

# Arm® Development Studio

Version 2020.1

## Heterogeneous system debug with Arm DS

**arm**

# Arm® Development Studio

## Heterogeneous system debug with Arm DS

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

### Release Information

### Document History

Issue	Date	Confidentiality	Change
2000-01	3 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

### Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

**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](https://developer.arm.com)

# Contents

## Arm® Development Studio Heterogeneous system debug with Arm DS

### **Preface**

About this book .....	9
-----------------------	---

### **Chapter 1**

#### **Introduction**

1.1 Heterogenous debug on Arm® Development Studio: Introduction .....	1-12
---	------

### **Chapter 2**

#### **Set up your target for debug**

2.1 Install the Linux image .....	2-14
2.2 Set up the hardware connections .....	2-15
2.3 Add the GCC compiler to Arm® Development Studio .....	2-18
2.4 Identify COM ports on the host PC .....	2-20
2.5 Configure the Terminal views .....	2-21
2.6 Determine the IP address of your development board .....	2-23
2.7 Set up a Remote System Explorer connection .....	2-24

### **Chapter 3**

#### **Debug an example project**

3.1 Set up the Cortex®-M application .....	3-27
3.2 Configure Arm® Debugger for Cortex®-M .....	3-29
3.3 Set up the Cortex®-A Linux application .....	3-32
3.4 Configure Arm® Debugger for Linux application debug .....	3-33

<b>Chapter 4</b>	<b><i>Debug applications on a heterogenous system</i></b>	
4.1	<i>Build and debug the Cortex®-M application</i>	4-36
4.2	<i>Debug the Linux Application and Kernel</i>	4-42
4.3	<i>Debug the Linux Kernel Module</i>	4-56
<b>Chapter 5</b>	<b><i>Store the Cortex-M image on an SD Card</i></b>	
5.1	<i>Create a Cortex®-M binary image (BIN)</i>	5-59
5.2	<i>Store Cortex®-M BIN file on SD Card</i>	5-60
5.3	<i>Run Cortex®-M BIN file from U-Boot</i>	5-61

# List of Figures

## Arm® Development Studio Heterogeneous system debug with Arm DS

Figure 2-1	An NXP i.MX7 SABRE board connected with JTAG, Ethernet, and USB UART connections. .	2-16
Figure 2-2	Autodetection results for the GCC 7.4.1 compiler. ....	2-18
Figure 2-3	Device Manager, showing the two COM ports associated with the development board. ....	2-20
Figure 2-4	Launch Terminal dialog box, showing settings for the Cortex-A Terminal view. ....	2-21
Figure 2-5	IP address, displayed by running ifconfig. ....	2-23
Figure 2-6	New Connection wizard, showing settings for an SSH Only RSE connection. ....	2-24
Figure 3-1	Installation of Cortex-M example project. ....	3-27
Figure 3-2	Installation of missing packs. ....	3-28
Figure 3-3	Console output of the Cortex-M build result. ....	3-28
Figure 3-4	Terminal view showing the boot of Linux. ....	3-29
Figure 3-5	Debug configurations for Cortex-M, and Target Configuration... button. ....	3-30
Figure 3-6	The output of the Cortex-M4 Terminal window. ....	3-31
Figure 3-7	The console output showing the Cortex-A build result. ....	3-32
Figure 3-8	Debug Configurations dialog box. ....	3-33
Figure 3-9	Output of the App Console and Cortex-M Terminal. ....	3-34
Figure 4-1	The selected components for the CMIMX7D7:Cortex-M4. ....	4-37
Figure 4-2	Console output showing the build results. ....	4-40
Figure 4-3	Console output showing the build results. ....	4-42
Figure 4-4	Debug Configurations dialog box showing Connection tab settings. ....	4-44
Figure 4-5	Debug Configurations dialog box showing Files tab settings. ....	4-45
Figure 4-6	Screenshot of the output of the Linux application. ....	4-46

Figure 4-7	Screenshot of settings for the CMSIS C/C++ project. ....	4-47
Figure 4-8	Screenshot of MCIMX7D7:Cortex-A7 selected. ....	4-48
Figure 4-9	Registers view showing MMU enabled. ....	4-50
Figure 4-10	Breakpoints view showing automatically created breakpoints. ....	4-51
Figure 4-11	Commands view showing output of info os-log command. ....	4-52
Figure 4-12	Debug Control view showing All Threads and Active Threads. ....	4-53
Figure 4-13	Expressions view showing the contents of the input expression. ....	4-54

# Preface

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

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

We introduce the heterogeneous 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 heterogeneous 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](mailto:errata@arm.com). Give:

- The title *Arm Development Studio Heterogeneous system debug with Arm DS*.
- The number 102021\_2010\_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

We introduce the heterogenous debug on Arm Development Studio workbook.

It contains the following section:

- [\*1.1 Heterogenous debug on Arm® Development Studio: Introduction on page 1-12.\*](#)

## 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 Phytex 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 on page 2-14.](#)
- [2.2 Set up the hardware connections on page 2-15.](#)
- [2.3 Add the GCC compiler to Arm® Development Studio on page 2-18.](#)
- [2.4 Identify COM ports on the host PC on page 2-20.](#)
- [2.5 Configure the Terminal views on page 2-21.](#)
- [2.6 Determine the IP address of your development board on page 2-23.](#)
- [2.7 Set up a Remote System Explorer connection on page 2-24.](#)

## 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
<ol style="list-style-type: none"> <li>1. Download and install a disk imager, for example <b>Win32 Disk Imager</b> from <a href="http://win32diskimager.sourceforge.net/">http://win32diskimager.sourceforge.net/</a>, and run the program.</li> <li>2. Select the image file named <code>core-image-base-imx7dsabresd-20170720.rootfs.sdcard</code>. Then, select the device letter of the SD card and click <b>Write</b>.</li> </ol> <p><b>Warning</b></p> <p>To avoid corrupting your existing data, ensure that you select the correct SD device.</p>	<p>Enter the following command, where <code>&lt;path/to/sd&gt;</code> is the path to your SD card:</p> <pre>sudo dd if=core-image-base-imx7dsabresd-20170720.rootfs.sdcard of=&lt;path/to/sd&gt; bs=1M</pre> <p><b>Warning</b></p> <p>To avoid corrupting your existing data, ensure that you select the correct SD device.</p>

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 ULINK™pro 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.

---



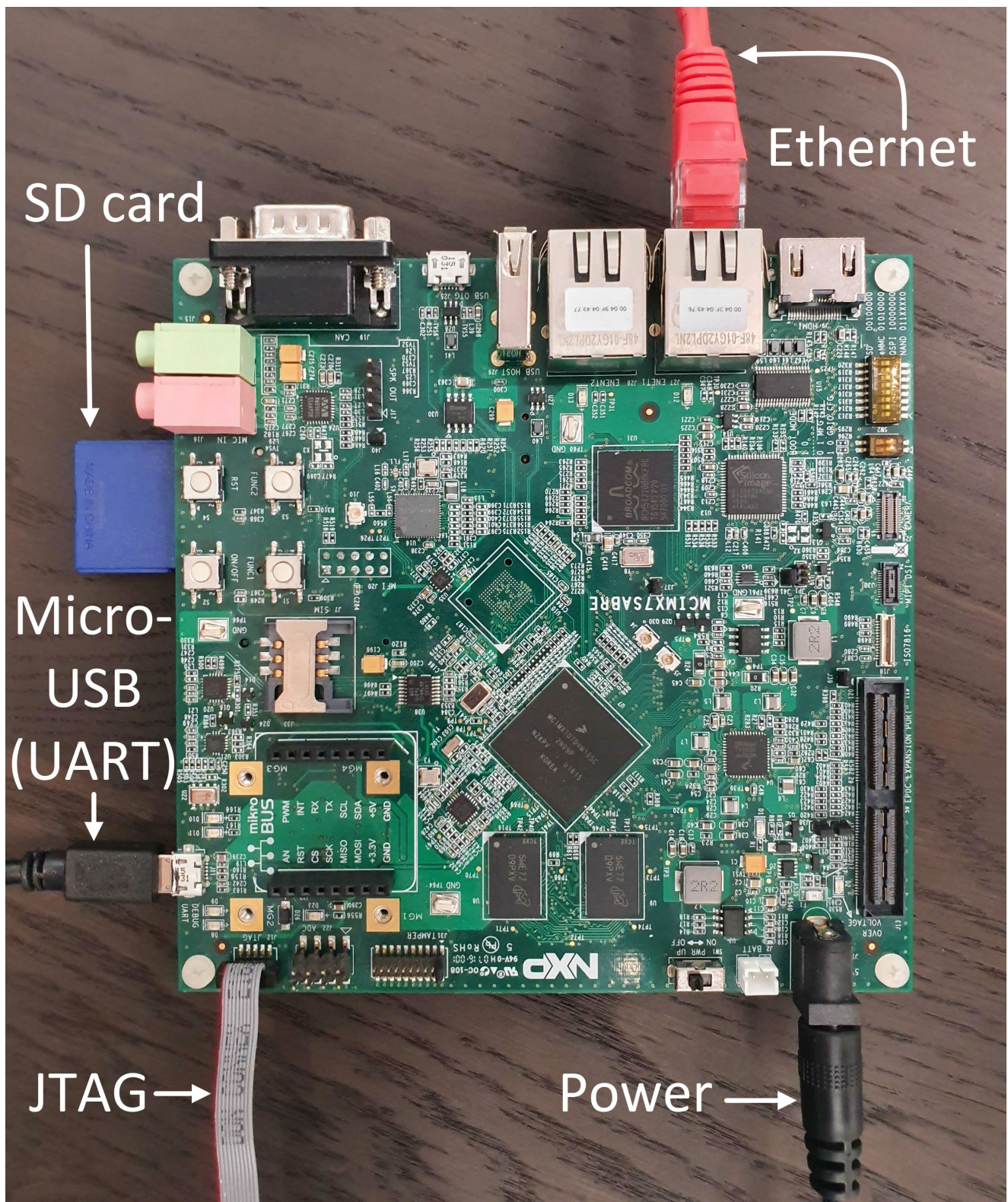


Figure 2-1 An NXP i.MX7 SABRE board connected with JTAG, Ethernet, and USB UART connections.



