Working with Cortex-M4 on i.MX6 SoloX COM Board

Embedded Artists AB

Jörgen Ankersgatan 12
SE-211 45 Malmö
Sweden

http://www.EmbeddedArtists.com

Feedback

We appreciate any feedback you may have for improvements on this document. Send your comments by using the contact form: www.embeddedartists.com/contact.

Table of Contents

1 Document Revision History 5

2 Introduction 6

2.1 Multi-Core.. 6

2.2 Additional Documentation.. 6

2.3 Conventions... 6

3 Hardware Related.. 7

3.1 Prerequisites.. 7

3.2 UART Interfaces on COM Carrier board version 1 7

3.2.1 Applications for Freescale Sabre Board 8

3.3 UART interfaces on COM Carrier board version 2 8

3.4 Terminal application 9

4 Download and Start an Application.................. 10

4.1 Update boot partition with needed files 10

4.2 Change the device tree file 11

4.3 Run from QSPI... 11

4.4 Run from TCM.. 11

4.5 Run from OCRAM 12

4.6 Run from DDR RAM 13

5 Remote communication applications (RPMsg) 14

5.1 Ping-pong application.. 14

6 FreeRTOS 15

6.1 Installation 15

6.1.1 File Structure 15

6.2 Board Support Package (BSP) 15

6.2.1 UART... 15

6.3 Build with ARM DS-5... 16

6.3.1 BSP files.. 18

6.4 Debug using DS-5 18

6.4.1 Setup the hardware 18

6.4.2 Import TCM version of “hello world” 19

6.4.3 Create a new Debug configuration 19

6.5 Build with ARM GCC 22

6.5.1 Install ARM GCC 22

6.5.2 Install MinGW 22

6.5.3 Install CMake... 25

6.5.4 Build Application 26

6.6 Build with Eclipse and ARM GCC 26

6.6.1 Install “GNU ARM Eclipse” plugins 27

6.6.2 Create project: New ... 27
6.6.3 Create project: Linked folders ... 28
6.6.4 Create project: Exclude from build ... 30
6.6.5 Create project: “Include” paths ... 31
6.6.6 Create project: Settings ... 32
6.6.7 Build application ... 36

6.7 Debug using Eclipse ... 36

6.7.1 LPC-Link 2 with J-Link firmware ... 36
6.7.2 J-Link GDB Server ... 36
6.7.3 J-Link script files ... 36
6.7.4 Connect LPC-Link 2 to the board ... 36
6.7.5 Create a debug configuration ... 38
6.7.6 Start a debug session ... 39

6.8 Build with IAR Embedded Workbench ... 41

7 Use DS-MDK for Application Development ... 42

7.1 Installation ... 42

7.2 Package Manager ... 42

7.3 UART Pin Muxing ... 43

7.4 Debug the M4 Application ... 43

7.4.1 Build the application ... 43
7.4.2 Setup the debug adapter ... 43
7.4.3 Create a debug configuration ... 44

7.5 Debug the Linux Application ... 47

7.5.1 Build the application ... 47
7.5.2 Setup Remote System Explorer (RSE) ... 47
7.5.3 Create Debug Configuration ... 49

7.6 Simultaneous Debugging ... 52

8 Troubleshooting ... 53

8.1 JTAG connection problem when Linux has booted ... 53

8.1.1 Description of problem ... 53
8.1.2 Solution ... 53

8.2 Allow user “root” to use an SSH connection ... 53

8.3 Linux (A9) terminal/console doesn’t accept input while debugging M4 ... 54

1 Document Revision History

Revision	*Date*	*Description*
A	2015-11-11	First release
B	2016-01-20	- Added description about how to build **FreeRTOS**: Chapter **Error! Reference source not found.** and Chapter 6 - Updated Chapter 4 to describe how to load an application to TCM, OCRAM, and DDR memory.
C	2016-09-02	Added chapter 8 (troubleshooting)
D	2017-03-06	- Added section 6.6 describing how to build using Eclipse - Added section 6.7 describing how to debug using Eclipse
E	2017-04-25	- Added chapter 7 describing how to use DS-MDK
F	2017-09-22	- Removed chapter about MQX. FreeRTOS is recommended to use. - Minor updates and clarifications to other sections. - Added section 8.3
G	2020-11-05	- Updated instructions with regard to the COM Carrier board V2. - Major updates to chapter 4

2 Introduction

This document provides you with step-by-step instructions for how to work with the Cortex-M4 microcontroller on the iMX6 SoloX COM Board (**EAC00244**). The iMX6 SoloX Developer's Kit (**EAK00245**) has been used when writing these instructions.

2.1 Multi-Core

The i.MX6 SoloX processor has two cores; one ARM Cortex-A9 core and one ARM Cortex-M4 core. This is also known as heterogeneous multiprocessing (HMP). When developing and application that will utilize both these cores there are a number of things you need to be aware of.

- Both cores might have access to peripheral blocks in the processor. For your application you have to decide which core that is responsible for a peripheral. This decision can affect, for example, the device tree (dtb) file used by Linux when initializing device drivers.
    - In the instructions a specific dtb file will be used that disable some peripherals conflicting with Cortex-M4
- Cortex-A9 is always the primary core that is the first to boot and responsible for starting Cortex-M4. This is done by the u-boot in our examples
- The Cortex-M4 application must be stored on the QSPI flash. In the examples the u-boot will write the application image to QSPI flash
- There are ways to communication between the cores. Chapter **Error! Reference source not found.**describes how to run an application that utilizes Multi-Core Communication (MCC).

2.2 Additional Documentation

Additional recommended documentation:

- *Getting Started with the i.MX6 SoloX Developer's Kit* – shows you how to get started with the hardware.

2.3 Conventions

A number of conventions have been used throughout to help the reader better understand the content of the document.

`Constant width text` – is used for file system paths and command, utility and tool names.

```
$ This field illustrates user input in a terminal running on the
development workstation, i.e., on the workstation where you edit,
configure and build Linux
```

```
# This field illustrates user input on the target hardware, i.e.,
input given to the terminal attached to the COM Board
```

```
This field is used to illustrate example code or excerpt from a
document.
```

This field is used to highlight important information

3 Hardware Related

3.1 Prerequisites

To be able to follow all the instructions in this document the following is required.

- One iMX6 SoloX Developer's Kit (**EAK00331**, **EAK00245**)
- If using the Developer's Kit version 1 (V1) you need two FTDI cables for console output/input from both the Cortex-A9 and the Cortex-M4. Please note that **only one cable** is included with the Developer's Kit V1. If you are using a Developer's Kit version 2 (V2) you don't need any FTDI cables.
- One Debug interface board with 10-pos FPC cable (included with Developer's Kit). Only needed when debugging with ARM DS-5 as described in section 6.4
- Keil ULINK-Pro. Only needed when debugging with ARM DS-5 as described in section 6.4
- ARM DS-5 commercial license. Only needed when debugging with ARM DS-5 as described in section 6.4

3.2 UART Interfaces on COM Carrier board version 1

Two consoles are needed when working with both the Cortex-A9 (running Linux) and the Cortex-M4 microcontroller. Connector J35 is used by Cortex-A9 and connector J15 is used by Cortex-M4 as shown in Figure 1 below.

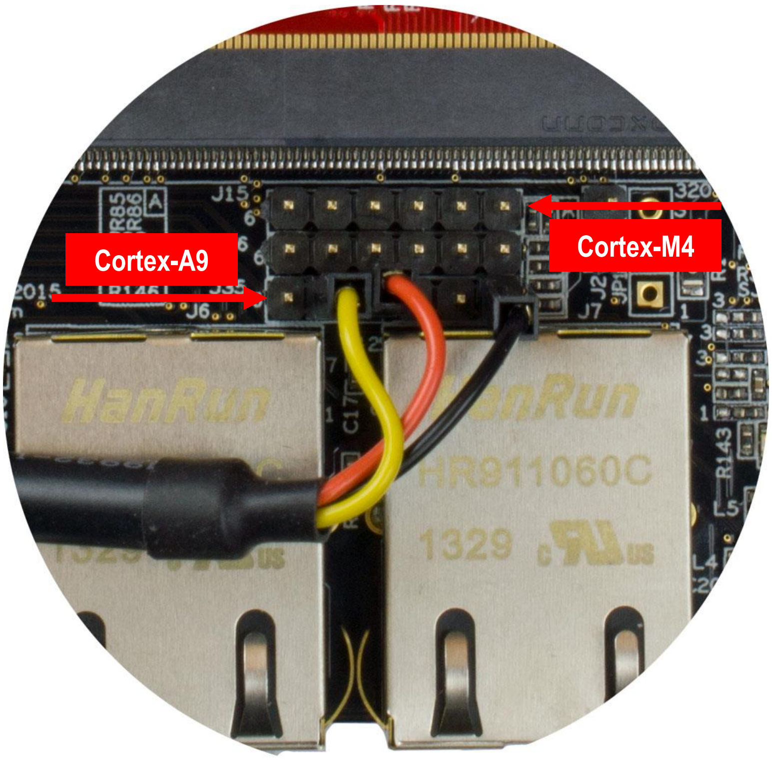

Figure 1 – COM Carrier board V1, UART connectors

3.2.1 Applications for Freescale Sabre Board

If you are testing pre-compiled applications developed for the Freescale Sabre board then console output will be available on different pins, that is, not on J15 connector. UART2 is used, but on pins that are available on the XBee connector (J17), see Figure 2.

- Pin 4 – RX on board, TX on FTDI cable (yellow)
- Pin 9 – TX on board, RX on FTDI cable (orange)
- Pin 10 – Ground (black)

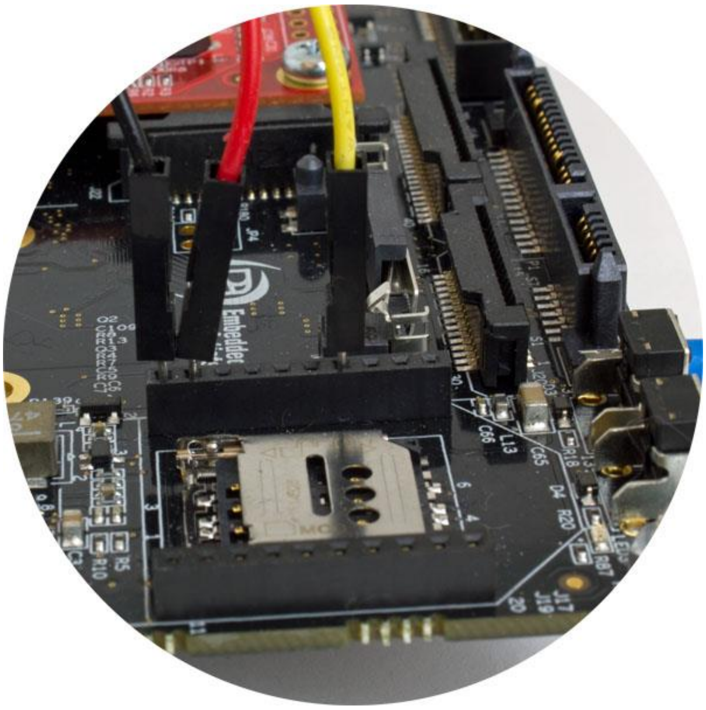

Figure 2 - UART2 on XBee connector

3.3 UART interfaces on COM Carrier board version 2

The COM Carrier board version 2 has a dual channel UART-to-USB bridge, meaning that you will get two UART interfaces via one USB cable connected between the micro-B USB connector (J16) on the carrier board and your PC.

There are jumpers on the carrier board that lets you select which UART interface that is connected to the UART-to-USB bridge, see Figure 3. Jumpers J19/J20 let you select between using UART-A or UART-C as console for the Cortex-A side. By default, these jumpers select the UART-A interface, that is, jumpers are in upper position. This is the position they should have for the iMX6 SoloX.

Jumpers J17/18 lets you select between using UART-B or UART-C as console for the Cortex-M side. By default, these jumpers are not inserted, but they **should be in upper position** for the iMX6 SoloX.

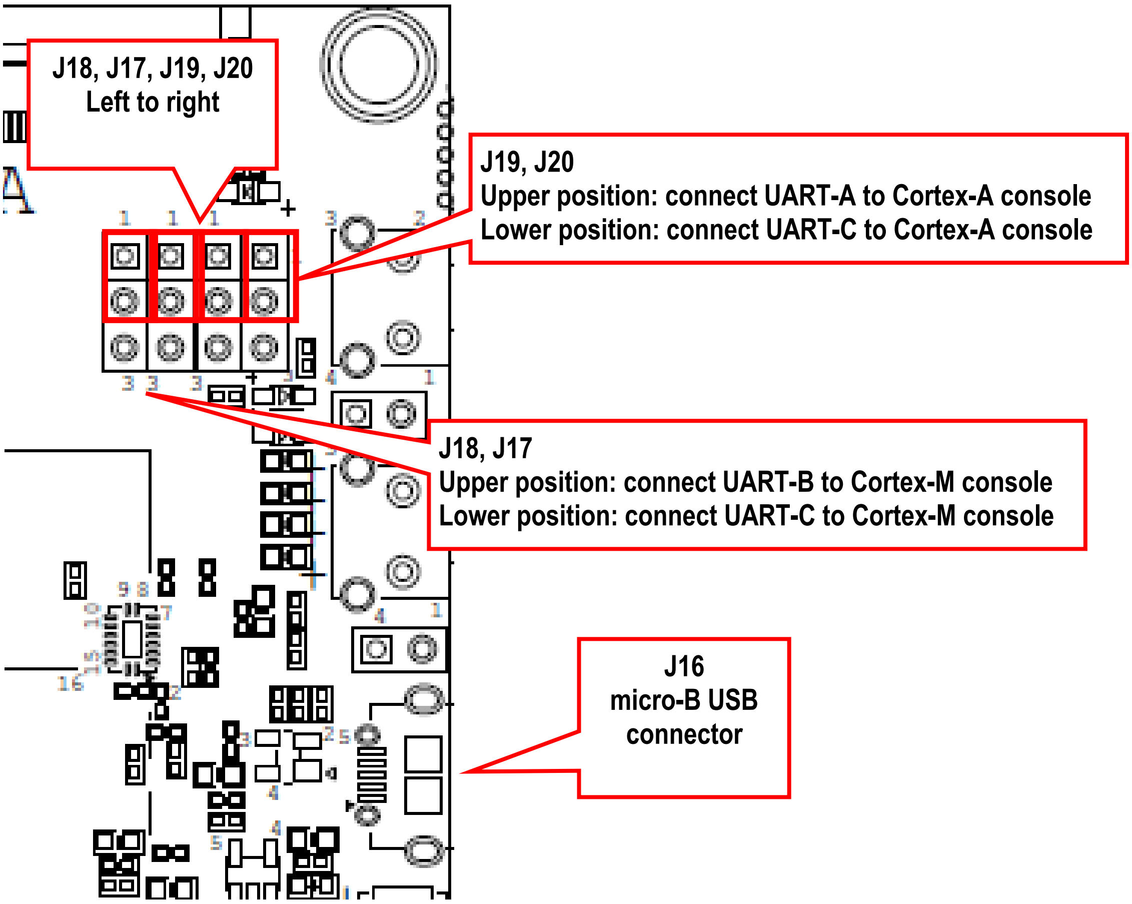

Figure 3 - COM Carrier board V2, UART interface connectors

3.4 Terminal application

You need a terminal application (two instances of it to connect both to the Cortex-A side and the Cortex-M side). We recommend Tera Term, but you can use the terminal application of your choice. Connect to the virtual COM ports using 115200 as baud rate, 8 data bits, 1 stop bit, and no parity.

4 Download and Start an Application

This section describes how to download and start a pre-compiled application.

4.1 Update boot partition with needed files

The remaining parts of this chapter assumes that the first partition of the eMMC contains the pre-compile applications. If you have programmed your board using a UUU bundle from **2020-11-06** or later the files will already have been copied to the eMMC flash. If you have programmed using an older version and don't want to update you can follow these instructions.

> **Note**: It is not necessary to have the M4 applications on the eMMC, but for simplicity the following instructions in this chapter assumes they are.

Download pre-compile applications

Go to http://imx.embeddedartists.com and download the file `compiled_cortex_m4_apps.zip`.

Direct link: http://imx.embeddedartists.com/imx6sx/compiled_cortex_m4_apps.zip

Copy via USB memory stick

There are several ways to copy these pre-compiled files to the eMMC, but here we will use a USB memory stick.

1. Unpack the file `compiled_cortex_m4_apps.zip` file and copy the unpacked files to the USB memory stick. This is something you do on your computer.
2. Boot into Linux and insert the USB memory stick into the USB host port on the carrier board. You will see output like below in the console when inserting the USB memory stick. The most important part is the last line that lists the device name (**`sda1`**).

```
[   23.104504] usb 1-1.2: new high-speed USB device number 4 using ci_hdrc
[   23.165591] usb 1-1.2: New USB device found, idVendor=0781, idProduct=5406,
bcdDevice= 0.10
[   23.173972] usb 1-1.2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[   23.194511] usb 1-1.2: Product: U3 Cruzer Micro
[   23.199055] usb 1-1.2: Manufacturer: SanDisk Corporation
[   23.204371] usb 1-1.2: SerialNumber: 0000185A49619848
[   23.225447] usb-storage 1-1.2:1.0: USB Mass Storage device detected
[   23.264533] scsi host0: usb-storage 1-1.2:1.0
[   24.315418] scsi 0:0:0:0: Direct-Access     SanDisk  U3 Cruzer Micro  2.18 PQ:
0 ANSI: 2
[   24.334542] scsi 0:0:0:1: CD-ROM            SanDisk  U3 Cruzer Micro  2.18 PQ:
0 ANSI: 2
[   24.345768] sd 0:0:0:0: [sda] 8015505 512-byte logical blocks: (4.10 GB/3.82
GiB)
[   24.364543] sd 0:0:0:0: [sda] Write Protect is off
[   24.373248] sd 0:0:0:0: [sda] No Caching mode page found
[   24.378630] sd 0:0:0:0: [sda] Assuming drive cache: write through
[   24.443649]  sda: sda1
```

3. Mount the USB memory stick and eMMC partition. The USB memory stick has in this example the device name **`sda1`** as can be seen in the output in step 2 above. The partition on the eMMC that we will use is available at **`/dev/mmcblk2p1`**.

```
# mkdir /mnt/usb
# mount /dev/sda1 /mnt/usb
```

```
# mkdir /mnt/mmcboot
# mount /dev/mmcblk2p1 /mnt/mmcboot
```

4. Copy the bin file(s) from the USB memory stick to the boot partition. In this example we are only copying `m4_hello_tcm.bin`.

```
# cp /mnt/usb/m4_hello_tcm.bin /mnt/mmcboot/
```

5. Unmount the devices

```
# umount /mnt/usb
# umount /mnt/mmcboot
```

4.2 Change the device tree file

Some of the u-boot environment variables need to be updated.

1. You must have booted into the U-boot console.
2. Change the device tree file (dtb) to use by Linux.

```
=> setenv fdt_file imx6sxea-com-kit_v2-m4.dtb
=> saveenv
```

4.3 Run from QSPI

In this section the application is copied from eMMC to QSPI flash and then started.

Make sure you have built an application for QSPI or selected a pre-built application for QSPI (name ends with _qspi). The application file must have been copied to eMMC as described in section 4.1 above.

1. You must have booted into the U-boot console.
2. Set the M4 file name in the `m4image` variable.

```
=> setenv m4image m4_hello_qspi.bin
```

3. Copy the Cortex-M4 application to QSPI flash.

```
=> run update_m4_from_sd
```

4. Boot the M4 application.

```
=> run m4boot
```

> **Note**: If you have modified the `m4boot` variable as described in the sections below you can revert back to the default setting (for QSPI booting) by running `env default -a`.

4.4 Run from TCM

Make sure you have built an application for TCM or selected a pre-built application for TCM (name ends with tcm). The application file must have been copied to eMMC as described in section 4.1 above.

1. You must have booted into the U-boot console.

2. Set the M4 file name in the `m4image` variable.

```
=> setenv m4image m4_hello_tcm.bin
```

3. Set the address where the application will run from (TCM memory in this case).

```
=> setenv m4runaddr 0x7f8000
```

4. Update the `m4boot` variable so it loads the image from eMMC to DDR memory, copies from DDR memory to TCM memory and then boots the application.

```
=> setenv m4boot 'run loadm4image; cp.b ${loadaddr} ${m4runaddr}
${filesize}; bootaux ${m4runaddr}'
```

5. Save the changes

```
=> saveenv
```

6. Boot the M4 application.

```
=> run m4boot
```

4.5 Run from OCRAM

Make sure you have built an application for OCRAM or selected a pre-built application for OCRAM (name ends with _ocram). The application file must have been copied to eMMC as described in section 4.1 above.

