Arm® Development Studio Morello Edition

Version 2020.1M0

Getting Started Guide

arm

102233_2020.1M0_00_en

Arm® Development Studio Morello Edition

Getting Started Guide

Release Information

Document History

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

Non-Confidential Proprietary Notice

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

developer.arm.com

Contents

Arm® Development Studio Morello Edition Getting Started Guide

List of Figures
Arm® Development Studio Morello Edition Getting Started Guide

List of Tables
Arm® Development Studio Morello Edition Getting Started Guide

Preface

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

It contains the following:

About this book

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

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction to Arm Development Studio Morello Edition

Arm Development Studio Morello Edition 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.

Chapter 2 Install and configure Arm® Development Studio

Arm Development Studio Morello Edition is available for Linux operating systems. This chapter covers installation requirements, the installation process, and various configuration settings applicable for your installation of Arm Development Studio Morello Edition.

Chapter 3 Arm® Debugger concepts

Introduces Arm Debugger and some important debugger concepts.

Chapter 4 Arm Development Studio Morello Edition IDE

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

Chapter 5 Arm Development Studio Morello Edition projects and examples

Explains how to create a new project and configure it. It also describes how to work with examples provided with Arm Development Studio Morello Edition.

Chapter 6 Writing code

The following topics describe how to use the editors when developing a project for an Arm target.

Chapter 7 Debugging code

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

Chapter 8 Tutorials

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

Chapter 9 Troubleshoot Arm® Development Studio Morello Edition

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

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

italic

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

bold

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

`monospace`

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

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

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

`monospace bold`
: Denotes language keywords when used outside example code.

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

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

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title *Arm Development Studio Morello Edition Getting Started Guide*.
- The number 102233_2020.1M0_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Other information

- *Arm® Developer*.
- *Arm® Documentation*.
- *Arm Morello Program*.
- *Arm Morello Program Community*.
- *Arm® Glossary*.

Chapter 1
Introduction to Arm Development Studio Morello Edition

Arm Development Studio Morello Edition 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.

Arm Development Studio Morello Edition includes these components:

Arm Development Studio IDE

An Eclipse-based IDE that allows you to manage your projects, configure your compilation and debug session settings, view visual representations of debug and trace sessions, and much more.

Arm Debugger

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

Morello examples

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

Arm Development Studio Morello Edition is to be used with the Morello Fixed Virtual Platform (FVP), the LLVM compiler with Morello support, and hardware platforms based on the Morello architecture. The Morello FVP and the LLVM compiler with Morello support is available separately from Arm Development Studio Morello Edition.

It contains the following sections:

1.1 Arm® Debugger

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

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.

To debug your code, you can also use the command-line version of Arm Debugger interactively or through automated scripts.

Related information

Debug control view

1.2 Morello Fixed Virtual Platform 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 Morello Edition supports the Morello FVP, that includes models of Rainier cores.

Related information

About the Component Architecture Debug Interface (CADI)
Arm Morello Program

1.3 Debug probes

Arm Development Studio Morello Edition supports various debug adapters and connections.

Debug adapters

Debug adapters vary in complexity and capability. When you use them with Arm Development Studio Morello Edition, 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
- Keil® ULINK™2
- Keil ULINKpro
- Keil ULINKpro D

Debug hardware configuration

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

Arm Development Studio Morello Edition 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.

Chapter 2
Install and configure Arm® Development Studio

Arm Development Studio Morello Edition is available for Linux operating systems. This chapter covers installation requirements, the installation process, and various configuration settings applicable for your installation of Arm Development Studio Morello Edition.

It contains the following sections:

2.1 Hardware and host platform requirements

For the best experience with Arm Development Studio Morello Edition, your hardware and host platform must meet the minimum requirements.

Hardware requirements

To install and use Arm Development Studio Morello Edition, 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.

Host platform requirements

Arm Development Studio Morello Edition supports the following host platforms:

- Red Hat Enterprise Linux 7 Workstation
- Ubuntu Desktop Edition 16.04 LTS
- Ubuntu Desktop Edition 18.04 LTS

Note

Arm Development Studio Morello Edition only supports 64-bit host platforms.

2.2 Debug system requirements

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

Bare-metal requirements

You require a debug unit to connect bare-metal hardware targets to Arm Development Studio Morello Edition. For a list of supported debug units, see *Debug Probes* on page 1-16.

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`.
- For ULINKpro and ULINKpro D, Arm Development Studio Morello Edition manages the firmware installation.

2.3 Installing on Linux

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

Note

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

Prerequisites

Download the Linux installation package from the *Arm Developer website*.

Procedure

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

  Note

  During the installation, Arm Development Studio Morello Edition 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 Morello Edition.

  Note

  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.

Note

Use `suite_exec` to configure the environment variables correctly for Arm Development Studio Morello Edition. 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.

2.4 Data collection in Arm® Development Studio

Arm periodically collects anonymous information about the usage of our products in order to understand and analyse what components or features you are using with the goal to improve our products and your experience with them. Product usage analytics contain information such as system information, settings, and usage of specific features of the product. You can enable or disable the feature in the product settings. Product usage analytics do not include any personal information.

Host information includes:

- Operating system name, version, and locale.
- Number of CPUs.
- Amount of physical memory.
- Screen resolution.
- Processor and GPU type.

Feature tracking information includes:

- Events - Records that a feature, described by its category and name, was used. No further information is collected.
- Numerical data - Tracks information that is related to time or size, expressed as a number. For example, how long an operation took or the size of a file that is produced by an operation.
- Textual data - Tracks static information. For example, the name of an Arm processor.

Disabling data collection

To disable data collection, from the main menu, select **Window** > **Preferences** > **Arm DS** > **General**, and deselect the **Allow collection of anonymous analytics data** option.

On disabling data collection, Arm Development Studio sends a final message to Arm to record that analytics capture is disabled. This final message is only used for reporting opt-out statistics, and no personal or system information is collected.

Related references

2.5 Arm® Development Studio IDE analytics data points on page 2-22

2.6 Arm® Debugger analytics data points on page 2-23

2.5 Arm® Development Studio IDE analytics data points

We periodically collect anonymous information about your use of the Arm Development Studio IDE. This information allows us to understand and analyze what features you are using, with the goal to improve our product and your experience with it.

Table 2-1 Arm DS IDE analytics data points

Category	Name	Description		Since
Utilities				
	Target Configuration Editor	Tracked	Use of the Target Configuration Editor	2019.0
		Reported	Percentage of users using the Target Configuration Editor	
		Data Type	Event	
		Send Policy	Once a day	
		Trigger Points	A file opened by the Target Configuration Editor	
Projects				
	IDE build	Tracked	Use of Eclipse/CDT project build	2019.0
		Reported	Percentage of users building projects in Eclipse	
		Data Type	Event	
		Send Policy	Once a day	
		Trigger Points	On project build	
	CMSIS target software pack	Tracked	Use of target software supplied as software packs	2019.0
		Reported	Percentage of users using each CMSIS target software pack	
		Data Type	Text	
		Send Policy	Once a day per unique value	
		Trigger Points	On project build	
Toolchains				
	Imported toolchain	Tracked	Imported toolchain identifier, family and version, used to build a project in the IDE	2019.0
		Reported	Percentage of users building with each imported toolchain	
		Data Type	Text	
		Send Policy	Once a day per unique value	
		Trigger Points	On project build	

2.6 Arm® Debugger analytics data points

We periodically collect anonymous information about your use of Arm Debugger. This information allows us to understand and analyze what features you are using, with the goal to improve our product and your experience with it.

Table 2-2 Arm Debugger analytics data points

Category	Name	Description		Since
Feature				
	Graphical sessions	Tracked	When a graphical debug session is initiated, not necessarily successful.	2019.0
		Reported	Percentage of users using the graphical user interface.	
		Data Type	Event	
		Send Policy	Once a day.	
		Trigger Points	On debug connection with the IDE.	
	Commandline sessions	Tracked	When commandline debug sessions are initiated, not necessarily successful.	2019.0
		Reported	Percentage of users using the commandline debugger.	
		Data Type	Event	
		Send Policy	Once a day.	
		Trigger Points	On debug connection with the CLI debugger.	
	Trace	Tracked	This can for instance record usage of the Trace view in the IDE or usage of the *trace dump* command.	2019.0
		Reported	Percentage of users using trace-related features.	
		Data Type	Event	
		Send Policy	Once a day.	
		Trigger Points	• A trace source is processed/decoded by the Trace view. • Searching for trace events in the Trace view. • The *Export Trace Report* action in the Trace view. • The start, stop, and dump actions in the Trace Control view. • A trace source is processed/decoded by the Events view. • **Any of the following debugger commands are run:** — *trace start* — *trace stop* — *trace report* — *trace dump*	
	Python scripting	Tracked	Use of Python scripts, excluding scripting in DTSL.	2019.0
		Reported	Percentage of users using Python scripting.	
		Data Type	Event	
		Send Policy	Once a day	
		Trigger Points	• A *source* command is executed for a file with a *.py* extension. • A *usecase* command is executed. • A breakpoint, with the script property set to a file with a *.py* extension, is hit.	

Table 2-2 Arm Debugger analytics data points (continued)

Category	Name	Description		Since
	OS Awareness	Tracked	Name of OS awareness configured. **Note** If the OS awareness is not supplied by Arm, the name is obfuscated using a one-way hashing algorithm. Linux application debug is not considered an OS awareness, but a target type, tracked with another another analytics data point.	2019.0
		Reported	Percentage of users for each OS awareness.	
		Data Type	Text	
		Send Policy	Once a day per unique value.	
		Trigger Points	On debug connection.	
	CPU cache	Tracked	Use of cache-related features.	2019.0
		Reported	Percentage of users using cache-related features.	
		Data Type	Event	
		Send Policy	Once a day.	
		Trigger Points	• Cache data is displayed in the Cache Data view. • Cache data is displayed in the Memory view. • The *cache list* or *cache print* debugger commands are run.	
	Types	Tracked	Type of debug target. For example, *Hardware*, *CADI Model*, *Linux Application*, and so on.	2019.0
		Reported	Percentage of users for each target type.	
		Data Type	Text	
		Send Policy	Once a day per unique value.	
		Trigger Points	On debug connection.	
	CPU architectures	Tracked	Name of the major CPU architecture version and profile connected to, for example, *Armv6-M*.	2019.0
		Reported	Percentage of users for each target type.	
		Data Type	Text	
		Send Policy	Once a day per unique value.	
		Trigger Points	On debug connection.	
	Number of cores	Tracked	Number of cores connected to in a single debug session.	2019.0
		Reported	Percentage of users for number of cores.	
		Data Type	Number	
		Send Policy	Once a day per unique value.	
		Trigger Points	On debug connection.	

Table 2-2 Arm Debugger analytics data points (continued)

Category	Name	Description		Since
	Probes	Tracked	Name of the debug probe. **Note** If support for this probe is not supplied by Arm, the name is obfuscated using a one-way hashing algorithm.	2019.0
		Reported	Percentage of users for each probe.	
		Data Type	Text	
		Send Policy	Once a day per unique value.	
		Trigger Points	On debug connection.	
	Platform manufacturer	Tracked	Manufacturer of the platform in a debug session **Note** If support for this probe is not supplied by Arm, the name is obfuscated using a one-way hashing algorithm.	2020.0
		Reported	Percentage of users for each target type.	
		Data Type	Text	
		Send Policy	Once a day per unique value.	
		Trigger Points	On debug connection.	
	Platform name	Tracked	Name of the platform in a debug session **Note** If support for this probe is not supplied by Arm, the name is obfuscated using a one-way hashing algorithm.	2020.0
		Reported	Percentage of users for each target type.	
		Data Type	Text	
		Send Policy	Once a day per unique value.	
		Trigger Points	On debug connection.	

2.7 Language settings

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

Procedure

- Launch the Arm Development Studio Morello Edition 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
    ```

2.8 Add a compiler toolchain

If you want to build projects using a toolchain in Arm Development Studio Morello Edition, you must first add the toolchain you want to use. You can add toolchains:

- Using the *Preferences dialog box* on page 2-27 in Arm Development Studio Morello Edition.
- Using the **add_toolchain** *utility* on page 2-29 available in Arm Development Studio Morello Edition.

You might want to add a compiler toolchain if:

- You want to build your application from within the Arm Development Studio Morello Edition IDE.
- You upgrade your version of Arm Development Studio Morello Edition 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 Morello Edition.

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

Note

Morello architecture related features are only supported in LLVM version 11.0.0 and above.

This section contains the following subsections:

- *2.8.1 Add a compiler toolchain using the Arm Development Studio Morello Edition IDE* on page 2-27.
- *2.8.2 Add a compiler toolchain using the command line* on page 2-29.
- *2.8.3 Configure a compiler toolchain using the command prompt* on page 2-30.

2.8.1 Add a compiler toolchain using the Arm Development Studio Morello Edition IDE

You can add compiler toolchains using the **Preferences** dialog box in Arm Development Studio Morello Edition.

Procedure

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

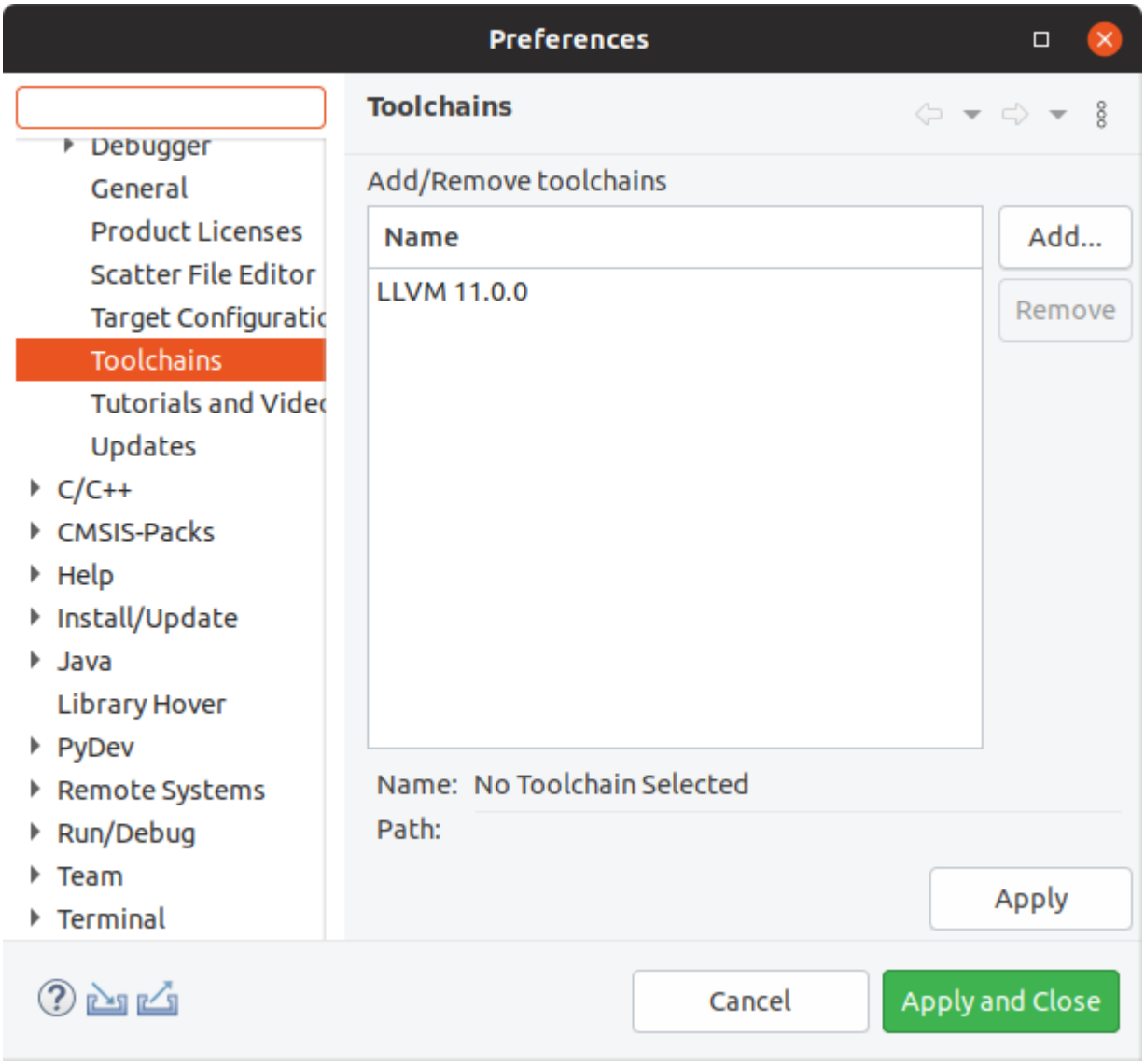


Figure 2-1 Toolchains Preferences dialog box

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 add the toolchain. Alternatively, click **Next** to manually enter or change the toolchain properties, and then click **Finish**.

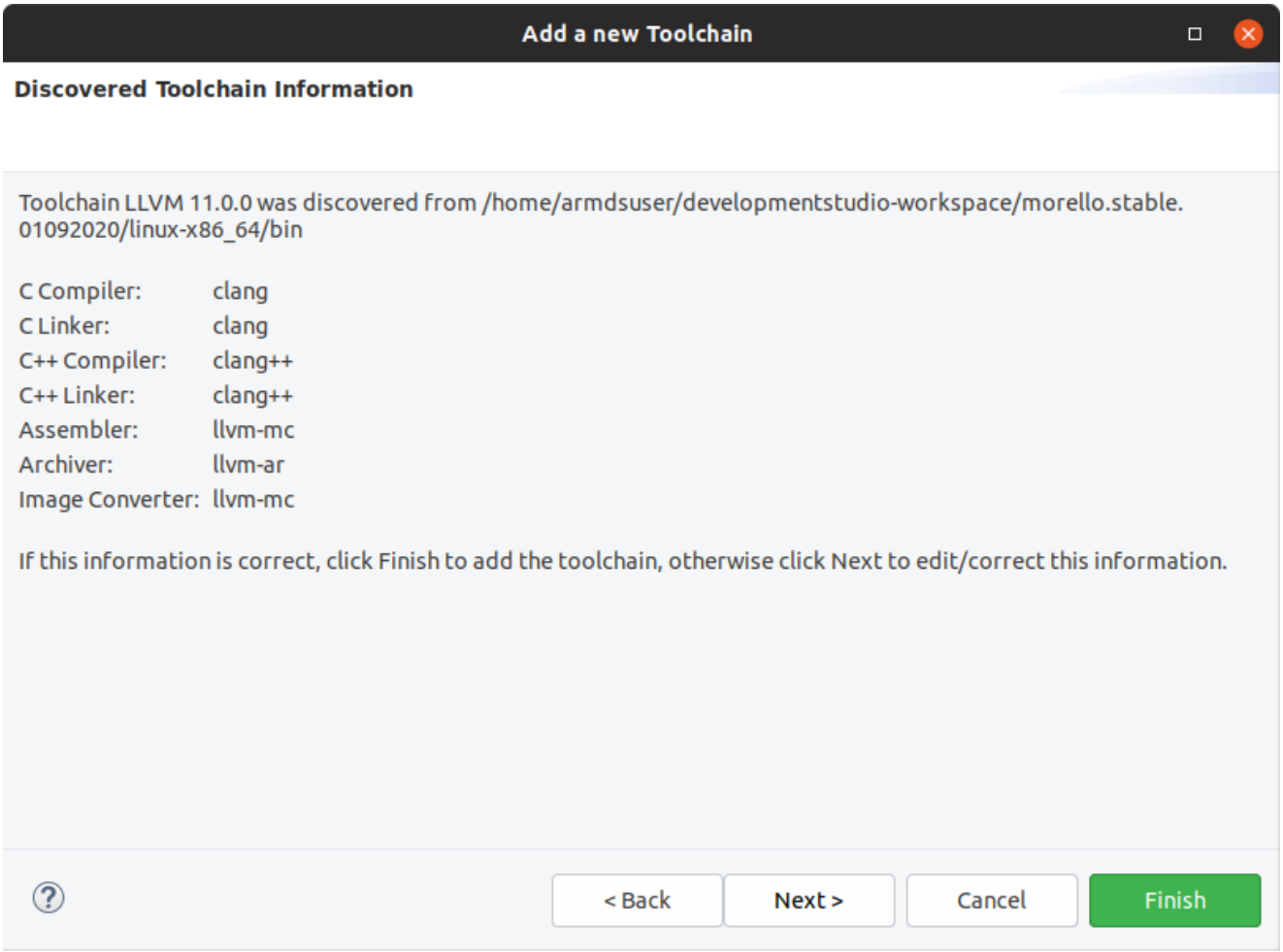


Figure 2-2 Properties for the new toolchain

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 is identical to a toolchain that Arm DS already knows about.