---

**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.

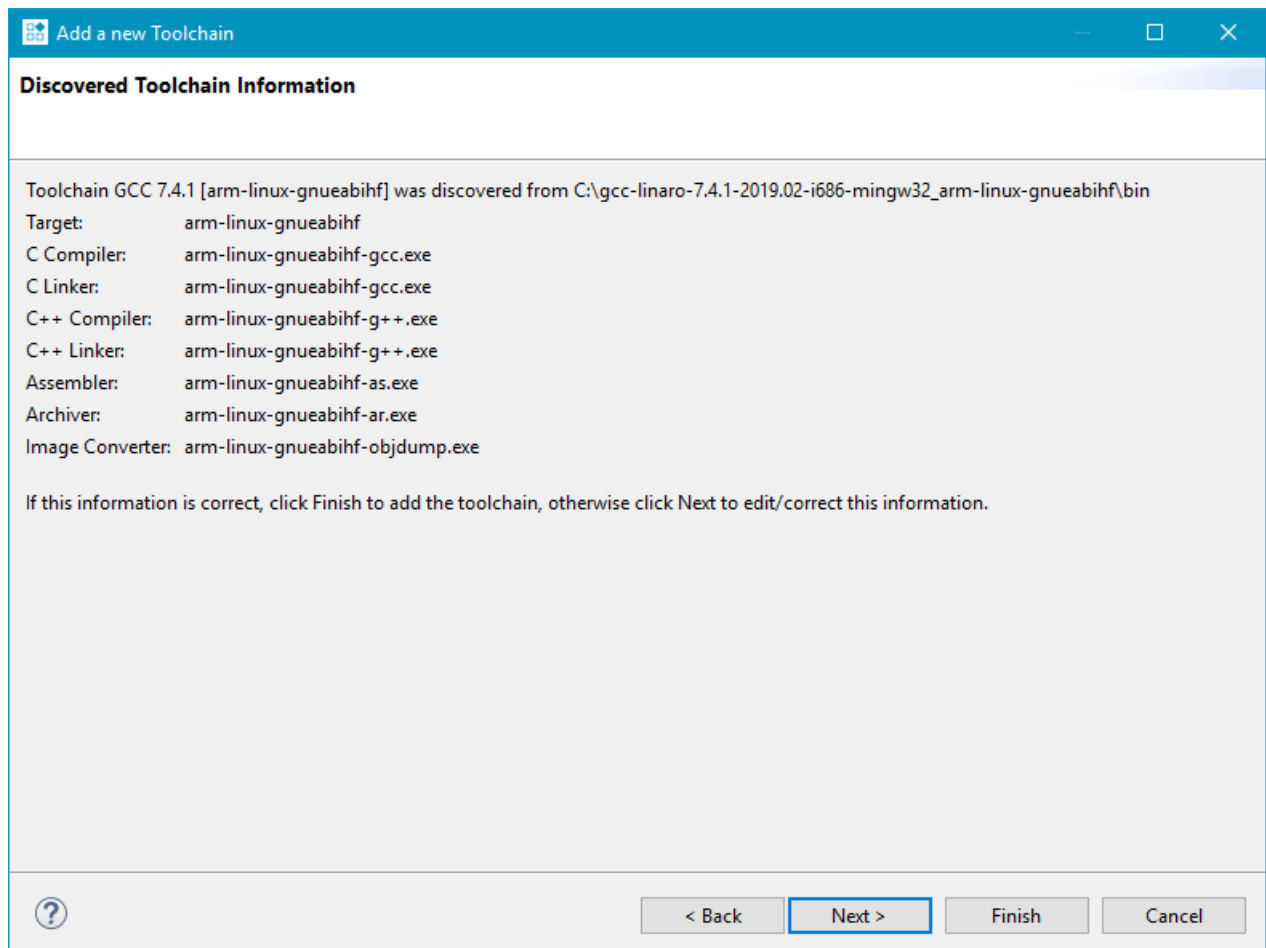


Figure 2-2 Autodetection results for the GCC 7.4.1 compiler.

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

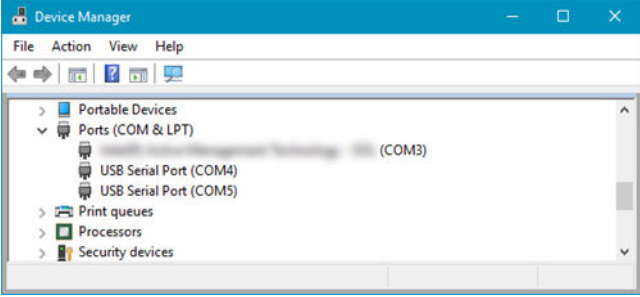
Identify and make note of the serial (COM) port numbers on your host PC. These COM port numbers are used later in the workbook, to configure the **Terminal** views in Arm Development Studio.

### 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:

Windows	Linux
<ol style="list-style-type: none"> <li>1. Open the Windows <b>Device Manager</b>.</li> <li>2. Expand <b>Ports (COM &amp; LPT)</b> 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.</li> </ol>  <p><b>Figure 2-3 Device Manager, showing the two COM ports associated with the development board.</b></p>	<ol style="list-style-type: none"> <li>1. Navigate to your <code>/dev/</code> 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 <code>/dev/ttyUSB0</code> and the Cortex-M4 is <code>/dev/ttyUSB1</code>. <p style="text-align: center;"><b>Note</b></p> <p>If the new devices are not shown, your host PC has not identified the development board.</p> </li> <li>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: <pre>sudo chmod 666 /dev/ttyUSB</pre> </li> </ol>

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.

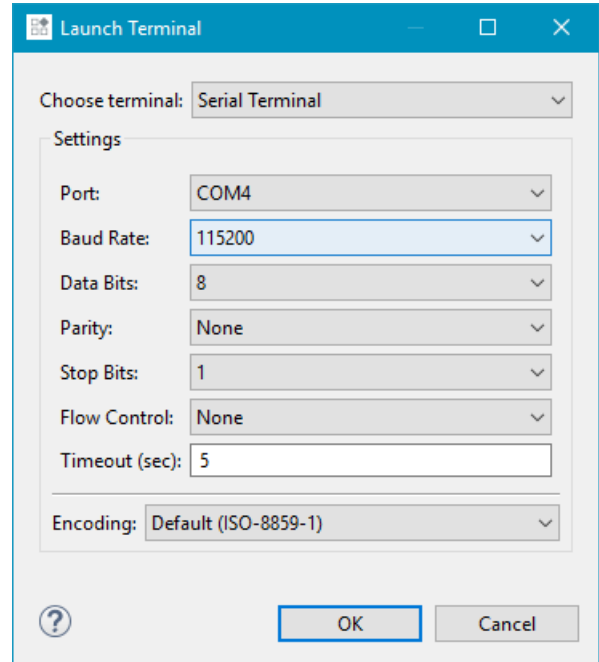
### Prerequisites

*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.

## 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.

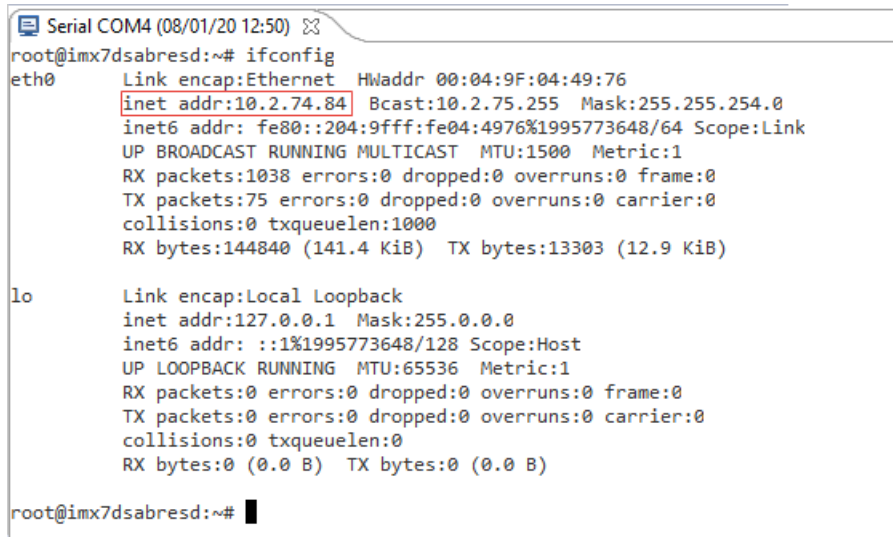
### Procedure

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:.



```

Serial COM4 (08/01/20 12:50)
root@imx7dsabresd:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:04:9F:04:49:76
          inet addr:10.2.74.84  Bcast:10.2.75.255  Mask:255.255.254.0
          inet6 addr: fe80::204:9fff:fe04:4976%1995773648/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1038 errors:0 dropped:0 overruns:0 frame:0
          TX packets:75 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:144840 (141.4 KiB)  TX bytes:13303 (12.9 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1%1995773648/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@imx7dsabresd:~#

```

Figure 2-5 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**.

