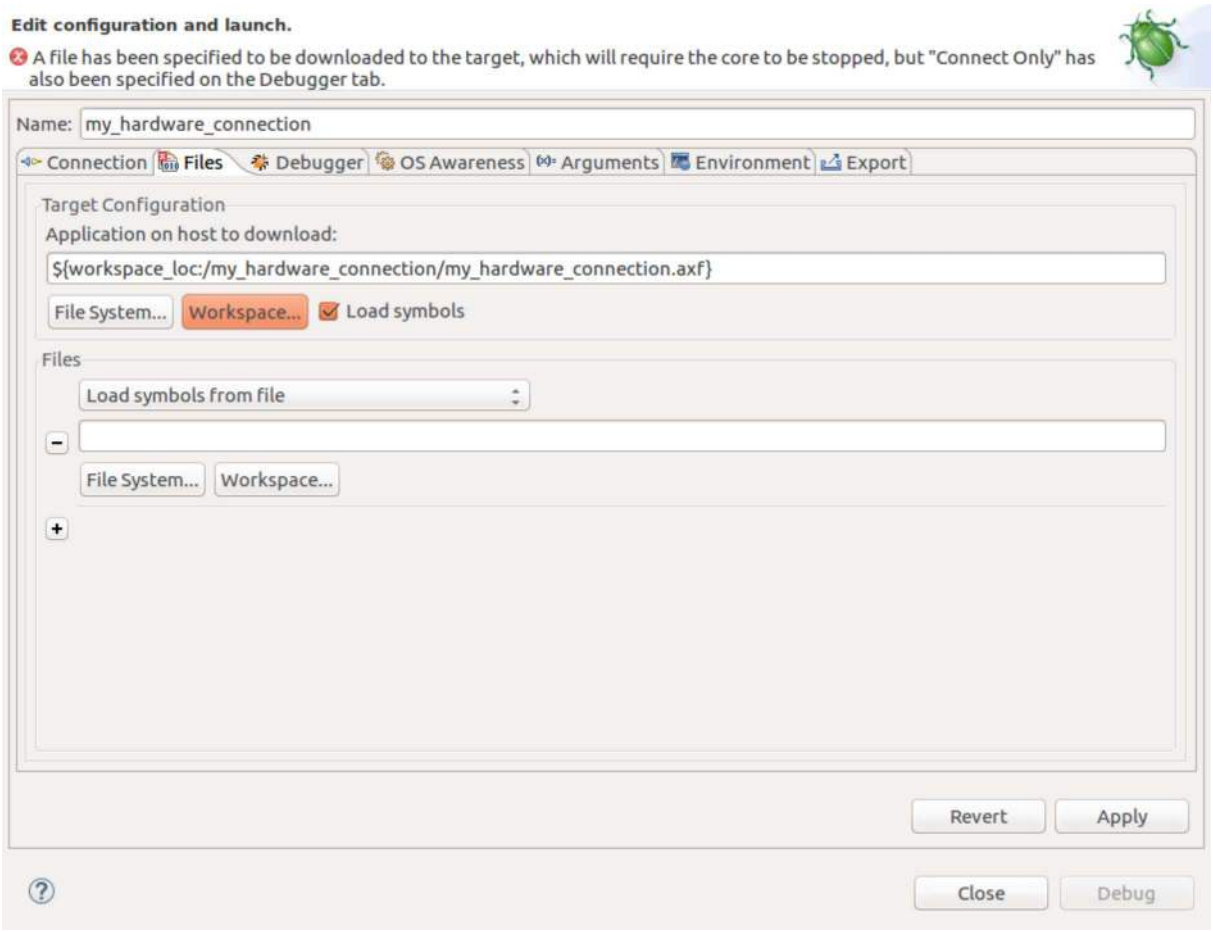
d) In the **Connections** area, enter the **Connection** name or IP address of your debug hardware adapter. If your connection is local, click **Browse** and select the connection using the **Connection Browser**.

Figure 8-1: Edit the Connection tab



4. Click the **Files** tab to specify your application and additional resources to download to the target:
   a) If you want 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) If you want to debug your application at source level, select **Load symbols**.
   c) If you want 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.

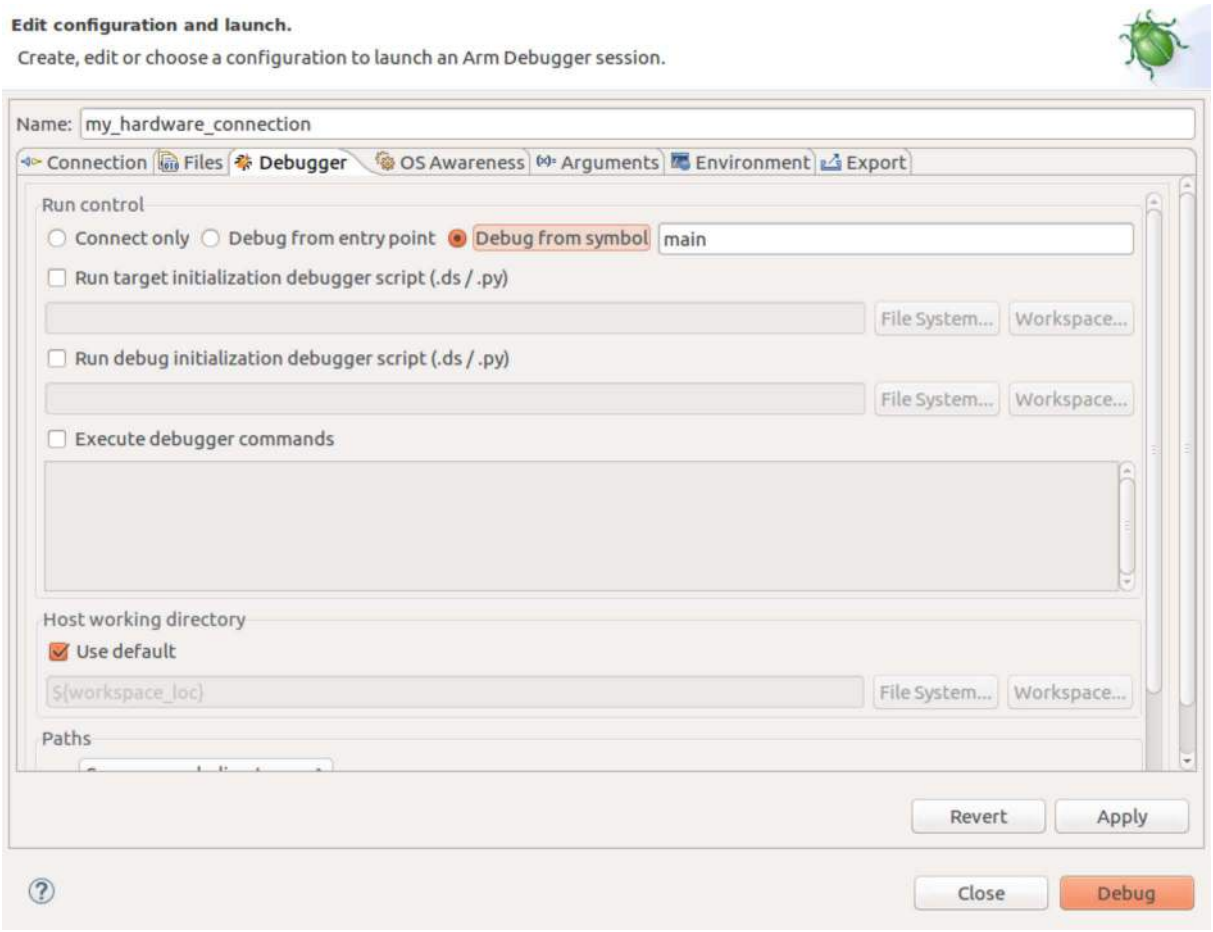
Figure 8-2: Edit the Files tab



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**.
   - If you need to run target or debugger initialization scripts, select the relevant options and specify the script paths.
   - If you need 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. If you want 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 to search for files of your application in the **Source search directory** field.
   d) If you need to use additional resources, click **Add resource (+)** to add resources, click **Remove resources (-)** to remove resources.

Figure 8-3: Edit the Files tab



6. If required, specify arguments to pass to the `main()` function. The methods of passing arguments are described in About passing arguments to main().
7. If required, use the **Environment** tab to create and configure environment variables to pass into the launch configuration when it is executed.
8. Click **Apply** to save the configuration settings.
9. Click **Debug** to connect to the target and start the debugging session.

8.6 Configuring a connection to a Linux application using gdbserver

For Linux application debugging, you can configure Arm® Debugger to connect to a Linux application using **gdbserver**.

Before you begin

- Set up your target with an Operating System (OS) installed and booted. Refer to the documentation supplied with your target for more information.

- Obtain the target IP address or name for the connection between the debugger and the debug hardware adapter. If the target is in your local subnet, click **Browse** and select your target.
- If required, set up a Remote Systems Explorer (RSE) connection to the target.

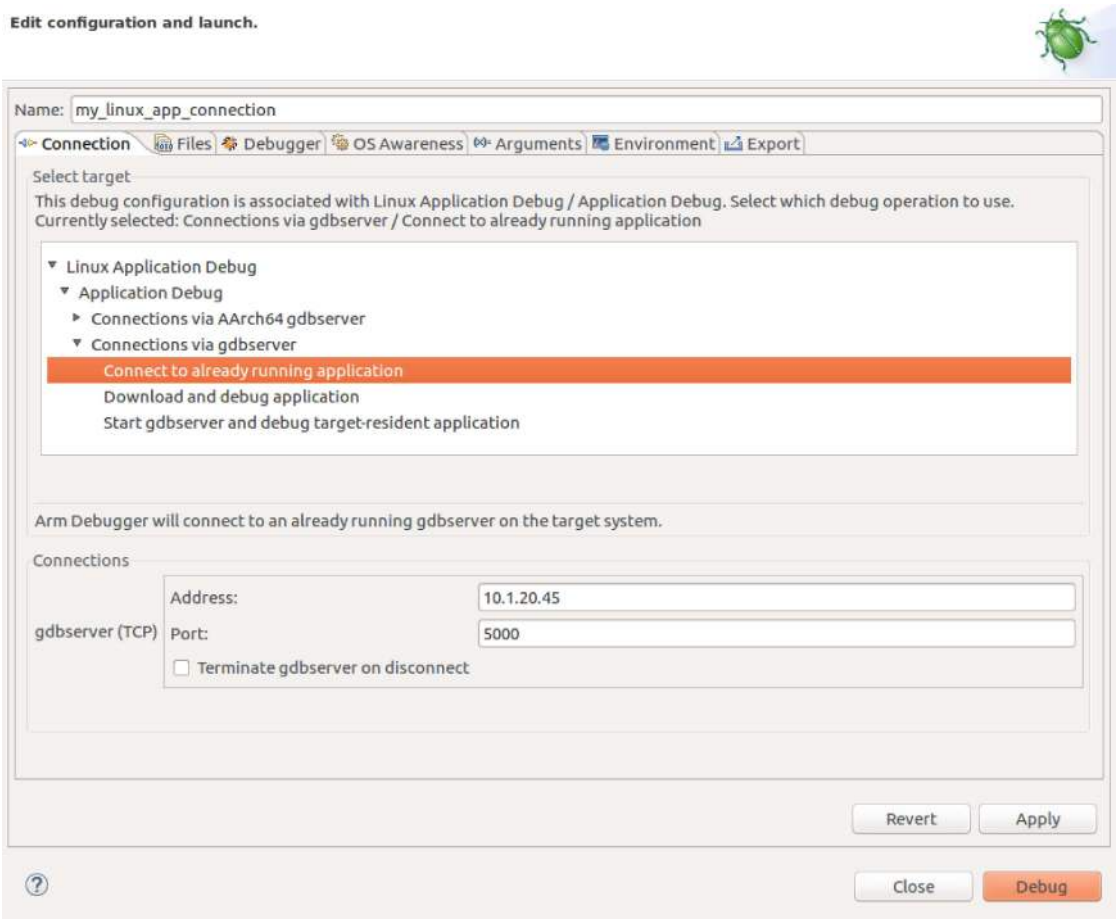

Note

- If you are connecting to an already running **gdbserver**, then you must ensure that it is installed and running on the target. To run **gdbserver** and the application on the target use: `gdbserver port path/myApplication`. Where `port` is the connection port between **gdbserver** and the application and `path/myApplication` is the application that you want to debug.
- If you are connecting to an AArch64 (Arm®v8-A, Armv8-R AArch64, or Armv9-A) target, select the options under **Connect via AArch64 gdbserver**.

Procedure

1. From the Arm Development Studio main menu, select **File > New > Linux Application Connection**.
2. In the **Linux Application Connection** dialog box, specify the details of the connection:
   a) Give the debug connection a name, for example **my_linux_app_connection**.
   b) If using an existing project, select **Use settings from an existing project** option.
   c) Click **Finish**.
3. In the **Edit Configuration** dialog box displayed:
   - If you want to connect to a target with the application and gdbserver already running on it:
     a. In the **Connection** tab, select **Connect to already running application**.
     b. In the **Connections** area, specify the address and port details of the target.
     c. If you want to terminate the gdbserver when disconnecting from the FVP, select **Terminate gdbserver on disconnect**.

Figure 8-4: Edit Linux app connection details



d. In the **Files** tab, use the **Load symbols from file** option in the **Files** panel to specify symbol files.

e. In the **Debugger** tab, specify the actions that you want the debugger to perform after connecting to the target.

f. If required, click the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.

g. If required, click the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.

- If you want to download your application to the target system and then start a gdbserver session to debug the application, select **Download and debug application**. This connection requires that `ssh` and `gdbserver` is available on the target.

  a. In the **Connections** area, specify the address and port details of the target you want to connect to.

  b. In the **Files** tab, specify the **Target Configuration** details:

     - Under **Application on host to download**, select the application to download onto the target from your host filesystem or workspace.

- Under **Target download directory**, specify the download directory location.
- Under **Target working directory**, specify the target working directory.
- If required, use the **Load symbols from file** option in the **Files** panel to specify symbol files.

c. In the **Debugger** tab, specify the actions that you want the debugger to perform after it connects to the target.

d. If required, click the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.

e. If required, click the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.

- If you want to connect to your target, start gdbserver, and then debug an application already present on the target, select **Start gdbserver and debug target resident application**, and configure the options.

a. In the **Model parameters** area, the **Enable virtual file system support** option maps directories on the host to a directory on the target. The Virtual File System (VFS) enables the FVP to run an application and related shared library files from a directory on the local host.

- The **Enable virtual file system support** option is selected by default. If you do not want virtual file system support, deselect this option.
- If the **Enable virtual file system support** option is enabled, your current workspace location is used as the default location. The target sees this location as a writable mount point.

b. In the **Files** tab, specify the location of the **Application on target** and the **Target working directory**. If you need to load symbols, use the **Load symbols from file** option in the **Files** panel.

c. In the **Debugger** tab, specify the actions that you want the debugger to perform after connecting to the target.

d. If required, click the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.

e. If required, click the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.

4. Click **Apply** to save the configuration settings.
5. Click **Debug** to connect to the target and start debugging.

8.7 Configuring a connection to a Linux kernel

Use these steps to configure a connection to a Linux target and load the Linux kernel into memory. The steps also describe how to add a pre-built loadable module to the target.

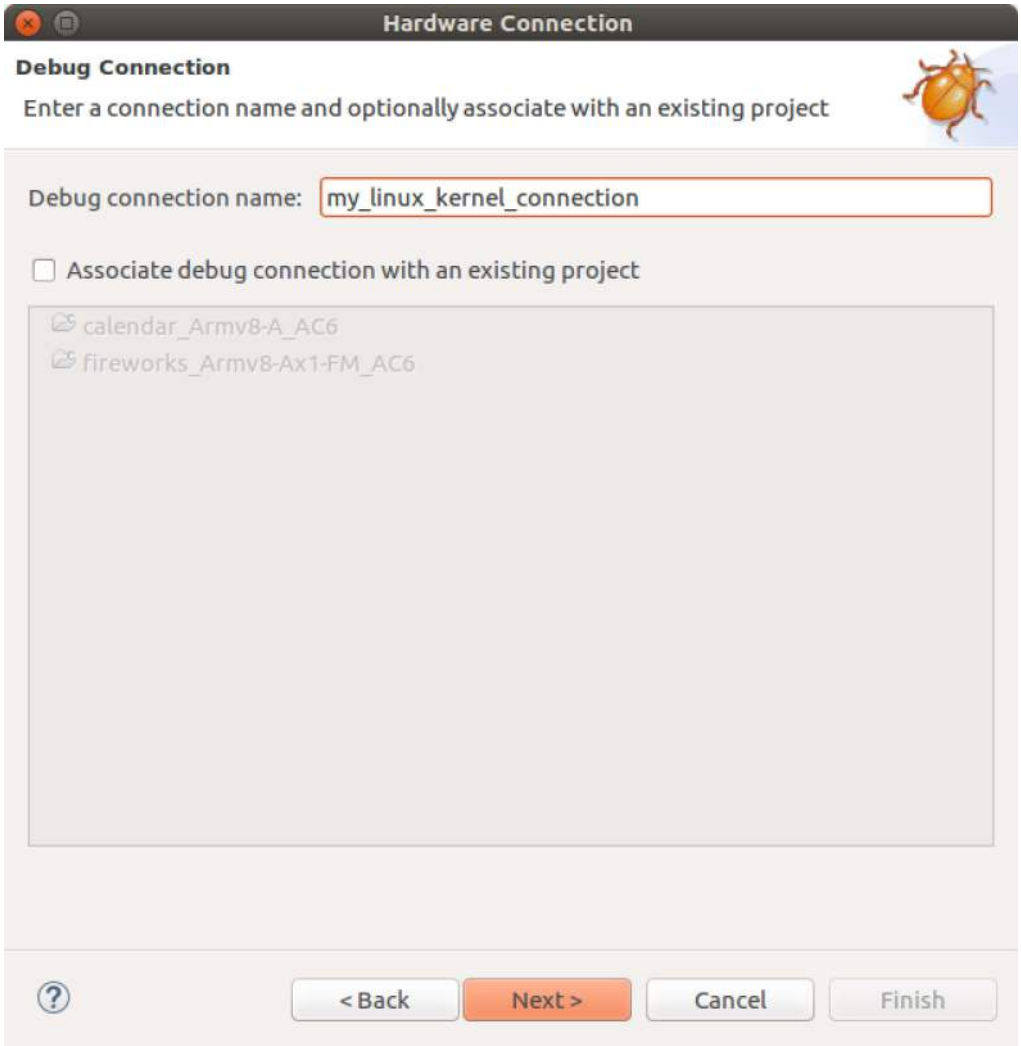
Before you begin

For a Linux kernel module debug, a Remote Systems Explorer (RSE) connection to the target might be required. If so, you must know the target IP address or name.

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_linux_kernel_connection** and click **Next**.
   b) In **Target Selection** select a target, for example `Juno Arm Development Platform (r2)` and click **Finish** to complete the initial configuration of the connection.

Figure 8-5: Name the Linux kernel connection



3. In the **Edit Configuration** dialog box, use the **Connection** tab to specify the target and connection settings:
   a) In the **Select target** panel, browse and select **Linux Kernel and/or Device Driver Debug** operation, and further select the processor core you require.
   b) Select your debug hardware unit in the **Target Connection** list. For example, **DSTREAM Family**.
   c) If you need to, **Edit** the Debug and Trace Services Layer (DTSL) settings in the **DTSL Configuration Editor** to configure additional debug and trace settings for your target.

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**.

4. Use the **Files** tab to specify your application and additional resources to download to the target:
   a) If you want 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) If you want to debug your application at source level, select **Load symbols**.
   c) If you want to load additional resources, for example, additional symbols or peripheral description files from a directory, add them in the **Files** area. Click **Add resource** to add resources, click **Remove resources** to remove resources.
5. Select the **Run control** area in the **Debugger** tab to configure debugger settings:
   a) Select **Connect only** and set up initialization scripts as required.


Note

Operating System (OS) support is automatically enabled when a Linux kernel `vmlinux` symbol file is loaded into the debugger from the Arm Debugger launch configuration. However, you can manually control this using the set os command.

For example, if you want to delay the activation of operating system support until the kernel has booted and the Memory Management Unit (MMU) is initialized, then you can configure a connection that uses a target initialization script to disable operating system support.

   b) Select **Execute debugger commands** option.
   c) In the field provided, enter commands to load debug symbols for the kernel and any kernel modules that you want to debug, for example:

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

add-symbol-file <path>/modex.ko
```


Note

- The path to the `vmlinux` must be the same as your build environment.
- In the example above, the kernel image is called `vmlinux`, but this could be named differently depending on your kernel image.
- In the example above, `S:0` loads the symbols for secure space with `0` offset. The offset and memory space prefix is dependent on your target. When working with multiple memory spaces, ensure that you load the symbols for each memory space.

   d) The debugger uses your workspace as the default working directory on the host. If you want to change the default location, deselect the **Use default** option under **Host working directory** and specify a new location.
   e) In the **Paths** area, specify any directories on the host to search for files of your application using the **Source search directory** field.
   f) If you need to use additional resources, click **Add resource (+)** to add resources, click **Remove resources (-)** to remove resources.
6. If required, specify arguments to pass to the `main()` function. The methods of passing arguments are described in About passing arguments to main().

7. If required, use the **Environment** tab to create and configure environment variables to pass into the launch configuration when it is executed.
8. Click **Apply** to save the configuration settings.
9. Click **Debug** to connect to the target and start the debugging session.

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

8.8 Configuring trace for bare-metal or Linux kernel targets

You can configure trace for bare-metal or Linux kernel targets using the DTSL options that Arm® Debugger provides.

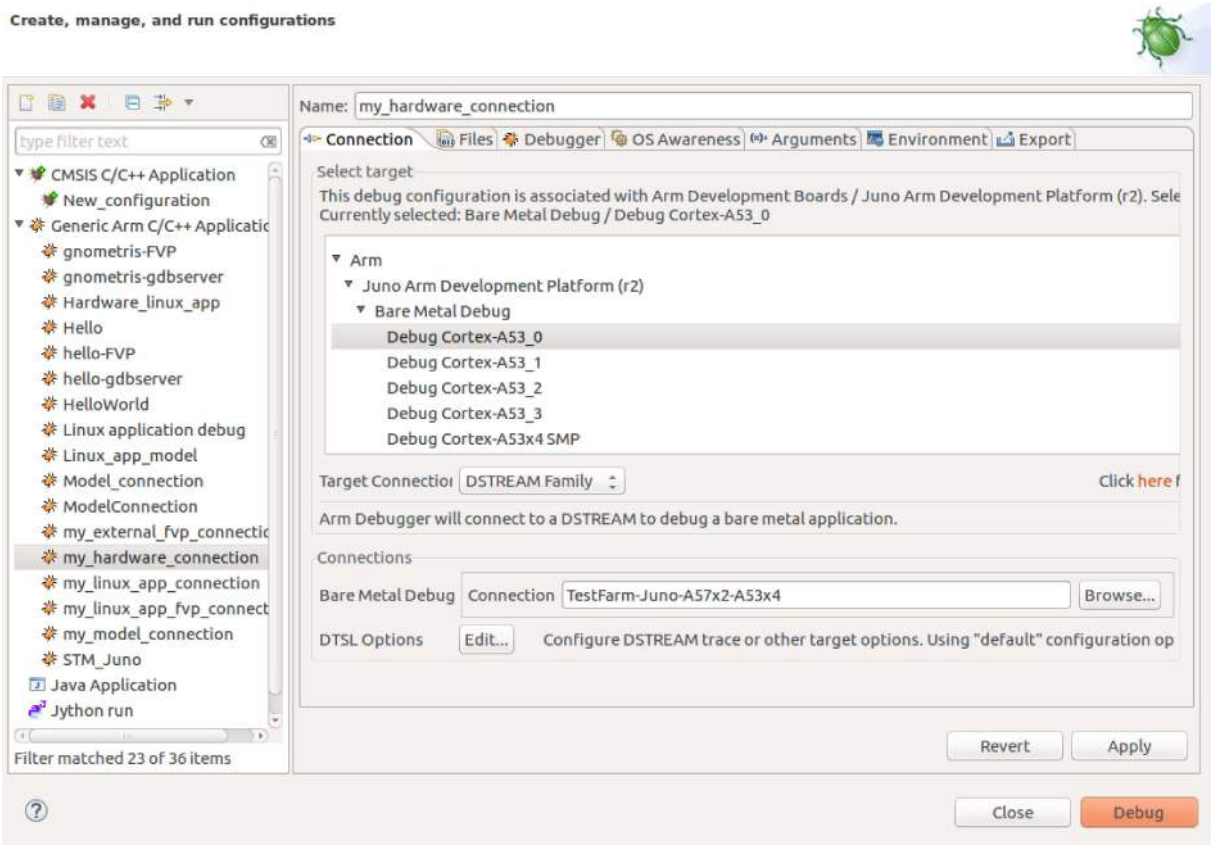
About this task

After configuring trace for your target, you can connect to your target and capture trace data.

Procedure

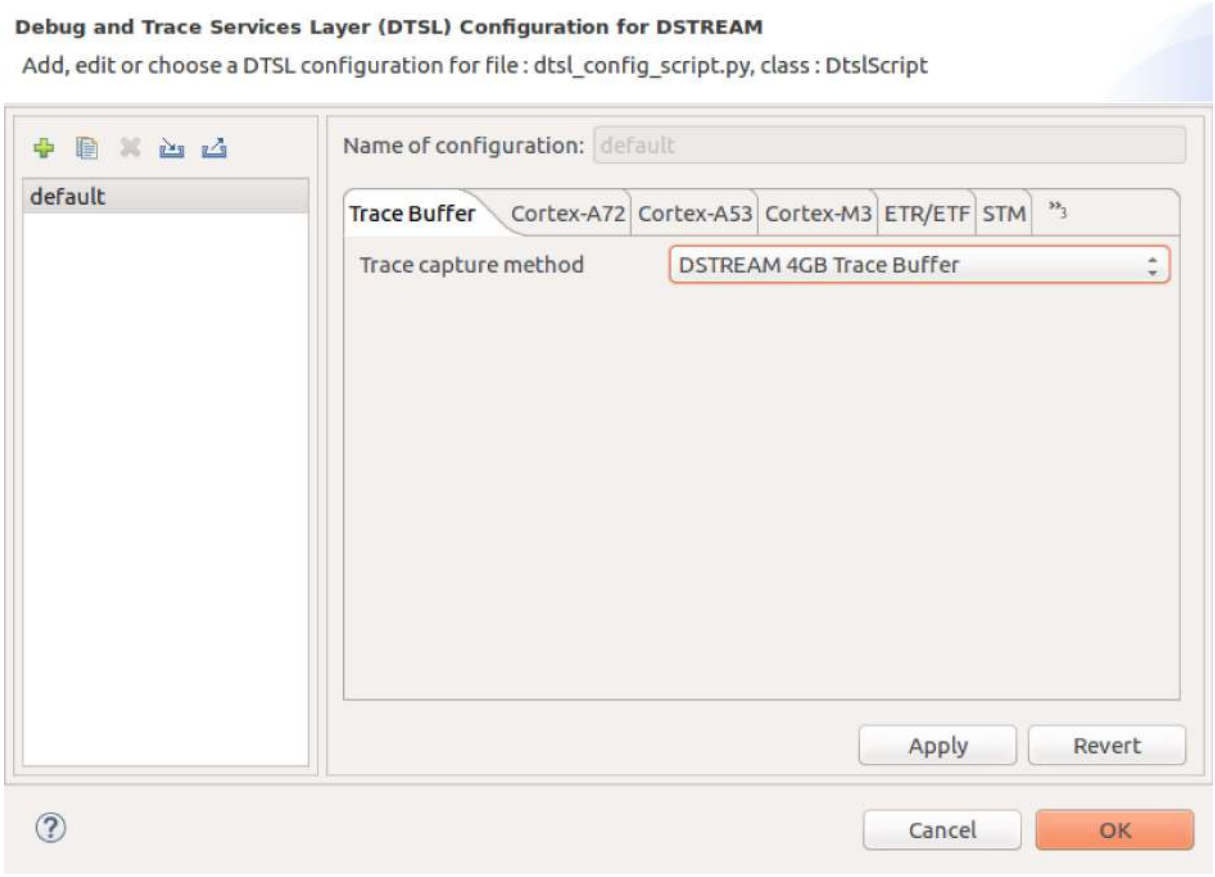
1. In Arm Debugger, select **Window > Perspective > Open Perspective > Other > Development Studio** .
2. Select **Run > Debug Configurations** to open the **Debug Configurations** launcher panel.
3. Select the **Arm Debugger** debug configuration for your target in the left-hand pane.
   If you want to create a new debug configuration for your target, then select **Arm Debugger** from the left-hand pane, and then click the **New** button. Then select your bare-metal or Linux kernel target from the **Connection** tab.

Figure 8-6: Select the debug configuration



4. After selecting your target in the **Connection** tab, click the **DTSL Options Edit** button. This shows the **DTSL Configuration** dialog box where you can configure trace.
5. Depending on your target platform, the **DTSL Configuration** dialog box provides different options to configure trace.

Figure 8-7: Select Trace capture method



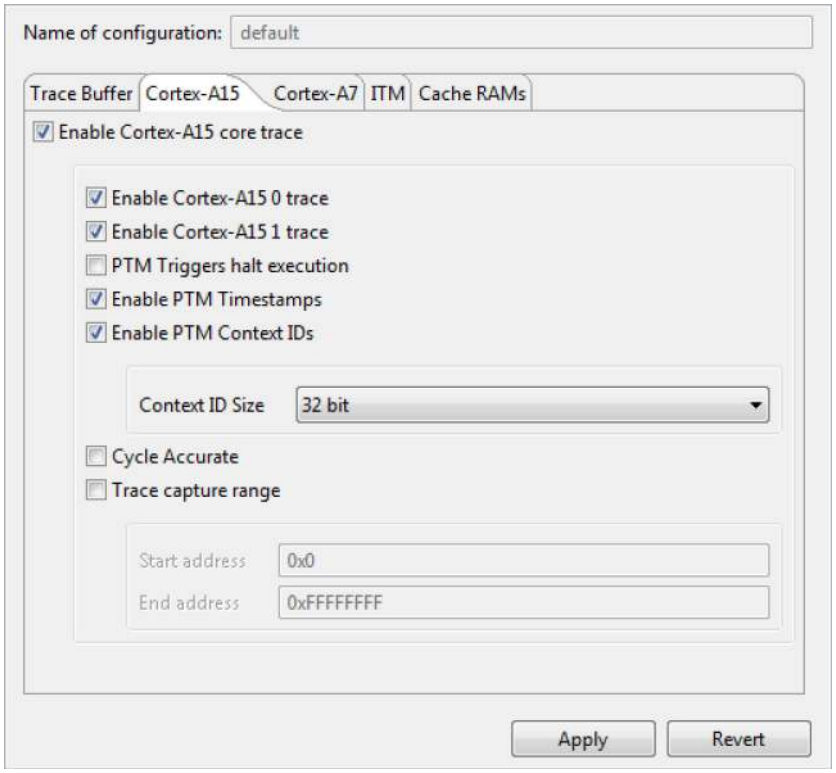
a) For **Trace capture method** select the trace buffer you want to use to capture trace.
b) The **DTSL Configuration** dialog box shows the processors on the target that are capable of trace. Click the processor tab you require. Then, select the option to enable trace for the individual processors you want to capture trace.
c) Select any other trace related options you require in the **DTSL Configuration** dialog box.
d) Click **Apply** and then click **OK**. This configures the debug configuration for trace capture.

6. Use the other tabs in the **DTSL Configuration** dialog box to configure the other aspects of your debug connection.
7. Click **Apply** to save your debug configuration. When you use this debug configuration to connect, run, and stop your target, you can see the trace data in the **Trace** view.

Note

The options to enable trace might be nested. In this example, you must select **Enable Cortex-A15 core trace** to enable the other options. Then you must select **Enable Cortex-A15 0 trace** to enable trace on core 0 of the Cortex®-A15 processor cluster.

Figure 8-8: Select the processors you want to trace



Related information

Configure DSTREAM-PT trace mode

8.9 Configuring an Events view connection to a bare-metal target

The **Events** view allows you to capture and view textual logging information from bare-metal applications. It also allows you to view packets generated by the Data Watchpoint and Trace (DWT) unit on M-profile targets. Logs are captured from your application using annotations that you must add to the source code.

Before you begin

- On M-profile targets, set the registers appropriately to enable the required DWT packets. See the Armv7-M Architecture Reference Manual for more information.
- Annotate your application source code with logging points and recompile it. See the `ITM and Event Viewer Example for Versatile Express Cortex-A9x4` provided with Arm® Development Studio examples for more information.

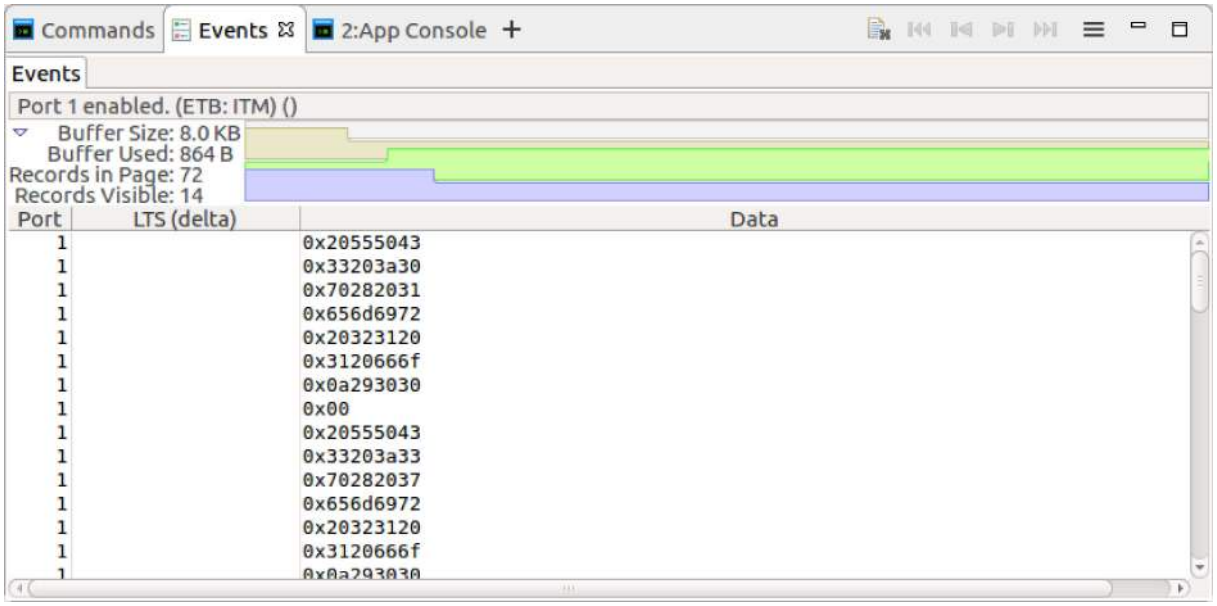
Procedure

1. Select **Debug Configurations...** from the **Run** menu.
2. Select **Generic Arm C/C++ Application** from the configuration tree and then click **New** to create a new configuration.
3. In the **Name** field, enter a suitable name for the new configuration, for example, **events_view_debug**
4. Use the **Connection** tab to specify the target and connection settings:
   a) Select the required platform in the **Select target** panel. For example, **ARM Development Boards > Versatile Express A9x4 > Bare Metal Debug > Debug Cortex-A9x4 SMP**.
   b) Select your debug hardware unit in the **Target Connection** list. For example, **DSTREAM Family**.
   c) In **DTSL Options**, click **Edit** to configure DSTREAM trace and other target options. This displays the **DTSL Configuration** dialog box.
   - In the **Trace Capture** tab, either select **On Chip Trace Buffer (ETB)** (for a JTAG cable connection), or **DSTREAM 4GB Trace Buffer** (for a Mictor cable connection).
   - In the **ITM** tab, enable or disable ITM trace and select any additional settings you require.
5. Click the **Files** tab to define the target environment and select debug versions of the application file and libraries on the host that you want the debugger to use.
   a) In the **Target Configuration** panel, specify your application in the **Application on host to download** field.
   b) If you want to debug your application at source level, select **Load symbols**.
   c) If you want to load additional resources, for example, additional symbols or peripheral description files from a directory, use the **Files** area to add them. Click **+** to add resources, click **-** to remove resources.
6. 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**.
   - If you need to run target or debugger initialization scripts, select the relevant options and specify the script paths.
   - If you need 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. If you want to change the default location, deselect the **Use default** option under **Host working directory** and specify a new location.
   c) In the **Paths** area, specify any directories on the host to search for files of your application using the **Source search directory** field.
   d) If you need to use additional resources, click **Add resource (+)** to add resources, click **Remove resources (-)** to remove resources.
7. If required, specify arguments to pass to the `main()` function. The methods of passing arguments are described in About passing arguments to main().
8. Click **Apply** to save the configuration settings.
9. Click **Debug** to connect to the target. Debugging requires the **Development Studio** perspective. If the **Confirm Perspective Switch** dialog box opens, click **Yes** to switch perspective.

When connected and the **Development Studio** perspective opens, you are presented with all the relevant views and editors.

10. Set up the **Events** view to show output generated by the System Trace Macrocell (STM) and Instruction Trace Macrocell (ITM) events.
    a) From the main menu, select **Window > Show view > Events**
    b) In the **Events** view, click , and select **Events Settings**.
    c) In **Select a Trace Source**, ensure that the trace source matches the trace capture method specified earlier.
    d) Select the required **Ports/Channels**.
    e) On M-profile targets, if required, select any DWT packets.
    f) Click **OK** to close the dialog box.
11. Run the application for a few seconds, and then interrupt it.
    You can view the relevant information in the **Events** view. For example:

Figure 8-9: Events view with data from the ITM source



8.10 Exporting or importing an existing Arm Development Studio launch configuration

In Arm® Development Studio, a launch configuration contains all the information to run or debug a program. 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.

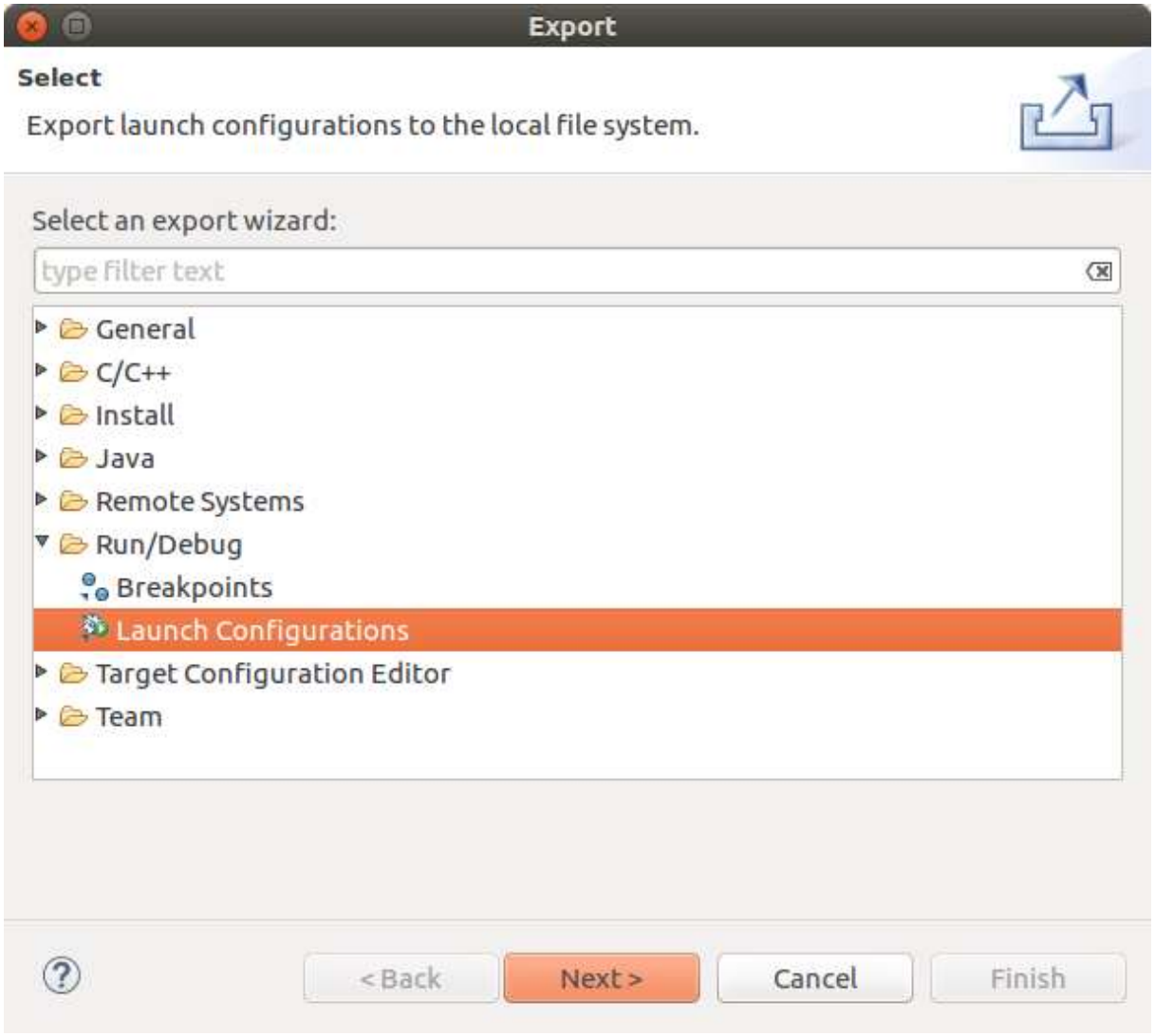

Note

- To use a launch configuration from the Development Studio command-line, you must create a launch configuration file using the Export tab in the **Debug Configurations** dialog box.
- You cannot import Development Studio command-line launch configurations.
- When exporting a launch configuration, Arm Development Studio resolves any Eclipse variables that you have used. Arm Development Studio does not resolve Eclipse variables when scripting or when using the Commands view.

Exporting an existing launch configuration

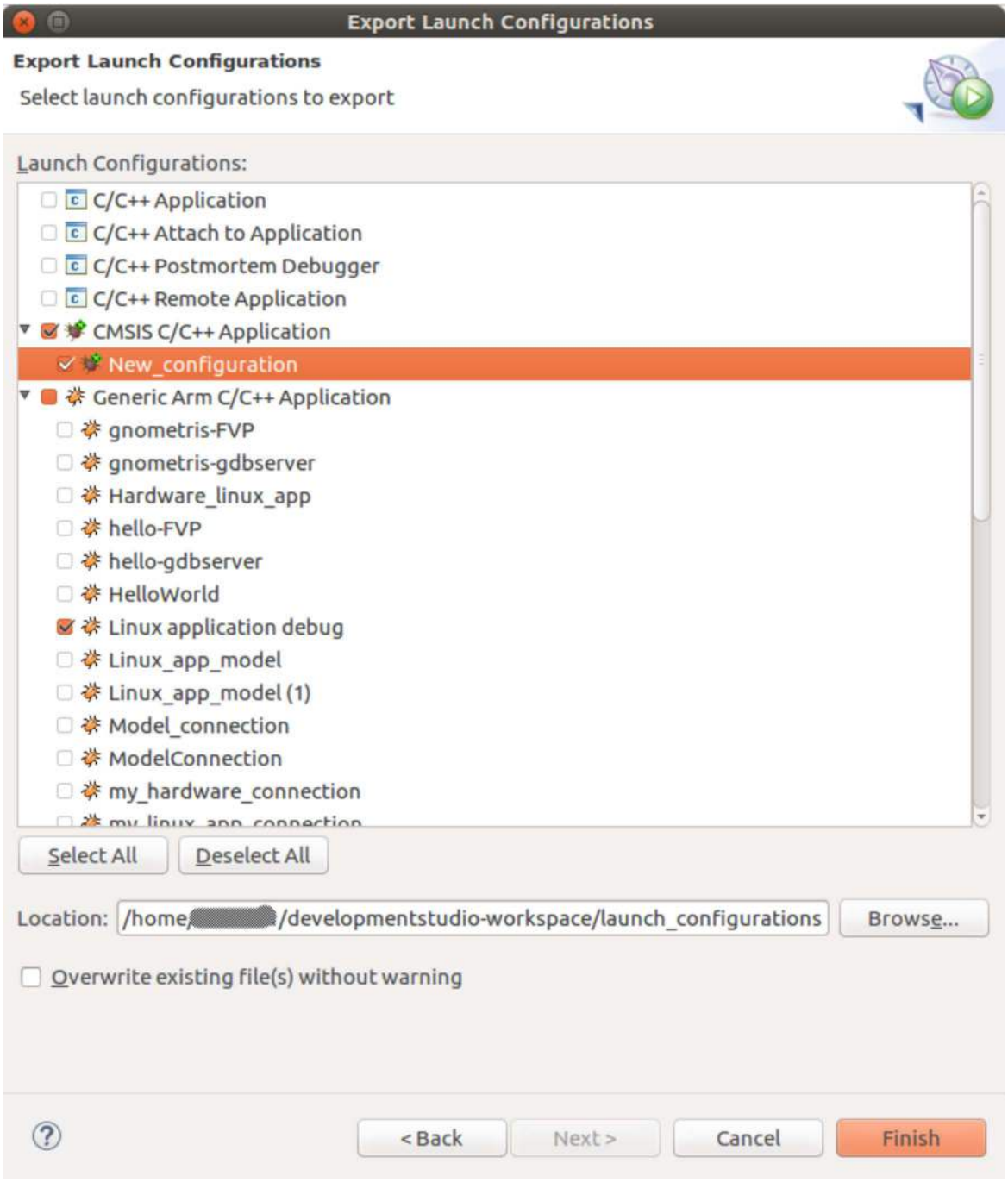
1. From the **File** menu, select **Export...**.
2. In the **Export** dialog box, expand the **Run/Debug** group and select **Launch Configurations**.

Figure 8-10: Export Launch Configuration dialog box



3. Click **Next**.
4. In the **Export Launch Configurations** dialog box:
   a. Depending on your requirements, expand the **CMSIS C/C++ Application** group or the **Generic Arm C/C++ Application** and select one or more launch configurations.
   b. Click **Browse...** and select the required location on your local file system and click **OK**.

Figure 8-11: Select Launch Configurations for export



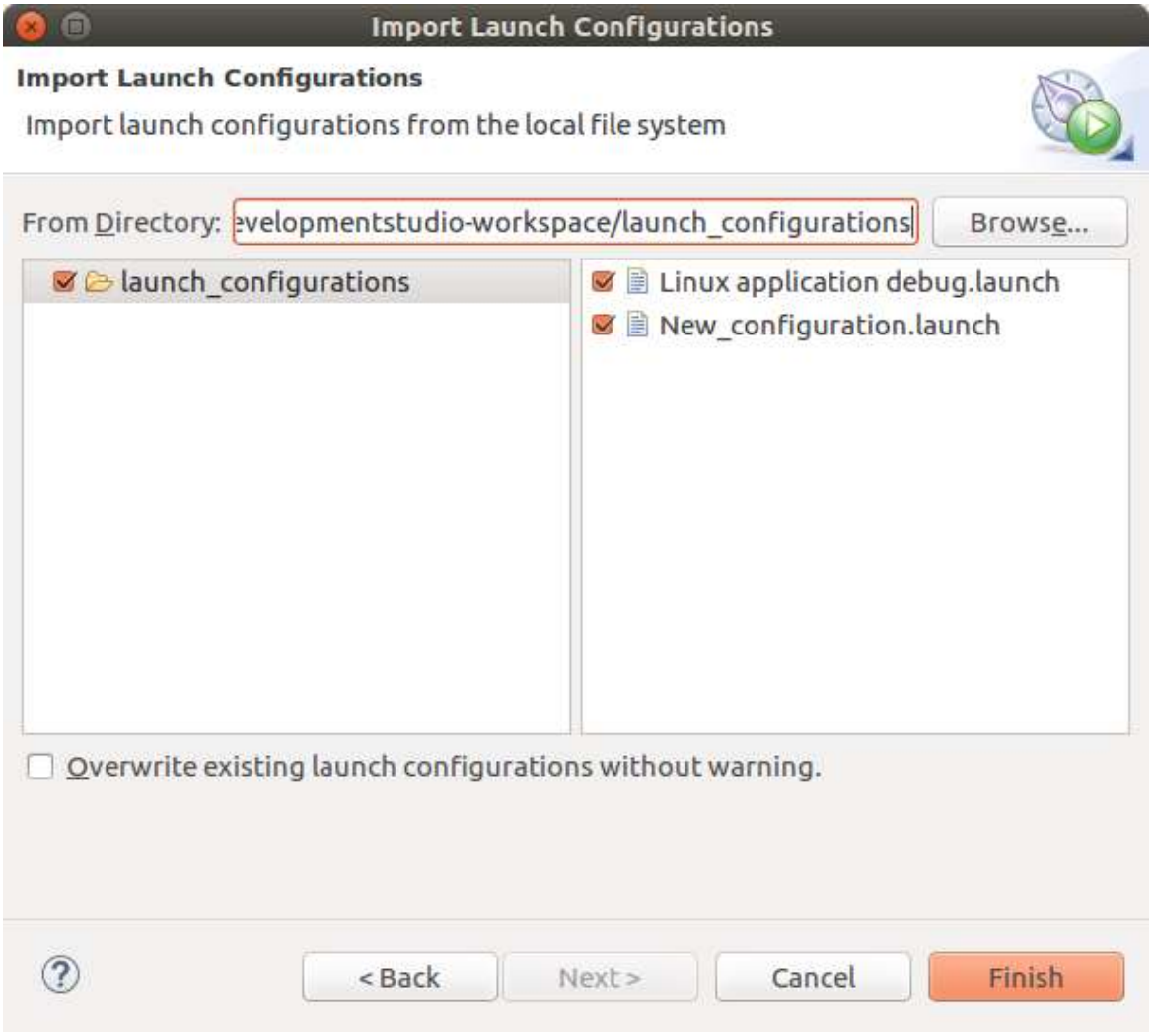
5. If necessary, select **Overwrite existing file(s) without warning**.
6. Click **Finish**.

The launch configuration files are saved in your selected location with an extension of `.launch`.

Importing an existing launch configuration

1. From the **File** menu, select **Import...**.
2. In the **Import** dialog box, expand the **Run/Debug** group and select **Launch Configurations**.
3. Click **Next**.
4. In the **Import Launch Configurations** dialog box:
   a. In **From Directory**, click **Browse** and select an import directory.
   b. In the selection panels, select the folder and the specific launch configurations you want.

Figure 8-12: Import launch configuration selection panel



   c. If necessary, select **Overwrite existing file(s) without warning**.
   d. Click **Finish** to complete the import process.

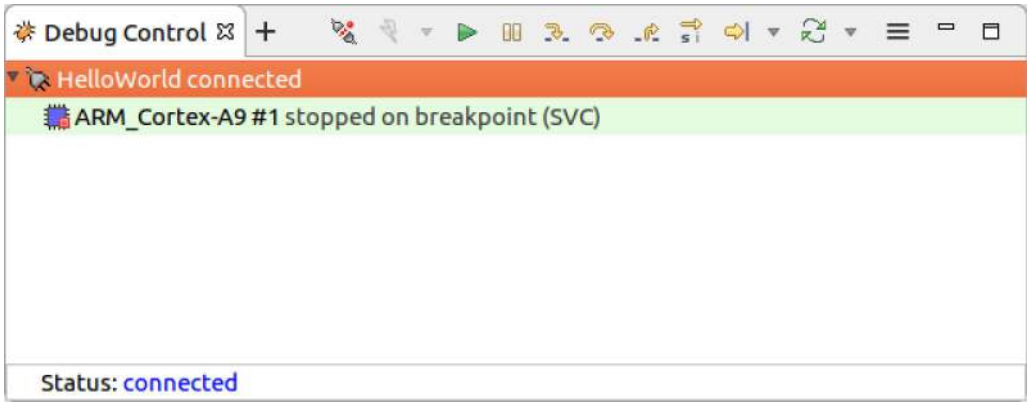
You can view the imported launch configurations in the **Debug Control** panel.

8.11 Disconnecting from a target

To disconnect from a target, you can use either the **Debug Control** or the **Commands** view.

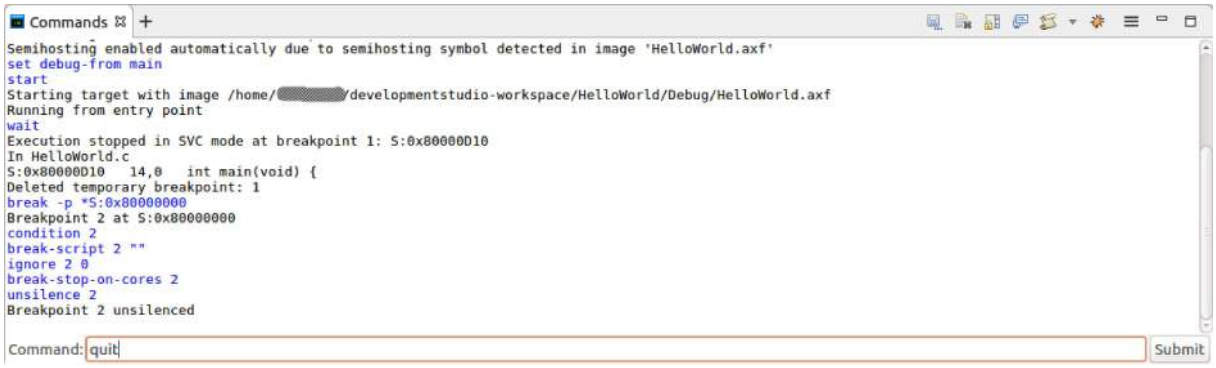
- If you are using the **Debug Control** view, on the toolbar, click  .

  Figure 8-13: Disconnecting from a target using the Debug Control view



- If you are using the **Commands** view, enter **quit** in the **Command** field and click **Submit**.

  Figure 8-14: Disconnecting from a target using the Commands view



The disconnection process ensures that the target's state does not change, except for the following:

- Any downloads to the target are canceled and stopped.
- Any breakpoints are cleared on the target, but are maintained in Arm® Development Studio.
- The DAP (Debug Access Port) is powered down.
- Debug bits in the DSC (Debug Status Control) register are cleared.

If a trace capture session is in progress, trace data continues to be captured even after Arm Development Studio has disconnected from the target.

9. Tutorials

Contains tutorials to help you get started with Arm® Development Studio.

9.1 Tutorial: Hello World

The Hello World tutorial is for new users, taking them through each step in getting their first project up and running.

9.1.1 Open Arm Development Studio for the first time

The first time you open Arm® Development Studio, you are prompted to add your license details. When you have completed the tasks in this section, you are ready to use Arm Debugger.

Before you begin

- Download and install Arm Development Studio, for either:
  - Linux: Installing on Linux
  - Windows: Installing on Windows
- If you or your company has purchased Arm Development Studio, you need one of the following:
  - Arm user-based licensing:

    The license server address or an activation code.
  - FlexNet license management:

    The license file or the license server address and port number.

About this task

Arm Development Studio is available for both Linux and Windows platforms.

Procedure

1. Open Arm Development Studio:
   - On Windows, select **Windows menu > Arm Development Studio <version>**
   - On Linux:
     - GUI: Use your Linux variant's menu system to locate Arm Development Studio.
     - Command line: Run `<installation_directory>/bin/armds_ide`
2. The first time you open Arm Development Studio, the **Product Setup** dialog box opens, which prompts you to add your product license. You can select one of the following:

- **Manage Arm User-Based Licenses** - select this option if you have purchased Arm Development Studio and it is licensed using Arm user-based licensing. After selecting this option, click **Finish** to open the **Arm License Management Utility** dialog box.


Note

Arm user-based licensing is only available to customers with a user-based licensing license. Documentation for user-based licensing is available at https://lm.arm.com. For assistance with user-based licensing issues, visit https://developer.arm.com/support and open a support case.

- **Add product license** - select this option if you have purchased Arm Development Studio and it is licensed by FlexNet licence management. After selecting this option:
  a. Click **Next**.
  b. Enter the location of your license file, or the address and port number of your license server, and click **Next**.
  c. The Arm Development Studio editions that you are entitled to use are listed. Select the edition that you require, and click **Next**.
  d. Check the details on the summary page. If they are correct, click **Finish**.
- **Obtain evaluation license** - select this option if you would like to evaluate the product. After selecting this option:
  a. Click **Next**.
  b. Log into your Developer account using your Arm Developer account email address and password. If you do not have an account, click **Create an account**.
  c. Select a network interface to which your license will be locked.
  d. Click **Finish**.

Results

Arm Development Studio opens. See IDE Overview, which describes the main features of the user interface.


Note

The workspace is automatically set by default, to either:

- Windows: `<userhome>\Development Studio Workspace`
- Linux: `<userhome>/developmentstudio-workspace`

You can change the default location by selecting **File > Switch Workspace**.

9.1.2 Create a project in C or C++

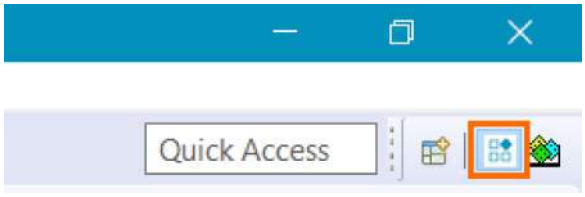
After installing and licensing Arm® Development Studio, we are going to create a simple Hello World C project and show you how to specify the base RAM address for a target. For the

remainder of this tutorial, we are going to use the Arm Compiler for Embedded 6 toolchain and our target is a Cortex®-A53 Fixed Virtual Platform, provided with Arm Development Studio.

Before you begin

- Complete Open Arm Development Studio for the first time
- Ensure you are in the **Development Studio** Perspective. This is the default perspective when Arm Development Studio is first opened. To return to it, click the **Development Studio** button in the top right corner.

Figure 9-1: Screenshot highlighting the button for the Development Studio Perspective.



Procedure

1. To create a new C project, select: **File > New > Project...**.
2. Expand the **C/C++** menu, and select **C project**, then click **Next**.

Note

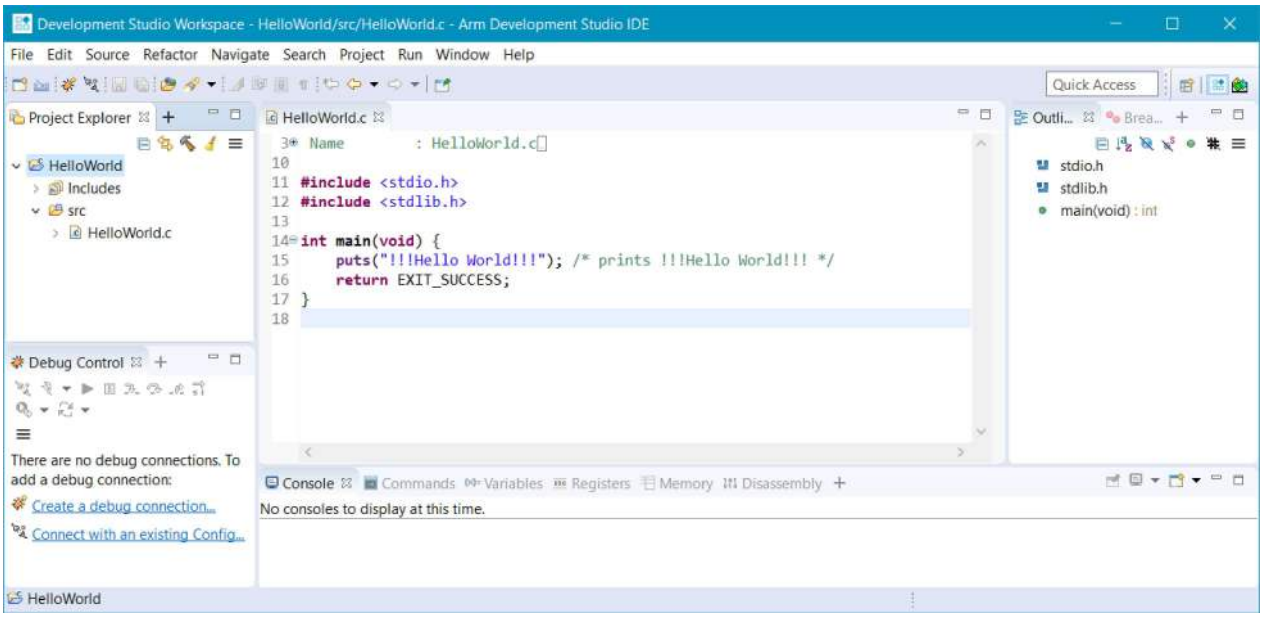
This tutorial also works with a C++ project.

---

3. In the **C Project** dialog box:
   a) In the **Project name** field, enter `HelloWorld`.
   b) Under **Project type**, select **Executable > Hello World ANSI C Project**.
   c) Under **Toolchains**, select **Arm Compiler 6**.
   d) Click **Finish**.

Results

Figure 9-2: The IDE after creating a new project



Next steps

You can add existing source files to your project by dragging and dropping the file into the project folder, or by selecting **File > Import > General > File System**.

9.1.3 Configure your project

Before you build the `HelloWorld` project, you must specify some configuration settings.

Before you begin

Complete Create a project in C or C++

About this task

You must specify:

- The target processor or architecture you want to compile for.
- That the compiler must add debug symbols into the image file, so that the debugger can debug it at source-level.
- The address in RAM in your FVP target where you want the linker to base your image.

This ensures that the application is built and loaded correctly on to your target, and that you can debug the image at source-level.

Procedure

1. In the **Project Explorer** view, right-click the `HelloWorld` project and select **Properties**. The **Properties for HelloWorld** dialog box opens.

2. Add debug symbols into the image file:
   a) Expand **C/C++ Build**, and select **Build Settings**.
   b) Ensure the **Configuration** is set to **Debug [Active]**.
3. Configure the target. In the **Tool Settings** tab, select **All Tools Settings > Target**:
   a) From the **Target CPU** dropdown, select **Cortex-A53 AArch64**.
   b) From the **Target FPU** dropdown, select **Armv8 (Neon)**.
4. Configure the image layout. In the **Tool Settings** tab, select **Arm Linker 6 > Image Layout**:
   a) In the **RO base address** field, enter `0x80000000`.
5. Click **Apply and Close**.
6. If you are prompted to rebuild the index, click **Yes**.

9.1.4 Build your project

You can now build your `HelloWorld` project!

Before you begin
Complete these tasks:

- Create a project in C or C++
- Configure your project

Procedure
In the **Project Explorer** view, right-click the `HelloWorld` project and select **Build Project**.

Results
When the project has built, in the **Project Explorer** view, under **Debug**, locate the `HelloWorld.axf` file.

The `.axf` file contains the object code and debug symbols that enable Arm® Debugger to perform source-level debugging.

Debug symbols are added at build time. You can either specify this manually, using the `-g` option when compiling with Arm Compiler for Embedded 6, or you can set this to be default behavior. See Configure your project for details.

9.1.5 Configure your debug session

In Arm® Development Studio, you configure a debug session with the **New Debug Connection** wizard. This wizard enables you to connect to your target.

About this task
Depending on your requirements, you can:

- Configure a connection to an FVP for bare-metal application debug
- From the command-line, configure a connection to an FVP for bare-metal application debug

- Configure a connection to an FVP for Linux application debug
- Configure a connection to an FVP for Linux kernel debug

The following example takes you through configuring a bare-metal **Model Connection** to a Cortex®-A53 Fixed Virtual Platform (FVP), using the project you created in the previous section of this tutorial.

Procedure

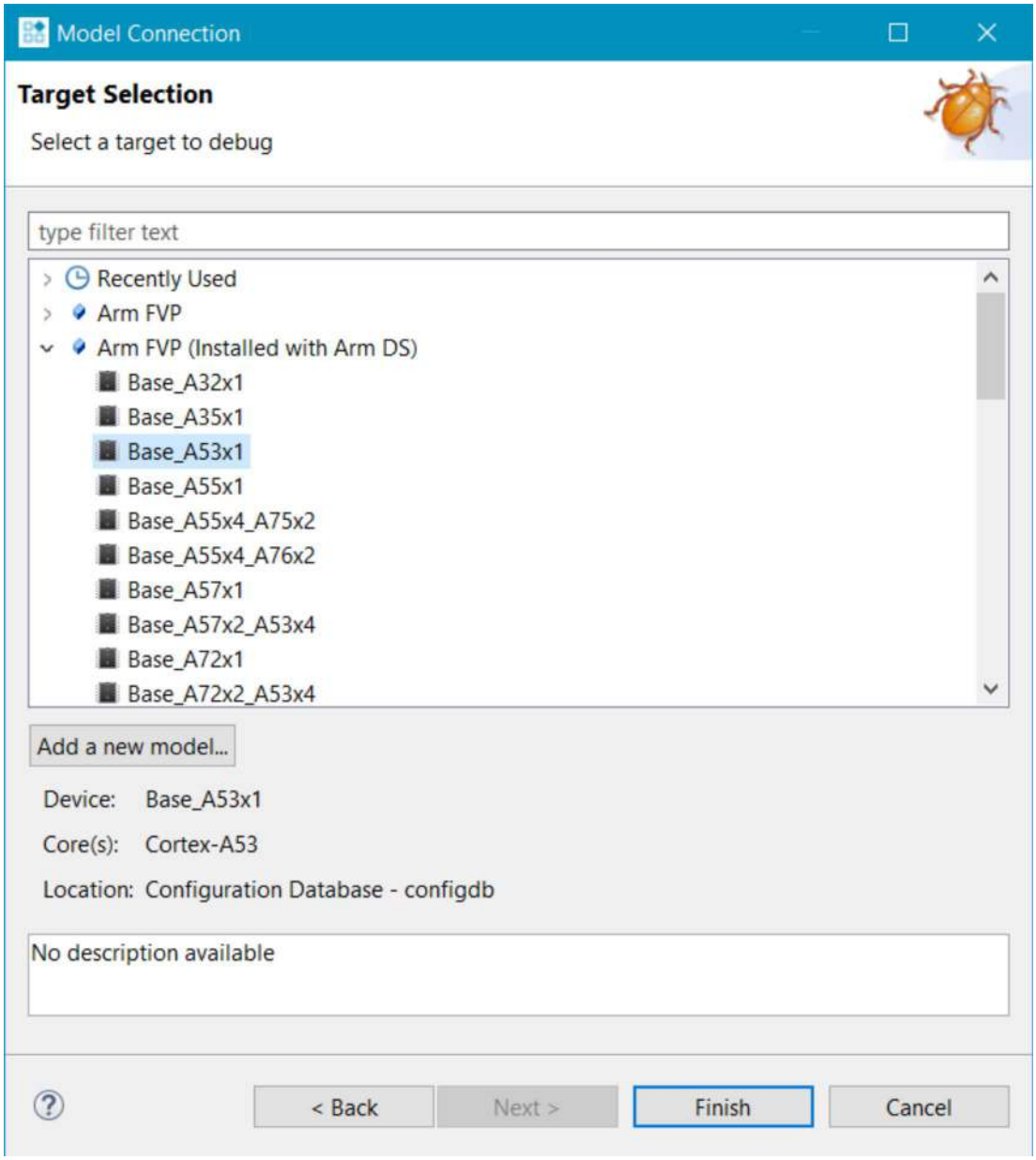
1. Create a `.ds` script so that the FVP handles semihosting, instead of Arm Debugger:
   a) From the main menu, select **File > New > Other...**.
   b) In the **New** dialog box, select **Arm Debugger > Arm Debugger Script** and click **Next**.
   c) Click **Workspace...** and select the **HelloWorld** project as the location for this script. Click **OK**.
   d) In the **File Name** field, name this script `use_model_semihosting` and click **Finish**. The empty script opens in the **Editor** window.
   e) Add the following code to the script and press **Ctrl + S** to save:

```
set semihosting enabled off
```

Figure 9-3: Editor window with semihosting script.

2. From the main menu, select **File > New > Model Connection**.
3. In the **Model Connection** dialog box, specify the details of the connection:
   a) Enter a name for the debug connection, for example **HelloWorld_FVP**.
   b) Select **Associate debug connection with an existing project**, and select the project that you created and built in the previous section Build your project.
   c) Click **Next**.
4. In the **Target Selection** dialog box, specify the details of the target:
   a) Select **Arm FVP (Installed with Arm DS) > Base_A53x1**.

Figure 9-4: Select Base_A53x1 model



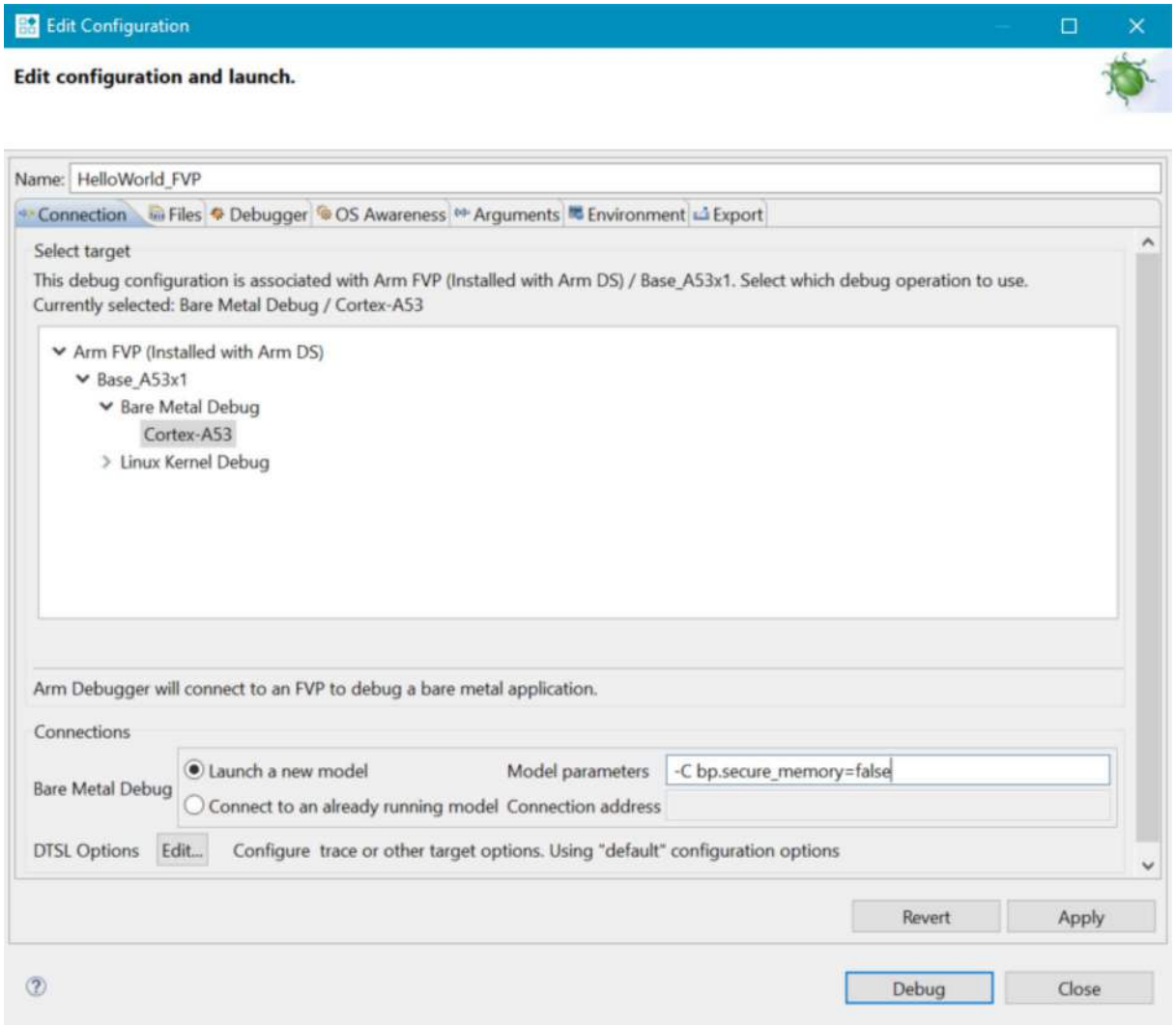
b) Click **Finish**.

5. In the **Edit Configuration** dialog box, ensure the right target is selected, the appropriate application files are specified, and the debugger knows where to start debugging from:
   a) Under the **Connection** tab, ensure that **Arm FVP (Installed with Arm DS) > Base_A53x1 > Bare Metal Debug > Cortex-A53** is selected.
   b) Under **Bare Metal Debug**, in the **Model parameters** field, add the following parameter:

```
-C bp.secure_memory=false
```

   This parameter disables the TZC-400 TrustZone memory controller included in the Base_A53x1 FVP. By default, the memory controller refuses all accesses to DRAM memory.

Figure 9-5: Edit configuration Connection tab



c) In the **Files** tab, select **Target Configuration > Application on host to download > Workspace**.
d) Click and expand the **HelloWorld** project and from the **Debug** folder, select `HelloWorld.axf` and click **OK**.

Figure 9-6: Select helloworld.axf file

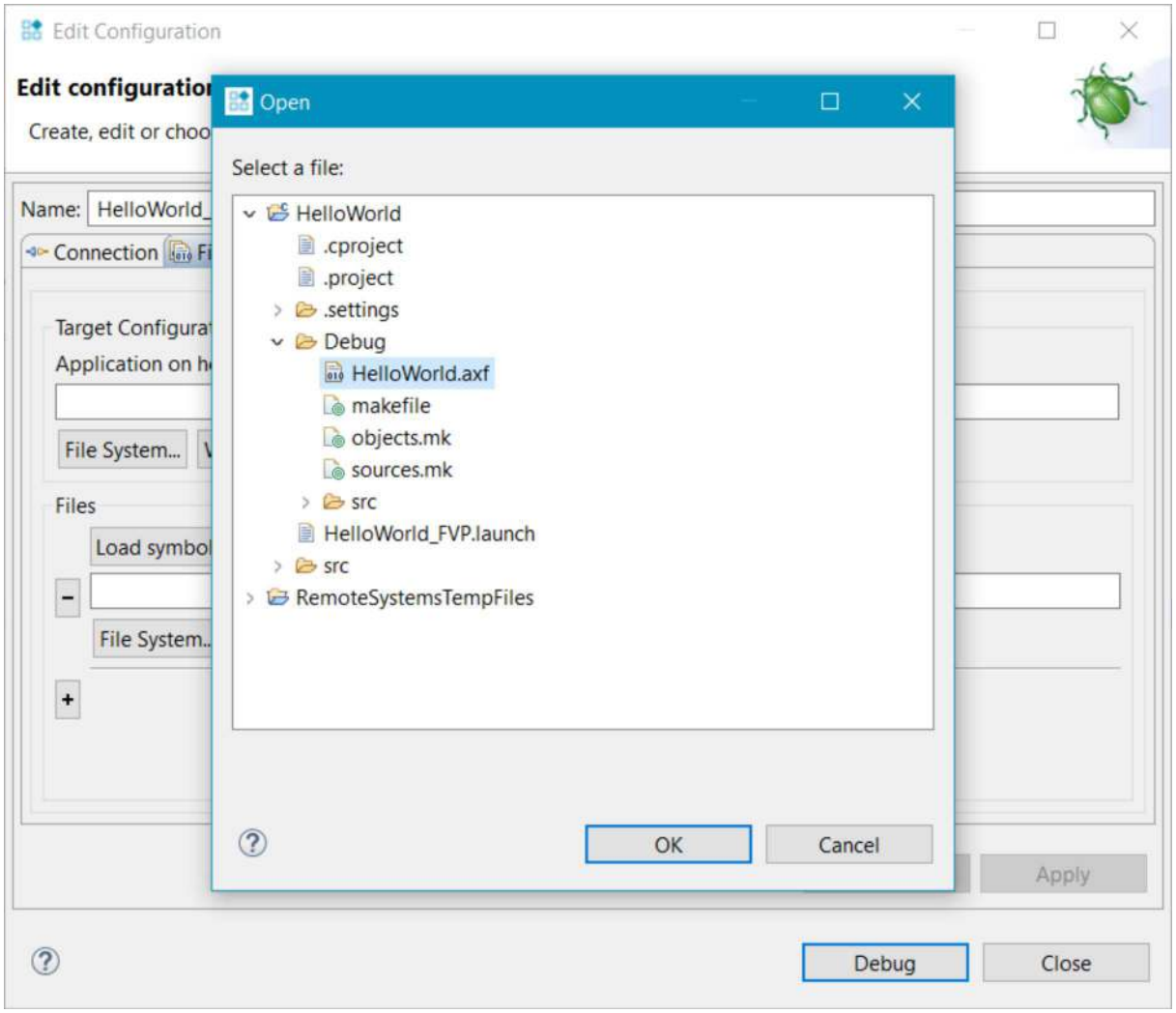
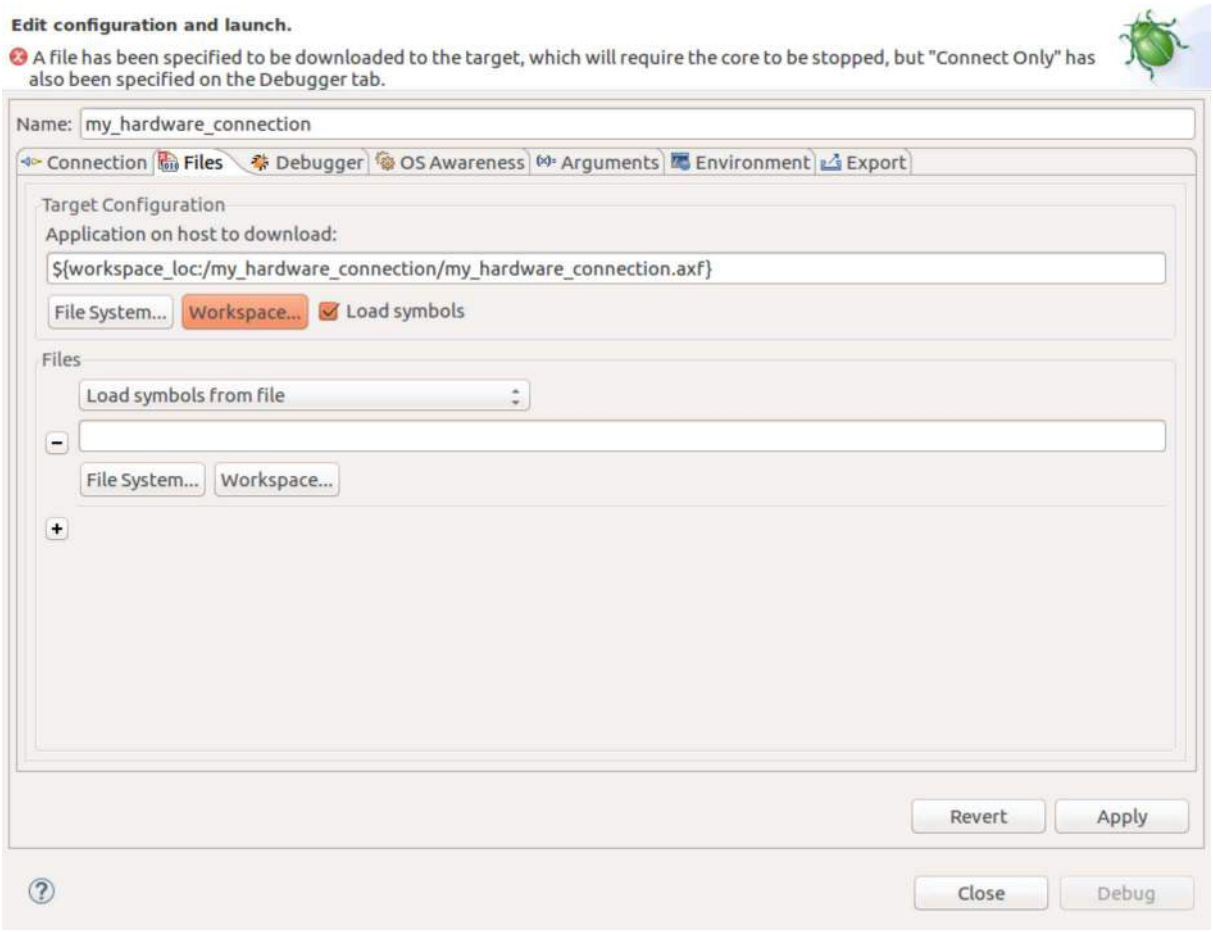
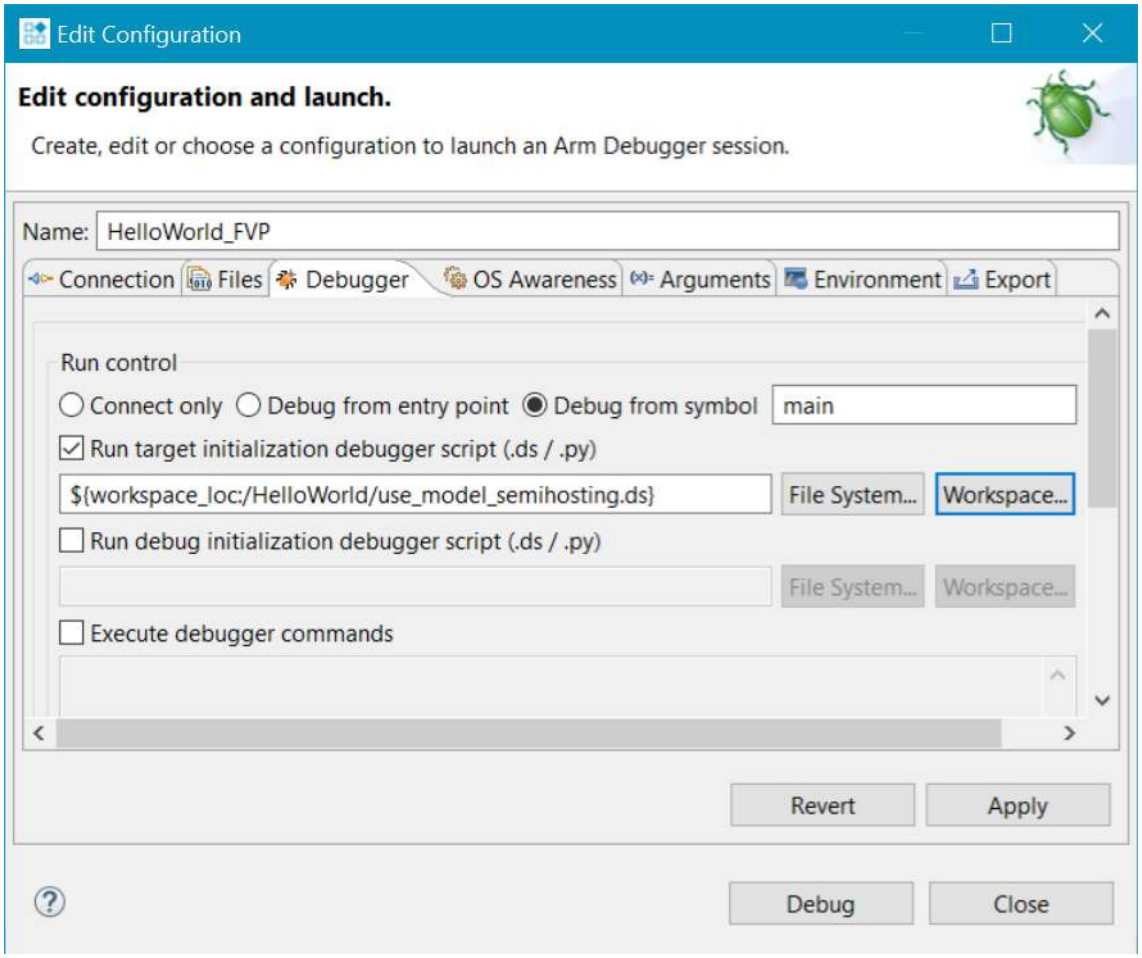


Figure 9-7: Edit configuration Files tab



e) In the **Debugger** tab, select **Debug from symbol**.
f) Enable **Run target initialization debugger script (.ds/.py)** and click **Workspace...**.
g) Select the `use_model_semihosting.ds` script and click **OK**.

Figure 9-8: Debug from symbol main

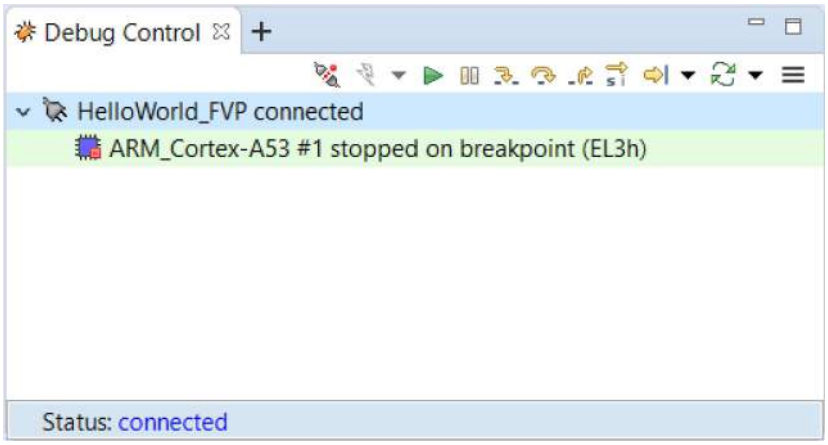


6. Click **Debug** to load the application on the target, and load the debug information into the debugger.

Results

Arm Development Studio connects to the model and displays the connection status in the **Debug Control** view.

Figure 9-9: Debug Control View



The application loads on the target, and stops at the `main()` function, ready to run.

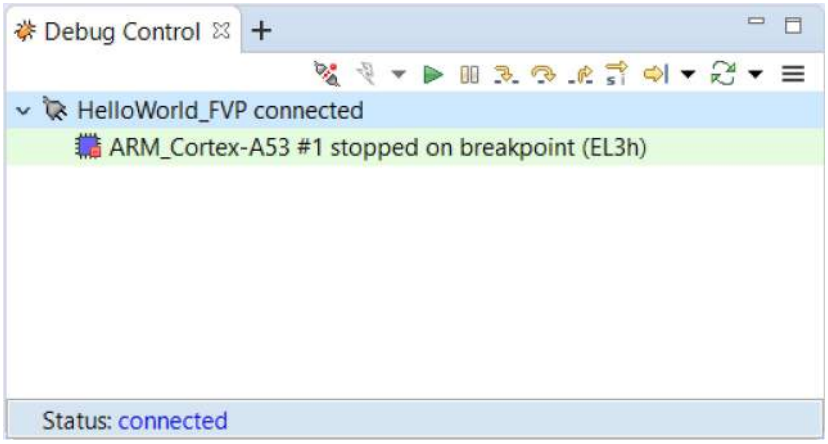
Figure 9-10: main () in code editor

9.1.6 Application debug with Arm Debugger

Now that you have created a debug configuration and the application is loaded on the target, it is time to start debugging and stepping through your application.

Running and stepping through the application

Use the controls provided in the **Debug Control** view to debug your application. By default, these controls do source level stepping.



 - Click to continue running the application after loading it on the target.

 - Click to interrupt or pause executing code.

 - Click to step through the code.

 - Click to step over a source line.

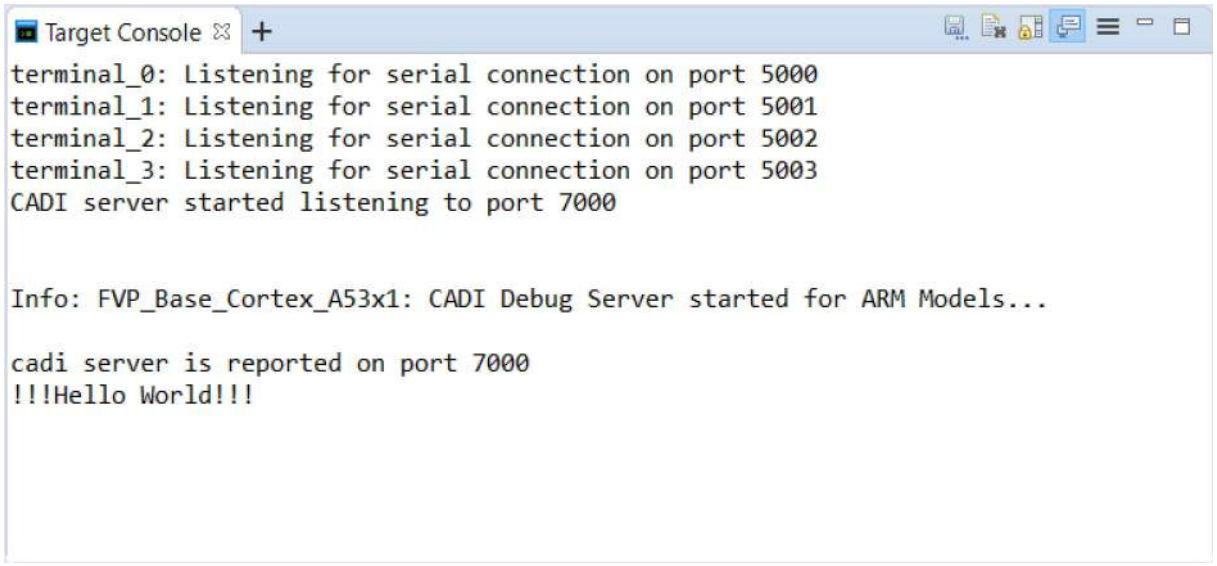
 - Click to step out.

 - This is a toggle. Click this to toggle between stepping instructions and stepping source code. This applies to the above step controls.

Other views display information relevant to the debug connection

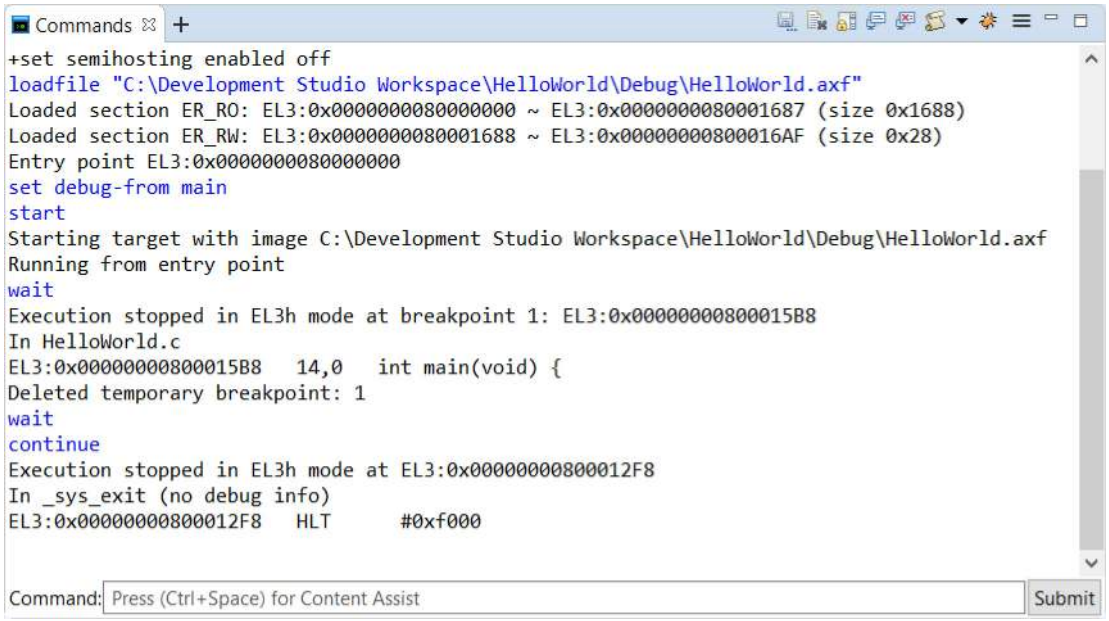
- **Target Console** view displays the application output.

Figure 9-11: Target console output



- **Commands** view displays messages output by the debugger. Also use this view to enter Arm® Debugger commands.

Figure 9-12: Commands view

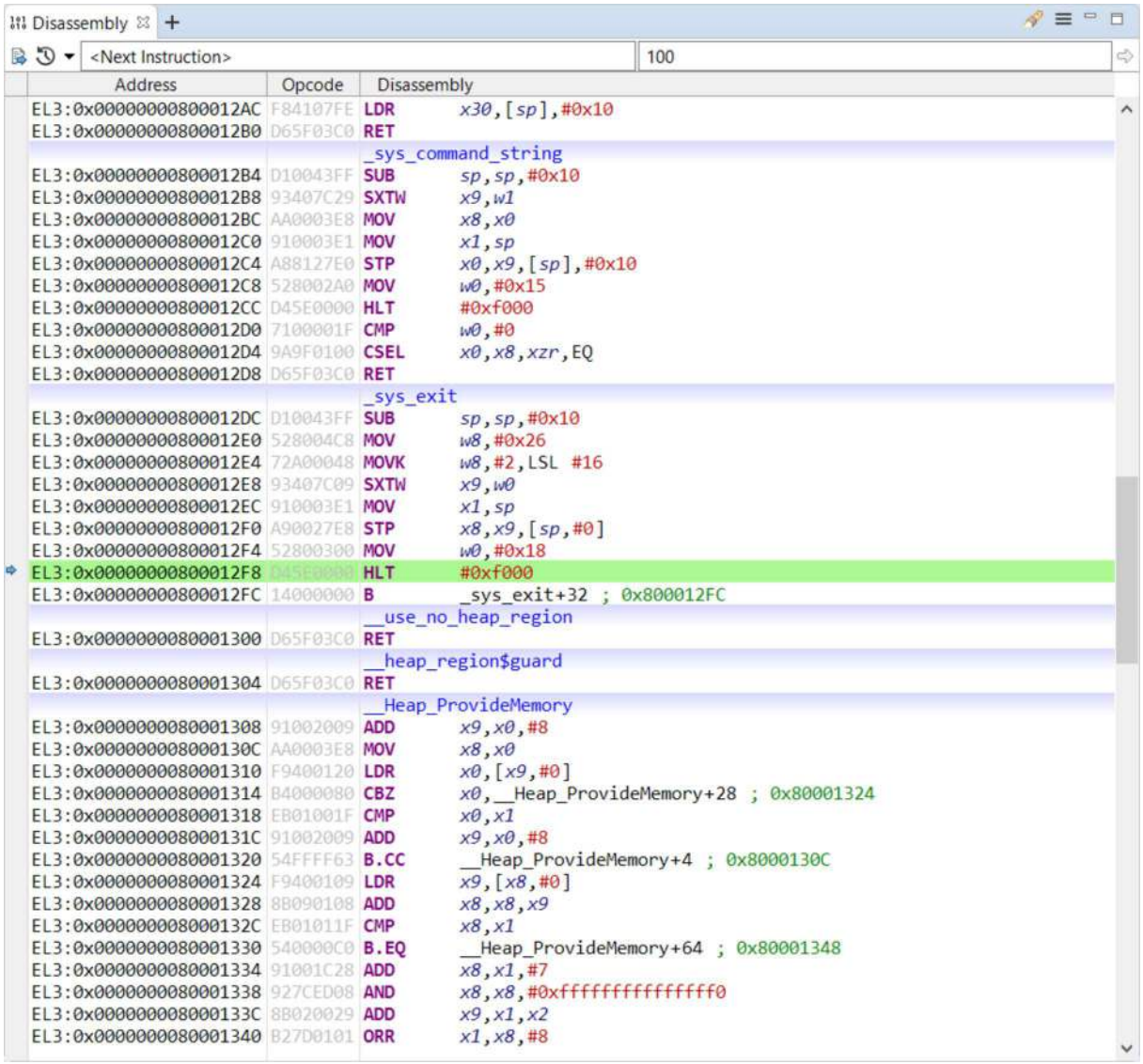


- C/C++ Editor view shows the active C, C++, or Makefile. The view updates when you edit these files.

Figure 9-13: Code Editor view

- **Disassembly** view shows the built program as assembly instructions, and their memory location.

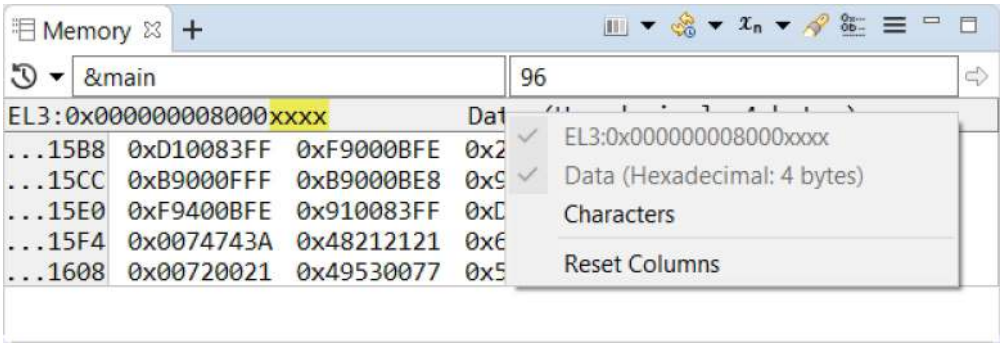
Figure 9-14: Disassembly view



indicates the location in the code where your program is stopped. In this case, it is at the `main()` function.

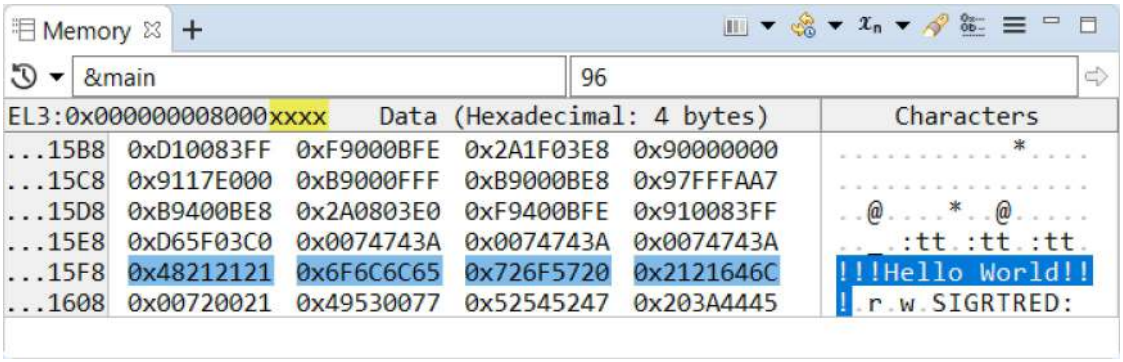
- **Memory** view shows how the code is represented in the target memory. For example, to view how the string `Hello World` from the application is represented in memory:
  1. Open the **Memory** view.
  2. In the **Address** field, enter `&main` and press **Enter** on your keyboard. The view displays the contents of the target's memory.
  3. Change the displayed number of bytes to `96` and press **Enter**.
  4. Right-click on the column headings, and select **Characters**.

Figure 9-15: Adding Characters column to Memory view



5. Select and highlight the words *Hello World*.

Figure 9-16: Memory view



In the above example, the **Memory** view displays the hexadecimal values for the code and the ASCII character encoding of the memory values, which enable you to view the details of the code.

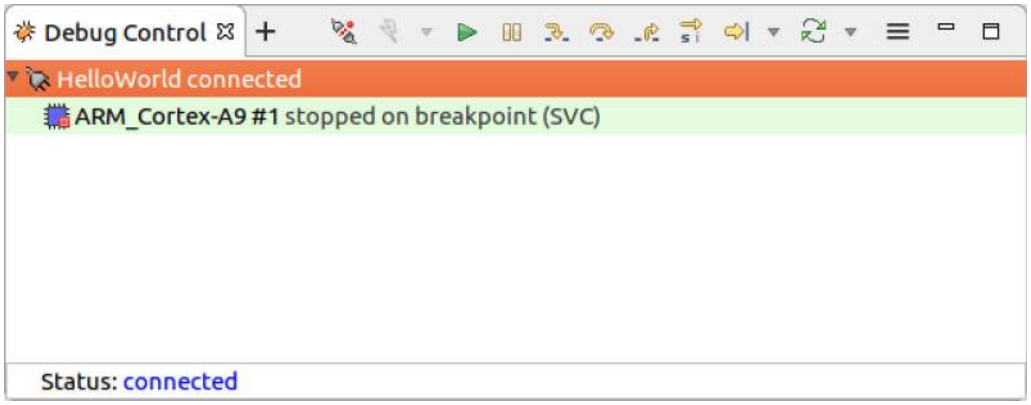
After completing your debug activities, you can disconnect the target.

9.1.7 Disconnecting from a target

To disconnect from a target, you can use either the **Debug Control** or the **Commands** view.

- If you are using the **Debug Control** view, click **Disconnect from Target** on the toolbar.

Figure 9-17: Disconnecting from a target using the Debug Control view



- If you are using the **Commands** view, enter **quit** in the **Command** field, then click **Submit**.

Figure 9-18: Disconnecting from a target using the Commands view

The disconnection process ensures that the state of the target does not change, except for the following case:

- Any downloads to the target are canceled and stopped.
- Any breakpoints are cleared on the target, but are maintained in Arm® Development Studio.
- The DAP (Debug Access Port) is powered down.
- Debug bits in the DSC (Debug Status Control) register are cleared.

If a trace capture session is in progress, trace data continues to be captured even after Arm Development Studio has disconnected from the target.

9.2 Tutorial: Using FVPs

The tutorial for using Fixed Virtual Platforms (FVPs) takes new users through basic scenarios of using FVPs with Arm® Development Studio.

9.2.1 Overview: FVPs

A Fixed Virtual Platform (FVP) is a simulated model of a development platform, including processor, memory, and peripherals.

You can use FVPs for bare-metal debugging and application development instead of a physical target. You can also capture trace output from an FVP. Some FVPs are provided with Arm® Development Studio. If required, you can manually add other FVPs to Arm Development Studio.

This tutorial builds on the Hello world tutorial and guides you through some of these usage scenarios.

Related information

Fast Models Fixed Virtual Platform Reference Guide

9.2.2 Launch and connect to an FVP in Arm Development Studio

You can launch and connect to Arm Fixed Virtual Platforms (FVPs) in Arm® Development Studio. This tutorial shows you how to launch an FVP, that is included with Arm Development Studio, using the Development Studio IDE.

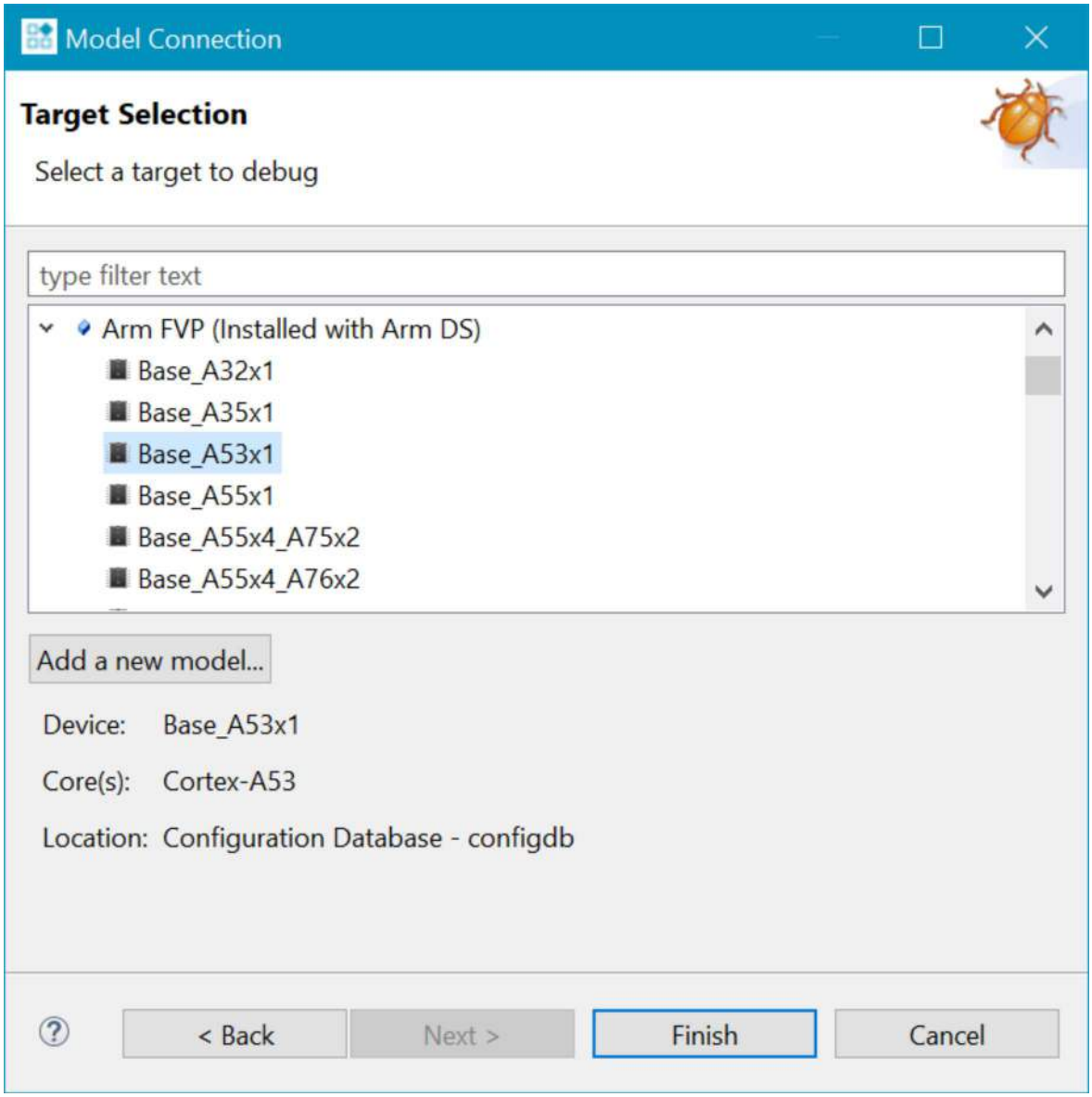
Procedure

1. Create a new model connection.
   a) Open the **Debug Connection** dialog box. From the main menu, select **File > New > Model Connection**. You can also select **Create a debug connection** in the **Debug Control** view to create a new connection.
   b) In the **Debug Connection** dialog box, specify the details of the connection. Enter a name for the debug connection and click **Next**.

If you have an existing project, select **Associate debug connection with an existing project**, and select the project that you want to debug.

   c) In the **Target Selection** dialog box, specify the details of the target. Under **Arm FVP (Installed with Arm DS)**, select the FVP you want to connect to and click **Finish**. For example, if you want to connect to a single-core Cortex®-A53 Base Platform FVP, select **Arm FVP (Installed with Arm DS) > Base_A53x1**.

Figure 9-19: Specify the details of the target.



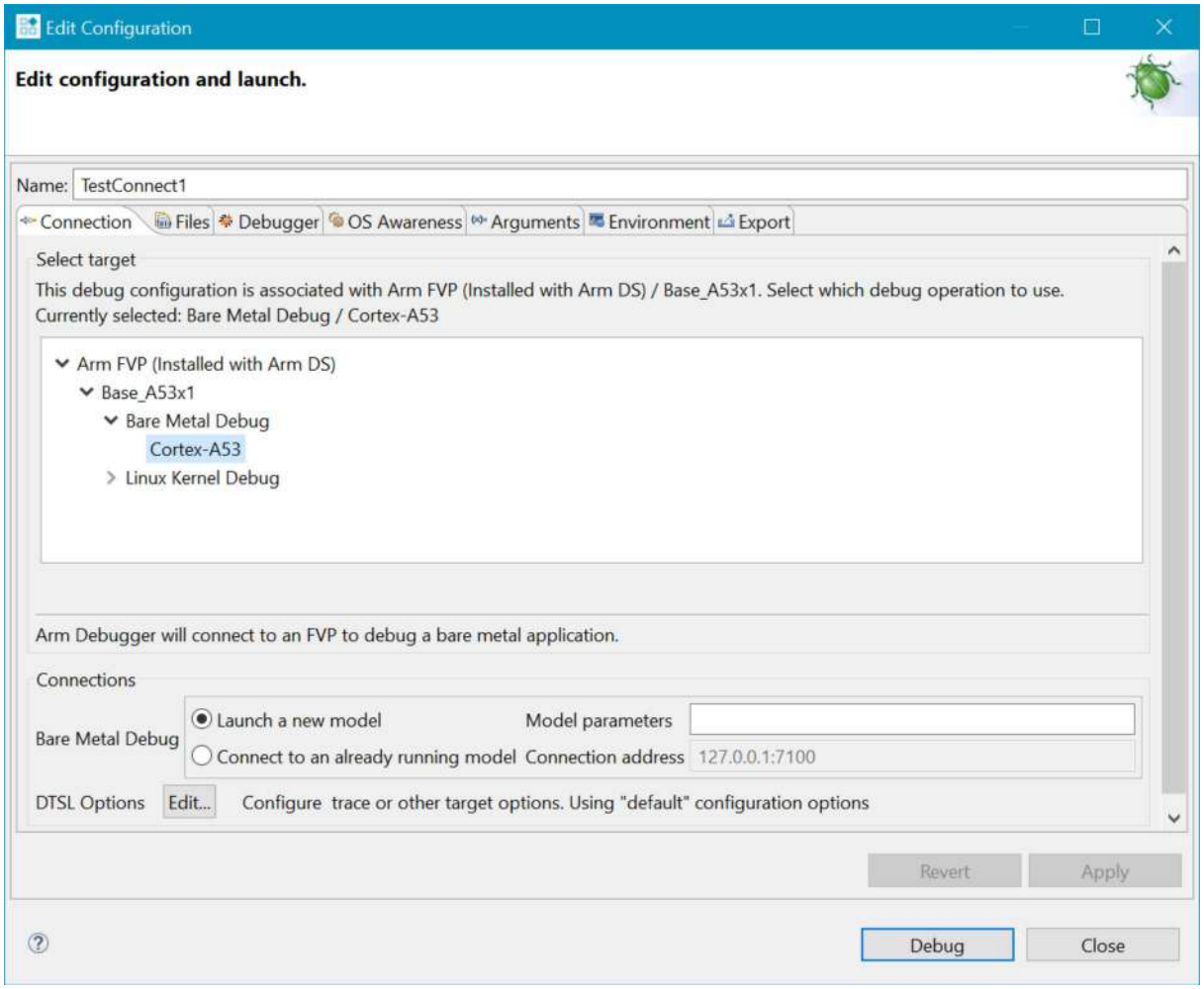
2. Edit your model configuration.
   a) In the **Edit Configuration** dialog box, under the **Connection** tab, ensure that you select the correct target for your debug operation. For example, to select a single-core Cortex-A53 for bare-metal debug, select **Arm FVP (Installed with Arm DS) > Base_A53x1 > Bare Metal Debug > Cortex-A53**.
   b) Under **Bare Metal Debug**, in the **Model parameters** field, specify the parameter for the connection as `-C bp.secure_memory=false`.

Note

For Cortex-M models, the parameter to add is `-C fvp_mps2.DISABLE_GATING=1`.

c) Click **Debug** to launch the connection to the FVP.

Figure 9-20: Edit the configuration and launch.



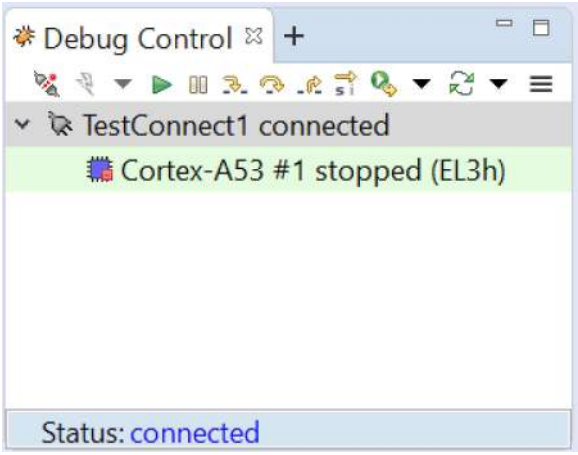
Results

- By default, the **CLCD window** launches. You can disable this default action with the parameter `-C bp.vis.disable_visualisation=1`. See Using the CLCD window for more information.

Figure 9-21: The CLCD window.

- The **Debug Control** view displays the status of the connection.

Figure 9-22: View the status of your connection.



Related information

Create a new model configuration

9.2.3 Configure a connection to an FVP for debug

In Arm® Development Studio, you can create and change configurations for a debugging session on a Fixed Virtual Platform (FVP) connection using the **Debug Configurations** dialog box.

Procedure

To open the **Debug Configurations** dialog box, right-click the FVP you want to configure in the **Debug Control** view, then click **Debug Configurations...**.

Results

In this dialog box, the tabs contain further options for your connection. For more information on the functionality of these tabs, see the following topics in the *Perspectives and Views* chapter of the *Arm Development Studio User Guide*.

- Connection tab
- Files tab
- Debugger tab
- OS Awareness tab
- Arguments tab
- Environment tab
- Export tab

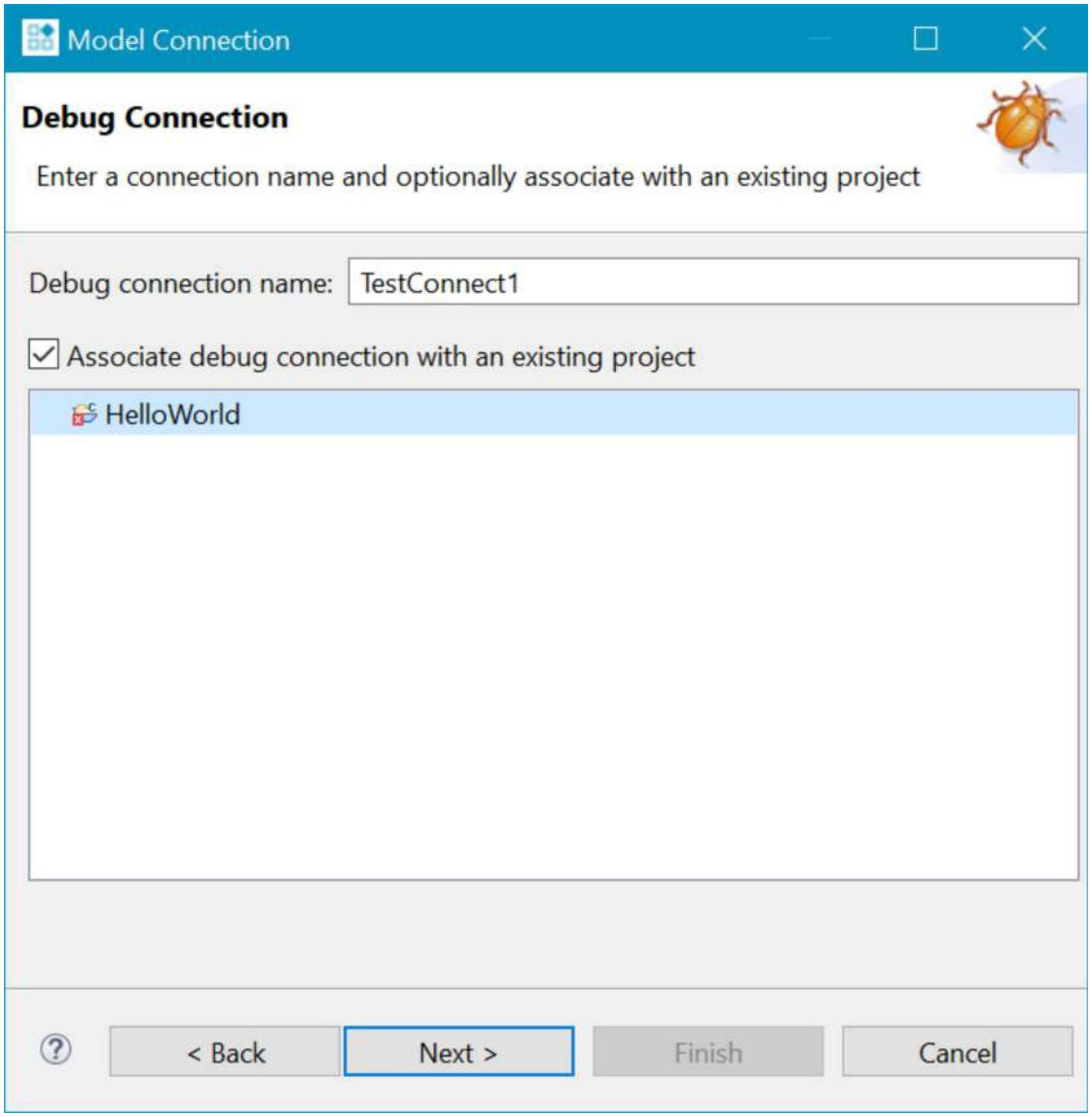
9.2.4 Run applications on an FVP

Arm FVPs (Fixed Virtual Platforms) can run applications in a simulation of real hardware.

Procedure

1. Link your project with the model configuration you are using and launch the model connection.
   - For new model connections:
     a. Open the **Debug Connection** dialog box. From the main menu, select **File > New > Model Connection**. You can also select **Create a debug connection** in the **Debug Control** view to create a new connection.
     b. In the **Debug Connection** dialog box, specify the details of the connection. Select **Associate debug connection with an existing project** and then select the project that you want to debug. Enter a name for the debug connection and click **Next**.

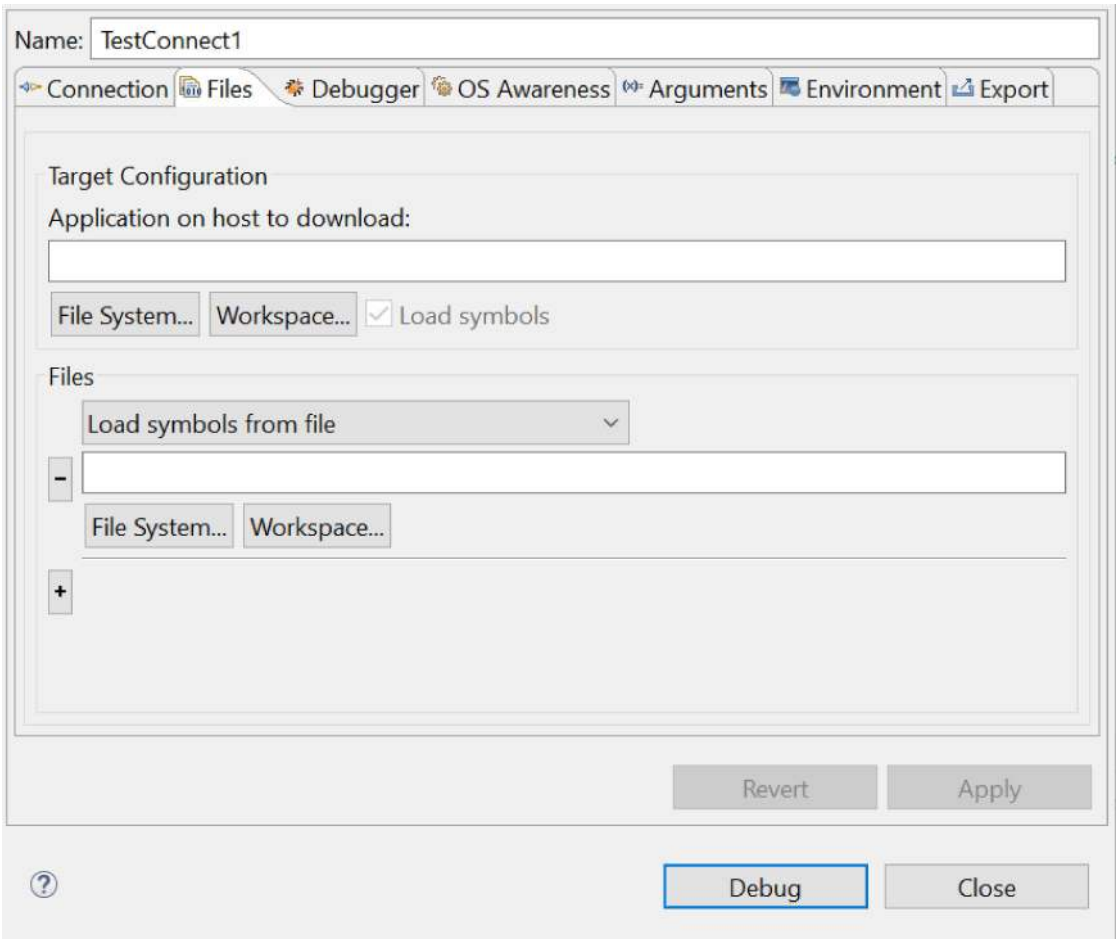
Figure 9-23: Associate the debug connection with your project.



c. In the **Target Selection** dialog box, specify the details of the target. Under **Arm FVP (Installed with Arm DS)**, select the FVP you want to connect to and click **Finish**.

- For existing model connections:
  a. Open the **Debug Configurations** window. Right-click your connection in the **Debug Control** view and select **Debug Configurations...**.
  b. In the **Files** tab, specify the location of the executable file for the project you want to debug. Click **Apply** then **Debug**.

Figure 9-24: Specify the location of your project's executable file.



2. To run the application, click  . Use the controls provided in the **Debug Control** view to debug your application. See Application debug with Arm Debugger for more information.
3. When you are finished, disconnect from the target.
   - If you are using the **Debug Control** view, click **Disconnect from Target** on the toolbar.
   - If you are using the Commands view, enter `quit` in the Command field, then click **Submit**.

9.2.5 Capture trace output from an FVP

Trace capture from a Fixed Virtual Platform (FVP) provides you with a detailed output of all the instructions that are executed in a debug session. You can enable trace capture in the **Debug and Trace Services Layer (DTSL) Configuration** dialog box.

Procedure

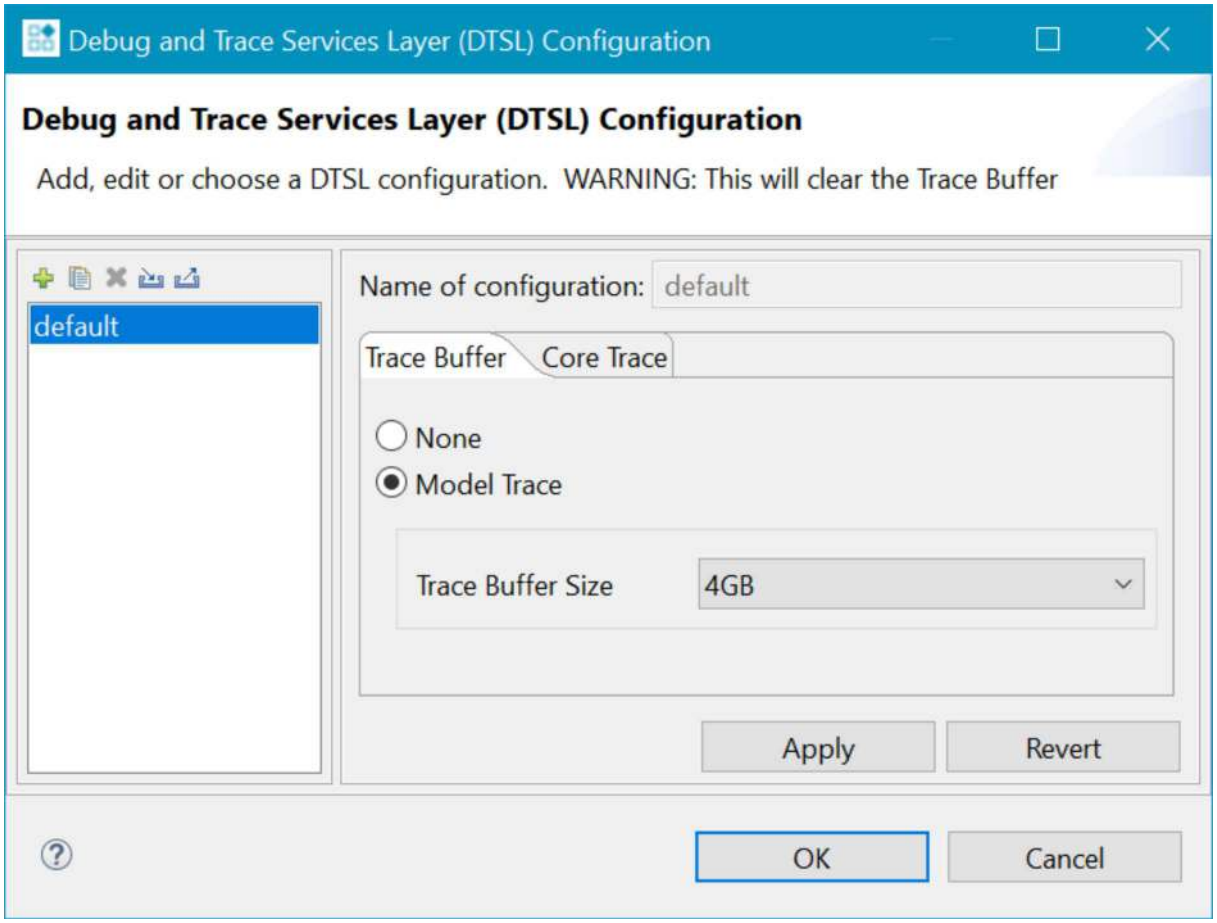
1. Open the **DTSL Configuration** dialog box. In the **Debug Control** view, right-click on the model configuration you want to enable trace capture on and select **DTSL Options**.

2. In the **Debug and Trace Services Layer (DTSL) Configuration** dialog box, select the **Model Trace** option under the **Trace Buffer** tab.


Note

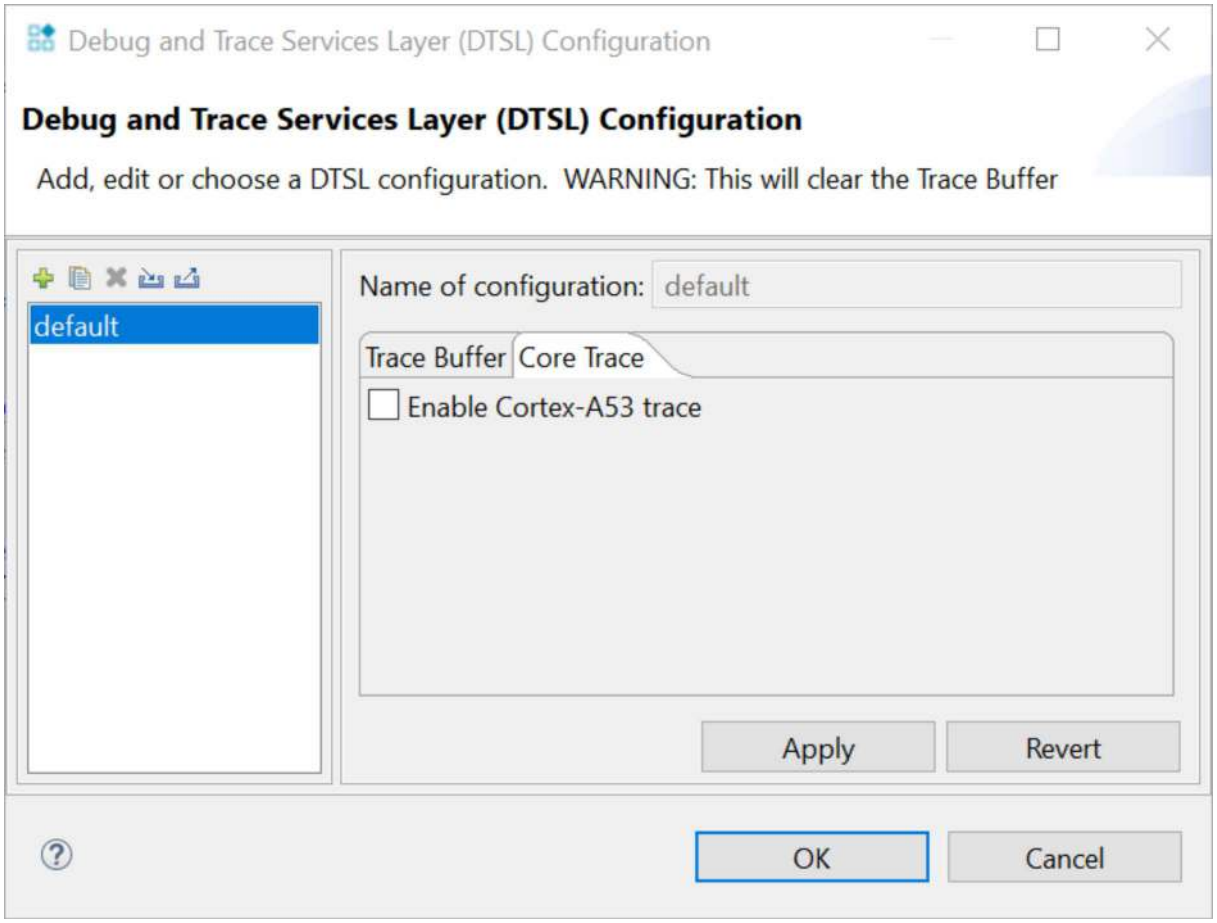
Here you can also change the trace buffer size in the **Trace Buffer Size** drop-down menu.

Figure 9-25: Trace Buffer tab.



3. In the **Core Trace** tab, select the processor on which you want to enable trace capture.

Figure 9-26: Core Trace tab.

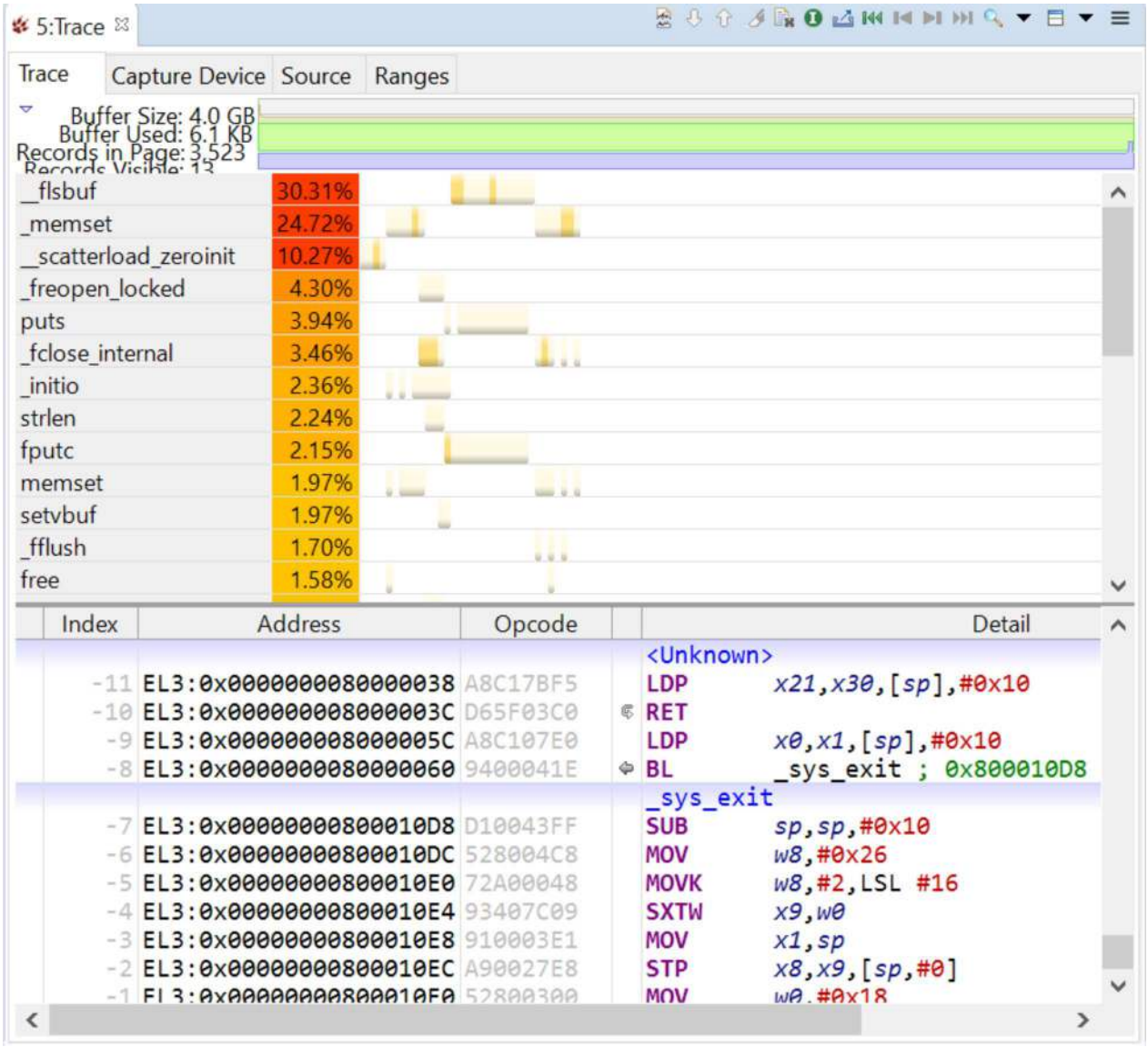


4. Apply your settings and close the dialog box, select **Apply** and then **OK**.

Next steps

In the **Trace** view, you can see all the instructions that are executed in a debug session.

Figure 9-27: Some example trace capture in the Trace view.



9.2.6 Add an external FVP to Arm Development Studio

Some Fixed Virtual Platforms (FVPs) are included with the installation of Arm® Development Studio. To use an Arm FVP that is not provided with Development Studio, you must first add it to the `PATH` environment variable of your OS.

Procedure

1. Add the `<install_directory>/bin` directory to your `PATH` environment variable and restart Arm Development Studio.
   - For Windows, enter `set PATH=<your model path>\bin;%PATH%`

- For Linux, enter `export PATH=<your model path>/bin:$PATH`

2. Ensure that the modified path is available for future sessions:
   - For Windows, right-click **This PC**, select **Properties > Advanced system settings**, then click **Environment Variables**. Then under **User Variables**, 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`.

10. Troubleshoot Arm Development Studio

Describes how to diagnose problems when debugging applications using Arm® Debugger.

10.1 Arm Linux problems and solutions

Lists possible problems when debugging a Linux application.

You might encounter the following problems when debugging a Linux application.

Arm Linux permission problem

If you receive a permission denied error message when starting an application on the target then you might have to change the execute permissions on the application:

```
chmod +x <myImage>
```

A breakpoint is not being hit

You must ensure that the application and shared libraries on your target are the same as those on your host. The code layout must be identical, but the application and shared libraries on your target do not require debug information.

Operating system support is not active

When Operating System (OS) support is required, the debugger activates it automatically where possible. If OS support is required but cannot be activated, the debugger produces an error. :

```
ERROR(CMD16-LKN36):
! Failed to load image "gator.ko"
! Unable to parse module because the operating system support is not active
```

OS support cannot be activated if:

- Debug information in the `vmlinux` file does not correctly match the data structures in the kernel running on the target.
- It is manually disabled by using the `set os enabled off` command.

To determine whether the kernel versions match:

- stop the target after loading the `vmlinux` image
- enter the `print init_nsproxy.uts_ns->name` command
- check that the `$1` output is correct:

```
$1 = {sysname = "Linux", nodename = "(none)", release = "3.4.0-rc3", version =
 "#1 SMP Thu Jan 24 00:46:06 GMT 2013", machine = "arm", domainname = "(none)"}
```

Related information

Configuring a connection to a Linux application using gdbserver
Configuring a connection to a Linux kernel

10.2 Enabling internal logging from the debugger

Describes how to enable internal logging to help diagnose error messages.

On rare occasions an internal error might occur, which causes the debugger to generate an error message suggesting that you report it to your local support representatives. You can help to improve the debugger, by giving feedback with an internal log that captures the stacktrace and shows where in the debugger the error occurs. To find out your current version of Arm® Development Studio, you can select **Help > About Arm Development Studio IDE** in the IDE, or open the product release notes.

To enable internal logging in the IDE, enter the following in the Commands view of the **Development Studio** perspective:

1. To enable the output of logging messages from the debugger using the predefined DEBUG level configuration: `log config debug`
2. To redirect all logging messages from the debugger to a file: `log file <debug.log>`

Enabling internal logging can produce very large files and slow down the debugger significantly. Only enable internal logging when there is a problem.

Related information

Commands view

10.3 FTDI probe: Incompatible driver error

When connecting your FTDI probe to Arm® Development Studio, you might see an error message when browsing for the probe.

The error is specific to Linux installations of Arm Development Studio:

```
Browsing failed: Incompatible virtual COM port driver (ftdi_sio) must be unloaded to
use FTDI MPSSE JTAG probe. See AN_220 FTDI Drivers Installation Guide for Linux.
```

Cause

The Linux operating system automatically loads an incompatible driver when the FTDI probe is plugged in.

Solution

1. To unload the incompatible driver, enter the following commands in your Terminal:

```
sudo rmmod ftdi_sio
sudo rmmod usbserial
```

2. Browse for your FTDI probe again, and it is now listed in the **Connection Browser**.

Related information

FTDI Drivers Installation Guide for Linux

10.4 Target connection problems and solutions

Lists possible problems when connecting to a target.

Failing to make a connection

The debugger might fail to connect to the selected debug target for the following reasons:

- You do not have a valid license to use the debug target.
- The debug target is not installed or the connection is disabled.
- The target hardware is in use by another user.
- The connection has been left open by software that exited incorrectly.
- The target has not been configured, or a configuration file cannot be located.
- The target hardware is not powered up ready for use.
- The target is on a scan chain that has been claimed for use by something else.
- The target hardware is not connected.
- You want to connect through gdbserver but the target is not running `gdbserver`.
- There is no ethernet connection from the host to the target.
- The port number in use by the host and the target are incorrect.

Check the target connections and power up state, then try and reconnect to the target.

Debugger connection settings

When debugging a bare-metal target the debugger might fail to connect for the following reasons:

- **Heap Base** address is incorrect.
- **Stack Base** (top of memory) address is incorrect.
- **Heap Limit** address is incorrect.
- Incorrect vector catch settings.

Check that the memory map settings are correct for the selected target. If set incorrectly, the application might crash because of stack corruption or because the application overwrites its own code.

Related information

Configuring a connection to a Linux application using gdbserver
Configuring a connection to a Linux kernel

11. Migrating from DS-5 to Arm Development Studio

Describes the differences between DS-5 Development Studio (DS-5) and Arm® Development Studio and provides information to aid migration from DS-5 to Arm Development Studio.

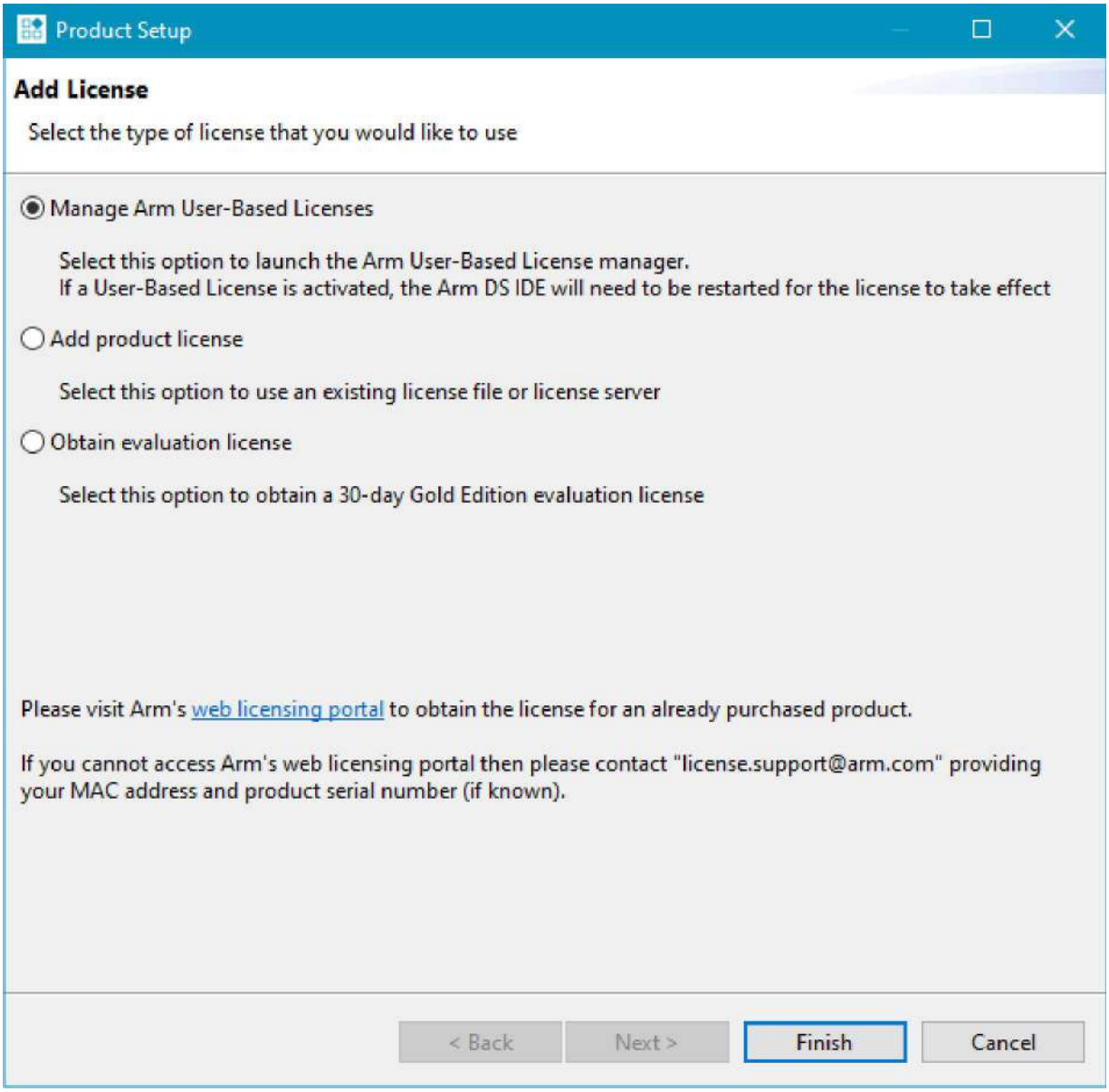
11.1 Add an Existing License Server

If Arm® Development Studio has no license information stored, you can add it when Arm Development Studio launches. This activity describes how to add an existing license server and specify the Arm Development Studio edition you want to use.

About this task

If no product license information exists for Arm Development Studio, the **Add License** dialog is shown when Arm Development Studio first opens:

Figure 11-1: Product Setup dialog box when you first open Arm Development Studio.



Note

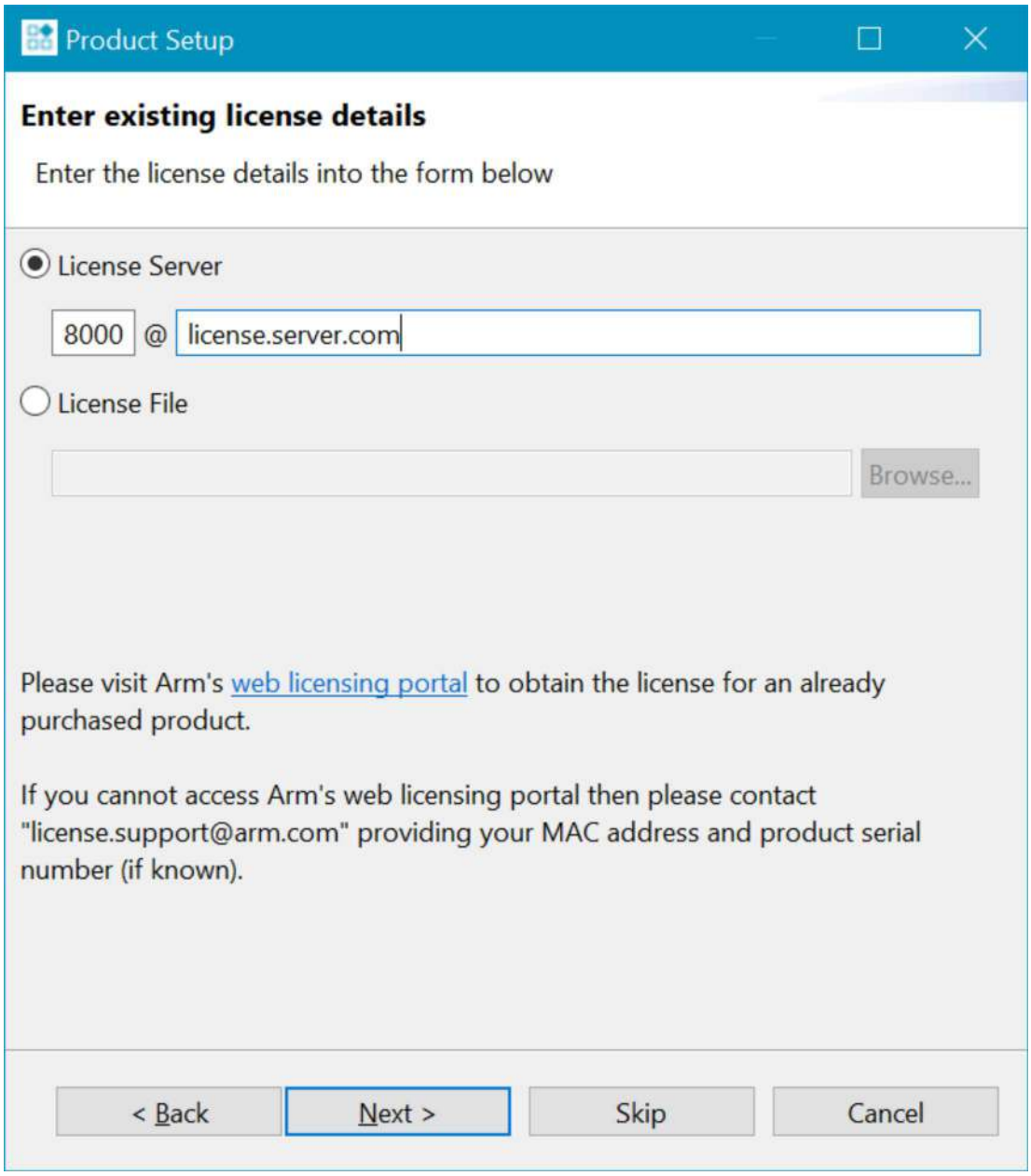
- To choose not to add a license to Arm Development Studio, you can click **Skip**. If you choose not to add a license, some functionality is disabled.
- You can add license information to Arm Development Studio at any time using the **Arm License Manager**. To open the **Arm License Manager**, select **Help > Arm License Manager**.

This activity assumes that you have not skipped adding a license to Arm Development Studio and that you are using an existing licence. If you are using user-based licensing, follow the instructions in Add a license using the Arm License Manager.

Procedure

1. Open Arm Development Studio IDE.
2. In the **Product Setup** dialog box, select **Add product license** and click **Next**.

Figure 11-2: Enter existing license details screen.

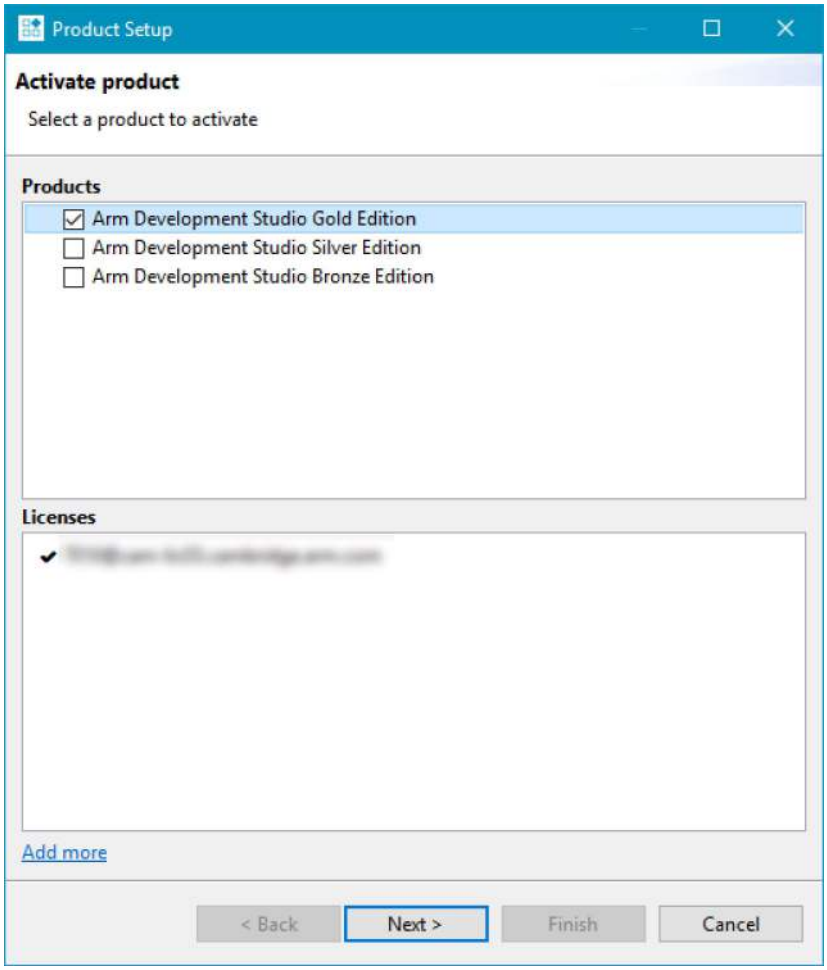


3. In the **Enter existing license details** screen, in the **License Server** field, enter the license server port and address and click **Next**.
4. In the **Activate Product** screen, select the appropriate Arm Development Studio edition from the provided list.

Note

Only editions enabled by your license file are listed.

Figure 11-3: Activate product screen.



5. To save and apply your changes, click **Next** and then **Finish**.

Next steps

You can change or add license information in Arm Development Studio using the **Arm License Manager**. You can access the **Arm License Manager** by selecting either **Help > Arm License Manager** or **Window > Preferences > Arm DS > Product Licenses**.

Related information

Add a license using Product Setup on page 27
Arm Compiler Licensing Configuration

11.2 Default Workspace Location

When launching for the first time, Arm® Development Studio uses a default workspace in your home directory. After the first launch, Arm Development Studio automatically opens the last used workspace.

The default workspace location is:

- Windows: *user directory*`/Development Studio Workspace`
- Linux: *home*`/Development Studio Workspace`

To change to a different workspace directory in Arm Development Studio, select **File > Switch > Workspace**.

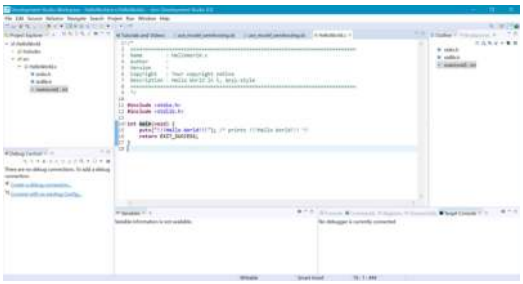
To change the default behavior and specify a workspace on startup, navigate to **Window > Preferences > General > Startup and Shutdown > Workspaces** and tick **Prompt for workspace on startup**.

11.3 Combined C/C++ and Debug Perspectives

Arm® Development Studio has a new IDE perspective, called Development Studio. The Development Studio perspective combines the DS-5 C/C++ and DS-5 debugger perspectives to display commonly used views in a single perspective.

This is the default perspective when Arm Development Studio opens:

Figure 11-4: Arm Development Studio IDE



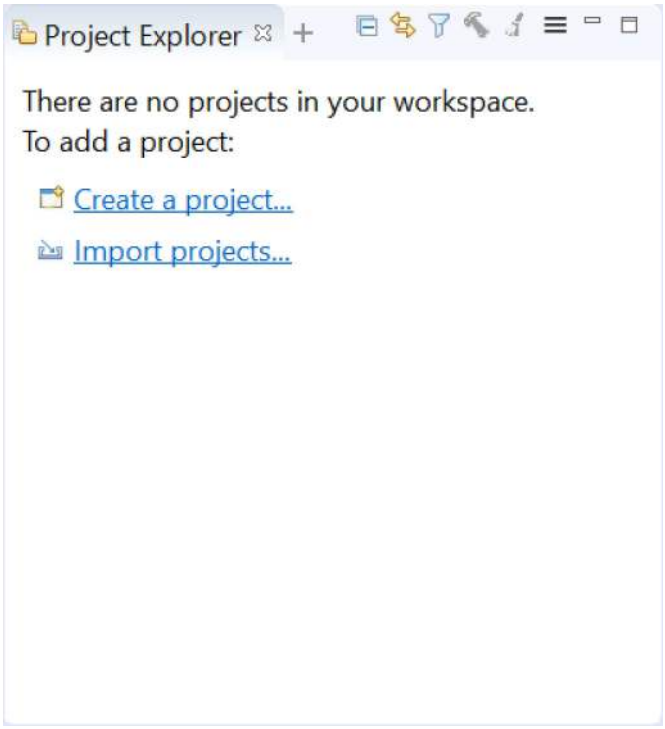
Project Explorer View

The **Project Explorer** view allows you to create and import projects.

Note

The **Import Project** option is only present if no projects are listed in the **Project Explorer** view.

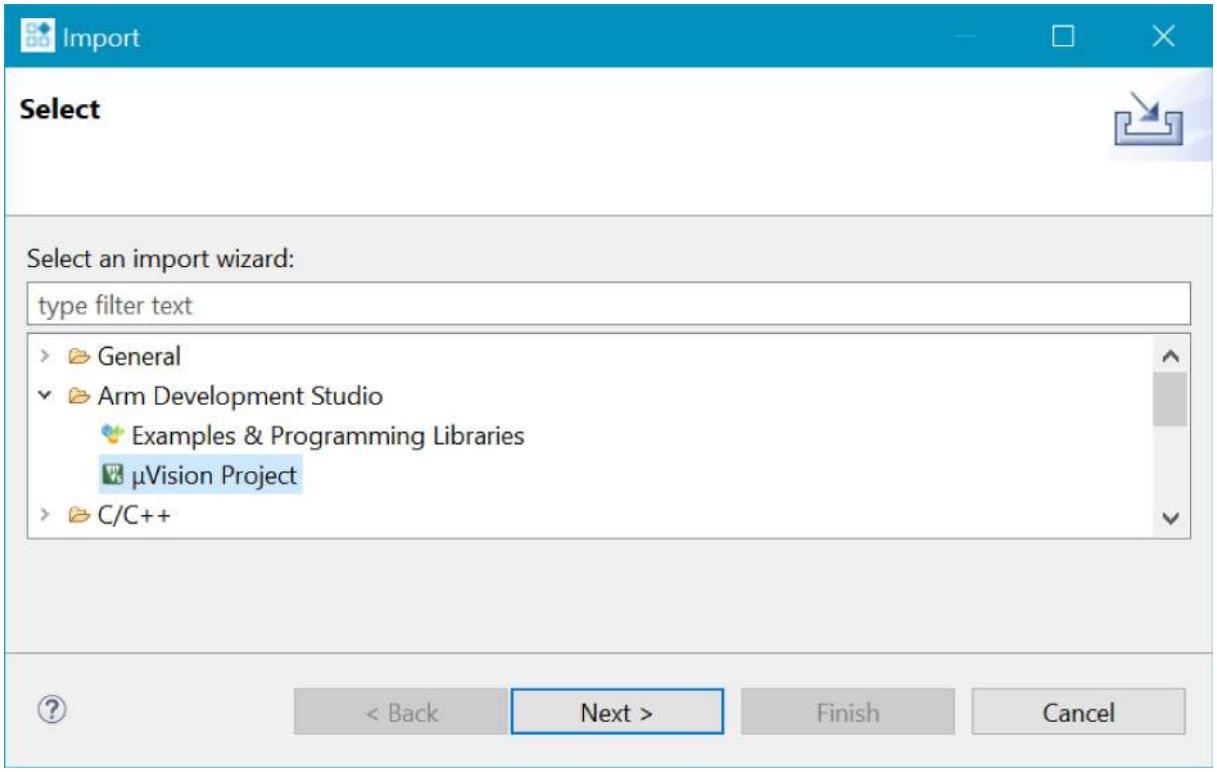
Figure 11-5: Project Explorer view in Arm Development Studio



In Arm Development Studio, you can now create and import existing µVision® projects:

1. From the toolbar, select **File > Import..** to open the **Import** dialog.

Figure 11-6: Import project dialog



2. Expand the Arm Development Studio drop-down, select **µVision Project** and click **Next**.

Debug Control View

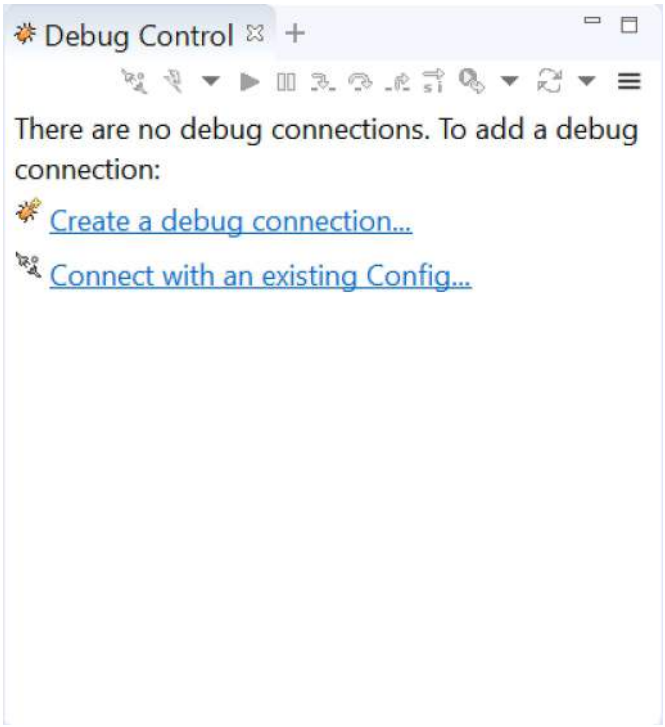
The **Debug Control** view allows you to create new debug connections and connect with existing configurations.

Note

The **Create a Debug Connection** option is only shown in the **Debug Control** view if no launch configurations exist in Arm Development Studio workspace. Arm Development Studio provides new methods to create hardware, Linux application and model connections. To read more about these new methods, see:

- Creating a new Hardware Connection
- Creating a new Linux Application Connection
- Creating a new Model Connection

Figure 11-7: Import project dialog



To connect to an existing configuration, click the **Connect with Existing Config** button.

Note

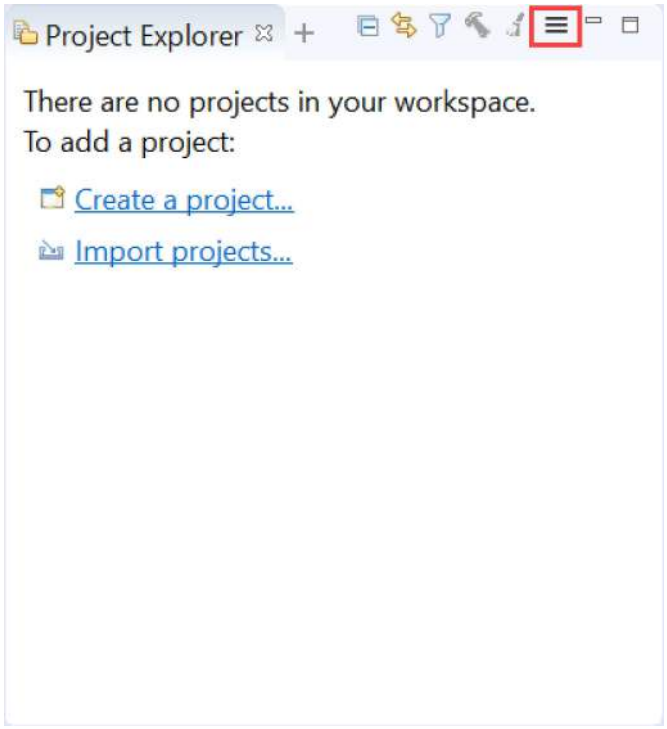
The **Connect with Existing Config** option is only shown if no launch configurations exist in the Arm Development Studio workspace.

General UI differences between DS-5 and Arm Development Studio

There are several minor UI features in Arm Development Studio that you must be aware of:

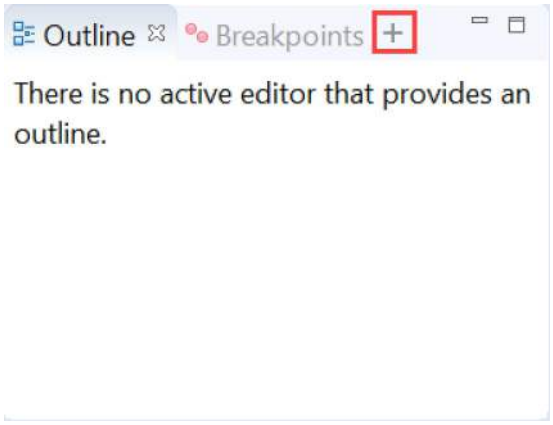
- In Arm Development Studio a three line button is used to display menu items instead of the inverted triangle used in DS-5.

Figure 11-8: Project Explorer view in Arm Development Studio



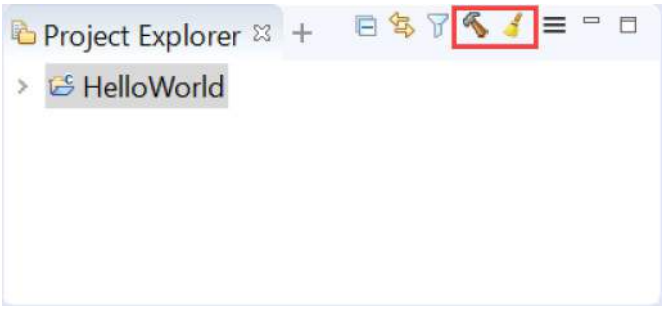
- To add more views to a Development Studio perspective in Arm Development Studio, you can either click **+**, or select **Window > Show View** and choose your view.

Figure 11-9: Add new view button in Arm Development Studio



- Use the **Builds the selected project** (hammer) and **Cleans the selected project** (broom) buttons in the **Project Explorer** view to build and clean the selected project.

Figure 11-10: Build and Clean project buttons in Project Explorer view



Related information

Debug Control view
Perspectives in Arm Development Studio

11.4 Migrate an existing DS-5 project

You can import DS-5 projects and launch configurations (.launch files) into Arm® Development Studio.

About this task

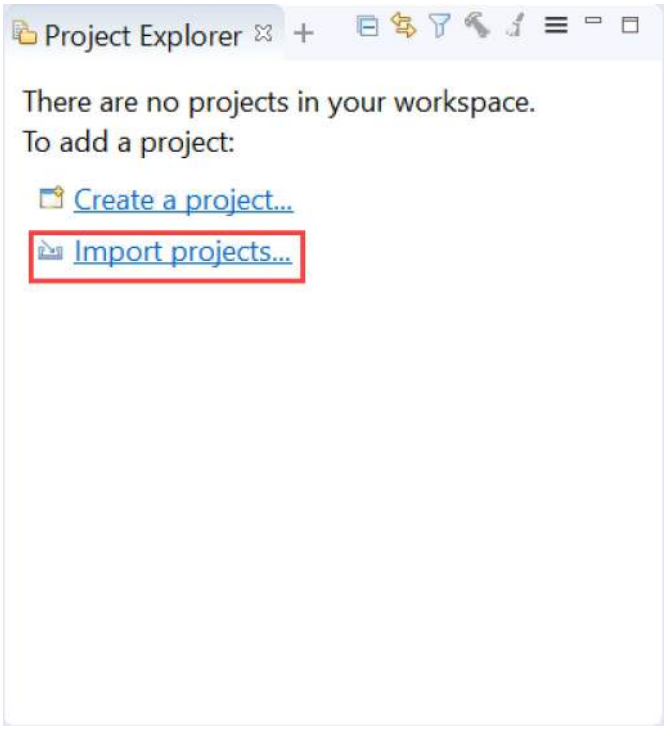
You can import existing DS-5 projects into Arm Development Studio, but projects and launch configurations imported into Arm Development Studio are not backward-compatible with DS-5.

Procedure

1. Choose one of these project import methods:
   - Click the **Import Project** option in the **Project Explorer** view.

The **Import Project** option only appears if no projects exist.

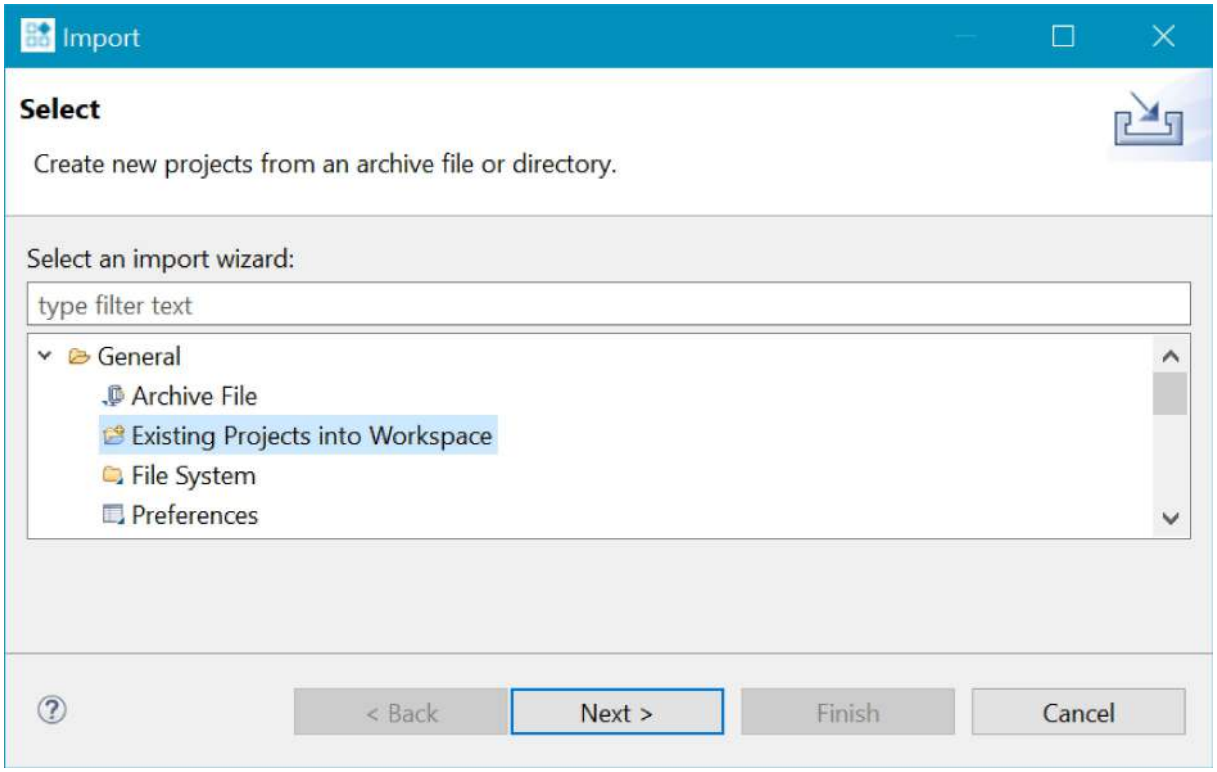
Figure 11-11: Import Project option in the Project Explorer view.



- Select **File > Import...**

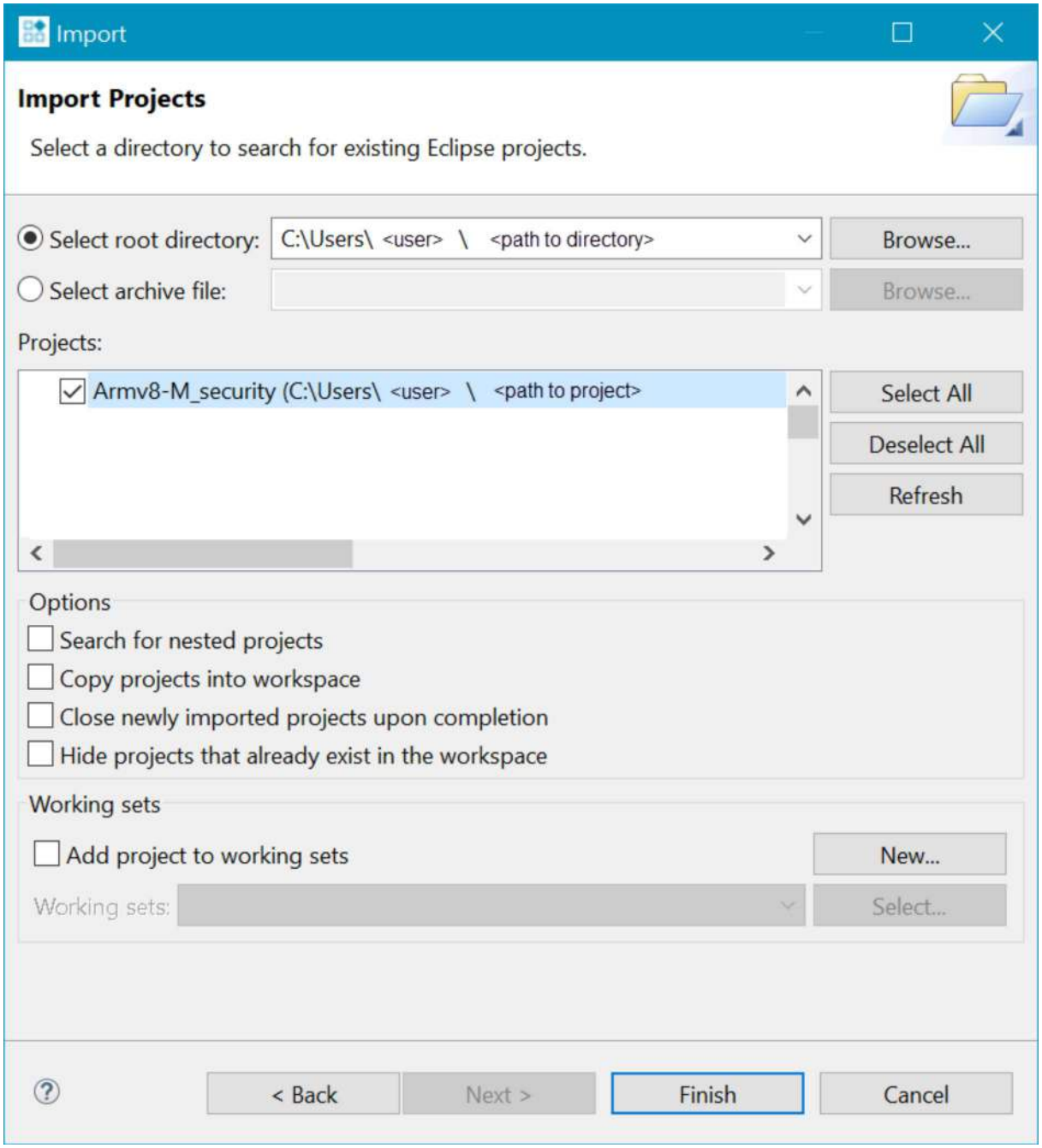
2. Select **General > Existing Projects into Workspace** and click **Next**.

Figure 11-12: Import dialog box



3. Select the existing DS-5 project(s) to import.
   a) Click **Browse...** to locate the projects.

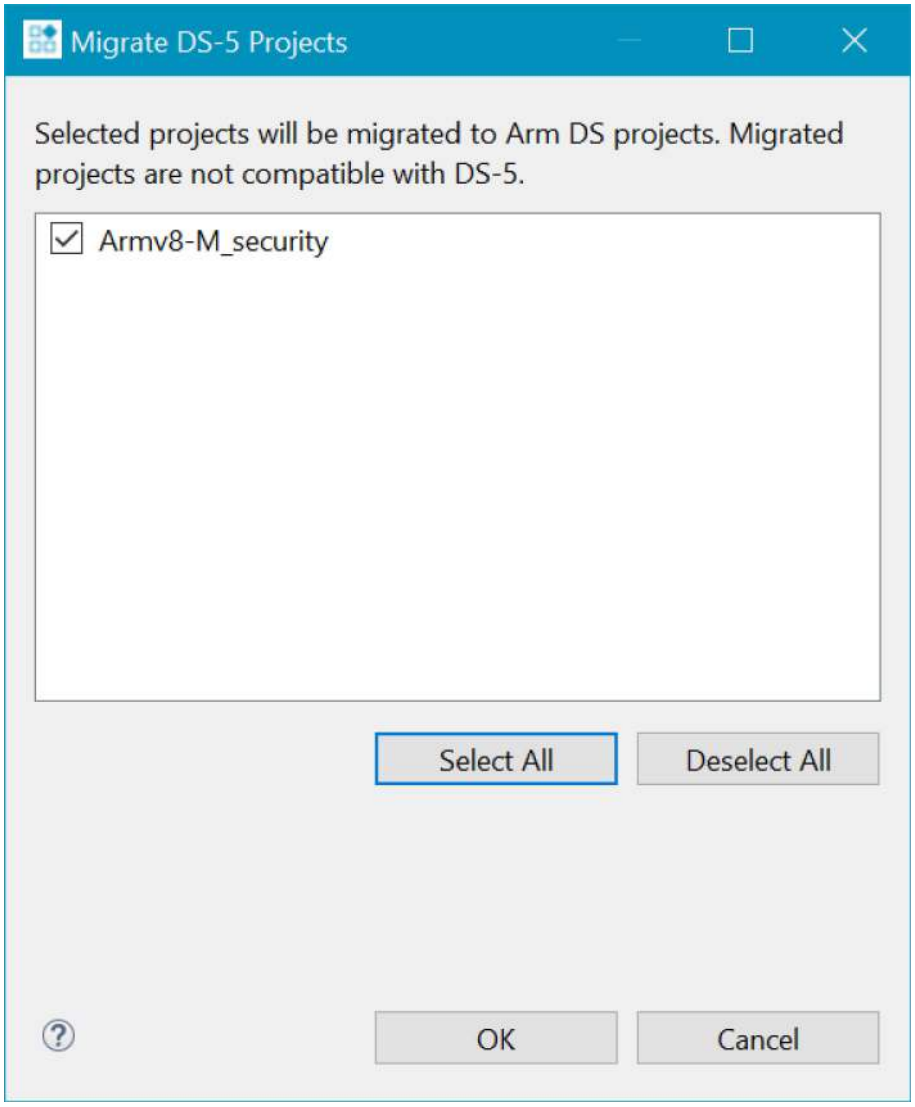
Figure 11-13: Import dialog browse for root directory



b) Ensure the projects to import are selected and click **Finish**.

4. Import the projects.
   a) Click **Select All** to import all the existing DS-5 projects or select the project(s) to import.

Figure 11-14: Import dialog browse for root directory



b) Click **OK**.

Results

The imported project(s) appear in the **Project Explorer** view.

Related information

Arm Compiler Licensing Configuration