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

The new toolchain is now added in Arm Development Studio Morello Edition.

2.8.2 Add a compiler toolchain using the command line

Use the `add_toolchain` utility from the command prompt to register a new toolchain.

Procedure

1. At the command prompt, 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_toolchain` utility as a non-interactive tool, add the `--non-interactive` option to the command.

For example, `add_toolchain <path_to_toolchain> --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 Morello Edition. You must do this before you can use the toolchain in the Arm DS IDE.

When you create a new project, the new toolchain is available in the list of toolchains.

Next Steps

To change the toolchain in an existing project to a newly registered toolchain, right-click the project and select **Properties** > **C/C++ Build** > **Tool Chain Editor**

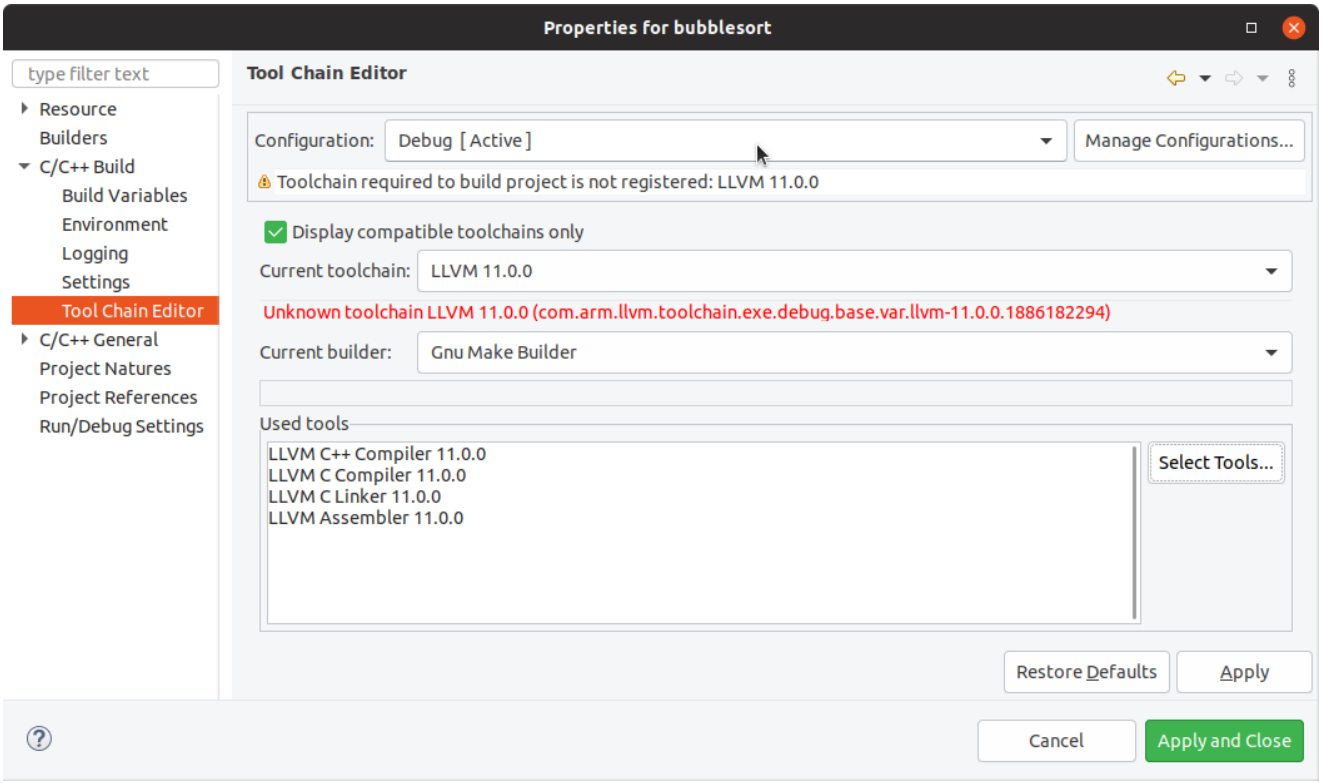


Figure 2-3 Changing the toolchain for a project

2.8.3 Configure a compiler toolchain using the 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.

Note

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

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>`

Tip

Run `suite_exec` with no arguments for the list of available toolchains.

Note

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

Related tasks

2.8.1 Add a compiler toolchain using the Arm Development Studio Morello Edition IDE on page 2-27
2.8.2 Add a compiler toolchain using the command line on page 2-29
2.8.3 Configure a compiler toolchain using the command prompt on page 2-30

2.9 Specify plug-in install location

By default, Arm Development Studio Morello Edition 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.

You override the default Arm Development Studio Morello Edition 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.

Prerequisites

- Access to your Arm Development Studio Morello Edition installation.

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

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

2.10 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.

Note

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

F9

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

Related information

Commands view

Debug Control view

Chapter 3
Arm® Debugger concepts

Introduces Arm Debugger and some important debugger concepts.

It contains the following sections:

3.1 Overview: Arm® Debugger and important concepts

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

Arm Debugger supports:

- Loading images and symbols.
- Running images.
- Breakpoints and watchpoints.
- Source and instruction level stepping.
- Accessing variables and register values.
- Viewing the contents of memory.
- Navigating the call stack.
- Handling exceptions and Linux signals.
- Debugging bare-metal code.
- Debugging from the command-line.
- 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.

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.

3.2 Debugger concepts

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

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 within 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.

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 *Controlling target execution* 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 master ports. See *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*

———— **Note** ————

Arm Development Studio Morello Edition supports the first-generation 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 within 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 within an application at which it is defined.

Variables can have values that are relevant within:

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

SMP

A Symmetric Multi-Processing (SMP) system has multiple processors with the same architecture. See *Debugging SMP systems* on page 3-41 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.

3.3 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 TraceMacrocell*
- *STM: System Trace Macrocell*

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, stop all the cores in an SMP group simultaneously, or halt heterogeneous cores simultaneously to allow whole system debug at a particular point in time.

3.4 Overview: Debugging multi-core (SMP) 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) 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 require no special setup process. Arm Debugger includes predefined configurations. The nature of the connection determines how Arm Debugger behaves, for example stopping and starting all cores simultaneously in a SMP system.

This section contains the following subsection:

- *3.4.1 Debugging SMP systems* on page 3-41.

3.4.1 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.

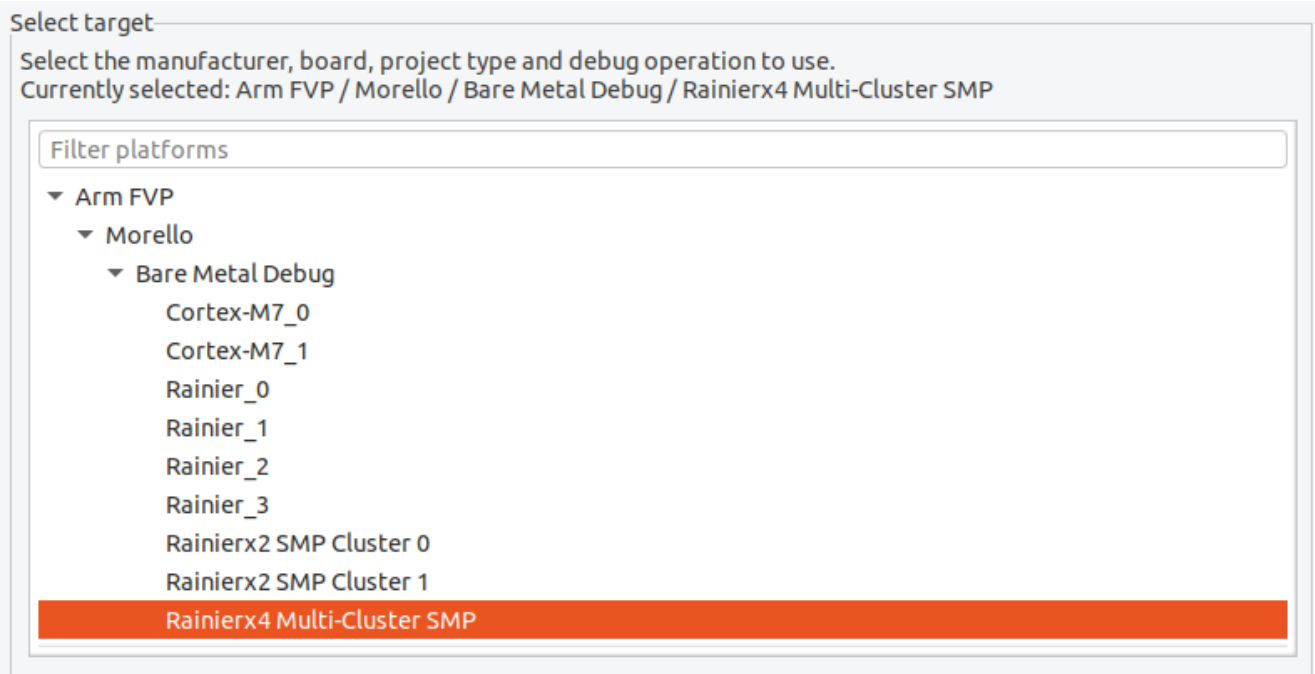


Figure 3-1 Arm Morello FVP SMP configuration

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.

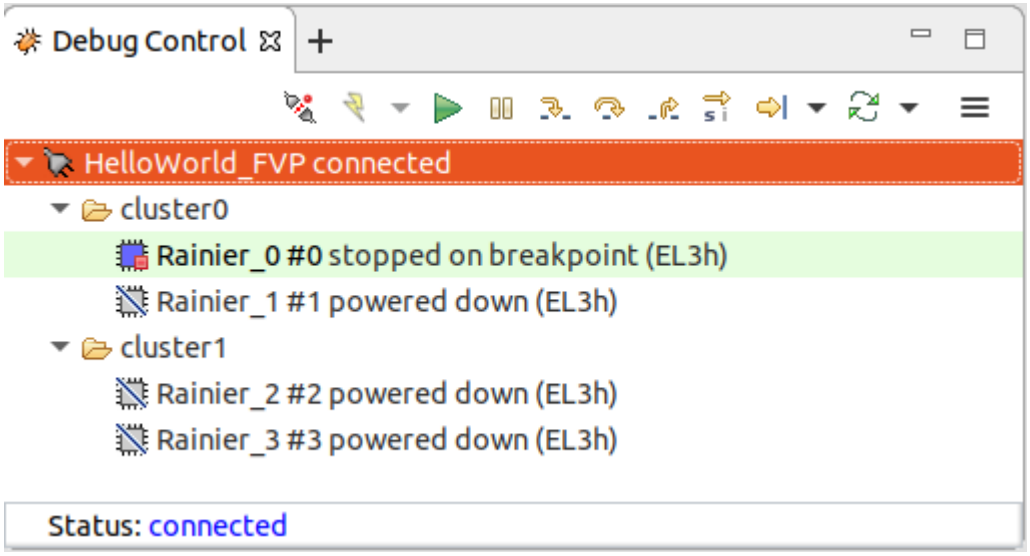


Figure 3-2 Core 0 stopped on step i command

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.

Related concepts

3.4.1 Debugging SMP systems on page 3-41

Chapter 4
Arm Development Studio Morello Edition IDE

The Arm Development Studio Morello Edition 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 Morello Edition uses the **Development Studio** perspective by default. To switch perspectives, from the main menu, select **Window** > **Perspective** > **Open Perspective**.

It contains the following sections:

4.1 Integrated Development Environment (IDE) Overview

The IDE contains a collection of views that are associated with a specific perspective.

Arm Development Studio Morello Edition uses the **Development Studio** perspective as default.

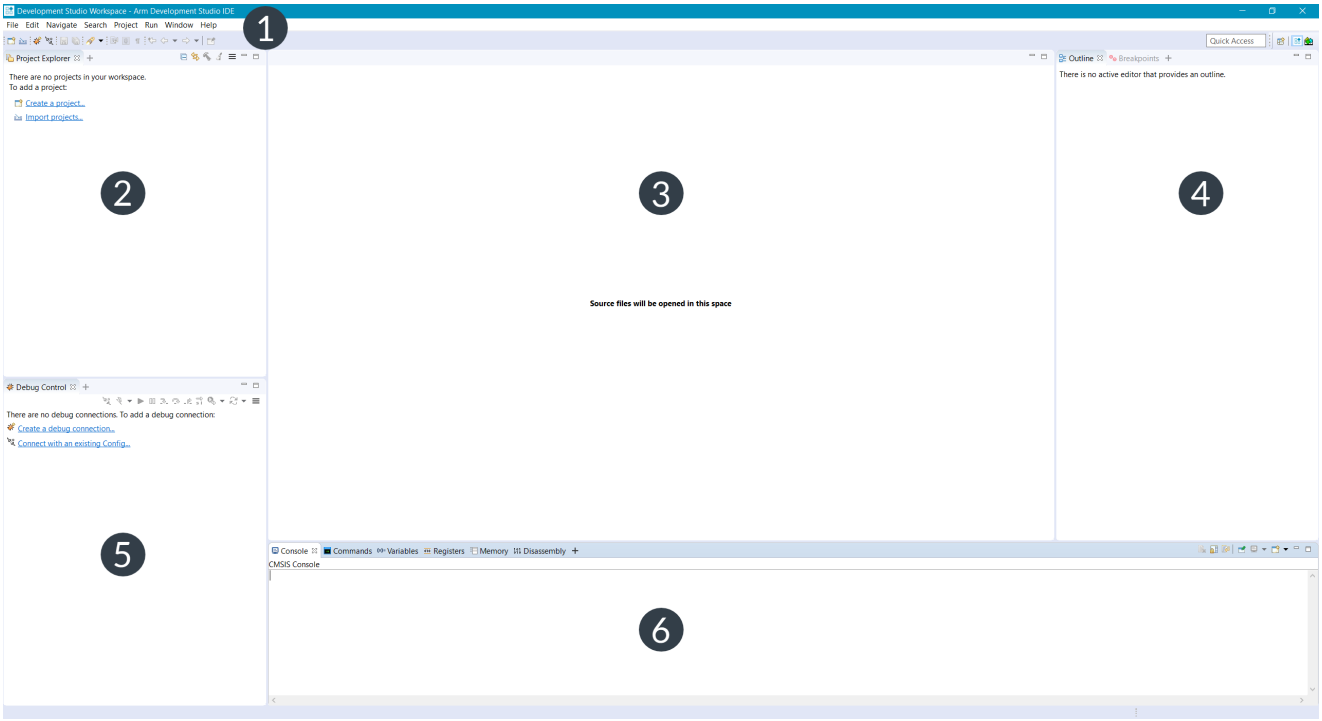


Figure 4-1 IDE in the Development Studio perspective.

1. Menus and Toolbars - The main menu and toolbar are located at the top of the IDE window. Other toolbars associated with specific features are located at the top of each perspective or view.
2. Project Explorer - Use this view to create, build, and manage your projects.
3. Debug Control - Use this view to create and control debug connections.
4. Editor window - Use this view to view and modify the content of your source code files. The tabs in the editor area show files that are currently open for editing.
5. Debug views - Views specific to Arm Debugger. Multiple views can be stacked in an area, creating tabs.
6. Area containing additional debug views. All areas are fully customizable to contain any desired view.

On exit, your settings save automatically. When you next open Arm Development Studio Morello Edition, 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.

4.2 Using the IDE

You can customise the Arm Development Studio Morello Edition IDE. It is possible to choose the views you can see by following the instructions in this section.

This section contains the following subsections:

4.2.1 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 Morello Edition launches for the first time, a default workspace is automatically created for you in `<user_home>/developmentstudio-workspace`.

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 Morello Edition. These resources might also cause a conflict later when you create and build projects.

Arm Development Studio Morello Edition automatically opens in the last used workspace.

Procedure

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

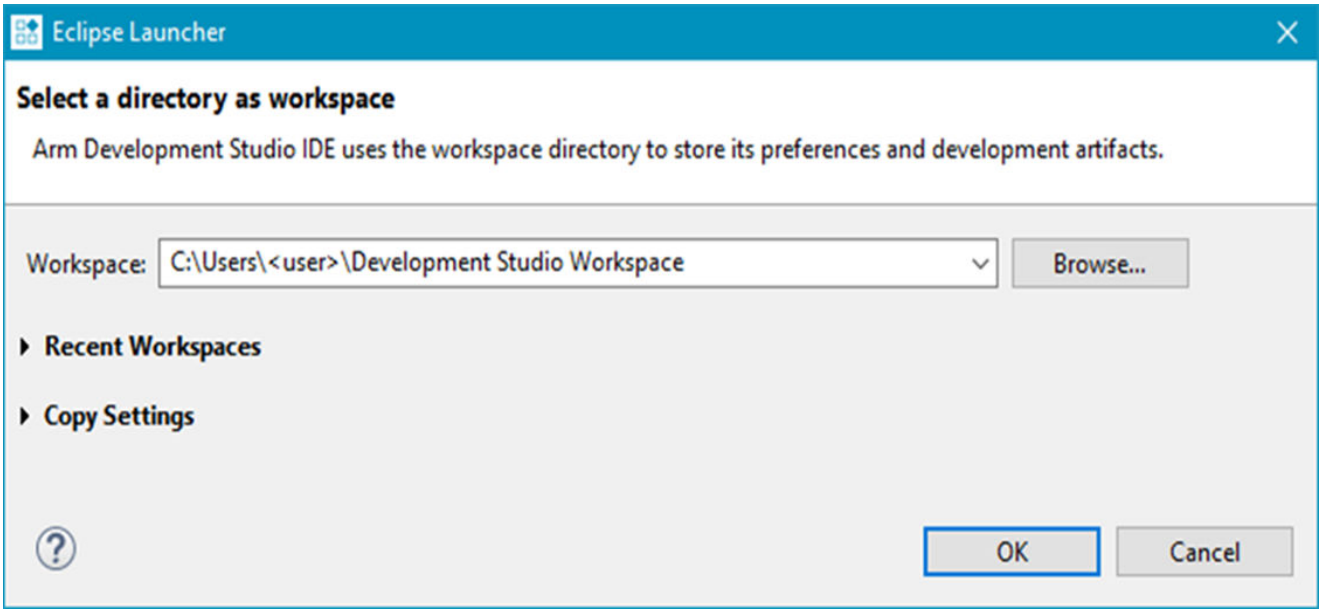


Figure 4-2 Workspace Launcher dialog box

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

Arm Development Studio Morello Edition relaunches in the new workspace.

4.2.2 Switching perspectives

Perspectives define the layout of your selected views and editors in the Arm Development Studio Morello Edition 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**.

Your selected perspective opens in the workspace.

Related information

Perspectives and views

4.2.3 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.