1. You must have booted into the U-boot console.
2. Set the M4 file name in the `m4image` variable.

```
=> setenv m4image m4_hello_ocram.bin
```

3. Set the address where the application will run from (OCRAM memory in this case).

```
=> setenv m4runaddr 0x910000
```

4. Update the `m4boot` variable so it loads the image from eMMC to DDR memory, copies from DDR memory to OCRAM memory and then boots the application.

```
=> setenv m4boot 'run loadm4image; cp.b ${loadaddr} ${m4runaddr}
${filesize}; bootaux ${m4runaddr}'
```

5. Save the changes.

```
=> saveenv
```

6. Boot the M4 application.

```
=> run m4boot
```

4.6 Run from DDR RAM

Make sure you have built an application for DDR RAM or selected a pre-built application for DDR RAM (name ends with _ddr). The application file must have been copied to eMMC as described in section 4.1 above.

1. You must have booted into the U-boot console.
2. Set the M4 file name in the `m4image` variable.

```
=> setenv m4image m4_hello_ddr.bin
```

3. Set the address where the application will run from (DDR memory in this case).

```
=> setenv m4runaddr 0x9ff00000
```

4. The default `loadm4image` variable will load to the address set in `loadaddr` variable. We don't want to set `loadaddr` to the same address as used by the M4 application since `loadaddr` will also be used when loading the kernel. Instead we create a new `loadm4image_ddr` variable that will load the application directly to the address where it will be started.

```
=> setenv loadm4image_ddr 'fatload mmc ${mmcdev}:${mmcpart}
${m4runaddr} ${m4image}'
```

5. Update the `m4boot` variable so it loads the image from eMMC to DDR memory and then boots the application.

```
=> setenv m4boot 'run loadm4image_ddr; bootaux ${m4runaddr}'
```

6. Save the changes.

```
=> saveenv
```

7. Boot the M4 application.

```
=> run m4boot
```

5 Remote communication applications (RPMsg)

5.1 Ping-pong application

The RPMsg ping-pong application is an example of communication between the Cortex-A9 core and the Cortex-M4 core using the RPMsg API.

1. Make sure the `m4_rpmsg_ping_qspi.bin` file is available on eMMC as described in section 4.1 above.
2. Follow the instruction in section 4.3 for how to run an application from QSPI memory, but use the file name **`m4_rpmsg_ping_qspi.bin`** instead of `m4_hello_qspi.bin`.
3. In the u-boot console add the boot argument `uart_from_osc` to `extra_bootargs` to make Cortex-A9 and Cortex-M4 UART clocks match.

```
=> setenv extra_bootargs uart_from_osc
=> saveenv
```

4. Boot the M4 application

```
=> run m4boot
```

5. In the console for the Cortex-M4 you will now see the output below

```
RPMSG PingPong FreeRTOS RTOS API Demo...
RPMSG Init as Remote
```

6. In the console for Cortex-A9 boot into Linux

```
=> boot
```

7. When Linux has booted you need to load the rpmsg pingpong module.

```
# modprobe imx_rpmsg_pingpong
```

8. You will now see messages in both consoles / terminals.

6 FreeRTOS

NXP has developed a number of sample applications and peripheral drivers for the Cortex-M4 bundled together with the real-time operating system **FreeRTOS**.

6.1 Installation

The bundle can be downloaded from NXP's website and the version used when writing these instructions is **v1.0.1**. Follow the link below to download the bundle.

https://www.nxp.com/webapp/Download?colCode=FreeRTOS_MX6SX_1.0.1_WIN

> **NOTE**: You need to register an account at nxp.com in order to get access to the FreeRTOS installation package.

6.1.1 File Structure

When FreeRTOS has been installed you will have a file structure as shown in Figure 4.

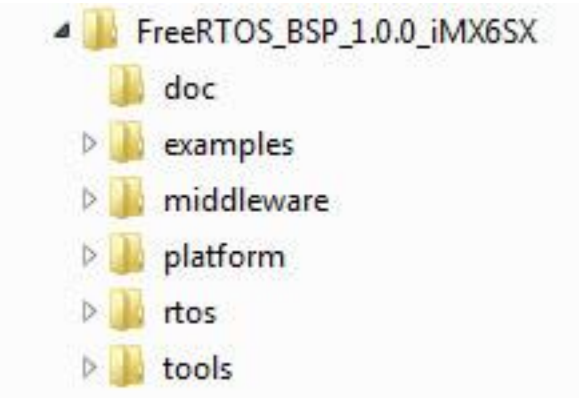

Figure 4 - FreeRTOS file structure

6.2 Board Support Package (BSP)

The board support package (BSP) that is available in the FreeRTOS package is for the Freescale/NXP i.MX6 SoloX Sabre board. Embedded Artists has at the time of writing this document not developed a BSP for the i.MX6 SoloX COM board / Developer's Kit. This means that changes (most often only pin muxing) might be necessary before building and running any of the examples.

BSP files are located in the directory <FreeRTOS>\examples\imx6sx_sdb_m4\ where <FreeRTOS> is the installation path to FreeRTOS.

6.2.1 UART

The pin muxing for UART2 must be changed in order for console output (printf) to be available on connector J15. For the Sabre board the GPIO1_IO06 and GPIO1_IO07 pins are used by UART2, but on the iMX6 SoloX Developer's Kit SD1_DATA0 and SD1_DATA1 must be used.

1. Open file <FreeRTOS>\examples\imx6sx_sdb_m4\pin_mux.c
2. Go to function configure_uart_pins
3. Go to the case statement and change the code as below (pin muxing is changed to use SD1_DATA0 and SD1_DATA1).

```
case UART2_BASE:
```

```
IOMUXC_SW_MUX_CTL_PAD_SD1_DATA0 = IOMUXC_SW_MUX_CTL_PAD_SD1_DATA0_MUX_MODE(4);
IOMUXC_SW_MUX_CTL_PAD_SD1_DATA1 = IOMUXC_SW_MUX_CTL_PAD_SD1_DATA1_MUX_MODE(4);
IOMUXC_SW_PAD_CTL_PAD_SD1_DATA0 = IOMUXC_SW_PAD_CTL_PAD_SD1_DATA0_PKE_MASK | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA0_PUE_MASK | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA0_PUS(2)   | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA0_SPEED(2) | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA0_DSE(6)   | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA0_SRE_MASK | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA0_HYS_MASK;
IOMUXC_SW_PAD_CTL_PAD_SD1_DATA1 = IOMUXC_SW_PAD_CTL_PAD_SD1_DATA1_PKE_MASK | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA1_PUE_MASK | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA1_PUS(2)   | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA1_SPEED(2) | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA1_DSE(6)   | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA1_SRE_MASK | \
                                  IOMUXC_SW_PAD_CTL_PAD_SD1_DATA1_HYS_MASK;

IOMUXC_UART2_IPP_UART_RXD_MUX_SELECT_INPUT =
        IOMUXC_UART2_IPP_UART_RXD_MUX_SELECT_INPUT_DAISY(2);
```

6.3 Build with ARM DS-5

This section describes how to setup ARM DS-5 to build the sample applications. The instructions are originally from the document found at the location below (`<FreeRTOS>` is the path to where the FreeRTOS bundle was installed).

```
<FreeRTOS>\doc\
Getting_Started_with_FreeRTOS_BSP_for_i.MX_6SoloX.pdf.
```

> **NOTE**: You need a commercial license in order to run ARM DS-5 and you must also have installed ARM DS-5 before following the instructions.

1. Start ARM DS-5
2. Import an application
   a. Go to File → Import → General → “Existing Projects into Workspace” and click the “Next” button as shown in Figure 5.

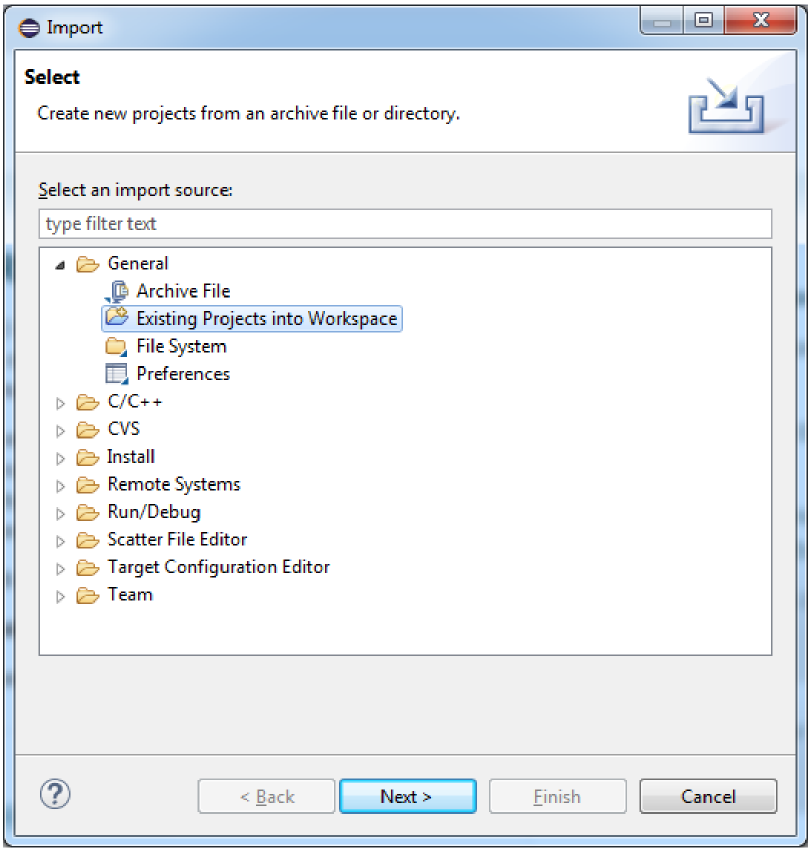

Figure 5 - Import Existing Projects

b. Browse to the DS-5 project files for the application to import. In this example it is the OCRAM version of “hello world” found at:
`<FreeRTOS>\examples\imx6sx_sdb_m4\demo_apps\hello_world\ds5`

c. Click the Finish button

3. Choose build target by clicking on the arrow to the right of the “hammer” in to toolbar, see Figure 6. When the target has been selected the project will be built. If target has previously been selected it is enough to click on the “hammer” icon.

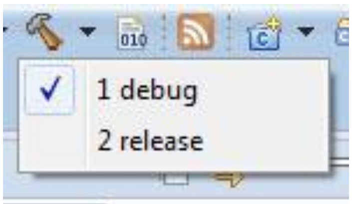

Figure 6 - Build targets

4. The built application is now available at the location below. There will be both an axf file and a bin file. It is the bin file that should be loaded to the iMX6 COM SoloX Board as described in chapter 4

```
<FreeRTOS>\examples\imx6sx_sdb_m4\demo_apps\hello_world\ds5\debug
```

6.3.1 BSP files

Section 6.2 described changes that must be made to BSP files. When a project has been imported to DS-5 it is possible to edit these files in DS-5 instead of an external editor. The files are found in the “board” folder in the project, see Figure 7.

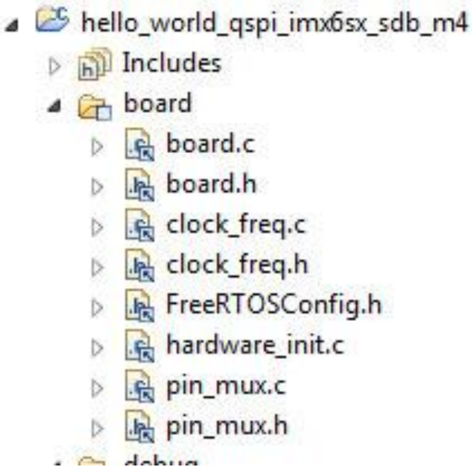

Figure 7 - DS-5 board folder

6.4 Debug using DS-5

With ARM DS-5, a Keil ULINK Pro, and a debug interface board it is possible to download and debug an application on the Cortex-M4.

6.4.1 Setup the hardware

Figure 8 and Figure 9 show how the ULINK Pro and debug interface board is connected to the iMX6 SoloX COM Board.

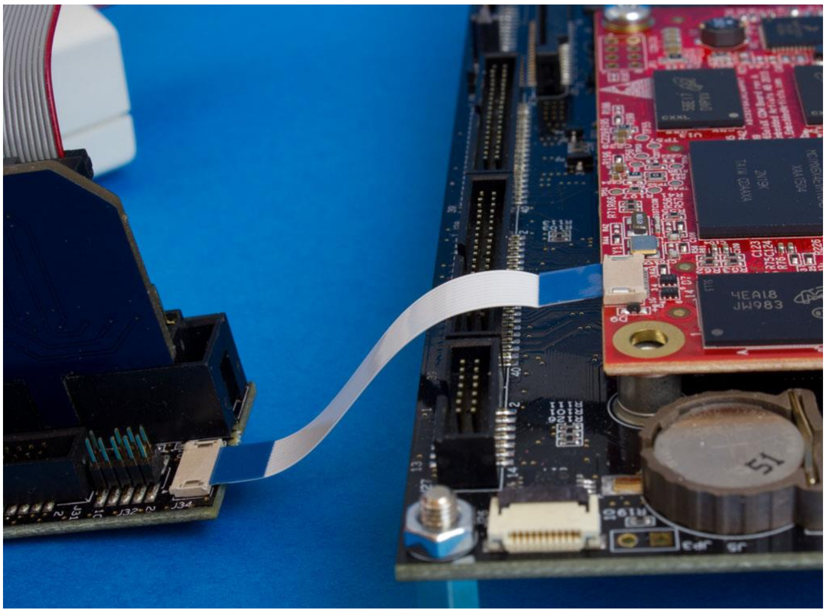

Figure 8 - Debug interface board connected to COM board

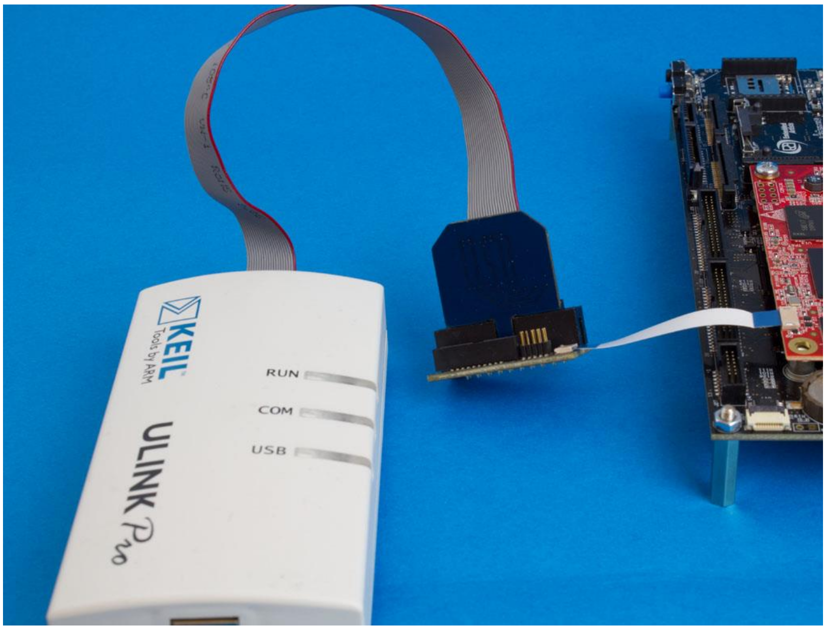

Figure 9 – ULINK Pro and debug interface board

6.4.2 Import TCM version of "hello world"

In this example we are going to debug the same application as was built in section 6.3 which is the TCM version of Hello World.

6.4.3 Create a new Debug configuration

To be able to download and debug a "Debug configuration" must be created.

1. Go to Run → Debug Configurations and select DS-5 Debugger as shown in Figure 10.

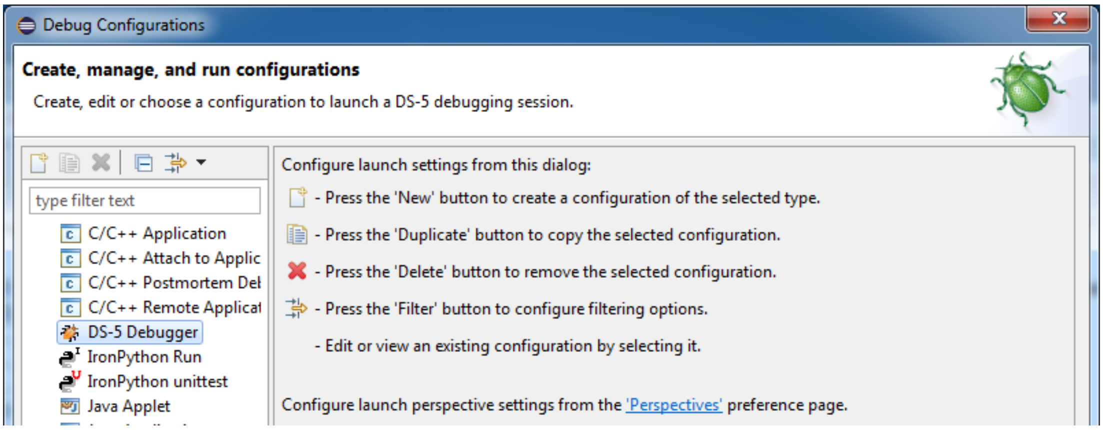

Figure 10 - Debug Configurations

2. Right click on DS-5 Debugger and select "New".
3. Give the configuration a name such as SoloX Cortex-M4 and then select the "Connection" tab as shown in Figure 11.

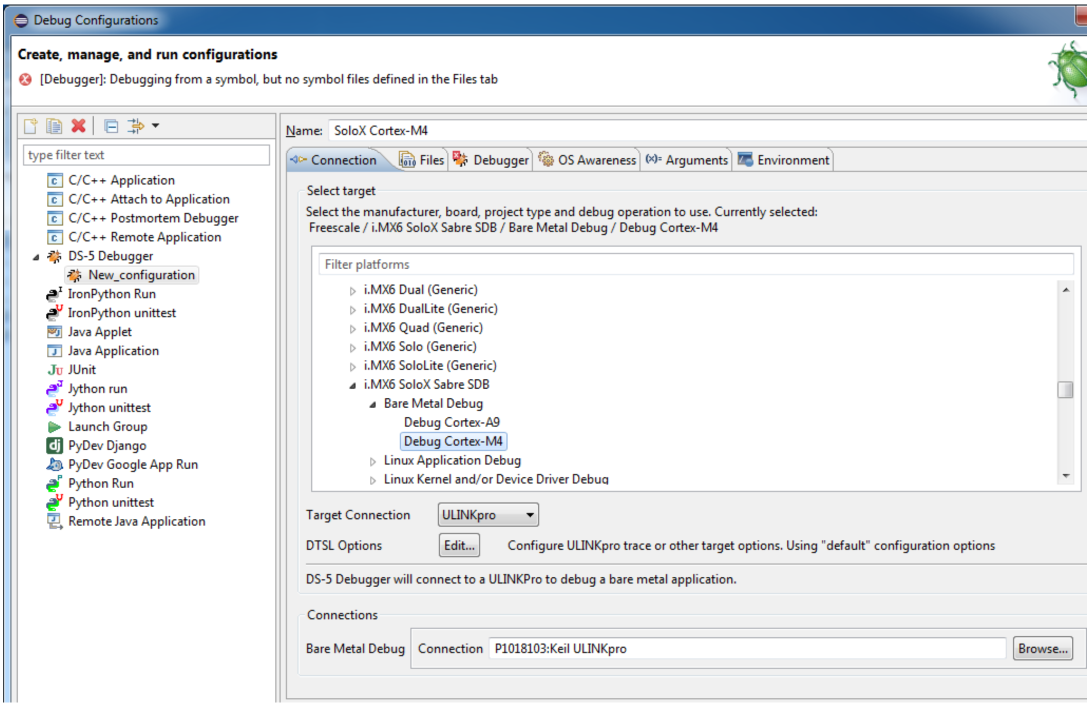

Figure 11 - Setup Debug Connection

4. In the “Connection” tab go to NXP → i.MX6 SoloX Sabre SDB → Bare Metal debug and choose “Debug Cortex-M4” as shown in Figure 11.
5. Still in the “Connection” tab select ULINKpro in the “Target Connection” list and then click the “Browse” button in the Connections section. Select the ULINKpro connection.

