Arm® Development Studio

Version 2021.1

Heterogeneous system debug with Arm Development Studio

102021_2021.1_00_en

Arm® Development Studio

Heterogeneous system debug with Arm® Development Studio

Release Information

Document History

Issue	Date	Confidentiality	Change
2000-01	03 July 2020	Non-Confidential	New document for Arm Development Studio 2020.0 documentation update 1
2010-00	28 October 2020	Non-Confidential	Updated document for Arm Development Studio 2020.1
2021.0-00	19 March 2021	Non-Confidential	Updated document for Arm Development Studio 2021.0
2021.1-00	09 June 2021	Non-Confidential	Updated document for Arm Development Studio 2021.1

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

Progressive terminology commitment

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

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

If you find offensive terms in this document, please contact *terms@arm.com*.

Contents

Arm® Development Studio Heterogeneous system debug with Arm Development Studio

List of Figures
Arm® Development Studio Heterogeneous system debug with Arm Development Studio

Preface

This preface introduces the *Arm® Development Studio Heterogeneous system debug with Arm Development Studio*.

It contains the following:
- *About this book* on page 9.

About this book

This workbook describes how to set up and debug the NXP i.MX7 SABRE board development board with Arm® Development Studio. It takes you through the process of installing a Linux image, and then guides you through a debug session with bare-metal and Linux applications.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction to the workbook
We introduce the heterogenous debug on Arm Development Studio workbook.

Chapter 2 Set up your target for debug
Learn how to set up a target development board in preparation for running and debugging an application in Arm Development Studio. This includes connecting the hardware, installing the GCC compiler, and configuring debug connections.

Chapter 3 Debug an example project
Learn how to import an example project and run it on your development board.

Chapter 4 Debug applications on a heterogenous system
Arm Debugger allows the viewing of multiple simultaneous debug connections. Connecting a debug probe allows you to debug the Linux kernel and bare-metal applications running on the Cortex-A and the Cortex-M cores. You can also debug the Linux application using `gdbserver`.

Chapter 5 Store the Cortex-M image on an SD Card
Store the Cortex-M image on an SD card for execution at start-up.

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 Heterogeneous system debug with Arm Development Studio*.
- The number 102021_2021.1_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*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction to the workbook

We introduce the heterogenous debug on Arm Development Studio workbook.

It contains the following section:

1.1 Heterogenous debug on Arm Development Studio: Introduction

We show you how to set up a NXP i.MX7 SABRE development board and use it to debug a Linux image on Cortex®-A cores, and bare-metal applications on Cortex-M cores. We provide a pre-made CMSIS-Pack project, and a project with adaptable source code files for in-depth debug.

Throughout, we use the *NXP i.MX7 SABRE board*. You can also follow the steps if you have a Toradex Colibri, Embedded Artists Dual uCOM, Novtech/96Boards Meerkat, or Phytec Phyboard Zeta i.MX7 board.

Chapter 2
Set up your target for debug

Learn how to set up a target development board in preparation for running and debugging an application in Arm Development Studio. This includes connecting the hardware, installing the GCC compiler, and configuring debug connections.

It contains the following sections:

2.1 Install the Linux image

To debug a Linux application on the Cortex-A7 cores, Linux must be installed on to an SD card. This is inserted into your development board.

A pre-configured Linux image with Arm Development Studio-specific debug settings is available for supported development boards. The *Keil website* lists all supported development boards.

Prerequisites

You need an SD card with a minimum of 8GB of space.

Procedure

1. Download the compressed Linux image, `core-image-base-imx7dsabresd-20170720.rootfs.sdcard.zip`, from *Arm Developer* and unzip the file.
2. Copy the Linux image onto an SD Card:

Windows	Linux
1. Download and install a disk imager, for example **Win32 Disk Imager** from *http://win32diskimager.sourceforge.net/*, and run the program. 2. Select the image file named `core-image-base-imx7dsabresd-20170720.rootfs.sdcard`. Then, select the device letter of the SD card and click **Write**. **Warning** To avoid corrupting your existing data, ensure that you select the correct SD device.	Enter the following command, where `<path/to/sd>` is the path to your SD card: `sudo dd if=core-image-base-imx7dsabresd-20170720.rootfs.sdcard of=<path/to/sd> bs=1M` **Warning** To avoid corrupting your existing data, ensure that you select the correct SD device.

3. Check that your development board is switched off, and then insert the SD card into your development board.

Next Steps

Set up the hardware connections on page 2-15.

2.2 Set up the hardware connections

Set up several physical connections to enable full debug of your development board.

Prerequisites

You require:

- An Ethernet cable.
- A JTAG debug probe, such as ULINKpro™ or the DSTREAM-ST.
- A USB to micro-USB connector.

Procedure

- Connect your development board to your host PC with the following connections:
  - An Ethernet connection between your host PC and the development board. This connection is required to debug Linux applications using `gdbserver`.
  - You must connect your debug probe to the development board through a JTAG connector. Your debug probe must also be connected to the host PC using either:
    - A USB connection, for DSTREAM-ST or ULINKpro.
    - An Ethernet connection, for DSTREAM-ST only..
  - A UART port connection from your development board to host PC. Boards either have an RS-232 connector or a USB interface that the operating system recognizes as virtual COM ports. This is used to interact with the Linux console.

    Note

    This workbook uses the ULINKpro debug probe and USB to micro-USB connector alongside the NXP i.MX7 SABRE board.

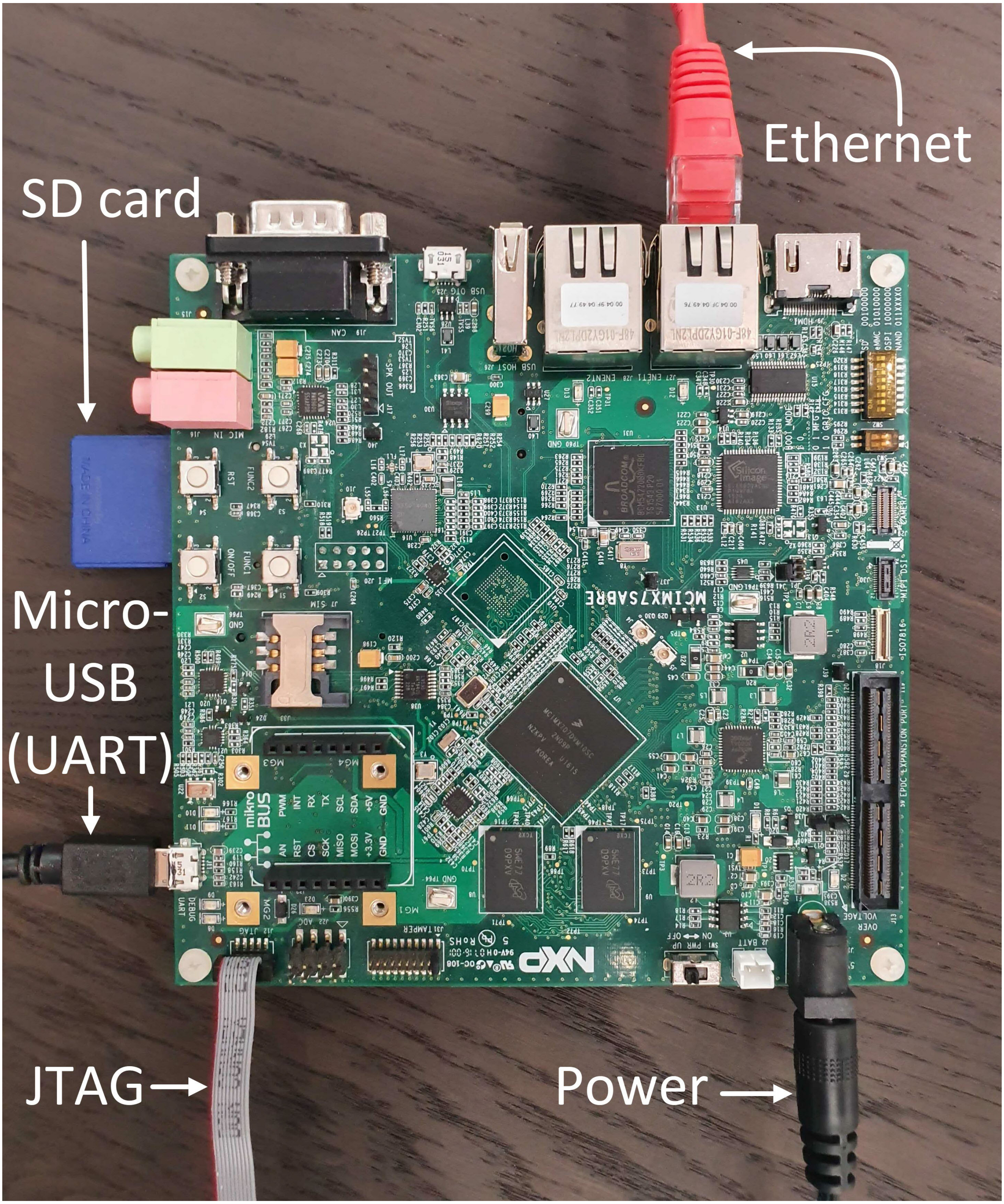


Figure 2-1 An NXP i.MX7 SABRE board connected with JTAG, Ethernet, and USB UART connections.

Note

If you are not sure how to connect your board, follow the instructions on the support page for the development board. For example, the *NXP i.MX7 SABRE* page.

Next Steps

Add the GCC compiler to Arm Development Studio on page 2-18.

Related information

Supported debug probes

2.3 Add the GCC compiler to Arm Development Studio

The projects in this workbook are pre-configured for the GCC 7.4.1 compiler. To build the projects, you must add the GCC 7.4.1 toolchain to Arm Development Studio.

Prerequisites

- Download and extract the *GCC 7.4.1* toolchain from the Linaro website.

  Note

  The project files require version 7.4.1 of GCC compiler.

  — For Windows, download `gcc-linaro-7.4.1-2019.02-i686-mingw32_arm-linux-gnueabihf.tar.xz`.
  — For Linux i686, download `gcc-linaro-7.4.1-2019.02-i686_arm-linux-gnueabihf.tar.xz`.
  — For Linux x86_64, download `gcc-linaro-7.4.1-2019.02-x86_64_arm-linux-gnueabihf.tar.xz`

Procedure

1. To open the **Preferences** dialog box, from the Arm Development Studio main menu, click **Window** > **Preferences**.
2. To open the **Add a new Toolchain** dialog box, select **Arm DS** > **Toolchains** from the sidebar, then click **Add…**.
3. Click **Browse…** and select the `bin` folder for the toolchain. Click **Next** to run autodetection.

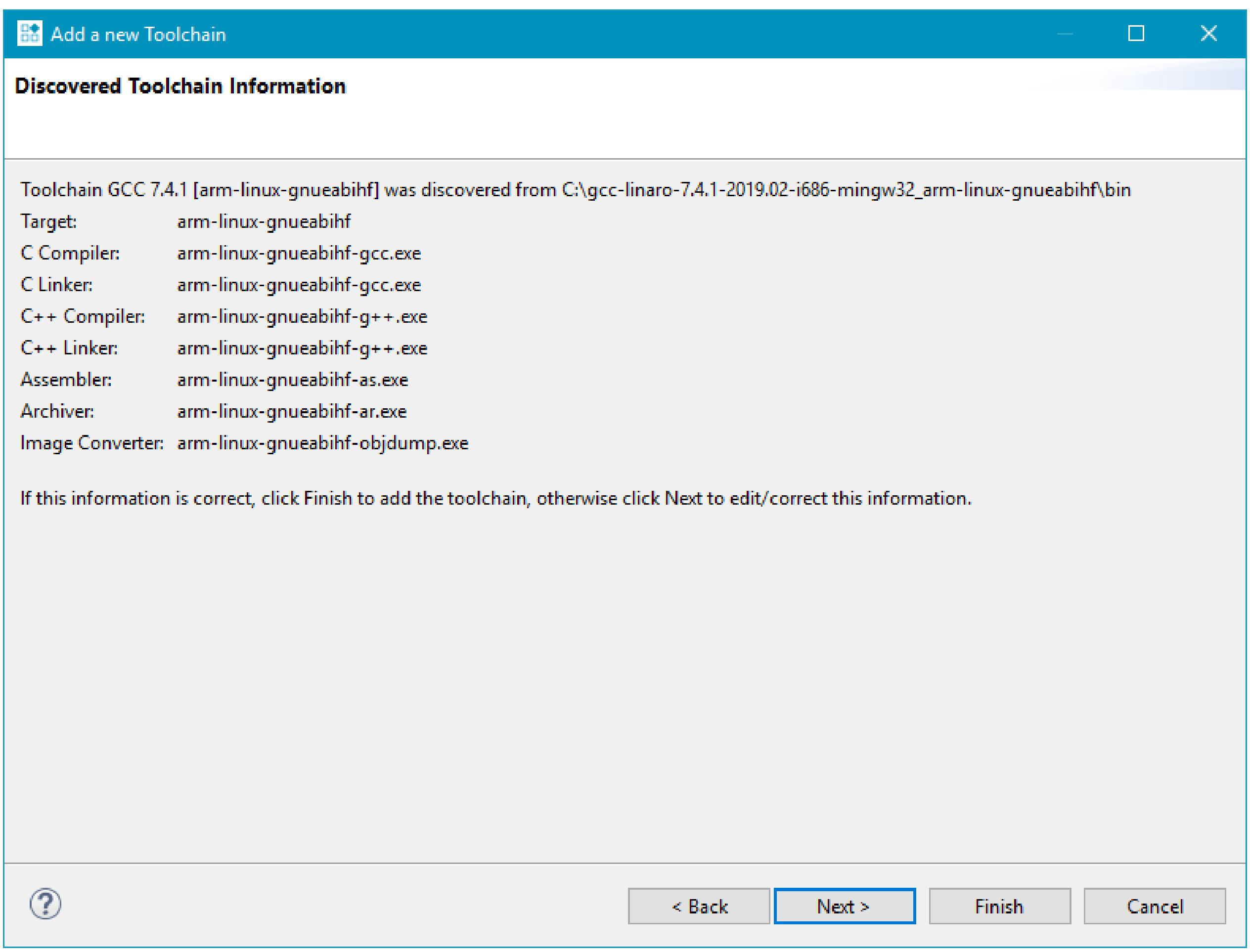


Figure 2-2 Autodetection results for the GCC 7.4.1 compiler.

4. Check that the toolchain was correctly autodetected, then click **Finish**.
5. In the **Preferences** dialog box, click **Apply** and restart Arm Development Studio.

Next Steps

Identify COM ports on the host PC on page 2-20.

2.4 Identify COM ports on the host PC

Identify and make note of the serial (COM) port numbers on your host PC. These COM port numbers are used later in the workbook, to configure the **Terminal** views in Arm Development Studio.

Prerequisites

Connect your host PC to the development board. For more information, see *Hardware Connection* on page 2-15.

Procedure

1. Identify the COM ports on your host PC:

   On Windows:

   1. Open the Windows **Device Manager**.
   2. Expand **Ports (COM & LPT)** to display the COM port numbers. The lower number is the COM port of the Cortex-A7 processor, while the higher number is the COM port of the Cortex-M4 processor.

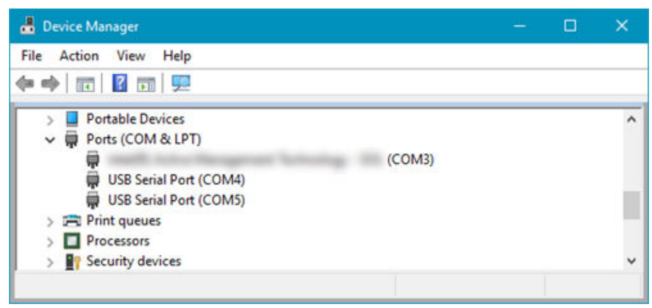


Figure 2-3 Device Manager, showing the two COM ports associated with the development board.

   On Linux:

   1. Navigate to your `/dev/` directory and identify the two new devices. The first device is the serial port of the Cortex-A7 processor, the second device is the serial port of the Cortex-M4 processor. If no other USB devices are connected to your host PC, the Cortex-A7 is `/dev/ttyUSB0` and the Cortex-M4 is `/dev/ttyUSB1`.

      Note

      If the new devices are not shown, your host PC has not identified the development board.

   2. Allow read/write permission to the development board. For example, to give read/write permissions to all users on your host PC, run the following command:

      ```
      sudo chmod 666 /dev/ttyUSB
      ```

2. Make a note of these COM port numbers.

Next Steps

These COM port numbers are needed to *configure the Terminal views* on page 2-21.

2.5 Configure the Terminal views

In Arm Development Studio, the **Terminal** view displays messages from your development board. You must configure the **Terminal** view for each COM port on your development board.

Prerequisites

Identify COM port numbers on your host PC on page 2-20.