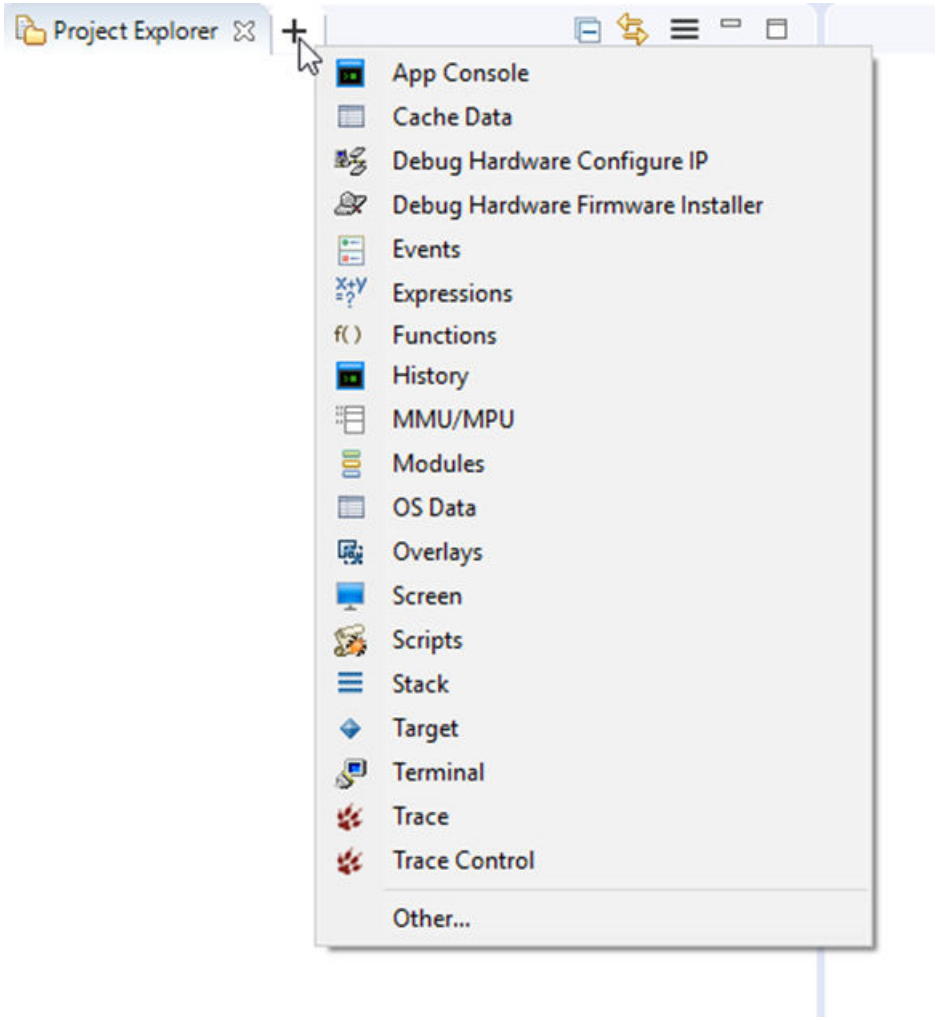


Figure 4-3 Adding a view in an area

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

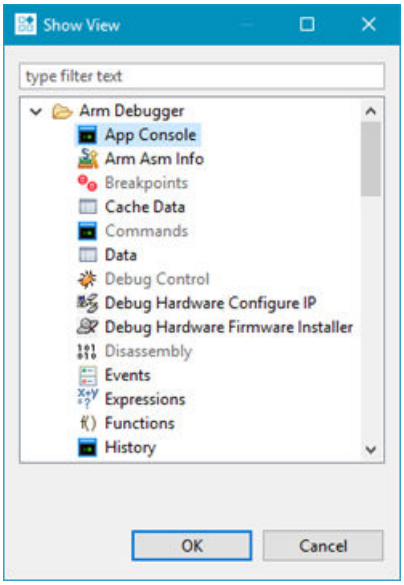


Figure 4-4 Adding a view in Arm Development Studio

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

The view opens in the selected area.

Related information

Perspectives and views

Related tasks

4.2.1 Changing the default workspace on page 4-47

4.2.2 Switching perspectives on page 4-47

4.2.3 Adding views on page 4-48

Chapter 5
Arm Development Studio Morello Edition projects and examples

Explains how to create a new project and configure it. It also describes how to work with examples provided with Arm Development Studio Morello Edition.

It contains the following sections:

5.1 Project types

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

Bare-metal Executable

Uses LLVM compiler with Morello support to build a bare-metal executable ELF image.

Bare-metal Static library

Uses LLVM compiler with Morello support to build a library of ELF object format members for a bare-metal project.

——— **Note** ———

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 LLVM compiler with Morello support 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.

———————————

5.2 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**.

   ———— **Note** ————

   C project

   This does not select a source language by default and leaves this decision up to the compiler. Both GCC and LLVM compiler with Morello support 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.

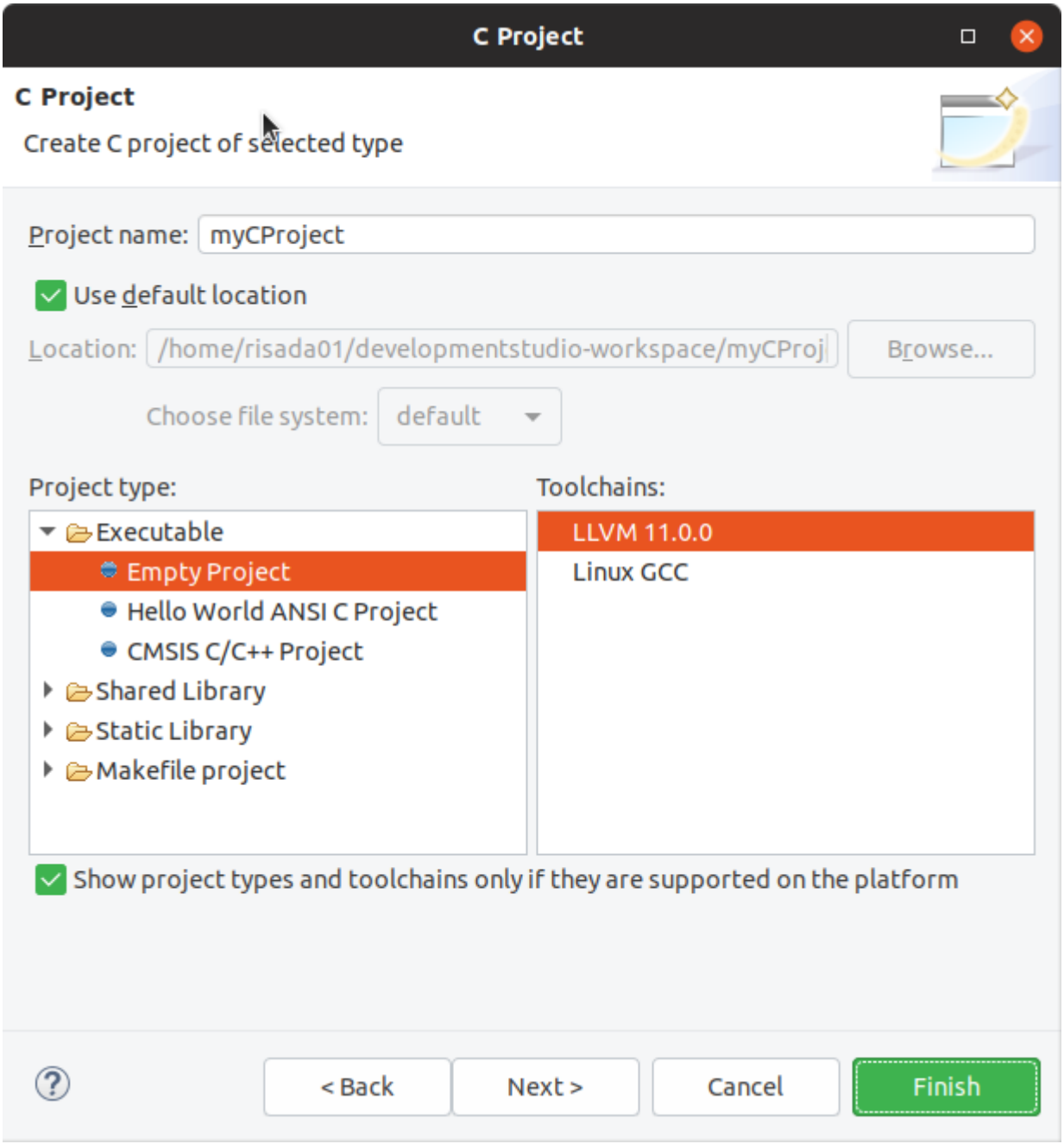


Figure 5-1 Creating a new C project

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

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

5.3 Create an empty Makefile project

Describes how to create an empty C or C++ Makefile project.

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.
   h. Click **Finish** to create your new project. The project is visible in the **Project Explorer** view.
2. Before you can build the project, you must point to the `Makefile` that contains the compilation tool settings:
   a. Right-click the project and then select **Properties** > **C/C++ Build**.
   b. 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 tasks

5.4 Create a new Makefile project with existing code on page 5-56

5.4 Create a new Makefile project with existing code

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

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.

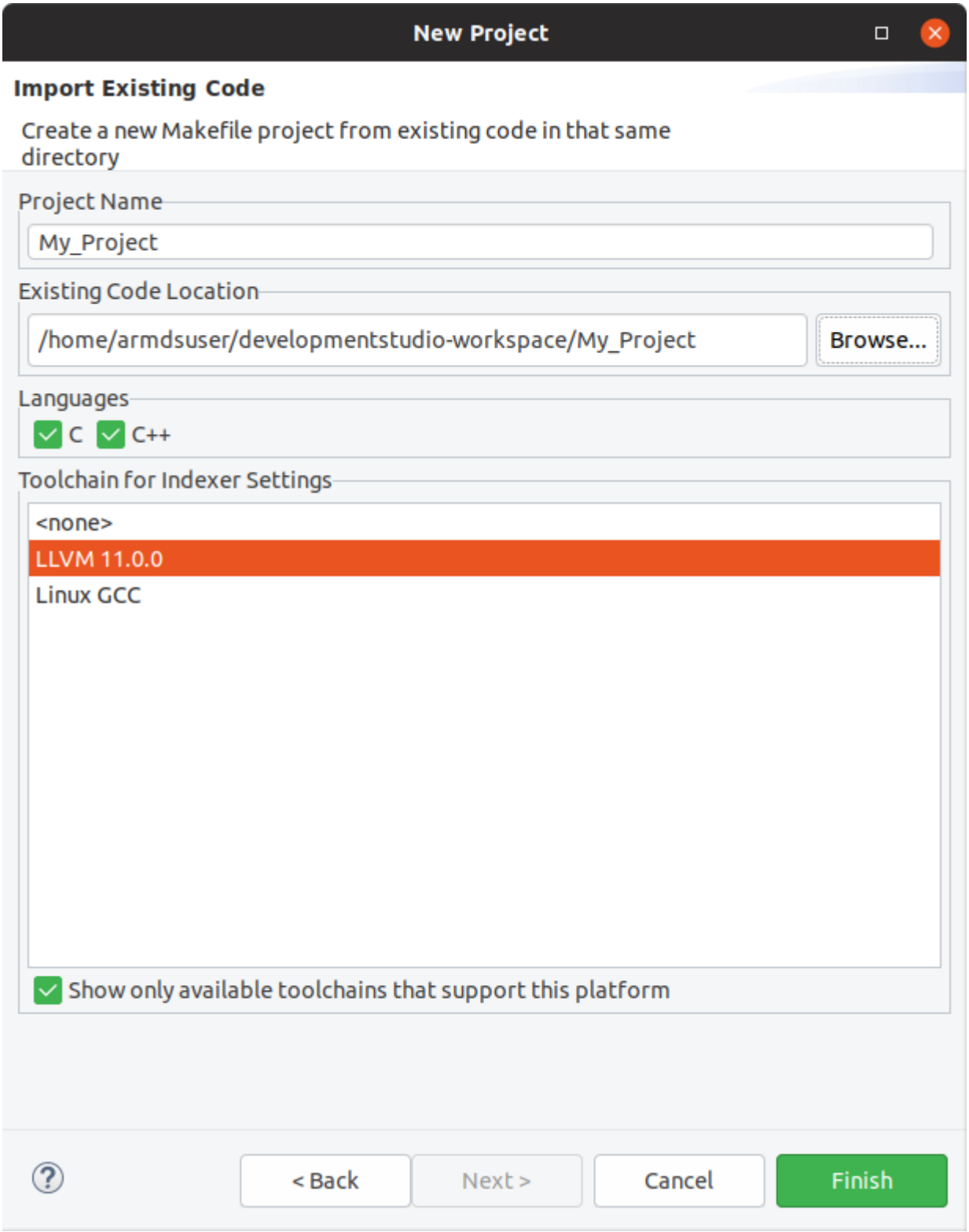


Figure 5-2 Creating a new Makefile project with existing code

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 tasks

5.3 Create an empty Makefile project on page 5-55

5.5 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.

   Note

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

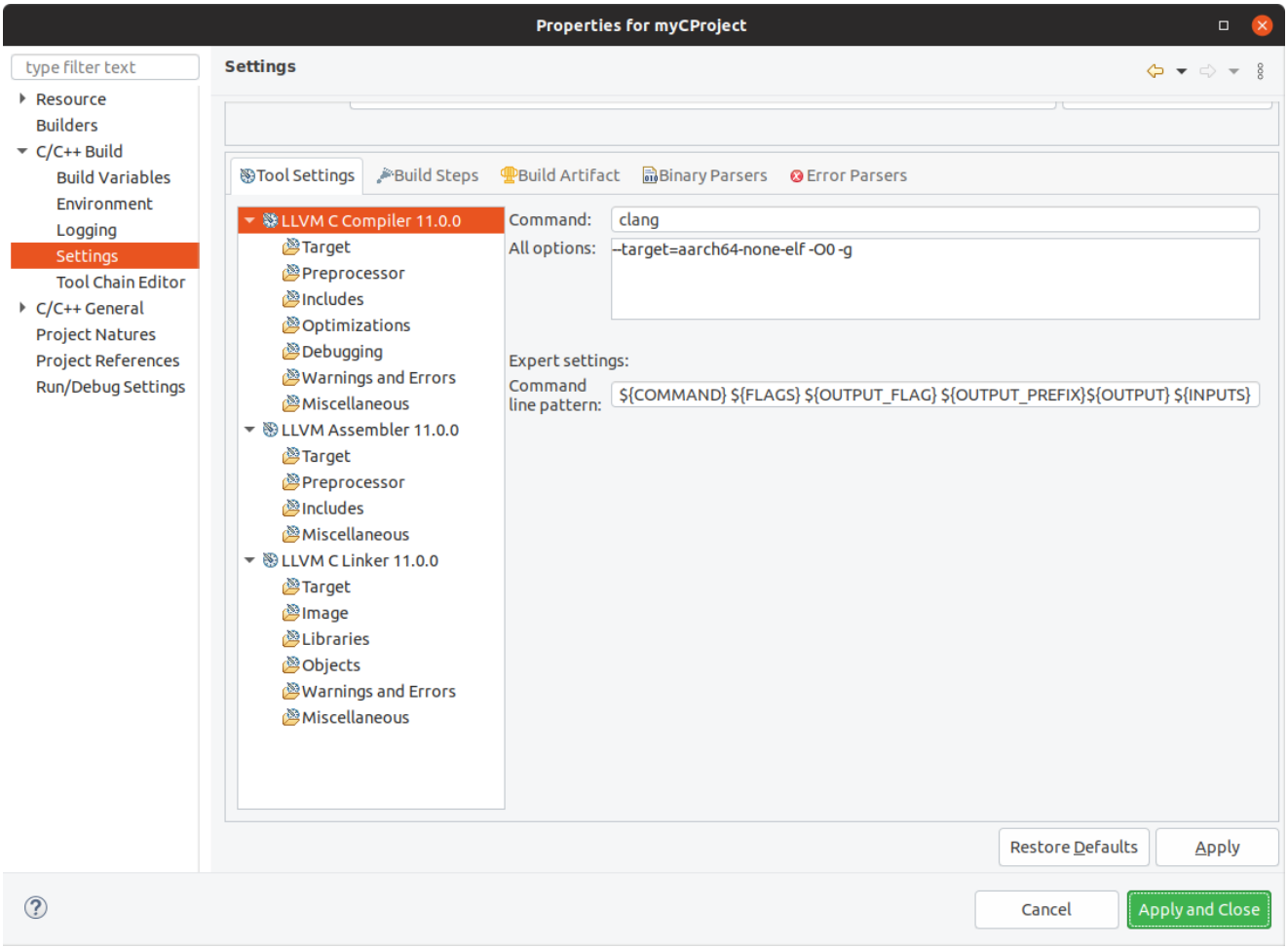


Figure 5-3 Typical build settings dialog box for a C project

5. Click **OK**.

The updated settings for your build configuration are saved.

5.6 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.

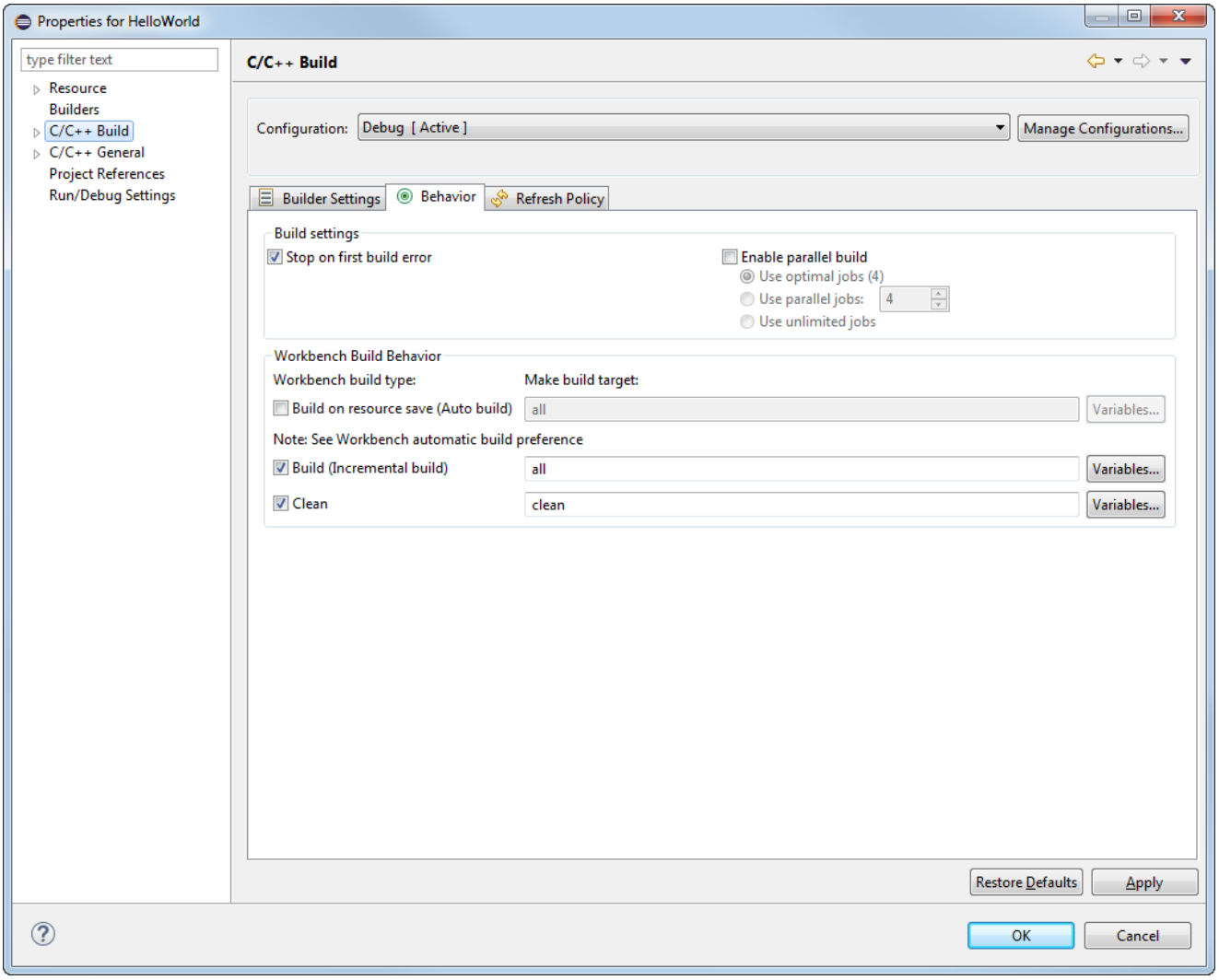


Figure 5-4 Workbench build behavior

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.

5.7 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**.

5.8 Adding a new source file to your project

There are several ways to add new source files to your project:

- You can drag and drop source files directly into a project, in the **Project Explorer** view of Arm Development Studio.
- You can create source files, or drag and drop files directly into the project folder, using the file system. To update the views in Arm Development Studio, click the relevant project in the **Project Explorer** view, and from the main menu select **File** > **Refresh**.
- Or, you can do the following:

Procedure

1. In the **Development Studio** perspective, right-click on the project and select **New** > **Source File** to display the **New Source File** dialog box.
   a. Alternatively, from the main menu, select **File** > **New** > **Source File**.

