MCUXpresso IDE User Guide

Rev. 10.1.0 — 14 November, 2017 User guide

14 November, 2017

1. Introduction to MCUXpresso IDE

MCUXpresso IDE is a low-cost microcontroller (MCU) development platform ecosystem from NXP, which provides an end-to-end solution enabling engineers to develop embedded applications from initial evaluation to final production.

The MCUXpresso platform ecosystem includes:

- MCUXpresso IDE [6], a software development environment for creating applications for NXP's ARM Cortex-M based MCUs including "LPC" and "Kinetis" ranges.
- MCUXpresso Config Tools[87], new in MCUXpresso IDE version 10.1.0, configuration tools comprising Pins, Clocks and Peripherals Tools designed to work with SDK projects are fully integrated and installed by default
- MCUXpresso SDKs [28], each offering a package of device support and example software extending the capability and park knowledge of MCUXpresso IDE.
- The range of LPCXpresso development boards, each of which includes a built-in "LPC-Link", "LPC-Link2", or CMSIS-DAP debug probe. These boards are developed in collaboration with Embedded Artists.
- The range of Tower and Freedom Development boards, most of which include an Open SDA debug circuit supporting a range of firmware options.
- The standalone "LPC-Link2" debug probe.

This guide is intended as an introduction to using MCUXpresso IDE. It assumes that you have some knowledge of MCUs and software development for embedded systems.

Note: MCUXpresso IDE is built on top of much of the technology contained within the LPCXpresso IDE. This means that for users familiar with LPCXpresso IDE, the new MCUXpresso IDE will look relatively familiar.

1.1 MCUXpresso IDE Overview of Features

The MCUXpresso IDE is a fully featured software development environment for NXP's ARM-based MCUs, and includes all the tools necessary to develop high-quality embedded software applications in a timely and cost effective fashion.

MCUXpresso IDE is based on the Eclipse IDE and includes the industry standard ARM GNU toolchain. It brings developers an easy-to-use and unlimited code size development environment for NXP MCUs based on Cortex-M cores (LPC and Kinetis). This new IDE combines the best of the widely popular LPCXpresso and Kinetis Design Studio IDEs, providing a common platform for all NXP Cortex-M microcontrollers. With full-featured free (code size unlimited) and an affordable professional edition, MCUXpresso IDE provides an intuitive and powerful interface with profiling, power measurement on supported boards, GNU tool integration and library, multicore capable debugger, trace functionality and more. MCUXpresso IDE debug connections support Freedom, Tower®, LPCXpresso and your custom development boards with industry- leading open-source and commercial debug probes including LPC-Link2, P&E and SEGGER.

The fully featured debugger supports both SWD and JTAG debugging, and features direct download to on-chip Flash.

For the latest details on new features and functionality, please visit:

http://www.nxp.com/mcuxpresso/ide

1.1.1 Summary of Features

Complete C/C++ integrated development environment

- Eclipse-based IDE with many ease-of-use enhancements
  - Built on Eclipse Neon (v4.6) and CDT (v9.1)
- The IDE installs with Eclipse Plugins offering
  - Git, FreeRTOS and support for P&E Micro debug probes
- The IDE can be further enhanced with many other Eclipse plugins
- Command-line tools included for integration into build, test, and manufacturing systems

Industry standard GNU toolchain (v6 update 2) including:

- C and C++ compilers, assembler, and linker
- Converters for SREC, HEX, and binary

Advanced project wizards

- Simple creation of preconfigured applications for specific MCUs
  - Extendable with MCUXpresso SDKs
- Device-specific support for NXP's ARM-based MCUs (including LPC and Kinetis)
- Automatic generation of linker scripts for correct placement of code and data into Flash and RAM
  - Extended support for flexible placement of heap and stack
- Automatic generation of MCU-specific startup and device initialization code
- No assembler required with Cortex-M MCUs

Advanced multicore support

- Provision for creating linked projects for each core in multicore MCUs
- Debugging of multicore projects within a single IDE instance, with the ability to link various debug views to specific cores