Procedure

1. To open the **Terminal** view, click **Window** > **Show View** > **Terminal** from the Arm Development Studio main menu.
2. To display the output of the Cortex-A processor:

   a. In the **Terminal** view toolbar, click to open the **Launch Terminal** dialog box.
   b. Edit the following fields, and then click **OK**:
      - **Choose terminal:** Serial Terminal
      - **Port:** Use the first of the new serial ports (for example, COM4 or `/dev/ttyUSB0`)
      - **Baud Rate:** 115200

   Do not change the values of other settings.

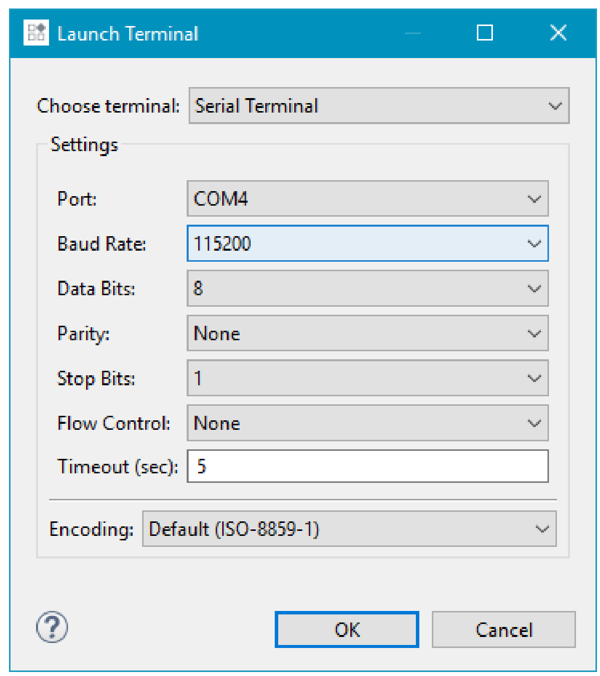


Figure 2-4 Launch Terminal dialog box, showing settings for the Cortex-A Terminal view.

Note

These settings are specific to the NXP i.MX7 SABRE board. For the correct terminal settings for your development board, refer to the support pages of the development board.

---

3. To display the output of the Cortex-M processor, we need to configure another instance of the **Terminal** view:

   a. In the **Terminal** view toolbar, click to open another **Launch Terminal** dialog box.
   b. Select the second serial port number (for example, COM5 or `/dev/ttyUSB1`), and use the same settings as the previous terminal.

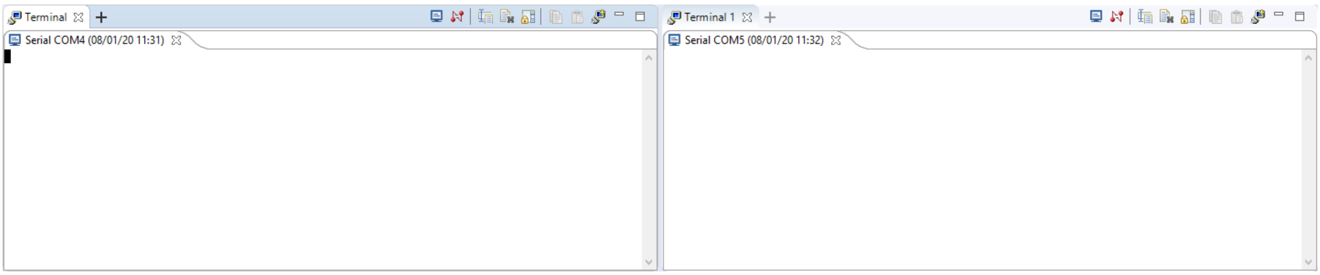


Figure 2-5 Terminal views for Cortex-A and Cortex-M.

Next Steps

Determine the IP address of your development board on page 2-23.

2.6 Determine the IP address of your development board

The IP address of your development board is required to set up a Remote System Explorer (RSE) connection. This allows you to debug using `gdbserver`.

Prerequisites

- Connect to the development board using an Ethernet connection. For more information, see *Set up the hardware connections* on page 2-15.
- *Configure the Terminal views* on page 2-21.

Procedure

1. Power up your development board and observe the Linux boot process on the Cortex-A7 **Terminal** view.
2. Log into Linux with the username `root`. No password is required.

   Note

   Your credentials might be different if you are not using the image downloaded from *Arm Developer*.

3. Enter `ifconfig` into the Cortex-A **Terminal** view.
4. Make a note of the IP address of your board. It is shown at `eth0`, under `inet addr:`.

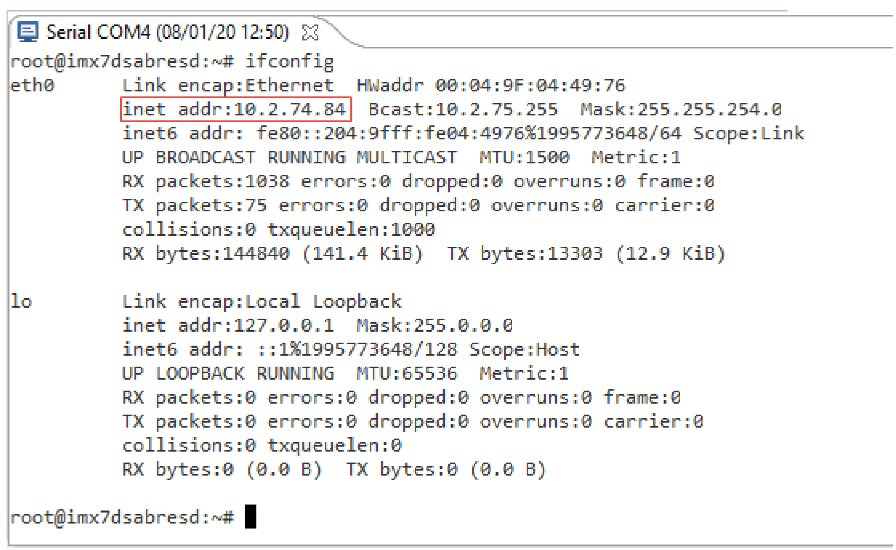


Figure 2-6 IP address, displayed by running ifconfig.

Next Steps

Set up the Remote System Explorer connection on page 2-24.

2.7 Set up a Remote System Explorer connection

The Remote System Explorer (RSE) is an interface for managing the development board using TCP/IP. This topic describes how to set up an RSE connection for use by Arm Debugger.

Prerequisites

- *Determine the IP address of your development board* on page 2-23.
- Connect to the development board with an Ethernet connection. For more information, see *Set up the hardware connections* on page 2-15.

Procedure

1. From the Arm Development Studio main menu, open **Window** > **Show View** > **Other...**, expand the **Remote Systems** folder then select **Remote Systems**. Click **OK**. The **Remote Systems** view opens.
2. To open the **New Connection** wizard, click in the **Remote Systems** view toolbar.
3. Select **SSH Only** and click **Next**.
4. Enter the IP address of the target into the **Host Name** field, and enter a name of your choice in the **Connection name** box.
5. To show your connection in the **Remote Systems** window, click **Finish**.

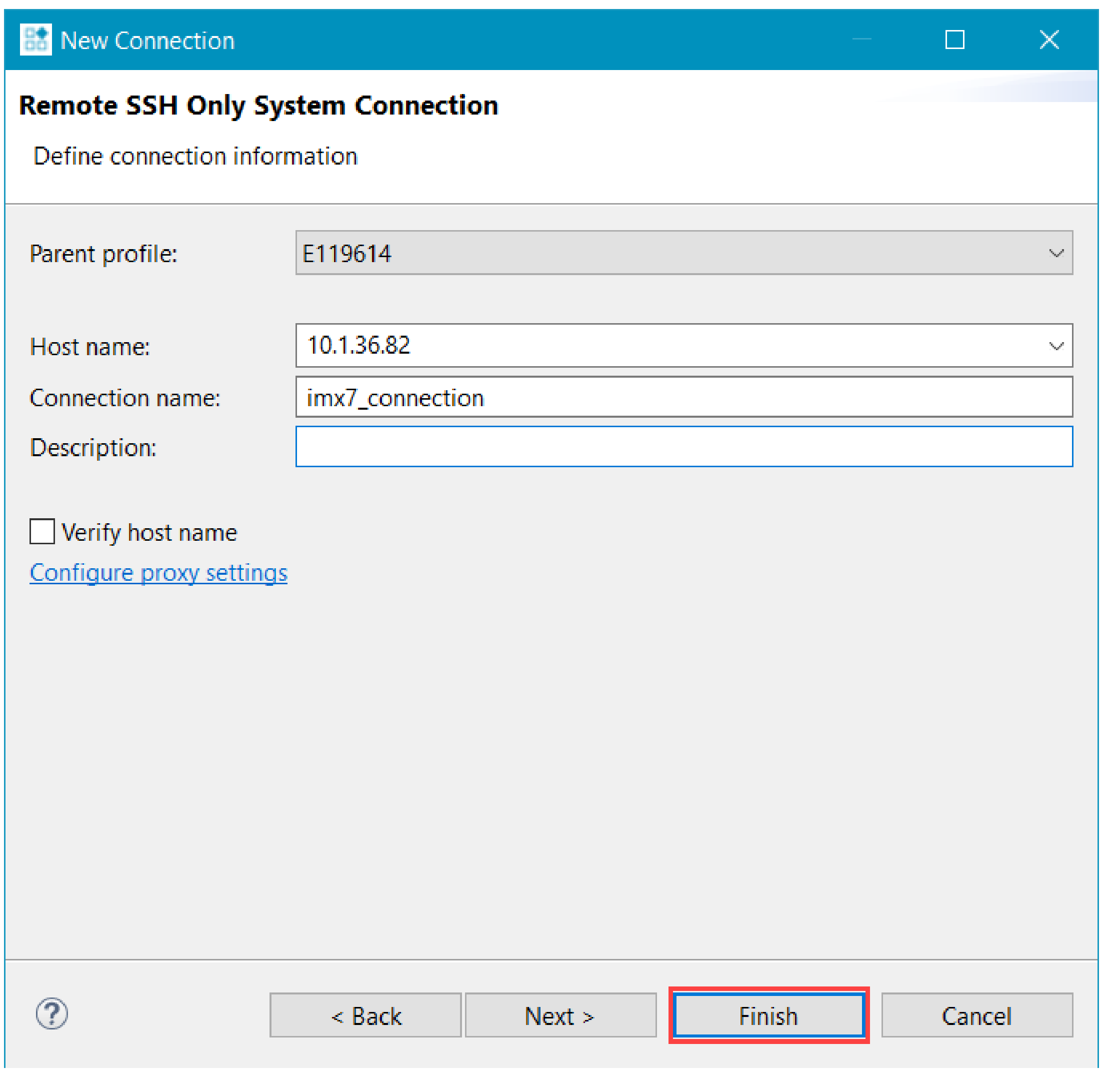


Figure 2-7 New Connection wizard, showing settings for an SSH Only RSE connection.

Next Steps

Learn how to set up Arm Debugger for an example project in *Debug an example project* on page 3-26. Alternatively, skip straight to an in-depth debug of a source code file in *Debug applications on a heterogenous system* on page 4-35.

Chapter 3
Debug an example project

Learn how to import an example project and run it on your development board.

It contains the following sections:

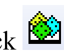
3.1 Set up the Cortex-M application

Setting up the project in Arm Development Studio allows you to connect to the Cortex-M application.

Prerequisites

Set up Arm Development Studio and your development board by following *Set up your target for debug* on page 2-13.

Procedure

1. To open the **CMSIS Pack Manager** perspective, click  in the top-right corner of Arm Development Studio.
2. Click the **Boards** tab and search for your board. In this example, we are using `MCIMX7D-SABRE`.
3. Click the **Examples** tab on the right-hand side.
4. Click **Install** next to the **RPMSG TTY CMSIS-RTOS2** example. If the example does not show, clear the **Only show examples from installed packs** check box.

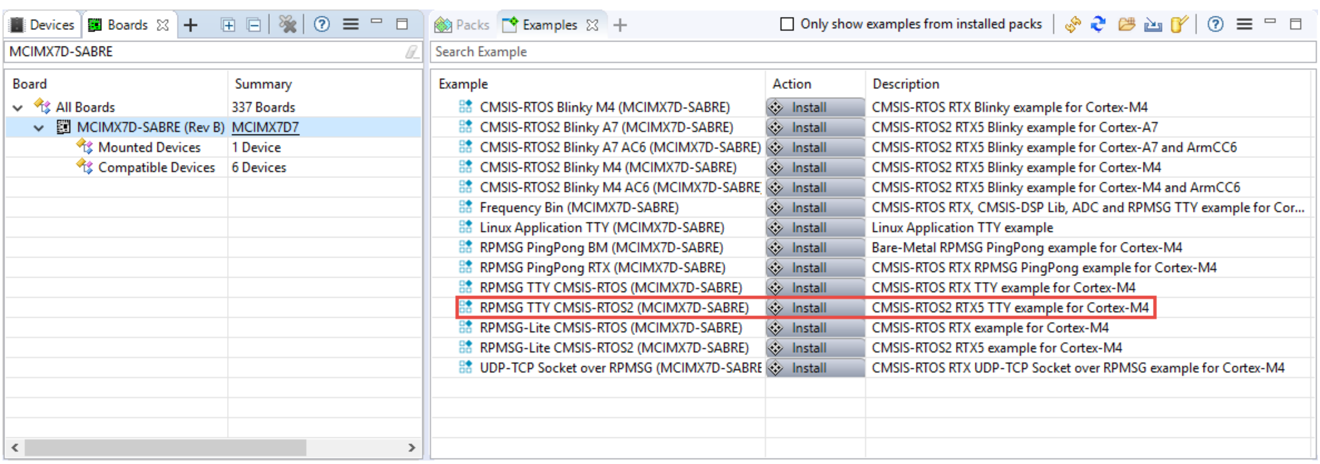


Figure 3-1 Installation of Cortex-M example project.

5. After the installation is complete, click **Copy** to copy the example into your workspace. Arm Development Studio automatically switches to the **Development Studio** perspective.

——— Note ———

If the example requires packs that are not installed, open the **Packs** tab, then click **Resolve Missing Packs** to download and install all required packs.

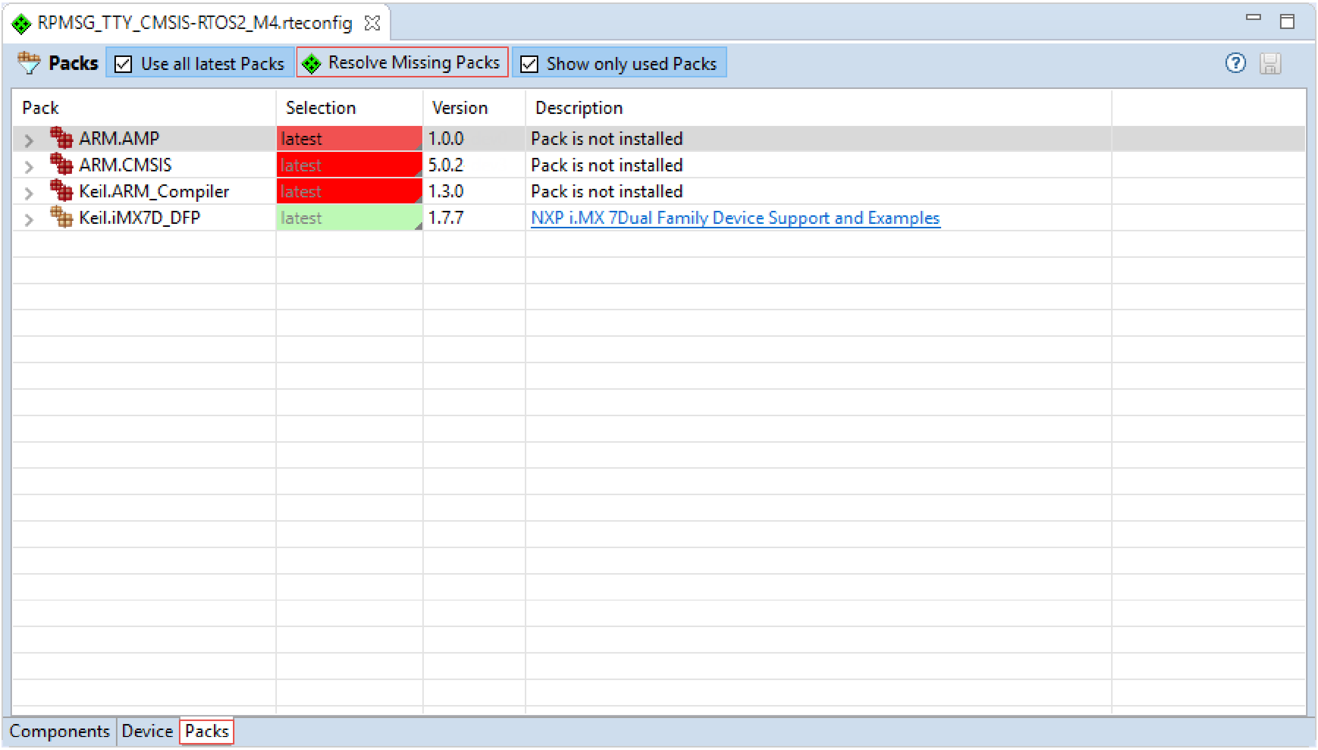


Figure 3-2 Installation of missing packs.