11.5 CMSIS Packs

Arm® Development Studio includes support for Common Microcontroller Software Interface Standard (CMSIS) Packs. CMSIS packs offer you a quick and easy way to create, build and debug

embedded software applications for processors that are based on Arm® Cortex®-M class and Cortex-A5/A7/A9.

CMSIS Packs are a delivery mechanism for software components, device parameters, and board support. A CMSIS Pack is a file collection that might include:

- Source code, header files, software libraries - for example RTOS, DSP and generic middleware.
- Device parameters, such as the memory layout or debug settings, along with startup code and Flash programming algorithms.
- Board support, such as drivers, board parameters, and debug connections.
- Documentation and source code templates.
- Example projects that show you how to assemble components into complete working systems.

CMSIS Packs are developed by various silicon and software vendors, covering thousands of different boards and devices. You can also use them to enable life-cycle management of in-house software components.

You can use the **CMSIS Pack Manager** perspective in Arm Development Studio to load and manage CMSIS Packs. The **New Project** wizard allows you to easily create a new project based on selected CMSIS Pack(s).

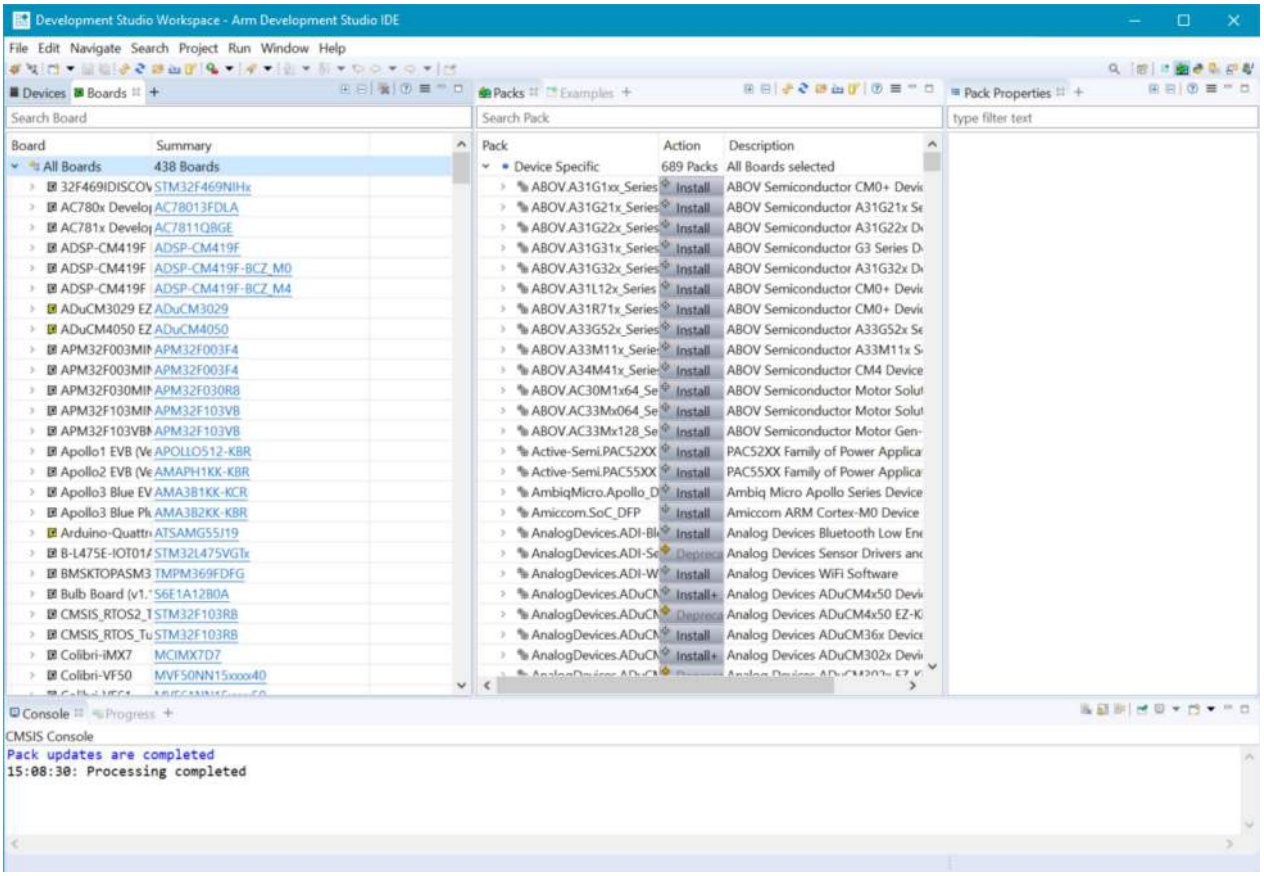
To create a new Pack-based project, install the Packs needed for your target board/device from the **CMSIS Pack Manager**, then use **File > New > Project** to create a new CMSIS C Project.


Note

If you have already installed some CMSIS Packs, you can redirect the CMSIS Pack Manager to the existing CMSIS Packs by setting **Window > Preferences > CMSIS Packs > CMSIS Pack root folder** to the location of the installation folder.

You can access the **CMSIS Pack Manager** perspective by navigating to **Window > Perspective > Open Perspective > CMSIS Pack Manager**.

Figure 11-15: CMSIS Pack Manager perspective

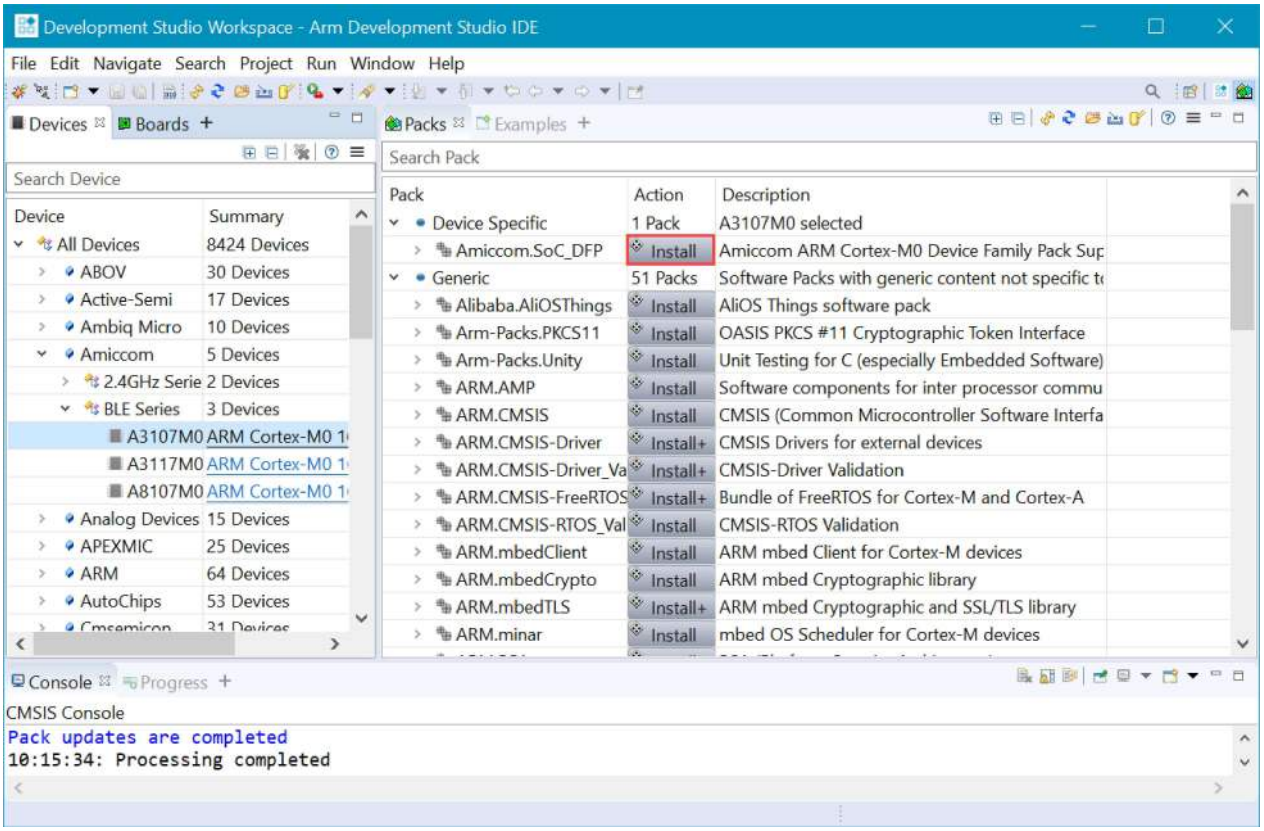


To install the CMSIS pack(s) you must select the device manufacturer and board in the **Devices** view, and, in the **Packs** view, click the appropriate **Install** icon next to the pack that you want to install.

Note

When you create a new project or hardware connection, boards or devices that have CMSIS packs installed are available as selectable targets. For more information on creating a hardware connection in Arm Development Studio, see Create a new Hardware Connection.

Figure 11-16: Installing a CMSIS pack for a device.

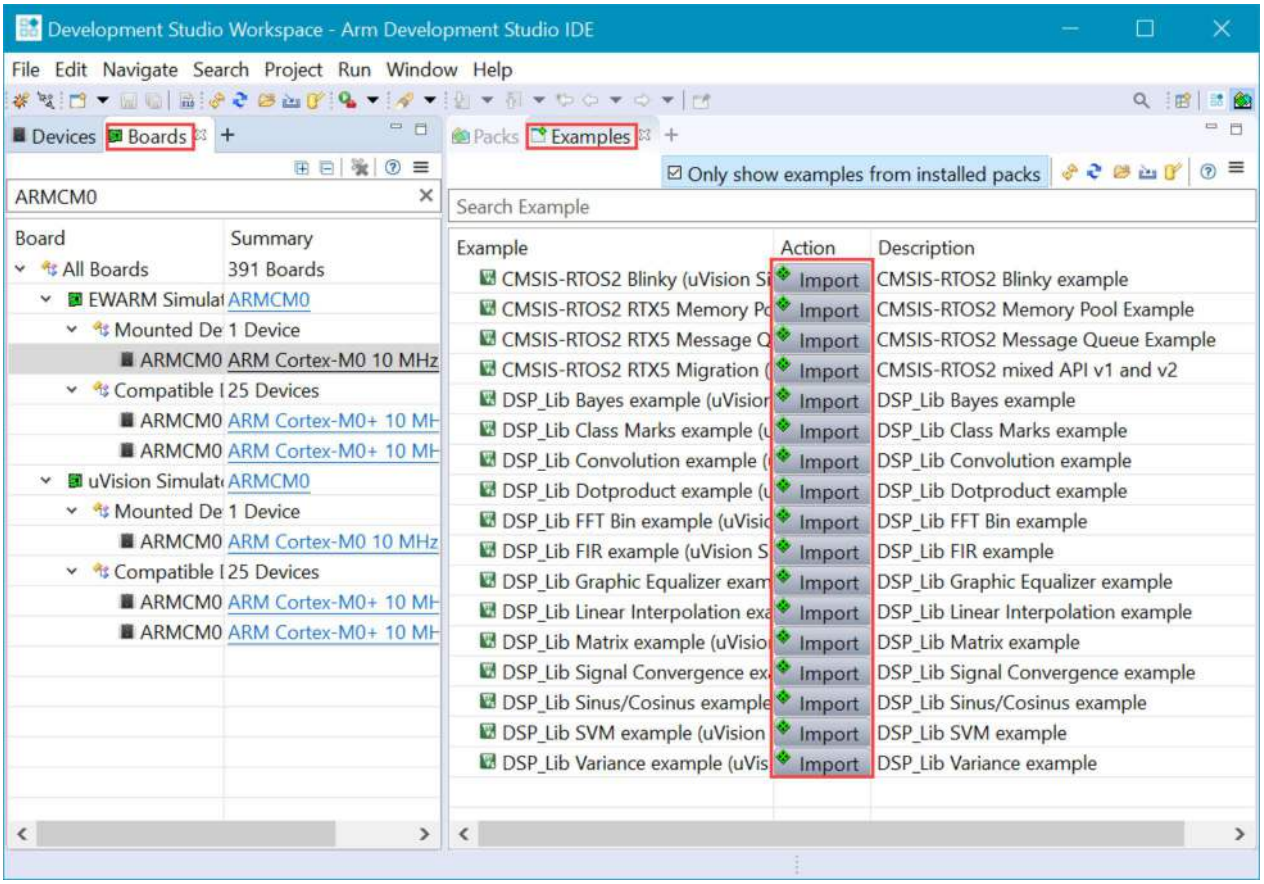


You can copy example CMSIS pack projects into the current workspace by opening the **Boards** view and selecting your target board. Then open the **Examples** view and click the **Import** icon next to your preferred example project.

Note

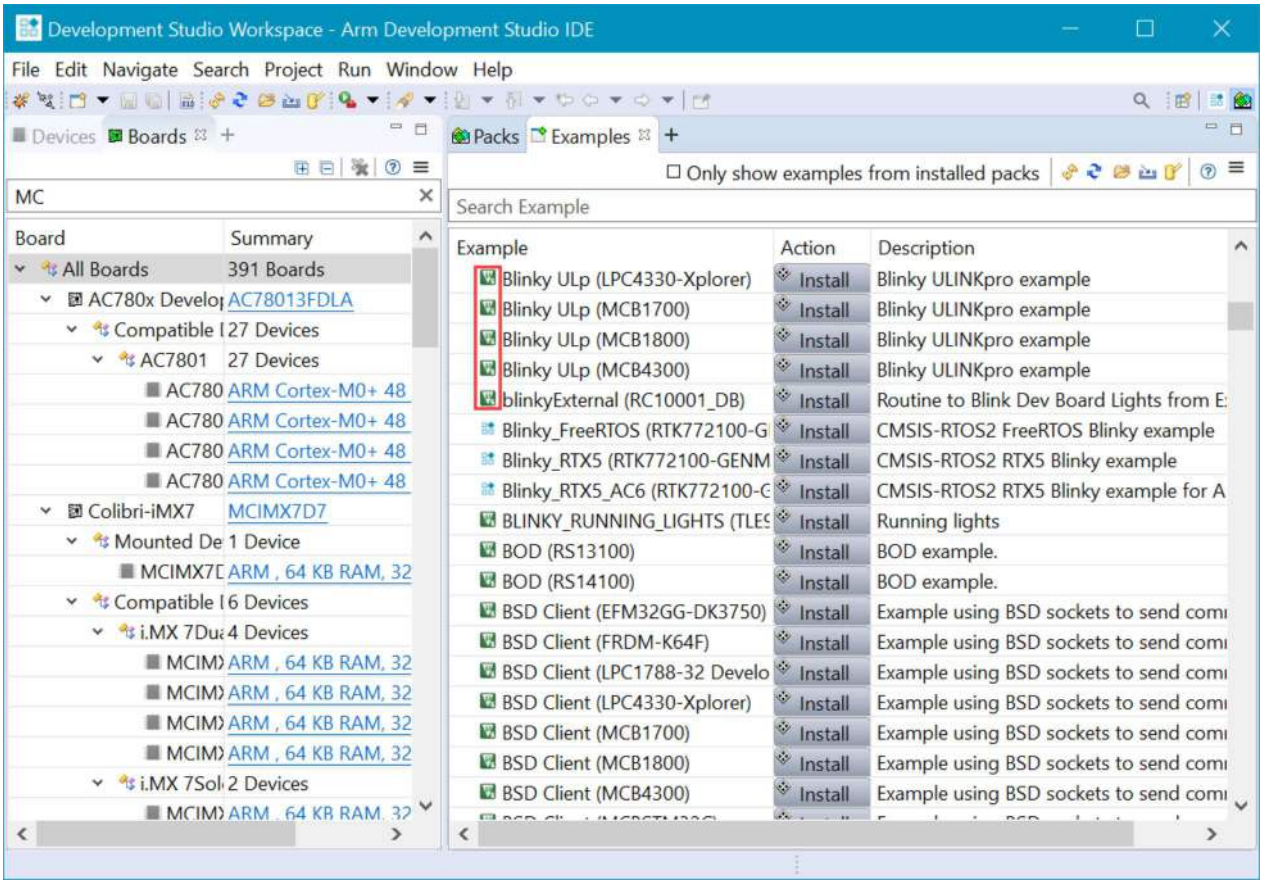
Not all CMSIS packs come with examples. Only examples for installed CMSIS packs are visible by default. Untick **Only show examples from installed packs** to see examples from packs that you have not yet installed.

Figure 11-17: Importing CMSIS Pack example projects



The **CMSIS Pack Manager** shows a µVision® icon if the example is a µVision project and requires conversion.

Figure 11-18: Example µVision projects



Related information

Imported µVision project limitations on page 218

11.6 Create a new Hardware Connection

Using the new **Hardware Connection** wizard in Arm® Development Studio, you can create a connection to a hardware target for debug activities.

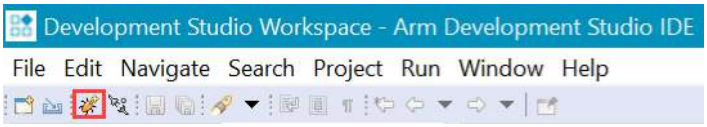
About this task

The **Hardware Connection** wizard allows you to select target information which comes from either a CMSIS pack or a configuration database. This topic describes how to connect to a hardware target using the **Hardware Connection** wizard, where the target is provided by a CMSIS Pack.

Procedure

1. Open the **New Debug Connection** dialog:
   - Click the **New Debug Connection** button at the top of the Development Studio perspective.

Figure 11-19: Create new debug connection from Development Studio perspective.

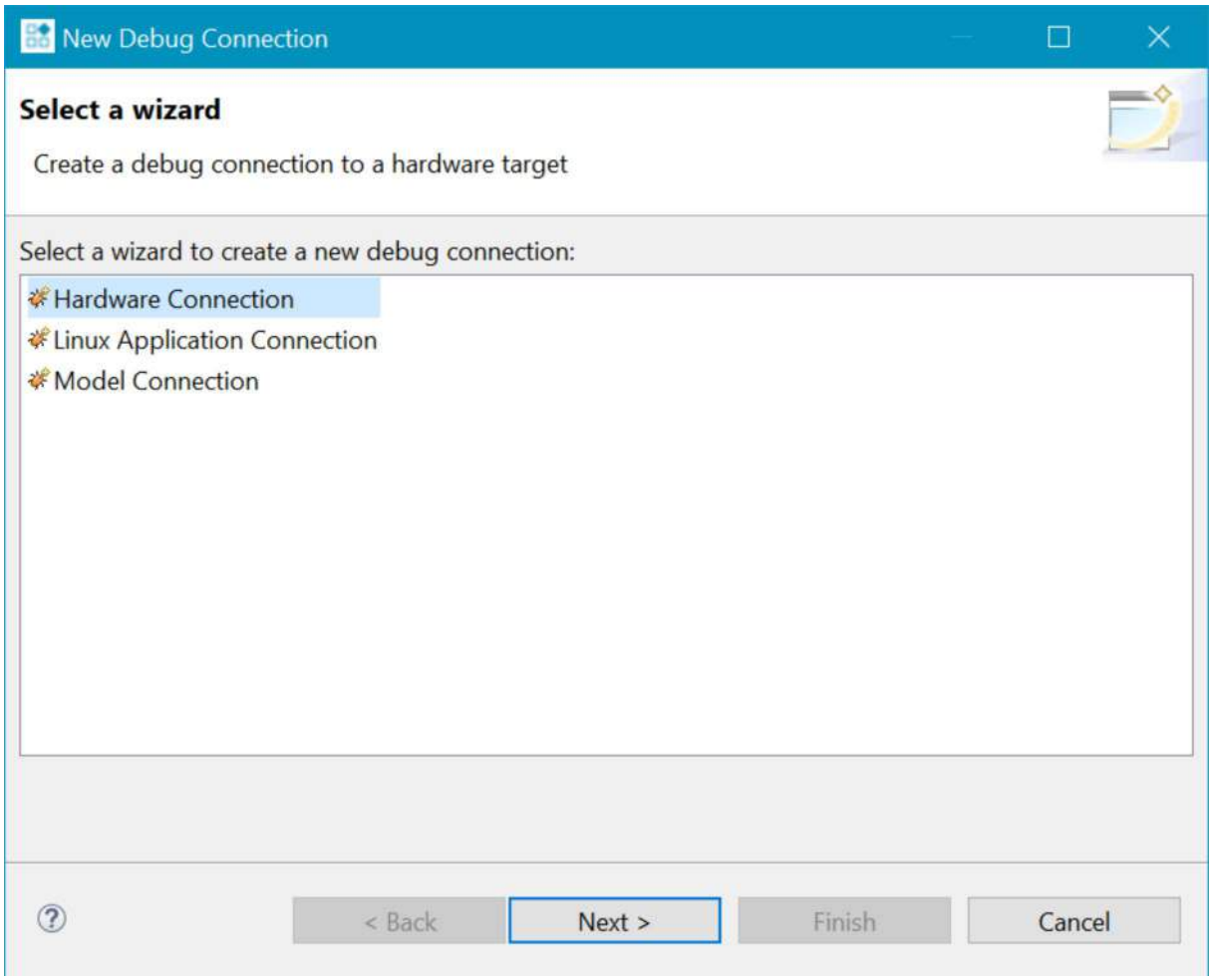


2. Select the **Hardware Connection** wizard and click **Next**.

Note

You can also open the **Hardware Connection** wizard by selecting **File > New > Hardware Connection**.

Figure 11-20: Open Hardware Connection Wizard.

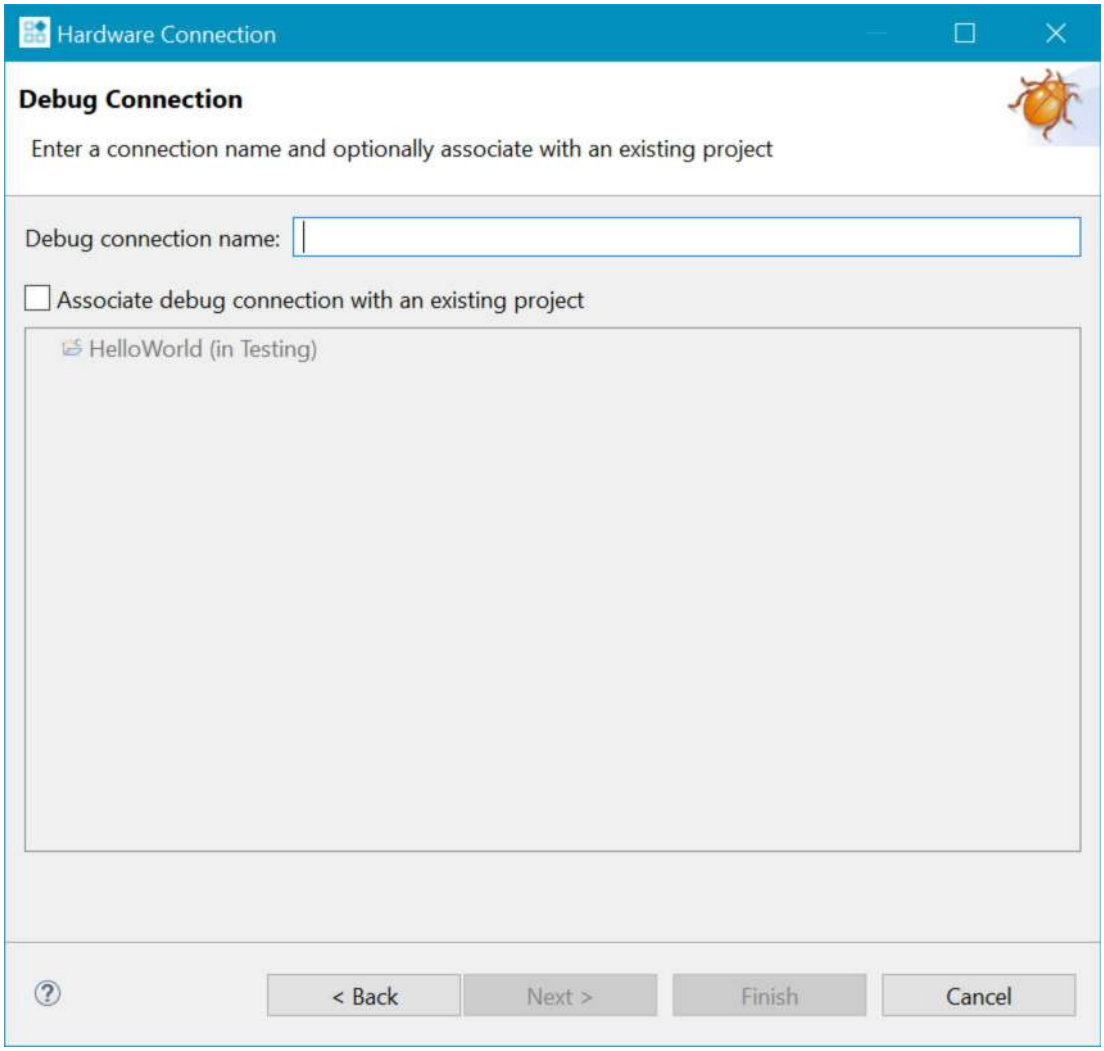


3. Enter a connection name in the **Debug connection name** field and click **Next**.

Note

To associate a new hardware connection with an existing project select **Associate debug connection with an existing project** and choose a project from the list provided.

Figure 11-21: Enter debug connection name.

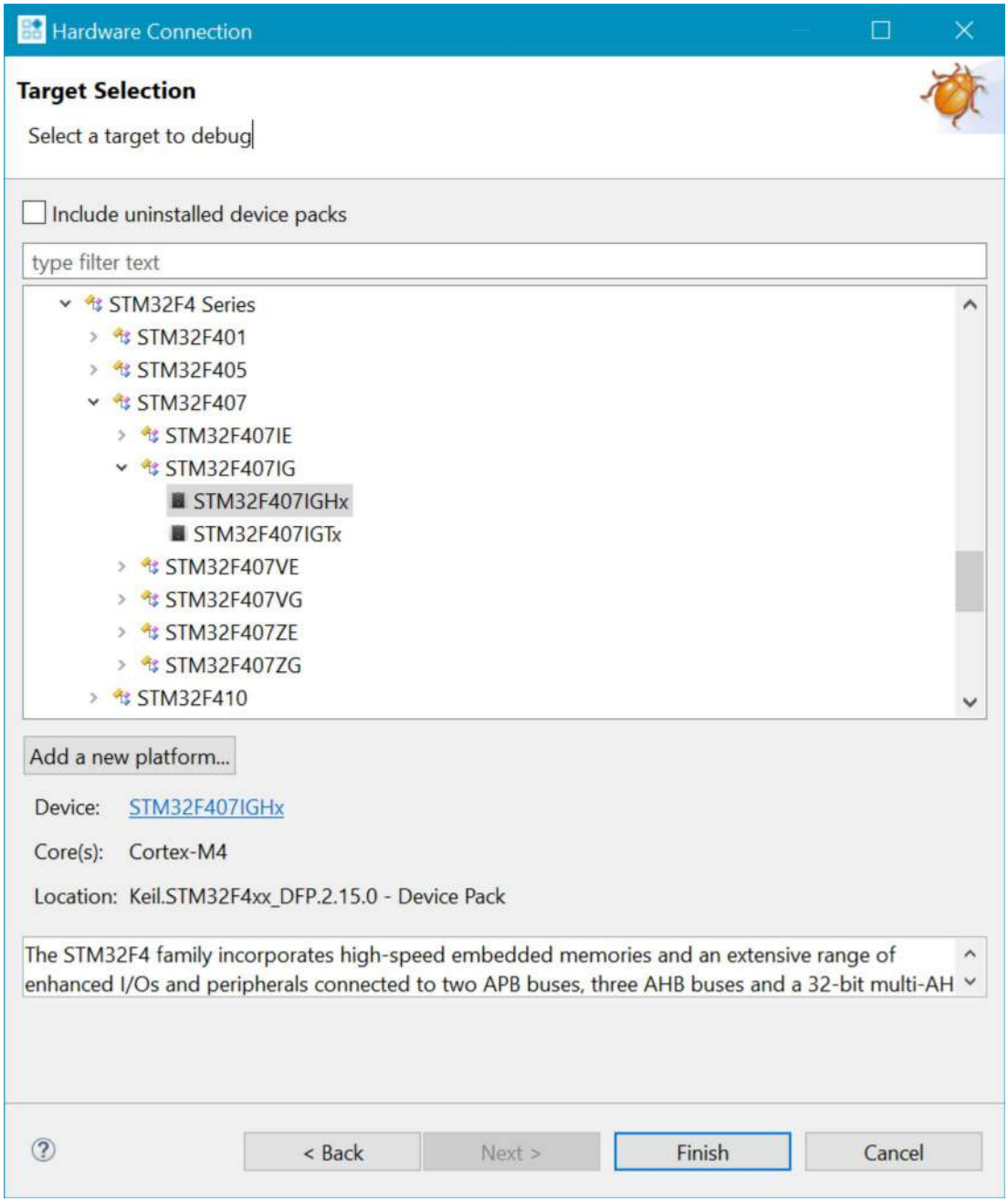


4. Select a hardware target to connect to from the available list and click **Finish**.

Note

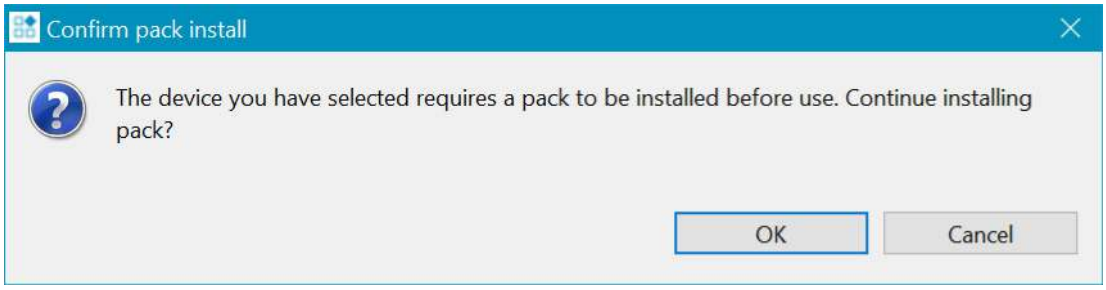
The **Location** entry of the selected target tells you whether the target support is provided by a CMSIS Pack or a Configuration Database (configdb).

Figure 11-22: Select hardware target.



If the selected target uses a CMSIS pack that is not installed, the dialog shown below appears:

Figure 11-23: Confirm CMSIS pack installation.

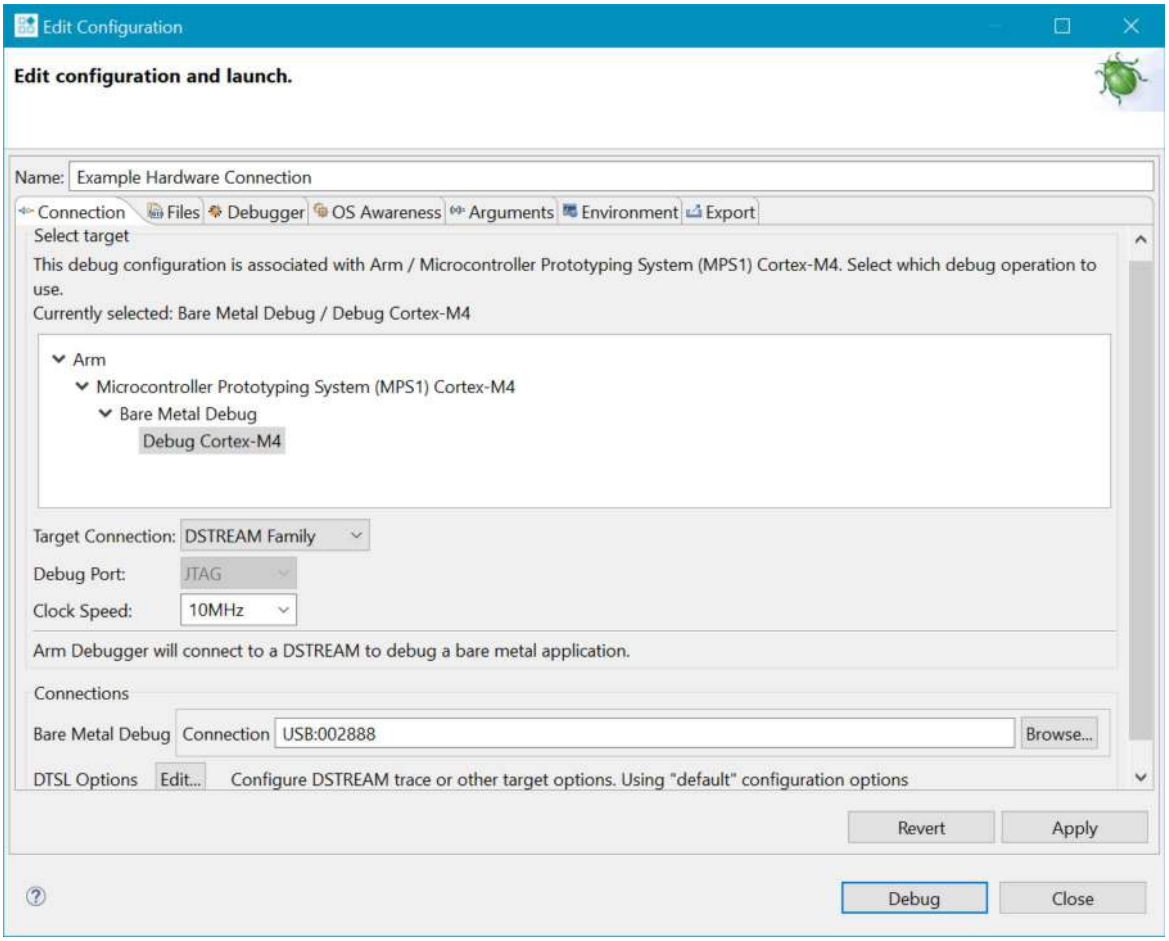


Click **OK** to confirm the pack installation.

When you select the hardware and install any required CMSIS packs, the Arm Development Studio **Edit Configuration** dialog launches. This is presented differently depending on the support for your selected target:

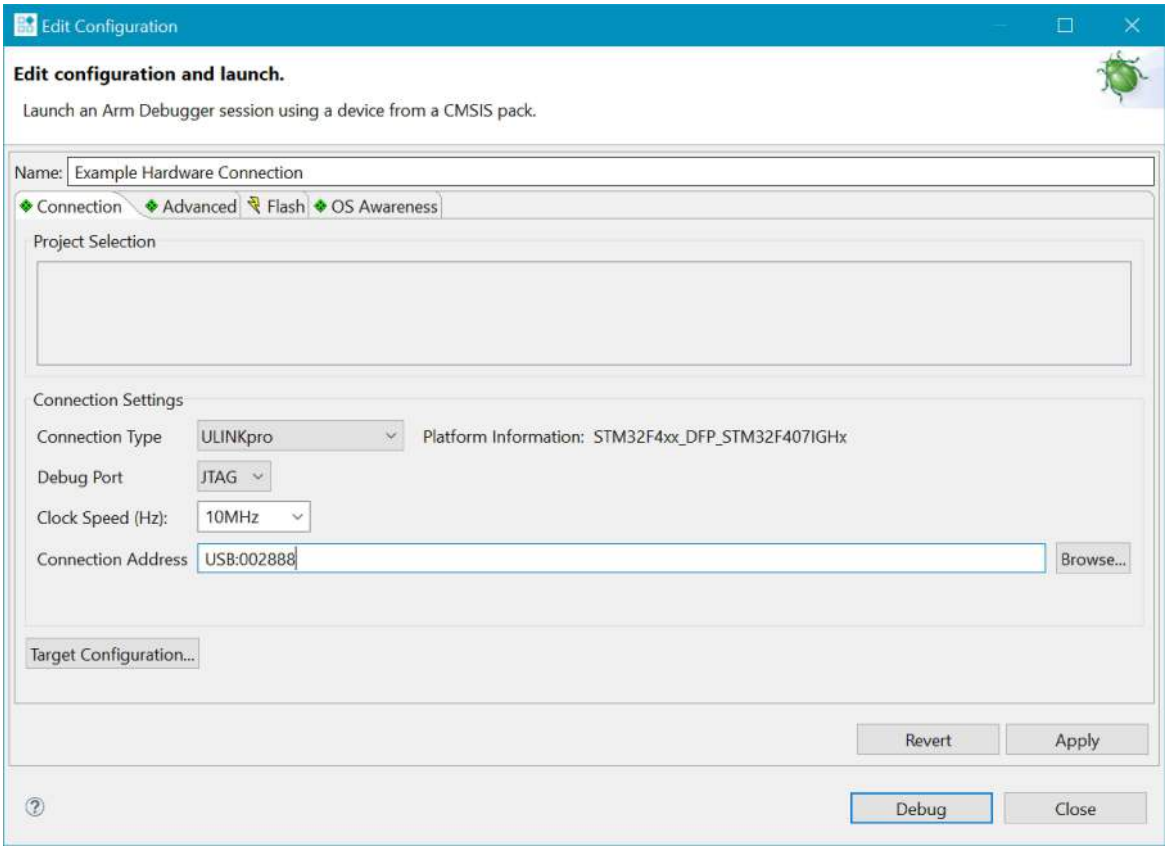
- If the device support for your selected target comes from a configdb, the **Edit Configuration** dialog functions the same as the DS-5 **Debug Configurations** screen, and looks like this:

Figure 11-24: Edit Configuration dialog for configdb targets



- If the device support for your selected target comes from a CMSIS Pack, the **Edit Configuration** dialog looks like this:

Figure 11-25: Edit Configuration dialog for CMSIS pack targets.



Note

This activity assumes the device support for your selected target comes from a CMSIS Pack.

5. Setup and connect to a target using the **Edit Configuration** dialog:
   - In the **Connection** tab:
     a. Select a debug adapter from the **Conection Type** drop-down list.
     b. Select a debug connection method (JTAG or SWD) from the **Debug Port** drop-down list.
     c. Enter a connection address for the debug adapter.

Note

You can browse for the debug adapter by clicking **Browse...**

     d. Click **Target Configuration...** to set the trace connection options.

- In the **Advanced** tab:
  a. If required, add an image file to download to the target by going to **File Settings** and clicking **Add an image**.
  b. Select the required **Run Control**, **Select and Reset**, **Reset Control** and **Scripts** options.
- In the **Flash** tab:
  a. If required, add a flash programming algorithm to the connection by selecting **Programming Algorithms** and then **Add a flash programming algorithm**.
- In the **OS Awareness** tab:
  a. If required, select an OS from the **Select OS awareness** drop-down list.
- In the **Connection** tab:
  a. Click **Apply** and then **Debug** to connect to your selected target and start an Arm Debugger session.

Results

The debug connection status appears in the **Debug Control** view and the created launch configuration appears in the **Project Explorer** view.

11.7 Connect to new or custom hardware

Arm® Development Studio provides a method to add new hardware target configurations for connection and debug purposes. This activity describes how to connect to new or custom hardware in Arm Development Studio.

About this task

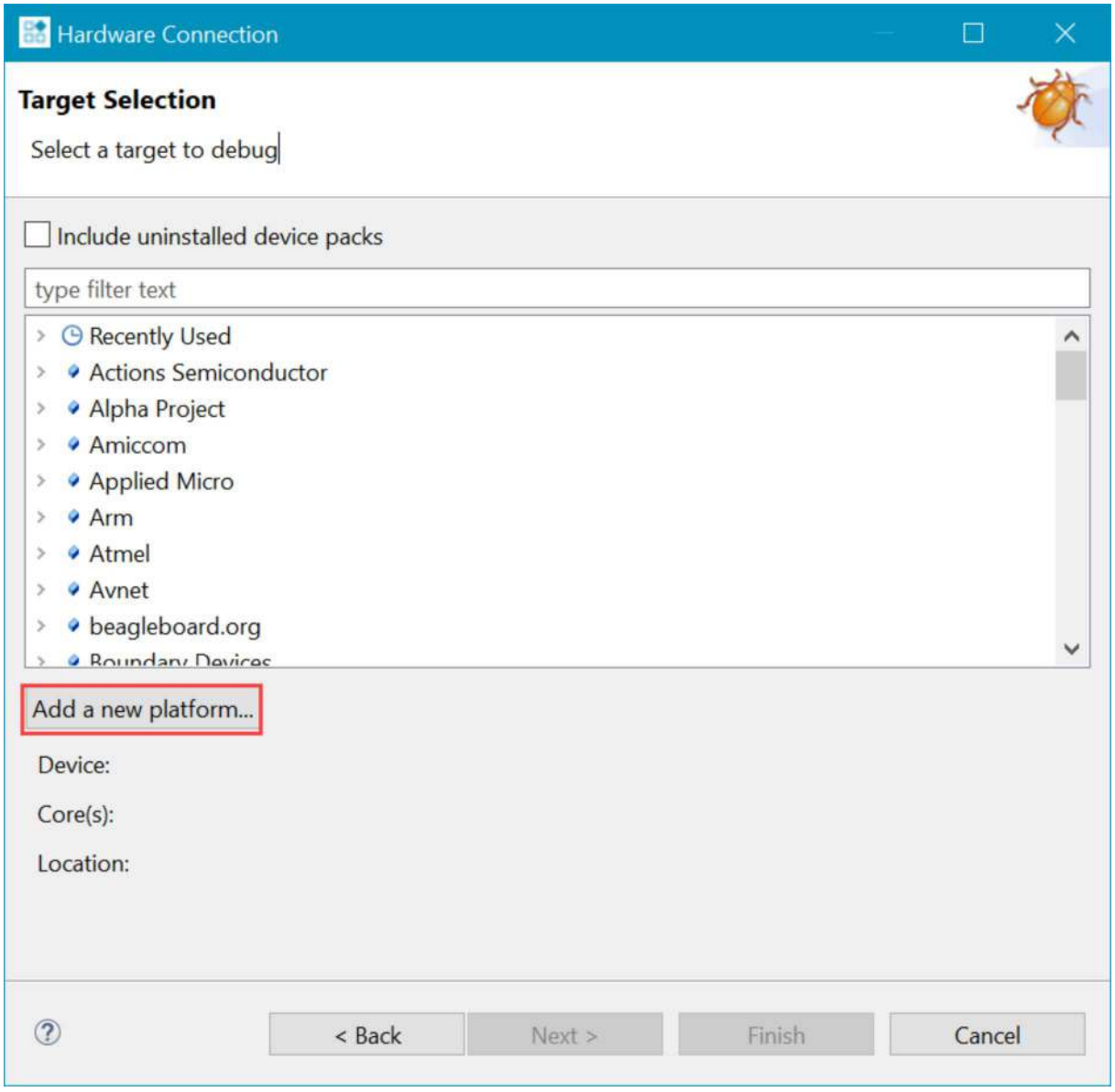
In DS-5, hardware configurations are added using a separate perspective, **Platform Configuration Editor**. In Arm Development Studio, adding hardware configurations is part of the new **Hardware Connection** wizard.

Procedure

1. Open the **Hardware Connection** wizard:
   a) Click the **New Debug Connection** icon in the **Debug Control** view, in the **View Menu** listing of the **Debug Control** view, or at the top of the Development Studio perspective.
   b) Select **Hardware Connection** and click **Next**.
2. Enter a connection name in the **Debug Connection name** field and click **Next**.
3. Click **Add a new platform...**.

If not already present, Arm Development Studio automatically creates a configuration database (**ExtensionDB**) to store the new hardware configuration.

Figure 11-26: Add new platform

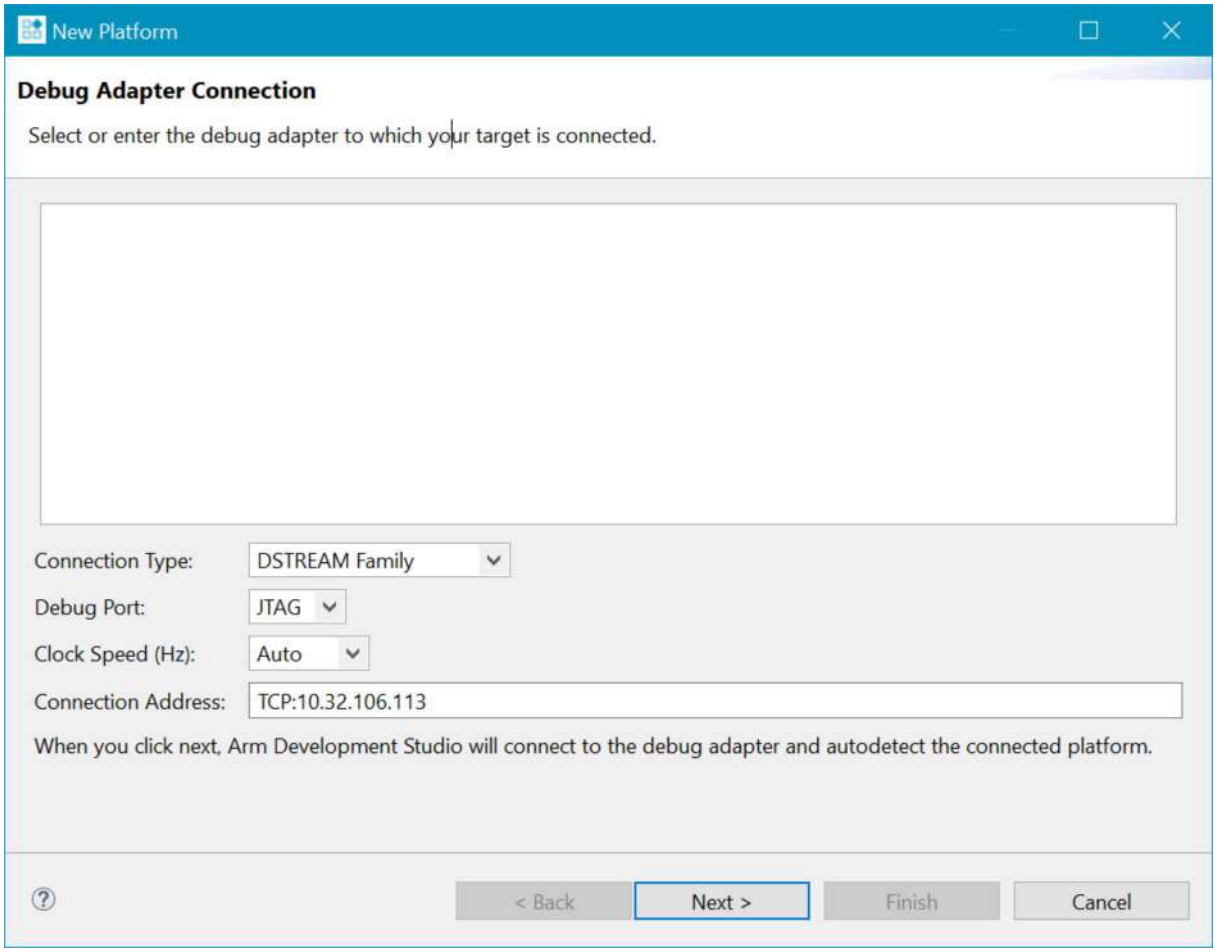


4. Select the appropriate debug adapter connection in the **Conection Type** drop-down list.

Note

The **Debug Adapter Connection** view automatically lists any debug adapters of the selected type. Unlike DS-5, Arm Development Studio can use ULINK devices for autodetection purposes. If the debug adapter is not discovered, you can enter the debug adapter connection information in the **Connection Address** field.

Figure 11-27: Select a debug adapter



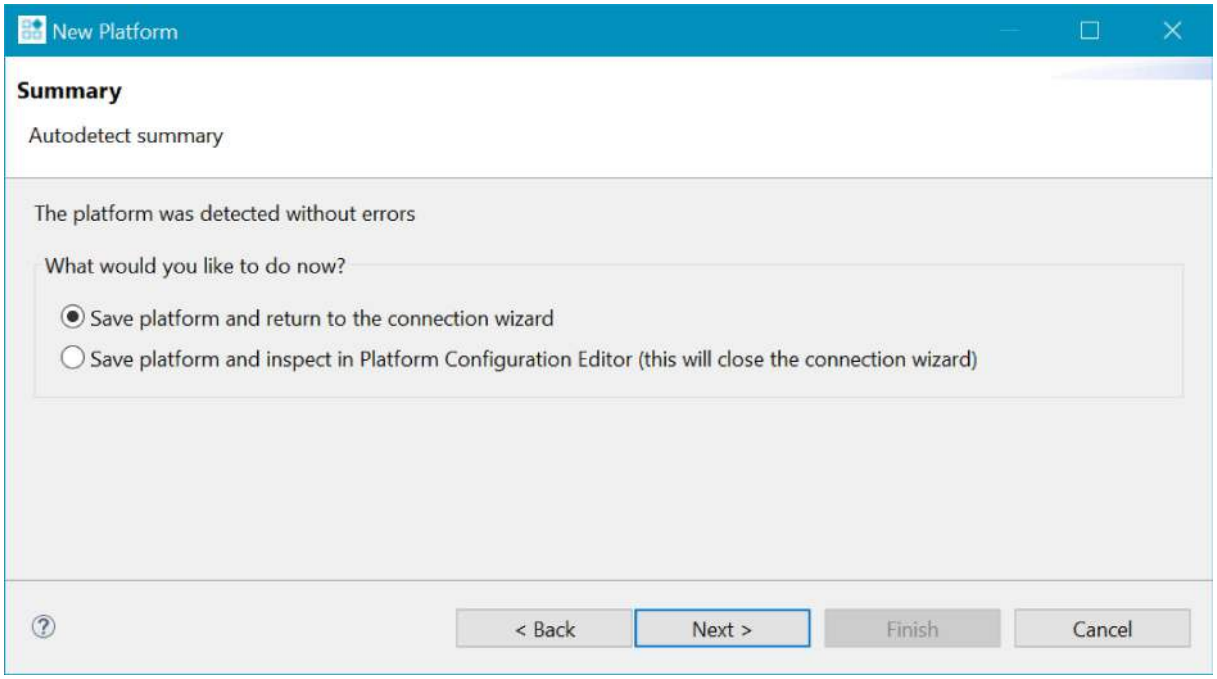
5. Click **Next** to start the hardware target autodetection process.

Note

A platform configuration is created for the attached target during the autodetection process. You might be prompted to update the debug adapter firmware before the autodetection process begins. The debug adapter firmware update process is the same as it is for DS-5.