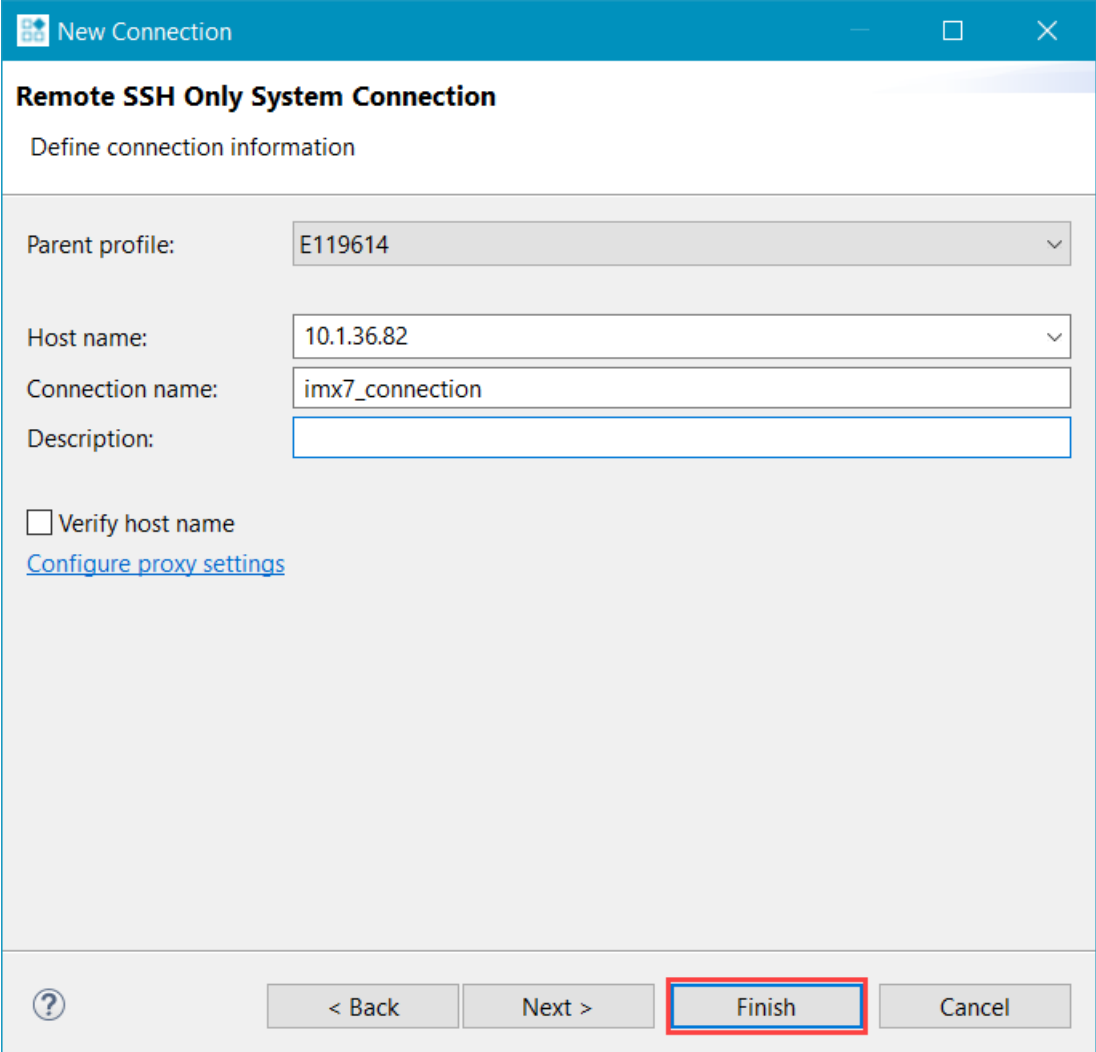


Figure 2-6 New Connection wizard, showing settings for an SSH Only RSE connection.



## 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 on page 3-27.\*](#)
- [\*3.2 Configure Arm® Debugger for Cortex®-M on page 3-29.\*](#)
- [\*3.3 Set up the Cortex®-A Linux application on page 3-32.\*](#)
- [\*3.4 Configure Arm® Debugger for Linux application debug on page 3-33.\*](#)


## 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.

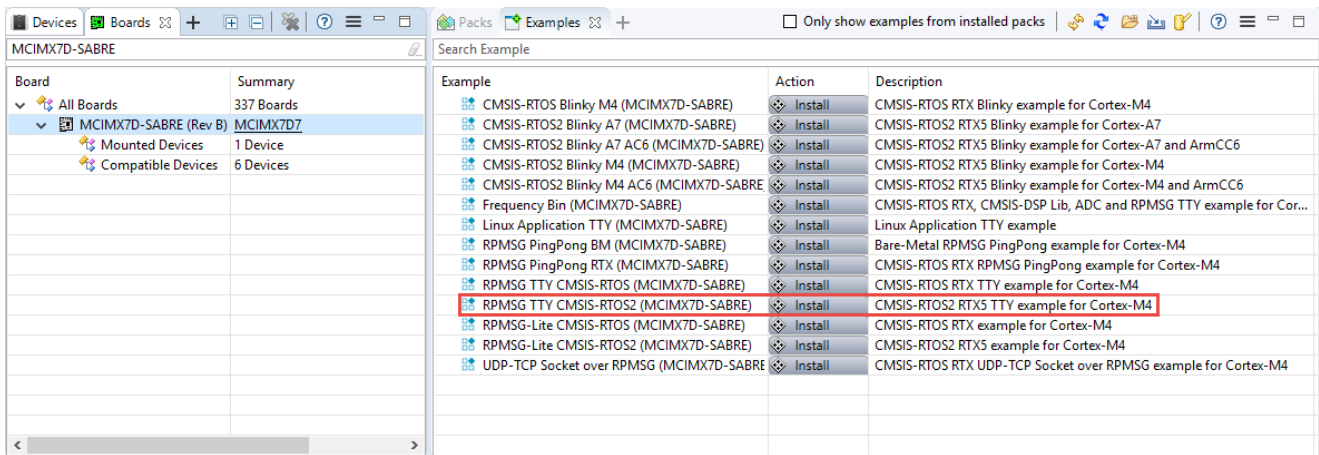


Figure 3-1 Installation of Cortex-M example project.

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.

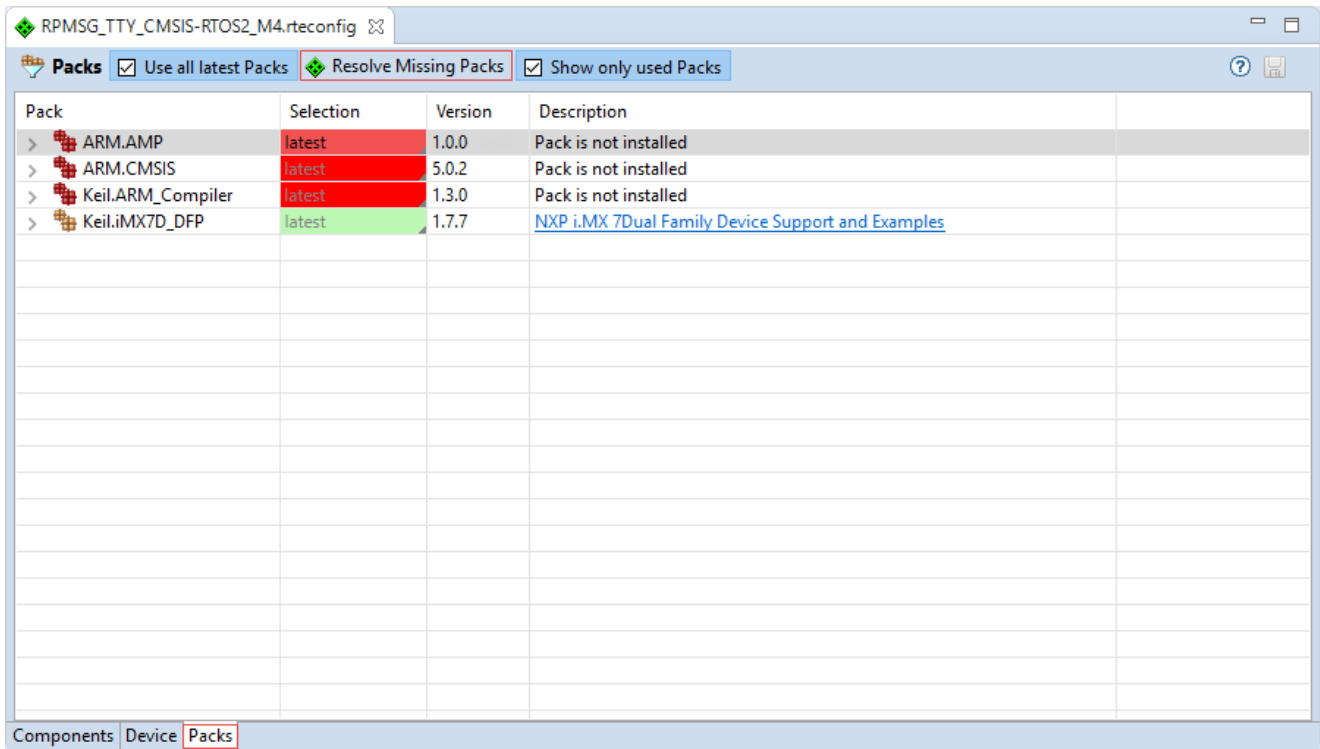


Figure 3-2 Installation of missing packs.

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

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

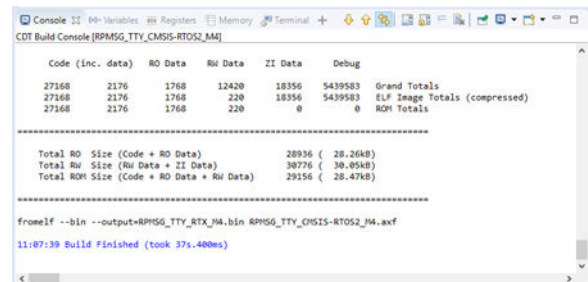


Figure 3-3 Console output of the Cortex-M build result.