---

6. To build the project, select it in the **Project Explorer** view and click  .

The **Console** view shows the build result:

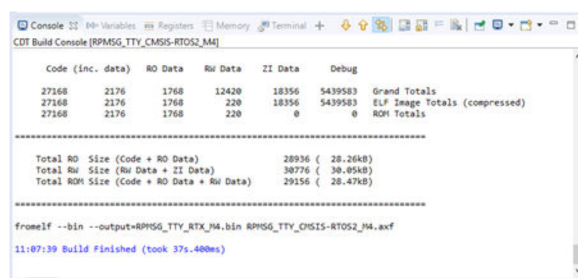


Figure 3-3 Console output of the Cortex-M build result.

Next Steps

Configure Arm Debugger for Cortex-M on page 3-29.

3.2 Configure Arm Debugger for Cortex-M

Now that we have configured the **Terminal** views and created the Cortex-M application, we need to configure Arm Debugger so that it can debug the Cortex-M application.

Prerequisites

Set up the Cortex-M application on page 3-27.

Procedure

1. Turn on your development board. The **Terminal** window of the Cortex-A7 displays the Linux boot process. Interrupt the boot process within a few seconds by pressing any key on your keyboard.

   ——— **Warning** ———

   You must interrupt the boot process at this point to connect the debug probe to the Cortex-M4 processor. If you do not stop the boot process in time, reset the board and repeat this step.

   ———————————

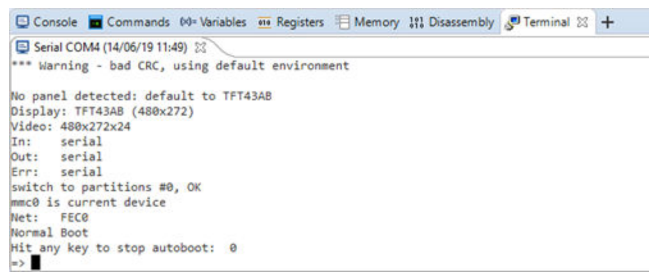


Figure 3-4 Terminal view showing the boot of Linux.

2. Right-click the **RPMSG_TTY_RTX_M4** project in the **Project Explorer** view and select **Debug As** > **Debug Configurations...** to open the **Debug Configurations** dialog box.
3. From the sidebar, select **CMSIS C/C++ Application** > **RPMSG_TTY_CMSIS-RTOS2_M4**.
4. Verify that your debug probe is correctly detected under **Connection Type** and **Connection Address**. From the **Debug Port** drop-down menu, select **JTAG**.

   ——— **Note** ———

   If your debug probe is not detected, click **Browse…** to list the available debug probes. All debug probes are listed in the drop-down menu for the **Connection Type**.

   ———————————

5. To set up the trace settings, click **Target Configuration…**.

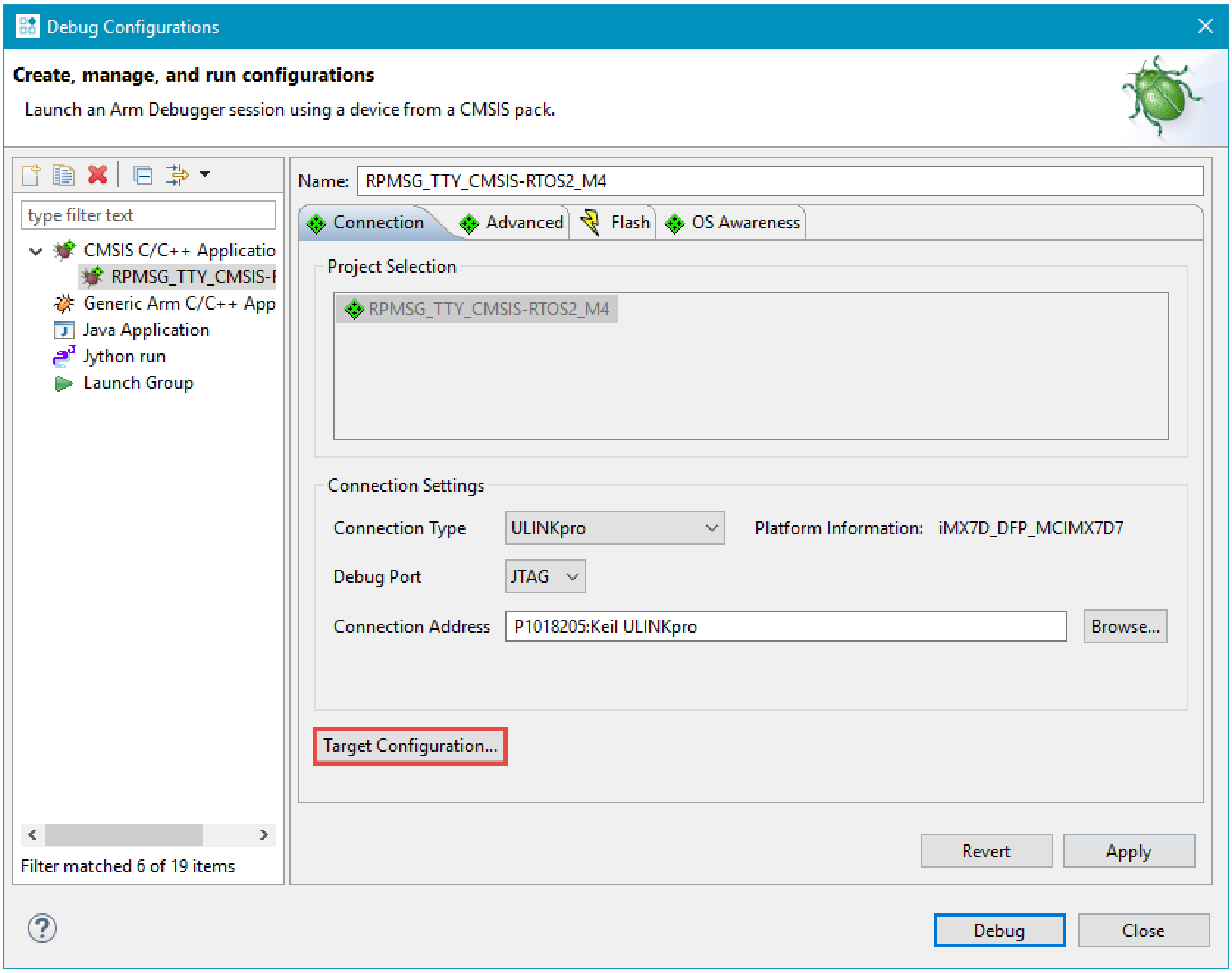


Figure 3-5 Debug configurations for Cortex-M, and Target Configuration... button.

a. Click the **Cortex-M4** tab and select **Enable Cortex-M4 core trace**.
b. Click the **Cortex-A7** tab and disable all trace options to avoid buffer overflows.
c. Click **OK** to confirm your changes and return to the **Debug Configurations** dialog box.

6. Click the **OS Awareness** tab and, from the drop-down menu, select **Keil CMSIS-RTOS RTX**.
7. Click **Debug**. Arm Development Studio switches to the **Development Studio** perspective. The application loads and runs until `main()`.

Note

If you see the error message `Failed to launch debug server`, this might indicate:
- An incorrect debug probe connection address is selected.
- The Linux boot process was not interrupted.

8. To start the Cortex-M4 application, click **Run** in the **Debug Control** view.

Observe the output of the application in the **Terminal** window of the Cortex-M4.

Figure 3-6 The output of the Cortex-M4 Terminal window.

Next Steps

Set up the Cortex-A Linux application on page 3-32.

3.3 Set up the Cortex-A Linux application

Copying and building the **Linux Application TTY** project allows you to set up the Cortex-A Linux application.

Prerequisites

- Complete *Configure Arm® Debugger for Cortex-M* on page 3-29.
- Add the GCC compiler to Arm Development Studio. For more information, see *Add the GCC compiler to Arm Development Studio* on page 2-18.

Procedure

1. To open the **CMSIS Pack Manager** perspective, click in the top-right corner of Arm Development Studio.
2. In the **Examples** tab, locate the **Linux Application TTY** example project and copy it to your workspace by clicking **Copy**. Confirm by clicking **Copy**.
3. To return to the **Development Studio** perspective, click in the top-right corner of Arm Development Studio.
4. To build the project, select it in the **Project Explorer** view and click .

The **Console** view shows the result of the build.

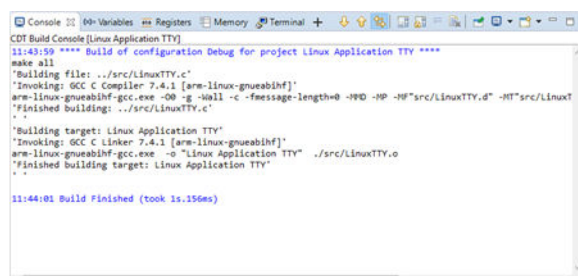


Figure 3-7 The console output showing the Cortex-A build result.

Next Steps

Configure Arm Debugger for Linux application debug on page 3-33.

3.4 Configure Arm Debugger for Linux application debug

Now the Cortex-A Linux application is built and the Cortex-M application is running, you must configure Arm Debugger to debug the Linux application.

Prerequisites

- *Set up the Cortex-A Linux application* on page 3-32.
- *Ensure you have set up an RSE connection* on page 2-24.

Procedure

1. In the Cortex-A **Terminal** view, enter `boot` to restart the Linux kernel boot process.
2. In the **Project Explorer** view, right-click on the **Linux Application TTY** project and select **Debug As** > **Debug Configurations...**.
3. Select **Generic Arm C/C++ Application** > **Linux Application TTY** from the sidebar.
4. In the **Connection** tab, select **Linux Application Debug** > **Application Debug** > **Connections via gdbserver** > **Download and debug application**.
5. Under **Connections**, select your new RSE connection from the drop-down menu.

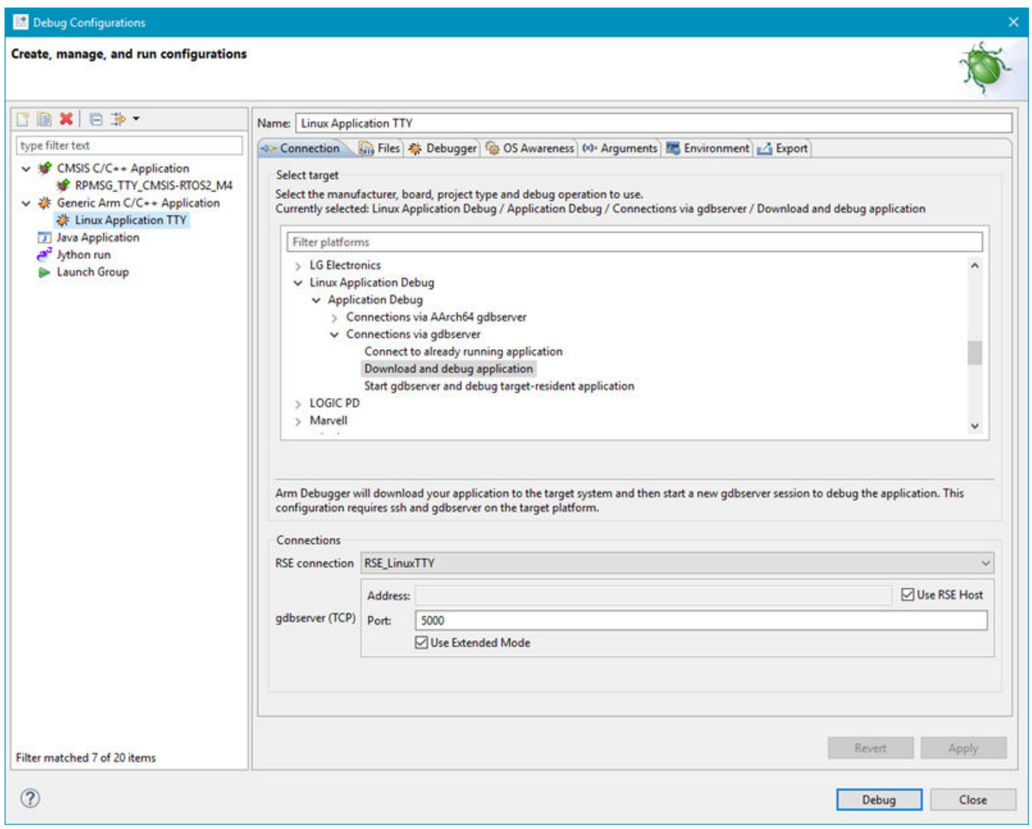


Figure 3-8 Debug Configurations dialog box.

6. Click the **Files** tab. Under **Target Configuration**:
   a. Select the workspace build target for the **Application on host to download**.
   b. Select an existing directory on the target file system, for example `/home/root/tmp/` as the **Target download directory**.
7. Click the **Debugger** tab. Under **Run Control**, select **Debug from symbol "main"**. Click **Debug**.

Note

If you are asked to log in, enter `root` as the username and leave the password field empty. Your credentials might be different if you are not using the image downloaded from *Arm Developer*.

If you are unable to connect to the Linux application TTY, generate new SSH keys for Secure Shell authentication by entering the following commands in the Cortex-A Linux **Terminal** view:

1. `ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key`
2. `ssh-keygen -t dsa -f /etc/ssh/ssh_host_rsa_key`
3. `ssh-keygen -t ecdsa -f /etc/ssh/ssh_host_ecdsa_key`
4. `ssh-keygen -t ed25519 -f /etc/ssh/ssh_host_ed25519_key/usr/sbin/sshd`

---

8. In the Cortex-A Linux **Terminal** view, update the `.dbt` file version by entering the following commands:
   a. `setenv fdt_file imx7d-sdb-m4.dtb; saveenv;`
   b. `setenv mmcargs 'setenv bootargs console=${console},${baudrate} root=${mmcroot} clk_ignore_unused'; saveenv;`
9. In the Cortex-A Linux **Terminal** view, load the kernel module that communicates with the Cortex-M4 application with this command:

   `modprobe -v imx_rpmsg_tty`

   Linux loads the kernel module, and displays the following output:

```
insmod /lib/modules/4.1.151.1.0+ga4d2a08/kernel/drivers/rpmsg/imx_rpmsg_tty.ko
imx_rpmsg_tty rpmsg0: new channel: 0x400 -> 0x0!
Install rpmsg tty driver!
```

10. You can now run the Cortex-A Linux application. Click in the **Debug Control** view.

The **App Console** view shows the messages output by the Linux application:

`Hello from M4!`

The **Terminal** for the Cortex-M4 shows the output of the microcontroller application:

`Hello from A7!`

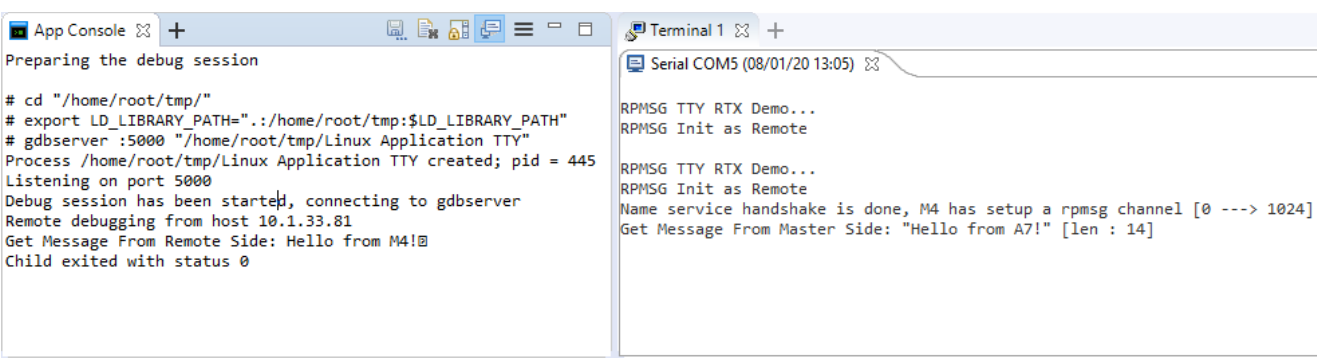


Figure 3-9 Output of the App Console and Cortex-M Terminal.

You have verified that your development environment can connect to both the Cortex-M and Cortex-A processors, and that the two example applications run successfully and communicate with each other.

Next Steps

To learn about in-depth debugging of source code, try *Debug applications on a heterogenous system* on page 4-35, or to save your current image, go to *Store the Cortex-M image on an SD Card* on page 5-61.

