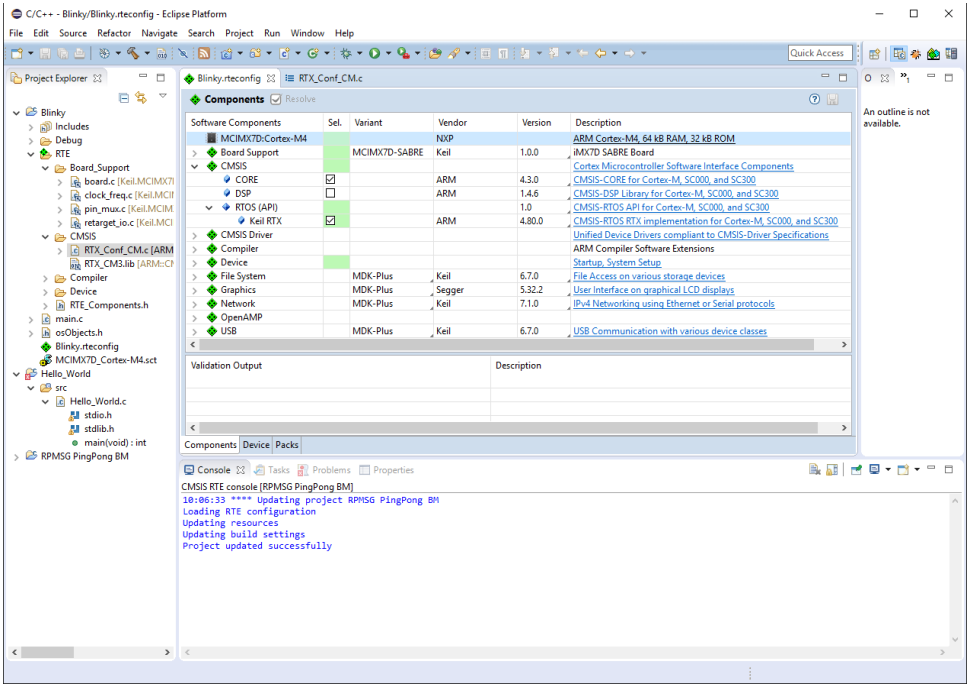
ARM KEIL

Microcontroller Tools

Getting started with DS-MDK

Create applications for heterogeneous ARM® Cortex®-A/Cortex-M devices



NOTE
We assume you are familiar with Microsoft Windows, the hardware, and the instruction set of the ARM® Cortex®-A and Cortex-M processors.

Every effort was made to ensure accuracy in this manual and to give appropriate credit to persons, companies, and trademarks referenced herein.

Preface

Thank you for using the DS-MDK Development Studio available from ARM. To provide you with the very best software tools for developing ARM based embedded applications we design our tools to make software engineering easy and productive. ARM also offers therefore complementary products such as the ULINK™ debug and trace adapters and a range of evaluation boards. DS-MDK is expandable with various third party tools, starter kits, and debug adapters.

Chapter overview

The book starts with the installation of DS-MDK and describes the software components along with complete workflow from starting a project up to debugging on hardware. It contains the following chapters:

DS-MDK introduction provides an overview about the DS-MDK, the software packs, and describes the product installation.

Working with example projects explains how to get started with supported development boards using pre-built projects to verify hardware and software functionality.

Creating projects from scratch guides you through the process of creating and modifying projects using CMSIS and device-related software components for the Cortex-M microcontroller. It also shows you how to develop applications for the Cortex-A processor running Linux.

Debug applications describes the process of how to connect to the target hardware and explains debugging applications on the target.

Store Cortex-M image gives further details on how to store the application image on the target and how to run it at start up time.

The **Appendix** contains further information, for example about the basic concepts of the Eclipse IDE and the most frequently used perspectives.

Contents

NOTE

This user's guide describes how to create applications with the Eclipse-based DS-MDK IDE and Debugger for ARM Cortex-A/Cortex-M based devices.

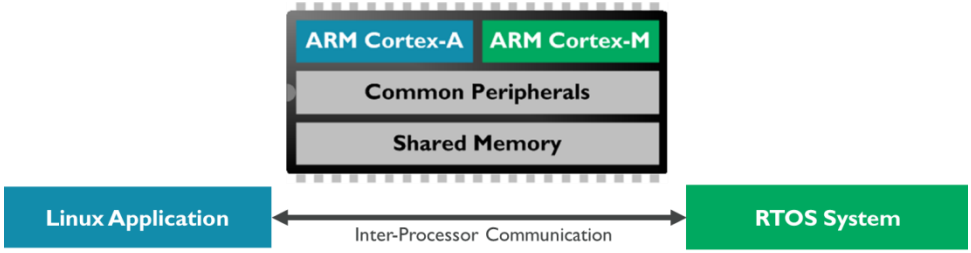
Refer to the ***Getting Started with MDK*** *user's guide for information how to create projects for ARM Cortex-M microcontrollers with the µVision® IDE/Debugger.*

DS-MDK introduction

DS-MDK combines the Eclipse-based DS-5 IDE and Debugger with CMSIS-Pack technology and uses software packs to extend device support for devices based on 32-bit ARM Cortex-A processors or heterogeneous systems based on 32-bit ARM Cortex-A and ARM Cortex-M processors.

Currently NXP i.MX 6, i.MX7 and VFxxx series devices are supported. These devices combine computing power for application-rich systems with real-time responsiveness: the DS-5 Debugger gives visibility to multi-processor execution and allows optimization of the overall software architecture.

Solution for heterogeneous systems



Heterogeneous systems usually consist of a powerful ARM Cortex-A class application processor and a deterministic ARM Cortex-M based microcontroller. These systems combine the best of both worlds: the Cortex-A class processor can run a feature-rich operating system such as Linux and enables the user to program complex applications with sophisticated human-machine interfaces (HMI). The Cortex-M class controller offers low I/O latency, superior power efficiency and a fast system start-up time for embedded systems.

Usually, both processors have access to a set of communication peripherals and shared memory. The biggest challenge with heterogeneous systems is the synchronization and inter-processor communication.

DS-MDK offers a complete software development solution for such systems:

- Manage Cortex-A Linux and Cortex-M RTOS projects in the same development environment.
- Use the Cortex Microcontroller Software Interface Standard (**CMSIS**) development flow for efficient Cortex-M programming. Add software packs any time to DS-MDK to make new device support and middleware updates independent from the toolchain. The IDE manages the provided software components that are available for the application as building blocks.
- Debug multicore software development projects with the full visibility offered by the DS-5 Debugger.

DS-MDK licensing

DS-MDK is part of the **Keil® MDK** and the product requires a valid license in order to use it.

For information on how to obtain and set-up the license, please refer to the following page: http://www.keil.com/mdk5/ds-mdk/licensing/

Software and hardware requirements

DS-MDK has the following minimum hardware and software requirements:

- *A workstation running Microsoft Windows, Red Hat Enterprise Linux or Ubuntu Desktop Edition (only 64-bit OS/platforms are supported)*
- *Dual-Core Processor with > 2 GHz*
- *4 GB RAM and 8 GB hard-disk space*
- *1280 x 800 or higher screen resolution*

Install DS-MDK

Download the **DS-MDK** installer for your host platform (Windows or Linux) from **www.keil.com/mdk5/ds-mdk/install** .

The installation procedures for Windows and Linux are different and are both described below.

Windows installation

Decompress the zip archive and run the installer setup.exe. Follow the instructions on the screen and make sure you install the device drivers for the debug probes.

To start DS-MDK, use **Eclipse for DS-MDK** from the Start menu (Windows 10: **All apps → ARM DS-MDK → Eclipse for DS-MDK**).

Linux installation

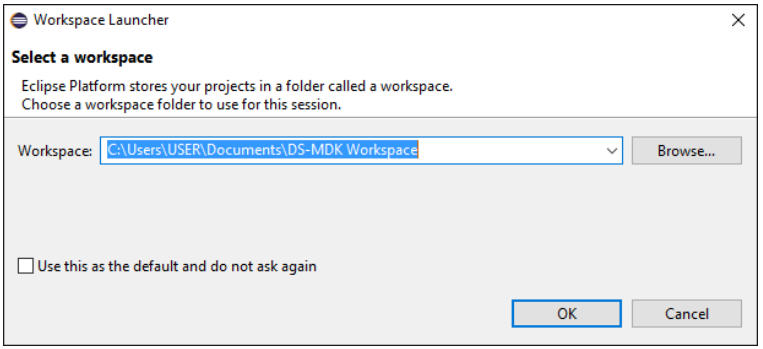
Extract the installer from the downloaded archive file, run (not source) *install.sh* and follow the on-screen instructions. The installer unpacks DS-MDK into your chosen directory, and optionally installs device drivers and desktop shortcuts.

Note: The installer includes device drivers that require you to run with root privileges.

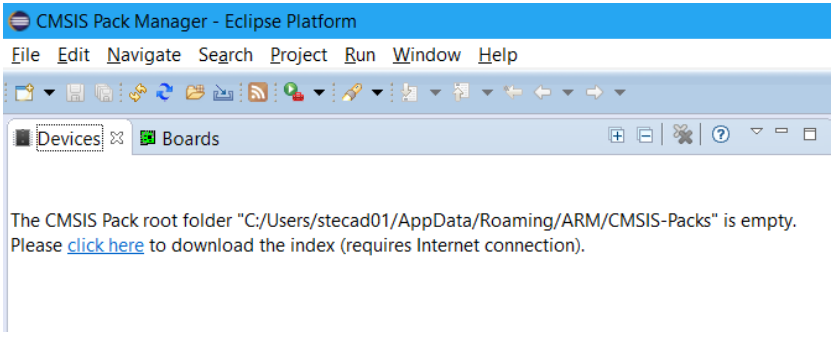
To start DS-MDK, from your desktop, select Eclipse for DS-MDK. Alternatively, launch *[DS-MDK install directory]/bin/eclipse* from the command line.

Run DS-MDK

The first time you run DS-MDK, a window would appear asking to specify a directory for your workspace (the area where your projects will be stored). For most users, the default suggested directory is the best option.

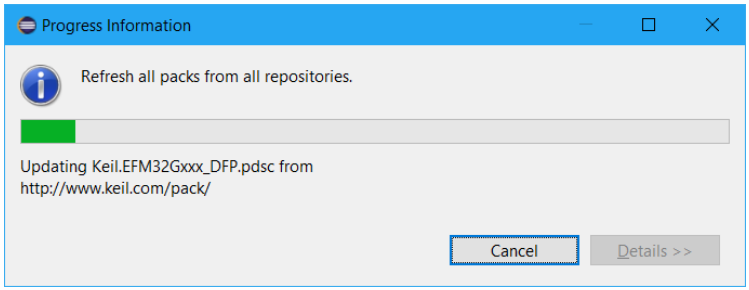


The Eclipse-based IDE opens in the *CMSIS Pack Manager* perspective and a warning message is shown if the default CMSIS Pack directory is empty.

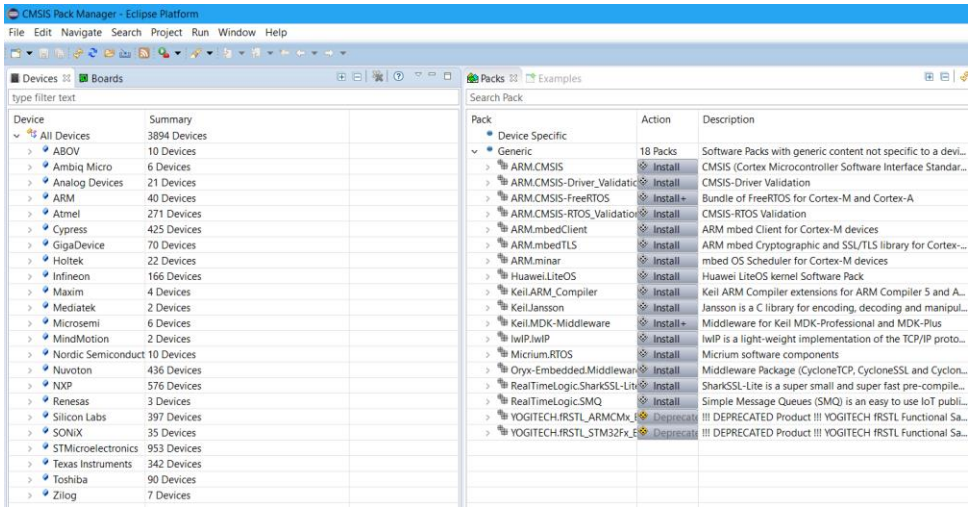


Click on the highlighted *click here* text to start populating the CMSIS Index: this operation requires an Internet connection to download the index files.

DS-MDK shows a progress bar during the download.



At the end of the process, the CMSIS Pack Manager view should be populated with the CMSIS Packs available.



NOTE

Currently, software packs for the NXP i.MX 6, i.MX 7 and VFxxx series are qualified for DS-MDK.

The **Console** window shows information about the Internet connection and the installation progress.

The device database (**www.keil.com/dd2**) lists all available devices and provides download access to the related software packs. If the Pack Manager cannot access the Internet, you use the **Import existing packs** icon or double-click on *.PACK files to manually install software packs.

Documentation and support

DS-MDK provides online manuals and context-sensitive help. The **Help** menu opens the main help system that includes the *CMSIS C/C++ Development User's Guide*, the *ARM DS-MDK Documentation*, the *RSE User Guide*, and other reference guides.

Many dialogs have context-sensitive **Help** buttons that access the documentation and explain dialog options and settings.

If you have suggestions or you have discovered an issue with the software, please report them to us. Support and information channels are accessible at **www.keil.com/support**.

Working with example projects

Install the Linux image

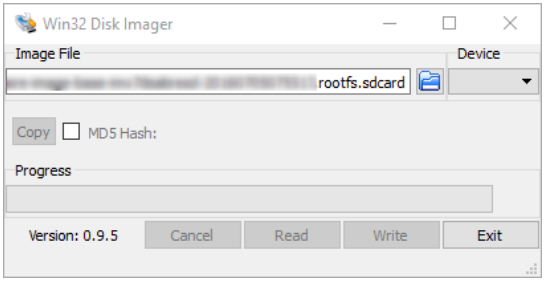
For every supported development board, a pre-configured Linux image with DS-MDK specific debug settings is available. This web page lists all supported development boards: **www.keil.com/mdk5/ds-mdk/install#boards**

☞ Download the compressed Linux kernel for your development board and unzip it.

Copy the Linux image to an SD-Card (Windows)

☞ Download and install the open source tool **Win32 Disk Imager** from **http://win32diskimager.sourceforge.net/** to flash the Linux kernel image onto an SD-Card.

Run the program. To write the image to the memory card, specify the location of the image file, select the **Device** letter of the SD card and press the **Write** button:



Copy the Linux image to an SD-Card (Linux)

☞ To write the image on the memory card on Linux it's sufficient to use the dd command where /dev/sdx is the device for your memory card.

> ***NOTE***
> *Make sure you select the right /dev/sdx device to avoid corruption of your data on your drives.*

```
# sudo dd if=image_file_name of=/dev/sdx bs=1M`
```

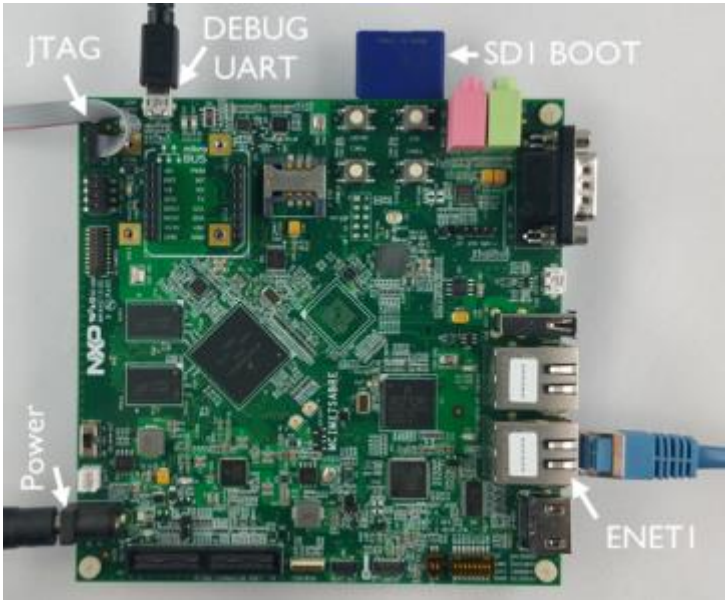
Hardware connection

In order to fully debug the target device you need to use a JTAG debugger such as DSTREAM or ULINK*pro*. The debugger needs to be connected to the host PC via USB (DSTREAM/ULINK*pro*) or Ethernet (DSTREAM only) and the target board via JTAG connector.