Fully featured native debugger supporting JTAG and SWD connection via LinkServer

- Built-in optimized Flash programming for internal and SPI Flash
- High-level and instruction-level debug
- Views of CPU registers and on-chip peripherals
- Support for multiple devices on the JTAG scan-chain

Full install and integration of 3rd party debug solutions from:

- P&E Micro
- SEGGER J-Link

Library support

- Redlib: a small-footprint embedded C library
  - RedLib-nf: a smaller footprint library offering reduced fprintf support
- Newlib: a complete C and C++ library
- NewlibNano: a new small-footprint C and C++ library, based on Newlib
- LPCOpen MCU software libraries
- Cortex Microcontroller Software Interface Standard (CMSIS) libraries and source code
- Extendable support per device via MCUXpresso SDKs

LinkServer Trace functionality

- Instruction trace via Embedded Trace Buffer (ETB) on certain Cortex-M3/M4 based MCUs or via Micro Trace Buffer (MTB) on Cortex-M0+ based MCUs

  - Providing a snapshot of application execution with linkage back to source, disassembly and profile
- SWO Trace on Cortex-M3/M4 based MCUs when debugging via LPC-Link2, providing functionality including:
  - Profile tracing
  - Interrupt tracing
  - Datawatch tracing
  - Printf over ITM

LinkServer Power Measurement

- On LPCXpresso boards, sample power usage at adjustable rates of up to 200 ksps; average power usage display option
- Explore detailed plots of collected data in the IDE
- Export data for analysis with other tools

MCUXpresso Configuration Tools

- New in MCUXpresso IDE version 10.1.0, MCUXpresso Config Tools [87], designed to work with SDK projects are fully integrated and installed by default, comprising:
  - Pins Tool
  - Clocks Tool
  - Peripherals Tool

1.1.2 Supported Debug Probes

MCUXpresso IDE installs with built in support for 3 debug solutions:

- **Native LinkServer** (including CMSIS-DAP) as also used in LPCXpresso IDE
  - this supports a variety of debug probes including OpenSDA programmed with CMSIS-DAP firmware, LPC-Link2 etc.
  - https://community.nxp.com/message/630896
- **P&E Micro**
  - this supports a variety of debug probes including OpenSDA programmed with P&E compatible firmware and MultiLink and Cyclone probes
  - http://www.pemicro.com/
- **SEGGER J-Link**
  - this supports a variety of debug probes including OpenSDA programmed with J-Link compatible firmware and J-Link debug probes
  - https://www.segger.com/

This support includes the installation of all necessary drivers and supporting software.

Please see Debug Solutions Overview Chapter [11] for more details.

Note: Kinetis Freedom and Tower boards typically provide an on-board OpenSDA debug circuit. This can be programmed with a range of debug firmware including:

- mBed CMSIS-DAP – supported by LinkServer connections
- DAP-Link – supported by LinkServer connections (DAP-Link is preferred to mBed CMSIS-DAP when available)
- J-Link – supported by SEGGER J-Link connections
- P&E – supported by P&E connections

The default firmware can be changed if required, for details of the procedure and range of supported firmware options please information visit: http://www.nxp.com/opensda

1.1.3 Development Boards

NXP Development board come in 3 families:

LPCXpresso Boards for LPC

The range of LPCXpresso boards that work seamlessly with the MCUXpresso IDE. These boards provide practical and easy-to-use development hardware to use as a starting point for your LPC Cortex-M MCU based projects.

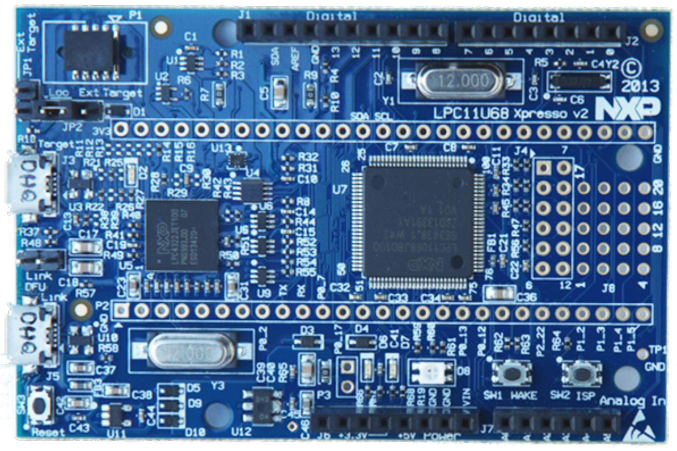


Figure 1.1. LPCXpresso V2 Board (LPCXpresso11U68)