Chapter 4
Debug applications on a heterogenous system

Arm Debugger allows the viewing of multiple simultaneous debug connections. Connecting a debug probe allows you to debug the Linux kernel and bare-metal applications running on the Cortex-A and the Cortex-M cores. You can also debug the Linux application using `gdbserver`.

It contains the following sections:

4.1 Build and debug the Cortex-M application

Debug bare-metal applications running on the Cortex-M with Arm Debugger.

This section contains the following subsections:

4.1.1 Create a Cortex-M application

For this project, the Cortex-M application uses a CMSIS C project type. In the Cortex-M application, you configure the RTX RTOS and its associated software components.

Prerequisites

You can create and debug an example project by following the steps in *Debug an example project* on page 3-26. Otherwise, set up Arm Development Studio and your development board by following *Set up your target for debug* on page 2-13.

Procedure

1. Set up the project:
   a. From the Arm Development Studio menu bar, choose **File** > **New** > **Project...**
   b. Select **C/C++** > **C Project** and click **Next**.
   c. Under **Project type**, select **Executable** > **CMSIS C/C++ Project**.
   d. Under **toolchains**, select **Arm Compiler 6**.
   e. Enter the project name **Blinky** and click **Next**.
   f. Select your development board from the list. In this example, we use the NXP i.MX 7 Sabre board, so select **MCIMX7D7:Cortex-M4**.
   g. In the **FPU** drop-down menu, select **None**. Click **Finish**

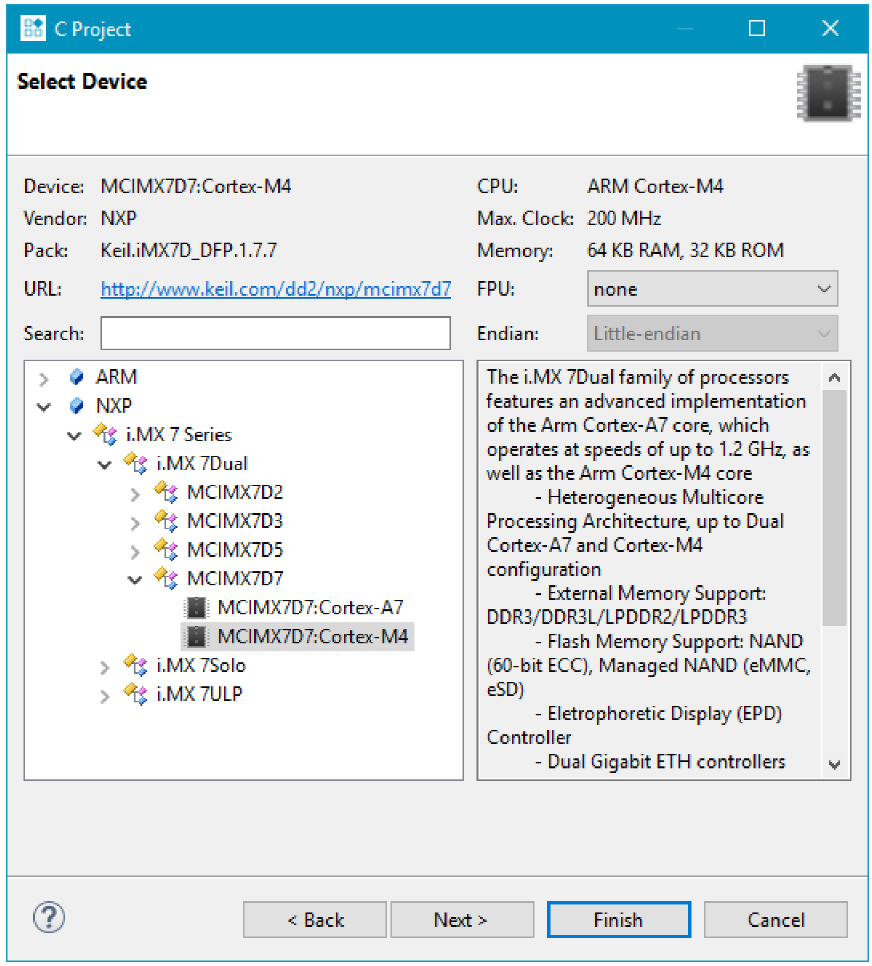
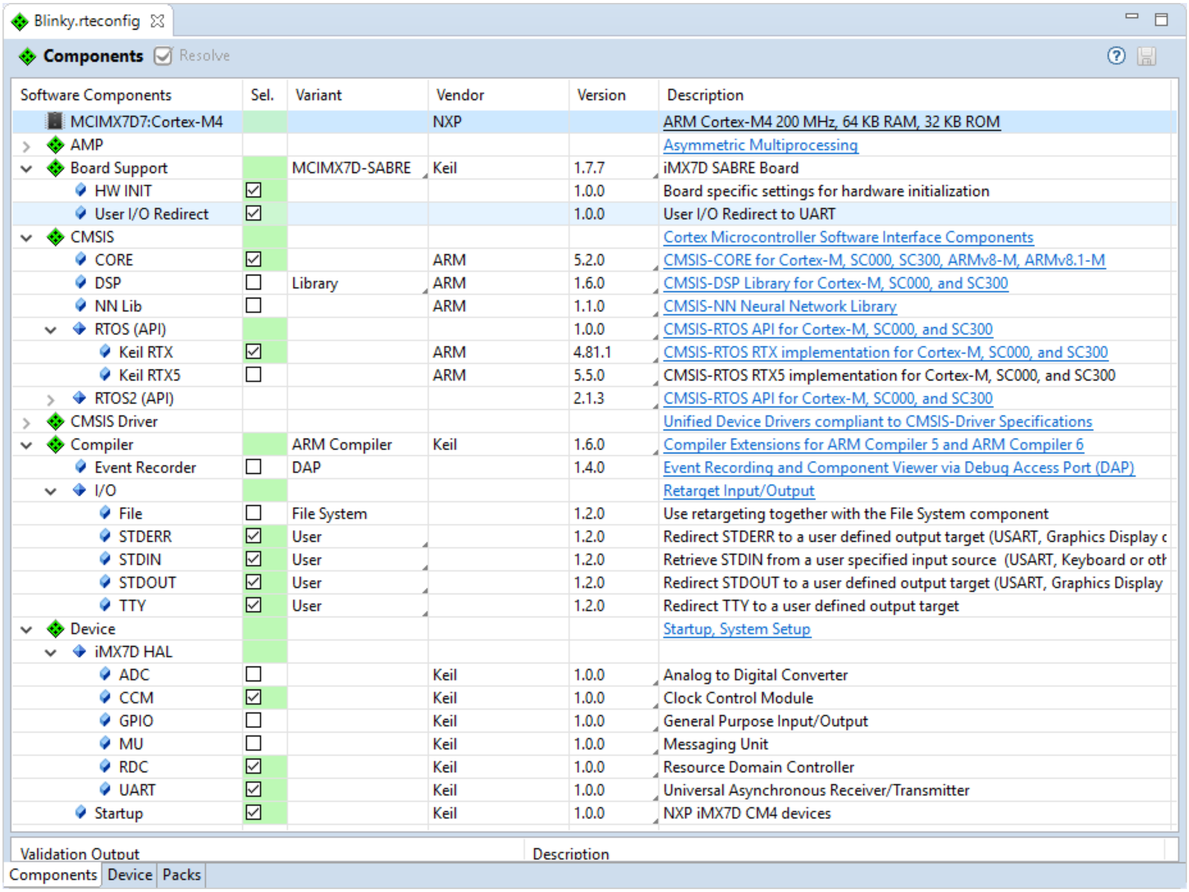


Figure 4-1 C project dialog box showing settings for the i.MX7 Sabre development board.

2. Select software components in the **Components** tab:
   - Under **Board Support**, select **iMX7D-SABRE** as the **Variant**, and select **HW INIT** and **User I/O Redirect** check boxes.
   - Under **CMSIS**, enable **CORE**, and under **CMSIS** > **RTOS (API)**, enable **Keil RTX**.
   - Under **Compiler** > **I/O**, change the **Variant** for **STDERR**, **STDIN**, **STDOUT**, and **TTY** to **User**, then enable each of these components.
   - Under **Device**, enable **Startup**, and under **Device** > **i.MX7D HAL**, enable **CCM**, **RDC**, and **UART**.



Software Components	Sel.	Variant	Vendor	Version	Description
MCIMX7D7:Cortex-M4			NXP		ARM Cortex-M4 200 MHz, 64 KB RAM, 32 KB ROM
AMP					Asymmetric Multiprocessing
Board Support		MCIMX7D-SABRE	Keil	1.7.7	iMX7D SABRE Board
HW INIT	☑			1.0.0	Board specific settings for hardware initialization
User I/O Redirect	☑			1.0.0	User I/O Redirect to UART
CMSIS					Cortex Microcontroller Software Interface Components
CORE	☑		ARM	5.2.0	CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M, ARMv8.1-M
DSP	☐	Library	ARM	1.6.0	CMSIS-DSP Library for Cortex-M, SC000, and SC300
NN Lib	☐		ARM	1.1.0	CMSIS-NN Neural Network Library
RTOS (API)				1.0.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
Keil RTX	☑		ARM	4.81.1	CMSIS-RTOS RTX implementation for Cortex-M, SC000, and SC300
Keil RTX5	☐		ARM	5.5.0	CMSIS-RTOS RTX5 implementation for Cortex-M, SC000, and SC300
RTOS2 (API)				2.1.3	CMSIS-RTOS API for Cortex-M, SC000, and SC300
CMSIS Driver					Unified Device Drivers compliant to CMSIS-Driver Specifications
Compiler		ARM Compiler	Keil	1.6.0	Compiler Extensions for ARM Compiler 5 and ARM Compiler 6
Event Recorder	☐	DAP		1.4.0	Event Recording and Component Viewer via Debug Access Port (DAP)
I/O					Retarget Input/Output
File	☐	File System		1.2.0	Use retargeting together with the File System component
STDERR	☑	User		1.2.0	Redirect STDERR to a user defined output target (USART, Graphics Display c
STDIN	☑	User		1.2.0	Retrieve STDIN from a user specified input source (USART, Keyboard or oth
STDOUT	☑	User		1.2.0	Redirect STDOUT to a user defined output target (USART, Graphics Display
TTY	☑	User		1.2.0	Redirect TTY to a user defined output target
Device					Startup, System Setup
iMX7D HAL					
ADC	☐		Keil	1.0.0	Analog to Digital Converter
CCM	☑		Keil	1.0.0	Clock Control Module
GPIO	☐		Keil	1.0.0	General Purpose Input/Output
MU	☐		Keil	1.0.0	Messaging Unit
RDC	☑		Keil	1.0.0	Resource Domain Controller
UART	☑		Keil	1.0.0	Universal Asynchronous Receiver/Transmitter
Startup	☑		Keil	1.0.0	NXP iMX7D CM4 devices

Figure 4-2 The selected components for the CMIMX7D7:Cortex-M4.

3. To add these components, click .
4. Configure the CMSIS-RTOS RTX kernel:
   a. In the **Project Explorer** view, expand **Blinky** > **RTE** > **CMSIS**, right-click on the file `RTX_Conf_CM.c`, and select **Open With** > **CMSIS Configuration Wizard**. Enter the following values:
      - Under **Thread Configuration**:
        — **Default Thread stack size [bytes]**: 512
        — **Main Thread stack size [bytes]**: 512
      - Under **RTX Kernel Timer input clock frequency [Hz]**:
        — **RTOS Kernel Timer input clock frequency [Hz]**: 240000000
   b. To save your changes, press **Ctrl** + **S**.

Next Steps

Create the source code files for your project on page 4-38.

4.1.2 Create the source code files

Arm Development Studio provides pre-configured code templates. You can use these templates to create source code files for your project.

Prerequisites

Create the Cortex-M application, see *Create a Cortex-M application* on page 4-36.

Procedure

1. In the **Project Explorer** view, right-click on the **Blinky** project and select **New** > **Files from CMSIS Template**.
2. Under **CMSIS**, select the **CMSIS-RTOS 'main' function** template, and then click **Finish**.
3. Replace the content of the new `main.c` file with this application-specific code:

```
/*----------------------------------------------------------------------
 * CMSIS-RTOS 'main' function template
 *---------------------------------------------------------------------*/

#define osObjectsPublic                     // define objects in main module
#include "osObjects.h"                      // RTOS object definitions

#ifdef _RTE_
  #include "RTE_Components.h"               // Component selection
#endif
#ifdef RTE_CMSIS_RTOS              // when RTE component CMSIS RTOS is used
  #include "cmsis_os.h"                     // CMSIS RTOS header file
#endif
#include "system_iMX7D_M4.h"
#include "retarget_io_user.h"
#include "board.h"
#include <stdio.h>

osThreadId tid_threadA;                       /* Thread id of thread A */

/*----------------------------------------------------------------------
 *      Thread A
 *---------------------------------------------------------------------*/
void threadA (void  const *argument) {
  volatile int a = 0;
  for (;;) {
      osDelay(750);
      printf("Blinky   threadA: Hello World!\n");
  }
}

osThreadDef(threadA,  osPriorityNormal, 1, 0);

/*
 * main: initialize and start the system
 */
int main (void) {
  /* Board specific RDC settings */
  BOARD_RdcInit();

  /* Board specific clock settings */
  BOARD_ClockInit();

  SystemCoreClockUpdate();
  InitRetargetIOUSART();

  tid_threadA = osThreadCreate(osThread(threadA), NULL);


#ifdef RTE_CMSIS_RTOS                   // when using CMSIS RTOS
  osKernelInitialize ();                // initialize CMSIS-RTOS
#endif

  /* Initialize device HAL here */

#ifdef RTE_CMSIS_RTOS                   // when using CMSIS RTOS
  osKernelStart ();                     // start thread execution
#endif

  /* Infinite loop */
  while (1)
  {
    /* Add application code here */
    osDelay(1000);
    printf("Blinky main loop: Hello World!\n");

  // initialize peripherals here

  // create 'thread' functions that start executing,
  // example: tid_name = osThreadCreate (osThread(name), NULL);

  osKernelStart ();                         // start thread execution
  }
}
```

4. To save your changes, press **Ctrl** + **S**.

Next Steps

Adapt the scatter file on page 4-40.

4.1.3 Adapt the scatter file

Edit the scatter file to insert your Cortex-M code into the Tightly Coupled Memory (TCM) of the development board.

On the i.MX 7 devices, several types of memory are available. For deterministic, real-time behavior, the Cortex-M4 must use the local TCM, that provides low-latency access. Multiple on-chip RAM areas (OCRAM) are available, but they are larger and not as fast.

The following table shows the memory regions and their load addresses for the different processors. By default, the scatter file template uses the start address `0x0` for the load region command.

Region	Size	Cortex-A7	Cortex-M4 (Code Bus)
OCRAM	128 KB	`0x00900000-0x0091FFFF`	`0x00900000-0x0091FFFF`
TCMU	32 KB	`0x00800000-0x00807FFF`	-
TCML	32 KB	`0x007F8000-0x007FFFFF`	`0x1FFF8000-0x1FFFFFFF`
OCRAM_S	32 KB	`0x00180000-0x00187FFF`	`0x00000000-0x00007FFF/0x00180000-0x00187FFF`

Prerequisites

Create the source code files, see *Create the source code files* on page 4-38.

Procedure

1. To put the Cortex-M4 code into the TCM of the i.MX 7:
   a. Open the **MCIMX7D7.sct** file from the **Project Explorer** view.
   b. Change the address of the load region `LR_IROM1` and load address `ER_IROM1` to `0x1FFF8000`:

```
; *************************************************************
; ** Scatter-Loading Description File generated by RTE CMSIS Plug-in **
; *************************************************************

LR_IROM1 0x1FFF8000 0x00008000 {    ; load region size_region
  ER_IROM1 0x1FFF8000 0x00008000 {  ; load address = execution address
   *.o (RESET, +First)
   .ANY (+RO)
  }
  RW_IRAM1 0x20000000 0x00008000 {
   .ANY (+RW +ZI)
  }
}
```

   c. To save your changes, press **Ctrl** + **S**.
2. To build the Cortex-M image, in the **Project Explorer** view, right-click on the **Blinky** project, and click **Build Project**.

Building the project compiles and links all related source files. The **Console** view shows information about the build process.