For the debug of Linux applications via gdbserver an Ethernet connection from the host PC to the board is required.

Another required connection during debug is the UART port used to interact with the Linux console: some boards have an RS232 connector whereas others have an USB interface that the operating system recognizes as virtual COM ports.

The picture below shows an example (NXP i.MX7 SABRE board) connected with JTAG, Ethernet and USB UART connections.



If you are not sure how to connect your board, please follow the instructions on the development board's support page.

Verify installation with example projects

Once you have selected, downloaded, and installed a software pack for your device, you can verify your installation using one of the examples provided in the software pack. For more information about the example used in this section, please refer to Remote Processor Messaging protocol example on page 63 in the Appendix.

Prepare terminal views

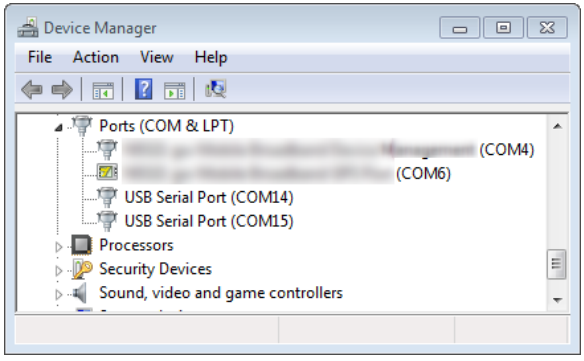
Many applications use a serial device to display messages. A *Terminal* window shows these messages from serial ports.

The NXP i.MX 7 SABRE development board for example contains a dual USB serial port device with two independent serial ports. The configuration of the serial port is slightly different between Windows and Linux platforms.

Windows

Connect the board to your computer. Windows installs the drivers automatically and adds two new USB Serial Ports to your system.

Check the exact numbers in the Windows **Device Manager** (to open it, type “device manager” in the Windows search bar):



The smaller number is the COM port of the Cortex-A processor, while the larger number is the COM port of the Cortex-M processor.

Linux

Connect the board to your computer. Linux should recognize the peripheral and you should be able to find ttyUSB0 and ttyUSB1 in your /dev/ directory.

Please make you set the right read/write permission to the device. For example, to give read/write permissions to all users on your machine type the following command:

```
root@imv7dsabresd:~# sudo chmod 666 /dev/ttyUSB*
```

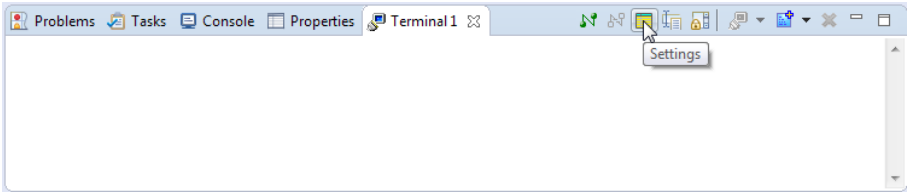
```
                    :~$ ls /dev/ -oa | grep USB
crw-rw-rw-   1 root 188,   0 Apr 21 15:36 ttyUSB0
crw-rw-rw-   1 root 188,   1 Apr 21 15:32 ttyUSB1
                    :~$ sudo chmod 666 /dev/ttyUSB*
                    :~$
```

The first device (e.g. /dev/ttyUSB0) is the serial port of the Cortex-A processor, while the second device (e.g. /dev/ttyUSB1) is the serial port of the Cortex-M processor.

Windows and Linux

☞ On DS-MDK, go to **Window → Show View → Other…** to open a *Terminal* view. Select **Terminal → Terminal** and click **OK**.

Open the settings dialog from the toolbar of the **Terminal 1** window:

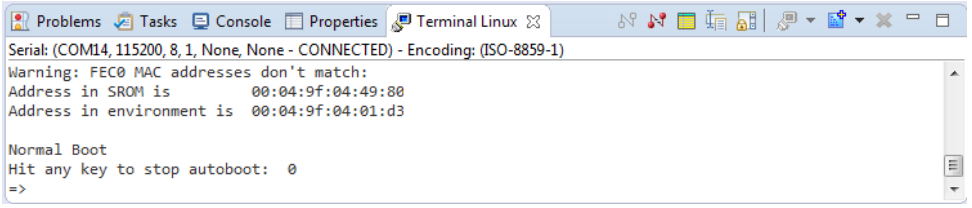


☞ Set the following and click OK:

- View Title: Terminal Linux
- Connection Type: Serial
- Port: Use the first of the new serial ports (e.g. COM14 or /dev/ttyUSB0)
- Baud Rate: 115200

NOTE
For the correct terminal settings and hardware connections of your development board refer to the board support pages.

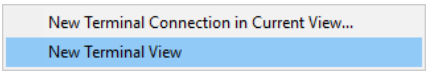
☞ Press the reset button on the development board to observe the boot process in the *Terminal* window. Press any keyboard key to interrupt the boot process:



NOTE

You must halt the boot loader at this point to be able to connect the ULINK*pro debug adapter to the Cortex-M processor.*

☞ Add another *Terminal* view to display the output of the Cortex-M processor. Simply use the drop-down selector next to the **New Terminal Connection in Current View…** icon and select **New Terminal View**:



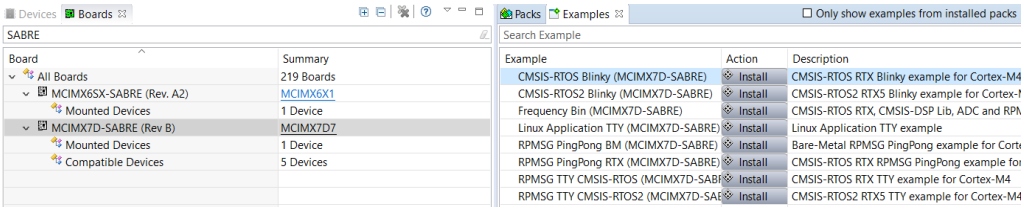
Select the second serial port number and leave the other settings as they are. Name the *Terminal* view **Terminal M4**.

Cortex-M application

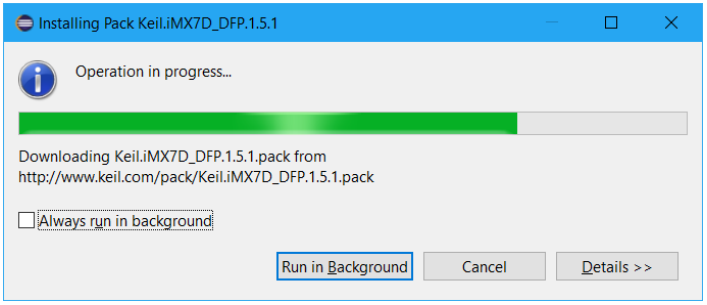
Copy the RPMSG TTY RTX example project

Select the device

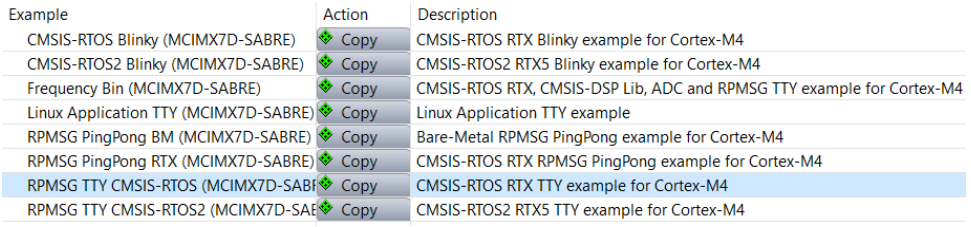
☞ In the **CMSIS Pack Manager** (🧩) perspective, select the board (**MCIMX7D-SABRE**) from the **Boards** tab on the left and click on **Examples** tab on the right-hand side of the window. Use filters in the toolbar to narrow the list of examples.



Click **Install** next to the **RPMSG TTY RTX** example if the packs are not installed (this might take a few minutes based on your internet connection).

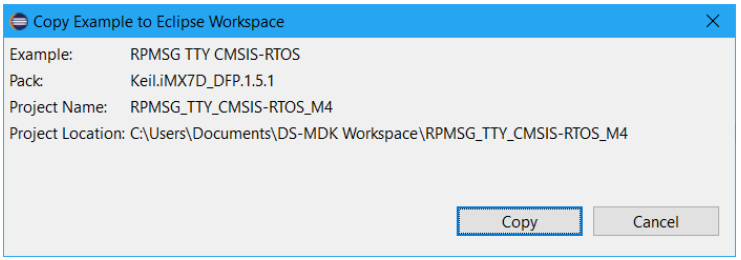


At the end of the installation the CMSIS Packs for the selected board should be installed locally and the examples are ready to be copied in your workspace.

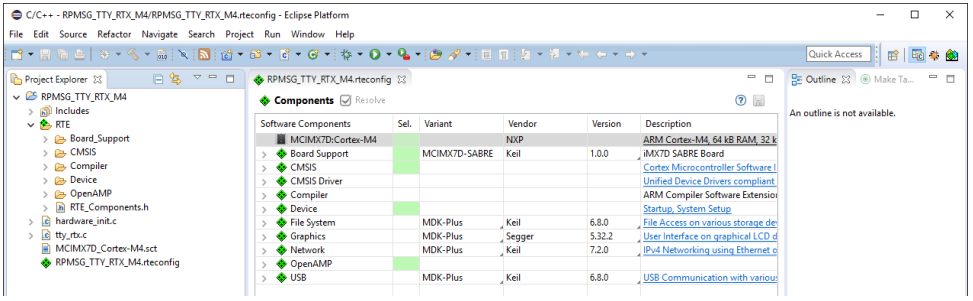


Click **Copy** next to the **RPMSG TTY RTX** example (make sure the corresponding pack is installed).

Confirm your selection by clicking on the **Copy** button.

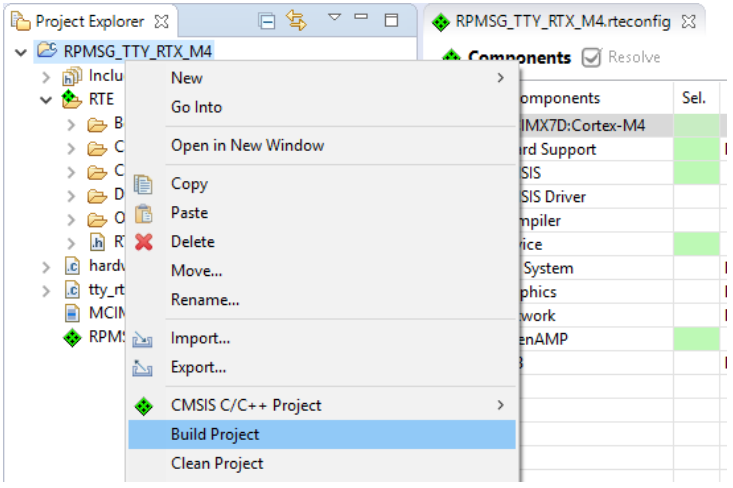


CMSIS Pack Manager copies the example into your workspace and switches to the *C/C++ perspective*:

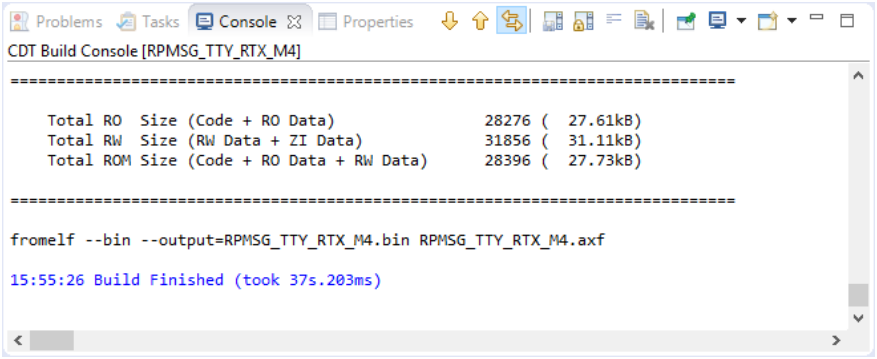


Build the application

Build the project from the context menu in the **Project Explorer**:

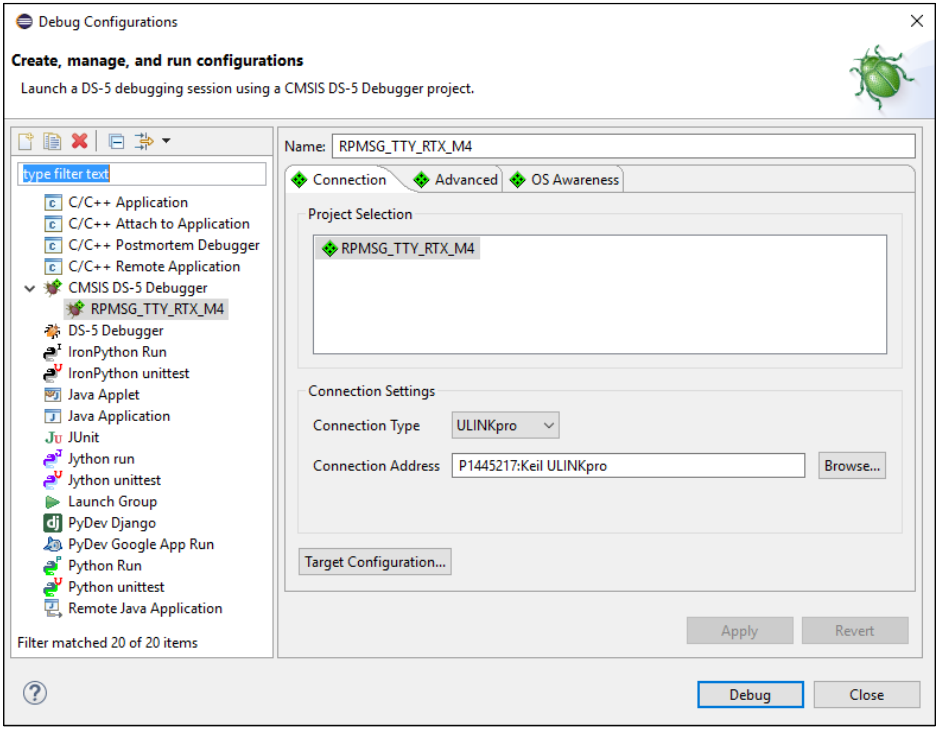


The *Console* window shows information about the build process:



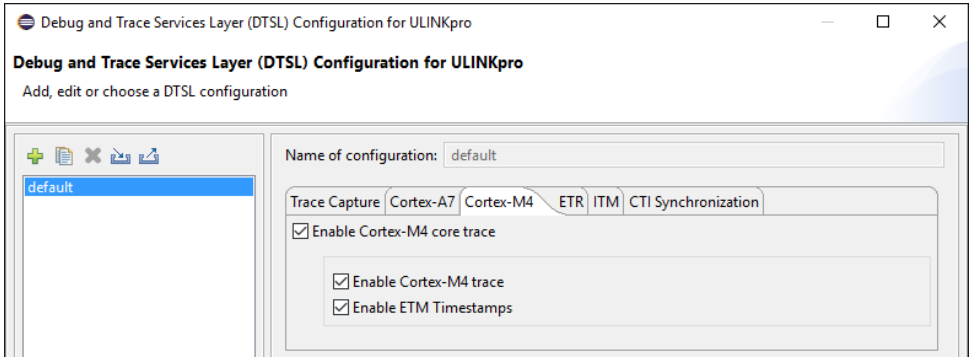
Configure CMSIS DS-5 debugger

☞ Right-click the **RPMSG_TTY_RTX_M4** project and select **Debug As → CMSIS DS-5 Debugger** to launch the debug configurations dialog:



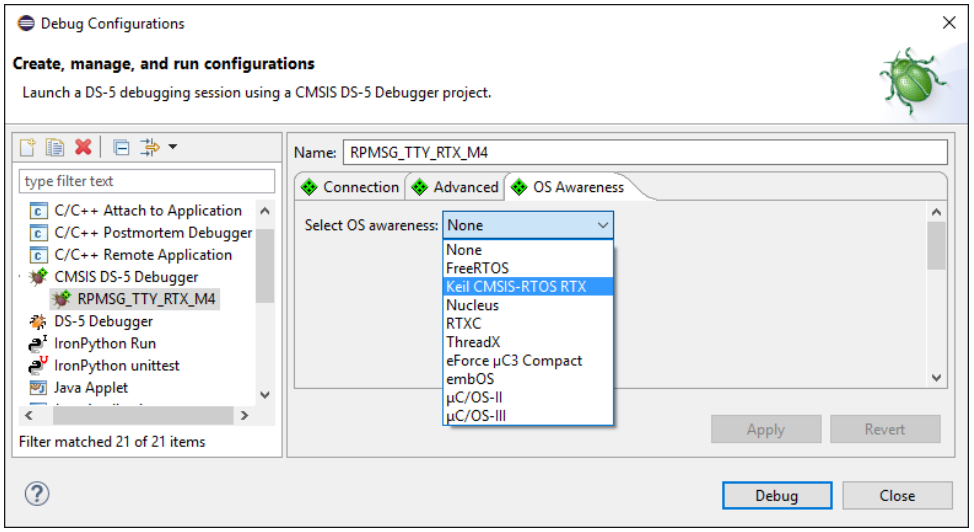
Verify the **Connection Settings** and ensure that ULINK*pro* is correctly detected. If in doubt, use **Browse…** to list available debug adapters.