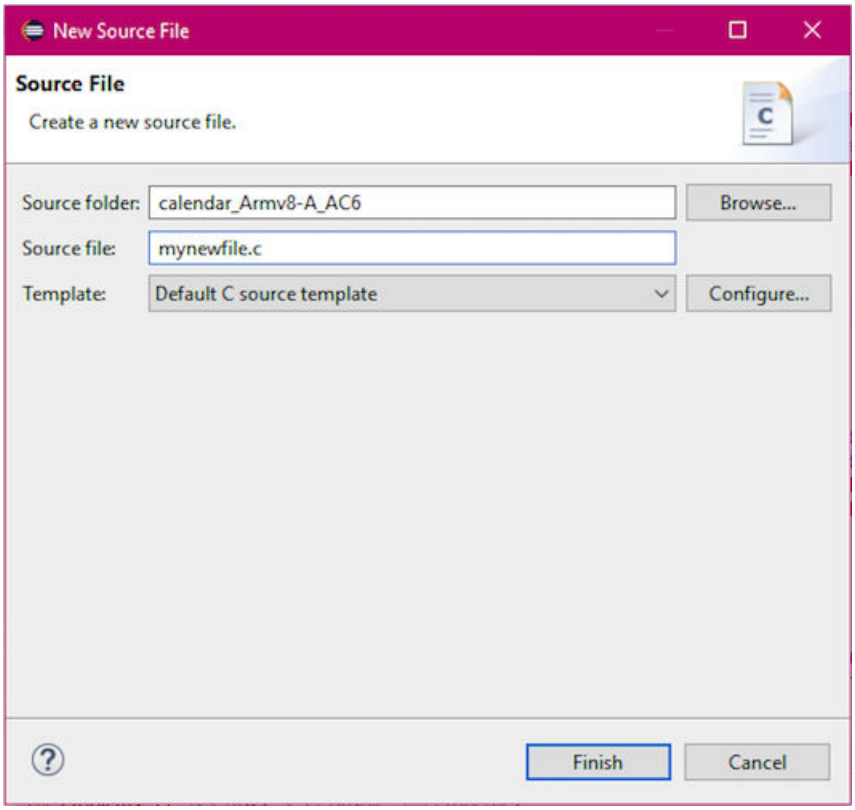


Figure 5-5 Adding a new source file to your project

2. The **Source folder** field tells you the project where the new source file is saved. If you want to save it to a different project, click **Browse…**, and select another project.
3. In the **Source file** field, enter a name for the new source file and include the file extension.
4. Select a source file template from the **Template** 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.

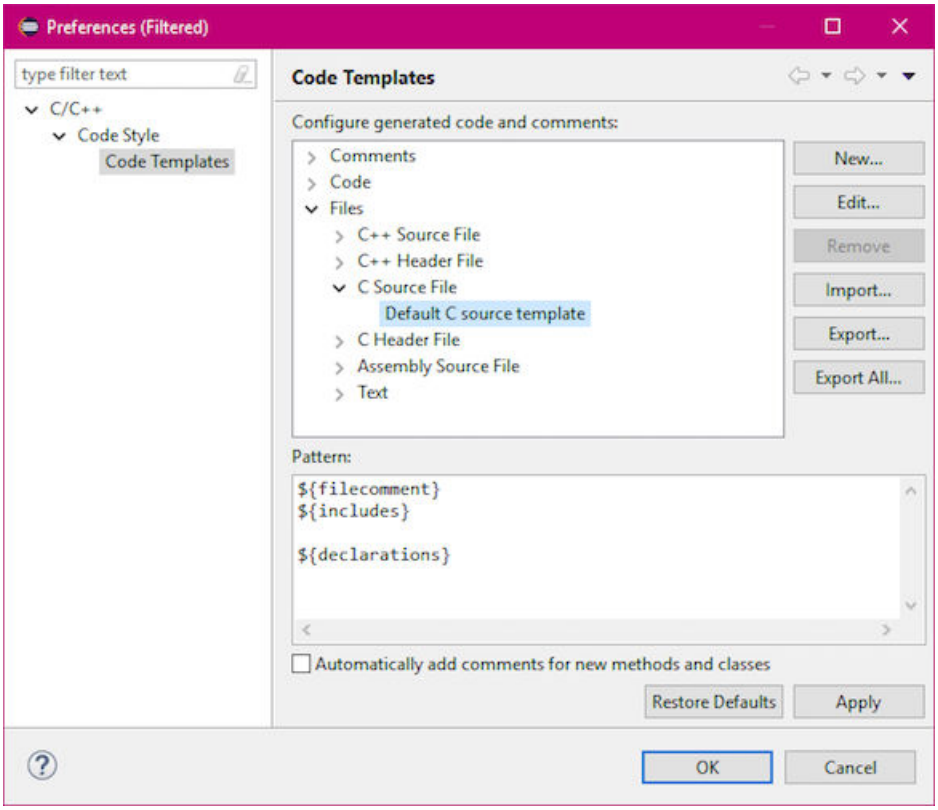


Figure 5-6 Code template configuration

5. Click **Finish**.

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

Related information

Arm Development Studio Perspectives and Views

Eclipse online documentation: Code templates

5.9 Sharing Arm® Development Studio projects

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

Note

- 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 within Arm Development Studio are organized into projects, and projects exist within your workspace. A workspace contains many files, including files in the `.metadata` directory, that are specific to an individual user or installation.

Within 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 within 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.

5.10 Importing and exporting options

A resource must exist in a project within 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.

5.11 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.

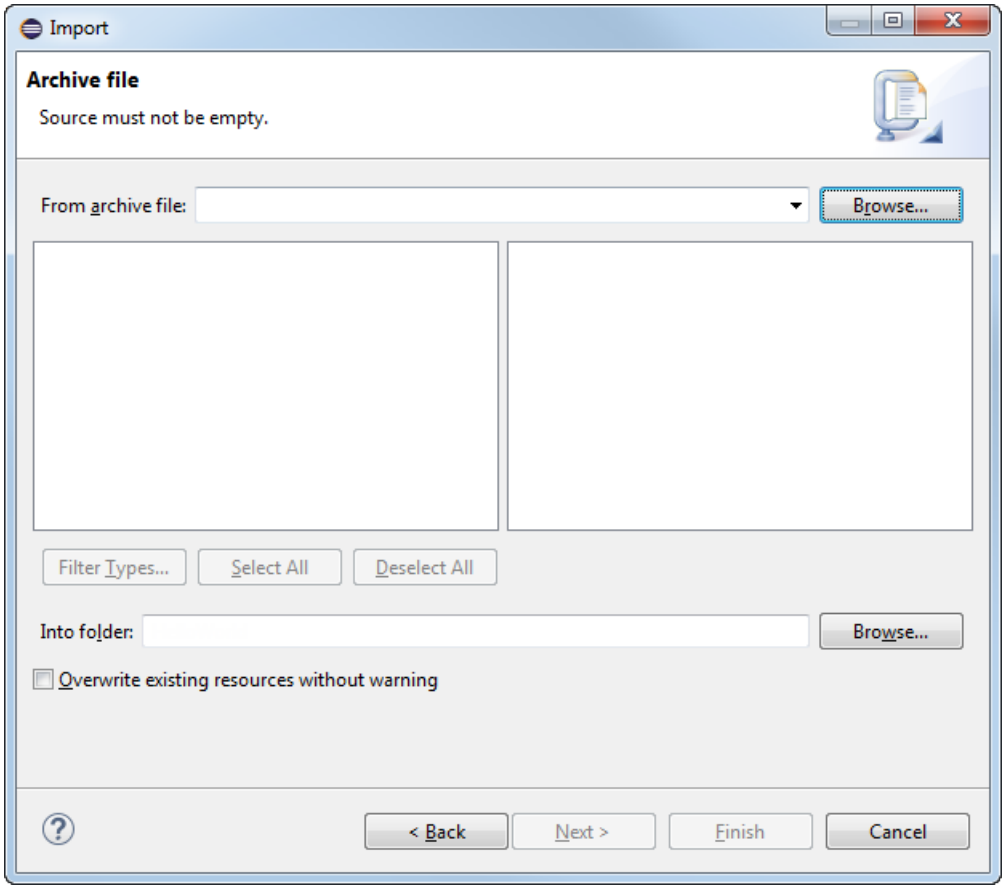


Figure 5-7 Typical example of the import wizard

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.

5.12 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.

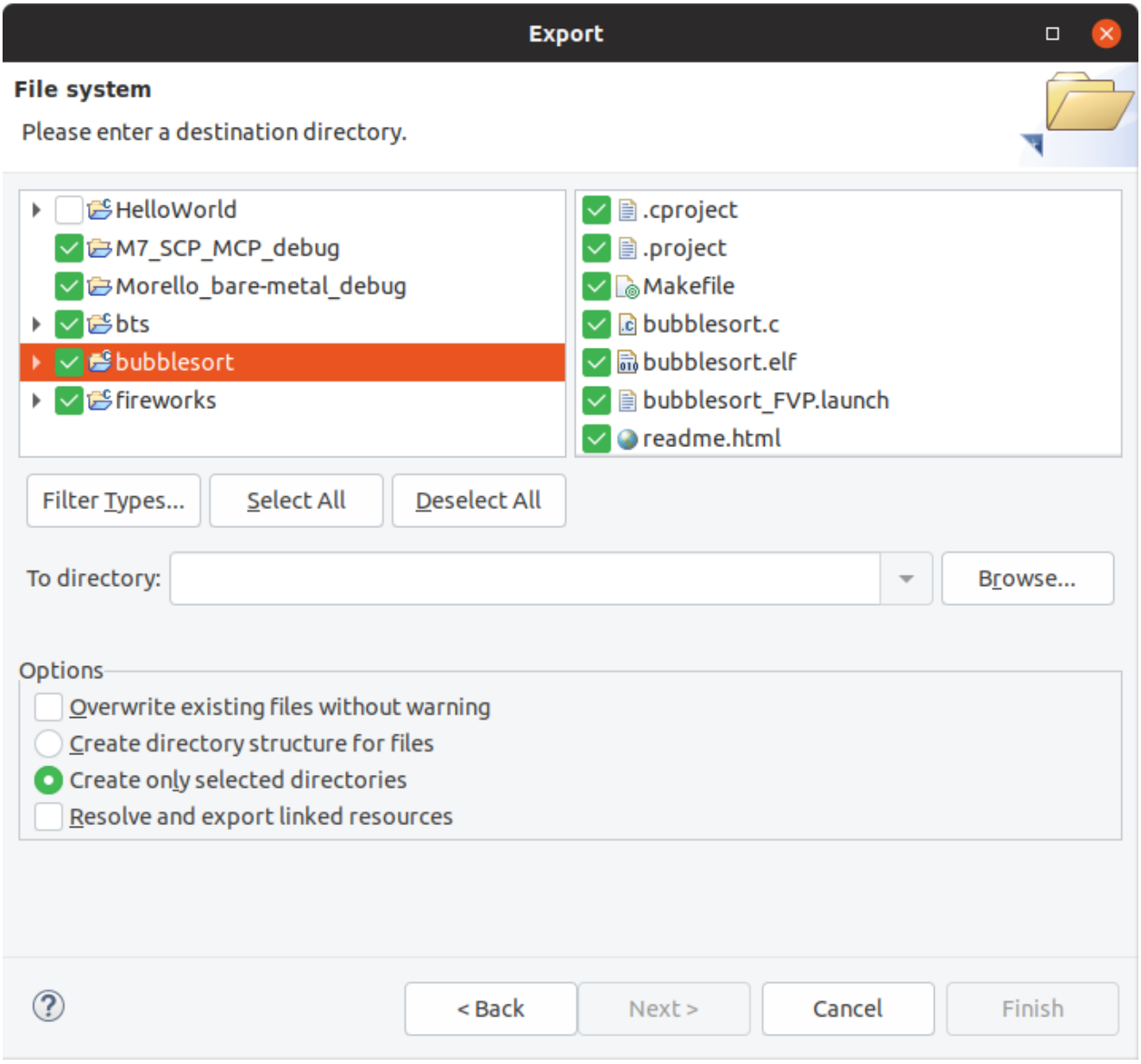


Figure 5-8 Typical example of the export wizard

5.13 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**

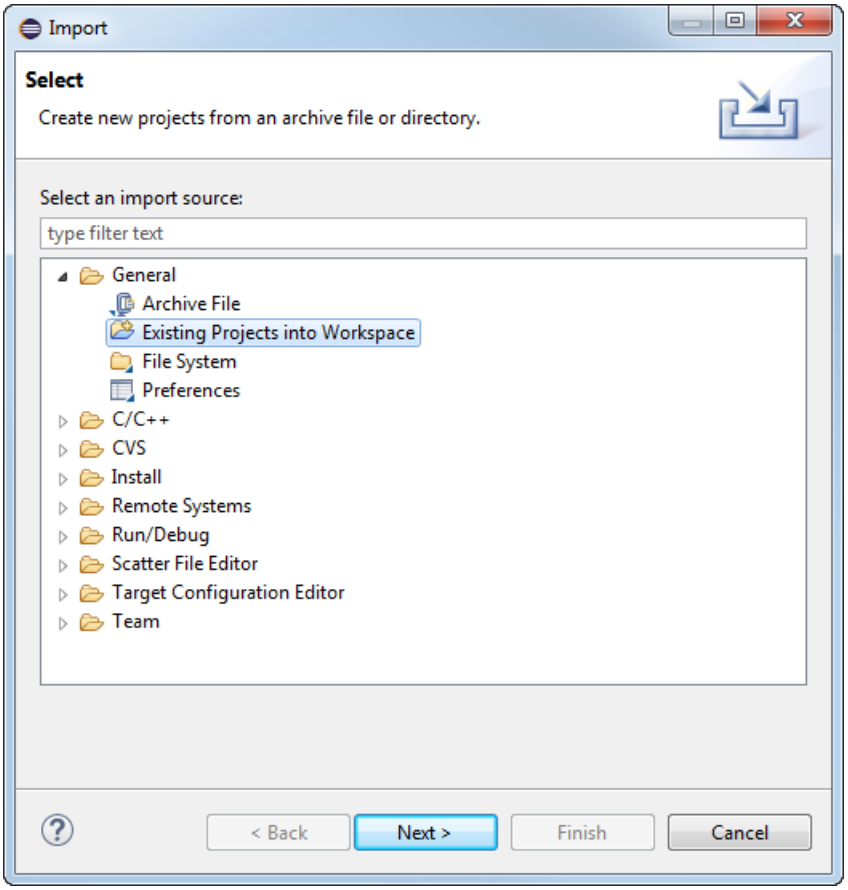


Figure 5-9 Selecting the import source type

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**.

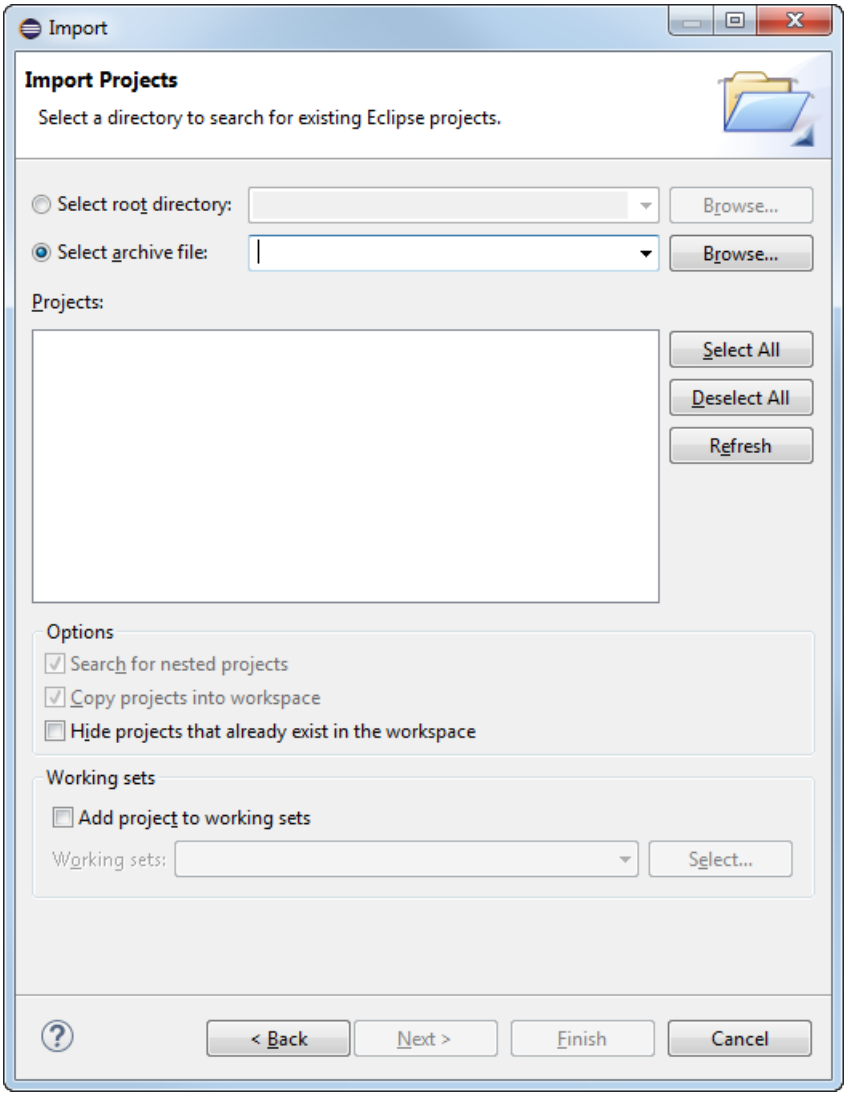


Figure 5-10 Selecting an existing Eclipse projects for import

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.

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

5.14 Examples provided with Arm Development Studio Morello Edition

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

Bare-metal software development examples for Morello that illustrate:

- Compiling C code for Morello using the LLVM compiler with Morello support.
- Running the executable image on the Morello FVP model.
- Debugging the executable image using the Arm Debugger, including viewing a disassembly of the Morello instructions and viewing the Morello-specific registers and capabilities.
- Capturing the history of instruction execution (“trace”) from the FVP model.

The code is located in the archive file `<examples_directory>/Morello_examples.zip`.

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 Morello Edition 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 tasks

5.15 Import the example projects on page 5-72

5.15 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. If you want to create a separate workspace for your example projects, select **File** > **Switch Workspace** > **Other** > **Browse** > **Make new folder**, and enter a suitable name.
3. In the main menu, select **File** > **Import...**.
4. Expand the **Arm Development Studio** group, select **Examples and Programming Libraries** and click **Next**.

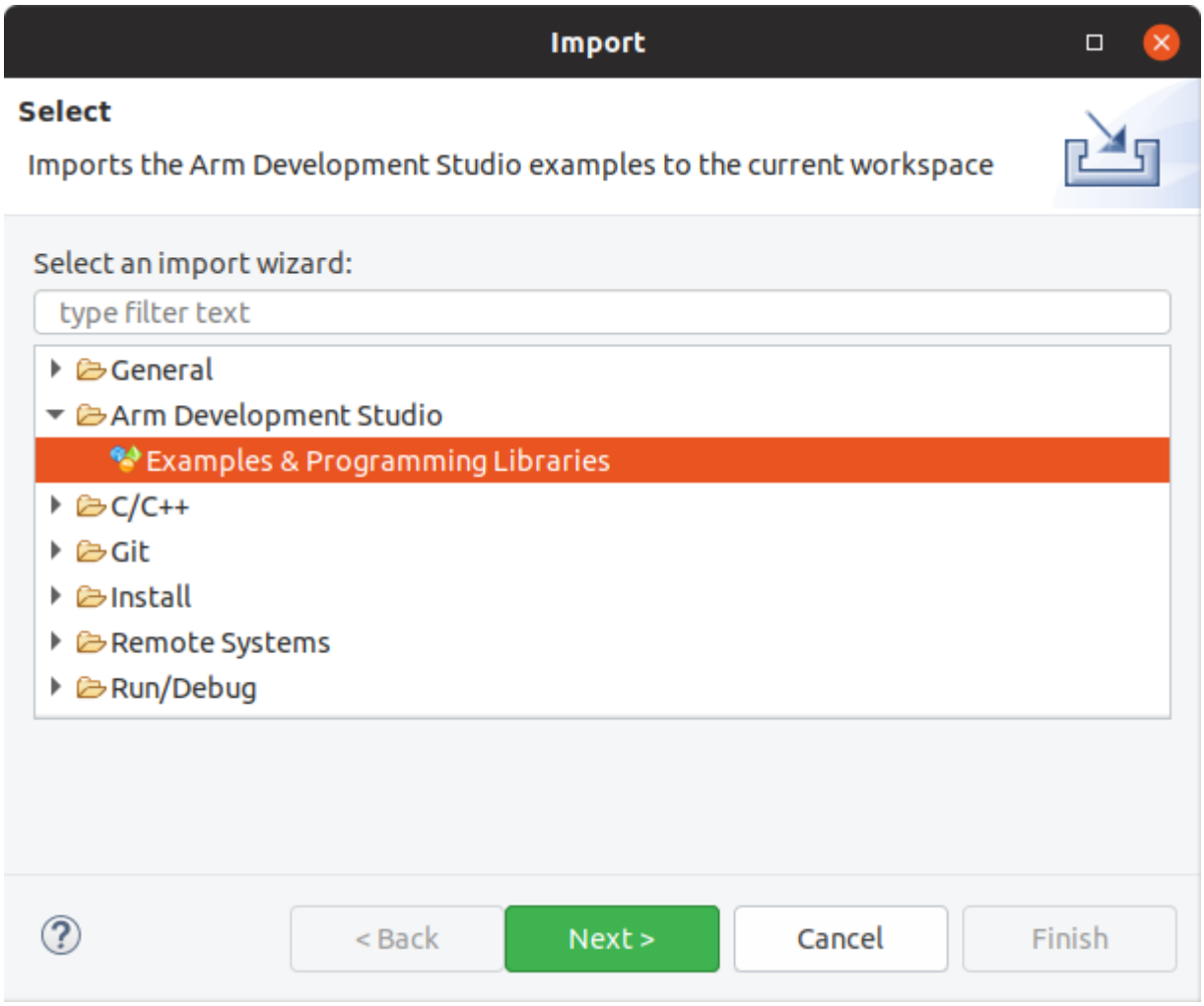


Figure 5-11 Import dialog box

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.