   Please note that the ULINK pro debug adapter must be connected to your computer before clicking the “Browse” button
6. Click on the “Files” tab and then the “Workspace” button. Select the `axf` file in the “debug” folder as shown in Figure 12.

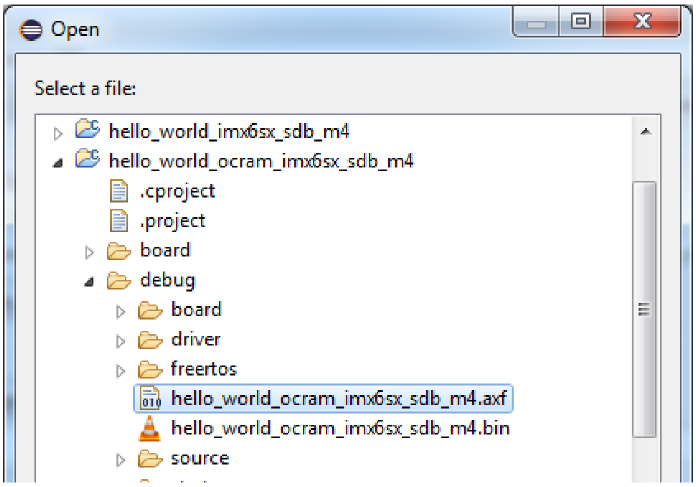

Figure 12 - Application to download

7. Go to the “Debugger” tab and select “Debug from entry point” as shown Figure 13.

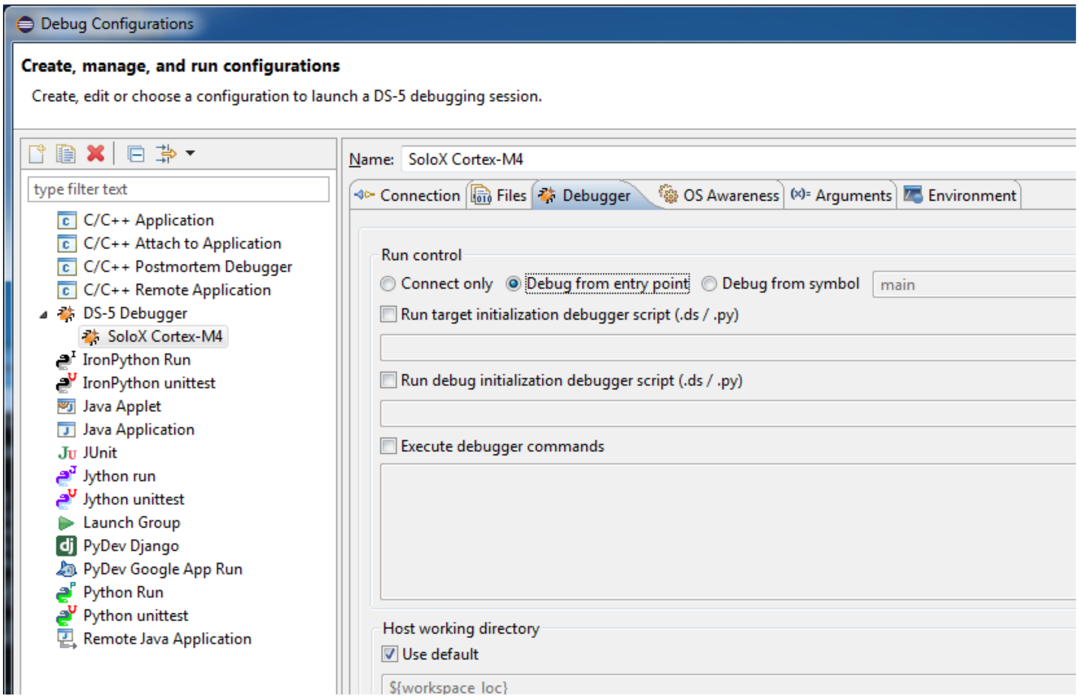

Figure 13 - Debug from entry point

8. Go to the “OS Awareness” tab and choose FreeRTOS in the list as shown in Figure 14.

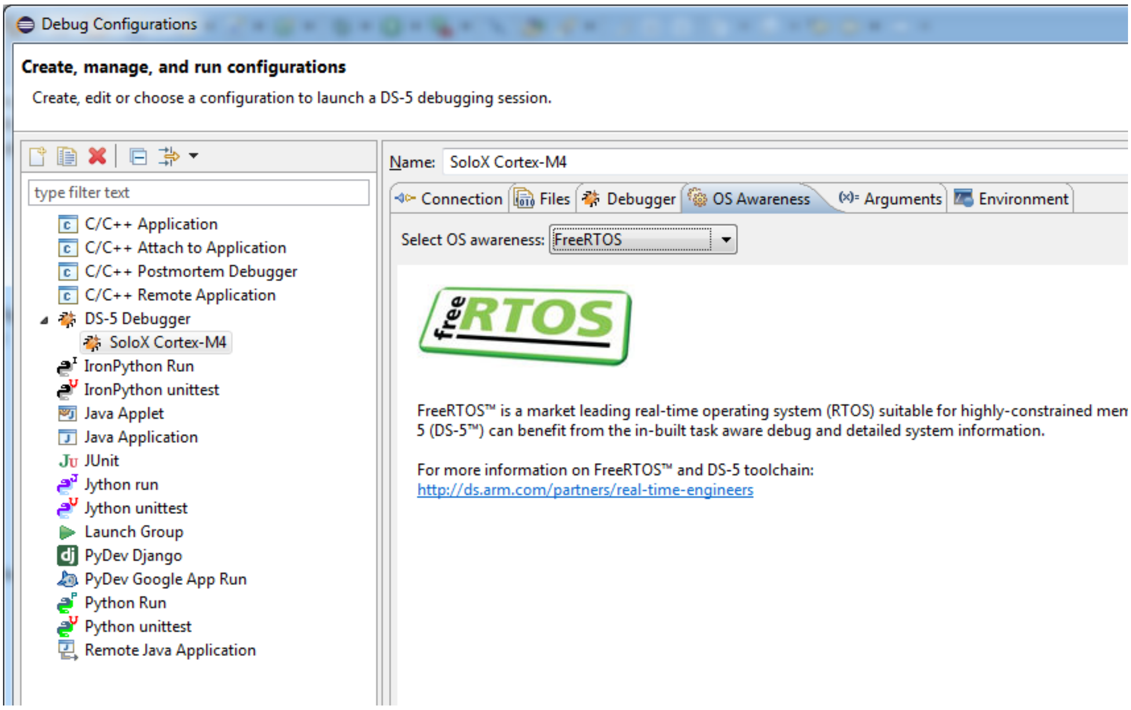

Figure 14 - OS Awareness

9. Click the “Apply” button and then the “Debug” button to initiate a debug session. When the application has been downloaded to the target it could look like Figure 15.

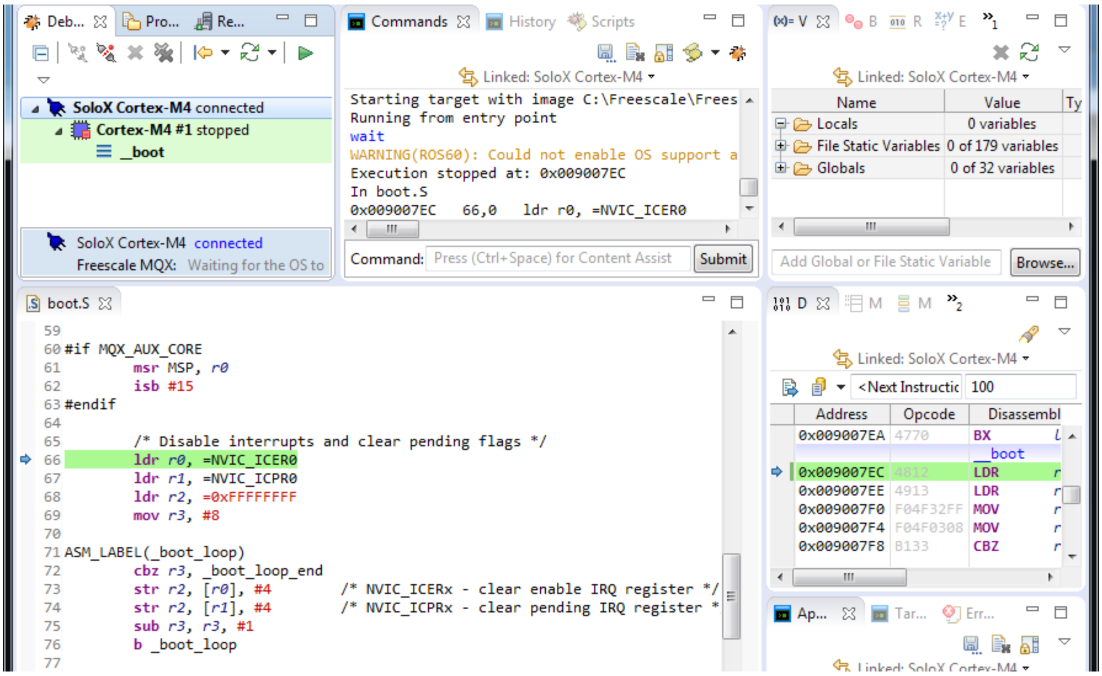

Figure 15 - Active debug session

NOTE 1: If you are not able to start the debug session please make sure that you have only booted into **u-boot** on the Cortex-A9 and not into Linux when you start the debug session.

NOTE 2: If the terminal/console attached to the A9-core (Linux) seem to be unresponsive, that is, it doesn't accept any input please read section 8.3 .

6.5 Build with ARM GCC

6.5.1 Install ARM GCC

Download and install GCC ARM Embedded. The file `gcc-arm-none-eabi-4_8-2014q1-20140314-win32.exe` was used when writing these instructions.

https://launchpad.net/gcc-arm-embedded/+download

6.5.2 Install MinGW

MinGW – native Windows port of the GNU Compiler Collection (GCC) is also needed to build the applications on a Windows machine.

1. Go to the link below and click the "Download" button
   http://sourceforge.net/projects/mingw/
2. Start the downloaded installation file and click the Install button and then click the "Continue" button on the dialog windows that will appear.

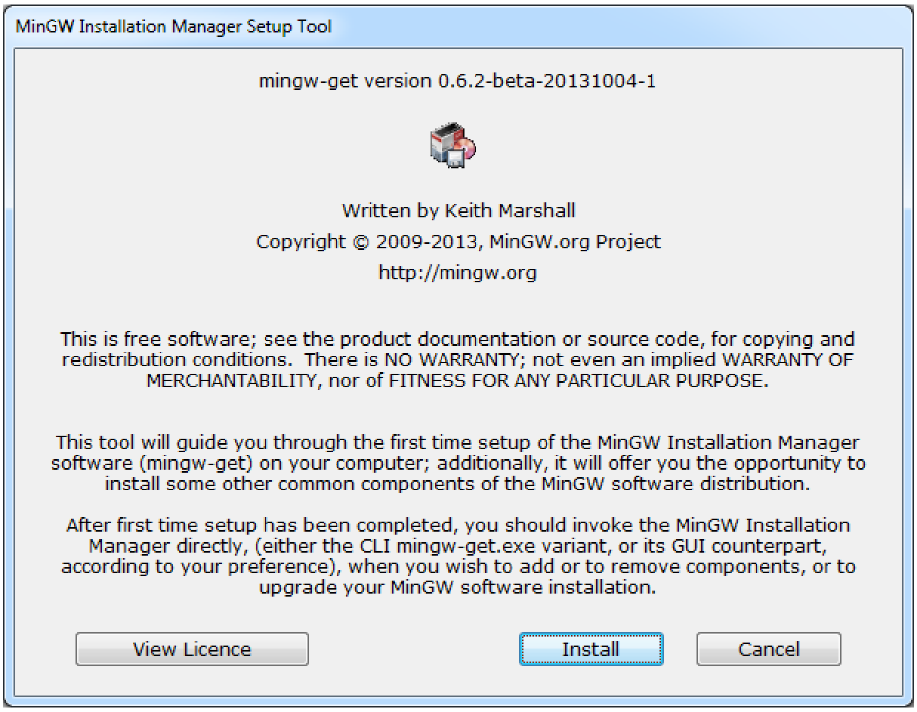

Figure 16 - MinGW Installation

3. When the installation manager window appears, as shown in Figure 17, choose `mingw32-base` and `msys-base` in the "Basic Setup" section.

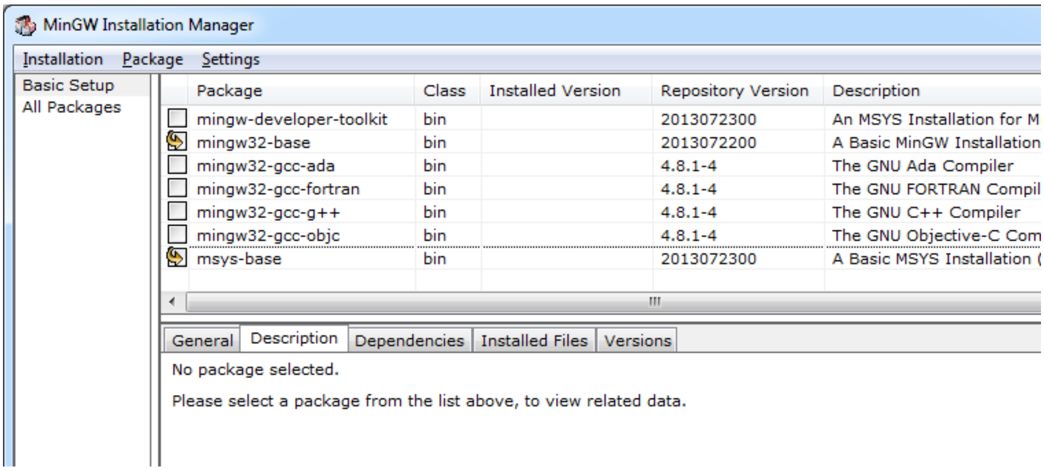

Figure 17 - MinGW Installation Manager

4. Click Installation →Apply Changes for the packages to be installed.

5. When the installation has finished add `C:\MinGW\bin` (if this is where you installed MinGW) to the PATH variable. There are several ways to add something to the PATH variable.

    a. In a command prompt write `set PATH=%PATH%;C:\MinGW\bin`

    b. To permanently add MinGW to PATH open System properties by (this applies for Windows 7) right clicking on Computer in an Explorer window and then select Properties. Click "Change settings" and then the Advanced tab as shown in Figure 18. Click on the "Environment Variables" button and edit the PATH variable as shown in Figure 19.

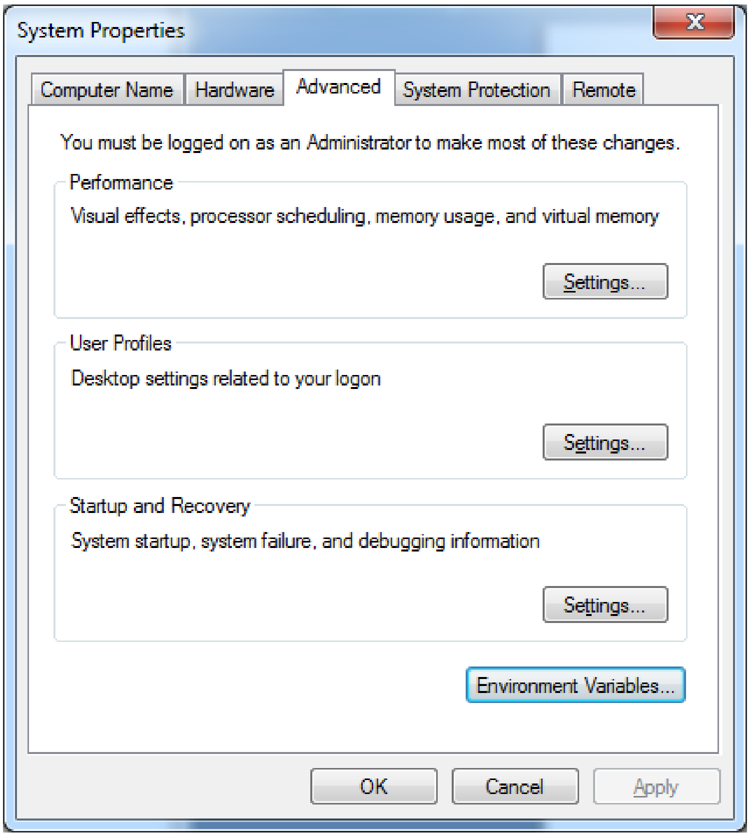

Figure 18 - System Properties in Windows

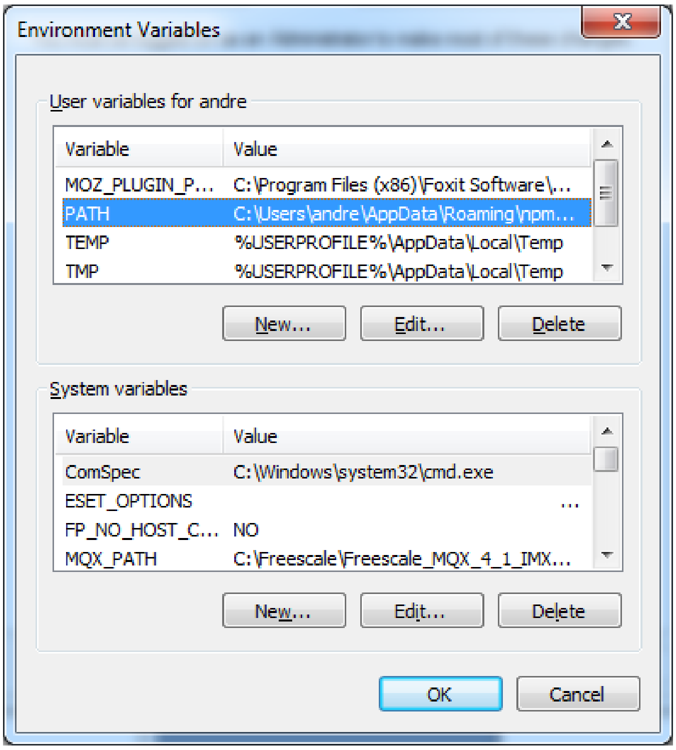

Figure 19 - Environment Variables in Windows

6. Create the `ARMGCC_DIR` environment variable

   a. Click the “New” button below “System variables” as seen in Figure 19.

b. Add `ARMGCC_DIR` as variable name and specify the path to ARM GCC as value. The default installation path of ARM GCC which has been installed when following these instructions is:

```
C:\Program Files (x86)\GNU Tools ARM Embedded\4.8 2014q1
```

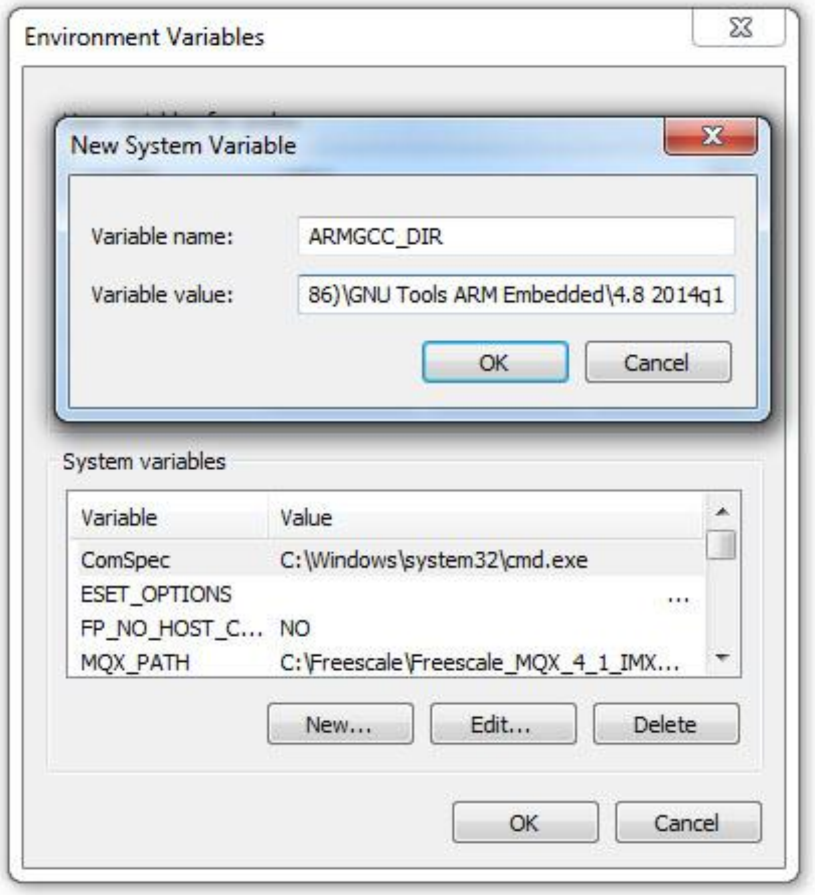

Figure 20 - ARMGCC_DIR variable

7. Click Ok and then Ok again.

6.5.3 Install CMake

Download and install CMake from the link below. Make sure to add CMake to the system path as shown in Figure 21.