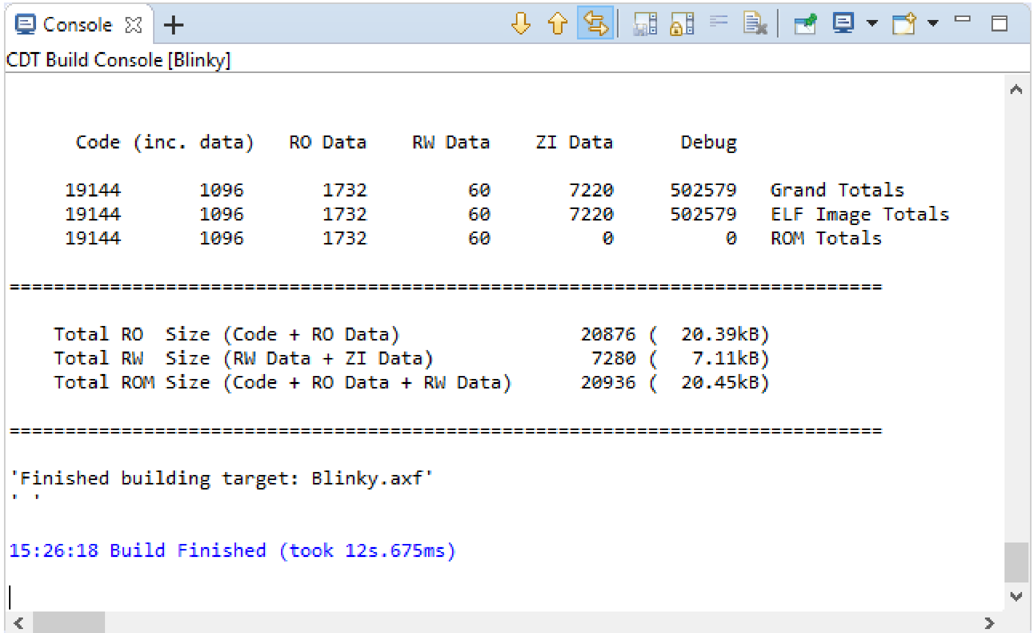


Figure 4-3 Console output showing the build results.

Next Steps

Create the Linux application for this project on page 4-43.

4.1.4 Debug the Cortex-M Blinky application

Configure Arm Debugger and debug the application running on the Cortex-M microcontroller.

Prerequisites

- Create your Cortex-M application. See *Create a Cortex-M application* on page 4-36.
- Set up the **Terminal** views and interrupt the boot of the Linux application. See *Configure the Terminal views* on page 2-21.

Procedure

1. Power-cycle your target board and press any key in the Linux **Terminal** view to interrupt the Linux kernel boot process.
2. To open the **Edit Configuration** dialog box, right-click the **Blinky** project and select **Debug As** > **Arm Debugger...**.
3. Verify the **Connection Settings**. Ensure **Debug Port** is set to **JTAG** and check that the connection detects your debug adapter.

   ——— **Note** ———

   If your debug adapter is not detected, click **Browse…** to list available debug adapters.

   ———————

4. (OPTIONAL) Enable collection of the instruction execution history (trace) from the Cortex-M4, using the Debug and Trace Services Layer (DTSL):
   a. Click **Target Configuration…**.
   b. Click the **Cortex-M4** tab and ensure **Enable Cortex-M4 core trace** is selected. For more efficient trace, clear **Enable ETM Timestamps**.
   c. Click the **Cortex-A7** tab and disable all trace options to avoid buffer overflows.
   d. Click **OK** to confirm your changes and return to the **Debug Configurations** dialog box.

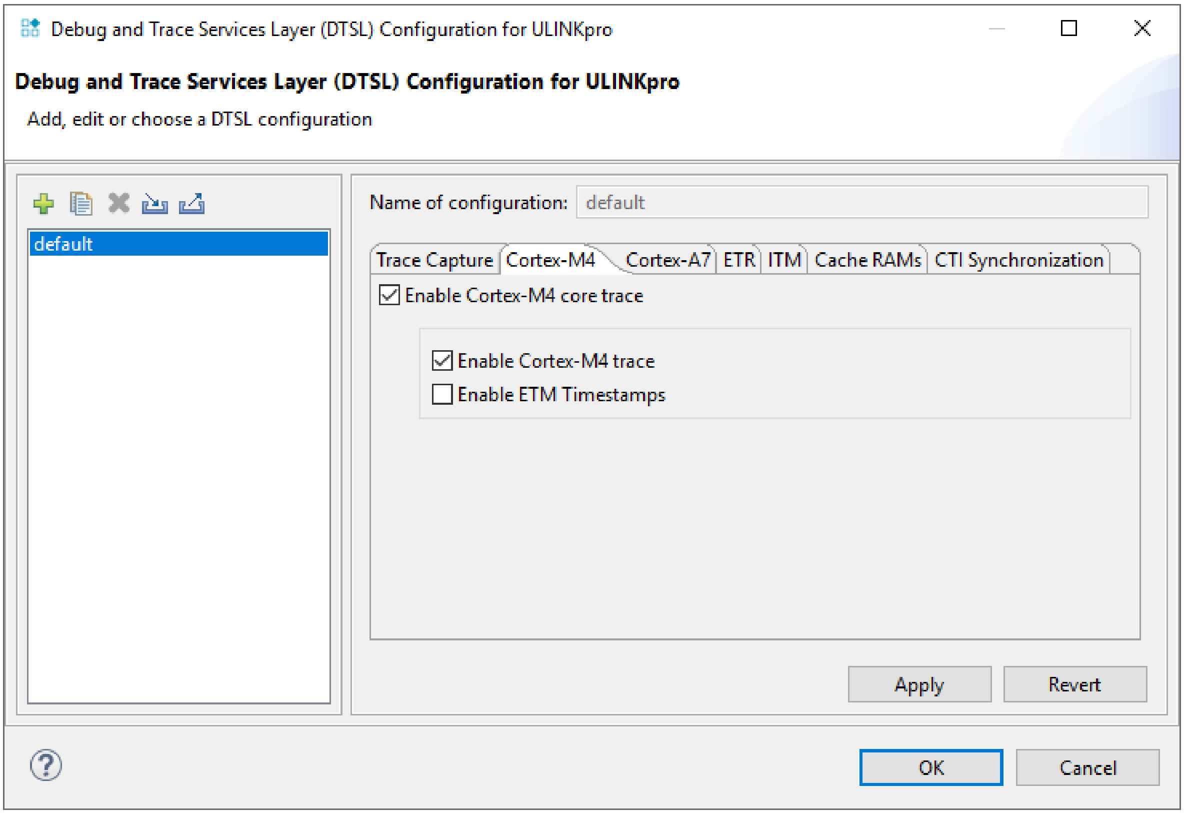


Figure 4-4 Cortex-M tab settings in the DTSL dialog.

5. To choose the operating system, click the **OS Awareness** tab and choose **Keil CMSIS-RTOS RTX** from the drop-down menu.
6. Click **Debug**. The application loads and runs until `main()`.

   Note

   If you see the following error message `Failed to launch debug server`, this might indicate:
   - An incorrect ULINKpro or DSTREAM connection address is selected.
   - The Linux boot process was not interrupted.

7. To start the Cortex-M4 application, click **Run** in the **Debug Control** view. Observe the output of the application in the **Terminal** window of the Cortex-M4, and the instruction execution history (trace) in the **Trace** view.

Next Steps

Create the Hello World Linux application on page 4-43.

Related information

Variables view

Registers view

Disassembly view

Memory view

Breakpoints view

Related tasks

4.1.1 Create a Cortex-M application on page 4-36
4.1.2 Create the source code files on page 4-38
4.1.3 Adapt the scatter file on page 4-40
4.1.4 Debug the Cortex-M Blinky application on page 4-41

4.2 Debug the Linux Application and Kernel

Create and debug a Linux Kernel project on Arm Cortex-A.

This section contains the following subsections:

4.2.1 Create the Hello World Linux application

Create and modify a project for an Arm Cortex-A class device running Linux.

Prerequisites

- You must have added GCC compiler to Arm Development Studio. See *Add the GCC compiler to Arm Development Studio* on page 2-18.

Procedure

1. To open the **New Project** dialog box, click **File** > **New** > **Project...** in the Arm Development Studio main menu.
2. Expand the **C/C++** folder, select **C Project**, and then click **Next**.
3. Select the **Hello World ANSI C Project** and the **GCC 7.4.1** toolchain.
4. Enter a **Project name**, such as **Hello_World**, and click **Finish**. The new project, **Hello_World**, is shown in the **Project Explorer** view.
5. Right-click on the project in the **Project Explorer** view and click **Build Project**.

The toolchain compiles and links all related source files. The **Console** view shows information about the build process.

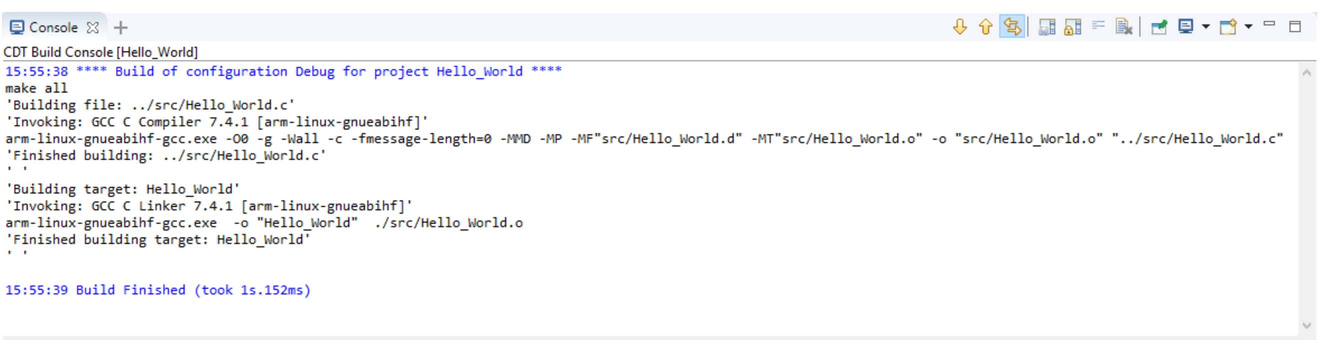


Figure 4-5 Console output showing the build results.

Next Steps

Debug your Linux application on page 4-43.

4.2.2 Debug the Hello World Linux application

This section explains how to debug a Linux application running on the Cortex-A7. Arm Debugger uses `gdbserver` for debugging Linux applications on the target hardware.

Prerequisites

- Set up the *Linux operating system* on page 2-14 on the target.
- *Configure the Terminal views* on page 2-21.

- *Create the Hello World Linux application* on page 4-43.
- Determine the IP address of your target.

Procedure

1. Set up a Remote Systems Explorer (RSE) connection to the target to download the application onto the file system of the target:
   a. Open the **Remote System Explorer** perspective.
   b. To open the **New Connection** wizard, click  in the **Remote Systems** view toolbar.
   c. Select **SSH Only** and click **Next**.
   d. Enter the IP address of the target in the **Host Name** field, and enter a name of your choice in the **Connection name** field. Click **Finish**.

   Note

   For more information, see *Set up a Remote System Explorer connection* on page 2-24.

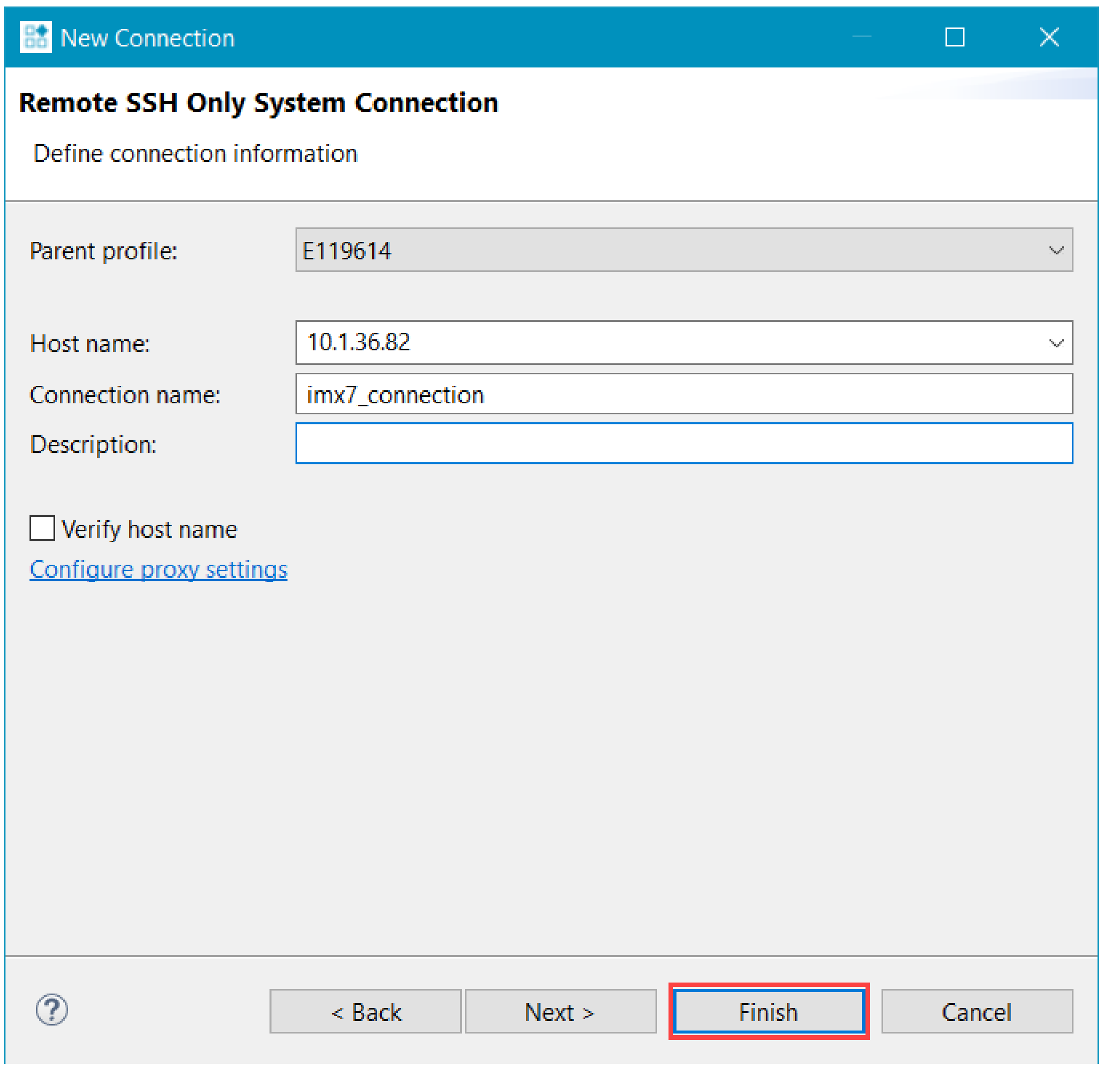


Figure 4-6 New Connection wizard, showing settings for an SSH Only RSE connection.

2. Return to the **Development Studio** perspective.
3. In the Linux **Terminal** view, enter `boot` to start the Linux system.
4. When the boot process has finished, log in with user name `root` and leave the password blank.

Warning

Your credentials might be different if you are not using the image downloaded from *Arm Developer*.

5. Configure the debugger:
   a. Right-click on the project **Hello_World** and select **Debug As** > **Debug Configurations...**
   b. Select **Generic Arm C/C++ Application** and click . Give your new configuration a name.
   c. In the **Connection** tab, select **Linux Application Debug** > **Application Debug** > **Connections via gdbserver** > **Download and debug application**.
   d. Under **Connections**, select your new RSE connection from the drop-down menu.