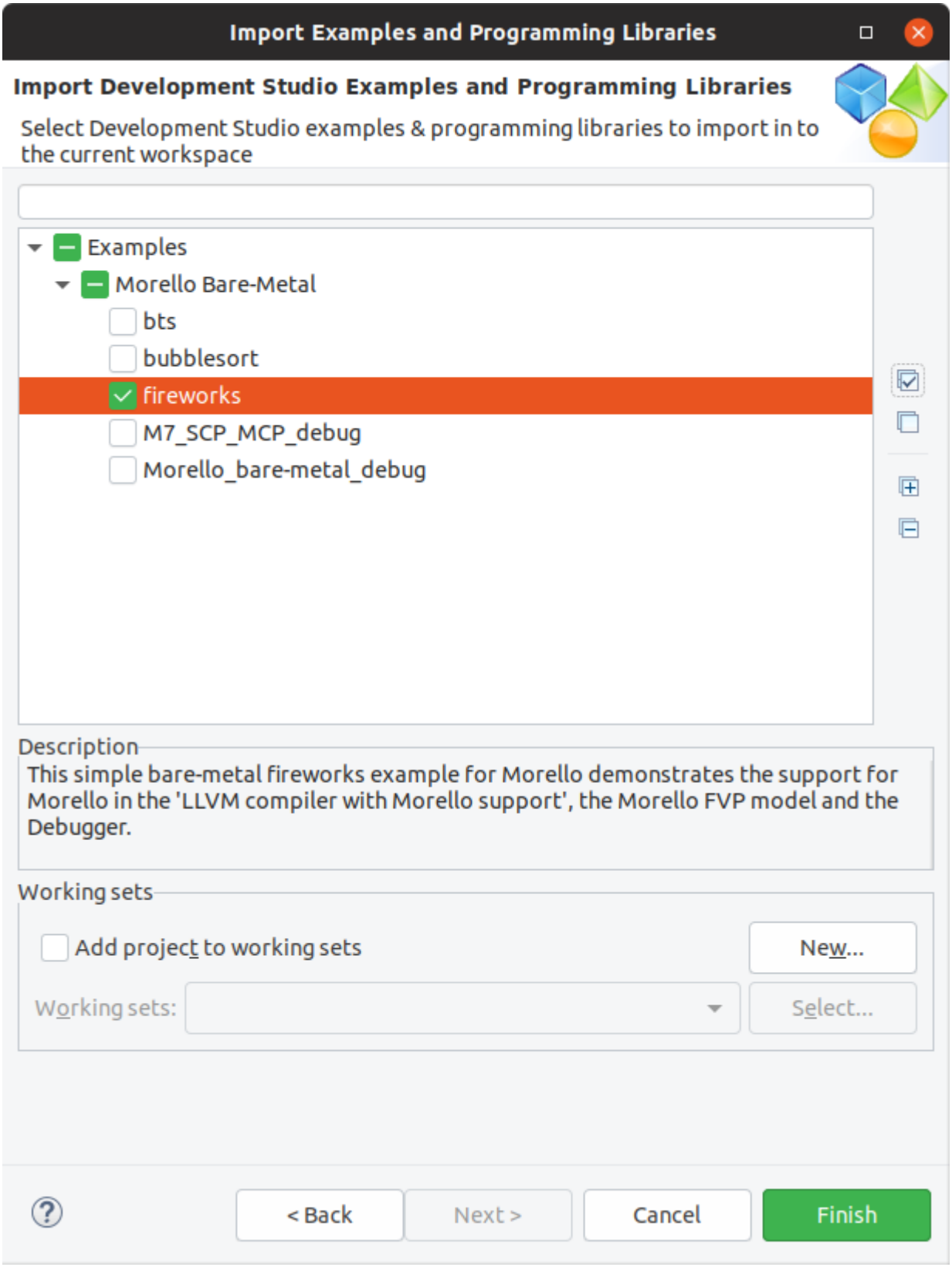


Figure 5-12 Select items to import

6. Click **Finish**.

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

Chapter 6
Writing code

The following topics describe how to use the editors when developing a project for an Arm target.

It contains the following sections:

6.1 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.

When you have two or more editor tabs open, you can tile them for side-by-side viewing by clicking on a tab and dragging 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 within 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 within 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 shorten 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 and can be accessed by selecting **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.

6.2 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.

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.

6.3 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.

6.4 ELF content editor - Header tab

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

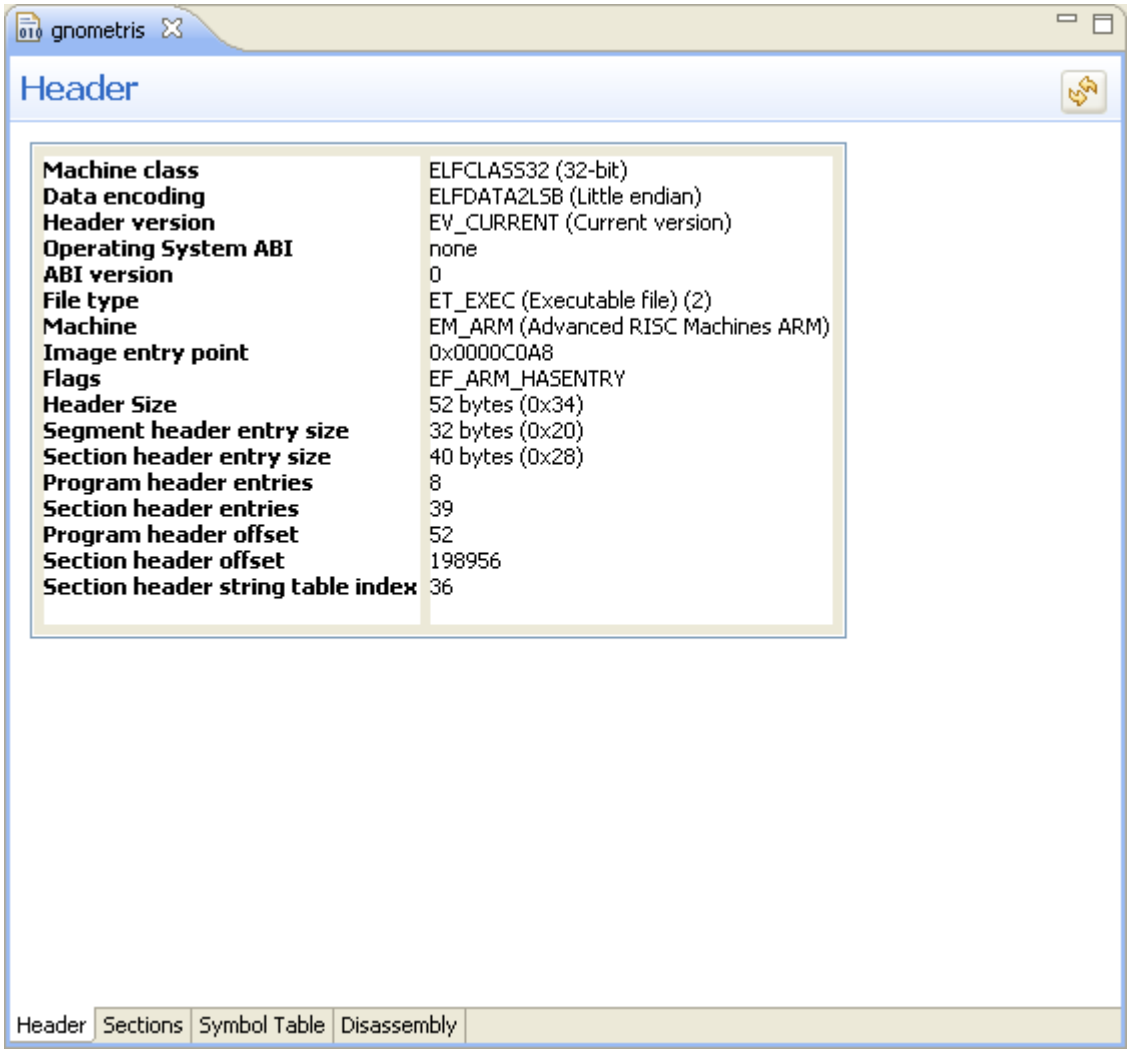
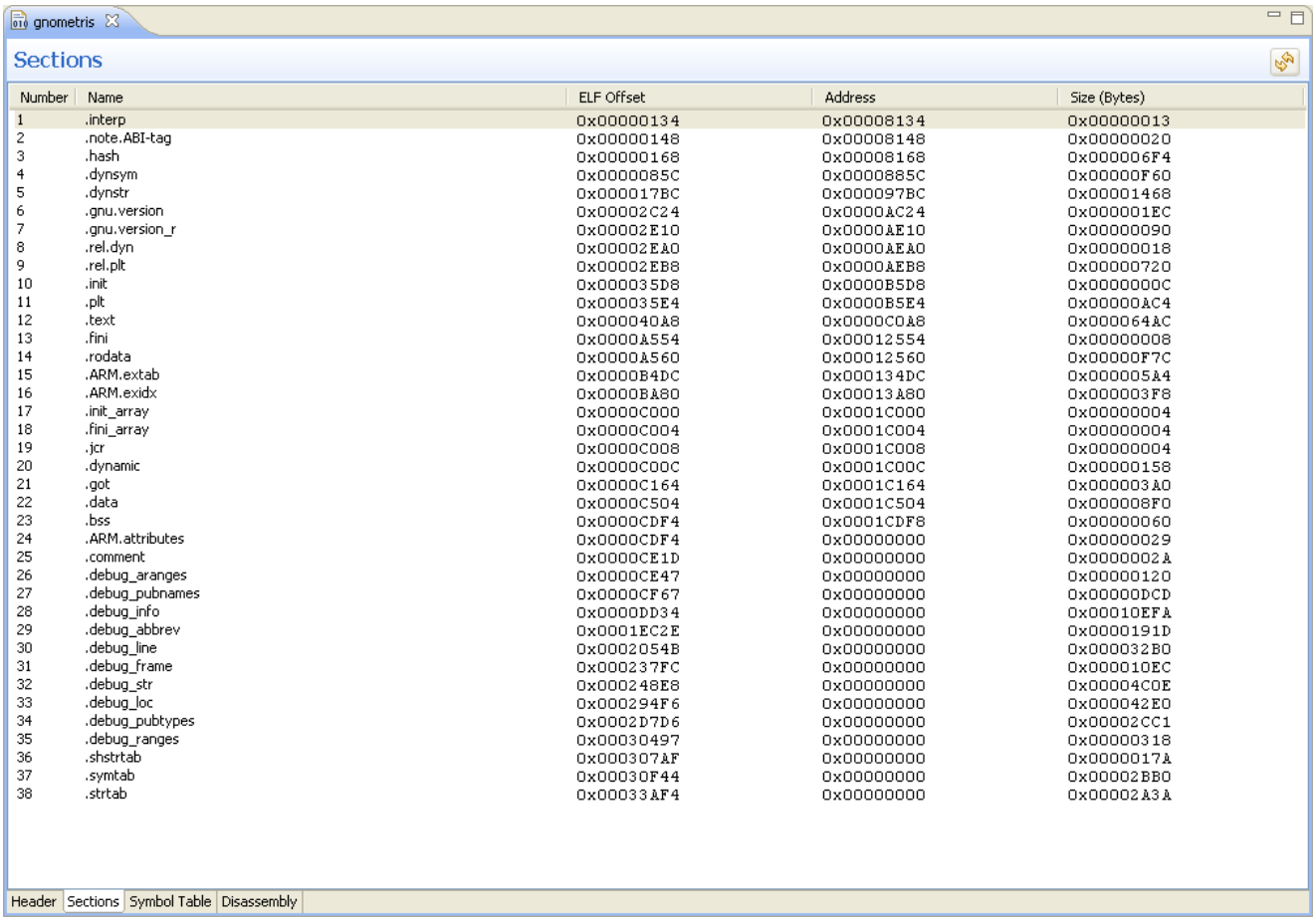


Figure 6-1 Header tab

6.5 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.



Number	Name	ELF Offset	Address	Size (Bytes)
1	.interp	0x00000134	0x00008134	0x00000013
2	.note.ABI-tag	0x00000148	0x00008148	0x00000020
3	.hash	0x00000168	0x00008168	0x000006F4
4	.dynsym	0x0000085C	0x0000885C	0x00000F60
5	.dynstr	0x000017BC	0x000097BC	0x00001468
6	.gnu.version	0x00002C24	0x0000AC24	0x000001EC
7	.gnu.version_r	0x00002E10	0x0000AE10	0x00000090
8	.rel.dyn	0x00002EA0	0x0000AEA0	0x00000018
9	.rel.plt	0x00002EB8	0x0000AEB8	0x00000720
10	.init	0x000035D8	0x0000B5D8	0x0000000C
11	.plt	0x000035E4	0x0000B5E4	0x00000AC4
12	.text	0x000040A8	0x0000C0A8	0x000064AC
13	.fini	0x0000A554	0x00012554	0x00000008
14	.rodata	0x0000A560	0x00012560	0x00000F7C
15	.ARM.extab	0x0000B4DC	0x000134DC	0x000005A4
16	.ARM.exidx	0x0000BA80	0x00013A80	0x000003F8
17	.init_array	0x0000C000	0x0001C000	0x00000004
18	.fini_array	0x0000C004	0x0001C004	0x00000004
19	.jcr	0x0000C008	0x0001C008	0x00000004
20	.dynamic	0x0000C00C	0x0001C00C	0x00000158
21	.got	0x0000C164	0x0001C164	0x000003A0
22	.data	0x0000C504	0x0001C504	0x000008F0
23	.bss	0x0000CDF4	0x0001CDF8	0x00000060
24	.ARM.attributes	0x0000CDF4	0x00000000	0x00000029
25	.comment	0x0000CE1D	0x00000000	0x0000002A
26	.debug_aranges	0x0000CE47	0x00000000	0x00000120
27	.debug_pubnames	0x0000CF67	0x00000000	0x00000DCD
28	.debug_info	0x0000DD34	0x00000000	0x00010EFA
29	.debug_abbrev	0x0001EC2E	0x00000000	0x0000191D
30	.debug_line	0x0002054B	0x00000000	0x000032B0
31	.debug_frame	0x000237FC	0x00000000	0x000010EC
32	.debug_str	0x000248E8	0x00000000	0x00004C0E
33	.debug_loc	0x000294F6	0x00000000	0x000042E0
34	.debug_pubtypes	0x0002D7D6	0x00000000	0x00002CC1
35	.debug_ranges	0x00030497	0x00000000	0x00000318
36	.shstrtab	0x000307AF	0x00000000	0x0000017A
37	.symtab	0x00030F44	0x00000000	0x00002BB0
38	.strtab	0x00033AF4	0x00000000	0x00002A3A

Figure 6-2 Sections tab

6.6 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.

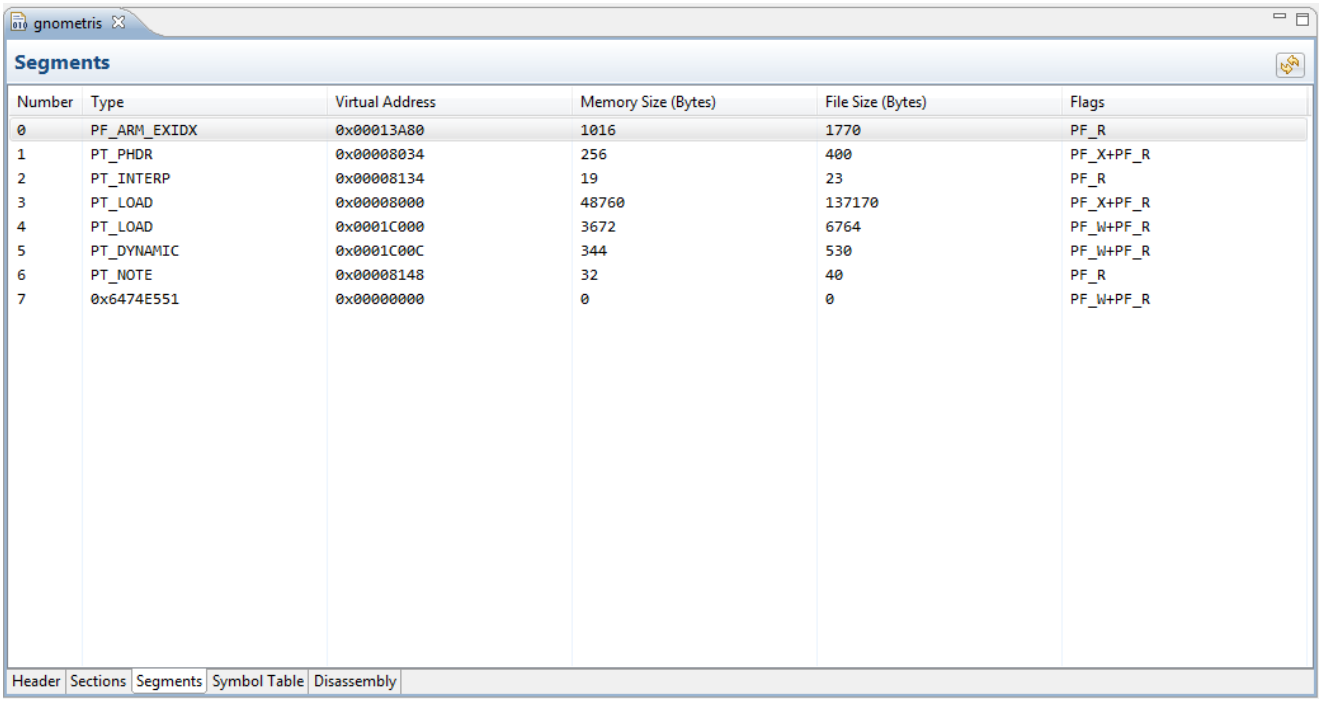


Figure 6-3 Segments tab

6.7 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.