## 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

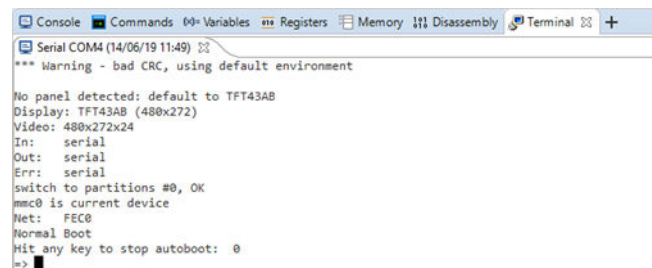
*Set up the Cortex-M application on page 3-27.*

### 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.



```

Serial COM4 (14/06/19 11:49)
*** Warning - bad CRC, using default environment

No panel detected: default to TFT43AB
Display: TFT43AB (480x272)
Video: 480x272x24
In: serial
Out: serial
Err: serial
switch to partitions #0, OK
mmc0 is current device
Net: FEC0
Normal Boot
Hit any key to stop autoboot: 0
=>

```

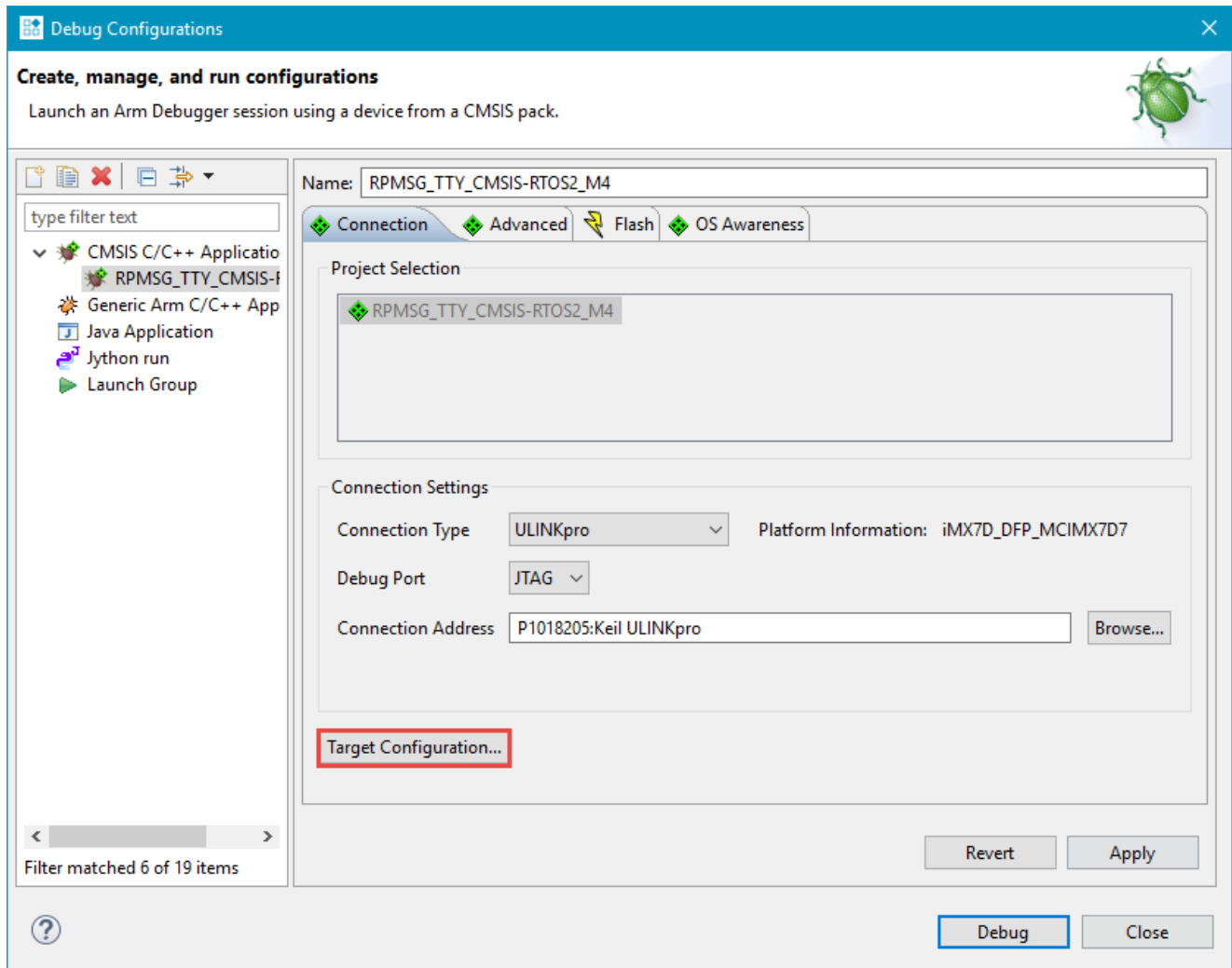
**Figure 3-4** Terminal view showing the boot of Linux.

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.**

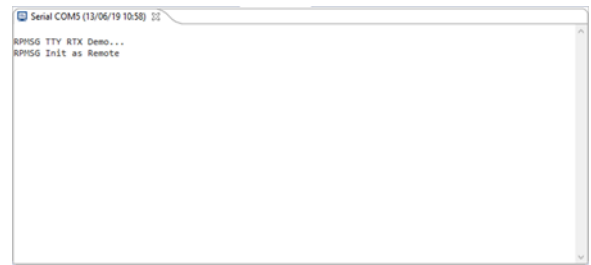
- a. Click the **Cortex-M4** tab and select **Enable Cortex-M4 core trace**.
- b. Click the **Cortex-A7** tab and disable all trace options to avoid buffer overflows.
- c. Click **OK** to confirm your changes and return to the **Debug Configurations** dialog box.
6. Click the **OS Awareness** tab and,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

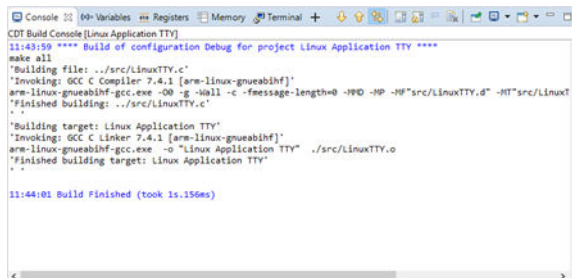
## Prerequisites

- 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.

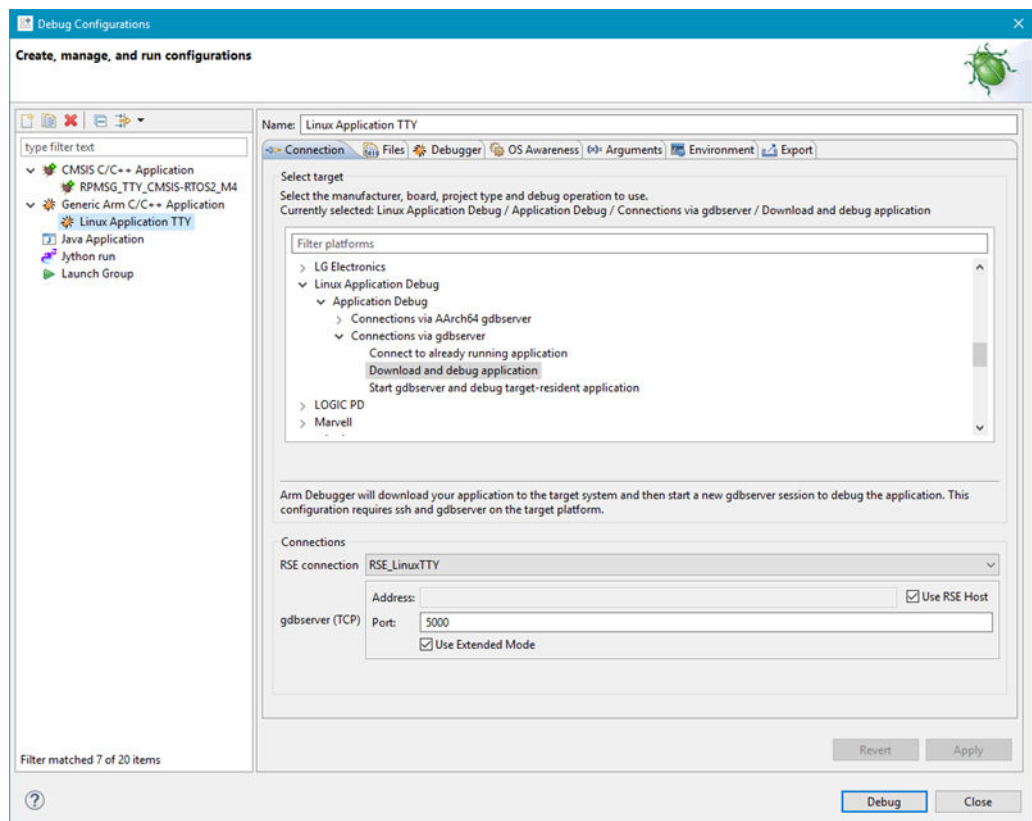


Figure 3-8 Debug Configurations dialog box.