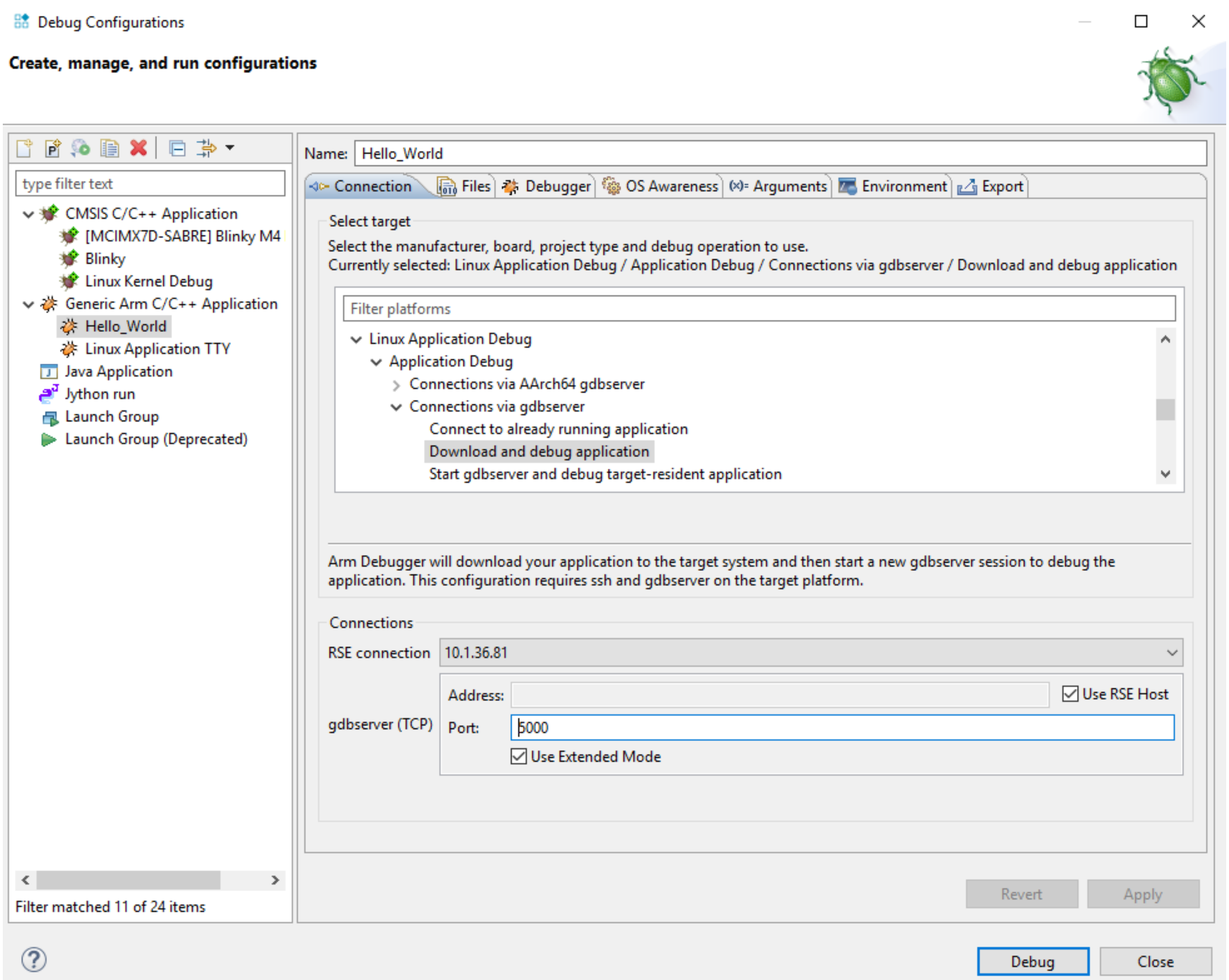


Figure 4-7 Debug Configurations dialog box showing Connection tab settings.

   e. Click the **Files** tab. Under **Target Configuration**:
      1. Click **Workspace…**. Select **Hello World** > **Debug** > **Hello_World.axf** for the **Application on host to download**.
      2. Select `/home/root` as the **Target download directory**.

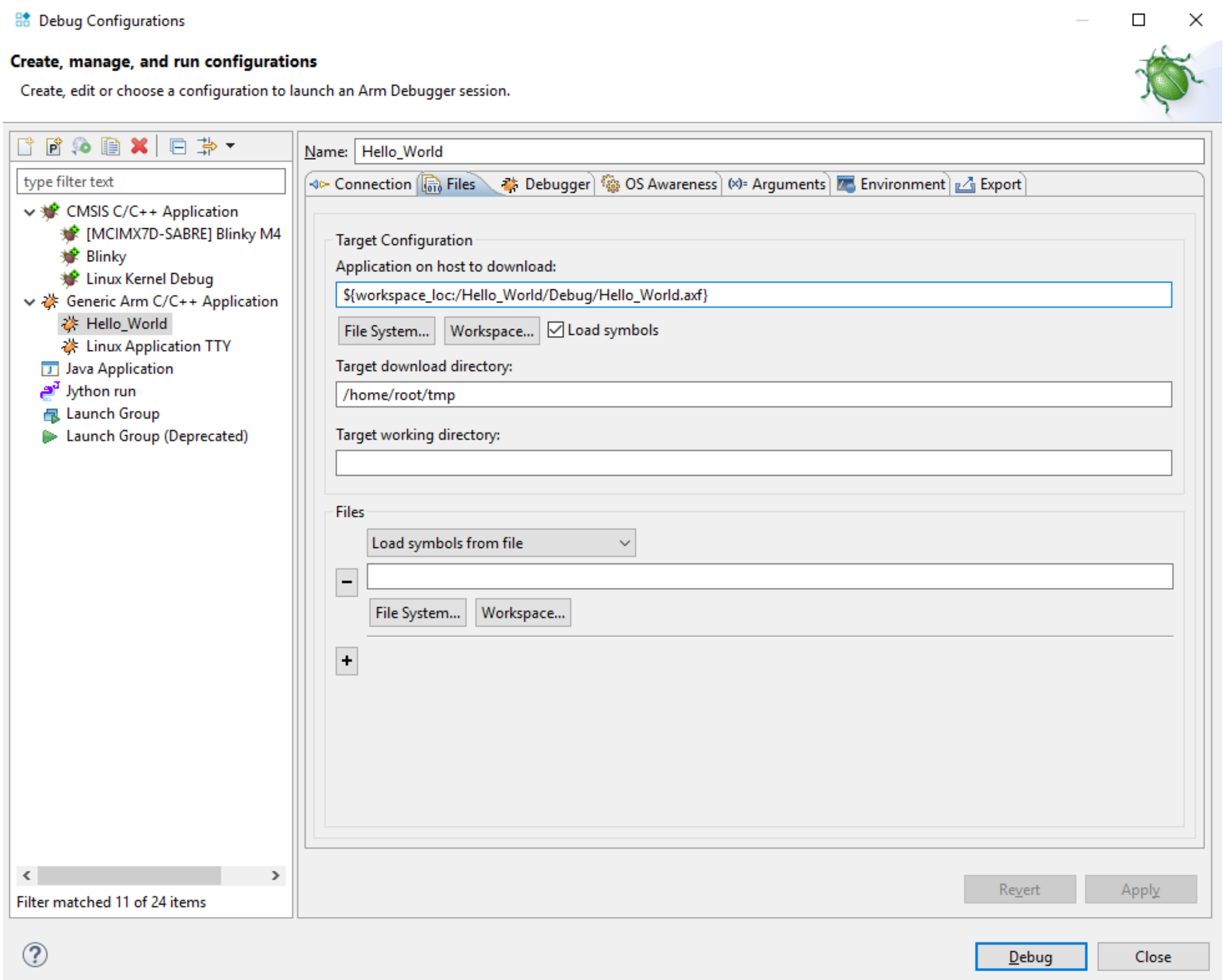


Figure 4-8 Debug Configurations dialog box showing Files tab settings.

f. Click the **Debugger** tab. Under **Run Control**, select **Debug from symbol "main"**. Click **Debug**.

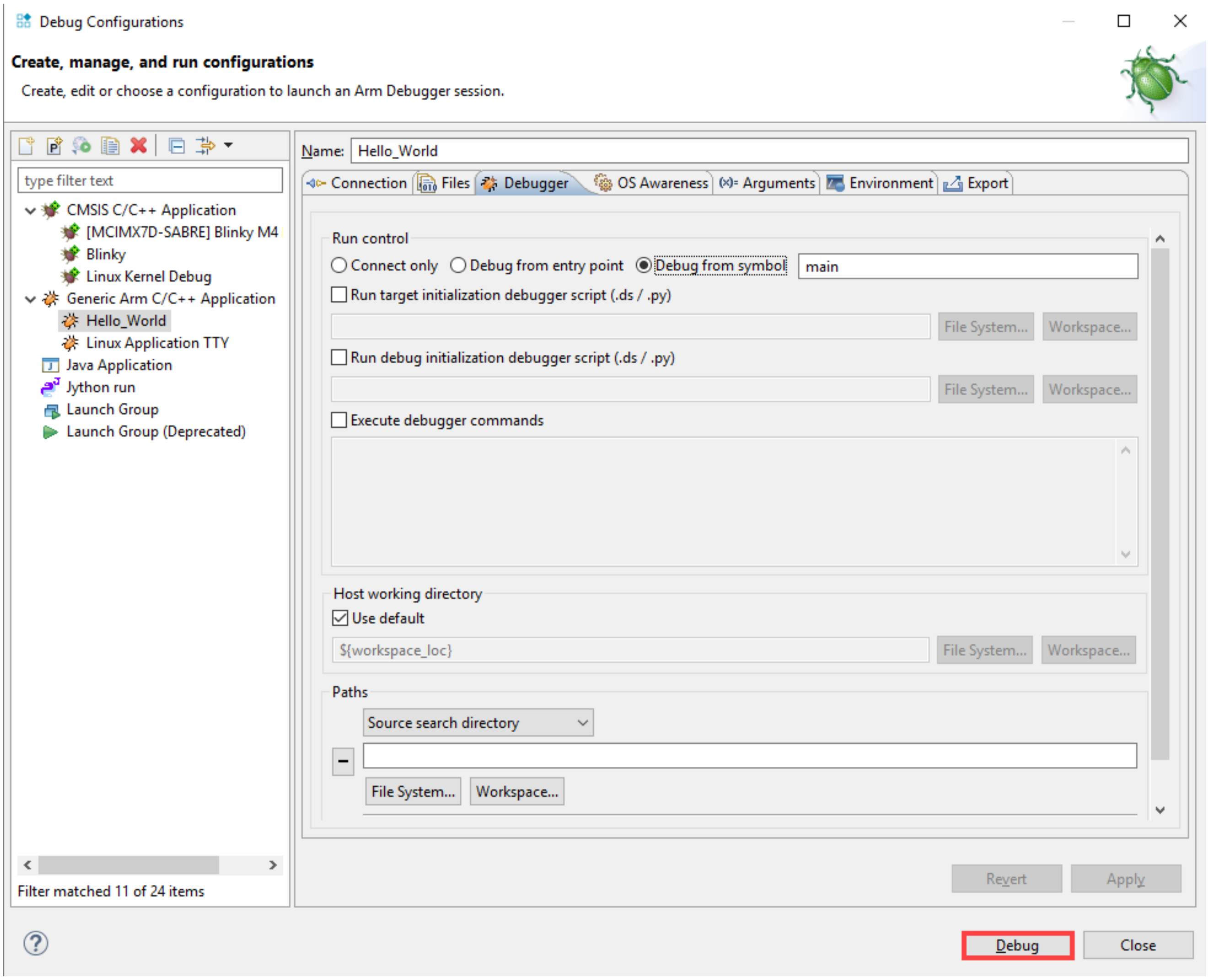


Figure 4-9 Debug Configurations dialog box showing Debugger tab settings.

———— **Note** ————

If you are asked to login, enter `root` as the username and leave the password field empty. Your credentials might be different if you are not using the image downloaded from *Arm Developer*.

————————————

6. In the **Debug Control** view, click  to run the application.

The **App Console** view shows the output of the application:

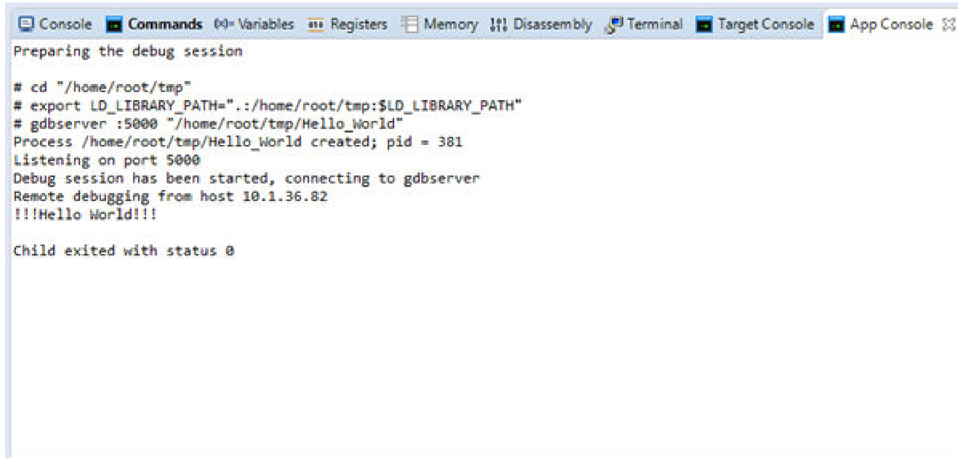


Figure 4-10 Screenshot of the output of the Linux application.

Next Steps

Create the Linux kernel debug project on page 4-48.

4.2.3 Create the Linux kernel debug project

Create a Linux kernel debug project, using a pre-configured Linux kernel image.

Arm provides:

- The Linux kernel, built with debug information.
- A complete `vmlinux` symbol file.
- File system.
- Full source code

These files are available to download from the Arm Development Studio *Related Software* page on Arm Developer.

Prerequisites

- Download the Linux kernel and `vmlinux` files from *Arm Developer*

Procedure

1. Unpack the Linux kernel sources, `kernel-source-imx7dsabresd-20170720.tar.gz`, into your currently active Arm Development Studio workspace.

   ——— **Note** ———

   On Windows platforms, the sources are not fully unpacked. Some symbolic links and case-sensitive source files are not created, because:

   - Windows does not support symbolic links.
   - Windows does not differentiate between uppercase and lowercase filenames. For example, Linux treats `foo.h` and `foo.H` as separate files whereas Windows treats them as the same file.

   These Windows features do not affect the tutorials in this workbook.

   ———————

2. Create a Linux kernel debug project:

   a. Click **File** > **New** > **Project...**.
   b. In the **New Project** wizard, select **C/C++** > **C project** and click **Next**.
   c. Under **Project type**, select **Executable** > **CMSIS C/C++ Project**. Under **Toolchains**, select **GCC 7.4.1**. Enter a suitable project name, such as `Linux Kernel Debug`, and click **Next**.

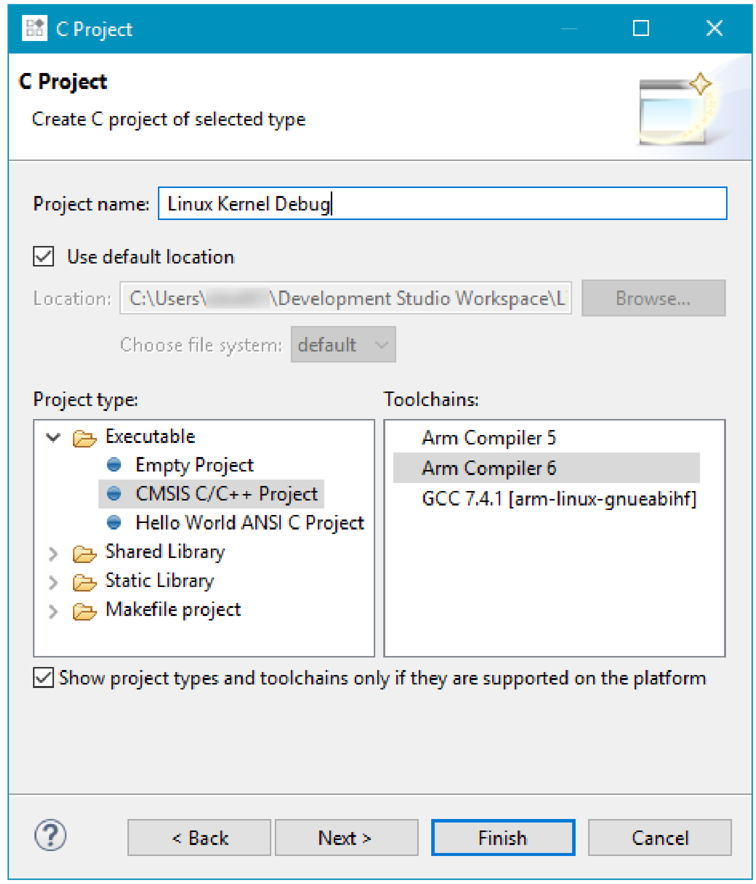


Figure 4-11 Screenshot of settings for the CMSIS C/C++ project.

d. Select the device **NXP** > **i.MX 7 Series** > **i.MX 7DUAL** > **MCIMX7D7** > **MCIMX7D7:Cortex-A7** and click **Finish**.