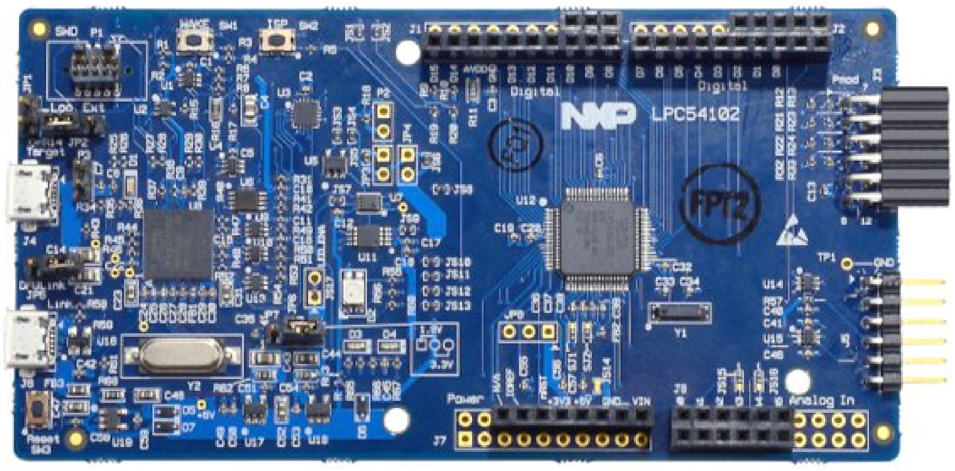


Figure 1.2. LPCXpresso V3 Board (LPCXpresso54102)

For more information, visit: http://www.nxp.com/lpcxpresso-boards

Freedom and Tower Boards for Kinetis

Similarly, for Kinetis MCUs there are many development boards available including the popular Freedom and Tower ranges of boards.

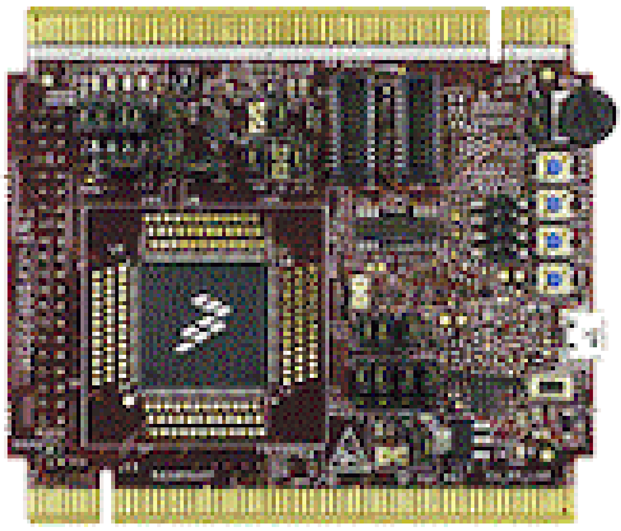


Figure 1.3. Tower (TWR-KV58F220M)

For more information, visit: http://www.nxp.com/pages/:TOWER_HOME

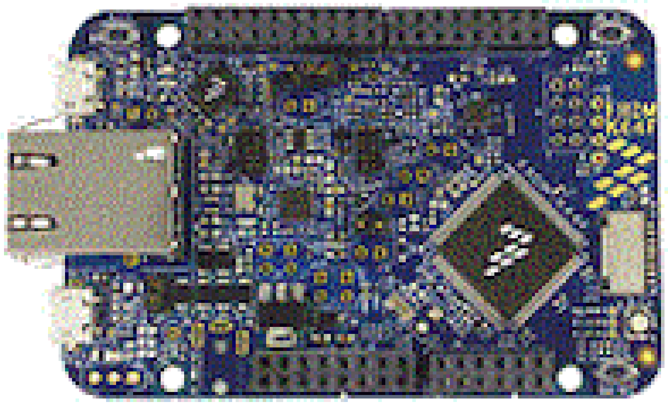


Figure 1.4. Freedom (FRDM-K64F)

For more information, visit: http://www.nxp.com/pages/:FREDEVPLA

2. IDE Overview

The following chapter provides a high level overview of the features offered by the IDE itself.