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 .dtb 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!
```

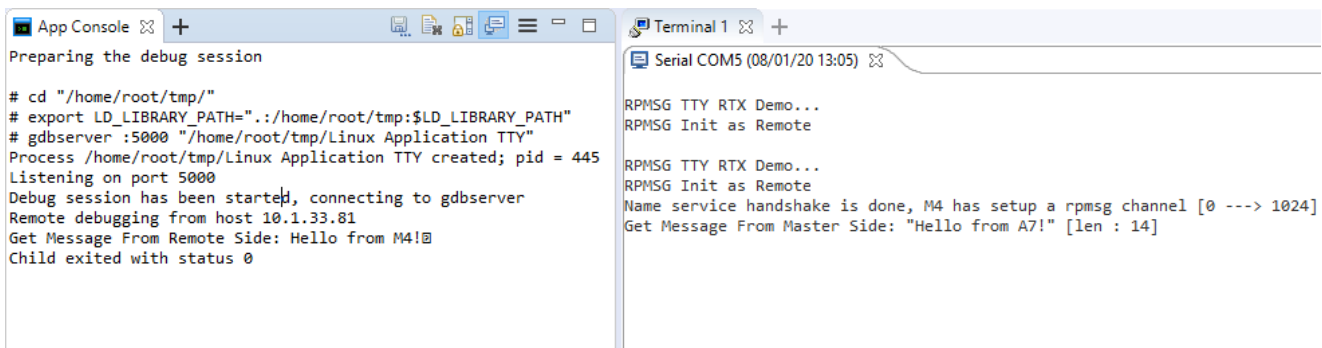
10. You can now run the Cortex-A Linux application. Click  in the **Debug Control** view.

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 5-58](#).

# 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 on page 4-36.](#)
- [4.2 Debug the Linux Application and Kernel on page 4-42.](#)
- [4.3 Debug the Linux Kernel Module on page 4-56.](#)

## 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 on page 4-36.](#)
- [4.1.2 Create the source code files on page 4-37.](#)
- [4.1.3 Adapt the scatter file on page 4-38.](#)
- [4.1.4 Debug the Cortex®-M Blinky application on page 4-40.](#)

### 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**
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**.

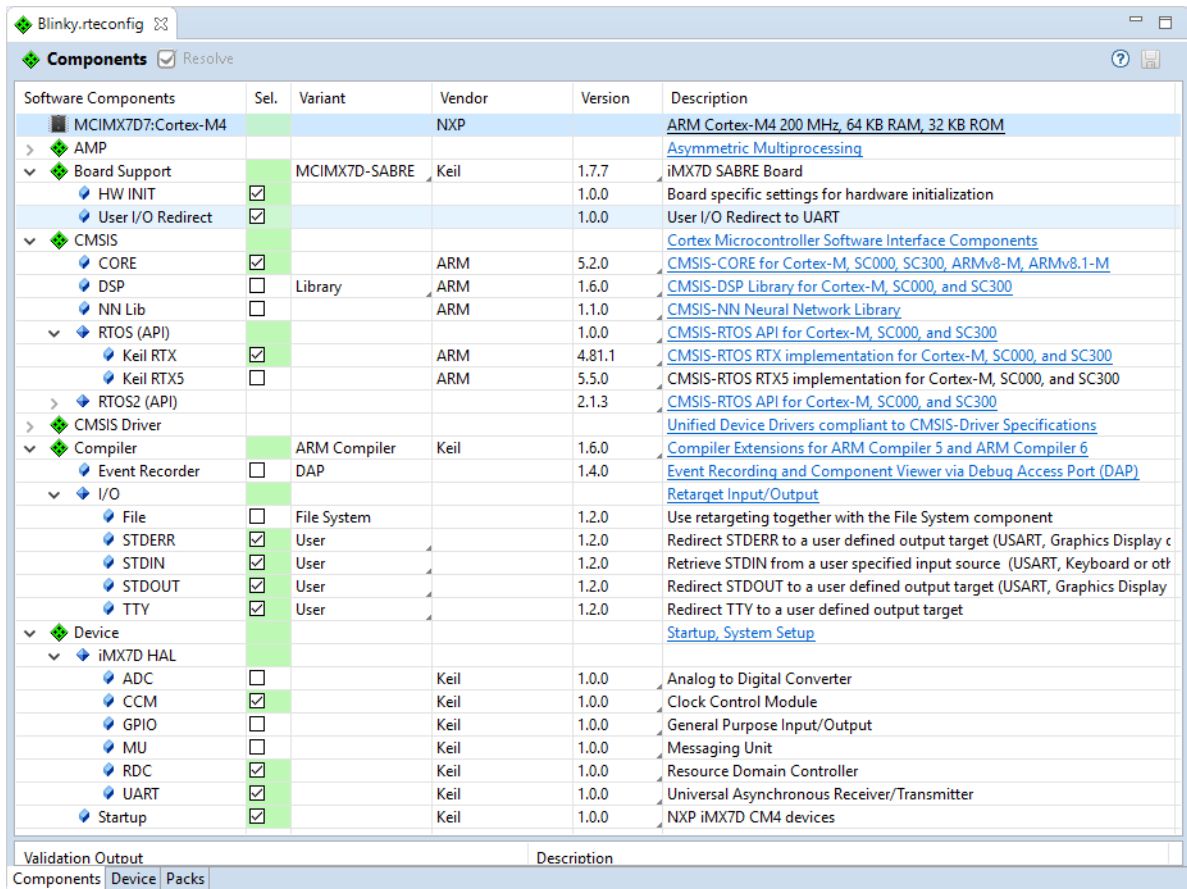



Figure 4-1 The selected components for the MCIMX7D7:Cortex-M4.

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 **RTOS 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 4-37.

### 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 4-38.](#)

### 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 4-37](#).

## 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.

```

CDT Build Console [Blinky]

Code (inc. data)  RO Data  RW Data  ZI Data  Debug
19144            1096      1732      60      7220    502579  Grand Totals
19144            1096      1732      60      7220    502579  ELF Image Totals
19144            1096      1732      60         0         0      ROM Totals

=====

Total RO Size (Code + RO Data)          20876 ( 20.39kB)
Total RW Size (RW Data + ZI Data)        7280 ( 7.11kB)
Total ROM Size (Code + RO Data + RW Data) 20936 ( 20.45kB)

=====

'Finished building target: Blinky.axf'
'
15:26:18 Build Finished (took 12s.675ms)

```

Figure 4-2 Console output showing the build results.

## Next Steps

*Create the Linux application for this project on page 4-42.*

### 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 (DTSI):
  - 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.
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 4-42.*

### **Related information**

*Variables view*

*Registers view*

*Disassembly view*

*Memory view*

*Breakpoints view*

## 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 on page 4-42.](#)
- [4.2.2 Debug the Hello World Linux application on page 4-42.](#)
- [4.2.3 Create the Linux kernel debug project on page 4-46.](#)
- [4.2.4 Configure Arm® Debugger for the Linux kernel debug project on page 4-48.](#)
- [4.2.5 Debug the Linux kernel: Pre-MMU stage on page 4-49.](#)
- [4.2.6 Debug the Linux kernel: Post-MMU stage on page 4-51.](#)

### 4.2.1 Create the Hello World Linux application

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

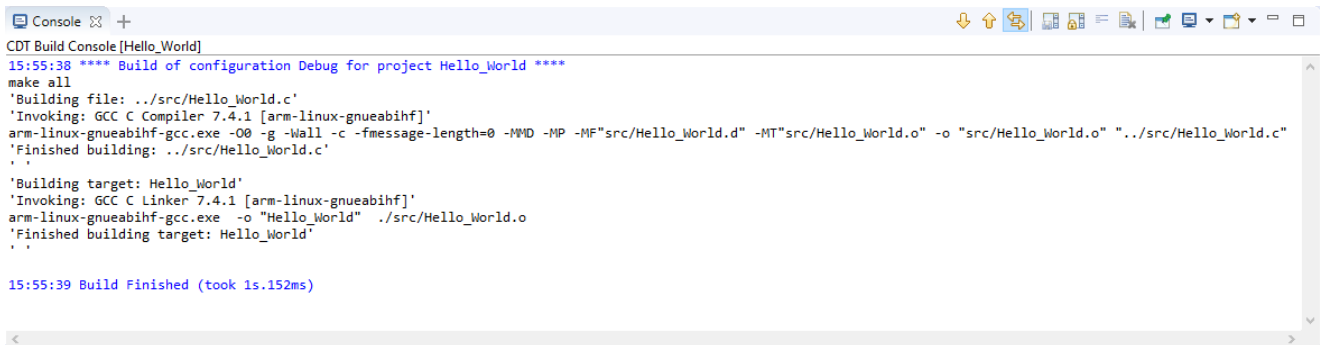
#### Prerequisites

- 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.



```
CDT Build Console [Hello_World]
15:55:38 **** Build of configuration Debug for project Hello_World ****
make all
'Building file: ../src/Hello_World.c'
'Invoking: GCC C Compiler 7.4.1 [arm-linux-gnueabi]'
arm-linux-gnueabi-gcc.exe -O0 -g -Wall -c -fmessage-length=0 -MMD -MP -MF"src/Hello_World.d" -MT"src/Hello_World.o" -o "src/Hello_World.o" "../src/Hello_World.c"
'Finished building: ../src/Hello_World.c'
'Building target: Hello_World'
'Invoking: GCC C Linker 7.4.1 [arm-linux-gnueabi]'
arm-linux-gnueabi-gcc.exe -o "Hello_World" ../src/Hello_World.o
'Finished building target: Hello_World'