http://www.cmake.org/cmake/resources/software.html

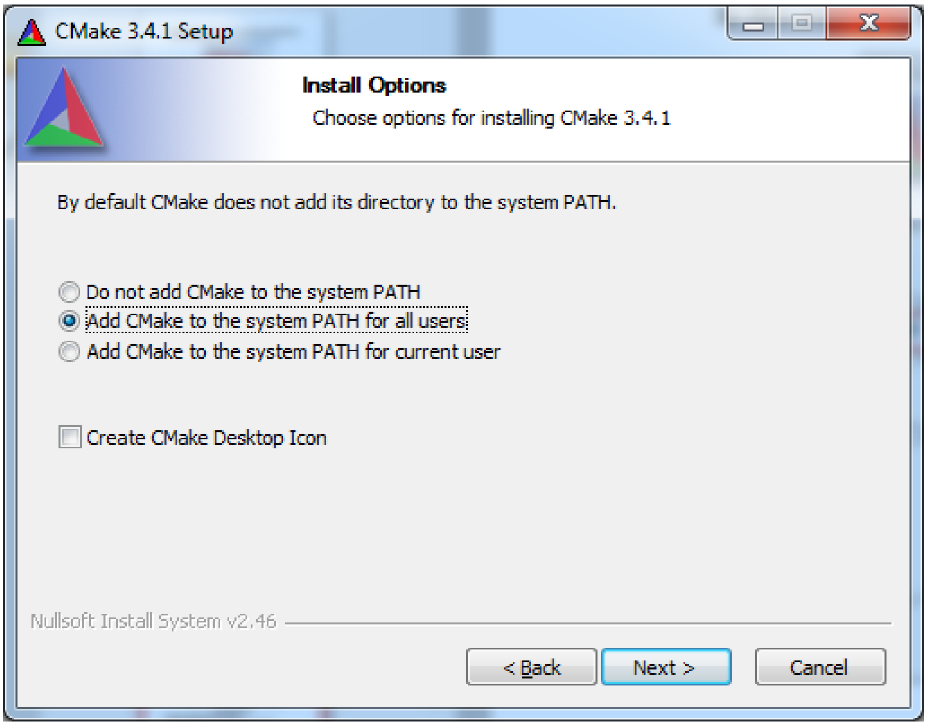

Figure 21 - CMake Install Options

6.5.4 Build Application

1. Open a GCC Command prompt. When ARM GCC was installed a shortcut was created in the start menu as shown in Figure 22.

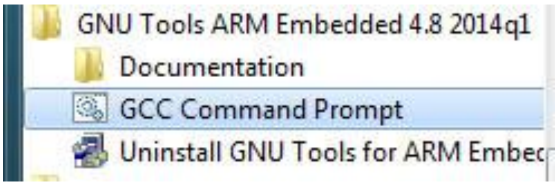

Figure 22 - GCC Command Prompt shortcut

2. Change directory to the application that should be built. In this example the `hello_world_qspi` application is built.

```
cd <FreeRTOS>\examples\imx6sx_sdb_m4\demo_apps\hello_world_qspi\armgcc
```

3. Run `build_debug.bat` to build the application
4. The output of the build will be both an elf file and a bin file located in the sub-directory `debug`. Use the instructions in chapter 4 to download the application to the iMX6 SoloX COM board.

6.6 Build with Eclipse and ARM GCC

How to install and use ARM GCC from the command line is described in section 6.5 above. Most often you however need to use a development environment (editor) when developing an application. This section will describe how you can setup Eclipse to use ARM GCC when developing the application.

NOTE: You must have followed the instructions in section 6.5 before continuing with the instructions in this section.

It is assumed that you have installed Eclipse with the CDT (C/C++ Development Tooling) plugin. Eclipse version 4.4.2 (Luna) and CDT 8.6.0 where used when writing these instructions.

6.6.1 Install "GNU ARM Eclipse" plugins

We will utilize CDT extensions called "GNU Arm Eclipse". Follow the instructions on the link below to install these extensions/plugins.

http://gnuarmeclipse.github.io/plugins/install/

6.6.2 Create project: New

Start by creating a new "C Project". Go to File → New Project and then select "C Project" under the "C/C++" group as shown in Figure 23.

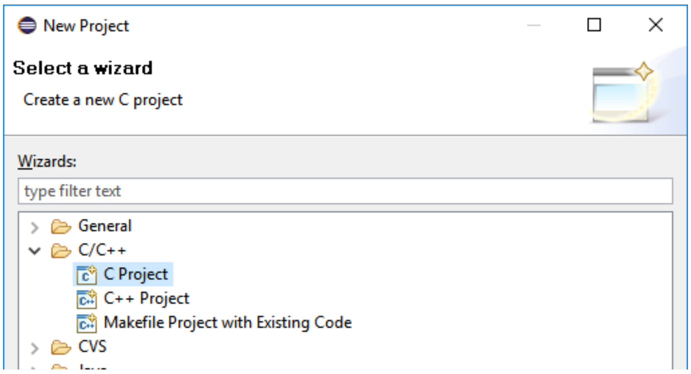

Figure 23 - Select project wizard

Click "Next", select "Empty Project", "Cross ARM GCC" as toolchain and give the project a name as shown in Figure 24.

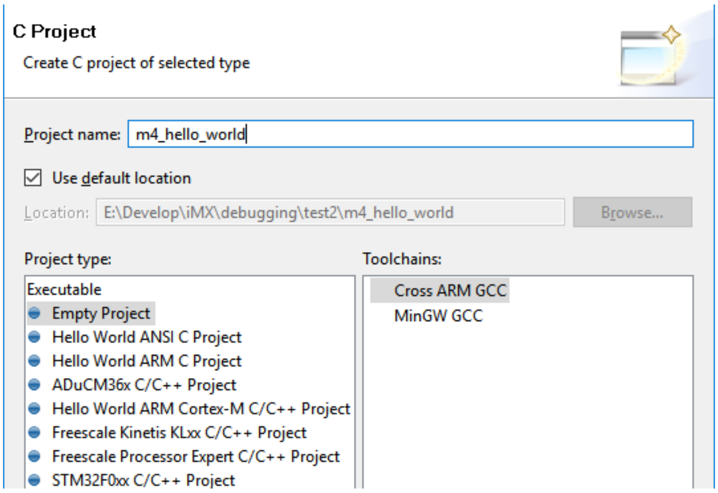

Figure 24 - Project type and toolchain

Click "Next" and then "Next" again. The toolchain and path should be selected. If "GNU Tools" hasn't been selected by default change to this as shown in Figure 25.

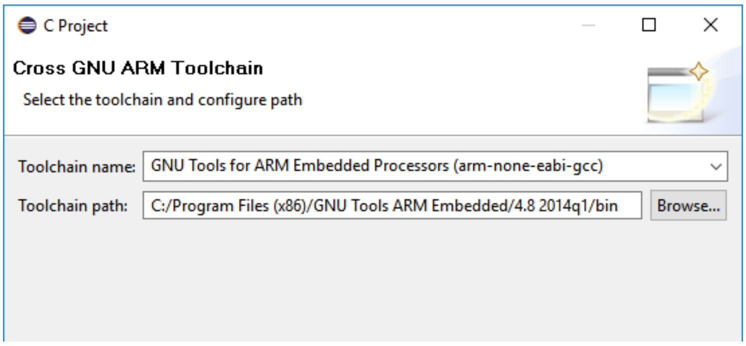

Figure 25 - GNU ARM Toolchain

6.6.3 Create project: Linked folders

Section 6.1.1 shows the file structure of the installed FreeRTOS bundle for iMX6. The source code that we need to build is located in several different folders and we need to add these to the Eclipse project. There are several ways to do this, but in this example we will use "linked folders" and keep the structure created when installing the bundle.

Begin by adding a linked folder to the demo application. In this example we will be using the "hello_world" demo. Click on the "Add Folder" icon in the toolbar as shown in Figure 26. Then select "Folder". An alternative way is to do this from the menu: File → New → Folder.

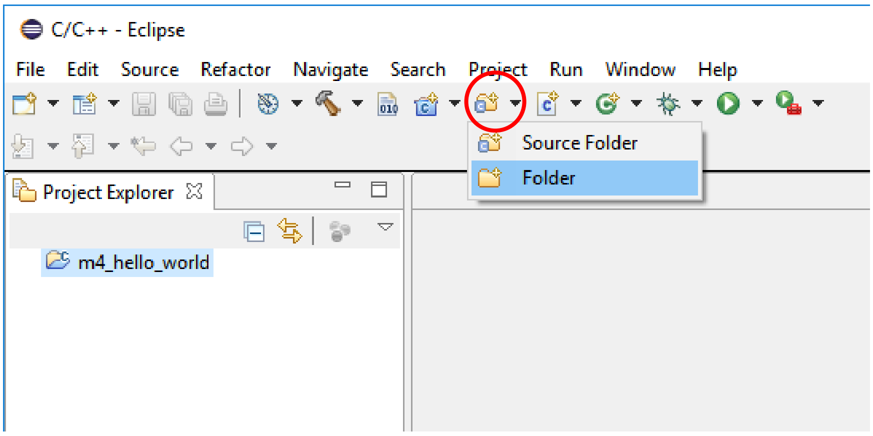

Figure 26 - Add folder

In the dialog window click on the "Advanced" button and then to "Link to alternate location" and browse to the `<FreeRTOS path>/examples/imx6sx_sdb_m4/demo_apps/hello_world` folder. This is shown in Figure 27.

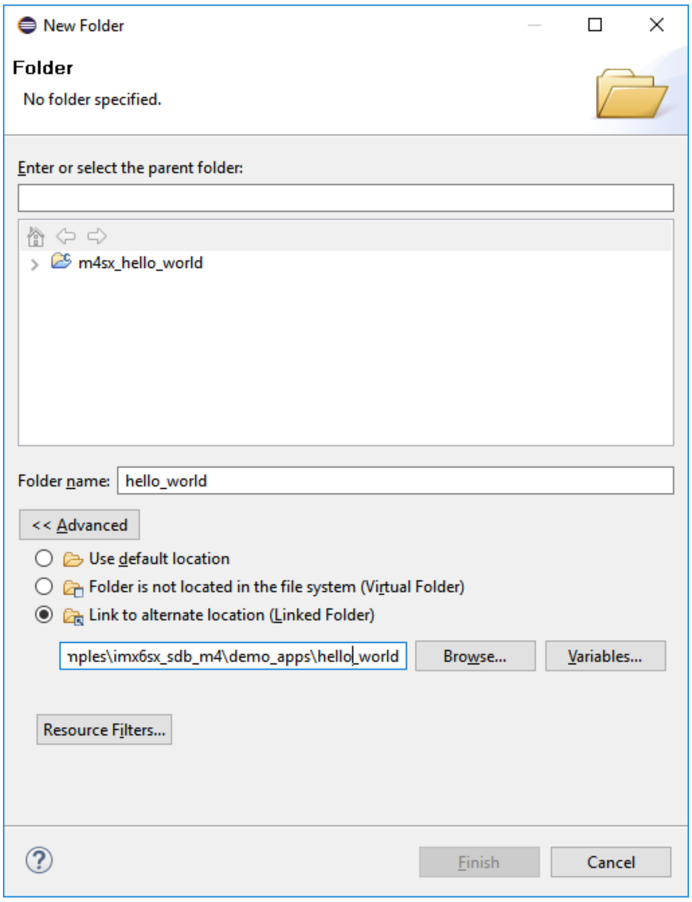

Figure 27 - Linked folder

Repeat the above steps for the following folders:

- `<FreeRTOS path>/examples/imx6sx_sdb_m4`
  - This folder contains board specific code
- `<FreeRTOS path>/platform`
  - Contains initialization and driver code for the iMX7 processor
- `<FreeRTOS path>/rtos/FreeRTOS`
  - The FreeRTOS code

When all folders have been added to the project it will look like in Figure 28.

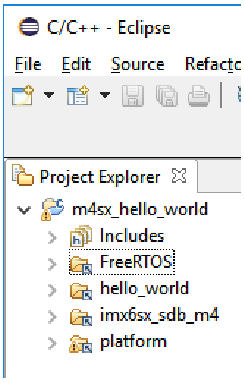

Figure 28 - File structure in Eclipse

6.6.4 Create project: Exclude from build

Some of the sub-folders added to project as described in section 6.6.3 shouldn't be part of the build. These can be excluded by right-clicking on the folder and then selecting "Resource Configurations" → "Exclude from Build". This is shown in Figure 29

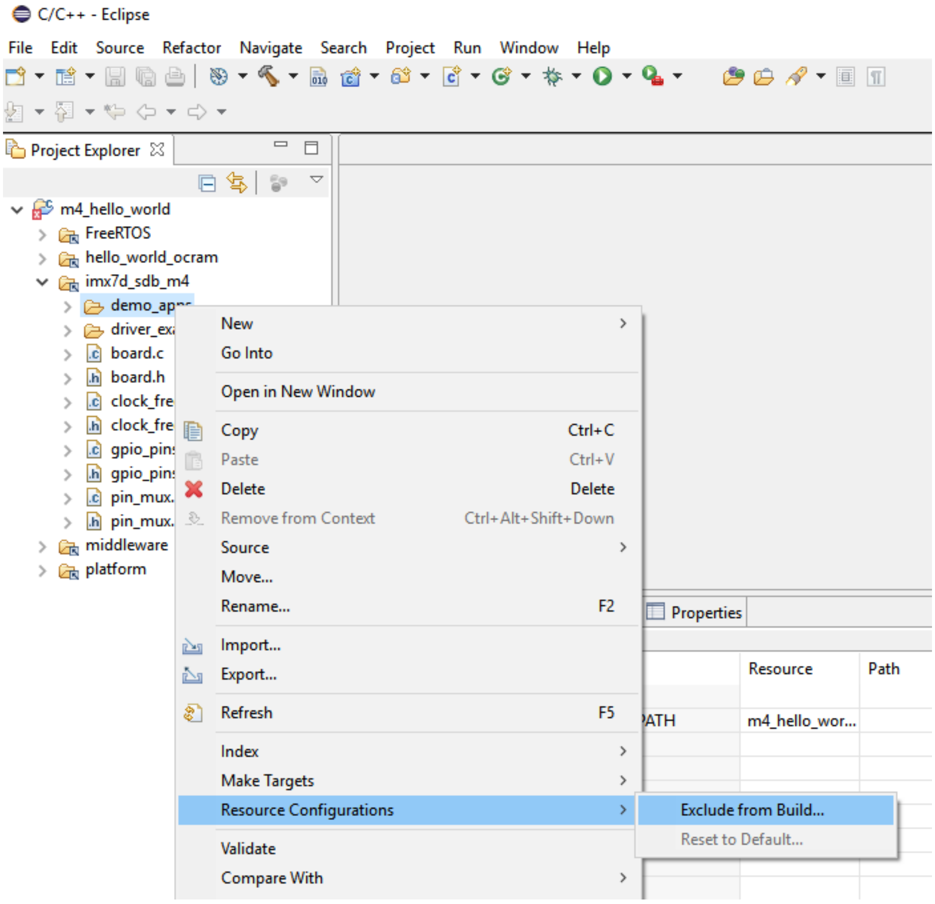

Figure 29 - Exclude folder from build

We must also specify which configurations to exclude the folders from. In our case we select both "Debug" and "Release" as shown in Figure 30.

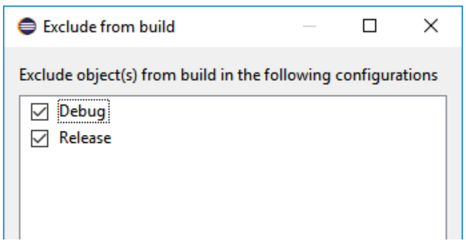

Figure 30 - Configurations to exclude from

Exclude all of the following files and folders:

- `imx6sx_sdb_m4/demo_apps`
  - The demo_apps folder contains several applications. We only want to build hello_world.
- `imx6sx_sdb_m4/driver_examples`
  - The driver_examples folder contains several applications. We only want to build hello_world.

- `FreeRTOS/Source/portable/IAR`
  - This folder contains code specific for the IAR compiler
- `FreeRTOS/Source/portable/RVDS`
  - This folder contains code specific for the RVDS compiler
- `FreeRTOS/Source/portable/MemMang/heap_2.c (also heap_3.c and heap_4.c)`
  - The MemMang folder contains several implementations of memory allocation routines. We can only use one and will keep the one implemented in heap_1.c. Exclude all other files.
- `platform/CMSIS/DSP_Lib`

6.6.5 Create project: "Include" paths

Header files are located at several different locations in this project structure. These header files must be found during a build. This can be done by right-clicking on the project and then select "Properties". Go to "C/C++ General" → "Paths and Symbols". Select "GNU C" as language and then click the "Add" button as shown in Figure 31.

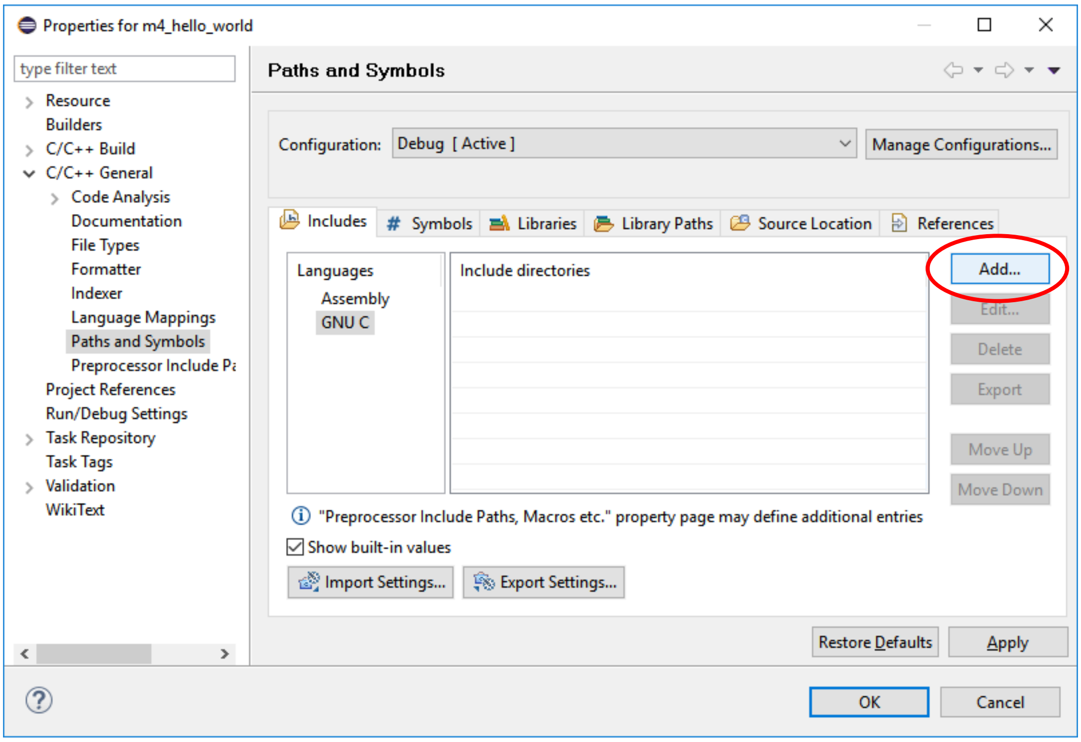

Figure 31 - Include paths

We are going to add the paths as relative to the workspace so click in the "Workspace" button and then browse to the folder to include. In Figure 32 it is shown how the "include" folder for FreeRTOS is selected.