☞ Click on **Target Configuration…** to setup the Debug and Trace Services Layer (DTSL).



- On the *Cortex-A7* tab, disable all trace options to avoid buffer overflows.
- On the *Cortex-M4* tab, check **Enable Cortex-M4 core trace**.

☞ In the *OS Awareness* tab select the real-time operating system used in your application from the drop-down menu.



Click **Debug**.

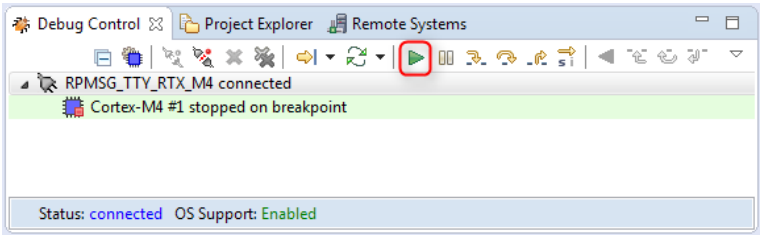
NOTE

*The error message "**Failed to launch debug server**" most likely indicates that an incorrect* ULINKpro *connection address is selected.*

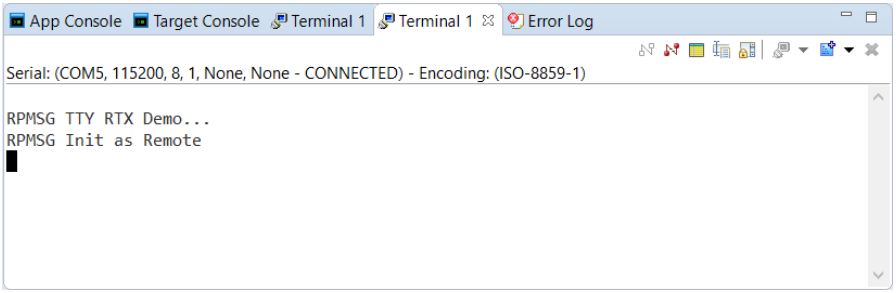
Run Cortex-M application

DS-MDK switches to the DS-5 Debug perspective. The application loads and runs until main.

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



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



NOTE

You can add another Terminal view to the debug perspective by using ***Window → Show View → Terminal****.*

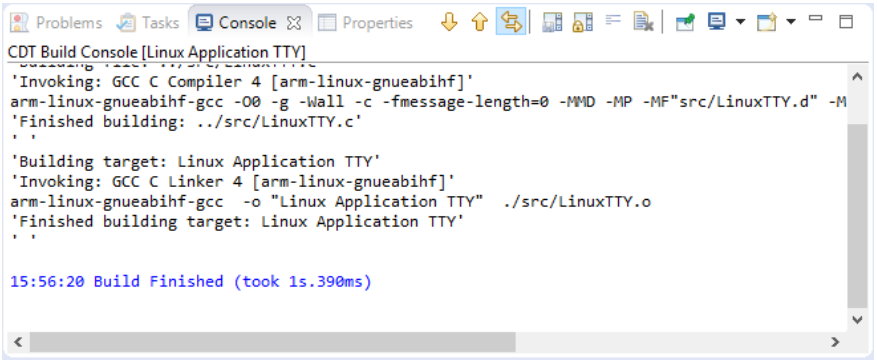
Cortex-A Linux application

Copy and build the Linux Application TTY

Switch back to the *CMSIS Pack Manager* perspective and copy the **Linux Application TTY** example project to your workspace.

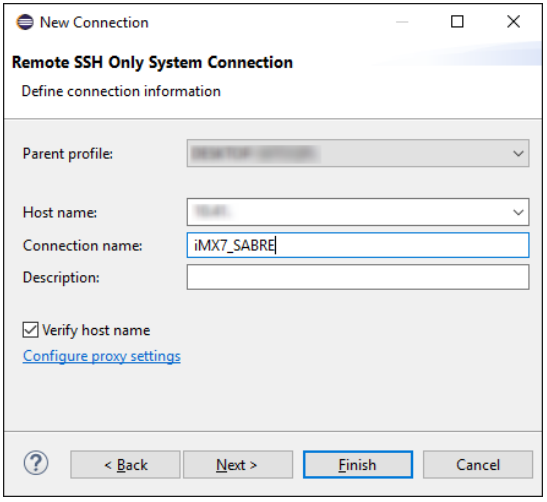
Build the project from the context menu in the **Project Explorer** in the same way we have done for the Cortex-M RPMSG TTY RTX example.

The *Console* should show an error-free build:



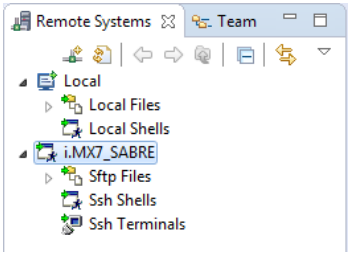
Setup RSE connection

Go to **Window → Open Perspective → Other...**, then select **Remote System Explorer**. Use the button to create a new connection. Select **SSH Only** and click **Next**.



RSE communicates with the target using TCP/IP. Enter the target's IP address into the **Host Name** field. Enter a meaningful name in the **Connection name** box:

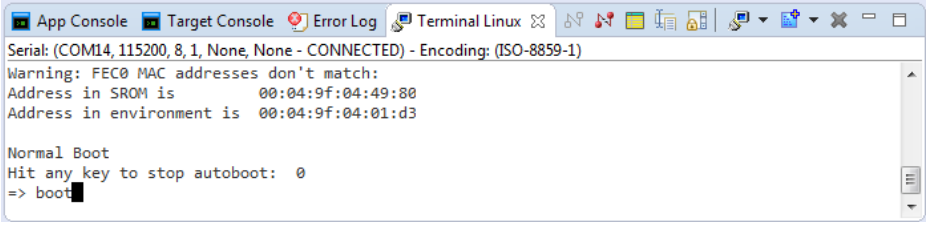
Click **Finish** to show your connection in the **Remote Systems** window:



Boot Linux

NOTE
If you are debugging a microcontroller application simultaneously, you need to run *the Cortex-M application, otherwise the prompt in the Terminal Linux is not accessible.*

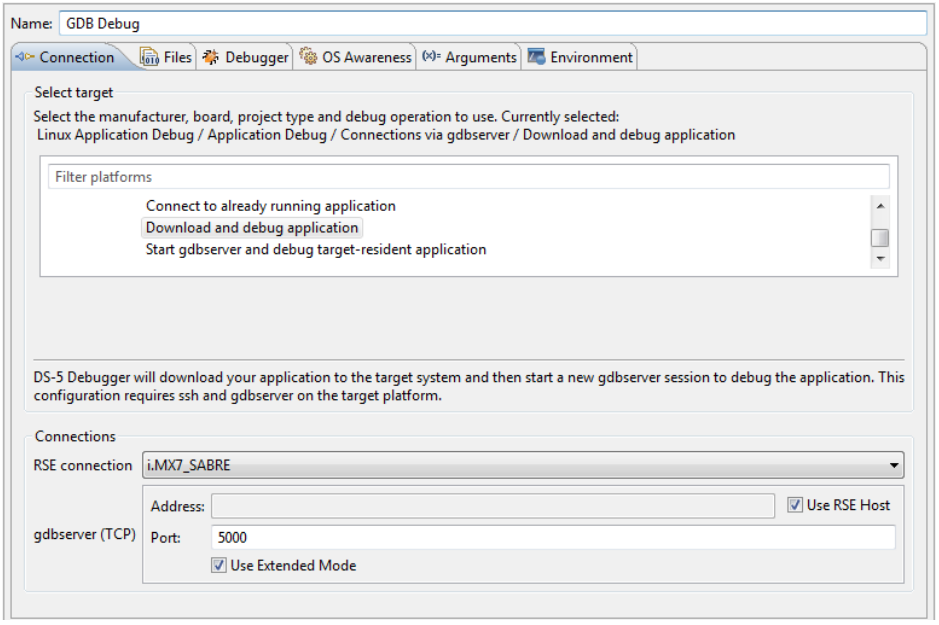
☞ In the *Terminal Linux* enter “boot” to start the Linux system if it hasn’t started yet:



When the boot process has finished, log in as **root** (no password required).

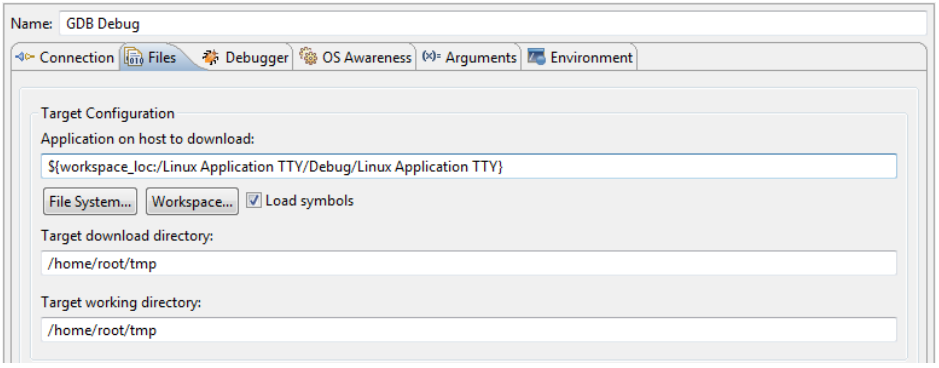
Configure DS-5 debugger

☞ Right-click on the project **Linux Application TTY** and select **Debug As → Debug Configurations…** In the *Debug Configurations* window, select DS-5 Debugger and then press the icon to create a new debug configuration. Name it `GDB Debug` and select in the *Connection* tab **Linux Application Debug → Application Debug → Connections via gdbserver → Download and debug application**. The RSE connection from the previous step shows up:



☞ On the *Files* tab, in **Target Configuration**, select the workspace build target for **Application on host to download**. Select an **existing** directory on the target file system, e.g. `/home/root/tmp` as the **Target download directory**.

Select an **existing** directory on the target file system, e.g. `/home/root/tmp` as the **Target working directory** (use the same directory as for **Target download directory**).



On the *Debugger* tab, under **Run Control** select **Debug from symbol** “main”. Click **Debug**.

If asked for login, please insert the credential for the Linux target. If you are using one of the images downloaded from www.keil.com please use *root* as username and leave the password field empty.

Run the Linux application

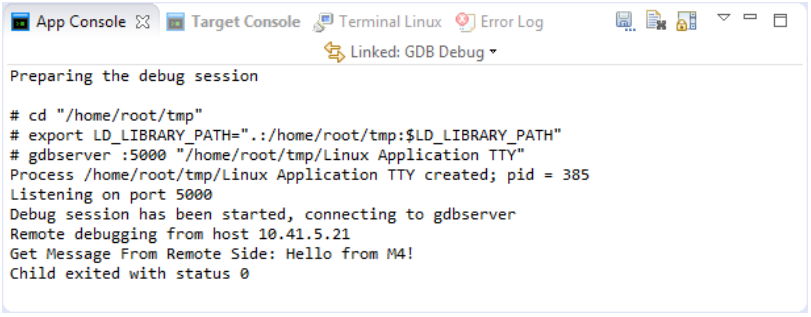
☞ In the *Terminal Linux*, load the kernel module that communicates with the Cortex-M4 application with this command:

```
root@imv7dsabresd:~# modprobe -v imx_rpmsg_tty
```

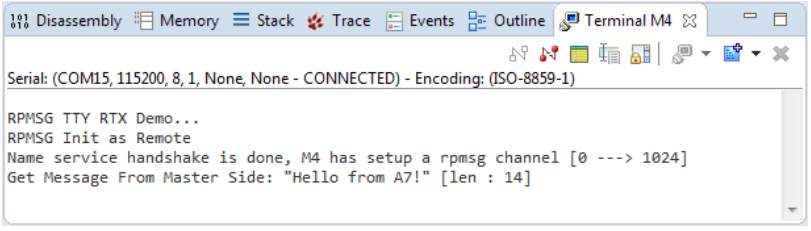
The kernel module will be loaded:

```
insmod /lib/modules/4.1.15-
1.1.0+ga4d2a08/kernel/drivers/rpmsg/imx_rpmsg_tty.ko
imx_rpmsg_tty rpmsg0: new channel: 0x400 -> 0x0!
Install rpmsg tty driver!
```

☞ Use the **Continue** button to run the Linux application. The *App Console* shows the application's messages:



Similarly, the *Terminal M4* shows the output of the microcontroller application:



NOTE

You can add another Terminal view to the Debug perspective by using ***Window → Show View → Terminal****.*

You have verified that your development environment can connect to both the Cortex-M and the Cortex-A processor. Try other example projects such as the **Frequency Bin** that demonstrates how to use the CMSIS-DSP library in the Cortex-M processor. The following chapters will explain how to create projects for both from scratch and how to debug these applications.

Creating projects from scratch

Create Cortex-M applications

This chapter guides you through the steps required to create and modify projects for the Cortex-M target in a heterogeneous system.

Blinky with CMSIS-RTOS RTX

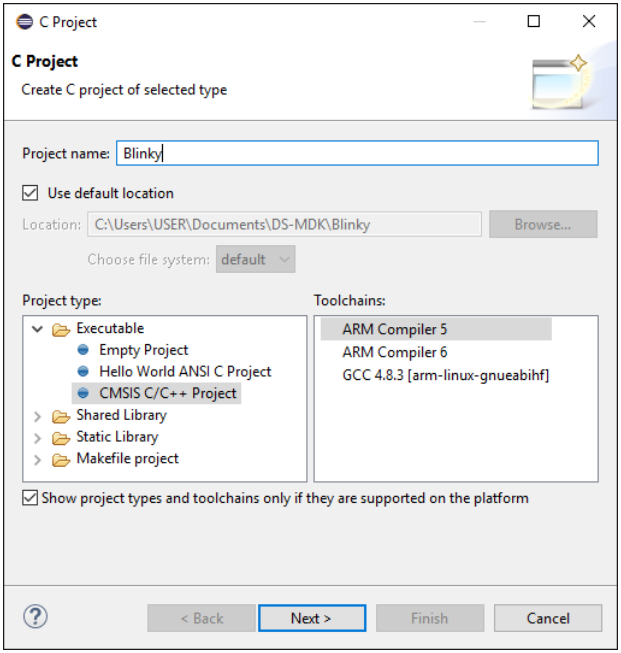
Follow these steps to create a project called **Blinky** using the real-time operating system CMSIS-RTOS RTX:

- **Setup the Project**: create a project and select the microcontroller device along with the relevant CMSIS components.
- **Select Software Components**: choose the required software components for the application.
- **Customize the CMSIS-RTOS RTX Kernel**: adapt the RTOS kernel.
- **Create the Source Code Files**: add and create the application files.
- **Build the Application Image**: compile and link the application.