2.1 Documentation and Help

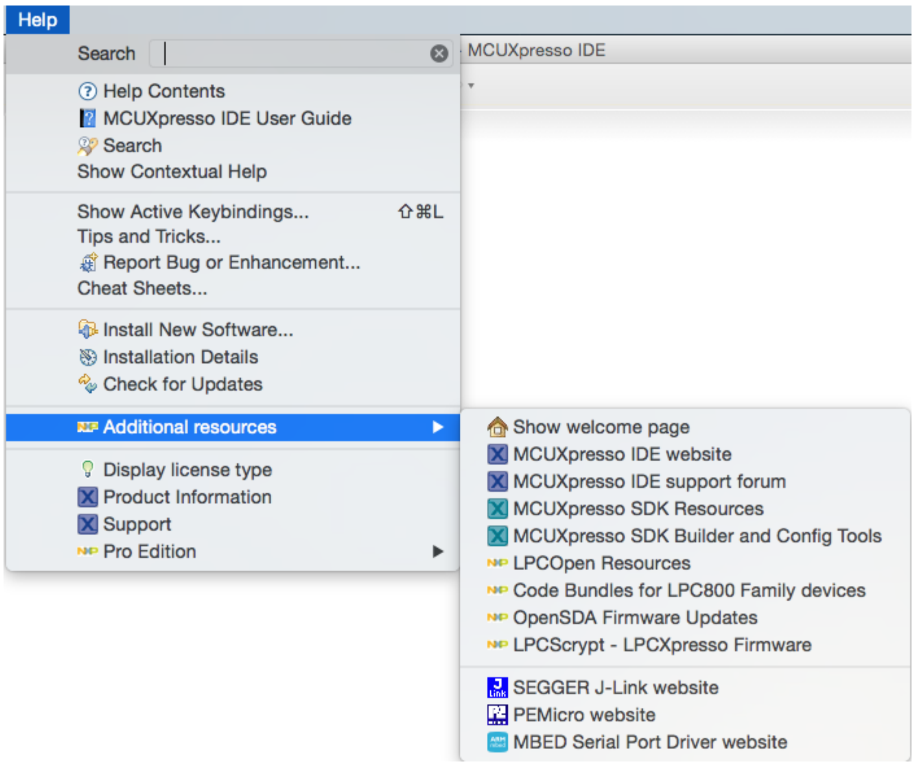
The MCUXpresso IDE is based on the Eclipse IDE framework, and many of the core features are described well in generic Eclipse documentation and in the help files to be found on the MCUXpresso IDE's **Help -> Help Contents** menu. It also provides access to the MCUXpresso IDE User Guide (this document), as well as the documentation for the compiler, linker, and other underlying tools.

MCUXpresso IDE documentation comprises a suite of documents including:

- MCUXpresso IDE Installation Guide
- MCUXpresso IDE User Guide
- MCUXpresso IDE LinkServer SWO Trace Guide
- MCUXpresso IDE LinkServer Instruction Trace Guide
- MCUXpresso IDE LinkServer Power Measurement Guide
- MCUXpresso IDE FreeRTOS Debug Guide
- MCUXpresso (IDE) Config Tools User's Guide

To obtain assistance on using MCUXpresso IDE, visit: http://www.nxp.com/mcuxpresso/ide