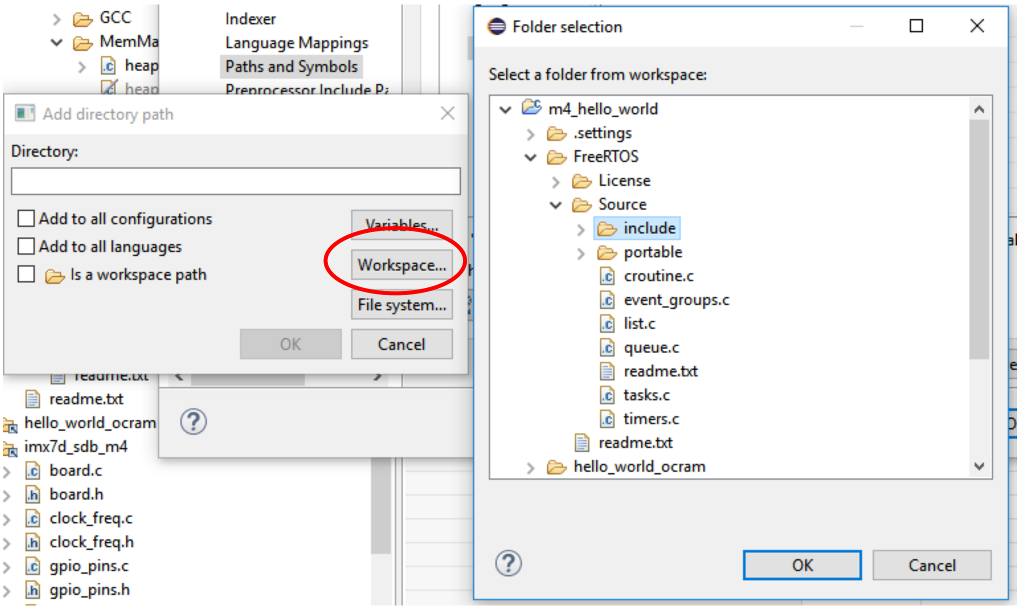

Figure 32 - Workspace folder

Add the following folders as include paths:

- `FreeRTOS/Source/include`
- `FreeRTOS/Source/portable/GCC/ARM_CM4F`
- `hello_world`
- `imx6sx_sdb_m4`
- `platform/CMSIS/Include`
- `platform/devices`
- `platform/devices/MCIMX6X/include`
- `platform/devices/MCIMX6X/startup`
- `platform/devices/drivers/inc`
- `platform/devices/utilities/inc`

6.6.6 Create project: Settings

There are a number of project settings that must be updated. Right click on the project and then select Properties.

By default "make" is used to build the application, but since we have installed mingw make we need to do an update to the toolchain setting. Go to "C/C++ build" → Settings and click on the "Toolchains" tab as shown in Figure 33. Change the value of the "Build command" field from "make" to "mingw32-make".

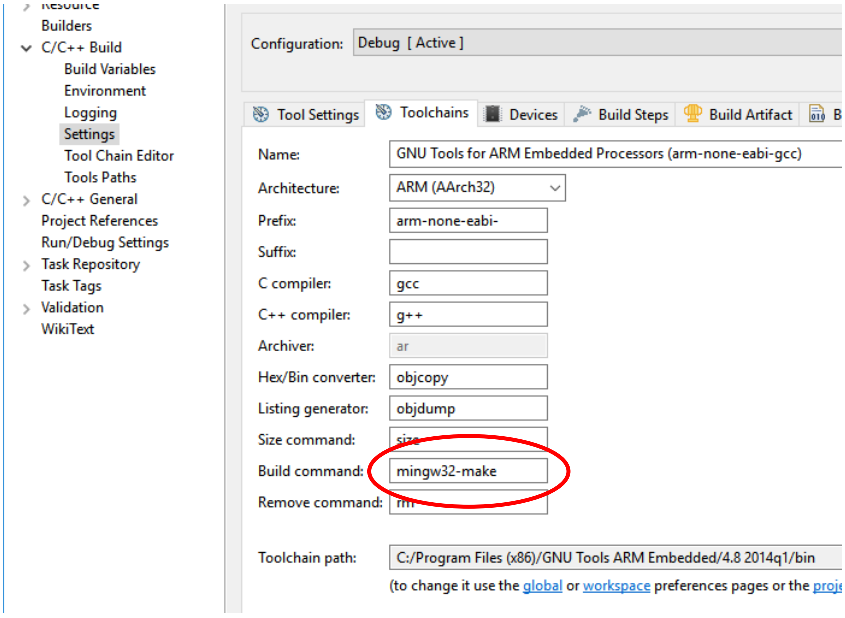

Figure 33 - Build command

Go to the “Tool Settings” tab and click on “Target Processor”. Change the values of the following fields. This is also shown in Figure 34.

- ARM family = cortex-m4
- Float ABI = FB instructions (hard)
- FPU Type = fpv4-sp-d16

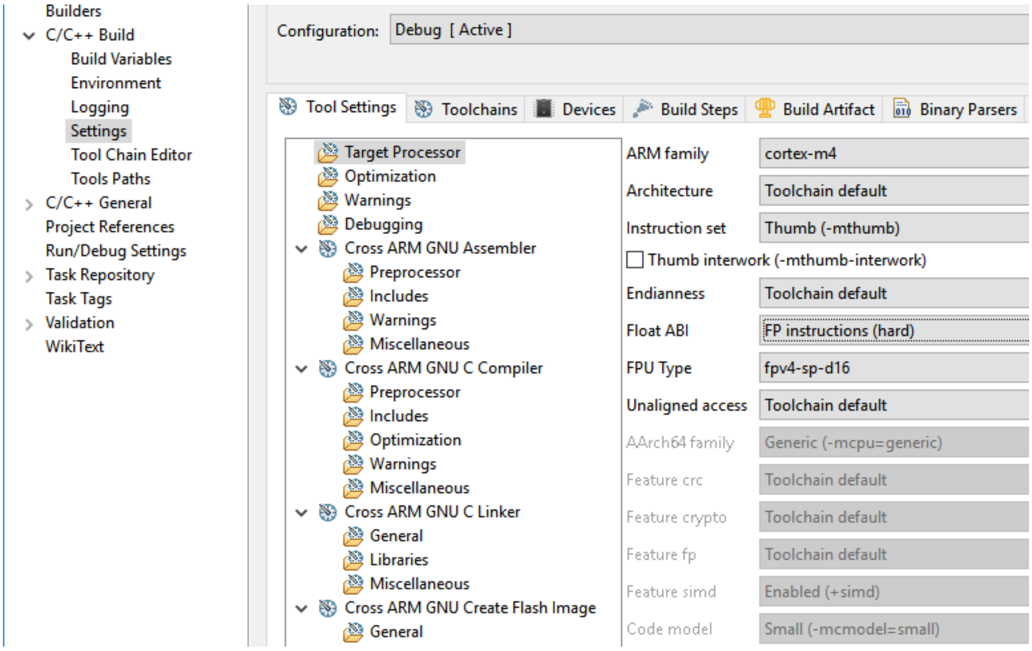

Figure 34 - Target processor

Still in the “Tool Settings” tab go to “Cross ARM GNU C Compiler” → Preprocessor. Add the symbols below:

- `CPU_MCIMX6X_M4`
- `__DEBUG`

- `__FPU_PRESENT`
- `ARM_MATH_CM4`

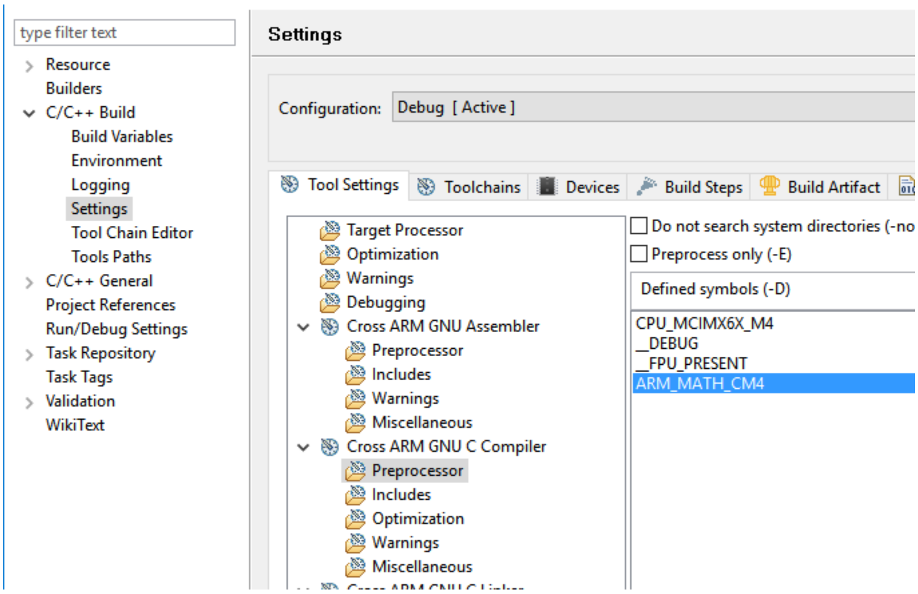

Figure 35 - Preprocessor symbols

Still in the "Tool Settings" tab go to "Cross ARM GNU C Linker" → General. Add the workspace path to the linker file that is going to be used. Since we are building an application for OCRAM we select `platform/devices/MCIMX7D/linker/gcc/MCIMX6X_M4_ocram.ld`.

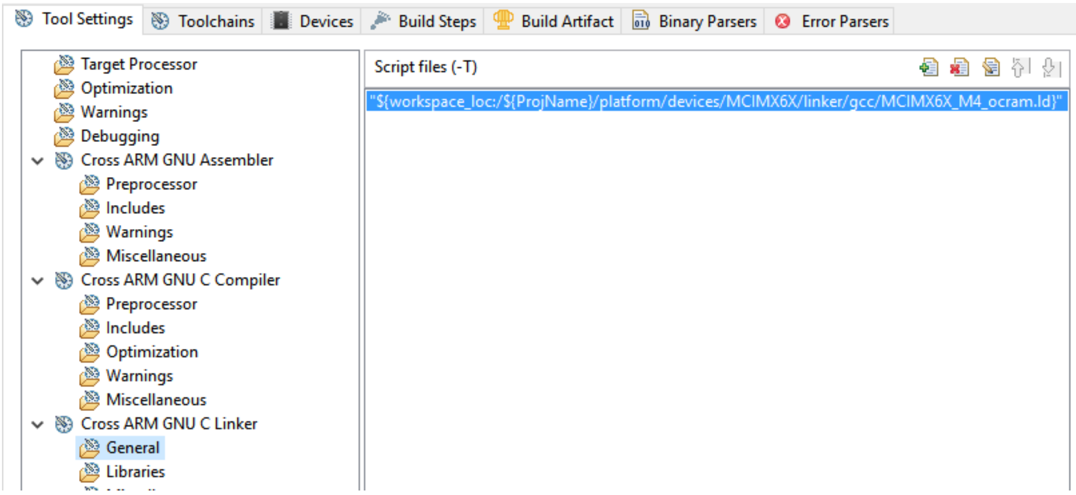

Figure 36 - Linker file

Still in the Linker group select "Miscellaneous". Check the "Use newlib-nano" checkbox and enter "-specs=nosys.specs" in the "Other linker flags" field. These settings are shown in Figure 37.

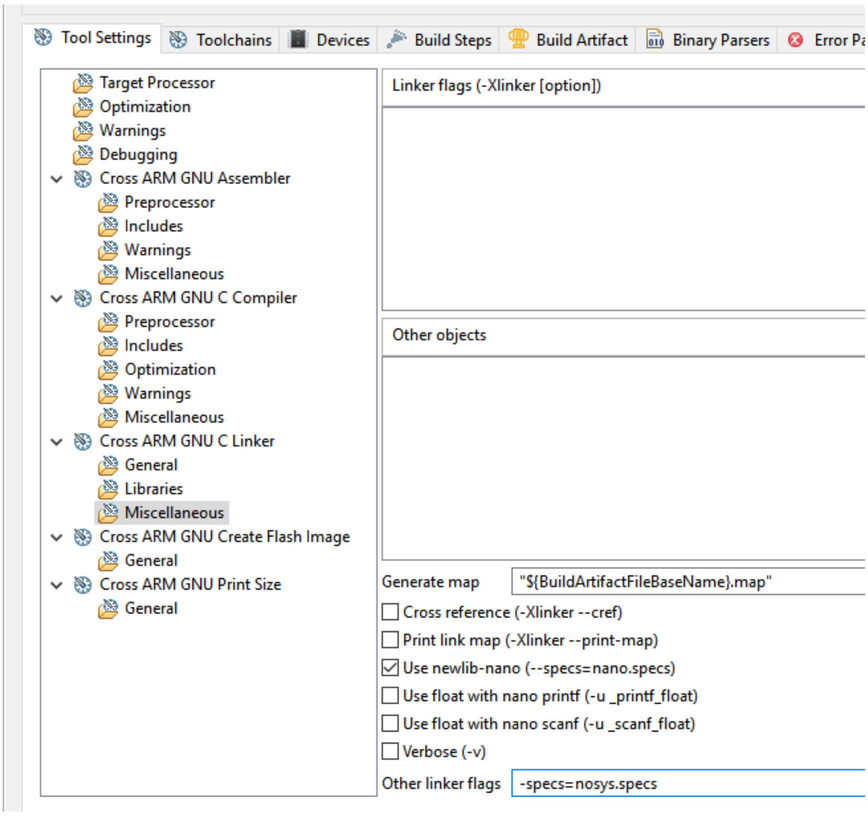

Figure 37 - Misc linker settings

In the "Tool Settings" tab go to "Cross ARM GNU Create Flash Image". Change output format to "Raw binary".

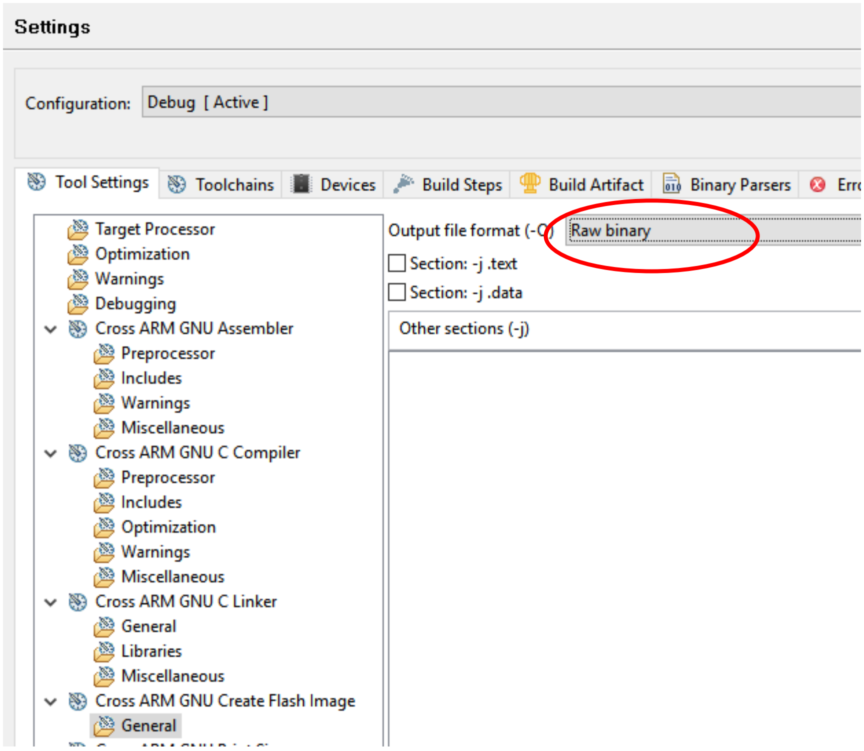

Figure 38 - Create Flash Image

6.6.7 Build application

Now it is time to build the application. This can, for example, be done by clicking on the "Build" icon in the toolbar as shown in Figure 39. It can also be done by right-clicking on the project and then click on "Build Project".

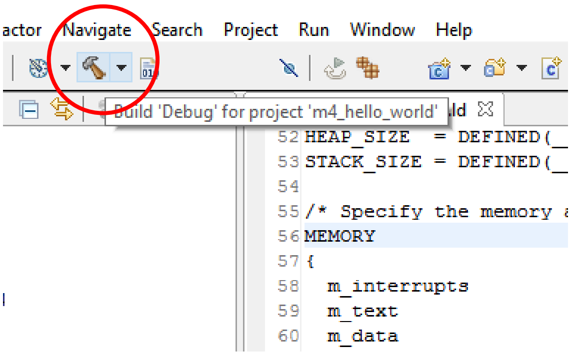

Figure 39 - Build icon

When the application has been built there will be a binary file in the project's "Debug" folder. Use the instructions in section 4.5 to run this application on target. It is also possible to download and debug the application by following the instructions in section 6.7 below.

6.7 Debug using Eclipse

Before following the instructions in this section you must have followed the instructions in section 6.6 and being able to build an application.

6.7.1 LPC-Link 2 with J-Link firmware

We are going to use an LPC-Link 2 with Segger's J-Link firmware as debug adapter. Follow the instructions on the link below to prepare an LPC-Link 2 with the J-Link firmware.

Instructions

https://www.segger.com/lpc-link-2.html

LPC-Link 2

http://www.embeddedartists.com/products/lpcxpresso/lpclink2.php

6.7.2 J-Link GDB Server

Segger's J-Link GDB Server is used when debugging the target. Download and install the "J-Link Software and Documentation Pack". This package contains the GDB server.

https://www.segger.com/downloads/jlink

6.7.3 J-Link script files

A script file is needed when connecting to the M4 core using J-Link. Segger has published script files for both the A9 core and M4 core. You need to download at least the script file for the M4 core.

https://wiki.segger.com/IMX6SX

6.7.4 Connect LPC-Link 2 to the board

Begin by connecting the LPC-Link 2 to the Debug interface board as shown in Figure 40.

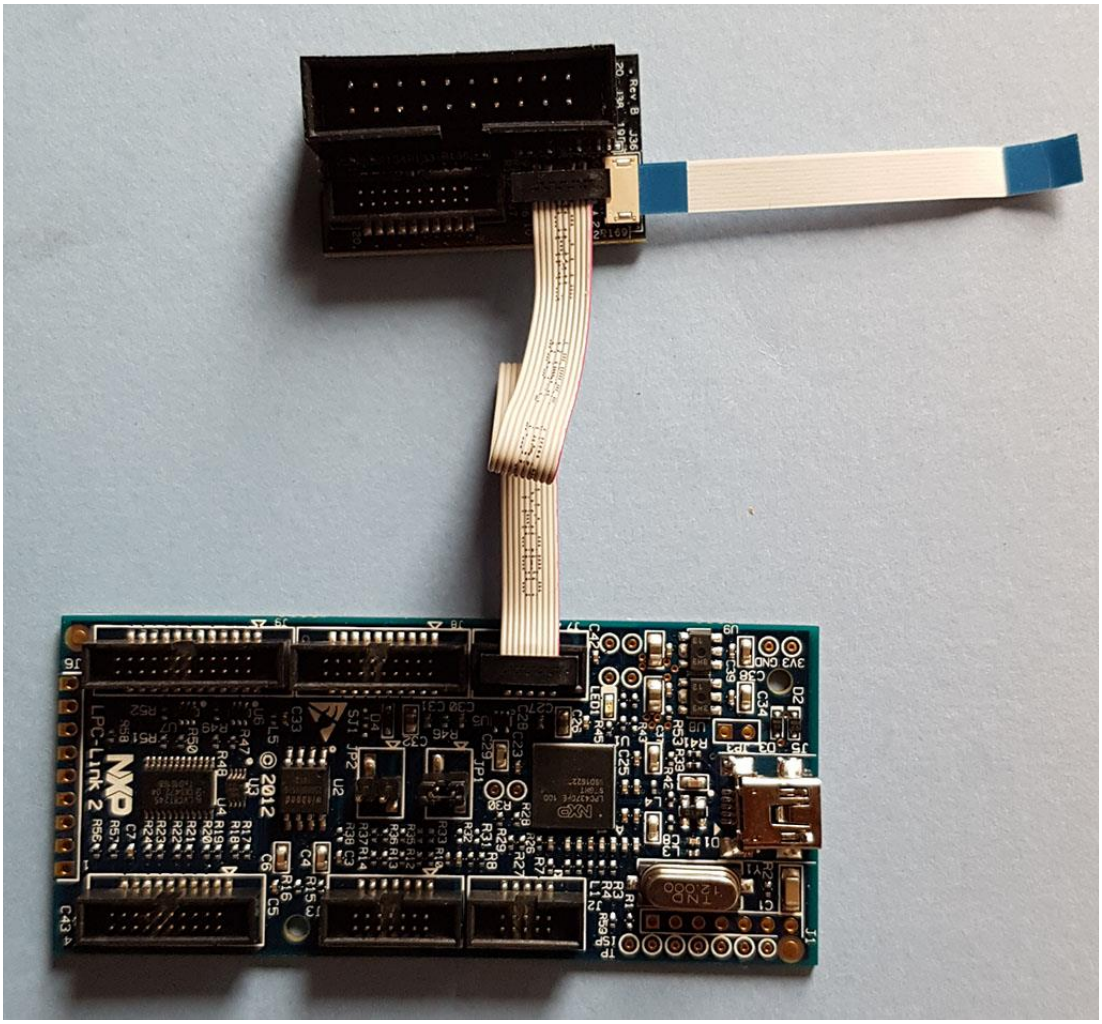

Figure 40 - LPC-Link 2 connected to Debug interface board

Connect the FPC cable for the Debug interface board to the connector on the COM Board as shown in Figure 41.