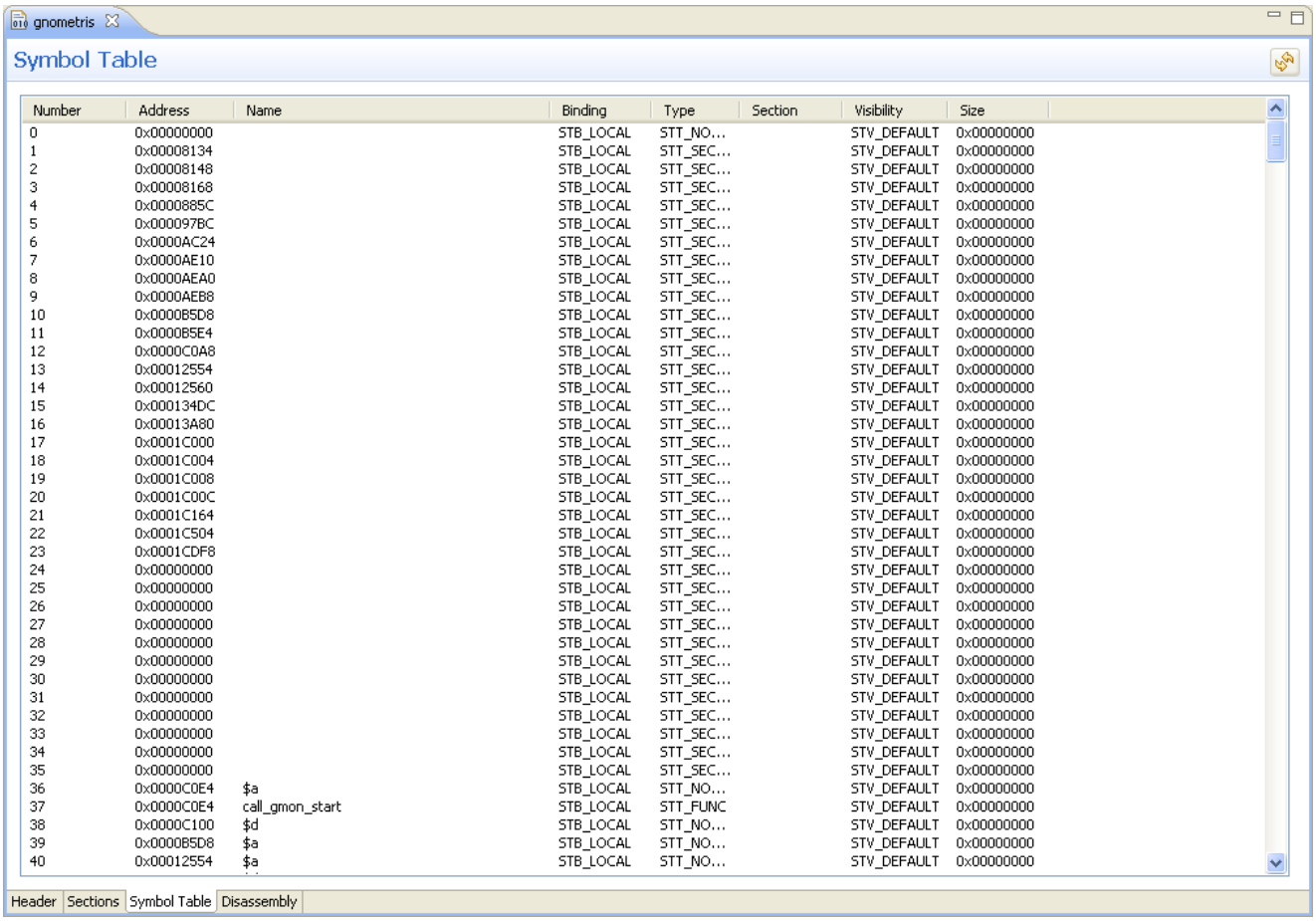


Figure 6-4 Symbol Table tab

6.8 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 can be used 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 use the **Copy** and **Find** options in the context menu by right-clicking in the **Disassembly** view.

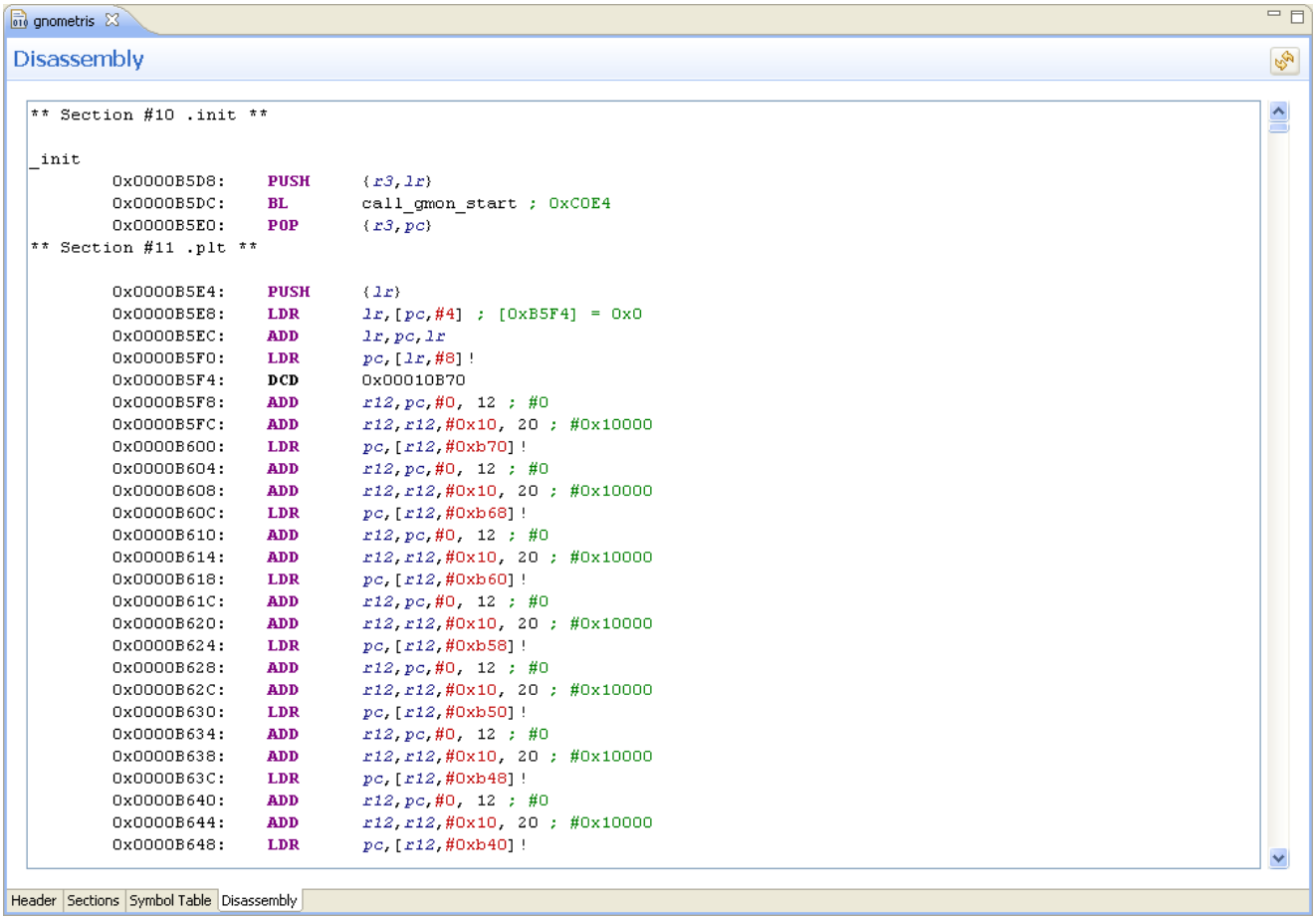


Figure 6-5 Disassembly tab

Chapter 7
Debugging code

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

It contains the following sections:

7.1 Overview: Debug connections in Arm® Debugger

Arm Debugger can debug bare-metal code on a Fixed Virtual Platform (FVP).

Bare-metal targets run without an underlying operating system. When you are debugging code on an FVP, you must use a CADI-compliant connection between the debugger and the FVP.

Related information

Configuring a connection to a Linux application using gdbserver

7.2 Using Fixed Virtual Platform (FVP)s with Arm Development Studio Morello Edition

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

Depending on your requirements, you can:

- *From the command-line, configure a connection to an FVP model for bare-metal application debug* on page 7-87
- *Create a new model configuration and import new FVP models*

Note

To use Arm FVPs that are not provided with the Development Studio installation:

1. Add the `<install_directory>/bin` directory to your `PATH` environment variable, for example `export PATH=<your_model_path>/bin:$PATH`
2. Restart Arm Development Studio.
3. To ensure that the modified path is available for future sessions, set up the `PATH` in the appropriate shell configuration file. For example, in `.bashrc`, add the line `export PATH=<your_model_path>/bin:$PATH`

7.3 Configuring a connection from the command-line to a Fixed Virtual Platform (FVP)

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

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.

Prerequisites

- 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, refer to the documentation for your compiler and linker.

Procedure

1. Open the Arm Development Studio command prompt:
   - On Linux, add the `<install_directory/bin>` location to your *PATH* environment variable and then open a bash shell.
2. Connect to the Morello FVP model and specify an image to load from your workspace. Replace `<username>` with your username and at the command prompt, enter:
   - On Linux: `./armdbg --cdb-entry "Arm FVP::Morello::Bare Metal Debug::Bare Metal Debug::Rainierx4 Multi-Cluster SMP" --image "/home/<username>/developmentstudio-workspace/bubblesort/bubblesort.elf`

Development Studio starts the Morello 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.

7.4 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.

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**.

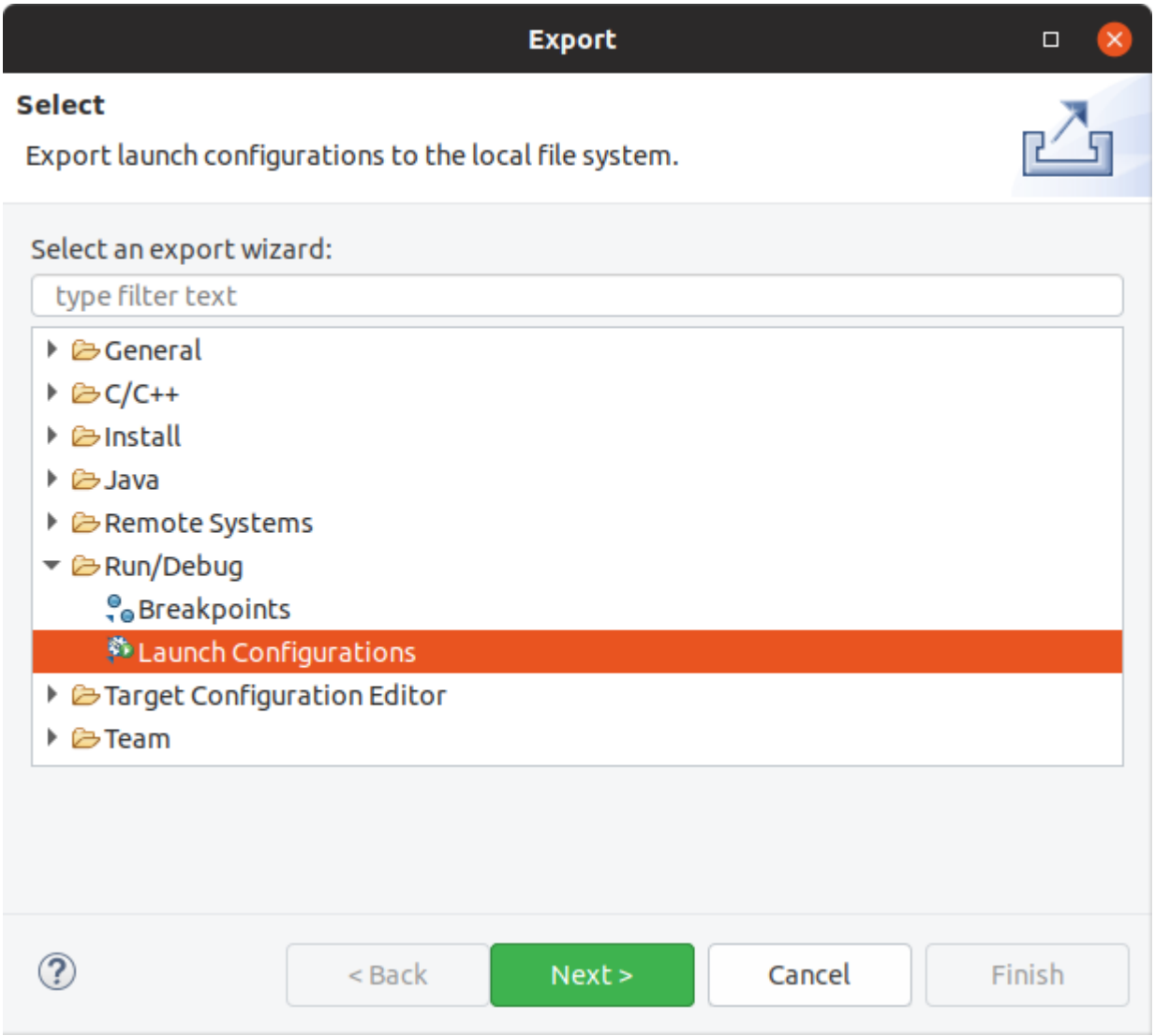


Figure 7-1 Export Launch Configuration dialog box

3. Click **Next**.

4. In the **Export Launch Configurations** dialog box:
   a. Expand 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**.

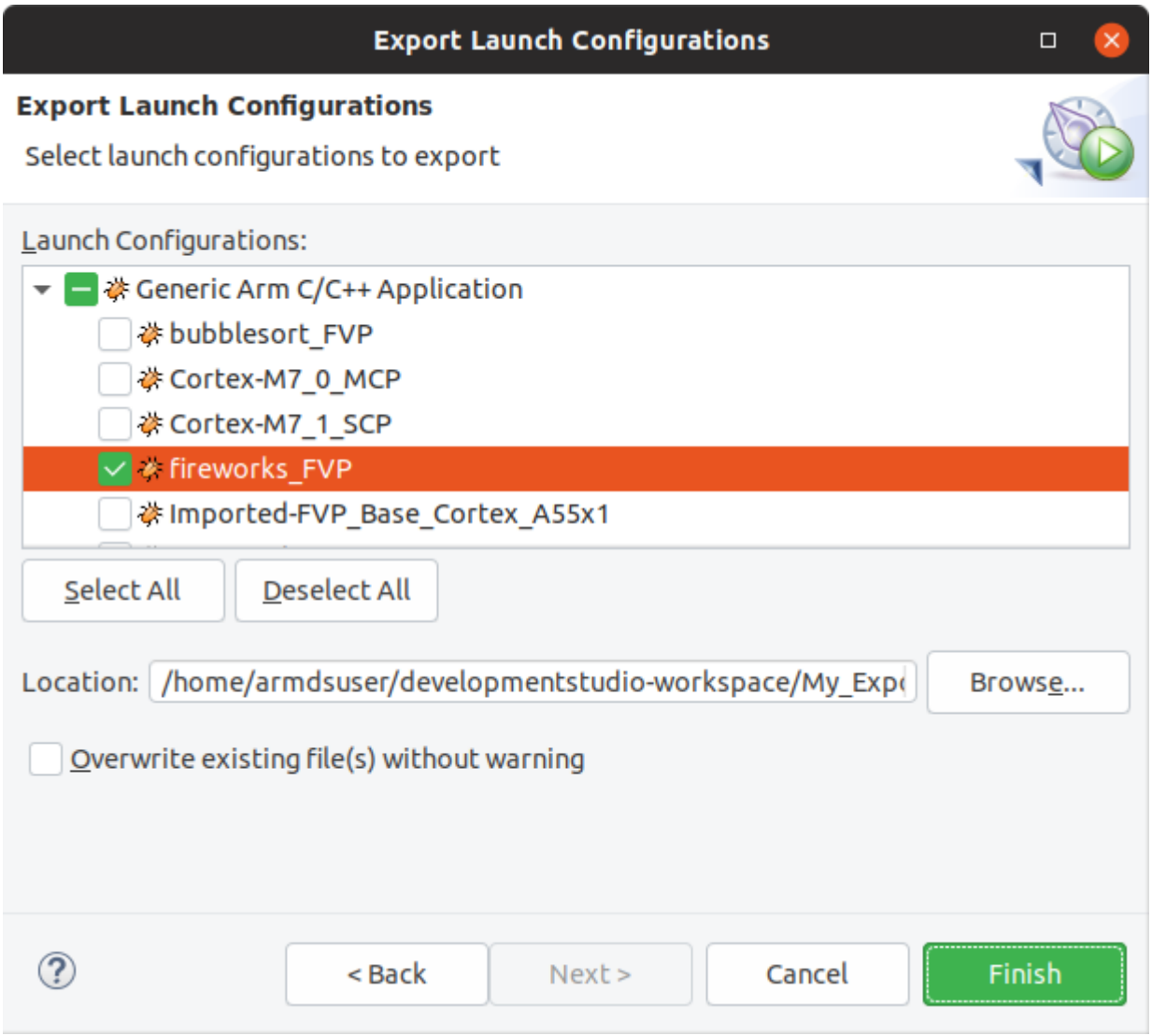


Figure 7-2 Select Launch Configurations for export

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.

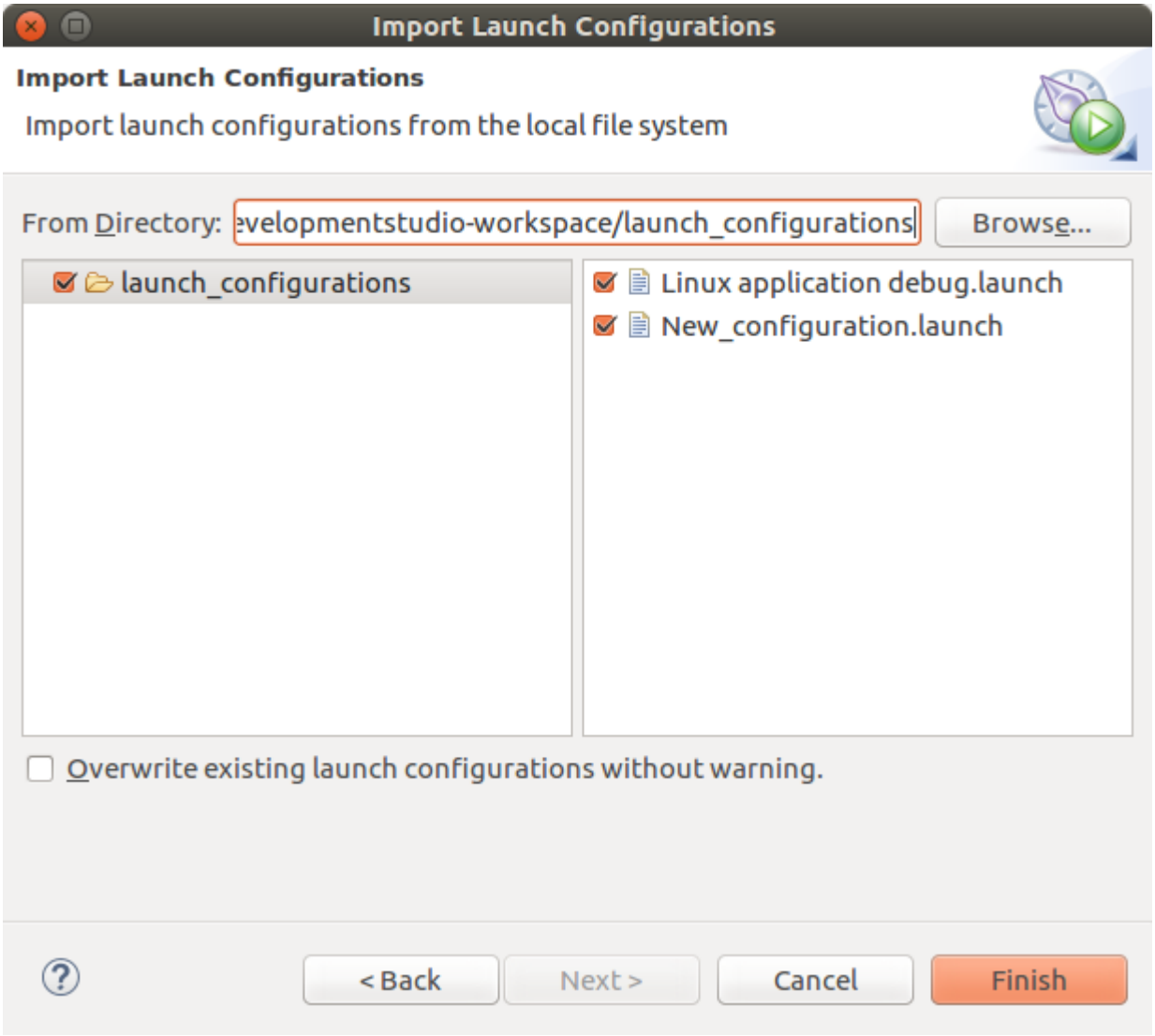


Figure 7-3 Import launch configuration selection panel

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.