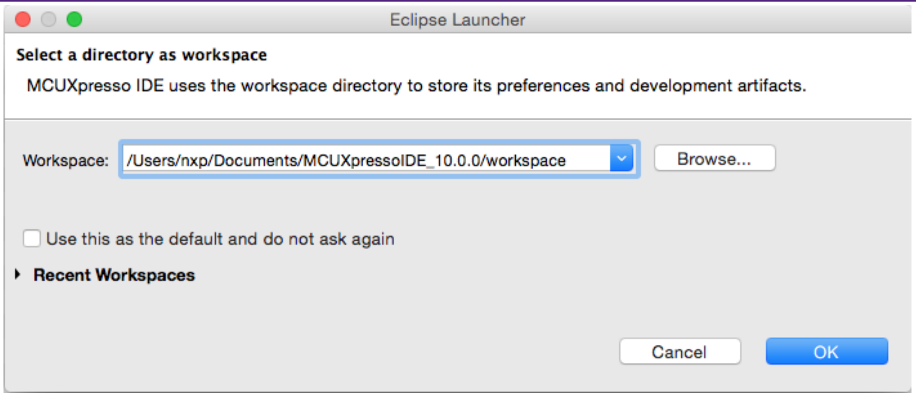
Related web links can be found at *Help -> Additional resources* as shown below:



When MCUXpresso IDE is started, a Welcome page is displayed (usually within the Editor view). This page contains product information including a link to the User Guide. If this page is not required on startup, it can be disabled via unticking the preference at *Preferences -> MCUXpresso IDE -> General -> Show welcome view*.

2.2 Workspaces

When you first launch MCUXpresso IDE, you will be asked to select a Workspace, as shown in Figure 2.1.


Eclipse Launcher

Select a directory as workspace

MCUXpresso IDE uses the workspace directory to store its preferences and development artifacts.

Workspace: /Users/nxp/Documents/MCUXpressoIDE_10.0.0/workspace

Use this as the default and do not ask again

Figure 2.1. Workspace selection

A Workspace is simply a directory used to store projects. MCUXpresso IDE can only access a single Workspace at a time, although it is possible to run multiple instances in parallel — with each instance accessing a different Workspace.

If you tick the **Use this as the default and do not ask again** option, then MCUXpresso IDE will always start up with the chosen Workspace opened; otherwise, you will always be prompted to choose a Workspace.

You may change the Workspace that MCUXpresso IDE is using, via the **File -> Switch Workspace** option.

2.3 Perspectives and Views

The overall layout of the main MCUXpresso IDE window is known as a Perspective. Within each Perspective are many sub-windows, called Views. A View displays a set of data in the IDE environment. For example, this data might be source code, hex dumps, disassembly, or memory contents. Views can be opened, moved, docked, and closed, and the layout of the currently displayed Views can be saved and restored.

Typically, the MCUXpresso IDE operates using the single **Develop Perspective**, under which both code development and debug sessions operate as shown in Figure 2.3. This single perspective simplifies the Eclipse environment, but at the cost of slightly reducing the amount of information displayed on screen.

Alternatively, the MCUXpresso IDE can operate in a “dual Perspective” mode such that the **C/C++ Perspective** is used for developing and navigating around your code and the **Debug Perspective** is used when debugging your application.

You can manually switch between Perspectives using the Perspective icons in the top right of the MCUXpresso IDE window, as shown in Figure 2.2.