6. When the autodetection process completes, choose whether or not to inspect the platform in the **Platform Configuration Editor**.

Figure 11-28: Select a debug adapter



- To continue using the **Hardware Connection** wizard, select **Save platform and return to the connection wizard**.
- To exit the **Hardware Connection** wizard and enter the **Platform Configuration Editor**, select **Save platform and inspect in Platform Configuration Editor**.

Note

The Arm Development Studio **Platform Configuration Editor** (**PCE**) functions the same as DS-5's **PCE**.

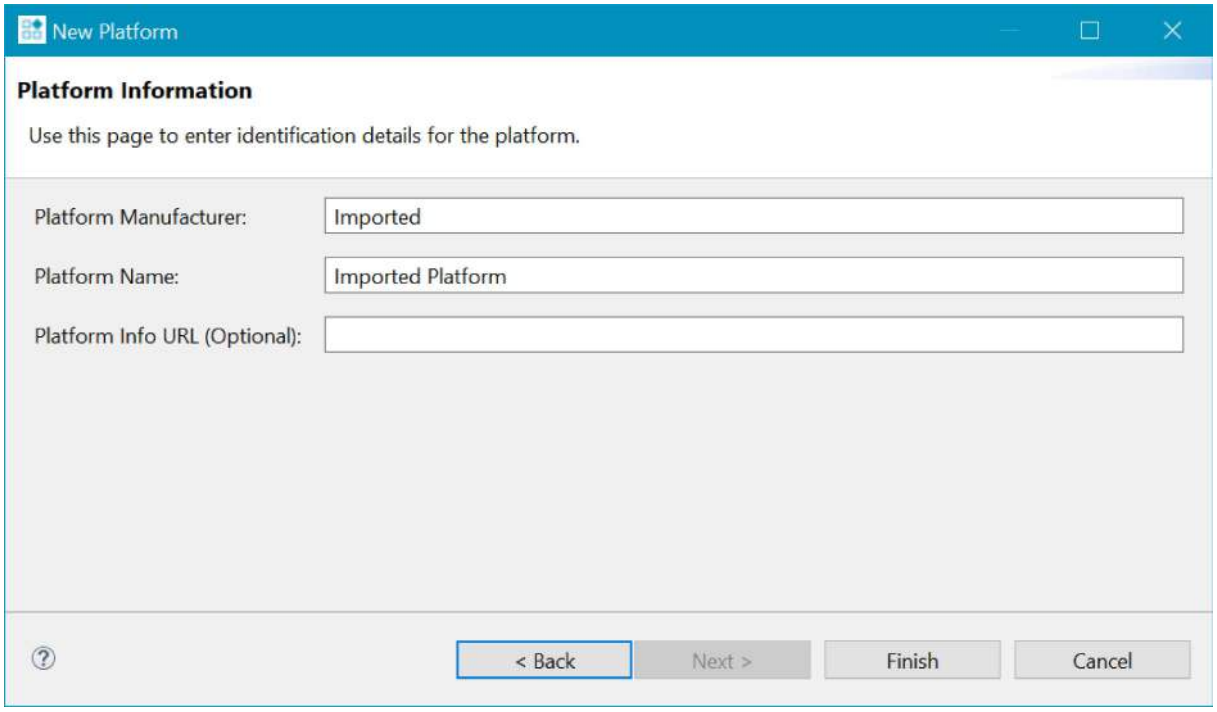
7. Click **Next**.

Note

This activity assumes that you have chosen to continue using the Hardware Connection wizard.

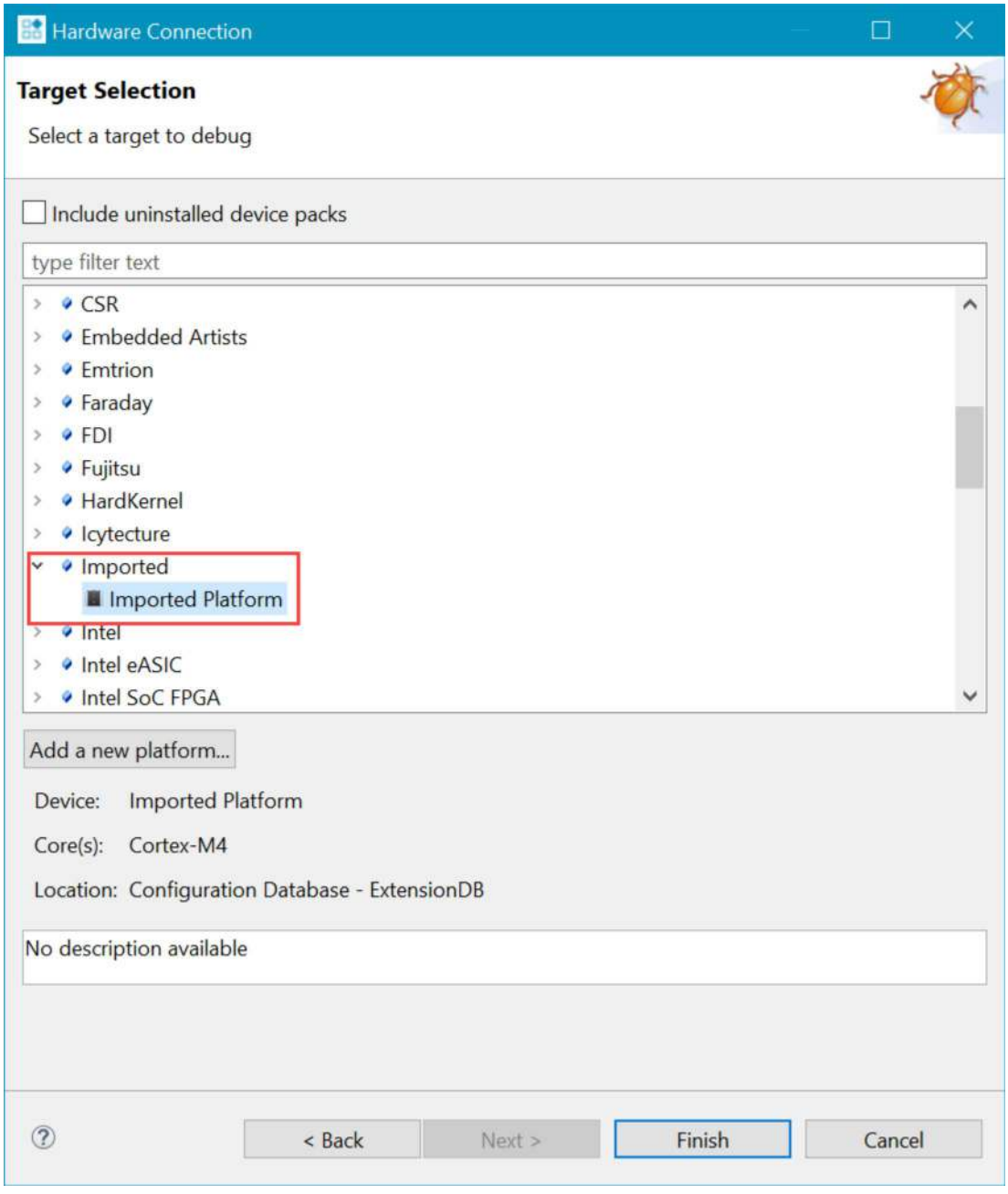
8. Enter platform identification details into the **Platform** fields and click **Finish**.

Figure 11-29: Enter identification details for the platform



9. Select the new hardware configuration in the **Target Selection** view and click **Finish**.

Figure 11-30: Select the new hardware configuration



Results

The new hardware target configuration appears in the **Edit Configuration** view.

11.8 Create a new Linux application connection

Arm® Development Studio provides a method to connect to and debug Linux applications using `gdbserver`.

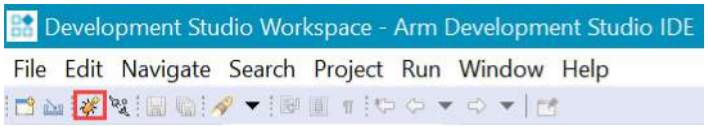
About this task

Arm Development Studio adds a new **Linux Application Connection** wizard to help you create connections to a Linux application running on a target. This activity describes how to connect to a Linux application in Arm Development Studio using the **Linux Application Connection** wizard.

Procedure

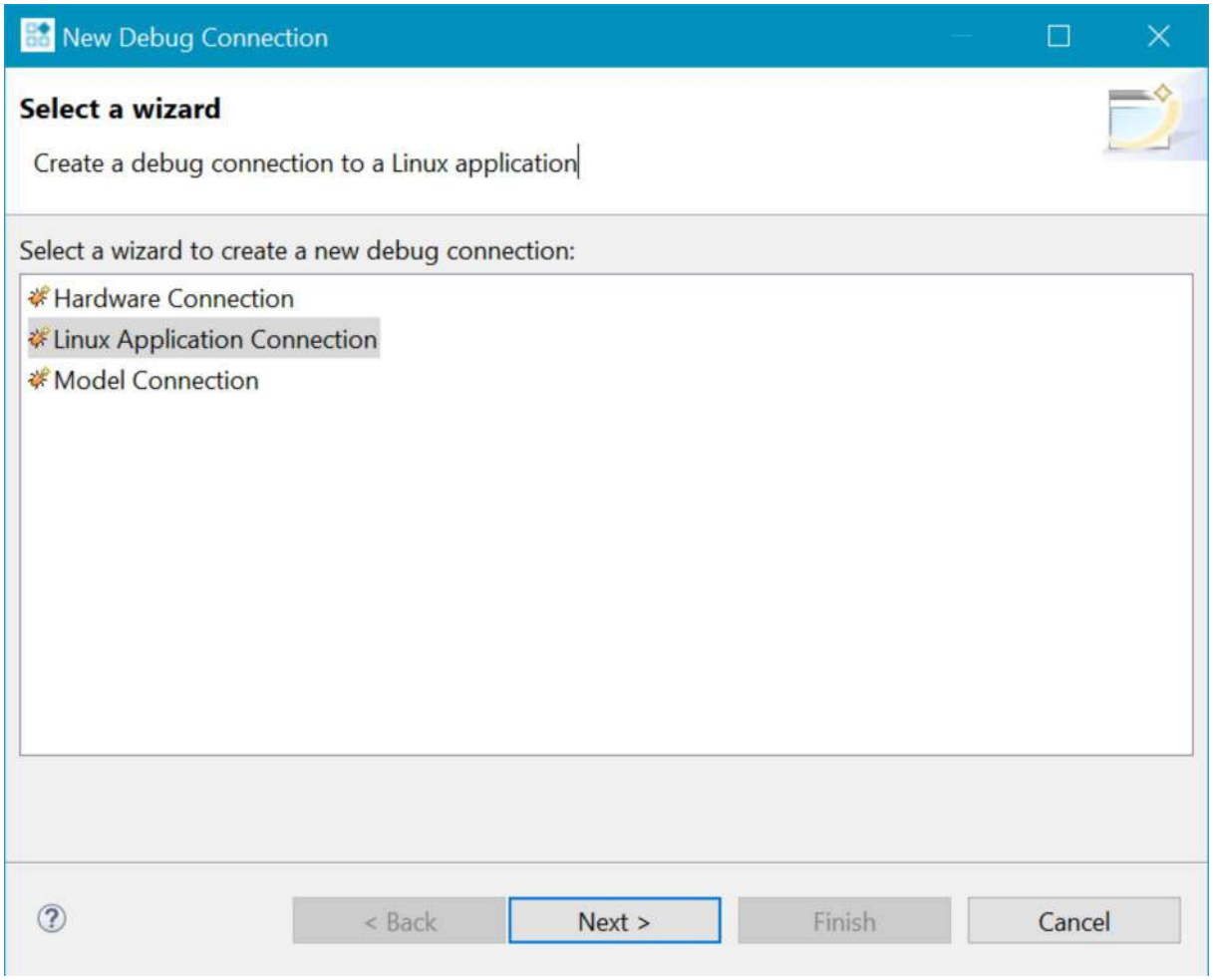
1. Open the **New Debug Connection** dialog:
   a) Click the **New Debug Connection** button at the top of the Development Studio perspective.

   Figure 11-31: Create new debug connection from Development Studio perspective.

2. Select **Linux Application Connection** and click **Next**.

Figure 11-32: Select Linux Application Connection



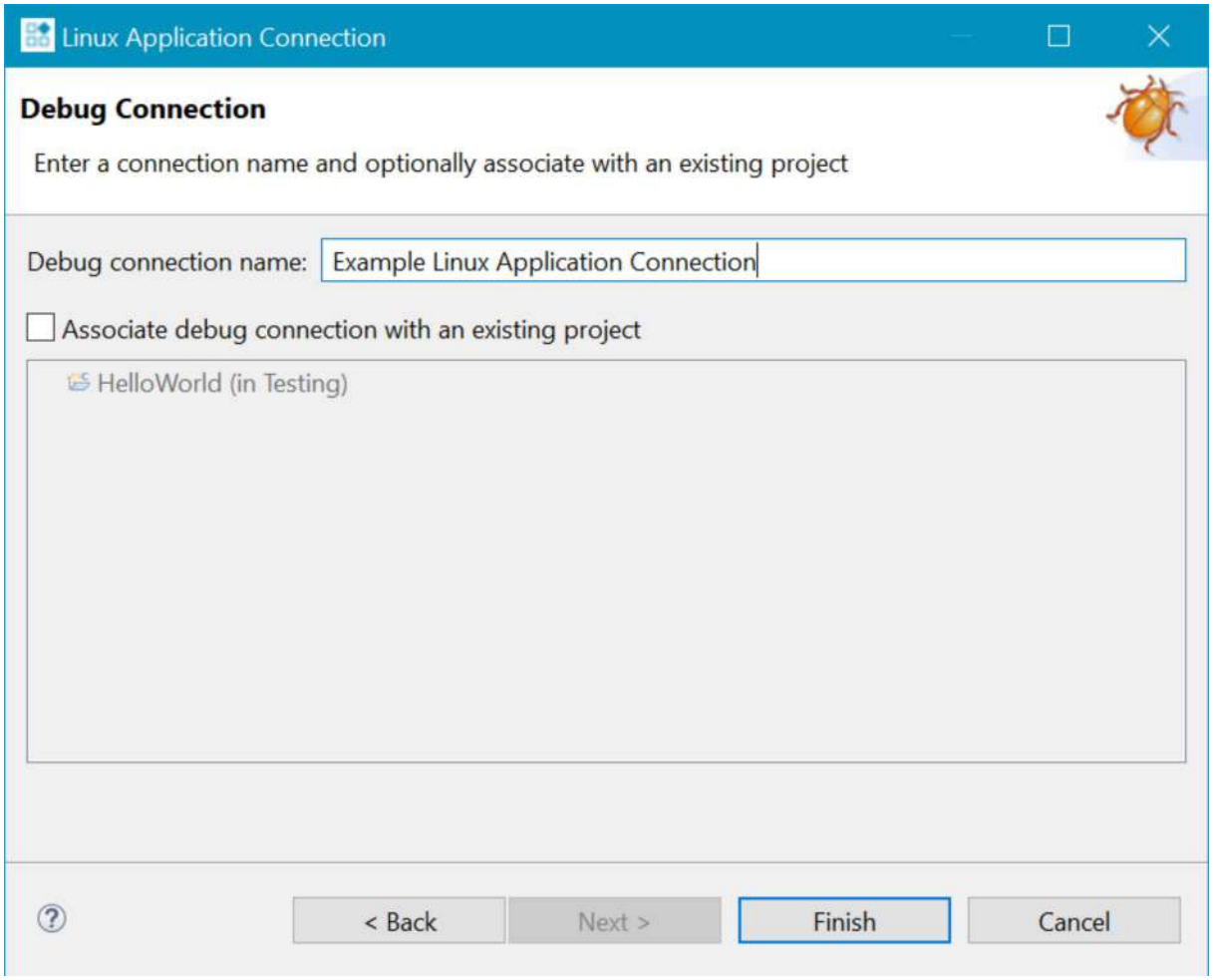
3. Enter a connection name in the **Debug connection name** field and click **Finish**.

Note

To associate a new Linux application connection with an existing project, select **Associate debug connection with an existing project** and choose a project from the provided list.

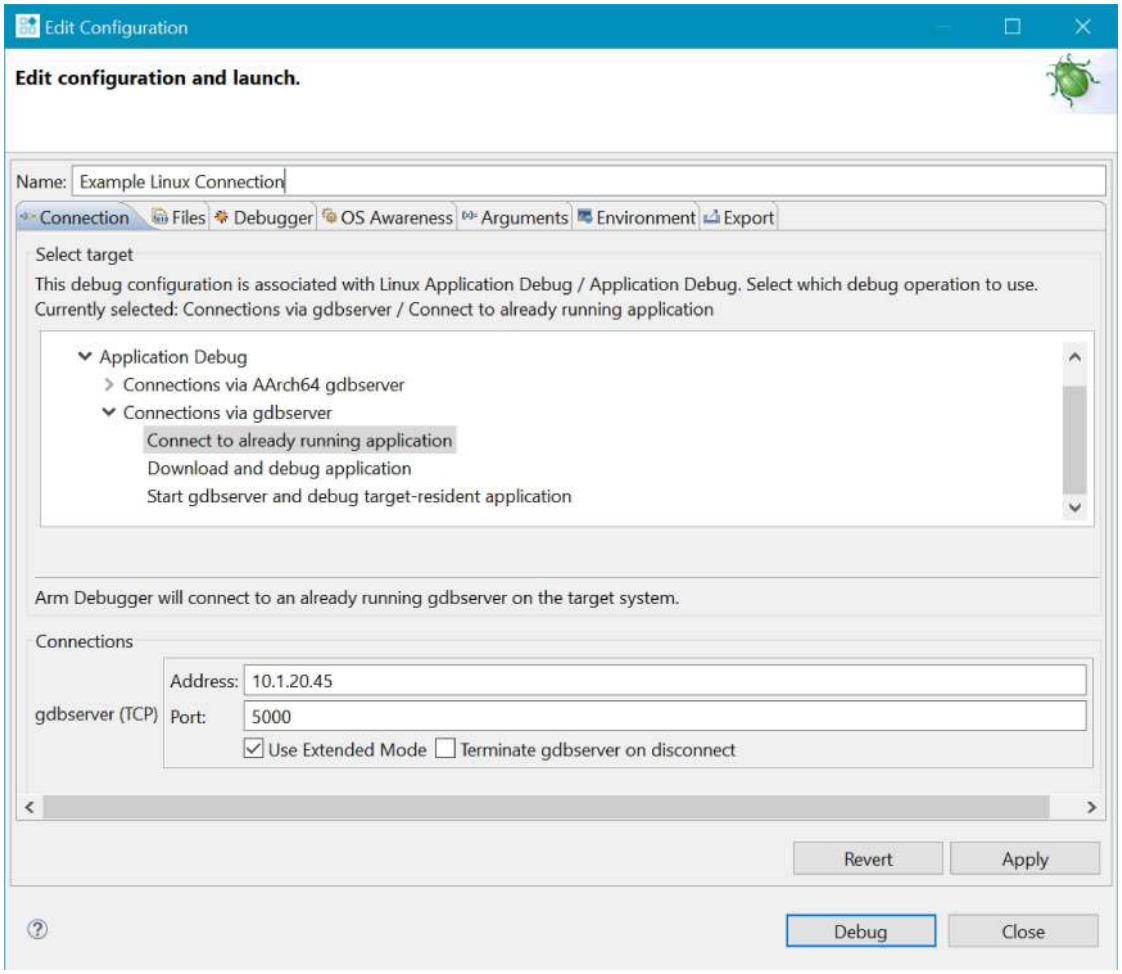
This activity assumes you have not associated the connection with an existing project.

Figure 11-33: Enter Linux Application Connection name



4. In the **Edit Configuration** dialog box:
   - If you want to connect to a target with the application and `gdbserver` already running on it:
     a. In the **Connection** tab, select **Connect to already running application**.
     b. In the **Connections** area, enter the **Address** and **Port** details of the target.
     c. If you want to terminate the `gdbserver` when disconnecting from the FVP, select **Terminate gdbserver on disconnect**.

Figure 11-34: Edit Linux app connection details



d. In the **Files** tab, use the **Load symbols from file** option in the **Files** panel to specify symbol files.

e. In the **Debugger** tab, specify the actions that you want the debugger to perform after connecting to the target.

f. If required, click the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.

g. If required, click the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.

- If you want to download your application to the target system and then start a `gdbserver` session to debug the application, select **Download and debug application**.

Note

This connection requires that `ssh` and `gdbserver` is available on the target.

---

a. In the **Connections** area, enter the **Address** and **Port** details of the target.
b. In the **Files** tab, specify the **Target Configuration** details:
   - Under **Application on host to download**, select the application to download onto the target from your host filesystem or workspace.
   - Under **Target download directory**, specify the download directory location.
   - Under **Target working directory**, specify the target working directory.
   - If required, use the **Load symbols from file** option in the **Files** panel to specify symbol files.
c. In the **Debugger** tab, specify the actions that you want the debugger to perform after it connects to the target.
d. If required, click the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.
e. If required, click the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.

- If you want to connect to your target, start `gdbserver`, and then debug an application already present on the target, select **Start gdbserver and debug target resident application**, and configure the options.
  a. In the **Model parameters** area, the **Enable virtual file system support** option maps directories on the host to a directory on the target. The Virtual File System (VFS) enables the FVP to run an application and related shared library files from a directory on the local host.
     - The **Enable virtual file system support** option is selected by default. If you do not want virtual file system support, deselect this option.
     - If the **Enable virtual file system support** option is enabled, your current workspace location is used as the default location. The target sees this location as a writable mount point.
  b. In the **Files** tab, specify the location of the **Application on target** and the **Target working directory**. If you need to load symbols, use the **Load symbols from file** option in the **Files** panel.
  c. In the **Debugger** tab, specify the actions that you want the debugger to perform after connecting to the target.
  d. If required, click the **Arguments** tab to enter arguments that are passed to the application when the debug session starts.
  e. If required, click the **Environment** tab to create and configure the target environment variables that are passed to the application when the debug session starts.

5. Click **Apply** to save the configuration settings.
6. Click **Debug** to connect to the target and start debugging.

Results

A debug connection is created to your chosen Linux application target.

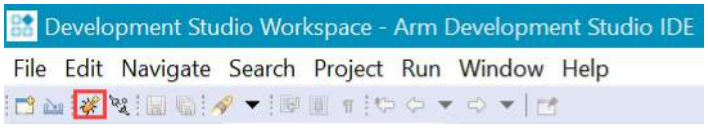
11.9 Create a new model connection

Arm® Development Studio provides a method to connect to and debug models using a new **Model Connection** wizard. This activity describes how to connect to a model that is shipped with Arm Development Studio, using the new **Model Connection** wizard.

Procedure

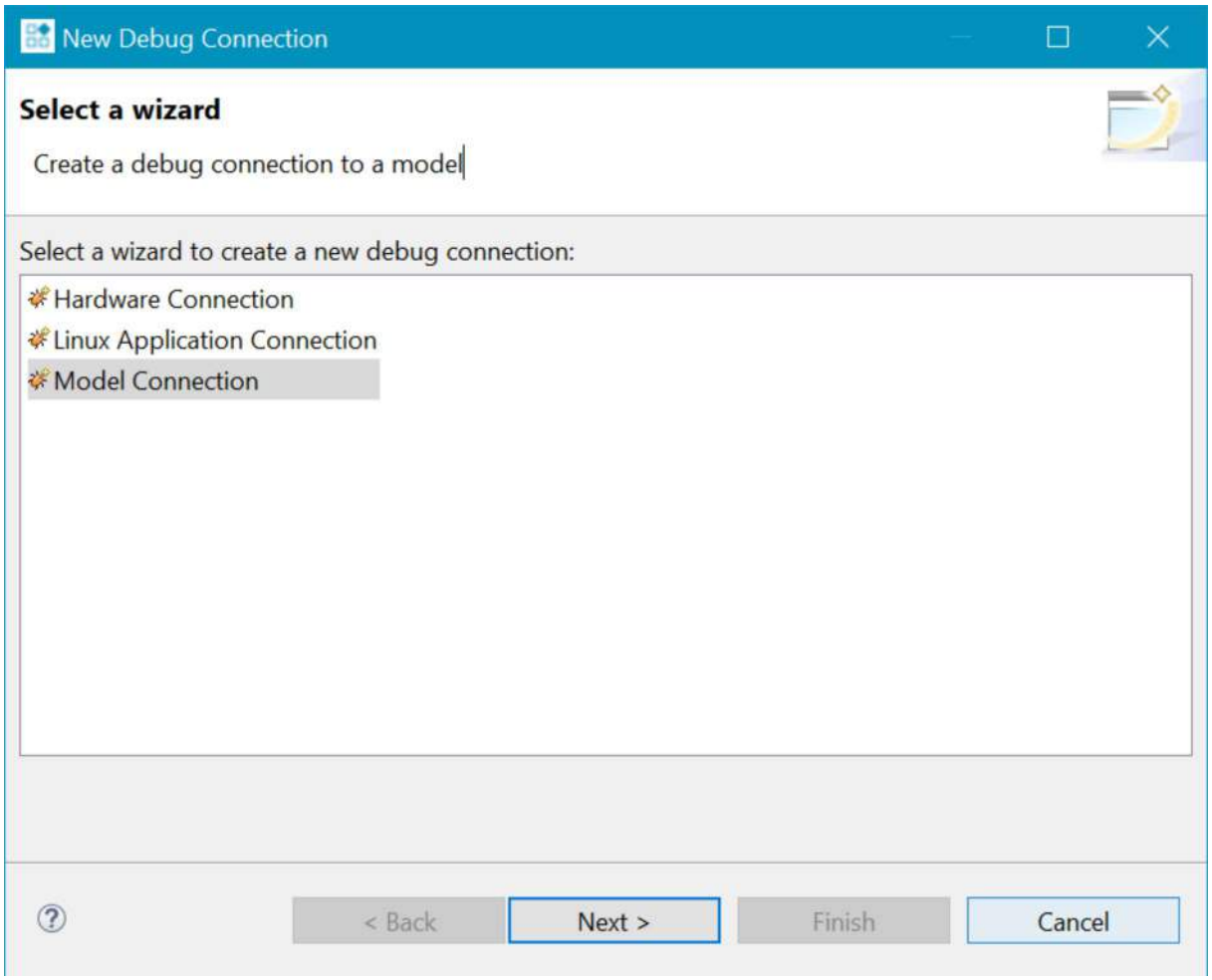
1. Open the **New Debug Connection** dialog:
   a) Click the **New Debug Connection** button at the top of the Development Studio perspective.

   Figure 11-35: Create new debug connection from Development Studio perspective.



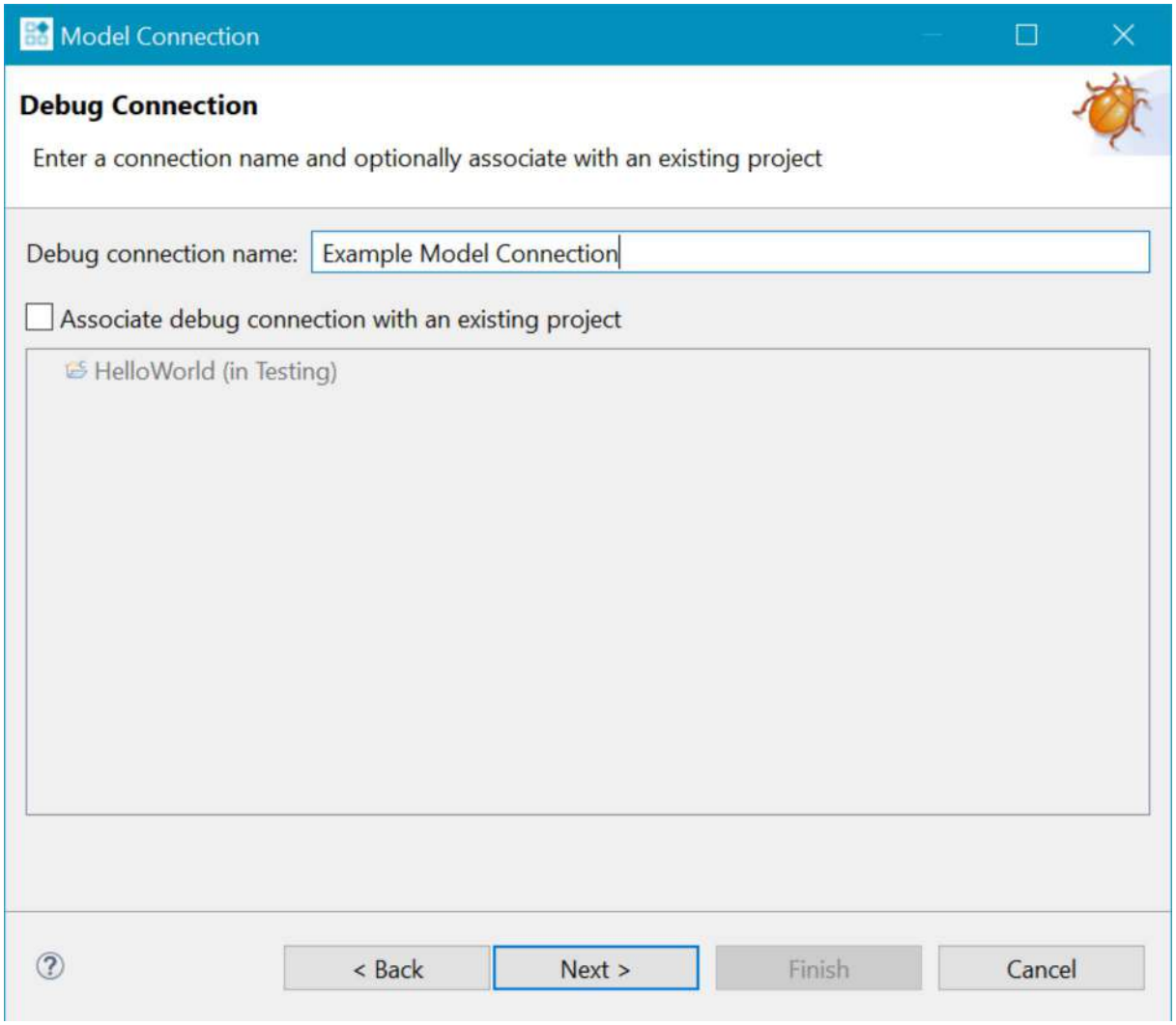
2. Select **Model Connection** and click **Next**.

   Figure 11-36: Select Model Connection wizard



3. Enter a new connection name in the **Debug connection name** field and click **Next**.

Figure 11-37: Enter Model Connection name

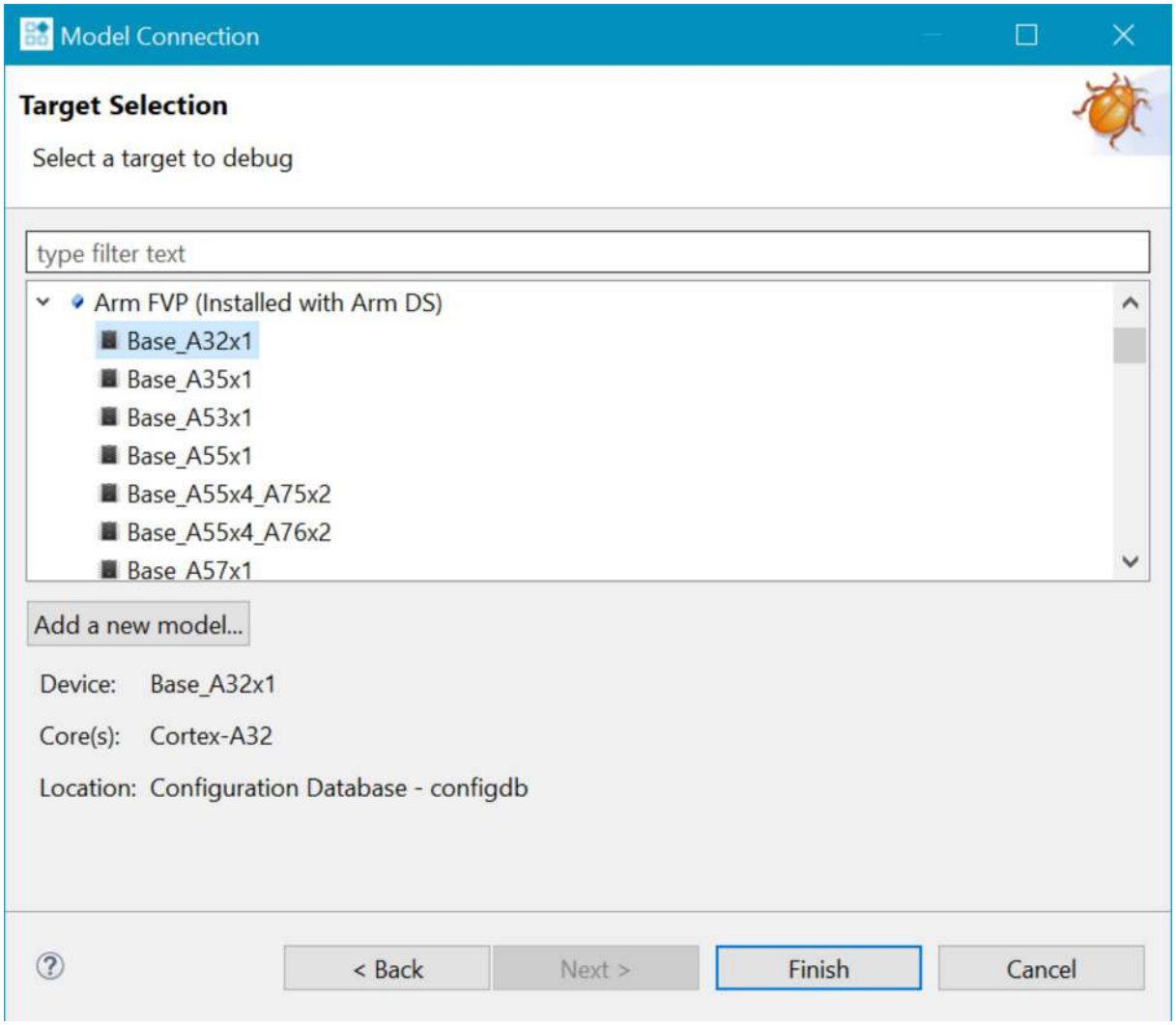


4. Select a model to connect to from the available list or click **Add a new model..** to add a new model configuration to Arm Development Studio.

Note

See Connect to new or custom models for more information about connecting to new models.

Figure 11-38: Select a target model for the connection



5. Click **Finish**.

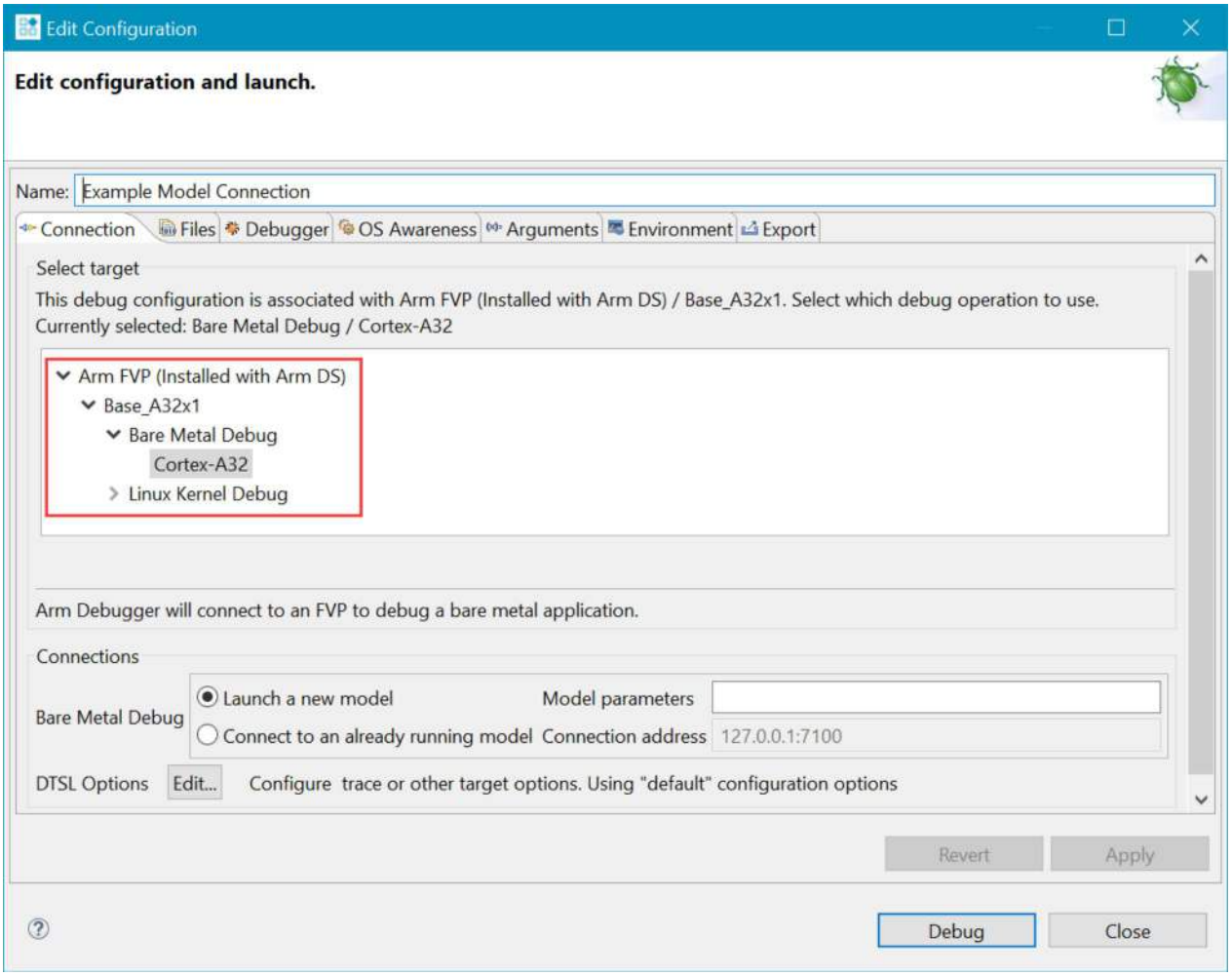
Results

The **Edit Configuration** dialog opens.

Note

The Arm Development Studio **Edit Configuration** dialog provides the same functions as the DS-5 **Debug Configurations** dialog. The main difference is that the Arm Development Studio **Edit Configuration** dialog only shows the configuration details for the selected model under the **Select target** field in the **Connection** tab.

Figure 11-39: Edit Configuration dialog



Next steps

Make any necessary changes to the debug configuration in the **Edit Configuration** dialog.

Related information

Debug Configurations - Connection tab
Debug Configurations - Files tab
Debug Configurations - Debugger tab
Debug Configurations - OS Awareness tab
Debug Configurations - Arguments tab
Debug Configurations - Environment tab
Debug Configurations - Export tab

11.10 Connect to new or custom models

Arm® Development Studio provides a method to add new model configurations for connection and debug purposes. This activity describes how to add model configurations to the configuration database using the new **Model Connection** wizard.

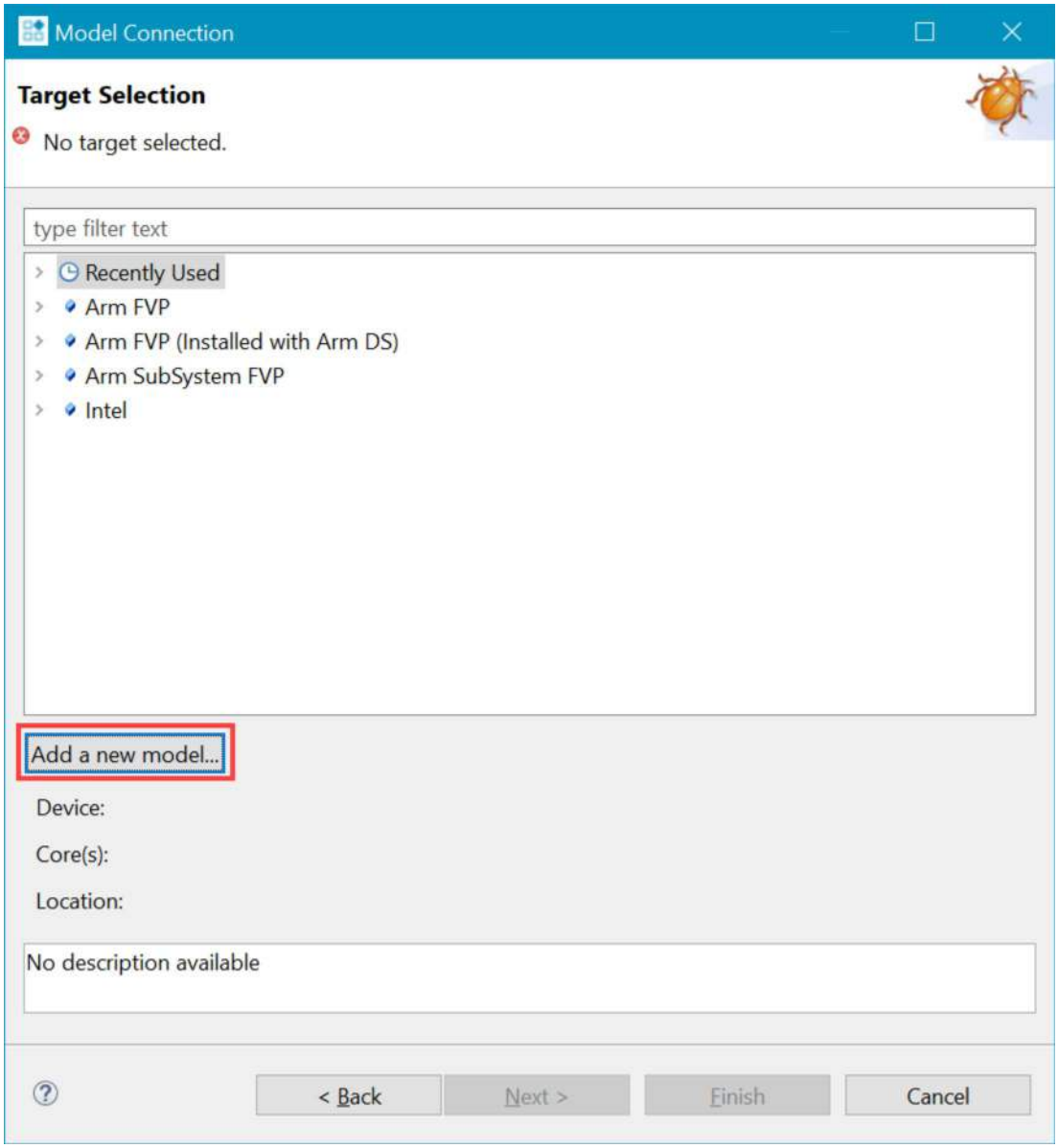
About this task

In addition to the **Model Configuration** wizard, you can add Arm Development Studio model configurations to the configuration database using the new **Model Connection** wizard.

Procedure

1. Open the **Model Connection** wizard:
   a) Select **File > New > Model Connection**.
2. Enter a connection name in the **Debug connection name** field and click **Next**.
3. Click **Add a new model...**

Figure 11-40: Add a new model in the Model Connection wizard.

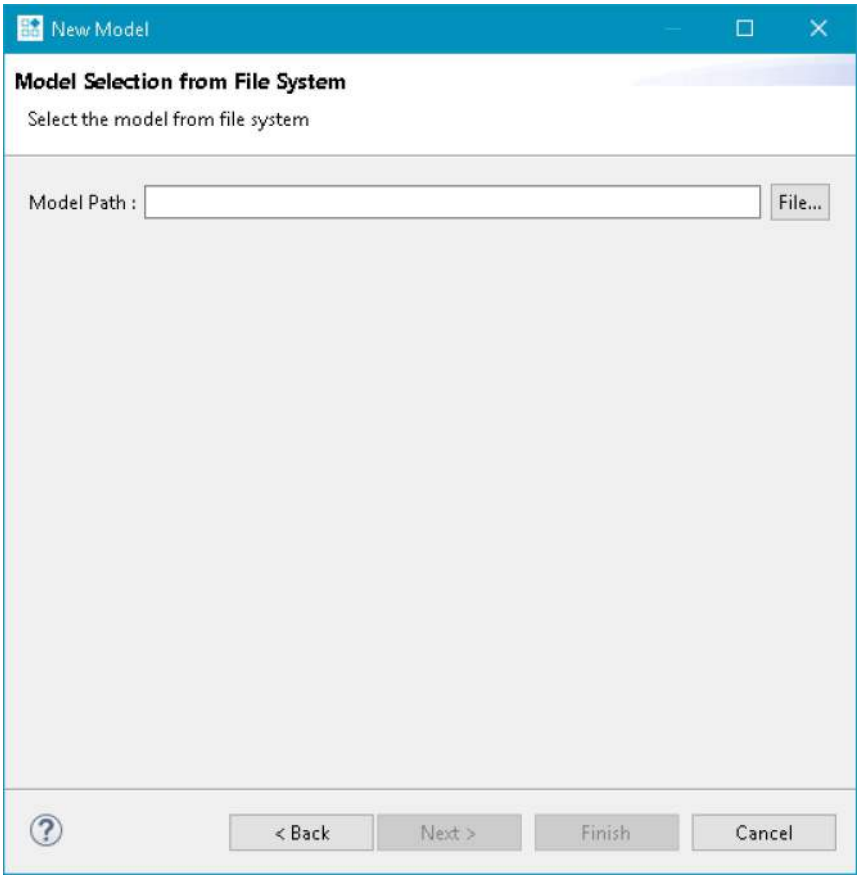


4. Select a model interface for connecting to your model. You have two interface options - Component Architecture Debug Interface (CADI) or Iris.