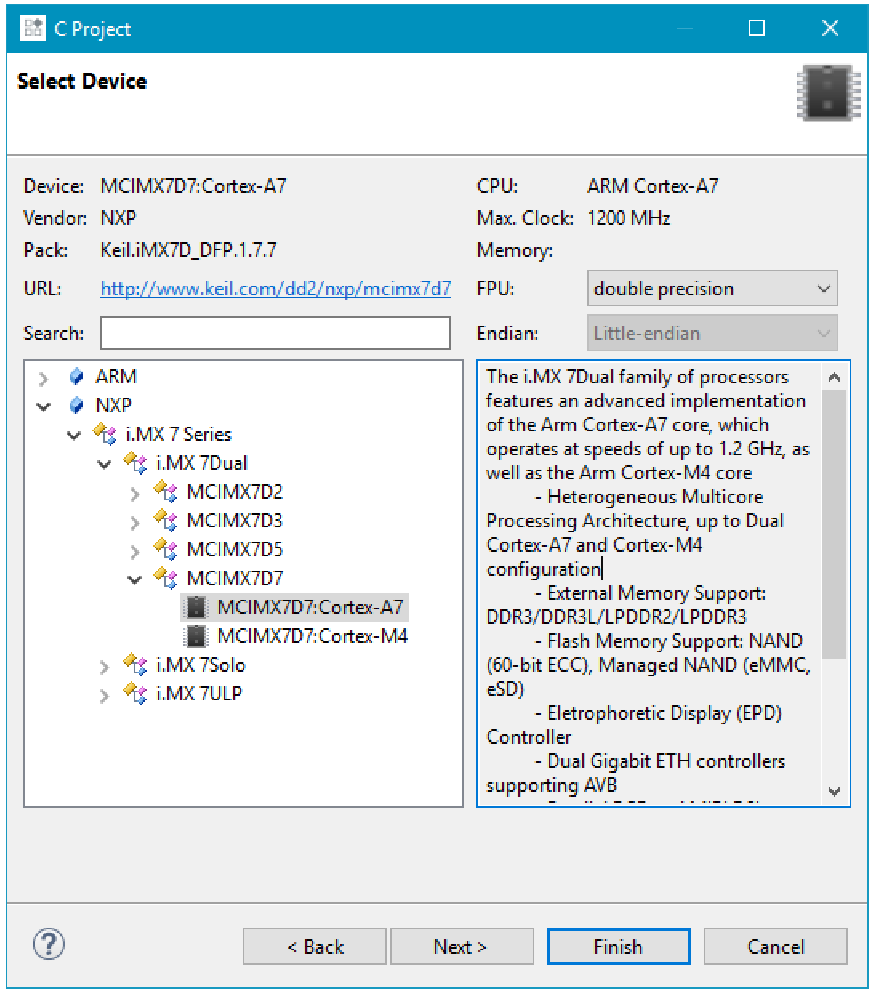


Figure 4-12 Screenshot of MCIMX7D7:Cortex-A7 selected.

3. Add the downloaded `vmlinux` file to the project folder using your system's file explorer.
4. Add a debugger script to the project:
   a. Right-click the project and select **New** > **Other...** to open the **New** dialog box.
   b. Select **Arm Debugger** > **Arm Debugger Script** and click **Next**. Name the file **stop.ds** and click **Finish**.
   c. In the **stop.ds** file, insert the code:

   ```
   stop
   set os physical-address 0x80008000
   ```

   `0x80008000` is the physical address at which the kernel is loaded. For this kernel, it is also the entry point of the kernel (the address to which U-Boot passes control to boot Linux, when it has completed its setup tasks).
   d. Press **Ctrl +S** to save your changes.

Next Steps

Configure Arm Debugger for the Linux kernel debug project on page 4-50.

4.2.4 Configure Arm Debugger for the Linux kernel debug project

Describes how to configure Arm Debugger for the Linux kernel, and begin debugging the kernel using breakpoints.

Prerequisites

You must *create the Linux kernel debug project* on page 4-48.

Procedure

1. Power-up your target board and interrupt boot of the Linux kernel by pressing any key in the Linux **Terminal** view.
2. Configure the debugger:
   a. Open the **Debug Configurations** dialog box; right-click on the **Linux Kernel Debug** project and click **Debug As...** > **Arm Debugger...**.
   b. In the **Connection** tab, under **CPU Instance**, select **SMP** from the drop down menu.
   c. Ensure the correct **Connection Type**, **Debug Port** and **Connection Address** have been selected for your debug probe.
   d. Click the **Advanced** tab. Under **Scripts**, enable **Run target initialization debugger script**. Click **Workspace..** and select the `stop.ds` script in your workspace. Click **OK**.
3. Click **Debug**. The **Commands** view shows the debug output.
4. Set a temporary hardware breakpoint on the entry point into the kernel. In the **Commands** view, enter `thbreak stext`.

   ——— **Note** ———

   The entry point is the address to which U-Boot passes control to boot Linux when it has completed its setup tasks. The entry point address might be different if you are not using the NXP i.MX7 SABRE board.

   ————————

5. Click  in the **Debug Control** view to run the target.
6. To boot the kernel, enter `boot` in the Linux **Terminal** view.

The code execution stops at the breakpoint, the **Command** view shows:

```
Enabled Linux kernel support for version "Linux 4.1.29-fslc+g59b38c3 #2 SMP PREEMPT
Wed Jun 21 16:36:50 CEST 2017 arm"
```

Execution stopped in SVC mode at breakpoint `1:S:0x80008000` indicates that Arm Debugger has read `init_nsproxy.uts_ns->name` to get the kernel name and version, and has successfully identified the kernel. Arm Debugger also sets a breakpoint automatically on `__enable_mmu()` to trap where the MMU gets turned on. You can see this breakpoint appear in the **Breakpoints** view.

The **Disassembly** view shows the assembly code at the entry point (labeled `stext`). If you have unpacked your kernel source code into the workspace, the **Editor** view shows the content of `head.S`.

——— **Note** ———

If `head.S` is not shown, from the **Debug Control** view drop-down menu, click **Path Substitution…**, navigate to the unpacked Linux source on your hard-drive and check that the final directory in the **Image Path** and **Host Path** correspond, then press **OK**.

————————

Next Steps

Debug the Linux kernel: Pre-MMU stage on page 4-51.

4.2.5 Debug the Linux kernel: Pre-MMU stage

After you have configured Arm Debugger, you can set breakpoints, set watchpoints, view registers, view memory, single-step, and other debug operations at this pre-MMU stage with source level symbols.

Prerequisites

You must *Configure Arm Debugger for the Linux kernel debug project* on page 4-50.

Procedure

1. At the kernel entry point, check the `Core` and `CP15` system registers in the **Registers** view to check that they are set as recommended by *kernel.org*. For example, you can verify that:
   a. The CPU is in SVC (supervisor) mode by selecting **Core** > **CPSR** > **M**.
   b. **R0** is `0`.
   c. **R2** contains a pointer to the device tree. Right-click **R2** and click **Show Memory Pointed To By R2**. Change the size of the memory displayed to 200 bytes, by entering `200` in the text entry box in the top-right of the **Memory** view.
   d. The MMU is off by selecting **CP15** > **VirtualMemoryControl** > **SCTLR** > **M**.
   e. The Data cache is off by selecting **CP15** > **VirtualMemoryControl** > **SCTLR** > **C**.
   f. The Instruction cache is either on or off by selecting **CP15** > **VirtualMemoryControl** > **SCTLR** > **I**.

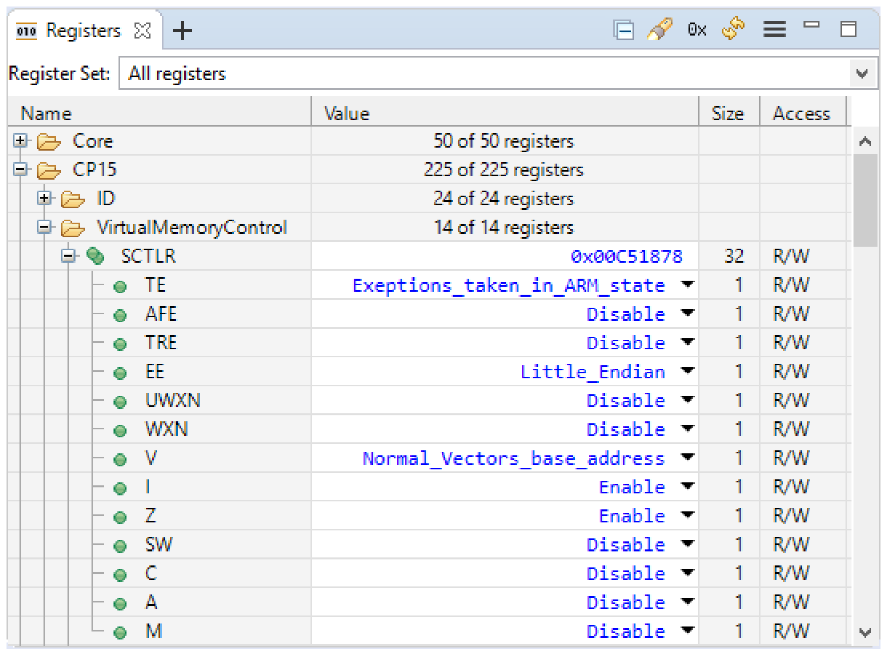


Figure 4-13 Registers view showing the values at the kernel entry point.

2. To see when the MMU is turned on:
   a. In the **Commands** field, enter `thbreak __turn_mmu_on` to set a breakpoint.
   b. To continue running the application, click .
   c. When `__turn_mmu_on` is reached, note the value of **SP**. This contains the virtual address of `__mmap_switched` and is the place the code jumps to after the MMU is enabled.
3. In general, it is not possible to single-step through `__turn_mmu_on`, so place a hardware breakpoint on the virtual address of `__mmap_switched`:

```
thbreak *$SP
```

4. Continue running by clicking . When the breakpoint at `__mmap_switched` is hit, the MMU is on.
5. Check that the MMU is now on by looking in the **Registers** view at **CP15** > **VirtualMemoryControl** > **SCTLR** > **M**. If the MMU is on, **Enable** is displayed.

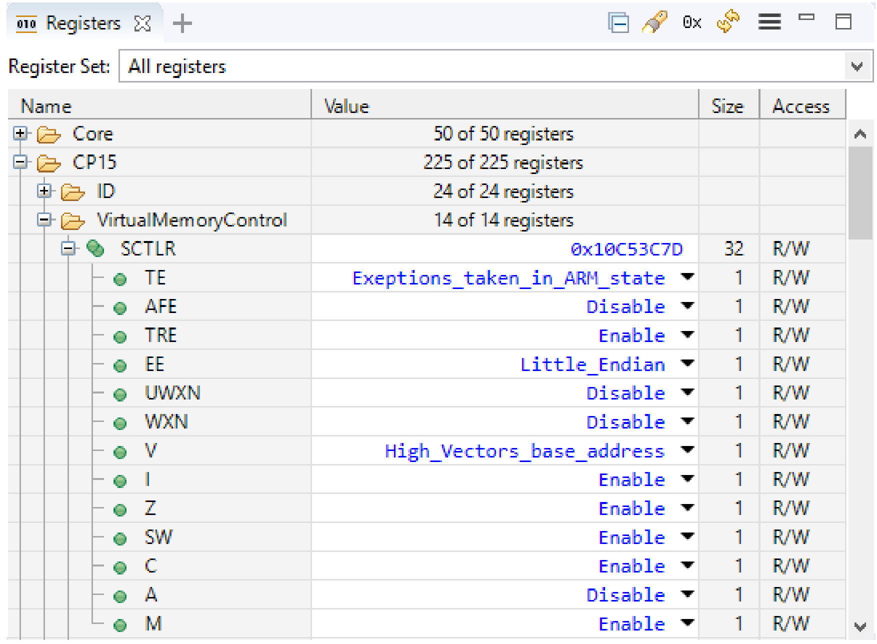


Figure 4-14 Registers view showing MMU enabled.

Arm Debugger also sets breakpoints automatically on `SyS_init_module()`, `SyS_finit_module()` and `SyS_delete_module()` to trap when kernel modules are inserted (`insmod`) and removed (`rmmod`). You can see these breakpoints appearing in the **Breakpoints** view.

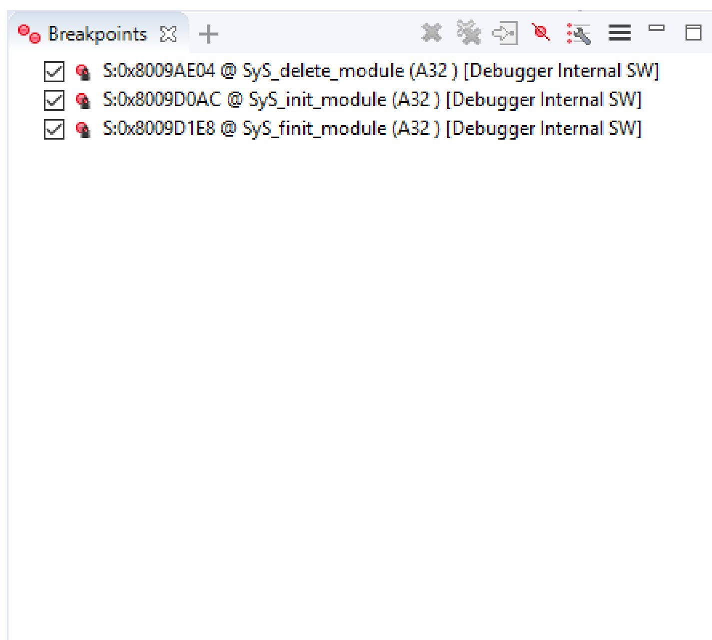


Figure 4-15 Breakpoints view showing automatically created breakpoints.

Next Steps

Debug the Linux kernel: Post-MMU stage on page 4-53.

4.2.6 Debug the Linux kernel: Post-MMU stage

Here we describe some of the debug options that are available, including how to set breakpoints, single-step through processes, and view details of the kernel in the post-MMU stage.

Prerequisites

You must *Debug the Linux kernel: Pre-MMU stage* on page 4-51.

Procedure

1. Set a breakpoint on the C code:
   a. Open `main.c` from the **Project Explorer** view.
   b. Set a breakpoint on `thbreak start_kernel`.
   c. Click ▶ to run to the breakpoint.
2. Set a breakpoint with:

   ```
   thbreak kernel_init
   ```

   then run to it.

   So far, CPU 0 is doing all the work, because CPU 1 is still powered down. CPU 1 is powered up as part of `kernel_init()`.

   A useful feature during kernel bring-up is to display early `printk` output in the command window of Arm Debugger.

   Note

   If the console is not enabled, then the serial port produces no output.

3. You can view the entire log so far with:

   ```
   info os-log
   ```

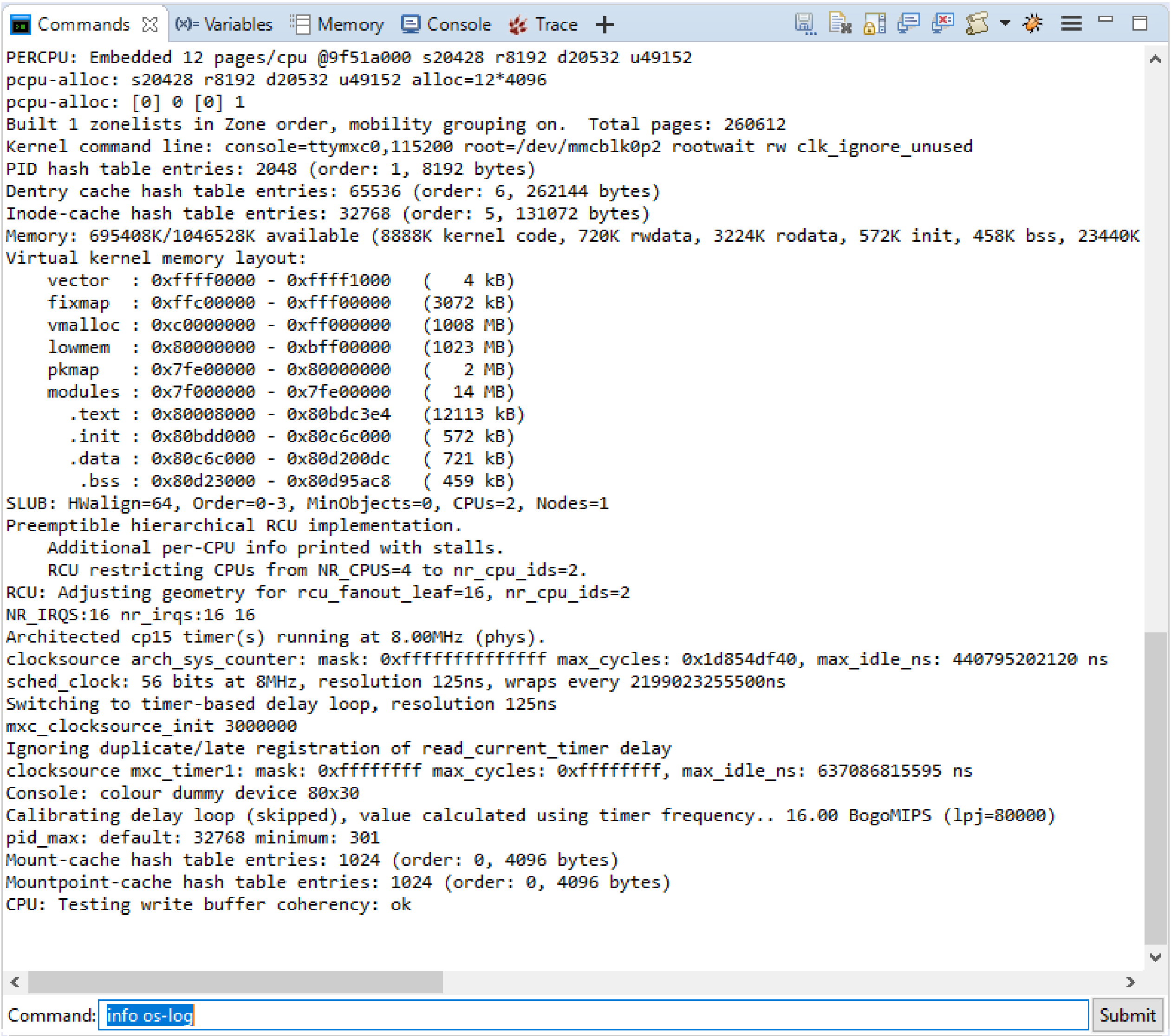


Figure 4-16 Commands view showing output of info os-log command.