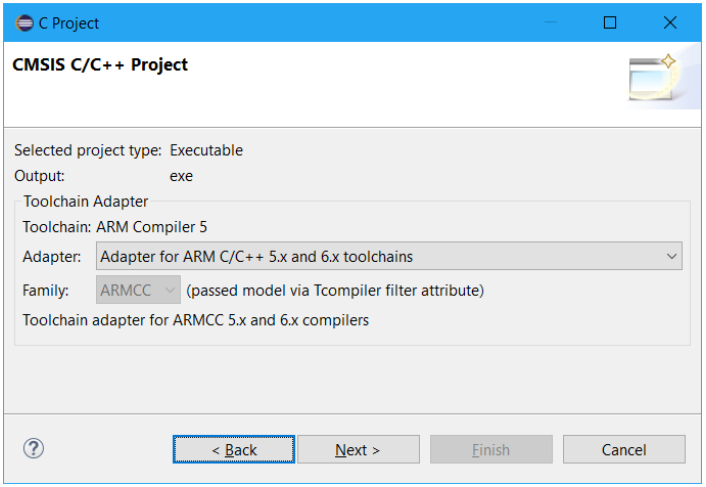
For the **Blinky** project, you will create and modify the *main.c* source file which contains the *main()* function that initializes the RTOS kernel, the peripherals, and starts thread execution. In addition, you will configure the system clock and the CMSIS-RTOS RTX.

Setup the project

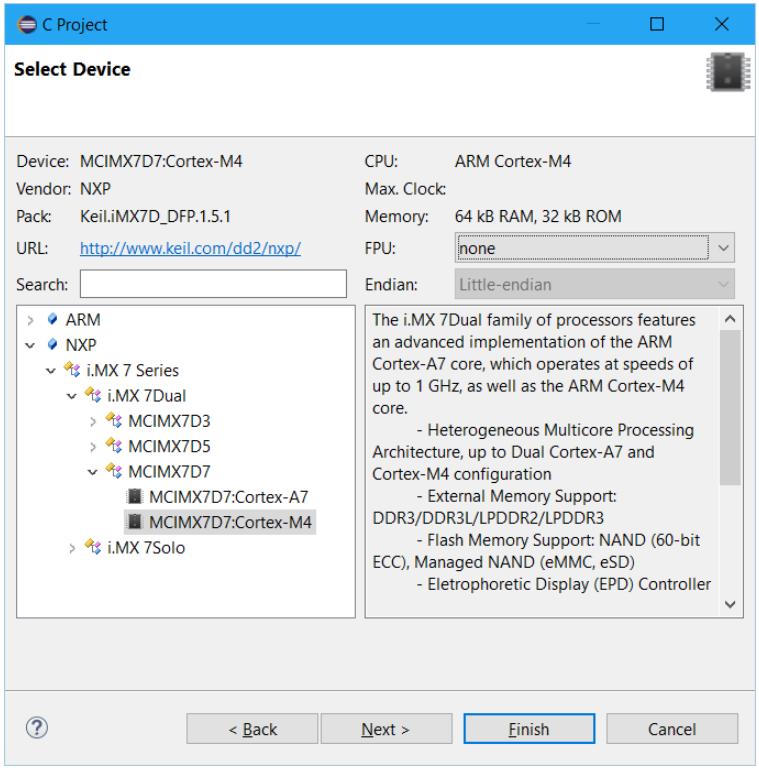
☞ From the Eclipse menu bar, choose **File → New → C Project**:



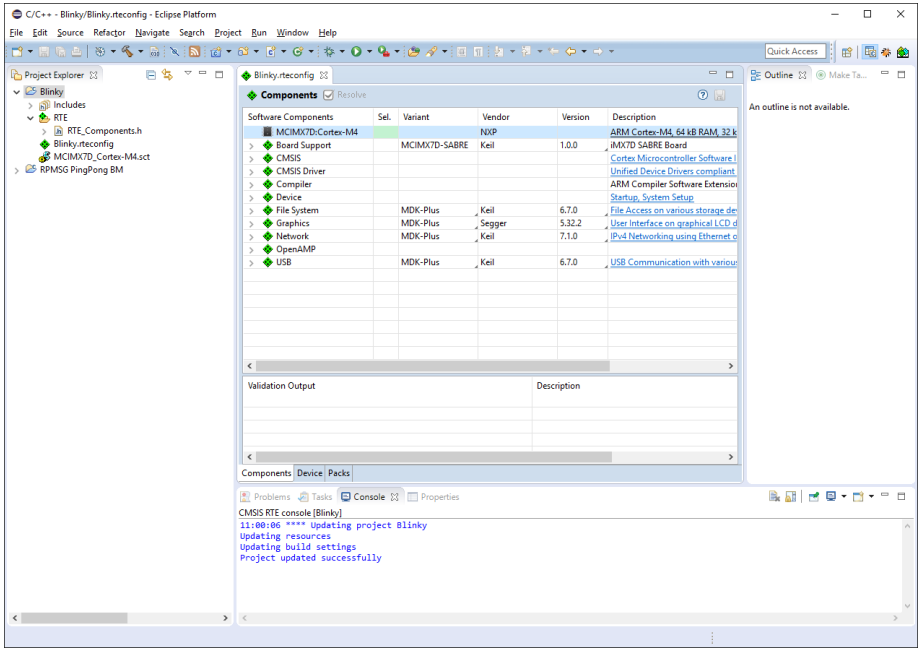
☞ Select **CMSIS RTE C/C++ Project**, enter a project name (for example **Blinky**) and click **Next**.



Select your target device from the list: in this example we would continue using **MCIMX7D7:Cortex-M4**. Make sure the selection on **FPU** is none so that we can avoid initializing it for our example.



Select the **NXP → i.MX 7 Series → i.MX Dual → MCIMX7D7 → MCIMX7D:Cortex-M4** device and click **Finish**.
The *C/C++ Perspective* opens and shows the project:

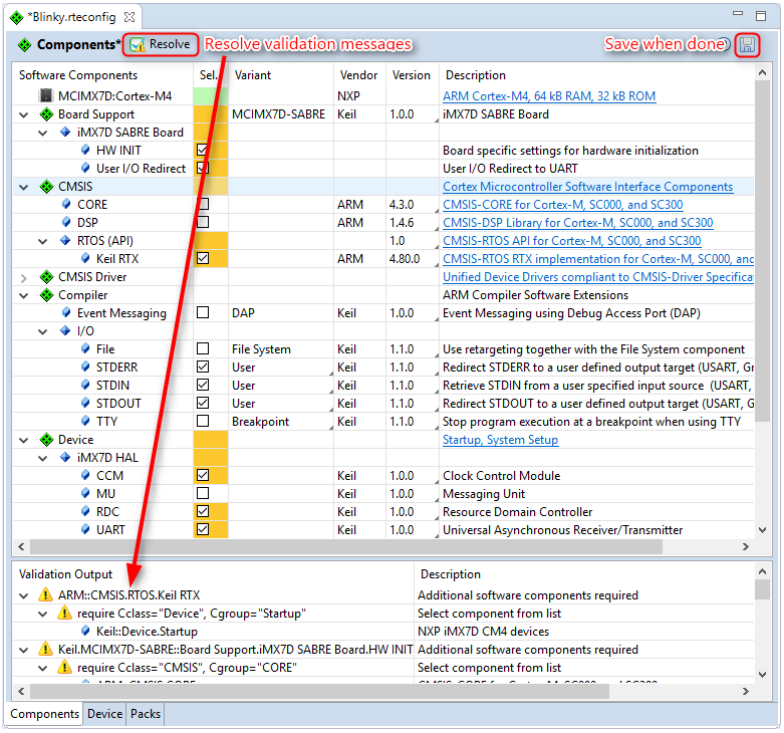


Select software components

☞ For the **Blinky** project based on CMSIS-RTOS RTX, you need to select the following components:

- **Board Support:iMX7D SABRE Board:HW INIT**
- **Board Support:iMX7D SABRE Board:User I/O Redirect**
- **CMSIS:RTOS (API):Keil RTX**.
- **Compiler:I/O:STDERR** configured as variant **User**
- **Compiler:I/O:STDIN** configured as variant **User**
- **Compiler:I/O:STDOUT** configured as variant **User**
- **Compiler:I/O:TTY** configured as variant **User**
- **Device:i.MX7D HAL:CCM**
- **Device:i.MX7D HAL:RDC**
- **Device:i.MX7D HAL:UART**

Use the **Resolve** button to add other required components automatically. Finally, s**ave** your selection:



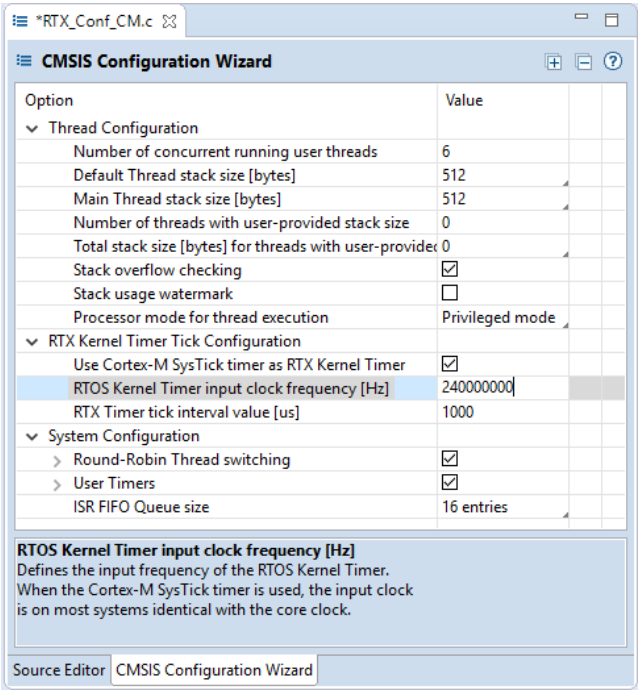
NOTE

Saving the RTE configuration triggers a project update and the selected software components become instantly visible in the Project Explorer.

Configure CMSIS-RTOS RTX kernel

☞ In the project, expand the group **RTE:CMSIS**, right-click on the file *RTX_Conf_CM.c*, and select **Open With → CMSIS Configuration Wizard**. Change the following settings:

- Default Thread stack size [bytes] 512
- Main Thread stack size [bytes] 512
- RTOS Kernel Timer input clock frequency [Hz] 240000000



☞ Save the file using or CTRL+S.

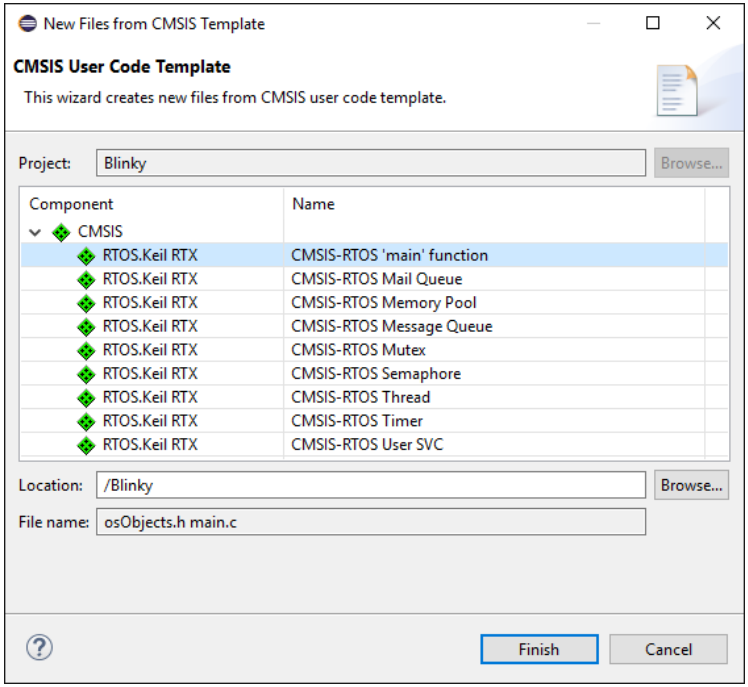
NOTE

If you have opened a file with the CMSIS Configuration Wizard once, your choice is stored and the file will be opened in this view automatically next time.

Create the source code files

Pre-configured user code templates contain routines that resemble the functionality of a software component.

☞ Right-click on the project and select **New → Files from CMSIS Template**.



Expand the software component **CMSIS** and select the template **CMSIS-RTOS 'main' function**. Click **Finish**. Replace the content of *main.c* with the following 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
  }
}
```

Save the file using  or CTRL+S

Adapt the scatter file

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

The following table shows the memories and their load addresses for the different processors:

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

By default, the scatter file template uses the start address 0x0 for the load region command.

- To put the Cortex-M4 code into the TCM of the i.MX 7, open the file *MCIMX7D_Cortex-M4.sct* and change the address of the load region to 0x1FFF8000:

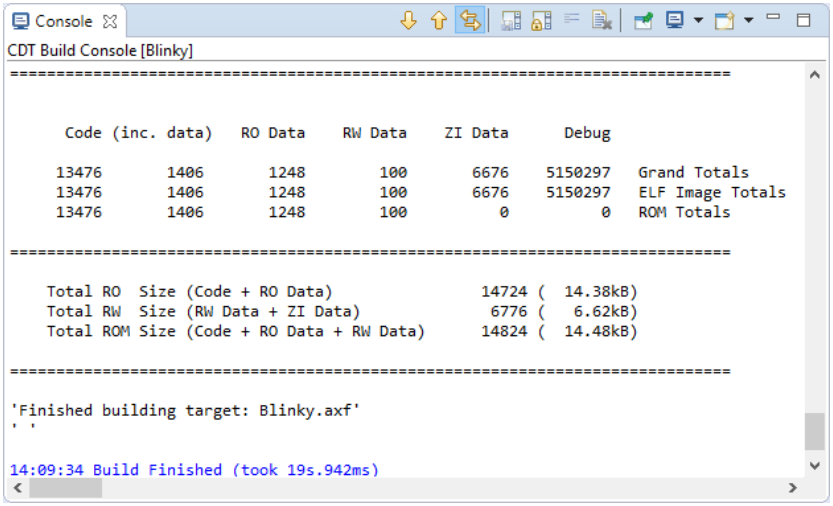
```
;
*******************************************************************
; ** Scatter-Loading Description File generated by RTE CMSIS Plug-in
**
;
*******************************************************************

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

- Save the file using  or CTRL+S.

Build the Cortex-M image

Right-click on the project name and select **Build Project** to build the application.

This step compiles and links all related source files. The *Console* shows information about the build process. An error-free build displays program size information:



Debug Cortex-M application on page 41 guides you through the required steps to connect your evaluation board to the workstation and to debug the application on the target hardware.

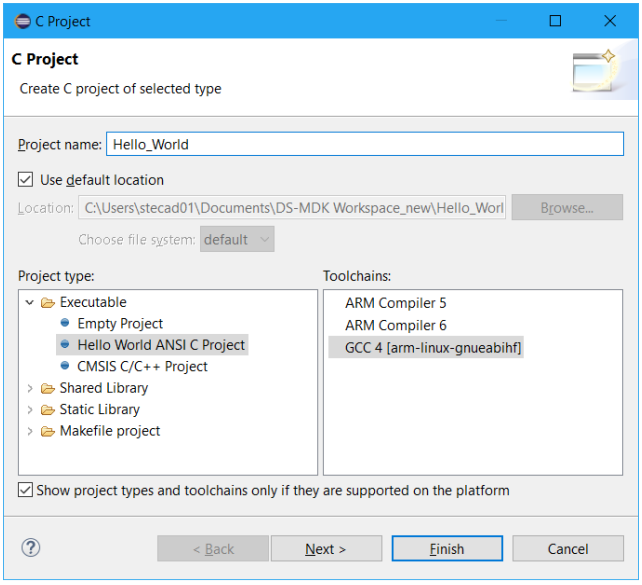
Create Linux applications

This chapter guides you through the steps required to create and modify projects for an ARM Cortex-A class device running Linux:

- **Setup the project**: create a project.
- **Build the application image**: compile and link the application.