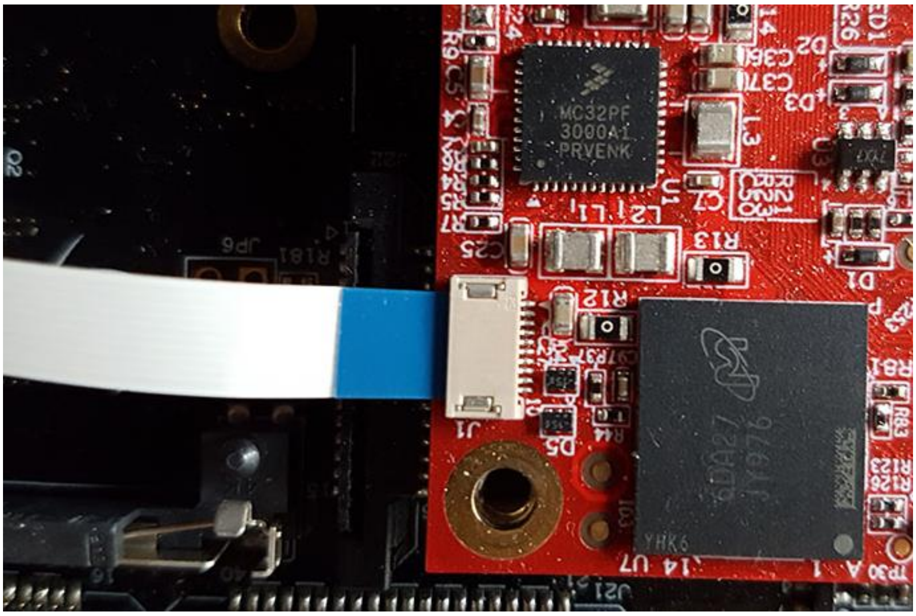

Figure 41 - Debug interface connected to COM board

Also make sure that the LPC-Link 2 board is connected to your PC via a USB cable.

6.7.5 Create a debug configuration

In Eclipse go to Run → Debug Configurations and then select “GDB SEGGER J-Link Debugging”. Create a new “launch configuration” by clicking on the icon shown in Figure 42.

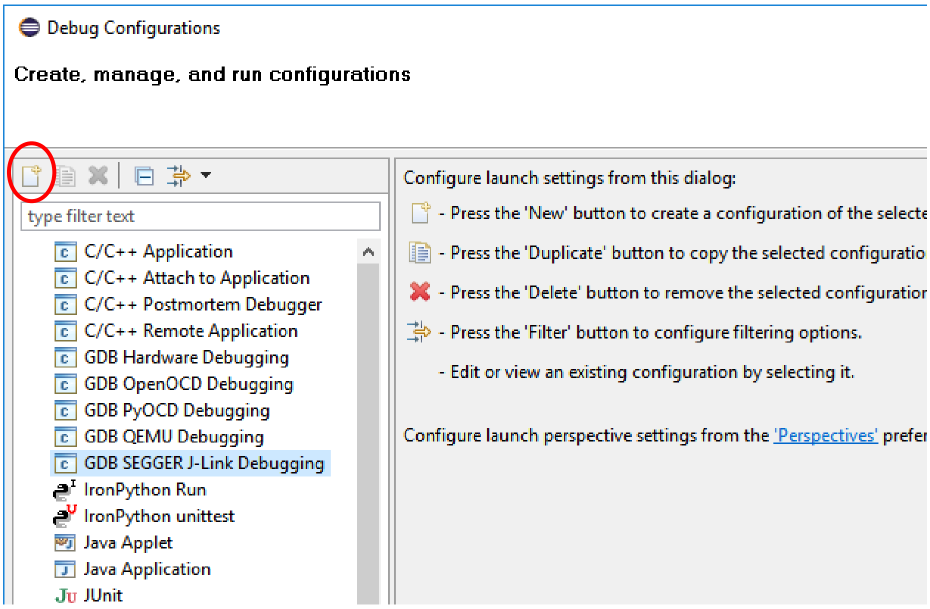

Figure 42 - Debug configuration

Go to the “Debugger” tab. The device name for i.MX 6SoloX is mcimx6s4. We can however not use this name since Segger LPC-Link 2 firmware considers this device to be a Freescale part and not an NXP part. The license for the firmware only allows debugging of NXP parts. The SoloX is now an NXP part, but the firmware hasn’t been updated.

1. Enter “m4” as device name instead of mximx6s4
2. Select “JTAG” as interface
3. In the “Other options” field add `-scriptfile` and the path to the script file downloaded in section 6.7.3 above.

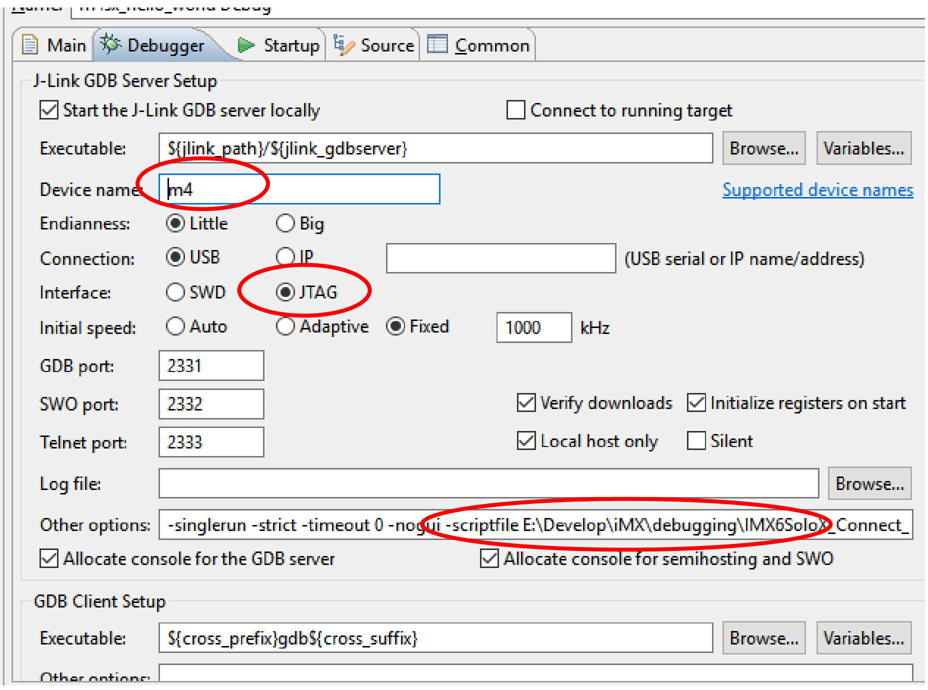

Figure 43 - Debugger tab

Go to the “Startup” tab and then “Runtime Options”. Select “RAM application” as shown in Figure 44.

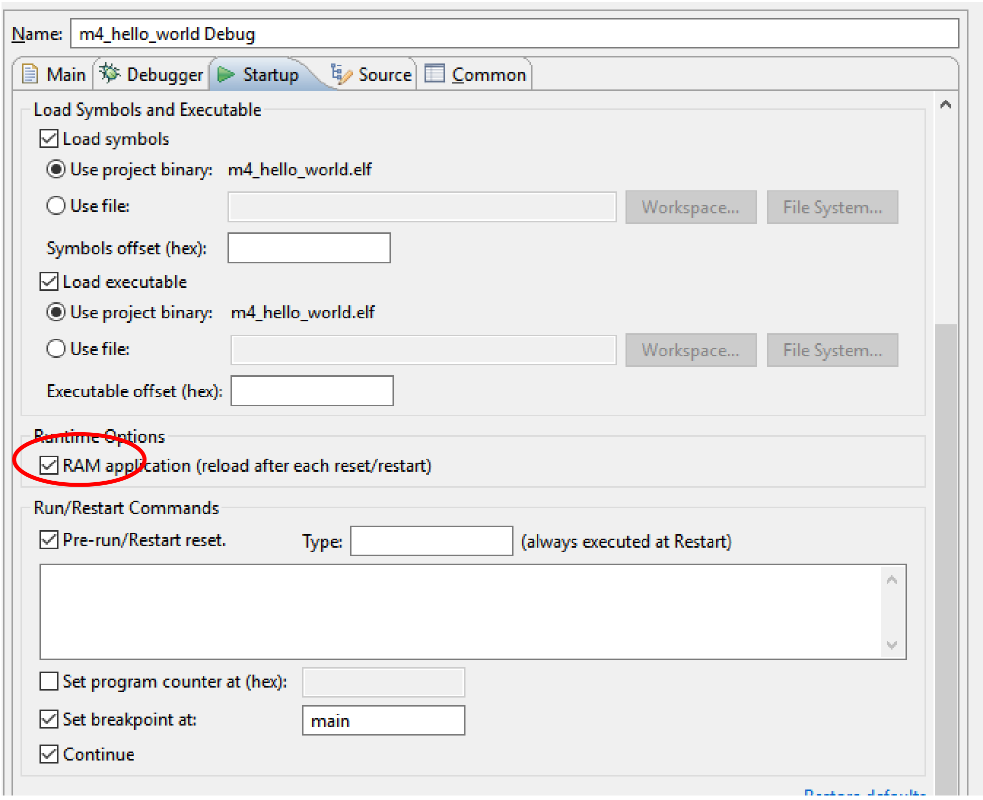

Figure 44 - Startup tab

6.7.6 Start a debug session

There are several ways to start a debug session. One way is to click on the “Debug” button if the “Debug configurations” window is still open as shown in Figure 45.

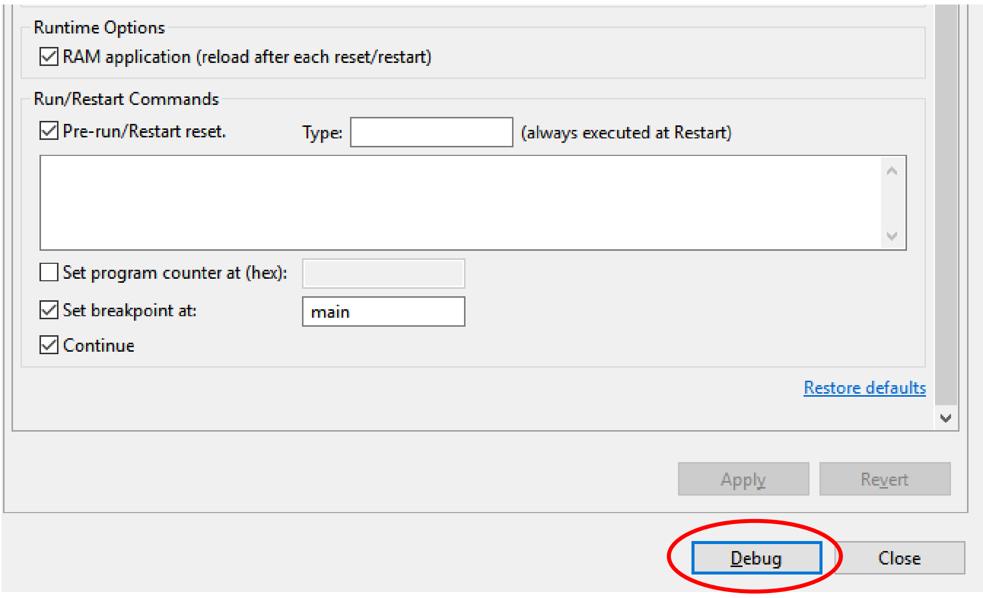

Figure 45 - Start Debug session

When starting the debug session the J -Link terms and conditions must be accepted by clicking the “Accept” button.

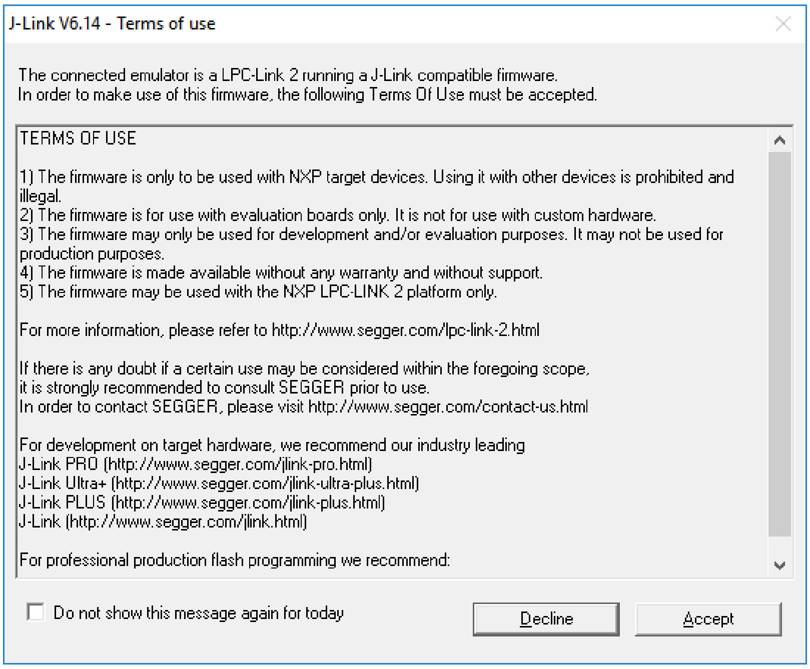

Figure 46 - J-Link Terms and conditions

Since we haven’t specified a correct device we have to select which target to debug. Select a generic Cortex-M4 as shown in Figure 47.

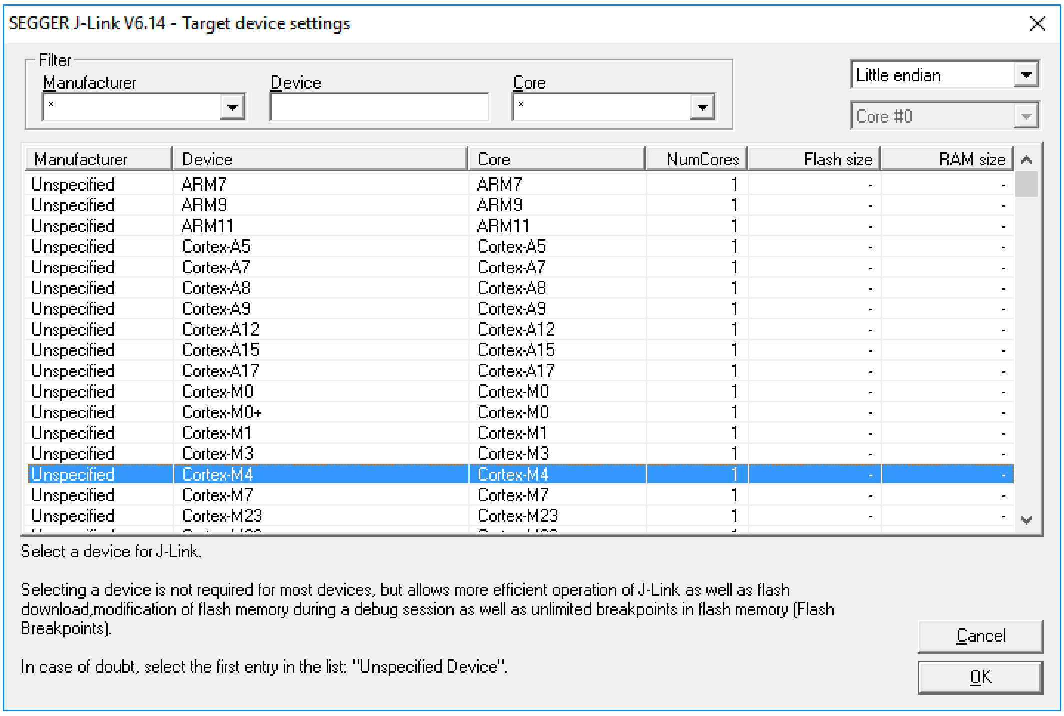

Figure 47 - J-Link device selection

Click Ok and the debug connection will be established.

> **NOTE 1**: We have seen that you might have to start an application on the Cortex-M4 before being able to debug a new application. Follow the instructions in section 4.5 to start an application.

> **NOTE 2**: One thing we have seen when debugging is that the second time you establish a debug session you can get a strange behaviour. The debug session will halt in the main function and you can single step, but when the FreeRTOS scheduler is started you end up in the prvPortStartFirstTask function and won't get out of this function. When writing these instructions we don't know the reason why this happens. The workaround is to reset the board between debug sessions.

6.8 Build with IAR Embedded Workbench

The FreeRTOS bundle contains project files for IAR Embedded Workbench and the documentation also contains instructions.

> **NOTE**: Embedded Artists has **not tested** the project files or documentation for IAR Embedded Workbench

7 Use DS-MDK for Application Development

DS-MDK is a commercial Eclipse based IDE and debugger from ARM/Keil. The development environment comes with support for NXP's application processors and especially those supporting Heterogeneous Multi-Processing such as the i.MX6 SoloX.

http://www2.keil.com/mdk5/ds-mdk/

This chapter describes how to install and use DS-MDK. The instructions are based on the document "Getting Started with DS-MDK" from ARM.

https://armkeil.blob.core.windows.net/product/gs_DS-MDK_5_24_2_en_rev3.pdf

7.1 Installation

Begin by installing MDK ARM. You will find the installer and instructions on the link below. Please note that MDK exists in a limited evaluation version, but it is a commercial product so if you want to continue to use it you need to buy a license.

https://www.keil.com/demo/eval/arm.htm

When MDK ARM has been installed download and install DS-MDK. Installer and instructions are available on the link below.

http://www2.keil.com/mdk5/ds-mdk/install/

When you start DS-MDK you have to specify where you installed MDK ARM and also choose a workspace directory for your project.

7.2 Package Manager

DS-MDK comes with a package manager that lets you install drivers and example programs for a specific device.

Open the Pack Manager by going to **Window** → **Perspective** → **Open Perspective** → **CMSIS Pack Manager** in the menu.

In the **Pack Manager**, go to **NXP** → **i.MX 6 Series** and then **i.MX 6SoloX**. In the Packs view click on **Install** button for the **Keil iMX6_DFP** package as shown in Figure 48.

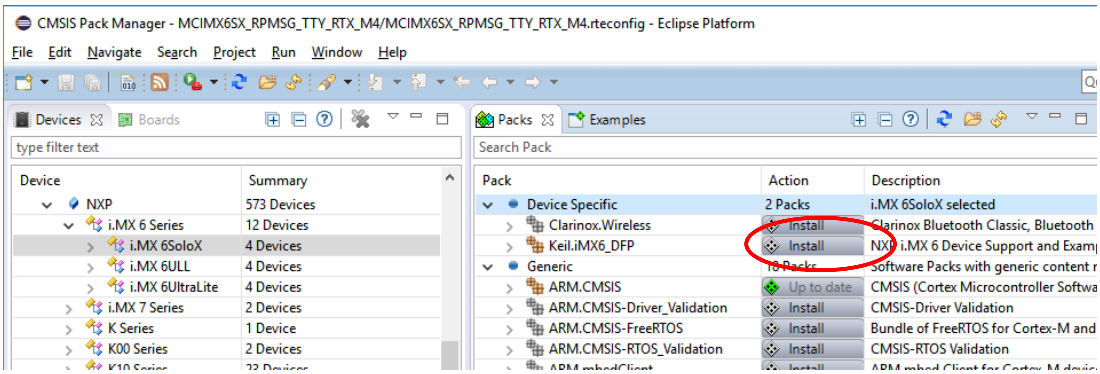

Figure 48 - CMSIS Pack Manager

When beginning with the application development it is recommended to use one of the existing example applications as a starting point. We are going to use the **RPMSG TTY** examples, that is, an application that show how to communicate between a Linux application running on the A9 core and an application running on the M4 core.