7.5 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  .

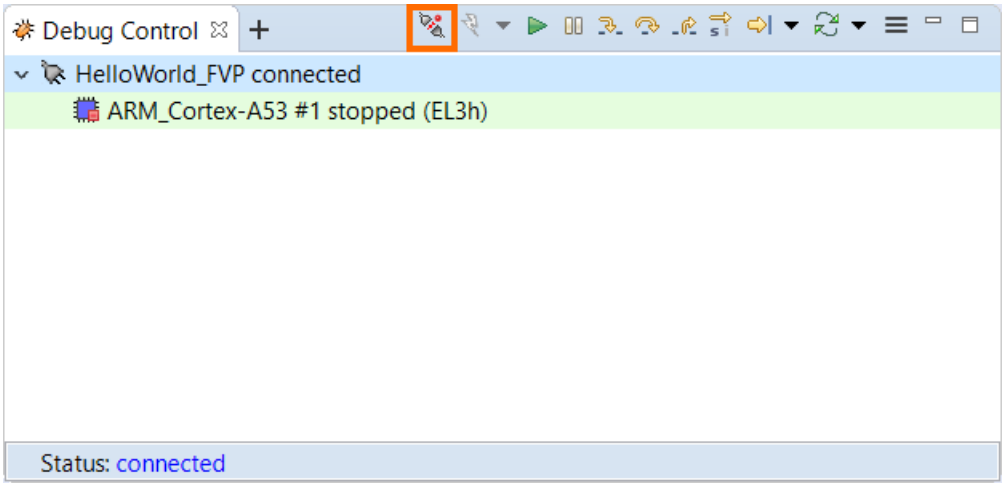


Figure 7-4 Disconnecting from a target using the Debug Control view

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

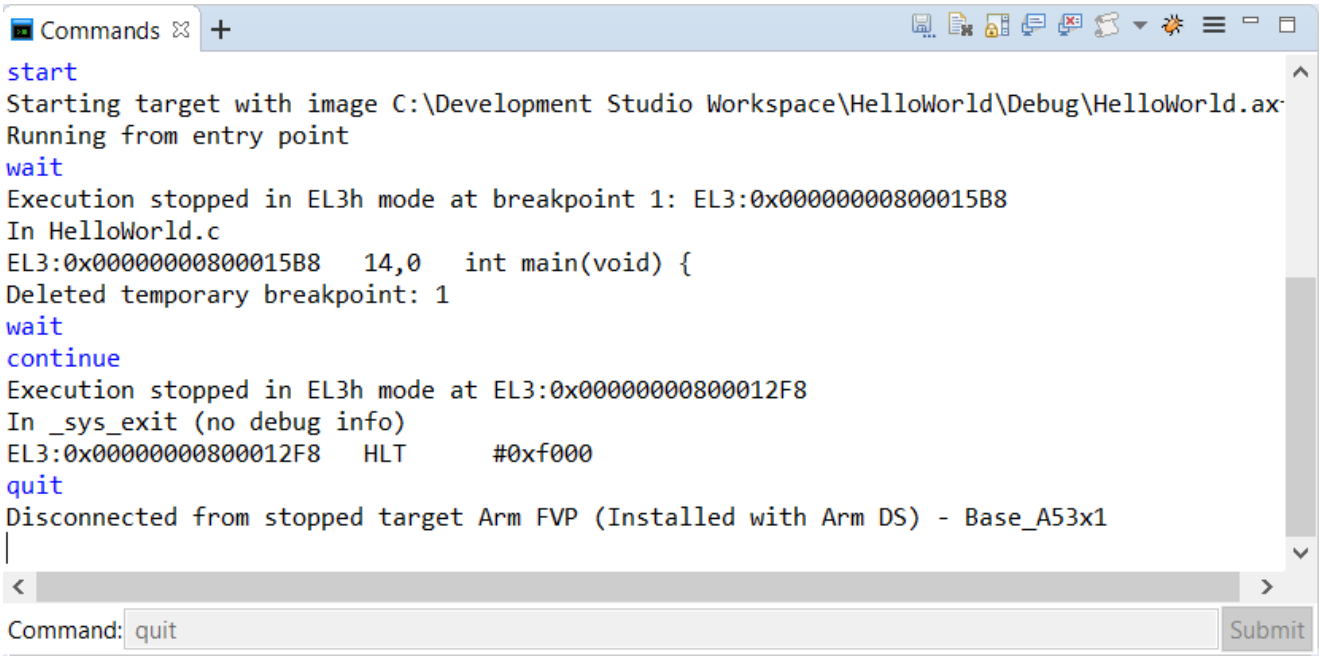


Figure 7-5 Disconnecting from a target using the Commands view

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.

Chapter 8
Tutorials

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

It contains the following section:

- *8.1 Tutorial: Hello World* on page 8-93.

8.1 Tutorial: Hello World

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

This section contains the following subsections:

8.1.1 Open Arm® Development Studio for the first time

The first time you open Arm Development Studio Morello Edition, it opens the Development Studio perspective.

Prerequisites

- Download and install Arm Development Studio:
  - Linux: *Installing on Linux* on page 2-20

Procedure

- Open Arm Development Studio:
  - On Linux:
    - GUI: Use your Linux variant's menu system to locate Arm Development Studio.
    - Command line: Run `<installation_directory>/bin/armds_ide`

Arm Development Studio opens. See *Integrated Development Environment (IDE) Overview* on page 4-46, which describes the main features of the user interface.

Note

The workspace is automatically set by default, to either:

- Linux: `<userhome>/developmentstudio-workspace`

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

8.1.2 Create a project in C/C++

After installing Arm Development Studio, we are going to create a simple Hello World C project, compile it using the LLVM compiler with Morello support, and run it on the Morello Fixed Virtual Platform (FVP).

Prerequisites

- Complete *Open Arm® Development Studio for the first time* on page 8-93
- Install the LLVM toolchain.
- 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.

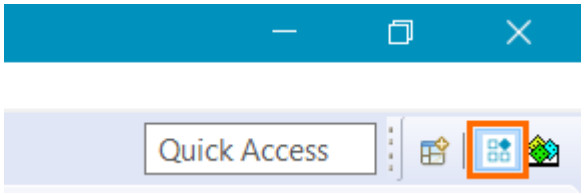


Figure 8-1 Screenshot highlighting the button for the Development Studio Perspective.

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**.
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 **LLVM 11.0.0**.
   d. Click **Finish**.

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

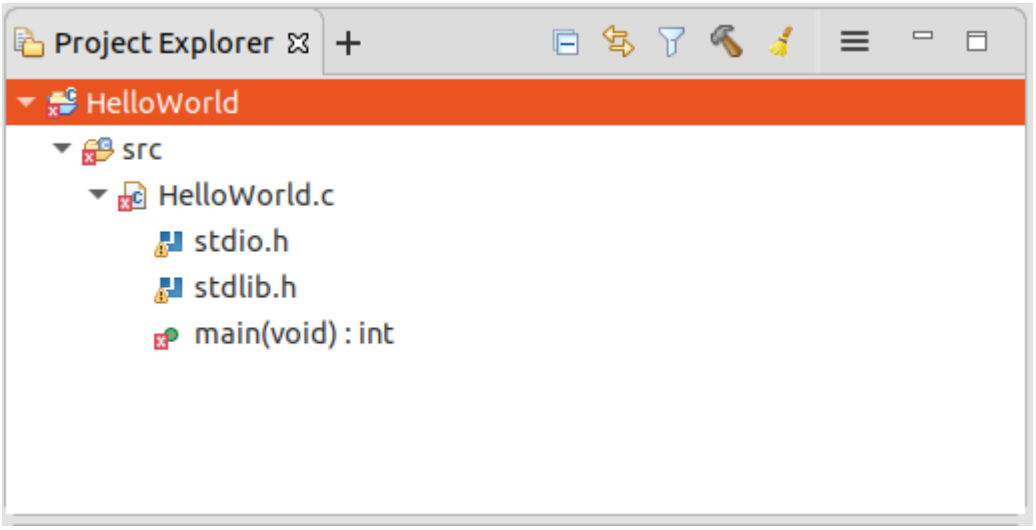


Figure 8-2 Hello World project in the Project Exporer view

8.1.3 Configure your project

Before you build the `HelloWorld` project, you must specify the compiler, assembler, and linker target options.

Prerequisites

Complete *Create a project in C/C++* on page 8-93

Procedure

1. In the **Project Explorer** view, right-click the `HelloWorld` project and select **Properties**. The **Properties for HelloWorld** dialog box opens.
2. Select **C/C++ Build** > **Settings** > **Tool Settings** tab.
3. Set the **LLVM C Compiler** options:
   a. In **Target**, enter *aarch64-none-elf*.
   b. In **Architecture**, enter *morello+c64*.
   c. In **ABI (-mabi)**, enter *purecap*.

4. Set the **LLVM Assembler** options:
   a. In **Target**, enter *aarch64-none-elf*.
5. Set the **LLVM C Linker** options:
   a. In **Target**, enter *aarch64-none-elf*.
   b. In **Architecture**, enter *morello+c64*.
   c. In **ABI (-mabi)**, enter *purecap*.
6. Click **Apply and Close**.
7. If you are prompted to rebuild the index, click **Yes**.

8.1.4 Build your project

You can now build your `HelloWorld` project!

Prerequisites

Complete these tasks:

- *Create a project in C/C++* on page 8-93
- *Configure your project* on page 8-94

Procedure

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

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.

8.1.5 Configure your debug session

In Arm Development Studio Morello Edition, you configure a debugging session by creating a debug connection to your target using the **New Debug Connection** wizard.

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

Prerequisites

Confirm that the Morello FVP is running and accepting connections before you attempt a debug connection. The *Morello bare-metal debug* example provided with Arm Development Studio Morello Edition contains details about how to prepare and run the model.

Procedure

1. From the main menu, select **File** > **New** > **Model Connection**.
2. 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* on page 8-95.
   c. Click **Next**.
3. In the **Model Connection** dialog box, specify the details of the model:
   a. Select **Arm FVP** > **Morello**.

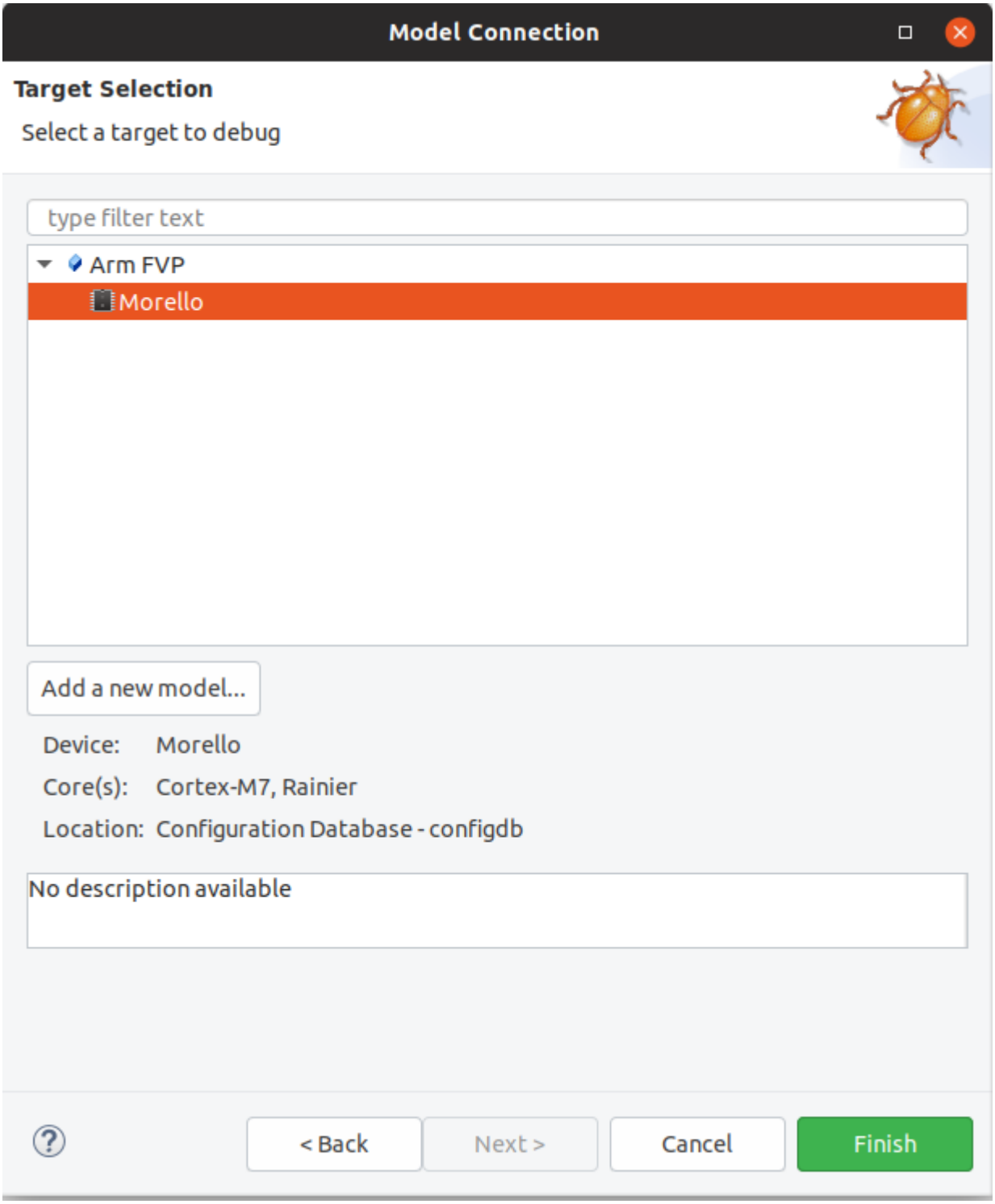


Figure 8-3 Select Morello model

b. Click **Finish**.

4. 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, select **Arm FVP** > **Morello** > **Bare Metal Debug** > **Rainierx4 Multi-Cluster SMP**.
   b. In the **Files** tab, select **Target Configuration** > **Application on host to download** > **Workspace**.

c. Click and expand the **HelloWorld** project and from the **Debug** folder, select `HelloWorld.axf` and click **OK**.
d. In the **Debugger** tab, select **Debug from symbol**.