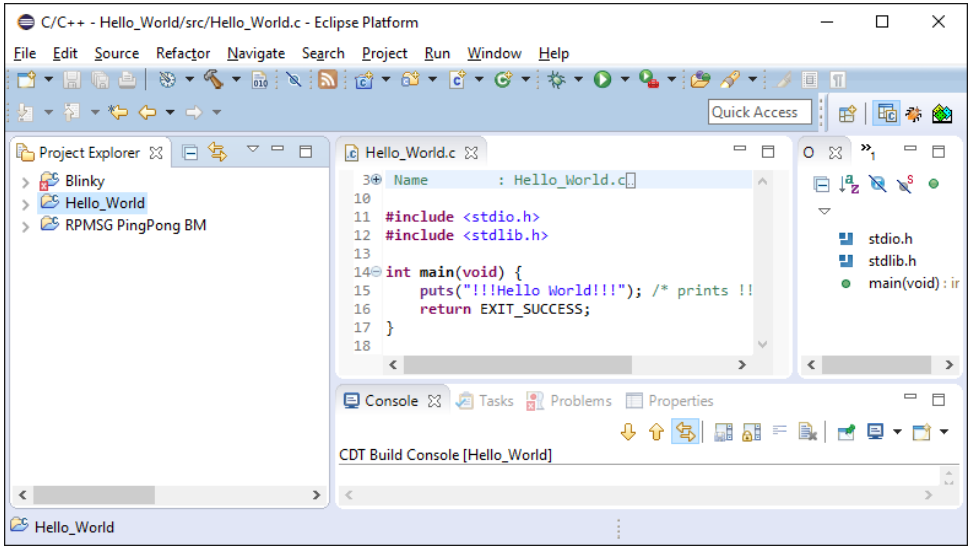
Setup the project

☞ From the Eclipse menu bar, choose **File → New → C Project**. Select the **Hello World ANSI C Project**:



☞ Enter a project name (for example **Hello_World**) and make sure that the **GCC [...] (built-in)** toolchain is selected before clicking **Finish**.

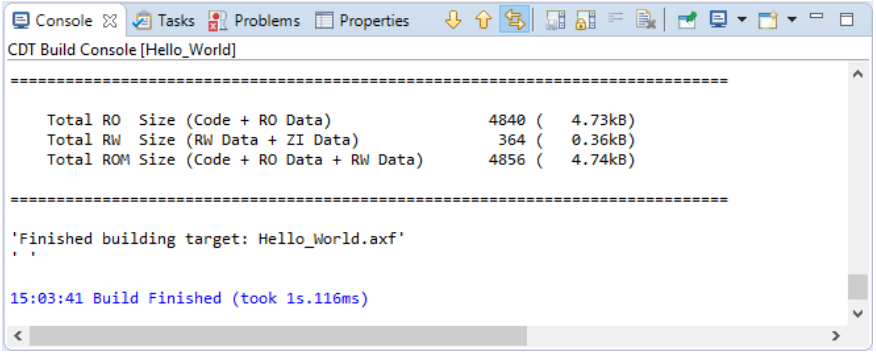
The *C/C++ Perspective* opens and shows the current project:



Build the application image

Right-click on the project name and select **Build Project**.

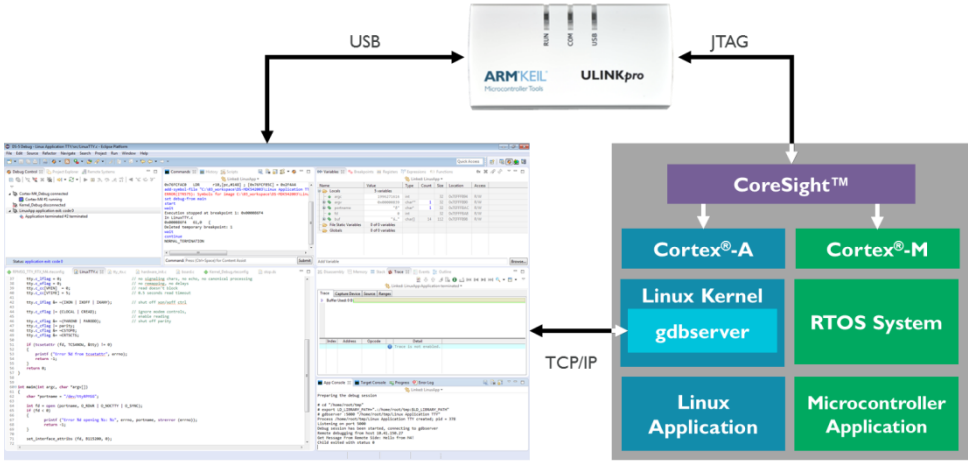
This step compiles and links all related source files. The *Console* shows information about the build process:



The chapter **Debug Linux application** on page 45 guides you through the required steps to connect your evaluation board to the workstation and to download the application to the target hardware.

Debug applications

The DS-5 Debugger can verify all software applications that execute on a heterogeneous computer system. It enables complete system visibility using multiple simultaneous debug connections:



- The **Cortex-M application** is debugged using a ULINK*pro* debug unit (refer to **www.keil.com/ulink** for more information). Users can analyze the microcontroller application with RTOS aware-debugging and peripheral views.
- The **Linux kernel** and **bare metal** applications running on the Cortex-A are also debugged using a ULINK*pro* debug unit. The debugger lists kernel threads and processes.
- A **Linux application** is debugged via **gdbserver** across a TCP/IP network link. The debugger supports multi-threaded application debugging and shows pending breakpoints on loadable modules and shared libraries.

Debug Cortex-M application

This section explains how to debug the microcontroller application running on the Cortex-M microcontroller. Once configured the debug configuration as shown in section *Configure CMSIS DS-5 debugger* at page 20, you can start the debugging session by clicking “Run” in the Debug Control view.

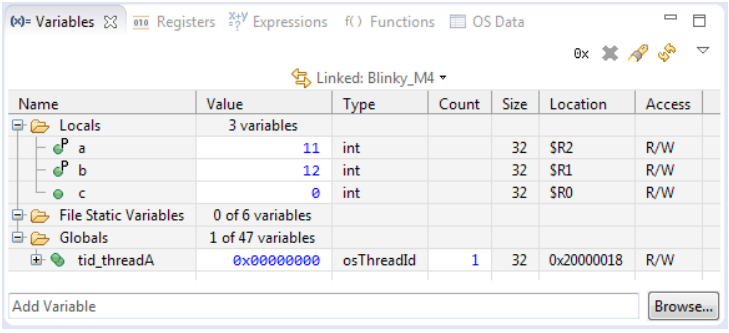
If specified in the configuration window, the debugger will run till the beginning of the function *main()*.

DS-MDK should automatically switch to the **Debug Perspective**, specifically designed to be used during the debug session on your device.

Let’s look at some of the Views available in DS-MDK.

Variables

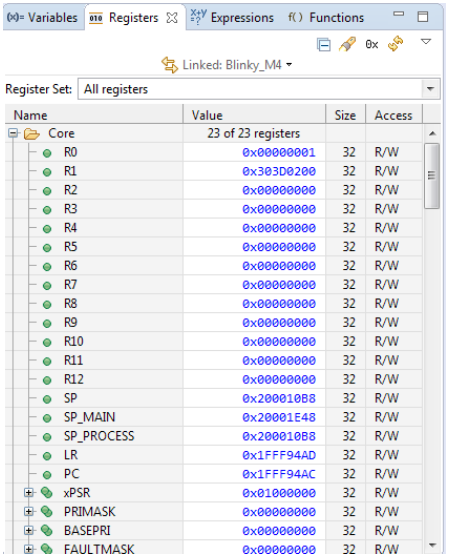
The Variables view shows the contents of local, file static, and global variables in your program. By default, the Variables view displays all the local variables. It also displays the file static and global variable folder nodes.



If you know the name of the specific variable you want to view, enter the variable name in the Add Variable field. This lists the variables that match the text you entered. Double-click the variable to add it to the Variables view.

Registers

The **Registers** view displays the contents of processor and peripheral registers available on your target and allows modifying them.

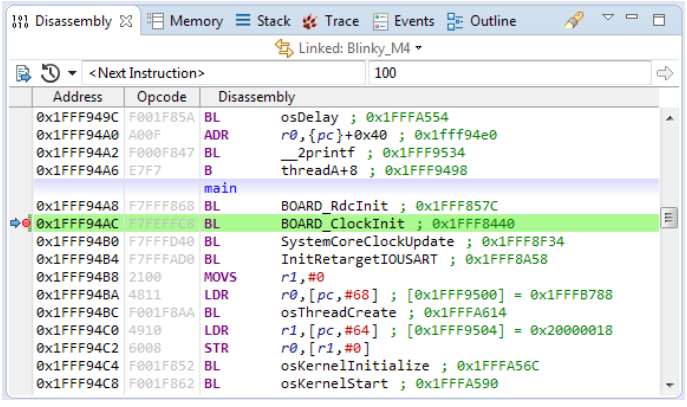


The search button at the top of the View allows searching for register by name to speed up debugging in targets with hundreds or thousands of different registers.

Disassembly

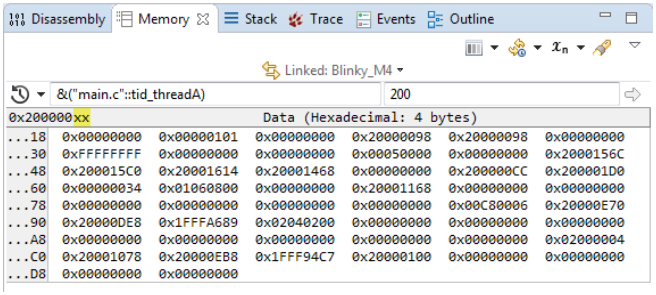
The **Disassembly** view gives you a glimpse over the assembly code running on the device. When the target is stopped, DS-MDK automatically highlights the next instruction to be executed (content of the Program Counter).

The view shows the address, the OpCode and the decoded version of each instruction and can be used, as an example, to debug issues related to invalid addresses.



Memory view

In order to display and to modify the contents of memory it's possible to use the Memory view. You can specify the start address of the memory range, either as an absolute address or as an expression, for example $pc+256. The size of the memory range to display, in bytes, is the offset value from the start address.



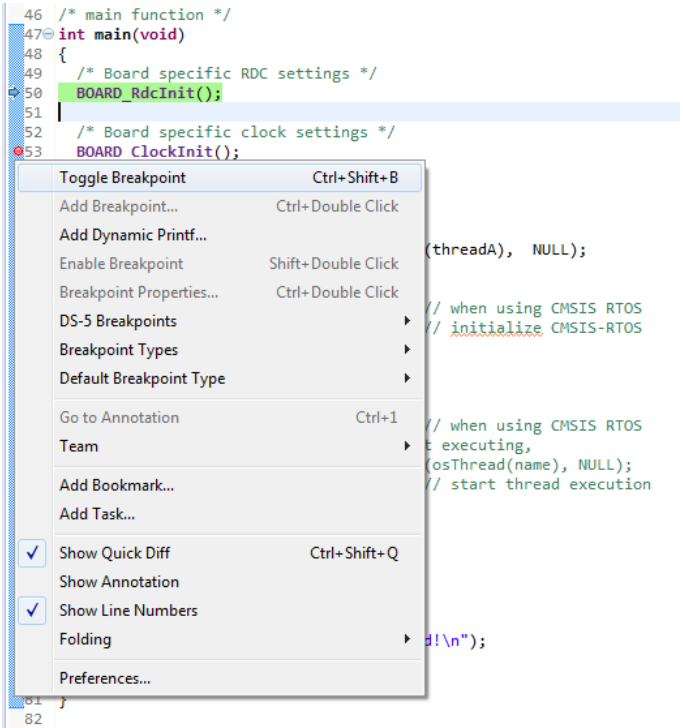
The memory view allows specifying both the address and the size as a formula. A few examples:

- &("main.c"::tid_threadA) refers to the address of variable tid_threadA in file main.c
- $PC refers to the value contained by the register PC
- sizeof(float) refers to the size of the type "float"

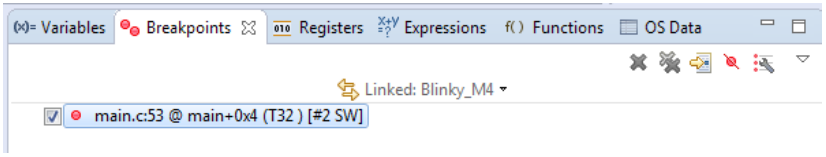
Please refer to the online manual for further options.

Breakpoints

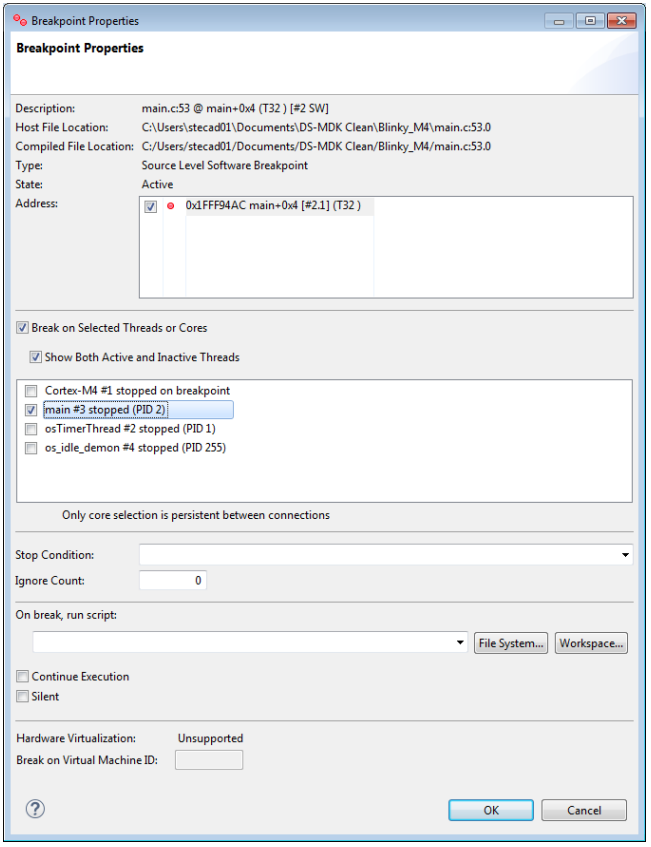
Breakpoints can be set either directly on the source code editor or in the **Breakpoints** view. In the source code editor, right-click on the left side on the line you would like the execution to stop and click on **Toggle Breakpoint**.



The breakpoint will appear in the list in the **Breakpoints** view where it can edited, disabled or removed.



It is possible to access to the properties of the breakpoint by right-click on the breakpoint and then select **Properties**. The Properties window, showed below, allows using some of the advanced functionalities of the DS-MDK debugger such as Thread specific breakpoint, advanced Stop conditions and the ability to run scripts when the program stops.



Please refer to the online help for a detailed explanation of all the functionalities accessible from the **Properties** window.

Debug Linux application

This section explains how to debug a Linux application running on the Cortex-A7.

The DS-5 Debugger uses *gdbserver* for debugging Linux on the target hardware. Before connecting, you must:

- Set up the target with Linux installed and booted. Refer to **Install the Linux image** on page 13.
- Obtain the target IP address or name for the connection between the debugger and the debug hardware adapter. If the target is in your local subnet, click Browse and select your target.

Next, set up a Remote Systems Explorer (RSE) connection to the target to download the application onto the target's file system. Refer to **Setup RSE connection** on page 23 for more information.

Configure the debugger as described in *Configure DS-5 debugger* at page 24 and launch the application.

DS-MDK uses the same debug perspective as for bare metal when debugging Linux application so you do not need to learn a new environment or set of Views in order to start debugging.

Debug the Linux Kernel

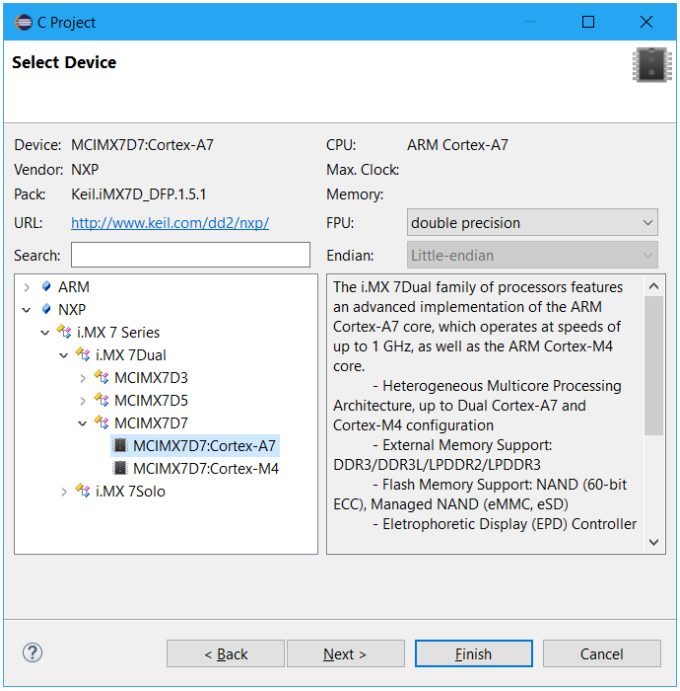
The DS-5 Debugger configuration dialog makes it easy to configure a debugging session to a specific target. The Linux kernel debug configuration type is primarily designed for post-MMU debug to provide full kernel awareness but – with some extra controls – can also be used for pre-MMU debug. This makes it possible to debug the Linux kernel, all the way from its entry point, through the pre-MMU stages, and then seamlessly through the MMU enable stage to post-MMU debug with full kernel awareness. You can do this all with source-level symbols, and without the need for tedious disconnecting, reconfiguring and reconnecting!