15:55:39 Build Finished (took 1s.152ms)
```

Figure 4-3 Console output showing the build results.

#### Next Steps

[Debug your Linux application on page 4-42.](#)

### 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 4-42.](#)
- Determine the IP address of your target.

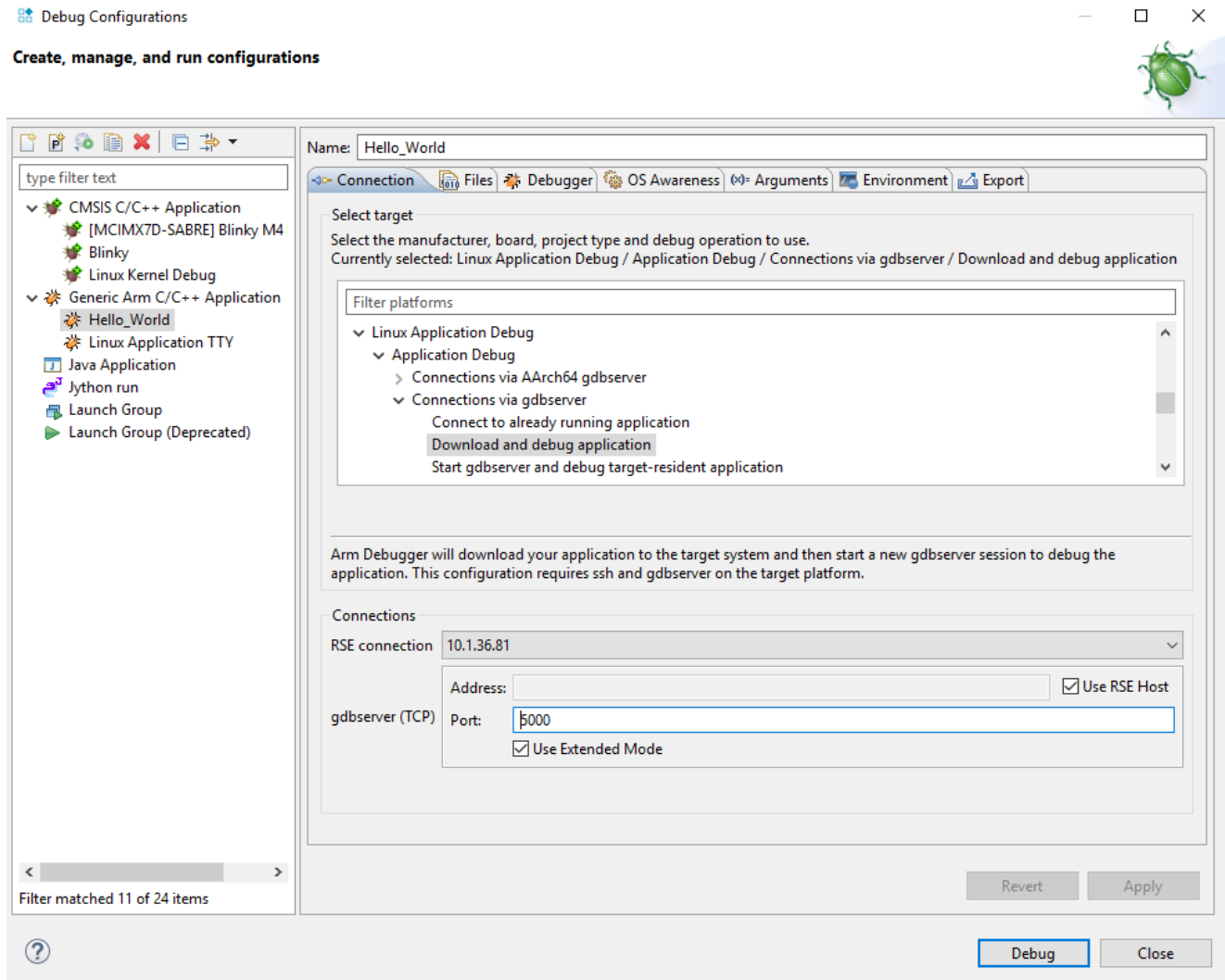
### Procedure

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**.
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** ————

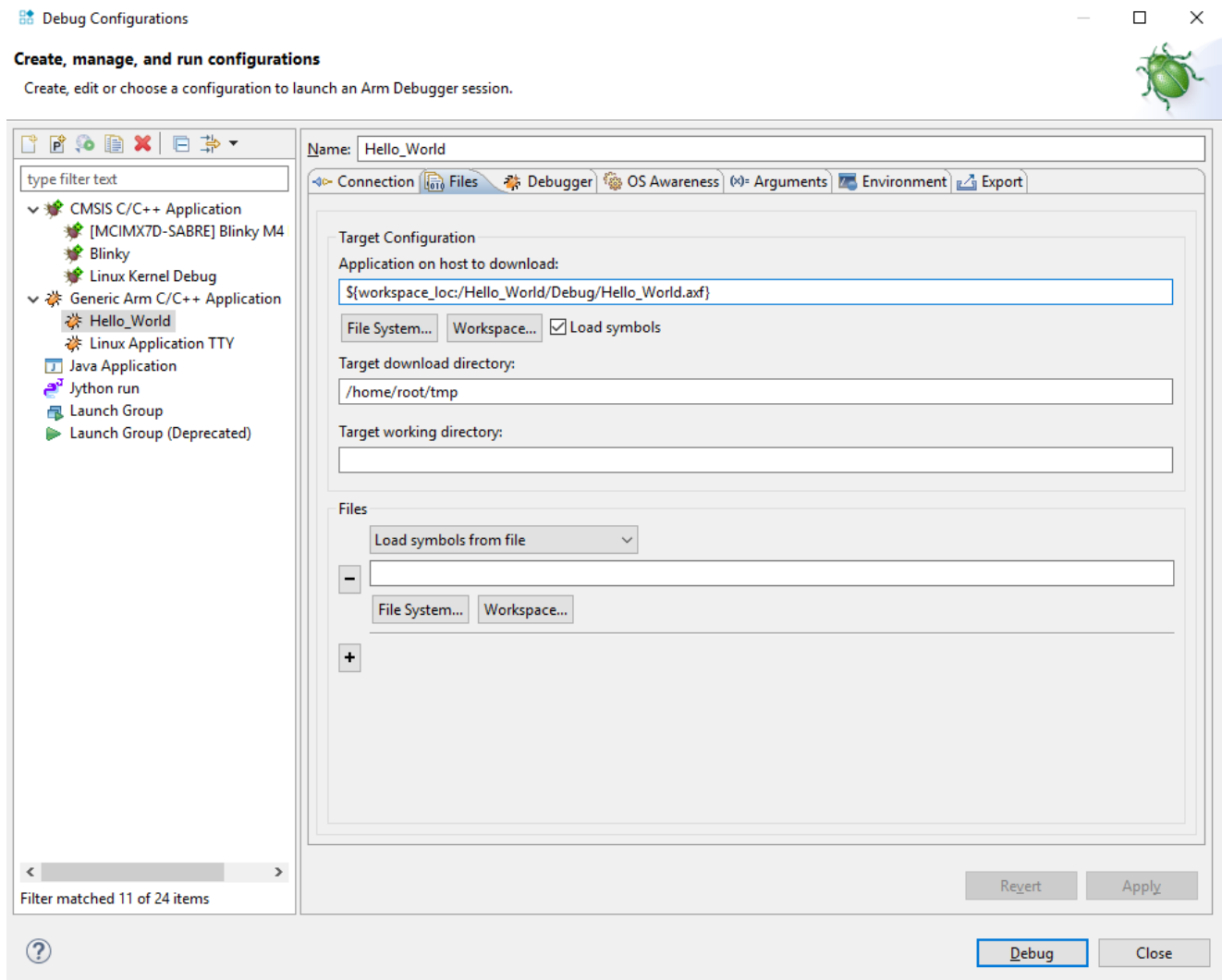
Your credentials might be different if you are not using the image downloaded from [Arm Developer](#).

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 4-4** 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 4-5** Debug Configurations dialog box showing Files tab settings.

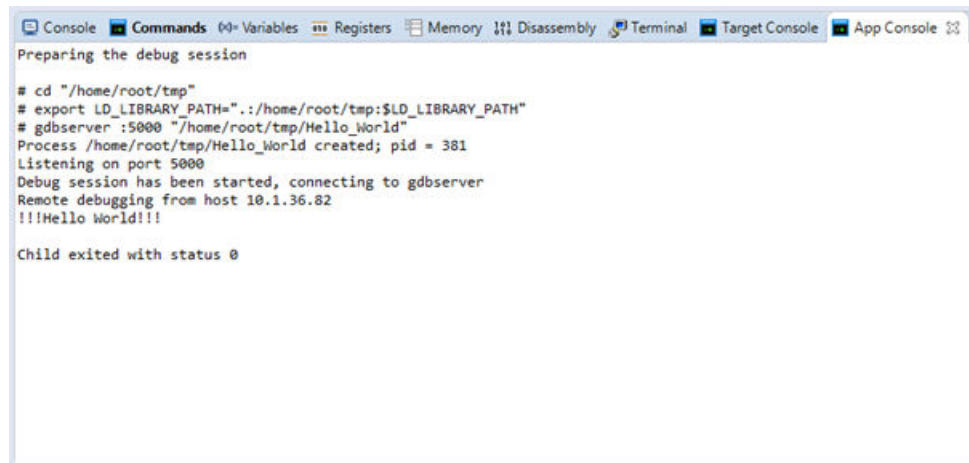
- f. Click the **Debugger** tab. Under **Run Control**, select **Debug** from symbol “main”. Click **Debug**.

**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:



```
Console Commands Variables Registers Memory Disassembly Terminal Target Console App Console
Preparing the debug session

# cd "/home/root/tmp"
# export LD_LIBRARY_PATH=".:/home/root/tmp:$LD_LIBRARY_PATH"
# gdbserver :5000 "/home/root/tmp/Hello_World"
Process /home/root/tmp/Hello_World created; pid = 381
Listening on port 5000
Debug session has been started, connecting to gdbserver
Remote debugging from host 10.1.36.82
!!!Hello World!!!