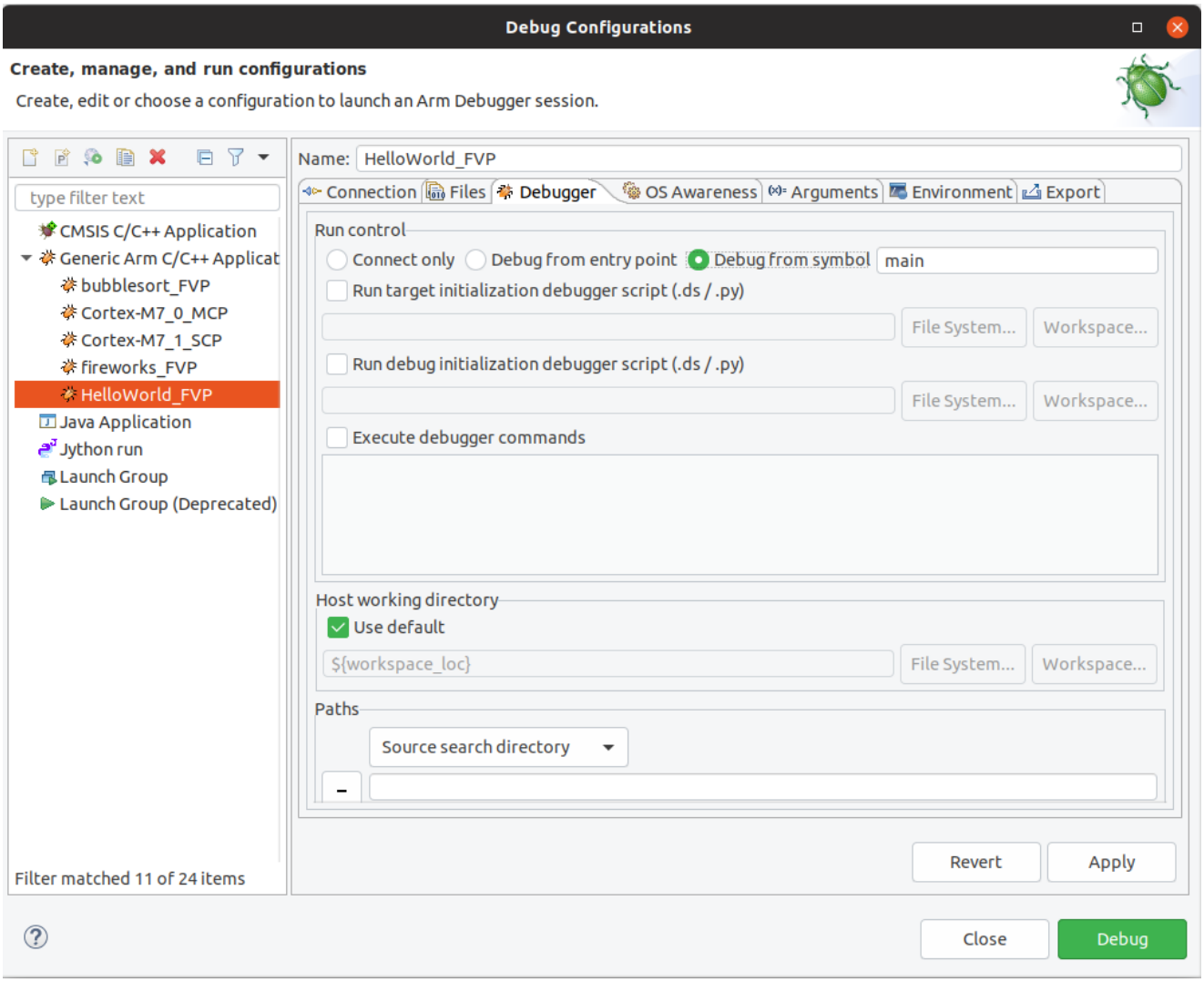


Figure 8-4 Debug from symbol main

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

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

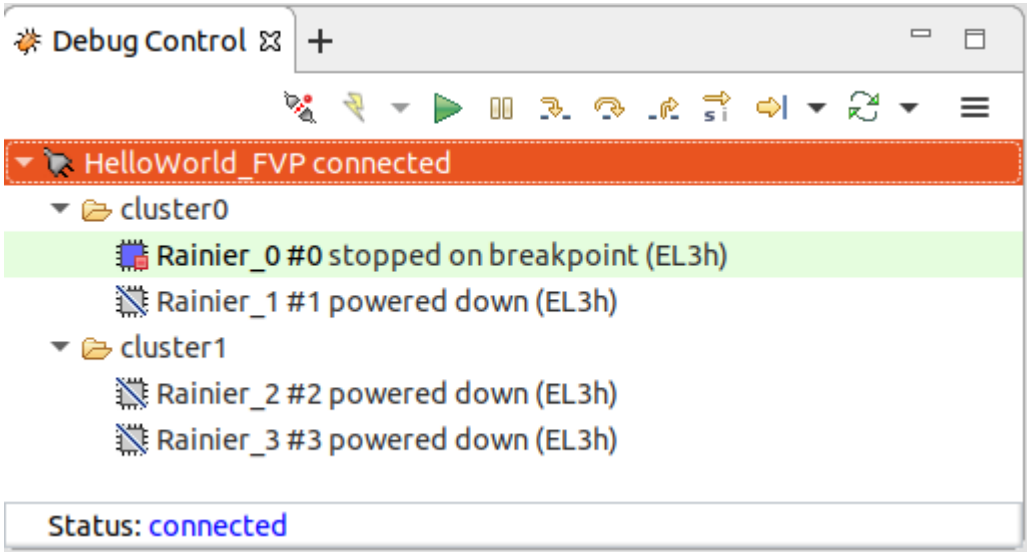


Figure 8-5 Debug Control View Rainier Core

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

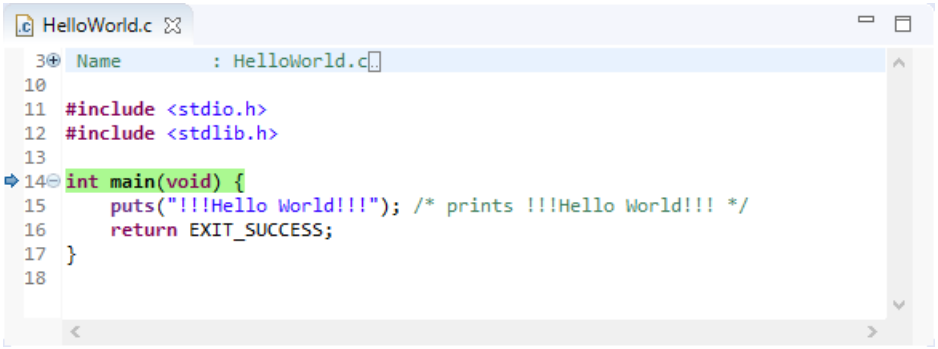


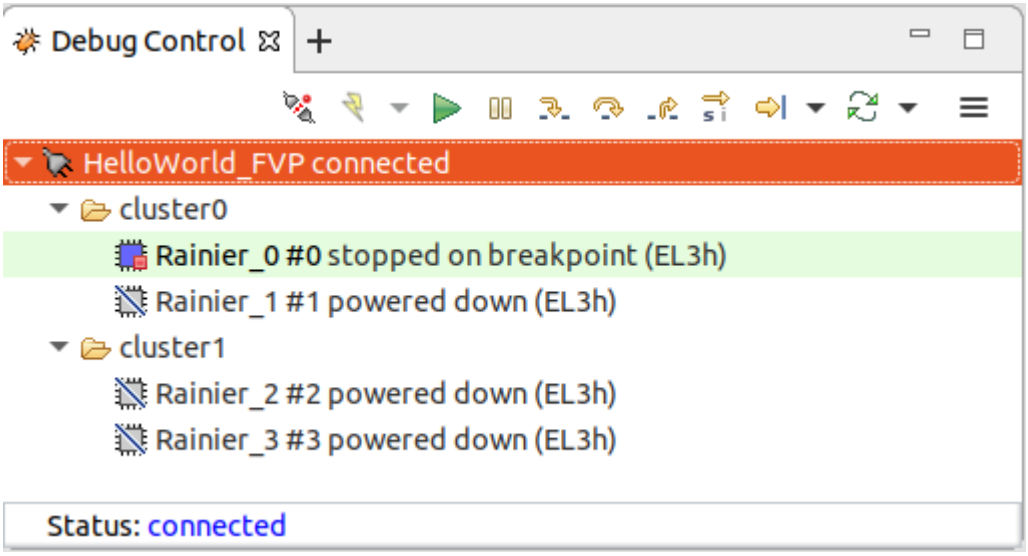
Figure 8-6 main () in code editor

8.1.6 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

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

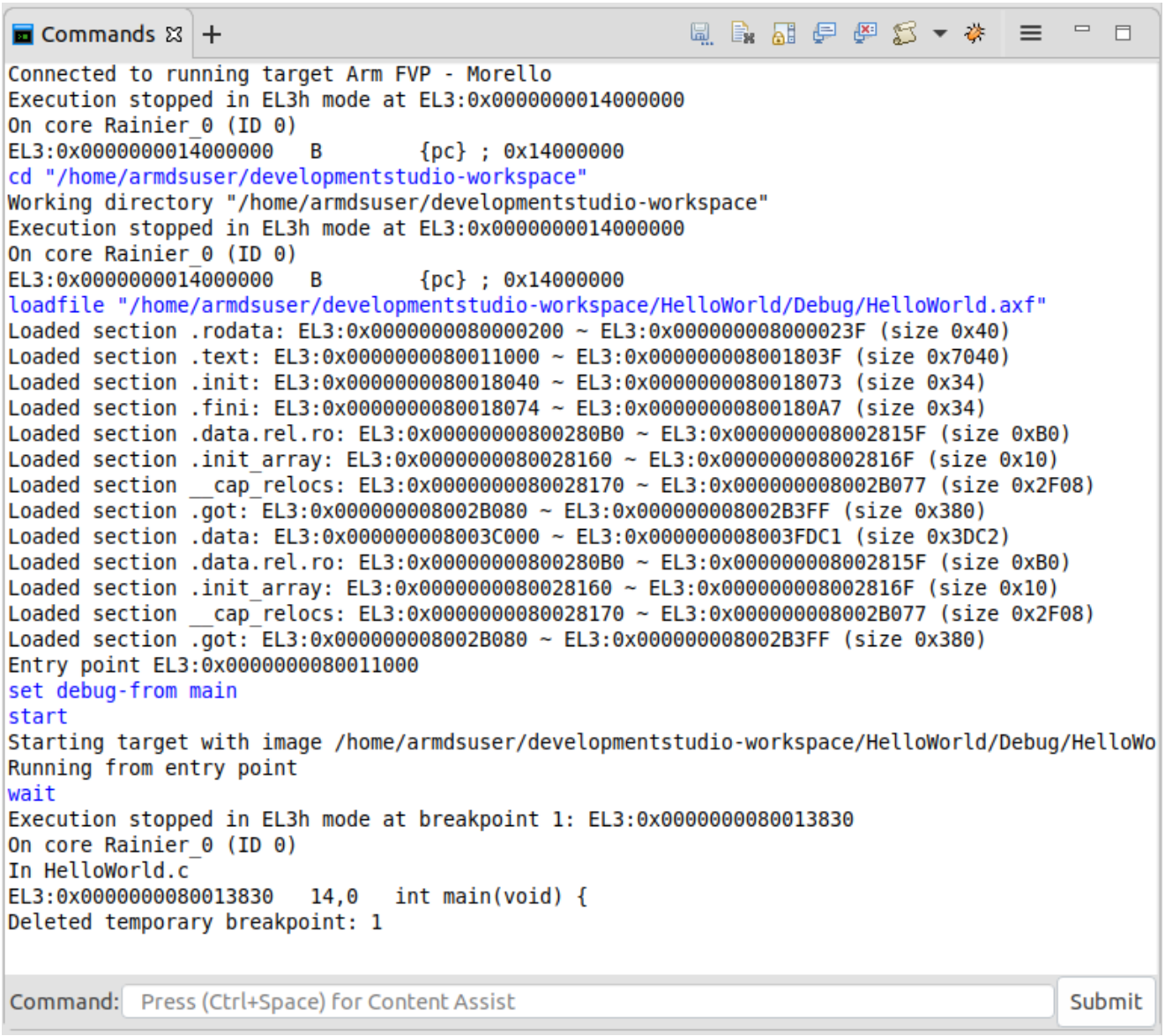


Figure 8-7 Commands view

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

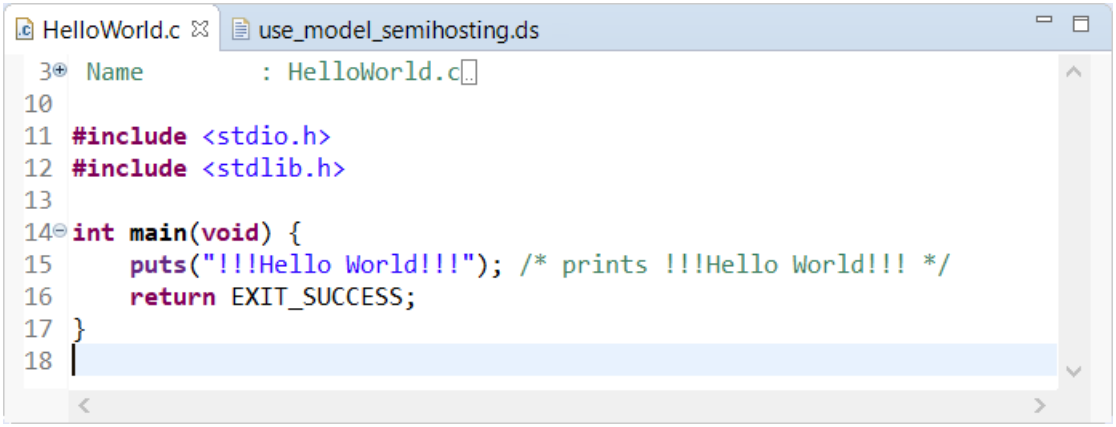


Figure 8-8 Code Editor view

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

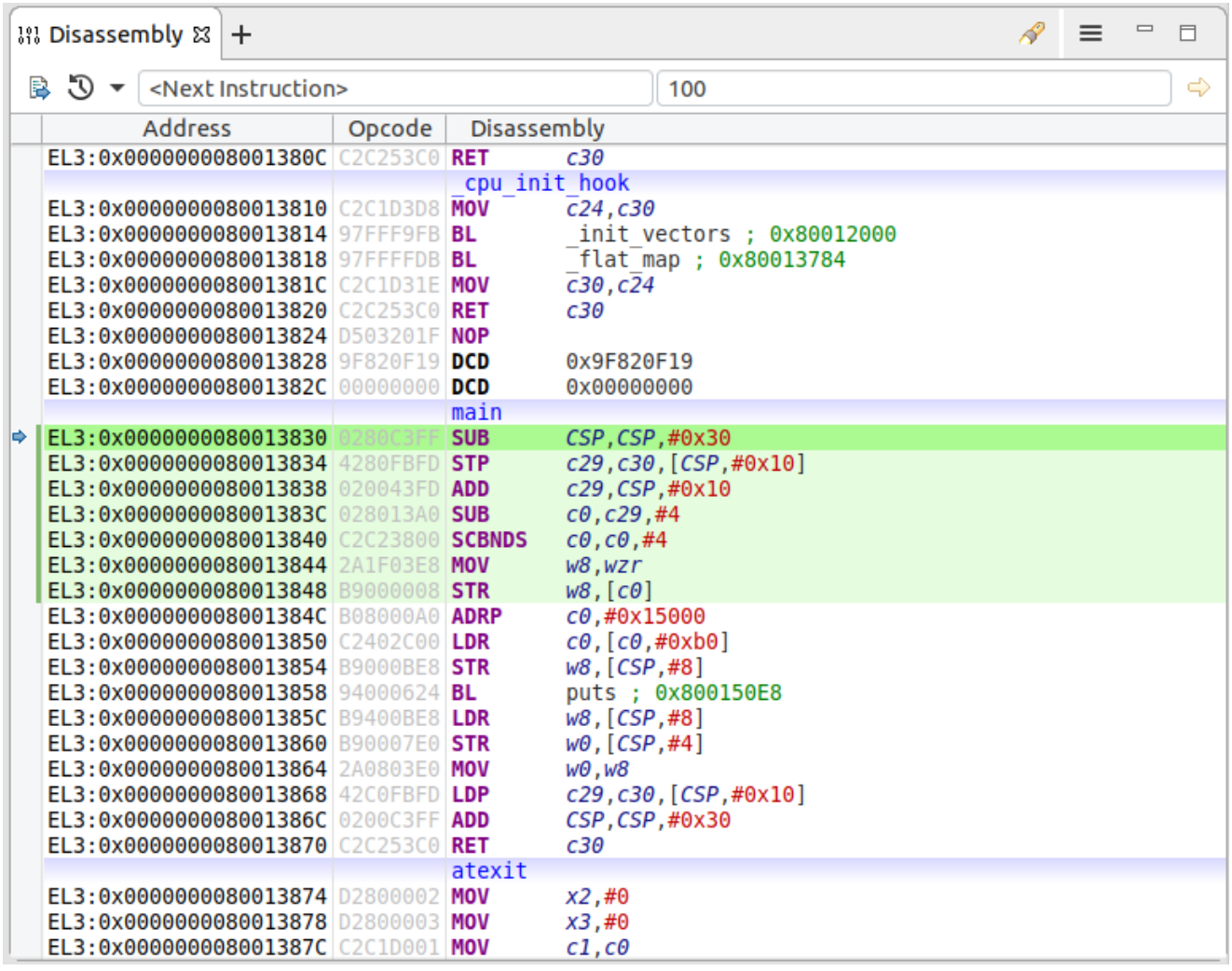


Figure 8-9 Disassembly view

 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.

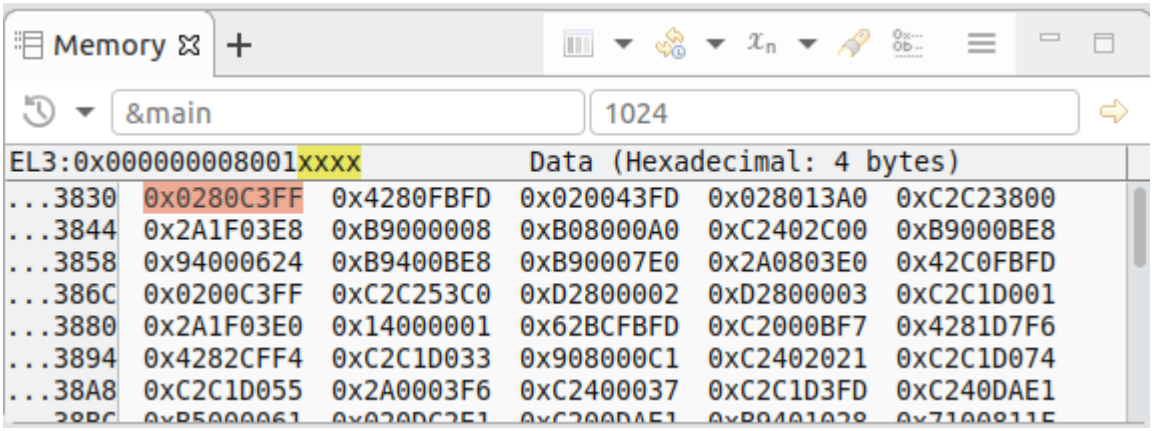


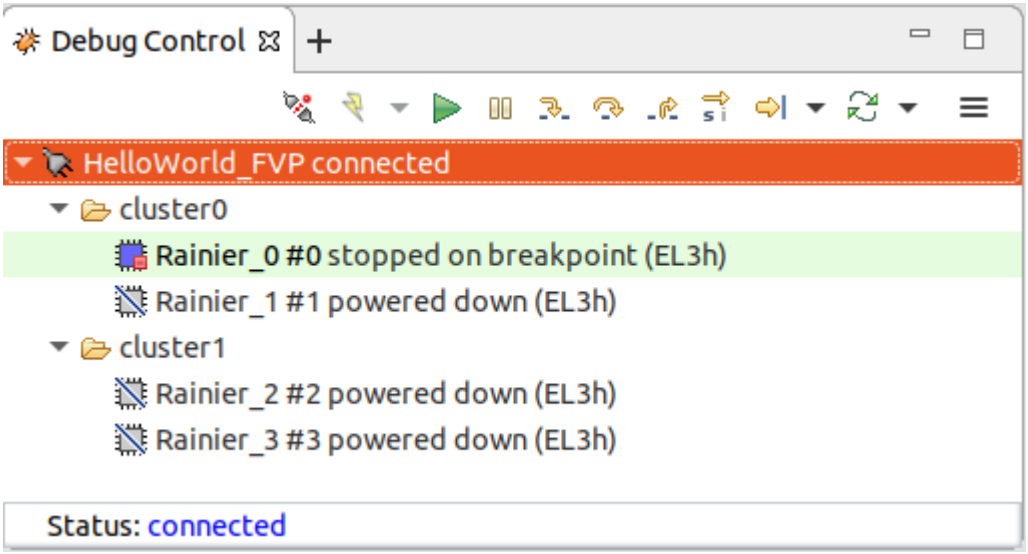
Figure 8-10 Memory view

After completing your debug activities, you can *disconnect the target* on page 8-102.

8.1.7 Disconnecting from a target

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

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



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

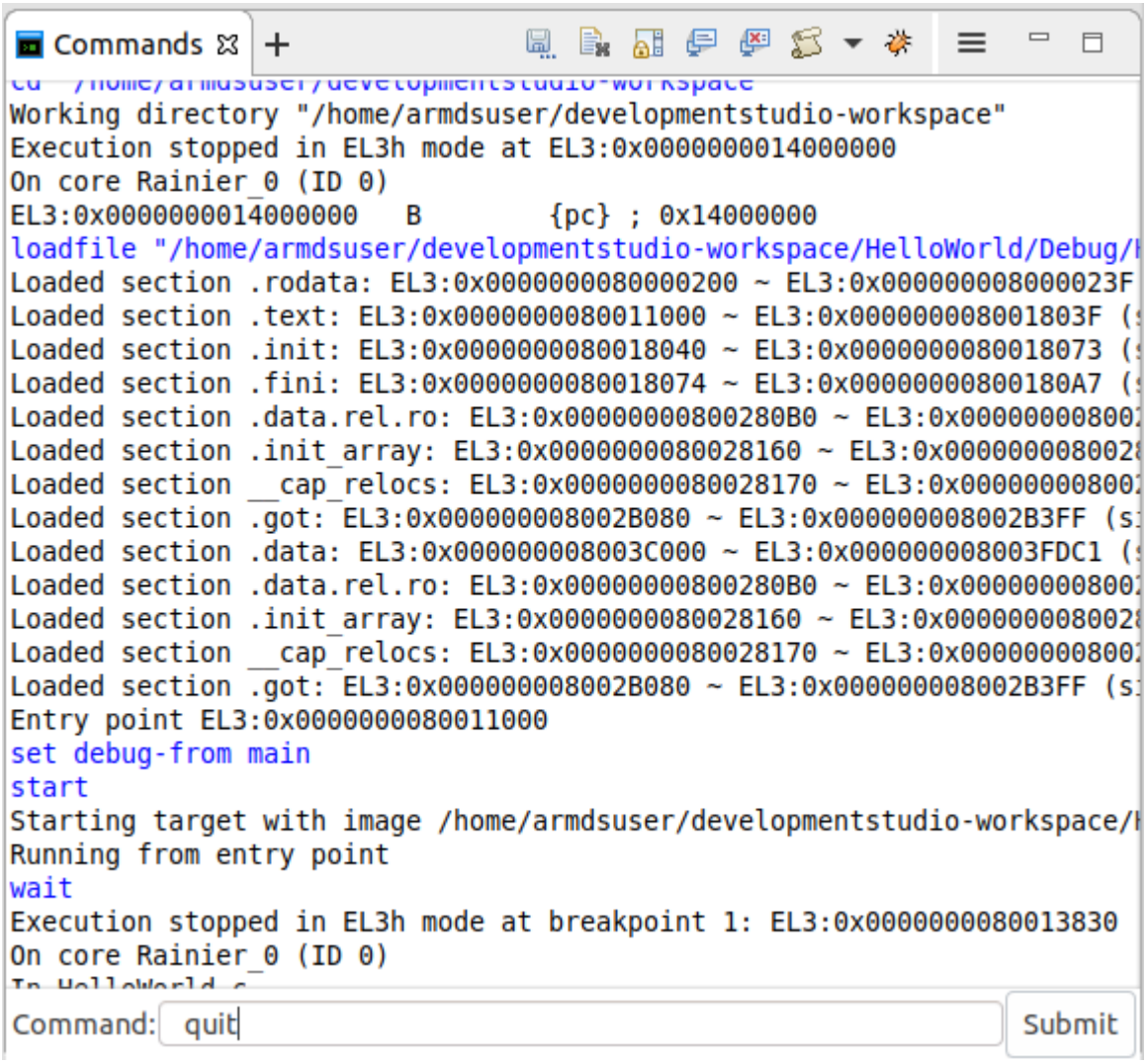


Figure 8-11 Disconnecting from a target using the Commands view

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.

Related tasks

8.1.1 Open Arm® Development Studio for the first time on page 8-93

8.1.2 Create a project in C/C++ on page 8-93

8.1.3 Configure your project on page 8-94

8.1.4 Build your project on page 8-95

8.1.5 Configure your debug session on page 8-95

Related references

8.1.6 Application debug with Arm Debugger on page 8-98

8.1.7 Disconnecting from a target on page 8-102

Chapter 9
Troubleshoot Arm® Development Studio Morello Edition

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

It contains the following sections:

9.1 Enabling internal logging from the debugger

On rare occasions an internal error might occur causing the debugger to generate an error message. You can help improve the debugger, by capturing the error message in an internal log file and sending it for analysis.

The log file captures the stacktrace and shows where in the debugger the error occured. You can send this feedback to the Arm Morello community forum.

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 within 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>`

Note

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

Related information

Commands view

Arm Morello community forum

9.2 Target connection problems and solutions

Lists possible problems when connecting to a target.

Failing to make a connection

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

- 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.

Additionally:

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