Go to the **Examples** tab in the Pack manager and then click on the **Copy** button for the **RPMSG TTY RTX** example as shown in Figure 49.

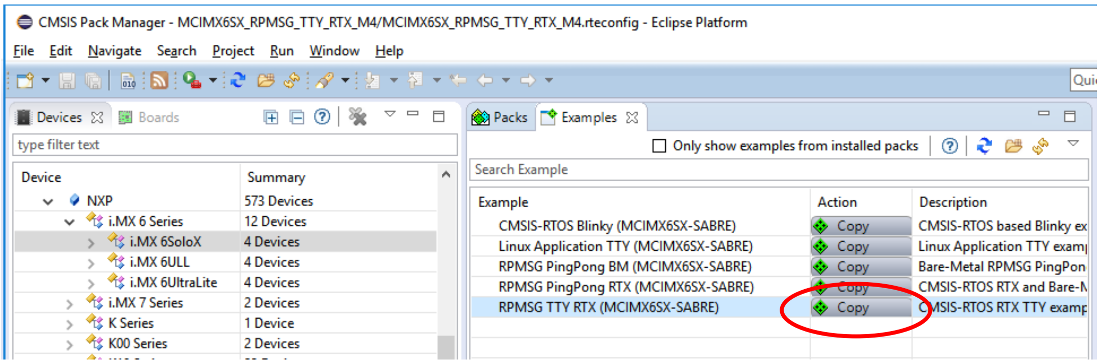

Figure 49 - RPMSG TTY Example

The application will now be added to your workspace. Go back to the **Pack Manager** and click on the **Copy** button for the **Linux Application TTY**. Now you have both the application that will run on the A9 core and the application that will run on the M4 core in your workspace.

7.3 UART Pin Muxing

The pin muxing for the application is done for NXP's Sabre board. You need to do the same changes as described in section 6.2.1 (for the FreeRTOS package). You should do these changes in the `configure_uart_pins` function in the `RTE/Board_Support/pin_mux.c` file.

7.4 Debug the M4 Application

7.4.1 Build the application

First build the application. Right-click on the RPMSG project and select **Build Project** as shown in Figure 50.

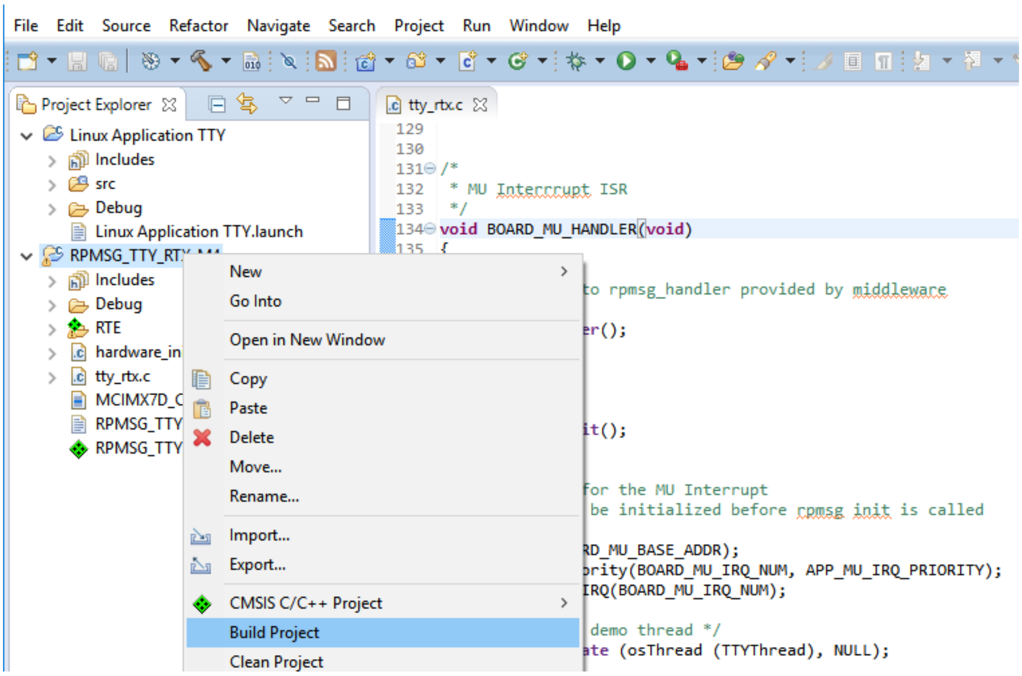

Figure 50 - Build Project

7.4.2 Setup the debug adapter

A debug adapter must be connected to the board before the application can be debugged. Section 6.4.1 shows how ULINKpro is connected to the board.

7.4.3 Create a debug configuration

Go to **Run → Debug configurations** in the menu. There should be a debug configuration called MCIMX6SX_RPMSG_TTY_RTX_M4 under the **CMSIS DS-5 Debugger** as shown in Figure 51.

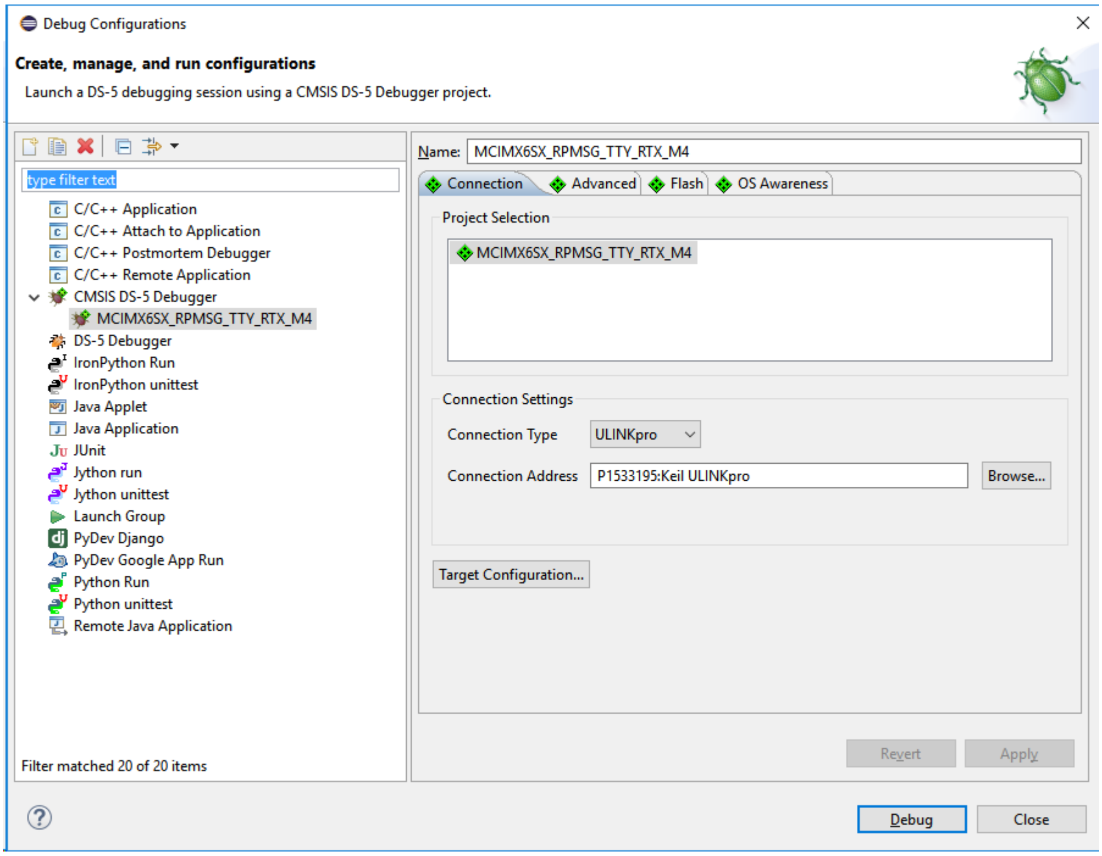

Figure 51 - CMSIS DS-5 Debug configuration

Click on the **Connection** tab and choose **Connection Type**. In Figure 51 a ULINKpro has been connected to the board. You have to select the debug adapter you are using and then click on the **Browse** button to find the actual connection (the adapter must be connected to your computer). When writing these instructions the following debug adapter types could be used.

- DSTREAM
- ULINKpro
- CMSIS-DAP

The default settings were used for all other settings. Below are screen shots for the other tabs.

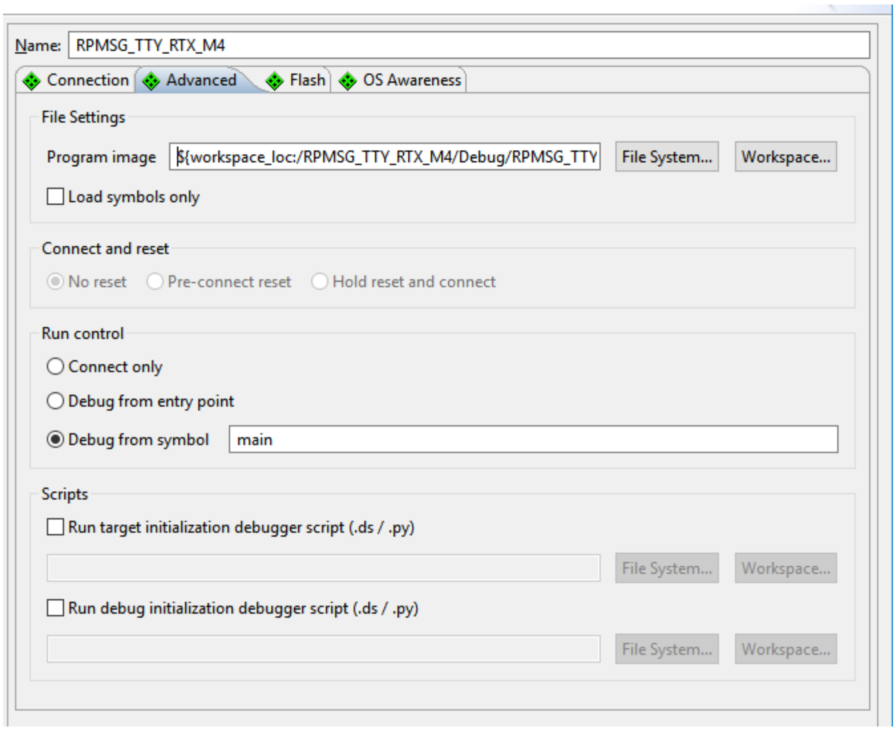

Figure 52 - Advanced tab

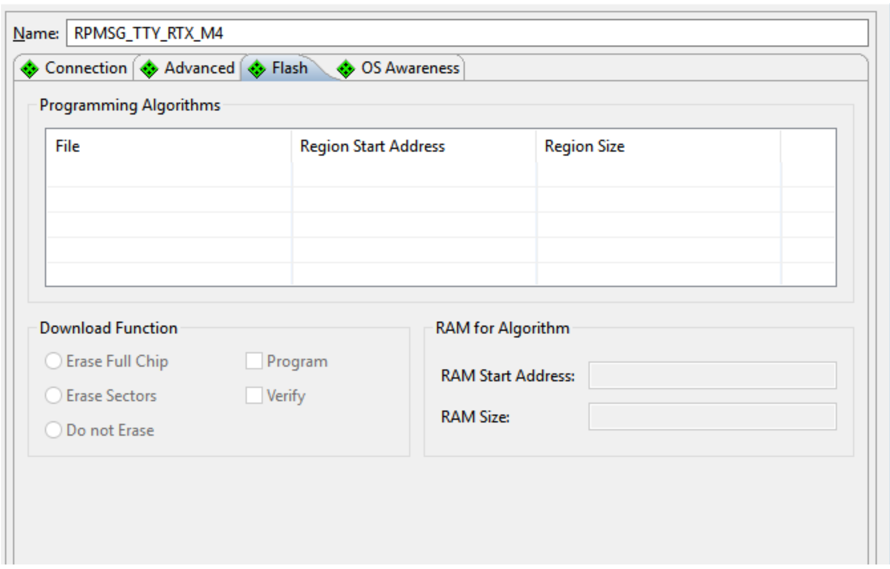

Figure 53 - Flash tab

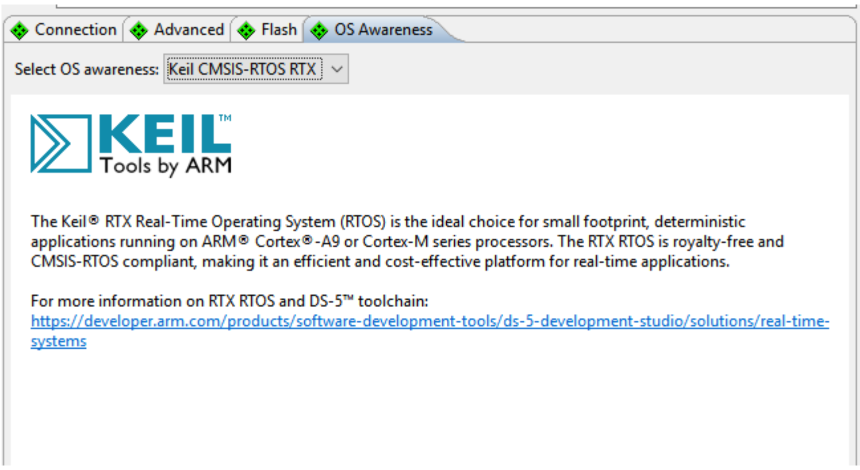

Figure 54 - OS Awareness

When the debug configuration is ready click on the **Debug** button and a debug session will be established as shown in Figure 55.

> **NOTE**: Make sure that you have only booted into **u-boot** on the Cortex-A9 and not into Linux. See section 7.6 for information about simultaneous debugging of Cortex-M4 and Cortex-A9.

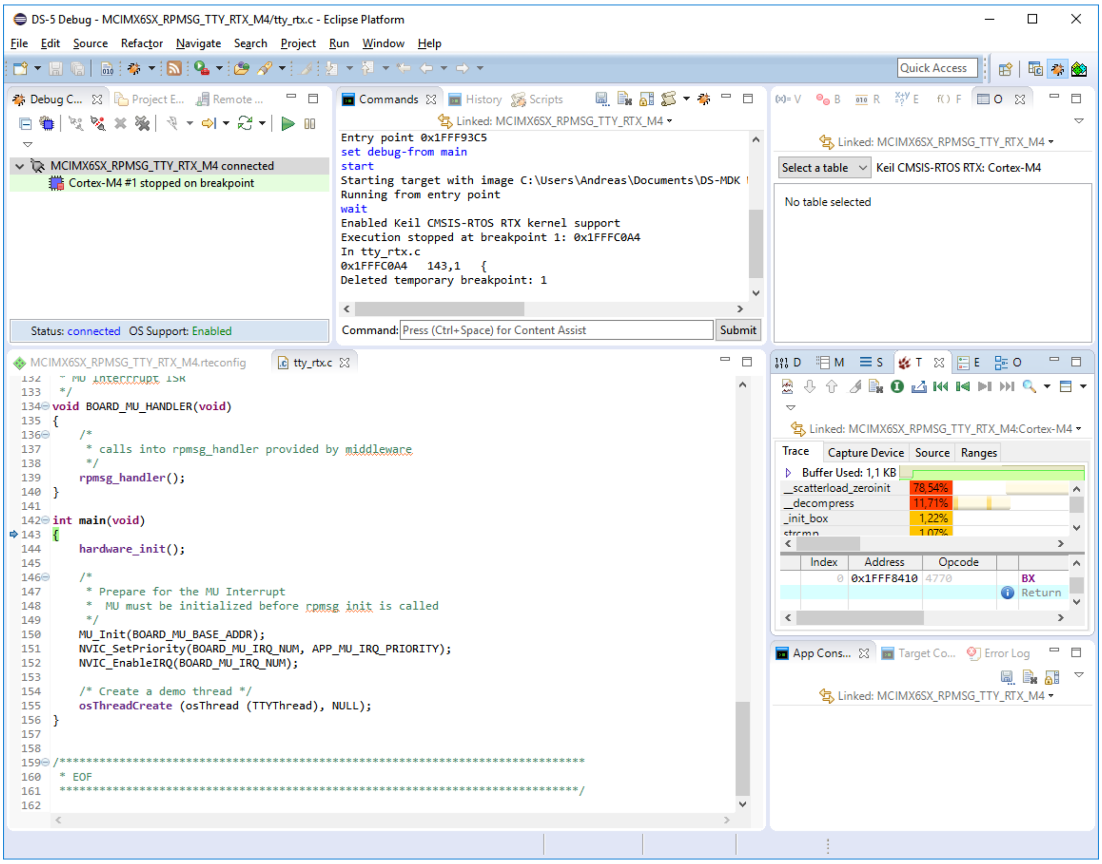

Figure 55 - Debug session

7.5 Debug the Linux Application

The Linux application will be debugged using `gdbserver` over a network connection. This means that there is no need to use the debug adapter (such as ULINKpro) when debugging the Linux application. It is however necessary to have the board connected to the same network as your development computer.

7.5.1 Build the application

First build the application. Right-click on the **Linux Application TTY** project and select **Build Project** as shown in Figure 56.

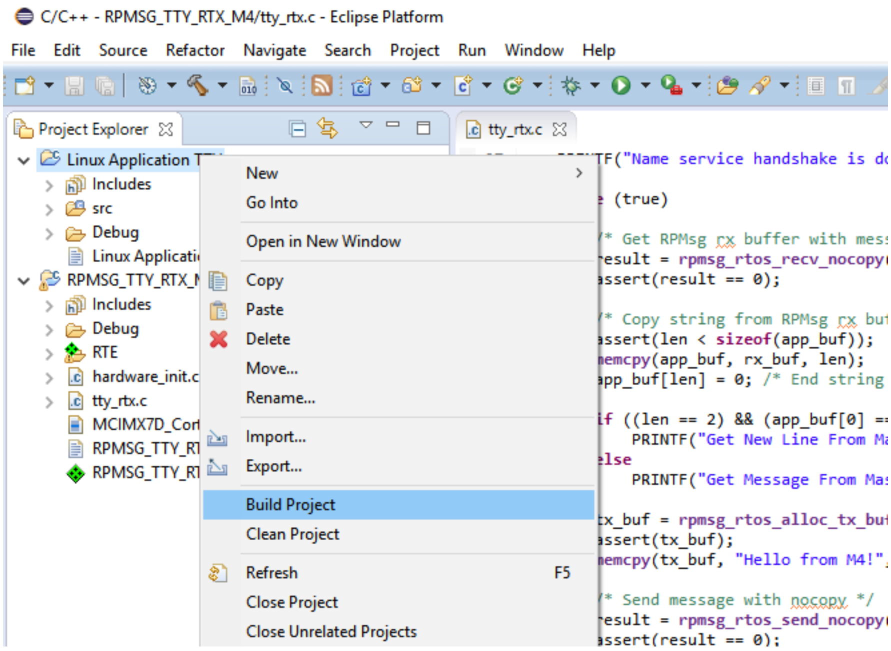

Figure 56 - Build Linux application

7.5.2 Setup Remote System Explorer (RSE)

First get the IP address of the board. You can get this by using the `ifconfig` utility as shown below via a terminal application connected to the board..