4. To view the log output line by line, as it happens, use:

   `set os log-capture on`

5. `kernel_init()` tries to start the `init` process. To see this, set a breakpoint at the end of `kernel_init()` then run to it (set the breakpoint in the `main.c` file in the Editor view). CPU 1 is now powered up.

   You can automate many of these steps, either with a script file, or by filling-in the **Debug Configuration**'s fields before launching.

6. The **Debug Control** view is currently showing the cores, but you can change this to show the threads. In the **Debug Control** view, right-click on the connection, then select **Display Threads**. The current thread (`init`), and groups for **Active Threads** and **All Threads** appear in the **Debug Control** view.

7. Delete all user breakpoints and continue by clicking . Let the kernel run all the way to the Login prompt. Log in as `root`.

8. Click **Interrupt** to stop the target. In the **Debug Control** view, expand **Active Threads** and **All Threads**. In **All Threads**, you can see a large number of threads/processes are created. Only two are actually running, one on each of the two cores. You can see these in **Active Threads**.

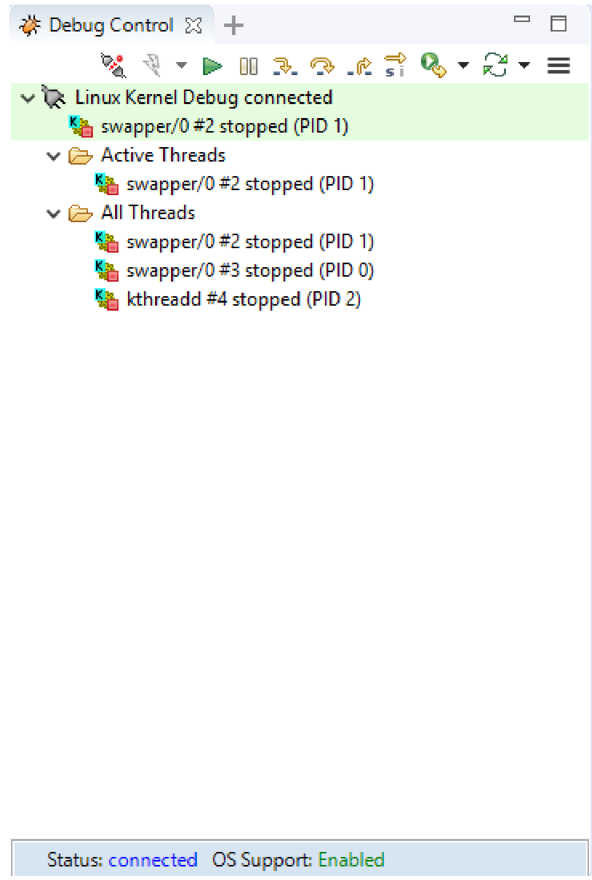


Figure 4-17 Debug Control view showing All Threads and Active Threads.

9. You can view the state of the cores, threads and processes on the command-line by entering:

```
info cores

info threads

info processes
```

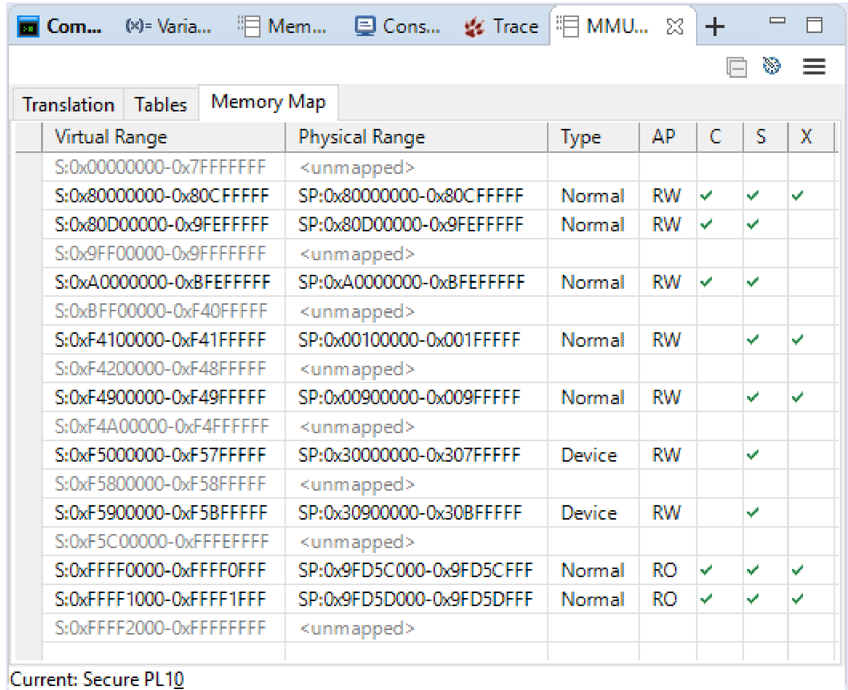
10. You can single-step a core or a thread/process:
    a. Select either the core or the thread/process in the **Debug Control** view.
    b. Click .

    —————— **Note** ——————

    When single-stepping though a process, it might get migrated to another core. If a breakpoint is set on a process, the debugger can track the migration of process-specific breakpoints to the other core.

    ————————————————

11. You can check the virtual-to-physical address map for Linux by using the MMU view:
    a. Click to continue running the target.
    b. Go to **Window** > **Show View** > **MMU**.
    c. Switch to the **Memory Map** tab and press the **Show Memory Map** button to refresh the values.



Virtual Range	Physical Range	Type	AP	C	S	X
S:0x00000000-0x7FFFFFFF	<unmapped>					
S:0x80000000-0x80CFFFFF	SP:0x80000000-0x80CFFFFF	Normal	RW	✓	✓	✓
S:0x80D00000-0x9FEFFFFF	SP:0x80D00000-0x9FEFFFFF	Normal	RW	✓	✓	
S:0x9FF00000-0x9FFFFFFF	<unmapped>					
S:0xA0000000-0xBFEFFFFF	SP:0xA0000000-0xBFEFFFFF	Normal	RW	✓	✓	
S:0xBFF00000-0xF40FFFFF	<unmapped>					
S:0xF4100000-0xF41FFFFF	SP:0x00100000-0x001FFFFF	Normal	RW		✓	✓
S:0xF4200000-0xF48FFFFF	<unmapped>					
S:0xF4900000-0xF49FFFFF	SP:0x00900000-0x009FFFFF	Normal	RW		✓	✓
S:0xF4A00000-0xF4FFFFFF	<unmapped>					
S:0xF5000000-0xF57FFFFF	SP:0x30000000-0x307FFFFF	Device	RW		✓	
S:0xF5800000-0xF58FFFFF	<unmapped>					
S:0xF5900000-0xF5BFFFFF	SP:0x30900000-0x30BFFFFF	Device	RW		✓	
S:0xF5C00000-0xFFFEFFFF	<unmapped>					
S:0xFFFF0000-0xFFFF0FFF	SP:0x9FD5C000-0x9FD5CFFF	Normal	RO	✓	✓	✓
S:0xFFFF1000-0xFFFF1FFF	SP:0x9FD5D000-0x9FD5DFFF	Normal	RO	✓	✓	✓
S:0xFFFF2000-0xFFFFFFFF	<unmapped>					

Figure 4-18 MMU view showing the Memory Map for Linux.

12. To look at the kernel's `thread_info` structure, stop the target and check the stack size of the kernel with:

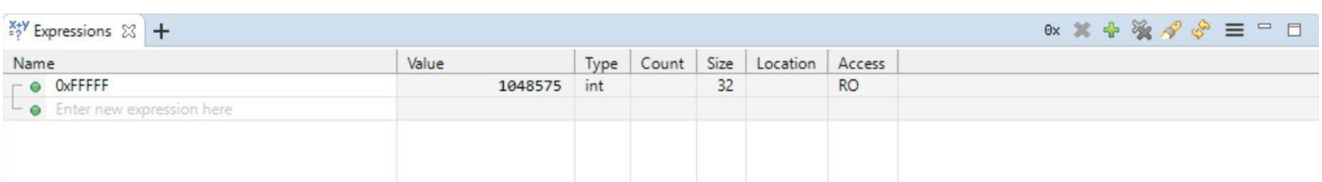
    ```
    show os kernel-stack-size
    ```

    For this Armv7 kernel, the kernel stack size is 8K.

13. In the **Expressions** view, enter a new expression in the field at the bottom of the view:

    ```
    (struct thread_info*)($sp_svc & ~0x1FFF)
    ```

    `0x1FFF` is 8K minus 1. Expand the tree structure to explore the contents of the Trace Control Block (TCB). The list of threads in the **Debug Control** view is created from the same information, so they should match. For example, the thread name is held in **task.comm**.



Name	Value	Type	Count	Size	Location	Access
0xFFFFF	1048575	int		32		RO
Enter new expression here						

Figure 4-19 Expressions view showing the contents of the input expression.

14. To get a simple view into the workings of the scheduler, set a breakpoint on `__schedule()` with:

    ```
    hbreak __schedule
    ```

    Note

    The use of `hbreak` results in a persistent hardware breakpoint instead of a temporary one.

Click ▶ to continue running the application. At each instance that the breakpoint is hit, click ▶ . You can see the names of the active threads change in **Active Threads**, and different threads are scheduled-in.

Note

Alternatively, instead of setting a breakpoint on `__schedule()`, try to set a breakpoint on `do_fork()`. If nothing forks, you can force a fork by entering a command, for example `ls`.

In summary, we have looked at how you can use Arm Development Studio to debug the Linux kernel, both in pre-MMU enabled and post-MMU enabled stages, and looked at a few of the internal features of the kernel.

Next Steps

You can *debug the Linux Kernel Module* on page 4-59.

Related tasks

4.2.1 Create the Hello World Linux application on page 4-43
4.2.2 Debug the Hello World Linux application on page 4-43
4.2.3 Create the Linux kernel debug project on page 4-48
4.2.4 Configure Arm Debugger for the Linux kernel debug project on page 4-50
4.2.5 Debug the Linux kernel: Pre-MMU stage on page 4-51
4.2.6 Debug the Linux kernel: Post-MMU stage on page 4-53

4.3 Debug the Linux Kernel Module

Create and debug a Linux Kernel Module for this example project.

This section contains the following subsection:

- *4.3.1 Debug a Linux kernel module* on page 4-59.

4.3.1 Debug a Linux kernel module

Configure Arm Debugger for the `imx_rpmsg_tty` module and debug the kernel module for your example project.

Prerequisites

You must *debug the Linux Kernel* on page 4-43.

Procedure

1. To create a Linux kernel module debug project, create a new **CMSIS C/C++ Project** called **Linux kernel module debug**.
2. Add the `vmlinux` file to the project folder using **Windows Explorer**.
3. Add a debugger script to the project:
   a. Right-click the project and select **New** > **Other...**. Select **Arm Debugger** > **Arm Debugger Script**.
   b. Name the file `stop.ds`.
   c. Open the file and add the following text:

   ```
   stop
   ```

   d. Save the script by pressing **Ctrl** + **S**.
4. Add another debugger script to the project:
   a. Right-click the project and select **New** > **Other...**. Select **Arm Debugger** > **Arm Debugger Script**.
   b. Name the file `load_ko.ds`.
   c. Open the file and add the following text:

   ```
   add-symbol-file imx_rpmsg_tty.ko
   ```

   ——— **Note** ———

   Make sure the `imx_rpmsg_tty.ko` file is stored in your active workspace so Arm Development Studio can find it. Otherwise, specify the full file path to it. You can download the file and the source code file from the board support page of your development board.

   ————————

   The `stop` command in the first script halts the processor before loading the kernel symbols, and the `add-symbol-file` command loads the kernel module object file.
5. Configure Arm Debugger for the Linux kernel module:
   a. Right-click the project and select **Debug As** > **Arm Debugger...**.
   b. On the **Connections** tab, set the **CPU Instance** to either **0** or **SMP**.
   c. On the **Advanced** tab, specify the path to the `vmlinux` file and enable **Load symbols only**.
   d. Set the initialization debugger scripts:
      1. Under Run control, select **Connect only**.
      2. Enable **Run target initialization debugger script (.ds/.py)** and add the `stop.ds` file from your active workspace.
      3. Enable **Run debug initialization debugger script (.ds/.py)** and add the `load_ko.ds` file from your active workspace.
   e. Apply the settings and click **Close**.

Warning

Do not click **Debug** yet.

6. Debug the kernel module:
   a. Restart your target, then press any key to interrupt the boot process.
   b. Debug and run the Cortex-M4 application `RPMSG TTY RTX`.
   c. Boot Linux by typing `boot` into the Linux **Terminal** view.
   d. At the Linux prompt, enter the following command to install the driver for the kernel module:

      ```
      modprobe imx_rpmsg_tty
      ```
   e. Debug and run the `Kernel_Debug` project.
   f. Open `imx_rpmsg_tty.c` and set breakpoints.
   g. Debug the Linux Application TTY:
      1. Check that the RSE connection is still active.
      2. Run the application. Arm Debugger stops at the breakpoint you set in the previous step.

You have now fully debugged a Linux image and Cortex-M bare-metal application.

Next Steps

Store the Cortex-M image on an SD Card on page 5-61.

Related tasks

4.3.1 Debug a Linux kernel module on page 4-59

Chapter 5
Store the Cortex-M image on an SD Card

Store the Cortex-M image on an SD card for execution at start-up.

It contains the following sections:

5.1 Create a Cortex-M binary image (BIN)

Create a binary image (`BIN`) with the `fromelf` utility application.

The Blinky project, that is configured in *Create a Cortex-M application* on page 4-36 is used as an example.

Prerequisites

Create the Cortex-M project by following the steps in *Debug applications on a heterogenous system* on page 4-35.

Procedure

1. Open the **Properties** dialog box; right-click on the project and select **Properties**.
2. From the side-bar, select **C/C++ Build** > **Settings**.
3. Click the **Build Steps** tab and under **Post-build steps**, enter the **Command**:

   ```
   fromelf --bin --output "Blinky.bin" "Blinky.axf"
   ```
4. Click **OK** to close the dialog box.
5. To generate the `BIN` file, rebuild the project by selecting it in the **Project Explorer** view and clicking  .

Next Steps

You can now store the generated `BIN` file on an SD card. See *Store Cortex-M BIN file on SD Card* on page 5-63.

5.2 Store Cortex-M BIN file on SD Card

Store your `BIN` image on SD card in the boot partition.

The SD Card has two partitions:

- The Linux file system partition.
- The FAT32 boot partition.

Prerequisites

Create a binary image (`BIN`) with the `fromelf` utility application, see *Create a Cortex®-M binary image (BIN)* on page 5-62.

Procedure

1. In the Linux **Terminal** view, list the partitions with the `fdisk` command:

```
fdisk -l
...
Device         Boot Start     End Sectors  Size Id Type
/dev/mmcblk0p1       8192   24575   16384    8M  c W95 FAT32 (LBA)
/dev/mmcblk0p2      24576 1236991 1212416  592M 83 Linux
```

2. To execute the `BIN` file at system startup, store the Cortex-M binary image in the FAT32 boot partition:
   a. Create a sub-directory on the Linux file system, for example:

   ```
   mkdir /media/sd0
   ```

   b. Mount the Linux file system partition for access with Remote System Explorer (RSE).

   ```
   mount -t vfat /dev/mmcblk0p1 /media/sd0
   ```

   c. Use RSE to copy the `BIN` file from your workspace to the `/media/sd0` directory.
   d. Unmount the partition to ensure that the file is written correctly:

   ```
   umount /media/sd0
   ```

   e. Reboot the system and press any key to interrupt U-boot.

Next Steps

Configure the U-Boot environment to start-up the `BIN` image file. See *Run Cortex-M BIN file from U-Boot* on page 5-64.

5.3 Run Cortex-M BIN file from U-Boot

Configure the U-Boot environment to start-up the `BIN` image file.

The Cortex-M `BIN` file is stored in the boot partition.

Prerequisites

Store the Cortex-M BIN file on SD Card on page 5-63.

Procedure

1. In the Linux **Terminal** view, use the `setenv` command to change the boot image to the new `BIN` file:

   ```
   setenv m4image Blinky.bin; save
   ```

   The `printenv` command shows the boot setup:

   ```
   printenv
   ...
   loadm4image=fatload mmc ${mmcdev}:${mmcpart} 0x7F8000 ${m4image}
   m4boot=run loadm4image; bootaux 0x7F8000
   m4image=Blinky.bin
   ```

2. Run `m4boot` to start the **Blinky** application:

   ```
   run m4boot
   ```

   Note

   For more information refer to the *U-Boot Command Line Interface* in the U-Boot user's manual.