The Linux kernel, already built with debug info and a complete *vmlinux* symbol file, file system, and full source code, is available from the respective board support pages (see **www.keil.com/mdk5/ds-mdk/install#boards**).

Unpack the Linux kernel sources (*kernel-source.tar.gz*) into your currently active DS-MDK Eclipse workspace. Be aware that on a Windows system you will not be able to fully unpack the sources. Some symbolic links and case-sensitive source files will not be created. Usually, this is not critical for Linux kernel debug.

Create a Linux Kernel debug project

☞ Create a new CMSIS C/C++ Project named **Linux Kernel Debug** and select NXP i.MX7Dual device *MCIMX7D7:Cortex-A7*.



Add the *vmlinux* file to the project folder using Windows Explorer. This file must match the kernel in the SD card on the board.

NOTE

The debug symbols in the vmlinux file have virtual addresses, so the usage of vmlinux file by the debugger assumes that the OS is up and running with the MMU enabled. It still can be used to debug pre MMU at source-level if there is no offset between physical and virtual addresses at the entry point.

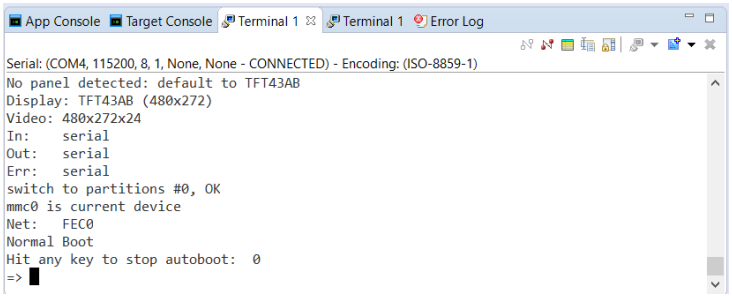
☞ Add a debugger script to the project (right-click the project and select **New → Other… → DS-5 Debugger → DS 5 Debugger Script**) called **stop.ds** containing:

```
stop
set os enabled off
```

When OS awareness is enabled and kernel symbols are loaded from the vmlinux file, DS-5 Debugger will try to read some kernel structures. If the MMU is not yet on, the debugger may try to access invalid addresses, leading to data aborts, which is undesirable. This OS awareness support feature can be temporarily

disabled during the pre-MMU debug stage with the CLI command `set os enabled off`, and later (post-MMU) re-enabled with the CLI command `set os enabled on`.

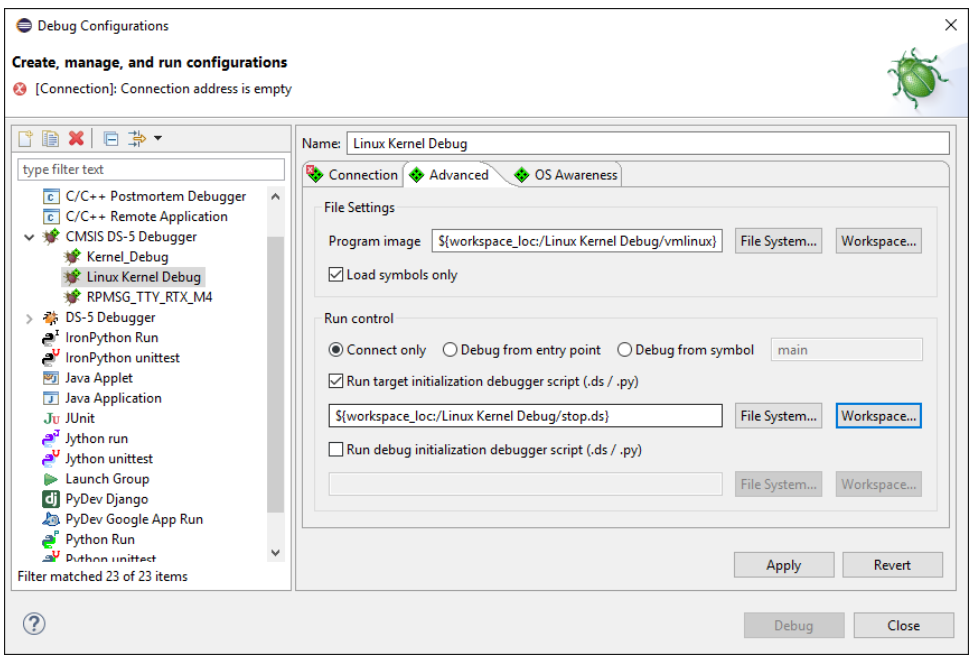
☞ Restart the board and make sure you stop the boot of the Linux kernel by pressing a button when U-Boot is initializing in the *Terminal* view.



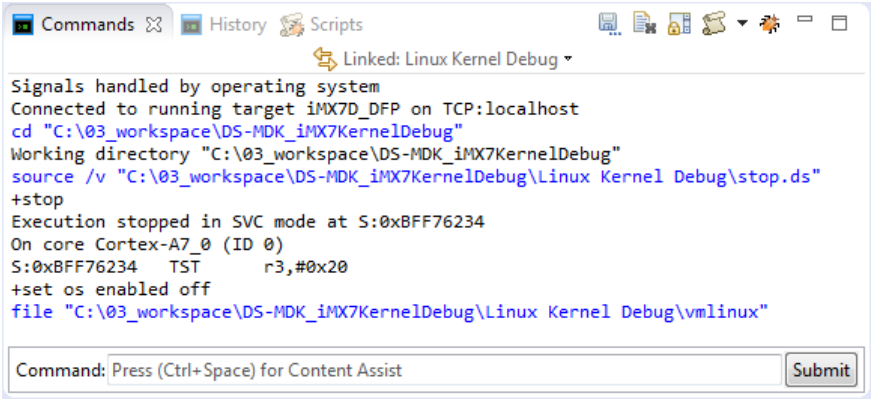
☞ Right-click on the project, select **Debug As…**, then select **CMSIS DS-5 Debugger…** to open the **Debug Configurations** dialog.

In the *Connection* tab, select CPU Instance = SMP.

In the *Advanced* tab, tick **Run target initialization debugger script**, and select the **stop.ds** script in the workspace:



☞ Click Debug. The *Commands* view will show:



- In the Command (CLI) entry box, set a temporary hardware breakpoint (`thbreak`) on the entry point into the kernel, by typing in:

```
thbreak 0x80008000
```

  Press the Submit button or the Enter key. `0x80008000` is the entry point for the kernel. This is the address to which U-Boot will pass control to boot Linux once it has completed its setup tasks.

- Run the target by pressing the **Continue** button (▶) in the *Debug Control* view, or press **F8**.

- In the Terminal view, tell U-Boot to boot the kernel, by typing in:

```
boot
```

Code execution will stop at the breakpoint, and the Disassembly view will show the assembly code at the entry point (labeled `stext`). If you have unpacked your kernel source code into the workspace, the Editor view will show the content of `head.S`.

If not, no source code is shown, because the path to the source code has not yet been configured. DS-5 Debugger will try to open `.../arch/arm/kernel/head.S` in its Editor view. If it does not find the kernel sources using the source paths within the vmlinux file, you can resolve this by setting a substitute source path, to re-direct paths from where the kernel was built, for example, from:

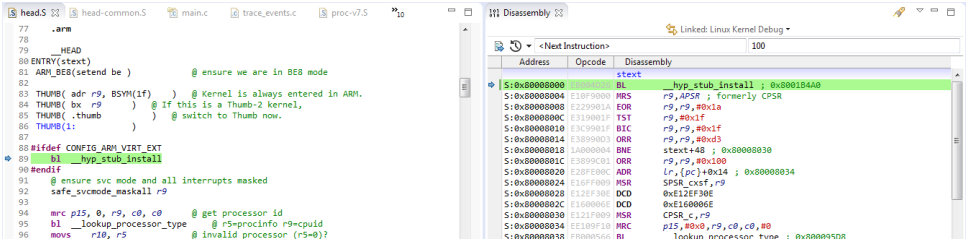
```
/home/munlin01/fsl-community-bsp-platform/build-core-image-
base/tmp/work-shared/imx7dsabresd/kernel-source
```

to a local copy of the kernel sources at:

```
C:\path\to\linux-imx\4.1.15-r0\git
```

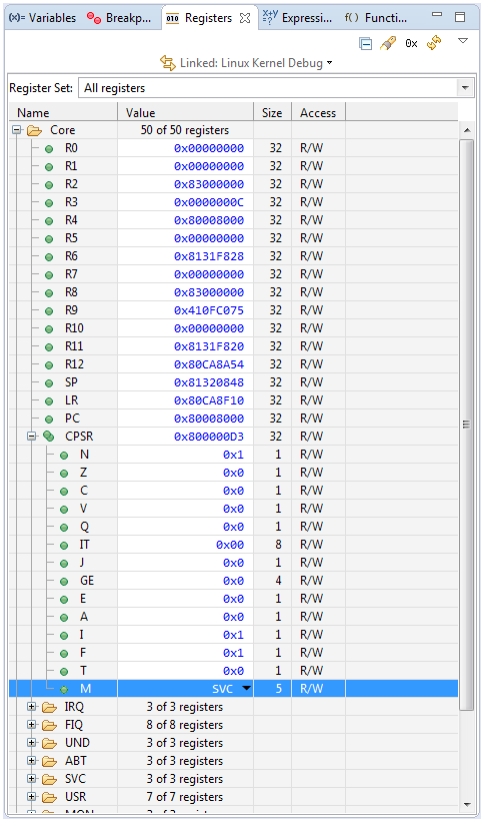
Make sure that the "Image Path" and "Host Path" both end with a corresponding directory.

`head.S` will now open in the *Editor* view, and the *Disassembly* view will show the symbol `stext`, at the entry point for the kernel. If it doesn't, choose the **Path Substitution…** command from the *Debug Control* view's drop-down menu (▽) and check that the final directory in the Image Path and Host Path correspond. Then right-click on an instruction in the *Disassembly* view, and select "Show in Source".



Debug the Kernel: Pre-MMU stage

You can now set breakpoints and watchpoints, view registers, view memory, single-step, and other usual debug operations at this pre-MMU stage, all with source level symbols.



☞ At the kernel entry point, you can check the Core and CP15 system registers in the Registers view to check that they are set as recommended by kernel.org. Observe that:

a. the CPU is in SVC (supervisor) mode; check **Core → CPSR → M → SVC**

b. **R0** is 0

c. **R2** contains a pointer to the device tree. Right-click R2 and select **Show Memory Pointed To By R2**. Change the size of the memory displayed to 200 bytes for example by entering 200 in the text entry box in the top right of the *Memory* view.

d. the MMU is off; check **CP15 → SecureBanked → S_SCTLR → M**

e. the Data cache is off; check **CP15 → SecureBanked → S_SCTLR → C**

f. the Instruction cache is either on or off; check **CP15 → SecureBanked → S_SCTLR → I**

☞ To see when the MMU will be turned on, set a breakpoint:

```
thbreak __turn_mmu_on
```

then continue running (or press F8). 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 will jump to after the MMU is enabled.

☞ 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
```

then continue running (press **F8**). When the breakpoint at `__mmap_switched` is hit, the MMU is on.

- Check that the MMU is now on, by looking in the Registers view at **CP15 → SecureBanked → S_SCTLR → M** (should show `Enable`).

Debug the Kernel: post-MMU stage

The main C code entry into the kernel, after all the architecture-specific setup has been done, is `start_kernel()` in `\source\init\main.c`.

- Set a breakpoint on it:

```
thbreak start_kernel
```

and then run to it.

- You can now safely enable OS support in DS-5 Debugger:

```
set os enabled on
```

- Check that the following appears in the Command view, to confirm Linux kernel support is enabled:

```
Enabled Linux kernel support for version "Linux 4.1.15-
1.1.0+ga4d2a08 #2 SMP PREEMPT Tue Jul 5 09:51:28 CEST 2016
arm"
```

- The same Linux version information can be reported manually using:

```
info os-version
```

which will show for example:

```
Operating system on: Linux 4.1.15-1.1.0+ga4d2a08 #2 SMP
PREEMPT Tue Jul 5 09:51:28 CEST 2016 arm
```

This is similar to:

```
output init_nsproxy.uts_ns->name
```

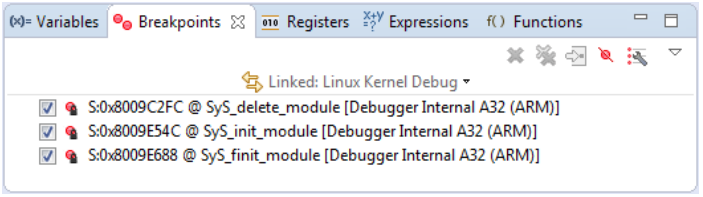
which will show for example:

```
{sysname = "Linux", nodename = "(none)", release = "4.1.15-
1.1.0+ga4d2a08", version = "#2 SMP PREEMPT Tue Jul 5 09:51:28
CEST 2016", machine = "armv7l", domainname = "(none)"}
```

This may take a few moments to display, because DS-5 Debugger has to process the debug symbols.

When OS awareness is enabled and kernel symbols are loaded from the vmlinux file, DS-5 Debugger will try to access some locations in the kernel. For example, it will try to read `init_nsproxy.uts_ns->name` to get the kernel name and

version. It will also set breakpoints automatically on `SyS_init_module()` and `SyS_delete_module()` to trap when kernel modules are inserted (`insmod`) and removed (`rmmod`). You will see these breakpoints appearing in the *Breakpoints* view:

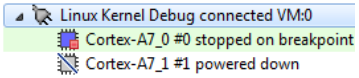


Set a breakpoint with:

```
thbreak kernel_init
```

then run to it.

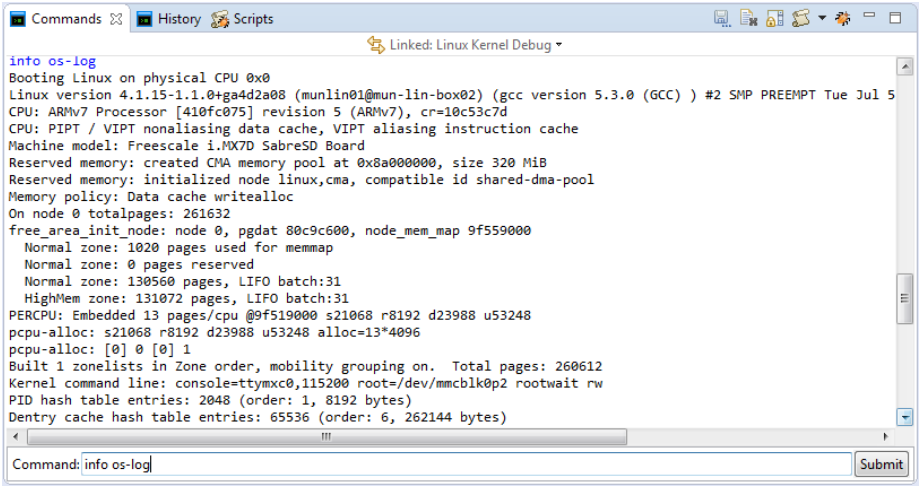
So far, CPU 0 has been doing all the work. Note that CPU 1 is still powered down:



A very useful feature during kernel bring-up is to display early printk output in DS-5 Debugger's command window.

Before the console has been enabled there will be no output from the serial port. You can view the entire log so far with:

```
info os-log
```



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 available in the *Editor* view). The init process now appears as an active thread. CPU 1 is now powered up.

Many of the above steps can be automated, either with a script file, or by filling-in the Debug Configuration's fields before launching (refer to the Appendix).

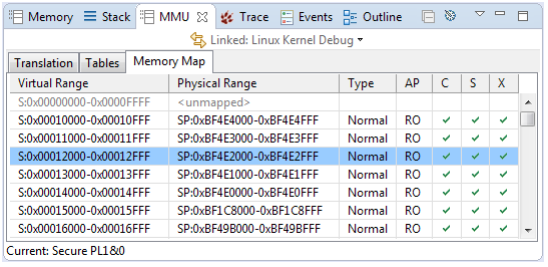
- Delete all user breakpoints and continue (**F8**). Let the kernel run all the way to the Login prompt. Login as root.

- Stop the target by pressing Interrupt (/F9). In the *Debug Control* view, expand "Active Threads" and "All Threads". In "All Threads", you will see a large number of threads/processes have been created. Only two were actually running, one on each of the two cores. You can see these in "Active Threads".

  Right-click on the connection and select Display Cores to see the state of both CPUs. You can view the state of the cores, threads and processes on the command-line with:

```
info cores
info threads
info processes
```

- It is possible to single-step a core or a thread/process. To do so, select either the core or the thread/process in the *Debug Control* view, then press Step (/F5). Note that when single-stepping though a process, it might get migrated to another core. If a breakpoint is set on a process, the debugger is able to track the migration of process-specific breakpoints to the other core.



- You can check the virtual-to-physical address map for Linux by using the MMU view. Continue to run the target (**F8**). Go to **Window → Show View → MMU**. Switch to the Memory Map tab and press the Show Memory Map button to refresh the values.

- Let's take a look at the kernel's `thread_info` structure. Stop the target, then check the kernel's stack size with:

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

For this ARMv7 kernel, the kernel stack size is **8K**.

In the Expressions view, add a new expression into the field (type in the field at the bottom on the view):

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

0x1FFF is 8K minus 1. Expand the tree structure to explore its contents. 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`.

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

```
hbreak __schedule
```

NOTE
This time use hbreak to have a persistent hardware breakpoint instead of a temporary one.

Then continue running (press **F8**). At the breakpoint, continue running (press **F8**) again and again, and see the names of the active threads changing in "Active Threads", and different threads are scheduled-in.

- Alternatively, instead of setting a breakpoint on `__schedule()`, try to set a breakpoint on `do_fork()`. If nothing forks, force a fork by typing e.g. '`ls`'.

In summary, we have looked at how DS-MDK can be used to debug the Linux SMP kernel, both in pre-MMU enabled and post-MMU enabled stages, and looked at a few of the kernel's internal features.

Debug a Linux Kernel module

Only a few things are required to make kernel module debugging work. This sections explains how to do this for the *imx_rpmsg_tty* module that is used in the example projects that are explained in detail on page **Error! Bookmark not defined.**.

Create a Linux Kernel module debug project

- Create a new CMSIS C/C++ Project named **Linux Kernel Module Debug**

  As with the Linux kernel debug, add the *vmlinux* file to the project folder using Windows Explorer.

- Add a debugger script to the project (right-click the project and select **New → Other… → DS-5 Debugger → DS 5 Debugger Script**) called **stop.ds** containing:

```
stop
```

☞ Add another debugger script to the project (right-click the project and select **New → Other… → DS-5 Debugger → DS 5 Debugger Script**) called **load_ko.ds** containing:

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

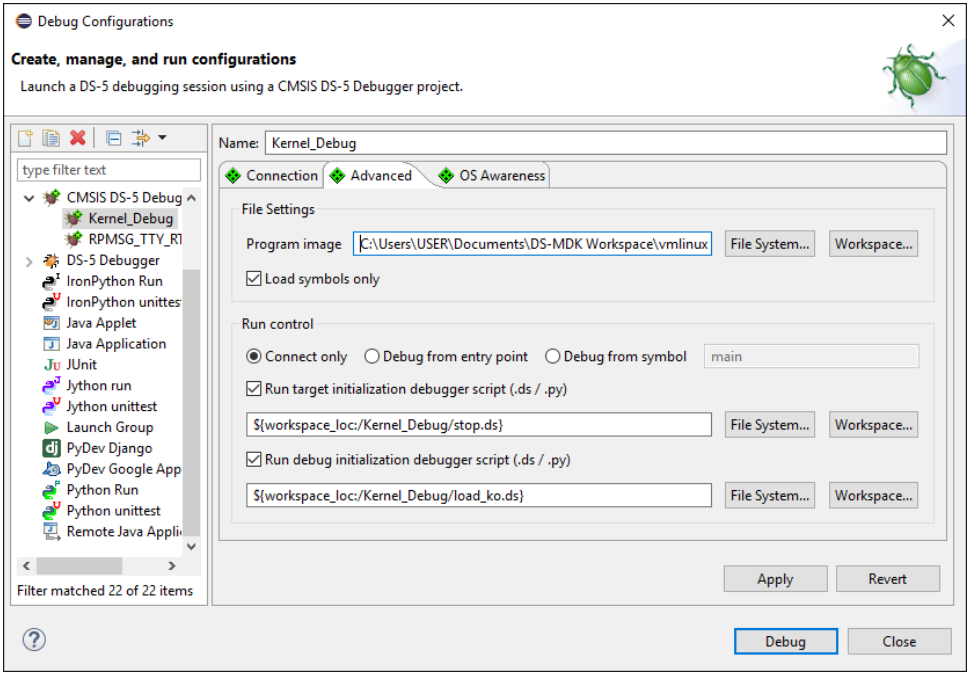
NOTE

Make sure that the file imx_rpmsg_tty.ko *is stored in the workspace so that DS-MDK can find it. Otherwise, specify the fully qualified 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 will halt the processor before loading the kernel symbols and the `add-symbol-file` command will load the kernel module object file.

☞ Right-click the project and select **Debug As → CMSIS DS-5 Debugger...**

On the *Connections* tab, set the CPU Instance to either 0 or SMP. Go to the *Advanced* tab and specify the path to the *vmlinux* file and enable **Load symbols only**. Also, set the initialization debugger scripts as shown here:



Apply the settings and press **Close** (do not press Debug yet!).

Debug the Kernel module

The following steps are required to come to a point where you can debug the kernel module:

Restart your target and halt in U-Boot.

Debug and run the Cortex-M4 application RPMSG TTY RTX.

Boot Linux.

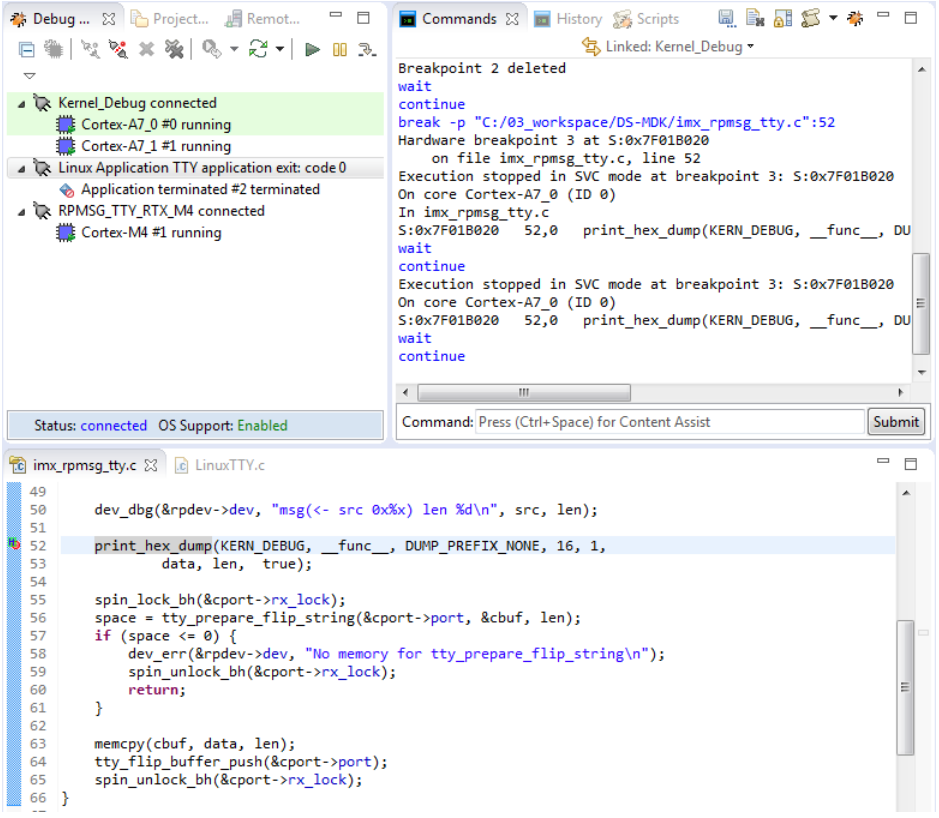
At the Linux prompt, issue the following command to install the driver for the kernel module:

```
modprobe imx_rpmsg_tty
```

Debug and run the Kernel_Debug project.

Now, you can open the *imx_rpmsg_tty.c* and set breakpoints.

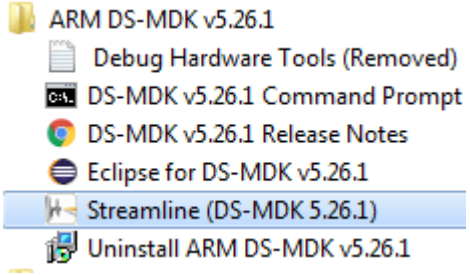
Finally, debug the Linux Application TTY as well (make sure that the RSE connection is still live). When you run the application, the debugger will stop at the breakpoint you have set in the previous step.



ARM Streamline

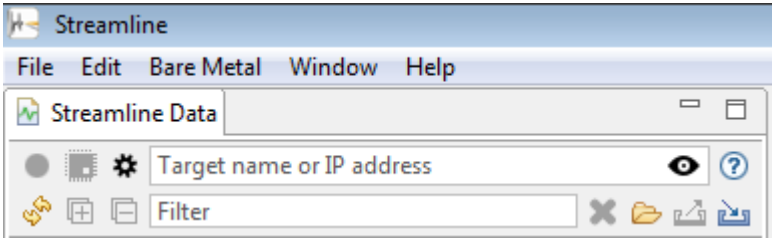
ARM Streamline performance analyzer gives you the ability to collect performance metrics, software tracing and statistical profiling from your Linux system and show that in its innovative user interface. Streamline helps you to identify code hotspots, system bottlenecks and other unintended effects of your code or the system architecture.

DS-MDK includes ARM Streamline in the MDK Professional edition: you can launch Streamline from the ARM DS-MDK Start menu.



Once launched, Streamline allows connecting via TCP/IP to a running Linux target. A target agent (gator) is required to run on the ARM Linux target for ARM Streamline to operate. If you downloaded the Linux image from http://www2.keil.com/mdk5/ds-mdk/install#boards, then gator is already installed so you do not need to rebuild the image.

To start collecting data, you can type the target hostname or IP address in the field box on the top-left side of the window and press the Start Capture button.



The interface would then show the acquired data in graphs which can be used to understand which parts of the code require optimizations or affect the performance of the system considerably.

For extra information on the capabilities of the product, please refer to the user guide available online at https://developer.arm.com/docs/100769/latest/.

Store Cortex-M image

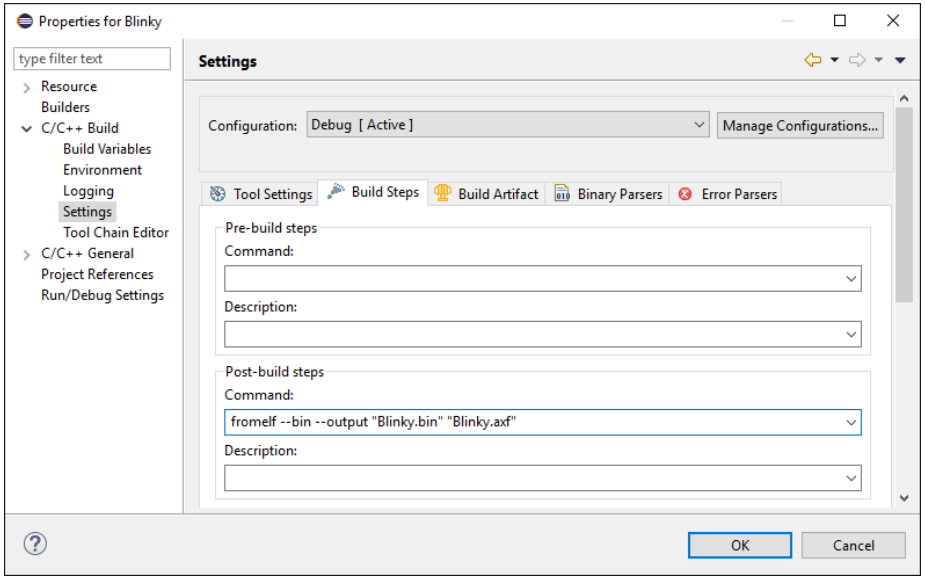
To store the Cortex-M image for execution at start up use the following steps:

1. Create a binary image (BIN) with the `fromelf` utility application.
2. Store this BIN image on SD card in the boot partition
3. Setup the U-Boot environment to start-up the BIN image file.

Create a Cortex-M binary image (BIN)

Right-click the project and select **Properties → C/C++ Build → Settings**. In the the **Build Steps** enter under **Post-build steps** the **Command:**

```
fromelf --bin --output "Blinky.bin" "Blinky.axf"
```



NOTE

This example shows the steps for the Blinky application from section ***Blinky with CMSIS-RTOS RTX*** *on page 28.*

Click OK and rebuild the project to get the BIN file generated.

Store Cortex-M BIN file on SD Card

The SD Card has two partitions:

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

☞ 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
```

☞ Store the Cortex-M binary image in the **FAT32** boot partition to be able to execute it at system startup:

1. Create a sub-directory on the Linux file system, for example:

   ```
   ~# mkdir /media/sd0
   ```

2. Mount the Linux file system partition for access with RSE.

   ```
   ~# mount –t vfat /dev/mmcblk0p1 /media/sd0
   ```

3. Use RSE to copy the BIN file from your workspace to the `/media/sd0` directory.

4. Unmount the partition to ensure that the file is written correctly:

   ```
   ~# umount /media/sd0
   ```

5. Reboot the system and halt in U-Boot.

Run Cortex-M BIN file from U-Boot

At this point, the Cortex-M BIN file is stored in the boot partition.