   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.

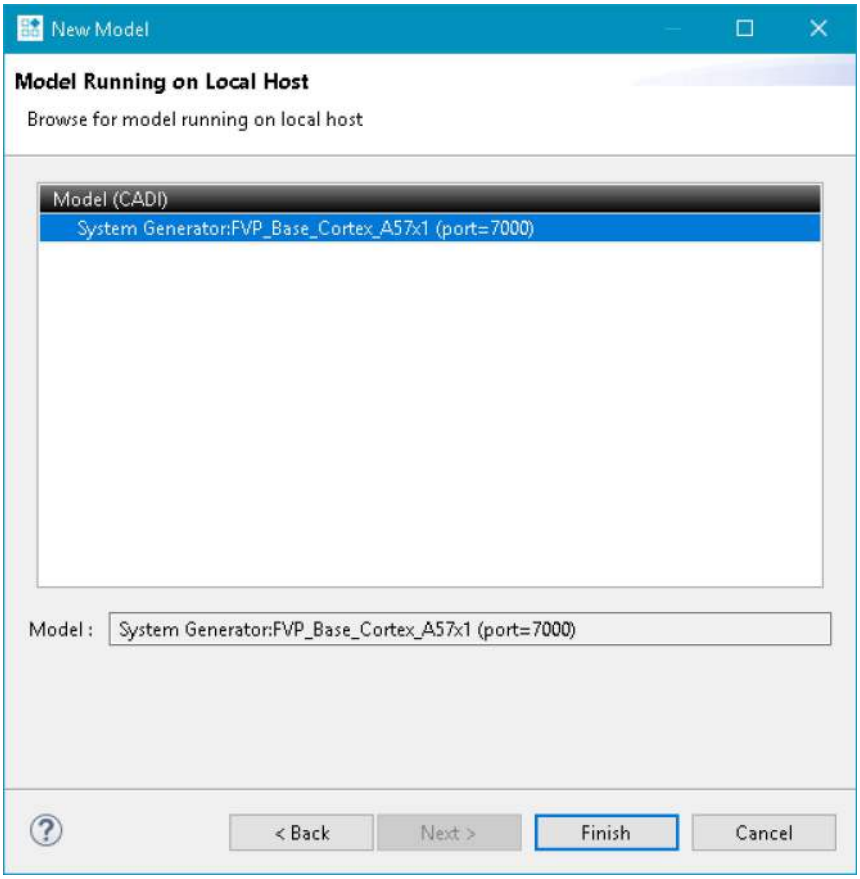
Figure 11-41: Select model from file system



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 11-42: Browse for model running on local host



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:

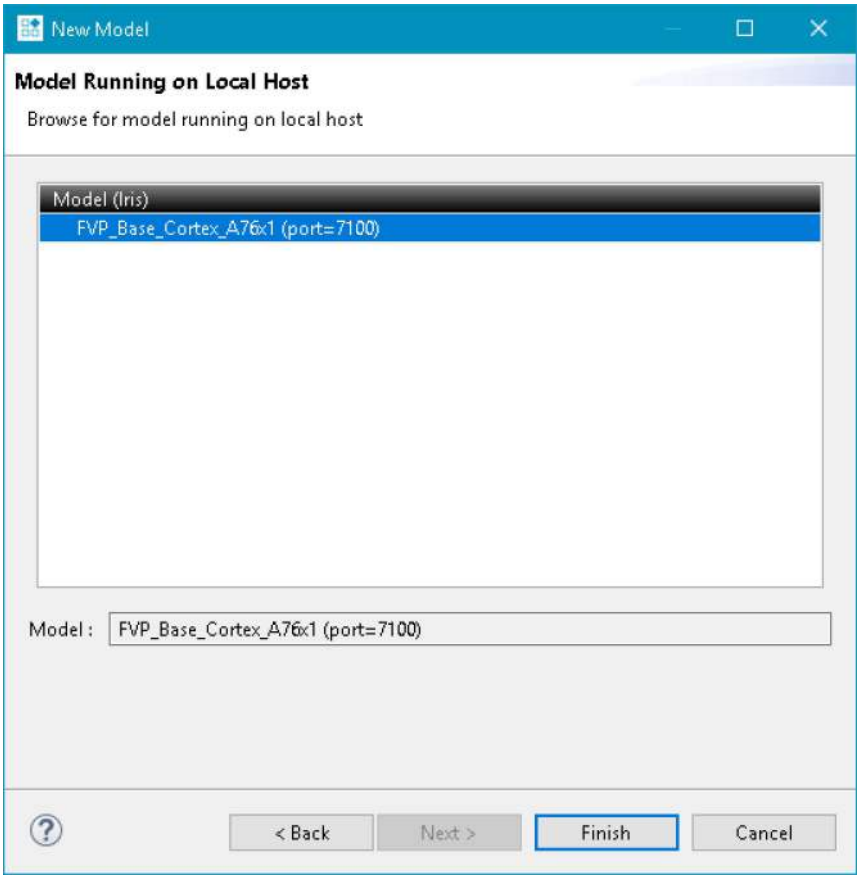
Note

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 11-43: Browse for model running on local host



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

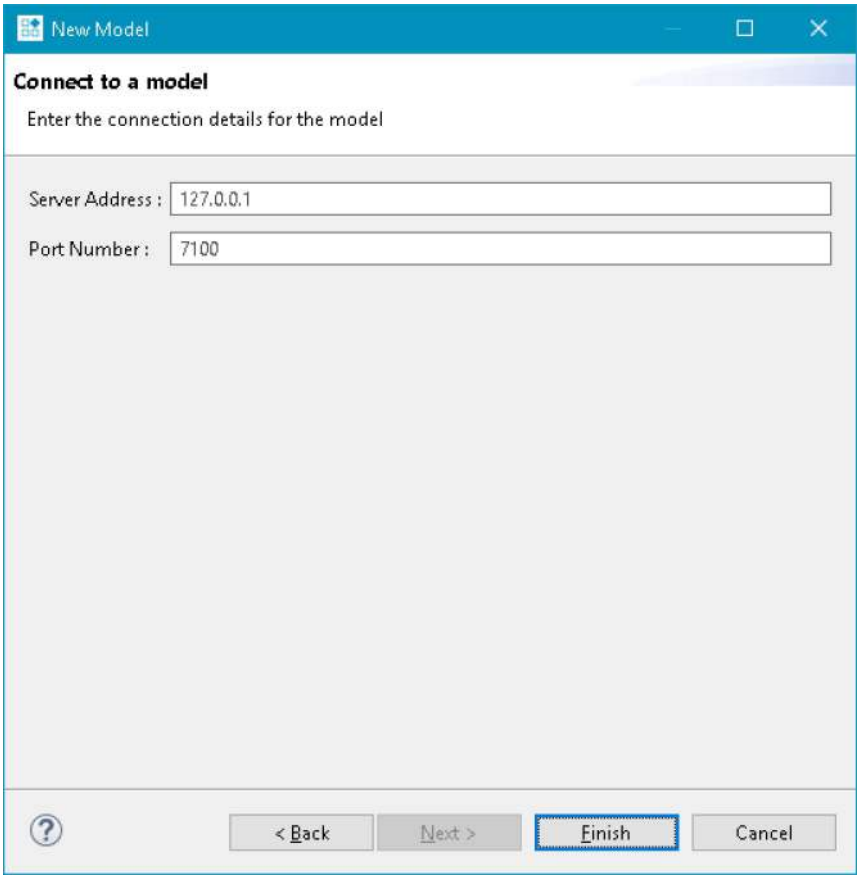
To connect to models running on the local host, you must first launch the model with the `--iris-server` switch before connecting to it. To connect to models running on a remote host, you must first launch the model with the `--iris-server --iris-allow-remote` switches.

---

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 11-44: Connect to model running on either local or remote host



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 necessary, 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

Create a new model configuration

11.11 Imported µVision project limitations

If you want to import µVision® projects into Arm® Development Studio, there are some limitations to be aware of.

The limitations are as follows:

- µVision project settings which affect target debug are not migrated to Arm Development Studio debug configurations, but are limited to the project build.
- µVision projects can have multiple project targets with variations in the Run-Time Environment (RTE) setting. In Arm Development Studio, a project:
  - Is limited to exactly one RTE configuration.
  - Does not support Eclipse's C/C++ Development Tooling (CDT) concept of project 'configurations'.

  Therefore, each µVision project target is imported as an individual Arm Development Studio project with its own copy of the project files.
- When you convert and import a µVision project, a copy of the project files are created and stored in your workspace directory.
- When you are preparing a µVision project for import into Arm Development Studio, you must ensure that all the files and folders that are specified in the project, are either in the same folder as the project file, or are in a subdirectory structure. If there are any files that are outside of the project folder, you must copy these into the project folder, and then manually resolve any relative dependencies.
- µVision Multi-Project-Workspace files (*.uvmpw) are not supported. Instead, you must import the projects included in the workspace individually, and set up project interdependencies manually.
- You cannot directly import µVision Multi-Project-Workspace (*.uvmpw) into Arm Development Studio. To use projects contained in .uvmpw files, you must import each project individually and manually configure their dependencies.
- You can only import µVision projects that specify fixed compiler versions. These compiler toolchains must also be installed in Arm Development Studio. This is because, in Arm Development Studio, the compiler version is configured per project target.
- User commands in µVision projects are not converted into the corresponding Arm Development Studio **Build Steps**. To check or edit the converted **Build Steps**, right-click the project, and select **Properties > C/C++ Build > Settings > Build Steps**.
- You must translate µVision Key Sequence for Tool Parameters to their corresponding variables in Arm Development Studio.

- The **ElfDwT** utility is not included in the Arm Development Studio installation. You must manually set up Signature Creator for NXP Cortex-M Devices (**ElfDwT**) as an Arm Development Studio post-build step. To set up a post-build step, right-click the project and select **Properties > C/C++ Build > Settings > Build Steps**.
- The fcarm utility is not included in the Arm Development Studio installation.
- In µVision source groups, software components and individual files can have specific assignments to memory regions which are evaluated when the tools generate the linker script. This feature is not available in Arm Development Studio, so you must manually edit the linker script file.

11.12 Other differences between DS-5 and Arm Development Studio

There are other small differences between DS-5 and Arm® Development Studio.

The differences are:

- The Linaro GCC 4.9-2014.04 [arm-linux-gnueabihf] compiler is not provided with Arm Development Studio. To use GCC (Linux or bare-metal) in Arm Development Studio, you can download the version you require from Linaro or Arm Developer, then add it as a toolchain (see Register a compiler toolchain)
- In Arm Development Studio, the IDE executable in the `bin` directory of the installation is named **armds_ide**. In DS-5, the executable is named **eclipse**.
- In Arm Development Studio, the default **Run control** option is **connect only** in the **Edit Configuration** view's **Debugger** tab. In DS-5, the default **Run control** option is **Debug from symbol** set to **main**.
- In Arm Development Studio, the command-line debugger is named **armdbg**. In DS-5, the command-line debugger is named **debugger**.
- In Arm Development Studio, the **Platform Configuration Editor (PCE)** is integrated into the new **Connection** wizard (see Connect to new or custom hardware).
- In Arm Development Studio, you must select **Properties > C/C++ Build > Environment > Append variables to native environment** for every project. If you select **Replace native environment with specified one**, the project might not build successfully.

Appendix A Terminology and Shortcuts

Supplementary information for new users of Arm® Development Studio.

A.1 Terminology

Arm® Development Studio documentation uses a range of terms. These are listed below.

Device
A component on a target that contains the application that you want to debug.

Dialog box
A small page that contains tabs, panels, and editable fields which prompt you to enter information.

Editor
A view that enables you to view and modify the content of a file, for example source files. The tabs in the editor area show files that are currently open for editing.

Flash Program
A term used to describe the storing of data on a flash device.

IDE
The Integrated Development Environment. A window that contains perspectives, menus, and toolbars. This is the main development environment where you can manage individual projects, associated sub-folders, and source files. Each window is linked to one workspace.

Panel
A small area in a dialog box or tab to group editable fields.

Perspective
Perspectives define the layout of your selected views and editors in Eclipse. They also have their own associated menus and toolbars.

Project
A group of related files and folders in Eclipse.

Resource
A generic term used to describe a project, file, folder, or a combination of these.

Send To
A term used to describe sending a file to a target.

Tab
A small overlay page that contains panels and editable fields within a dialog box to group related information. Clicking on a tab brings it to the top.

Target
A development platform on a printed circuit board or a software model that emulates the expected behavior of Arm hardware.

View

Views provide related information, for a specific function, corresponding to the active file in the editor. They also have their own associated menus and toolbars.

Wizard

A group of dialog boxes to guide you through common tasks. For example, creating new files and projects.

Workspace

An area on your file system used to store files and folders related to your projects.

A.2 Keyboard shortcuts

A list of the most common keyboard shortcuts available for use with Arm® Development Studio.

F3

Click an assembly instruction and press F3 to see help information about the instruction.

F10

Press F10 to access the main menu. You can then navigate the main menu using the arrow keys.

Alt+F4

Exit Arm Development Studio.

Alt+Left arrow

Go back in navigation history.

Alt+Right arrow

Go forward in navigation history.

Ctrl+Semicolon

In the Arm assembler editor, add comment markers to a selected block of code in the active file.

Ctrl+Home

Move the editor focus to the beginning of the code.

Ctrl+End

Move the editor focus to the end of the code.

Ctrl+B

Build all projects in the workspace that have changed since the last build.

Ctrl+F

Open the Find or Find/Replace dialog box to search through the code in the active editor. Some editors are read-only and therefore disable this functionality.

Ctrl+F4

Close the active file in the editor view.

Ctrl+F6
Cycle through open files in the editor view.

Ctrl+F7
Cycle through available views.

Ctrl+F8
Cycle through available perspectives.

Ctrl+F10
Use with the arrow keys to access the drop-down menu.

Ctrl+L
Move to a specified line in the active file.

Ctrl+Q
Move to the last edited position in the active file.

Ctrl+Space
Auto-complete selected functions in editors.

Shift+F10
Use with the arrow keys to access the context menu.

Ctrl+Shift+F
Activate the code style settings in the **Preferences** dialog box and apply them to the active file.

Ctrl+Shift+L
Open a small page with a list of all keyboard shortcuts.

Ctrl+Shift+R
Open the **Open resource** dialog box.

Ctrl+Shift+T
Open the **Open Type** dialog box.

Ctrl+Shift+/
In the C/C++ editor, add comment markers to the start and end of a selected block of code in the active file.