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr CA:71:64:BD:1A:20
          inet addr:192.168.1.72  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80:
```

In DS-MDK, go to **Window** → **Perspective** → **Open Perspective** → **Other** and then **Remote System Explorer**. Click on the icon shown in Figure 57 to create a connection.

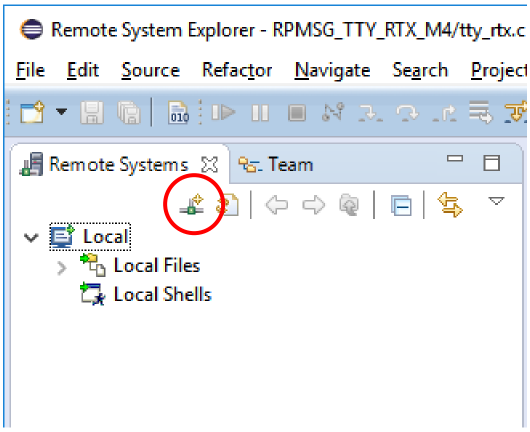

Figure 57 - RSE Perspective

Choose **SSH Only** as connection type as shown in Figure 58 and then click **Next**.

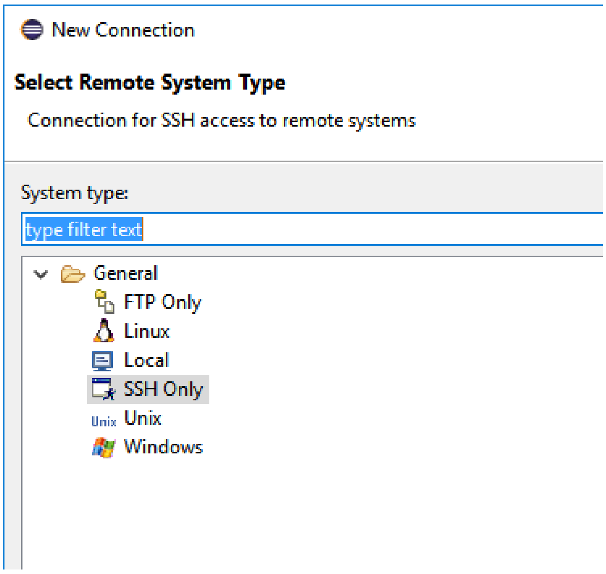

Figure 58 - Remote System Type

Enter the IP address in the **Host name** field as shown in Figure 59 and then click **Finish** to create the connection.

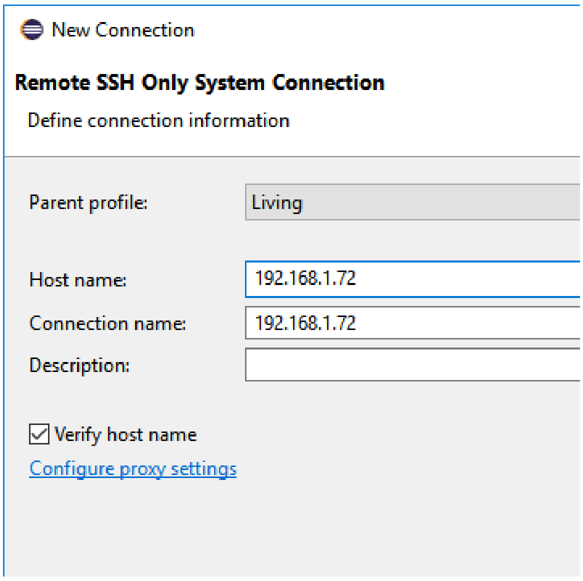

Figure 59 - Host name / IP address

It could now look like in Figure 60. If you click on **Sftp Files → My Home** you will see the home directory on the target. You will be asked to enter the user name (`root`) and password (`pass`) to login.

> **NOTE**: By default root is not permitted to login over SSH. Read section 8.2 for a solution to this problem.

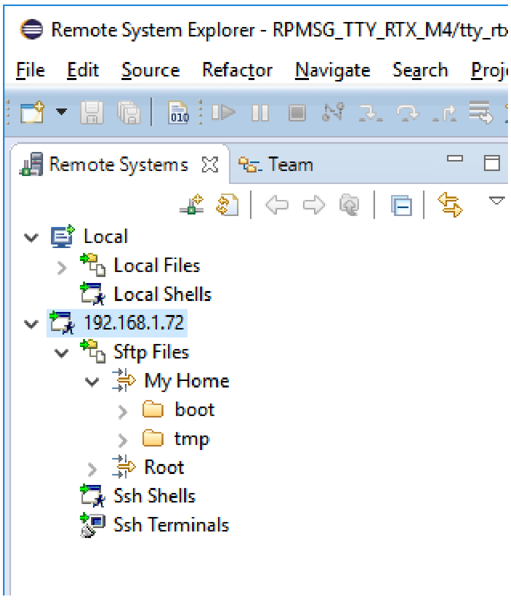

Figure 60 - Created RSE connection

7.5.3 Create Debug Configuration

Go to **Run → Debug configurations** in the menu. There should be a debug configuration called MCIMX6SX_Linux_Application_TTY under the **DS-5 Debugger** as shown in Figure 61. Click on this configuration and go to the **Connection** tab. Select **Download and debug application** and make sure the RSE connection we created earlier is used under **Connections**.

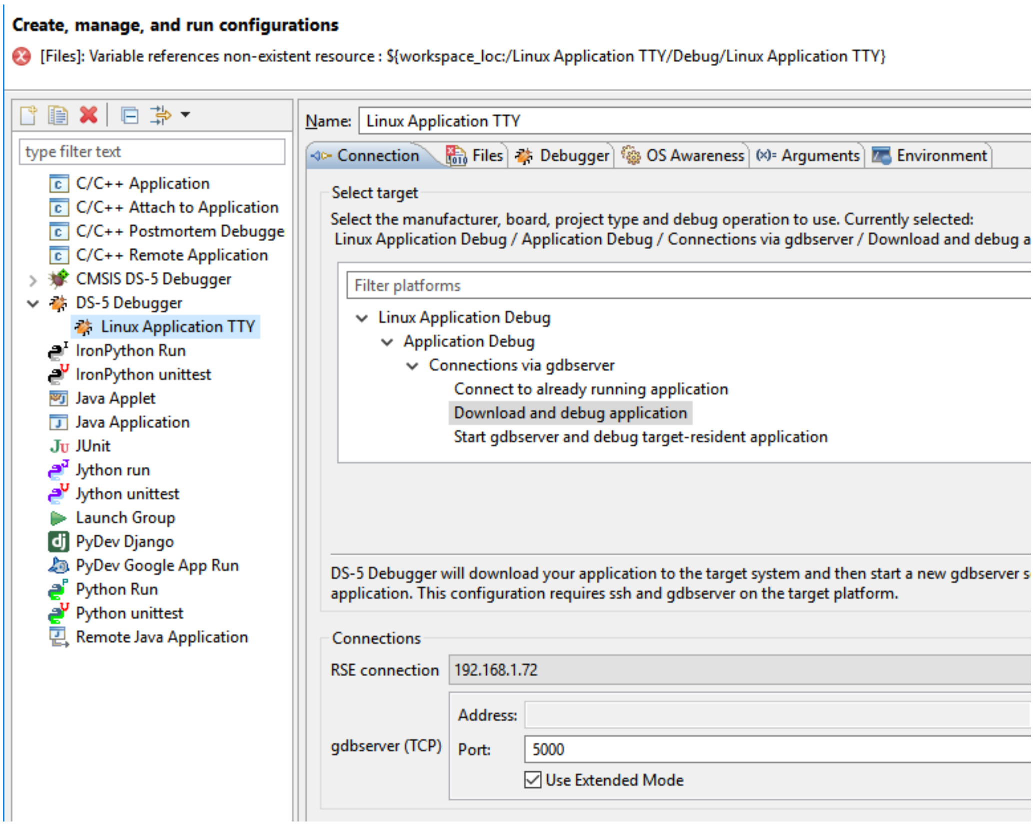

Figure 61 - DS-5 Debugger configuration

Go to the **Files** tab and select download and working directory. In this example we are using `/home/root/tmp` as shown in Figure 62.

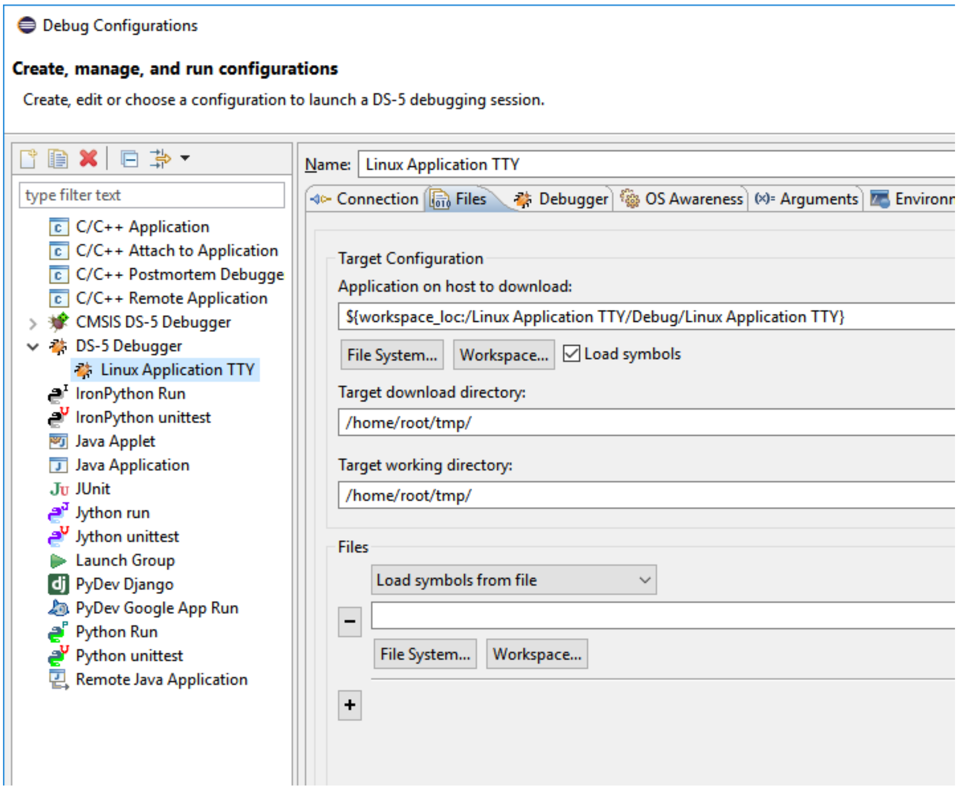

Figure 62 - Files tab

In the **Debugger** tab make sure **Debug from symbol** is chosen and the symbol is set to **main** as shown in Figure 63.

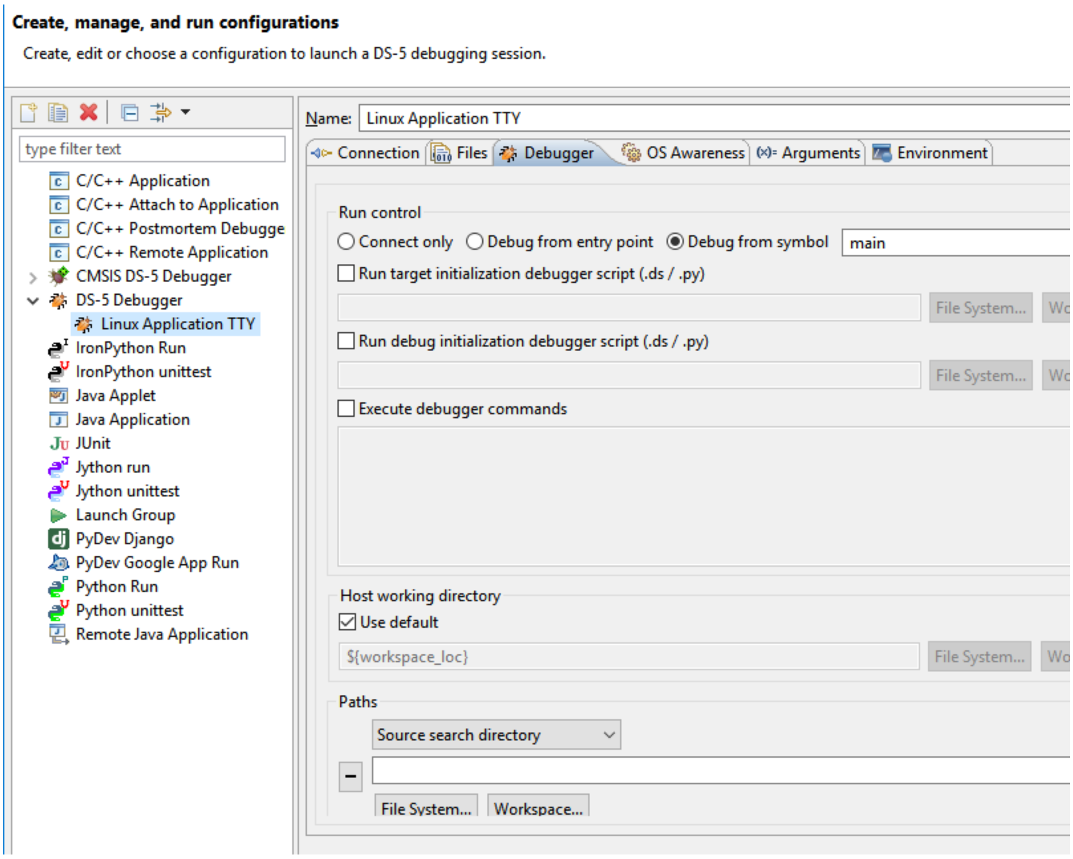

Figure 63 - Debugger tab

Click on the **Debug** button to start the debug session.

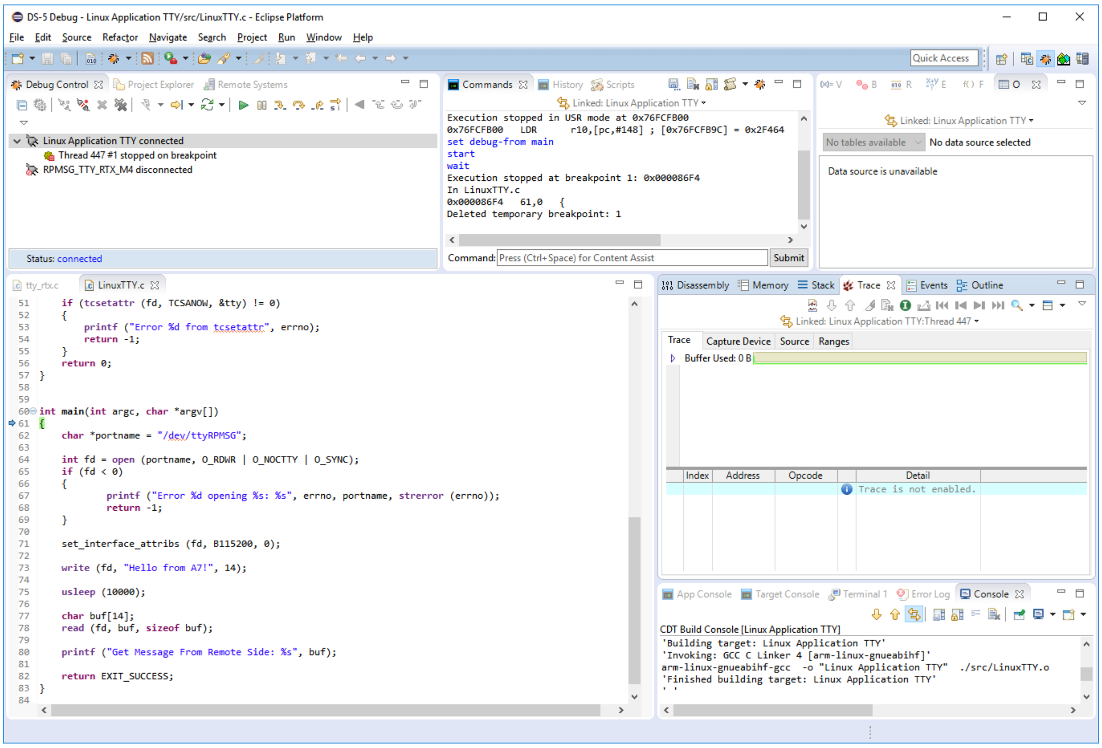

Figure 64 - Debug session of Linux application

7.6 Simultaneous Debugging

Follow these steps to simultaneously debug RPMSG_TTY_RTX_M4 and **Linux Application TTY**.

1. Boot into u-boot
2. Change device tree file (dtb) file and also `mmcargs` . The boot argument `uart_from_osc` must be set to make Cortex-A9 and Cortex-M4 UART clocks match

```
=> setenv fdt_file imx6sxea-com-kit-m4.dtb
=> setenv mmcargs "${mmcargs} uart_from_osc"
=> save
```

3. Now start the debug session of RPMSG_TTY_RTX_M4 as described in section 7.4 above.
4. It is not possible to interact with u-boot while RPMSG_TTY_RTX_M4 is halted until RDC has been initialized. RDC will be initialized in `BOARD_RdcInit` which is called from `hardware_init`. Let at least the call to the function `hardware_init` execute and you will be able to interact with u-boot.
5. Enter `boot` in the u-boot console to boot Linux

```
=> boot
```

6. To be able to use the RPMsg TTY channel a kernel module must be loaded. When Linux has booted run the following:

```
# modprobe imx_rpmsg_tty
imx_rpmsg_tty rpmsg0: new channel: 0x400 -> 0x0!
Install rpmsg tty driver!
```

7. You can double-check that the module has been loaded by using `lsmod`.

```
# lsmod
Module                  Size  Used by
imx_rpmsg_tty           3418  0
...
```

8. When the module has been loaded, start the debug session of the Linux application as described in 7.5 above.
9. You should now be able to debug the Linux application, for example, single step and when a message is sent to the M4 application the M4 debug session should halt on the breakpoint at `rpmsg_rtos_recv_nocopy`.

8 Troubleshooting

8.1 JTAG connection problem when Linux has booted

8.1.1 Description of problem

It is not possible to make a debug connection to the target via JTAG when Linux has booted. It is possible to establish a connection before Linux has booted, such as when the u-boot bootloader is active.

8.1.2 Solution

The possible solutions were originally described on the NXP community:

https://community.nxp.com/thread/376786

Method 1 – Disable 'clock off' wait state in cpuidle driver

It is possible to disable clock off through sysfs.

```
# cd /sys/devices/system/cpu/cpu0/cpuidle/state1
```

Make sure this is the correct state or else change to one of the other state folders. Reading the `desc` node should give the result "Clock off" as shown below.

```
# cat desc
Clock off
```

Disable the wait state

```
# echo 1 > disable
```

Method 2 – Disable gating of the ARM clock domain

If you need to debug during startup you need to modify the source code. Apply the patch below.

```
diff --git a/arch/arm/mach-imx/pm-imx6.c b/arch/arm/mach-imx/pm-
imx6.c

index e1a45e2..feadccb 100644
--- a/arch/arm/mach-imx/pm-imx6.c
+++ b/arch/arm/mach-imx/pm-imx6.c
@@ -552,8 +552,8 @@ int imx6q_set_lpm(enum mxc_cpu_pwr_mode mode)
        case WAIT_CLOCKED:
                break;
        case WAIT_UNCLOCKED:
-               val |= 0x1 << BP_CLPCR_LPM;
-               val |= BM_CLPCR_ARM_CLK_DIS_ON_LPM;
                break;
        case STOP_POWER_ON:
                val |= 0x2 << BP_CLPCR_LPM;
```

8.2 Allow user "root" to use an SSH connection

By default the user "root" is not permitted to login via an SSH connection. By following these instructions "root" will be permitted to login through an SSH connection. It is, however, not recommended to use on a final application, but during development it can be permitted.

1. Open the configuration file for the SSH server

```
# nano /etc/ssh/sshd_config
```

2. Find the line that starts with #PermitRootLogin and remove the '#' (hash) character. If you cannot find this line just add it to the file (without the hash)

```
PermitRootLogin yes
```

3. Save the file and exit the editor (in `nano` it is Ctrl-X followed by Y and Enter).
4. Restart the SSH server

```
# /etc/init.d/sshd restart
```

8.3 Linux (A9) terminal/console doesn't accept input while debugging M4

When you are debugging the M4-core and more specifically when you have halted the M4-core from within the debugger it can seem as the Linux terminal/console is unresponsive (doesn't accept any input).

Solution

First of all make sure you have updated u-boot and Linux to the version (or later) publish 2017-09-22. In this release u-boot was updated to include RDC initialization. The commit is available below in case you need to run on older versions.

https://github.com/embeddedartists/uboot-imx/commit/8bbbd16c8f846f530ccd1f7ee931aff05099f944

Secondly your M4-application must have assigned the M4 to domain 1 as shown below. If you are using the example code from NXP this call is being made in `board.c` → `BOARD_RdcInit`. `BOARD_RdcInit` is called from `hardware_init.c` → `hardware_init`.

```
RDC_SetDomainID(RDC, rdcMdaM4, BOARD_DOMAIN_ID, false);
```