Child exited with status 0
```

Figure 4-6 Screenshot of the output of the Linux application.

## Next Steps

Create the Linux kernel debug project on page 4-46.

### 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**.

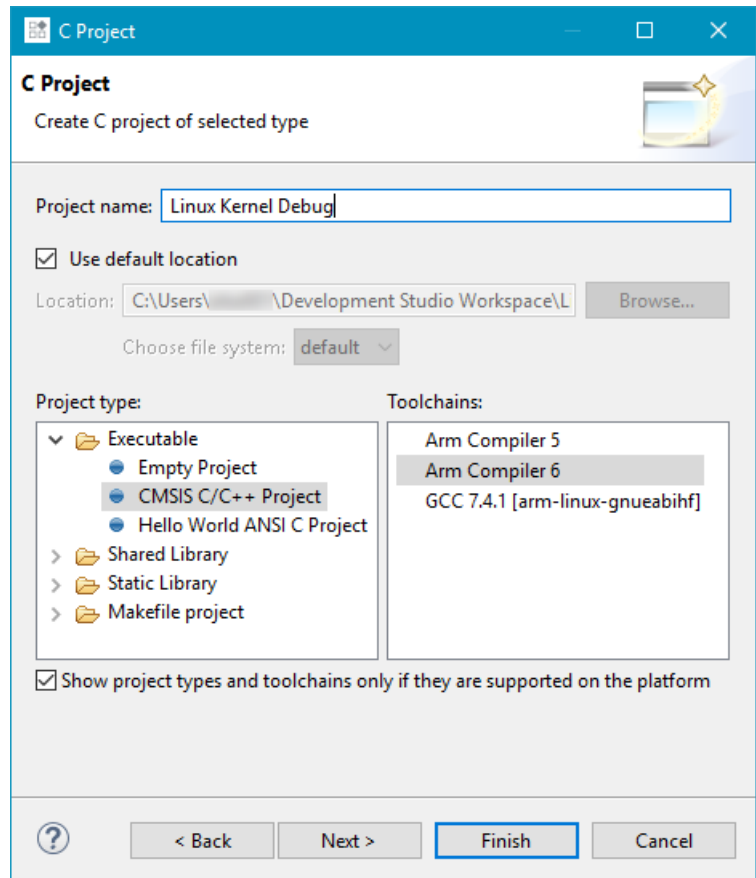


Figure 4-7 Screenshot of settings for the CMSIS C/C++ project.

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

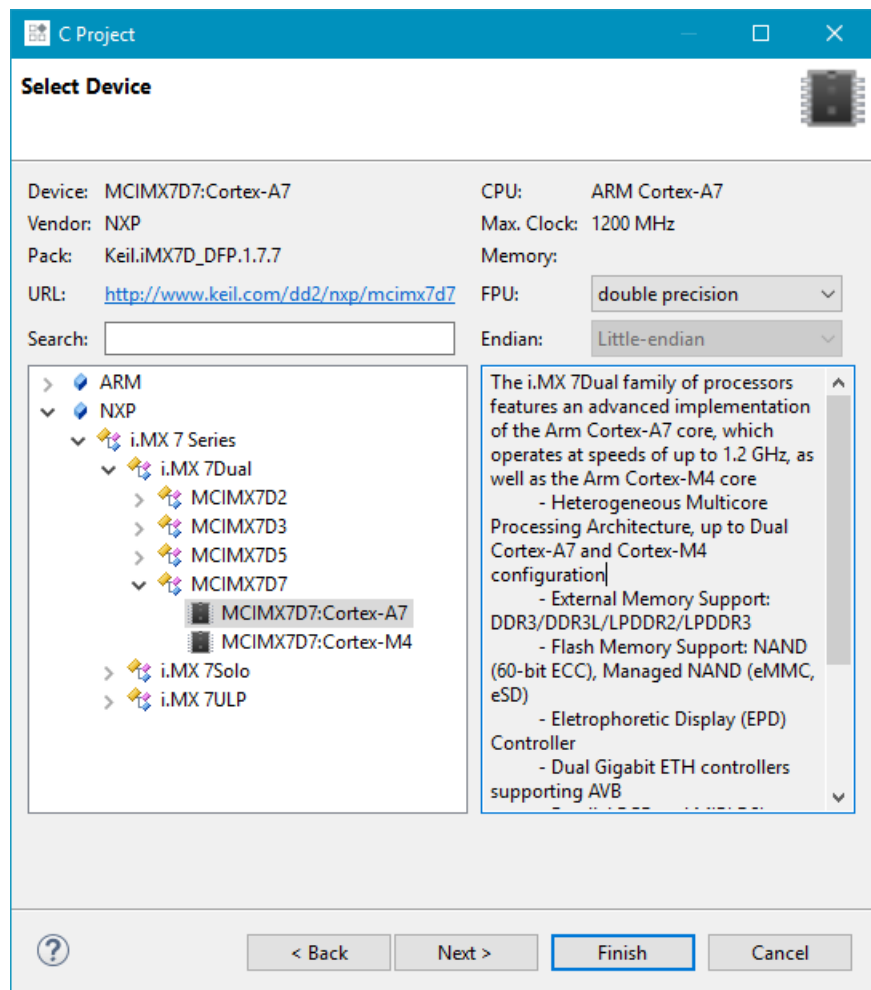


Figure 4-8 Screenshot of MCIMX7D7:Cortex-A7 selected.

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 4-48.*

#### 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 4-46](#).

## 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`.
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`.

## Next Steps

[Debug the Linux kernel: Pre-MMU stage on page 4-49](#).

### 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

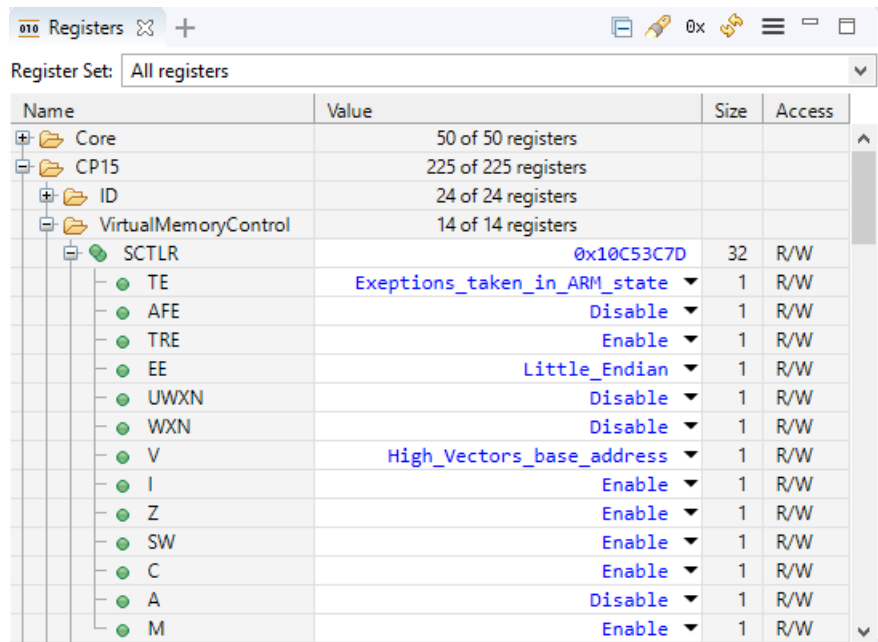
You must [Configure Arm® Debugger for the Linux kernel debug project on page 4-48](#).

## 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**.
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 4-9 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.

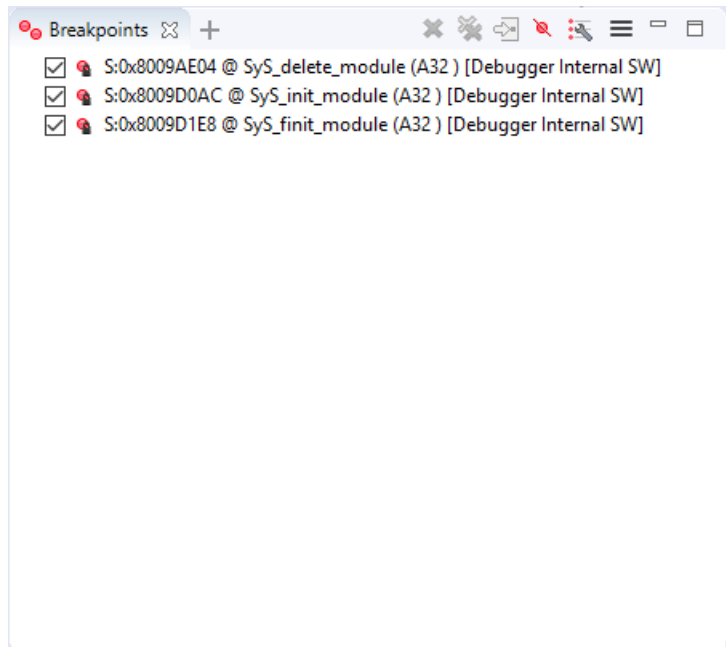


Figure 4-10 Breakpoints view showing automatically created breakpoints.

### Next Steps

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


## 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 4-49.*

### 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()`.

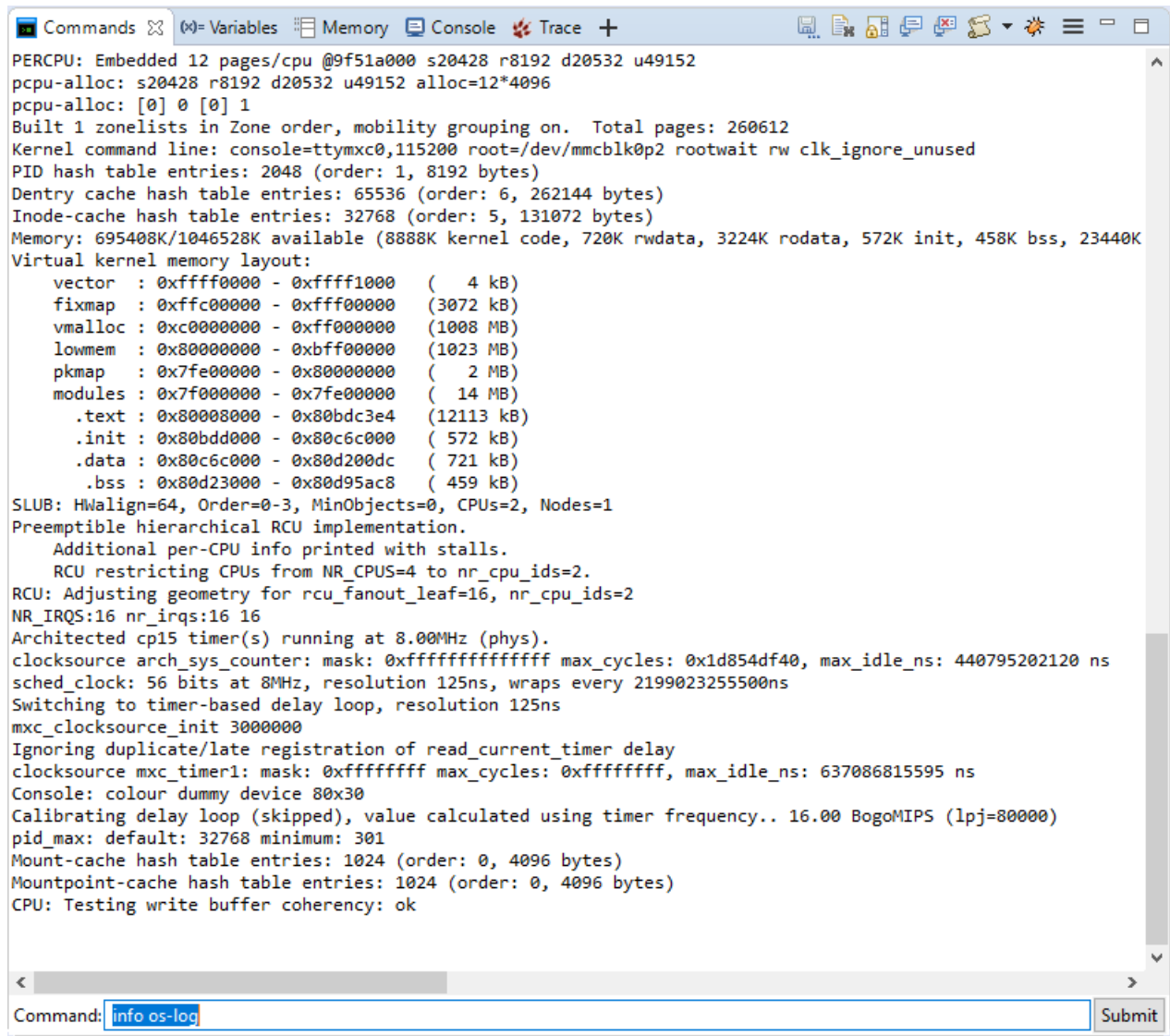
A useful feature during kernel bring-up is to display early `printk` output in the command window of Arm Debugger.

————— **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
```



```

PERCPU: Embedded 12 pages/cpu @9f51a000 s20428 r8192 d20532 u49152
pcpu-alloc: s20428 r8192 d20532 u49152 alloc=12*4096
pcpu-alloc: [0] 0 [0] 1
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 260612
Kernel command line: console=ttyMXC0,115200 root=/dev/mmcblk0p2 rootwait rw clk_ignore_unused
PID hash table entries: 2048 (order: 1, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 695408K/1046528K available (8888K kernel code, 720K rwdata, 3224K rodata, 572K init, 458K bss, 23440K
Virtual kernel memory layout:
  vector : 0xffff0000 - 0xffff1000 ( 4 kB)
  fixmap : 0xffc00000 - 0xffff0000 (3072 kB)
  vmalloc : 0xc0000000 - 0xff000000 (1008 MB)
  lowmem : 0x80000000 - 0xbff00000 (1023 MB)
  pkmap : 0x7fe00000 - 0x80000000 ( 2 MB)
  modules : 0x7f000000 - 0x7fe00000 ( 14 MB)
  .text : 0x80008000 - 0x80bdc3e4 (12113 kB)
  .init : 0x80bdd000 - 0x80c6c000 ( 572 kB)
  .data : 0x80c6c000 - 0x80d200dc ( 721 kB)
  .bss : 0x80d23000 - 0x80d95ac8 ( 459 kB)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=2, Nodes=1
Preemptible hierarchical RCU implementation.
  Additional per-CPU info printed with stalls.
  RCU restricting CPUs from NR_CPUS=4 to nr_cpu_ids=2.
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=2
NR_IRQS:16 nr_irqs:16 16
Architected cp15 timer(s) running at 8.00MHz (phys).
clocksource arch_sys_counter: mask: 0xffffffffffffff max_cycles: 0x1d854df40, max_idle_ns: 440795202120 ns
sched_clock: 56 bits at 8MHz, resolution 125ns, wraps every 219902325500ns
Switching to timer-based delay loop, resolution 125ns
mxc_clocksource_init 3000000
Ignoring duplicate/late registration of read_current_timer delay
clocksource mxc_timer1: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 637086815595 ns
Console: colour dummy device 80x30
Calibrating delay loop (skipped), value calculated using timer frequency.. 16.00 BogoMIPS (lpj=80000)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 1024 (order: 0, 4096 bytes)
Mountpoint-cache hash table entries: 1024 (order: 0, 4096 bytes)
CPU: Testing write buffer coherency: ok