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
```

Run `m4boot` to start the Blinky application:

```
=> run m4boot
```

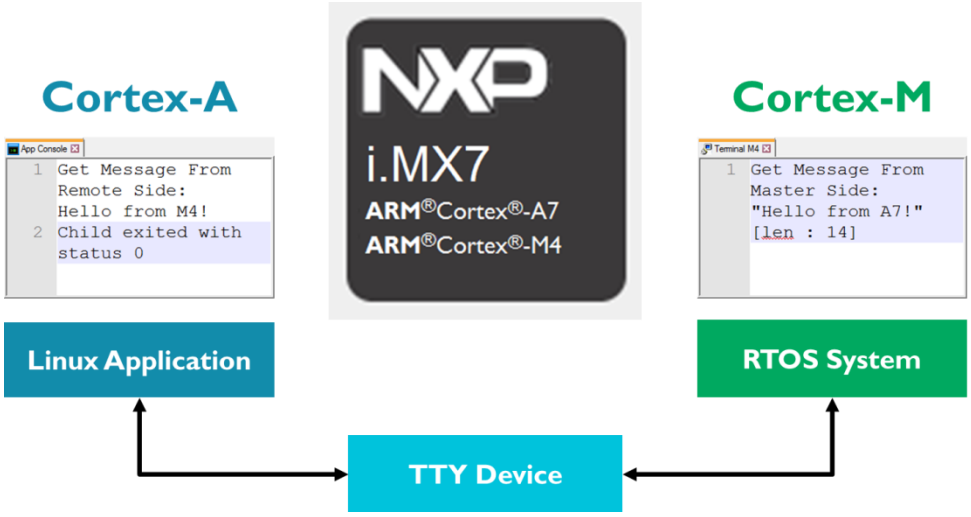
NOTE

For more information refer to the U-Boot Command Line Interface in the U-Boot user's manual (www.denx.de/wiki/DULG/UBoot).

Appendix

Remote Processor Messaging protocol example

The device family packs for NXP's i.MX devices contain two example projects that show how the two processors communicate with each other using the remote processor messaging protocol (RPMSG) via a TTY serial device.



The *Linux Application TTY* runs on the Cortex-A processor and writes a message to a TTY device. The terminal of the *RPMSG TTY RTX* application running on the Cortex-M processor shows this message. The application itself responds on the TTY device. The Linux application reads this message and shows it in its **App Console**.

Eclipse IDE

DS-MDK is an Integrated Development Environment (IDE) that combines the Eclipse IDE with the compilation and debug technology of ARM.

Use DS-MDK as a project manager to create, build, debug, monitor, and manage projects for ARM targets. It uses a single folder called a workspace to store files and folders related to specific projects.

Users can extend its abilities by installing plug-ins written for the Eclipse platform, such as the **CMSIS Pack Manager** and **Remote System Explorer**, included in DS-MDK.

Perspectives

DS-MDK have multiple perspectives: each perspective contains an initial set and layout of views that help you to create, build and debug projects. While working with DS-MDK, you will switch perspectives frequently. It is always possible to change a perspective layout and to add new views to it.

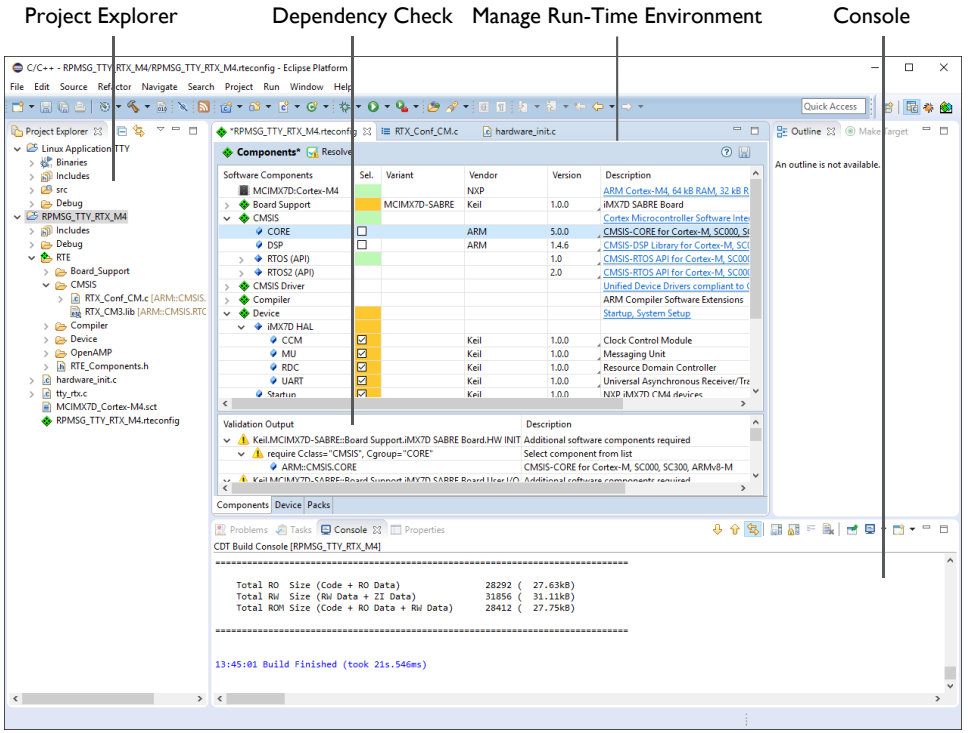
DS-MDK uses mainly these perspectives:

- **C/C++ Perspective**
- **CMSIS Pack Manager Perspective**
- **Remote System Explorer Perspective**
- **DS-5 Debug Perspective**

C/C++ perspective

By default, this perspective consists of the Project Explorer, an editor area and views for tasks, properties, and a message console.

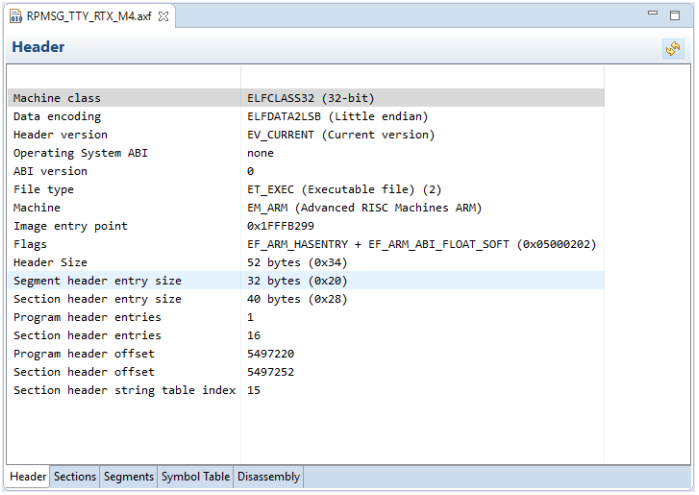
The editor area shows C/C++ source code as well as graphical representations of various configuration files such as the Run-Time Environment configuration file, the AXF file, the scatter file, and files with CMSIS configuration wizard annotations.



For more information, refer to the *C/C++ Development User's Guide* and the *CMSIS C/C++ Development User's Guide* available from the Eclipse help system (**Help → Help Contents**).

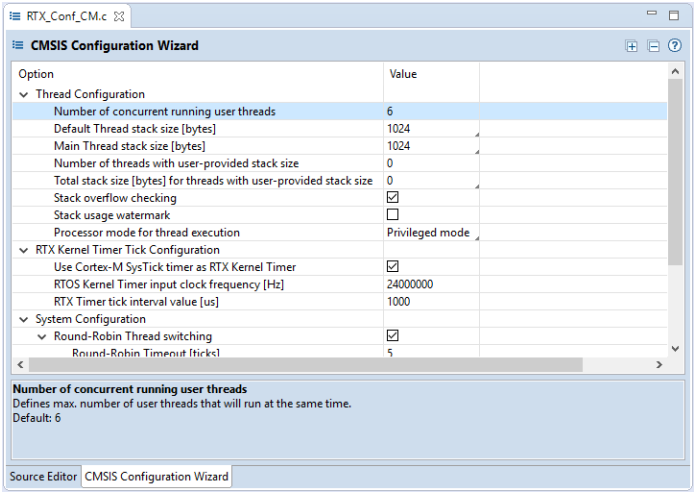
ELF file viewer

An ELF file is the executable image generated by the ARM linker that contains object code and debug information. Open it from the Project Explorer to inspect the contents of the image.



CMSIS Configuration Wizard

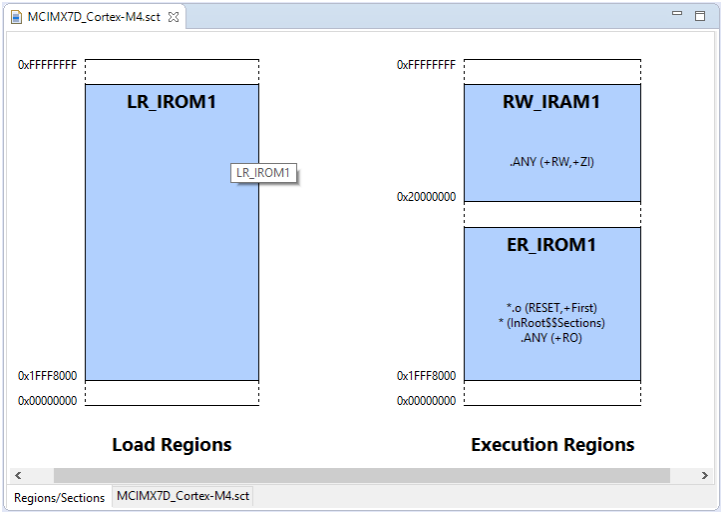
Right-click on a file in the Project Explorer and select **Open With → CMSIS Configuration Wizard** to modify files with CMSIS configuration wizard annotations in a graphical editor. Verify and adapt the contents directly in the graphical representation of the text file.



Scatter File Viewer

Scatter files (*.sct) are used to specify the memory map of an image to the linker. The **Scatter File Viewer** lets you inspect this text file in a graphical representation. Use the *filename.sct* tab to edit the scatter file contents (refer to **Save the file using** or CTRL+S

Adapt the scatter file on page 35).



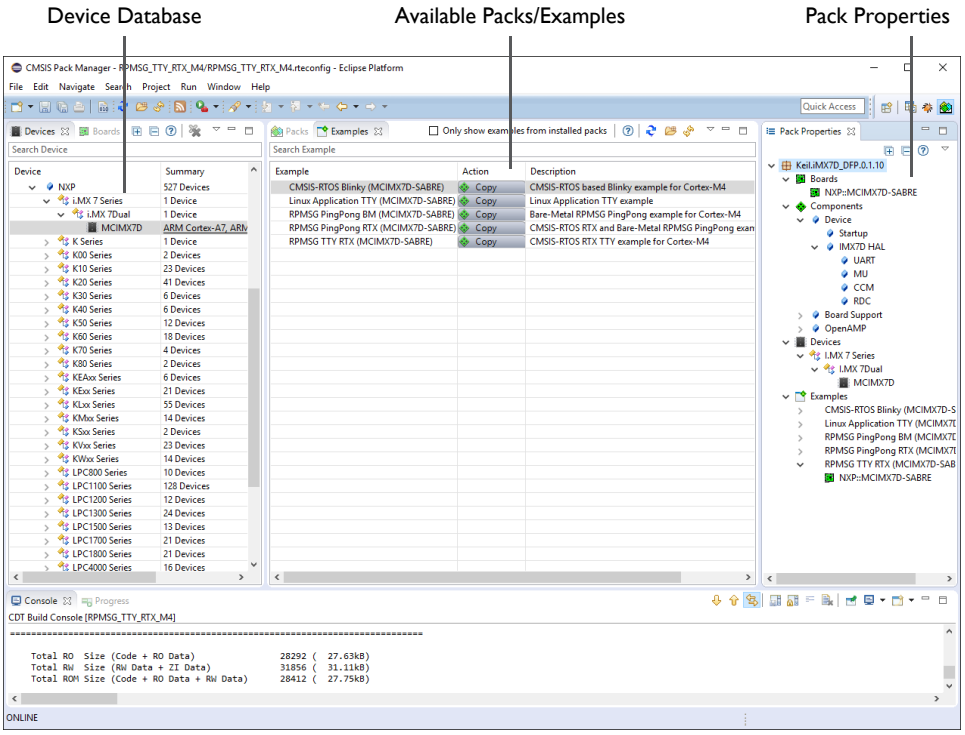
If you want to learn more about the scatter loading mechanism, look for the documentation at https://developer.arm.com.

CMSIS Pack Manager perspective

The Pack Manager perspective offers the following functionality:

- Install or update software packs.
- List devices and boards supported by software packs.
- List example projects from software packs.

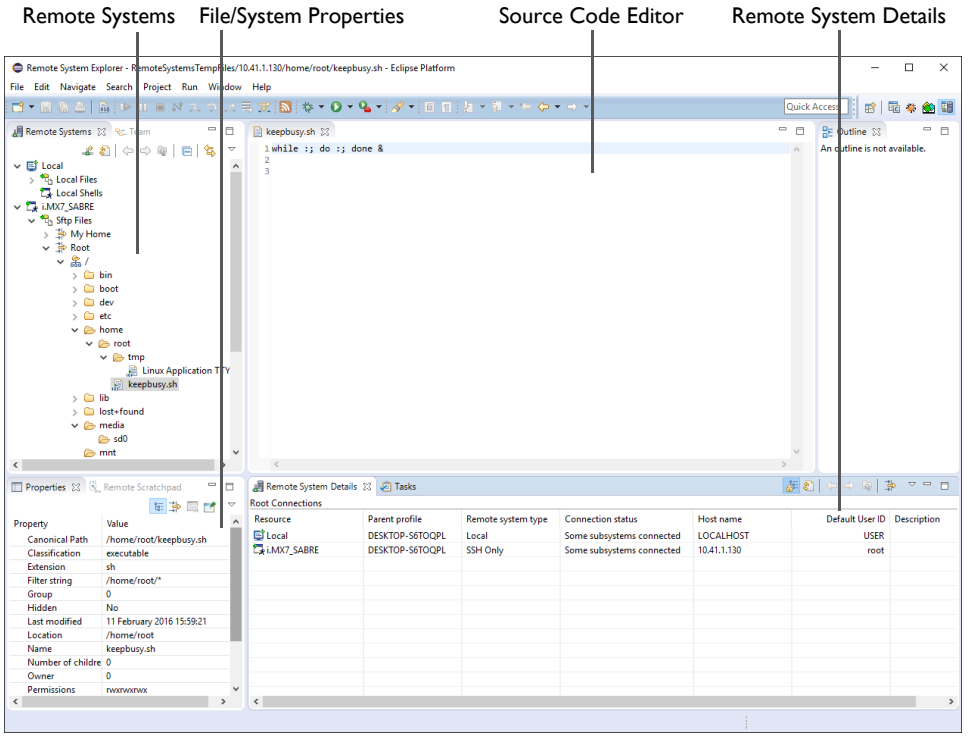
Use the  icon and select **CMSIS Pack Manager**, to open this perspective.



For more information, refer to the *CMSIS C/C++ Development User's Guide* available from the Eclipse help system (**Help → Help Contents**).

Remote System Explorer perspective

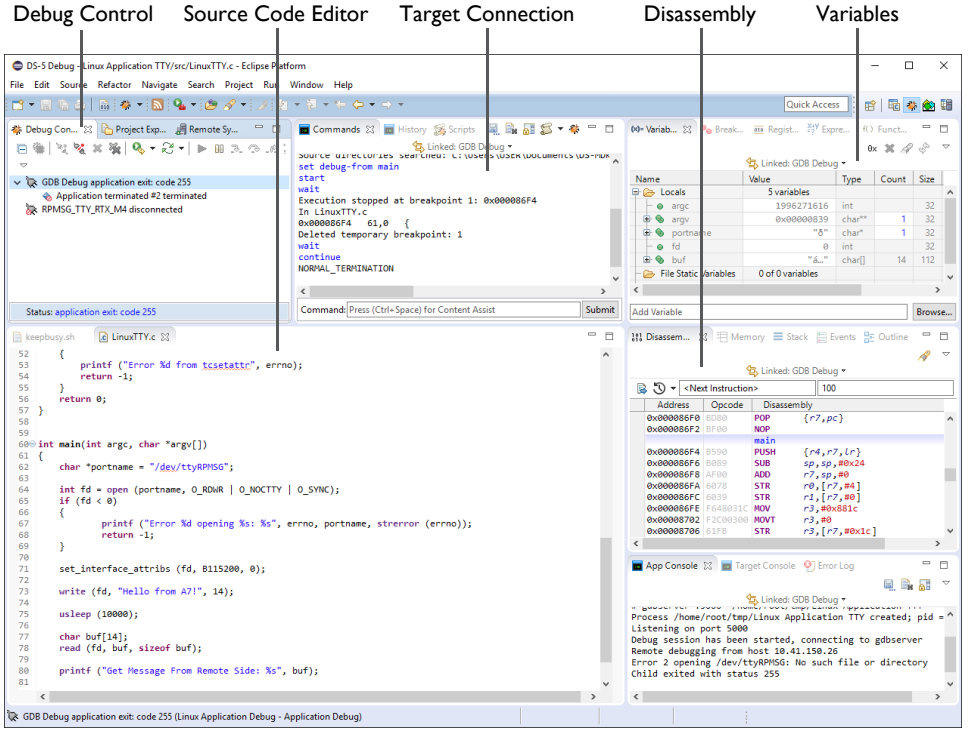
The **Remote System Explorer (RSE)** is a workbench perspective that allows you to connect and work with a variety of remote systems. With predefined plug-ins, you can look at remote file systems, transfer files between hosts, do remote search, execute commands and work with processes.



For more information, refer to the *RSE User Guide* in the Eclipse help system (**Help → Help Contents**).

DS-5 Debug perspective

The DS-5 Debugger allows you to debug bare-metal, RTOS, and Linux applications with comprehensive and intuitive views, including synchronized source and disassembly, call stack, memory, registers, expressions, variables, threads, breakpoints, and trace.



For more information, refer to the *ARM DS-5 Debugger Documentation* in the *ARM DS-MDK Documentation* available from the Eclipse help system (**Help → Help Contents**).

Additional links

Kernel.org: **http://www.kernel.org/doc/Documentation/arm/Booting**

Debugging with scripts:
https://developer.arm.com/docs/dui0446/latest/debugging-with-scripts

Debug configurations: **https://developer.arm.com/docs/dui0446/latest/ds-5-debug-perspectives-and-views/debug-configurations-debugger-tab**

Index