```

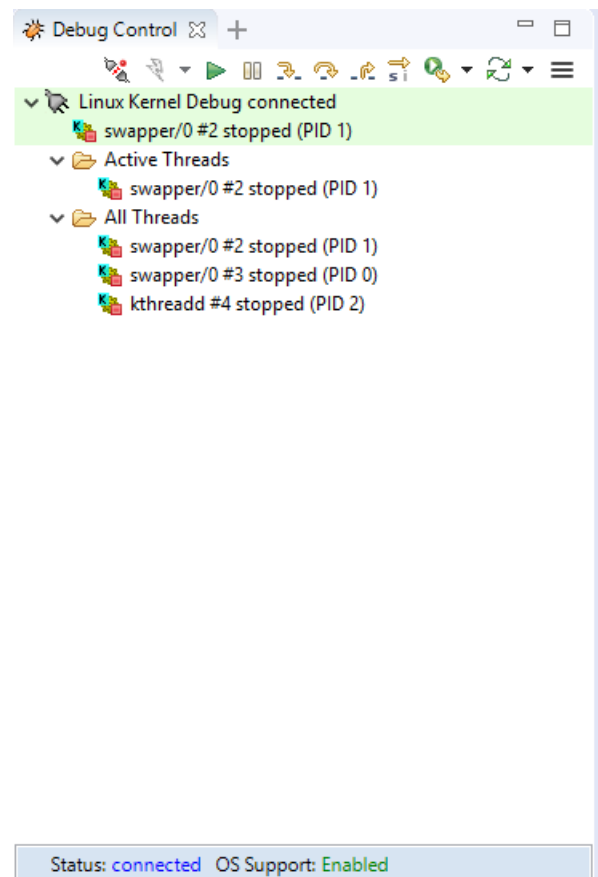
Command:

Figure 4-11 Commands view showing output of info os-log command.

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

```
set os log-capture on
```
- 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.
- 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.
- Delete all user breakpoints and continue by clicking . Let the kernel run all the way to the Login prompt. Log in as root.
- 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 4-12 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 .

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.

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	Cc
(struct thread_info*)(\$sp_svc & ~0x1FFF)	0x880B4000	struct thread_info*	^
flags	0	long unsigned int	
preempt_count	0	int	
addr_limit	0	mm_segment_t	
task	0x880B0000	struct task_struct*	
state	0	volatile long int	
stack	0x880B4000	void*	
usage		atomic_t	
flags	2097216	unsigned int	
ptrace	0	unsigned int	
wake_entry		struct llist_node	
on_cpu	1	int	
last_wakee	0x00000000	struct task_struct*	
wakee_flips	0	long unsigned int	
wakee_flip_decay_ts	0	long unsigned int	
wake_cpu	0	int	
on_rq	1	int	
prio	120	int	
static_prio	120	int	
normal_prio	120	int	
rt_priority	0	unsigned int	
sched_class	0x80880D4C	const struct sched_class*	
se		struct sched_entity	
rt		struct sched_rt_entity	
dl		struct sched_dl_entity	
policy	0	unsigned int	
nr_cpus_allowed	1	int	
cpus_allowed		cpumask_t	
rcu_read_lock_nesting	0	int	
rcu_read_unlock_special		union rcu_special	
rcu_node_entry		struct list_head	
rcu_blocked_node	0x00000000	struct rcu_node*	
tasks		struct list_head	
pushable_tasks		struct plist_node	
pushable_dl_tasks		struct rb_node	
mm	0x00000000	struct mm_struct*	
active_mm	0x80C83F38	struct mm_struct*	
vmacache_seqnum	0	u32	
vmacache		struct vm_area_struct*[]	
rss_stat		struct task_rss_stat	
exit_state	0	int	
exit_code	0	int	
exit_signal	0	int	
pdeath_signal	0	int	
jobctl	0	unsigned int	
personality	0	unsigned int	
in_execve	0	unsigned int	
in_jowait	0	unsigned int	
sched_reset_on_fork	0	unsigned int	
sched_contributes_to_load	0	unsigned int	
atomic_flags	0	long unsigned int	
restart_block		struct restart_block	
pid	1	pid_t	
tgid	1	pid_t	
real_parent	0x80C71BB0	struct task_struct*	
parent	0x80C71BB0	struct task_struct*	
children		struct list_head	
...		...	



Figure 4-13 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 4-56.

## 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 4-56.](#)

### 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 4-42.](#)

#### 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 5-58.*

# 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 \(BIN\) on page 5-59.](#)
- [5.2 Store Cortex®-M BIN file on SD Card on page 5-60.](#)
- [5.3 Run Cortex®-M BIN file from U-Boot on page 5-61.](#)

## 5.1 Create a Cortex®-M binary image (BIN)

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

The `:doc:`Blinky <../C3debug own/create-cortex-m-applications>`` project 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 5-60](#).

## 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 5-59](#).

### 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 5-61](#).

## 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 5-60.](#)

### 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.