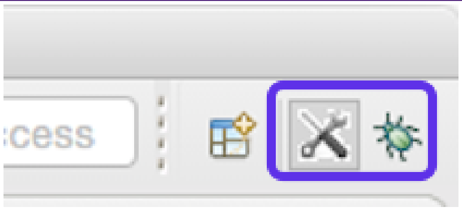
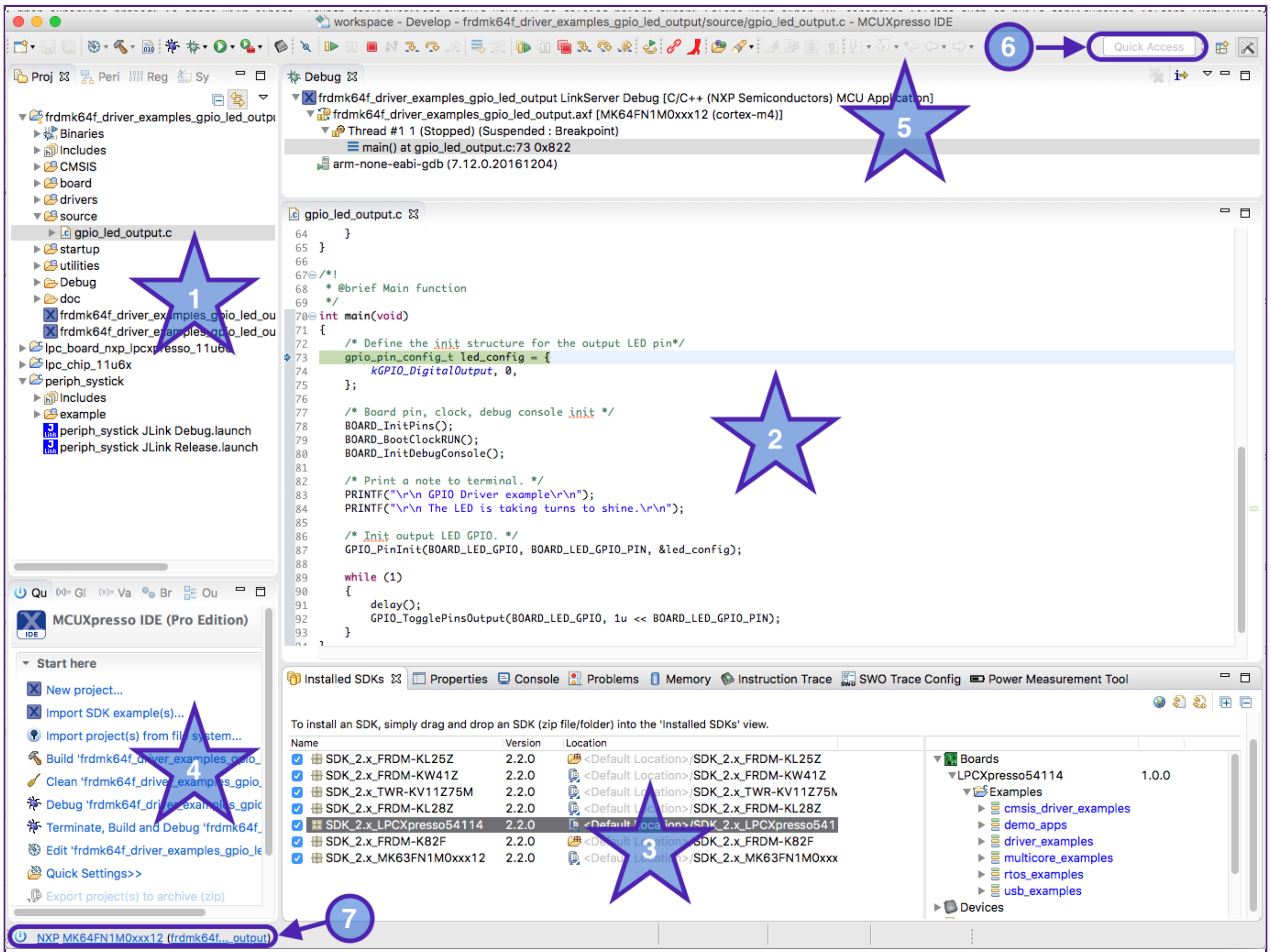


Figure 2.2. Perspective selection

All Views in a Perspective can also be rearranged to match your specific requirements by dragging and dropping. If a View is accidentally closed, it can be restored by selecting it from the **Window -> Show View** dialog. The default layout for a perspective can be restored at any time via **Window -> Perspective -> Reset Perspective**.

2.4 Major Components of the Develop Perspective


workspace - Develop - frdmk64f_driver_examples_gpio_led_output/source/gpio_led_output.c - MCUXpresso IDE
Quick Access
Proj
Peri
Reg
Sy
Debug
gpio_led_output.c
Installed SDKs
Properties
Console
Problems
Memory
Instruction Trace
SWO Trace Config
Power Measurement Tool
MCUXpresso IDE (Pro Edition)
Start here
New project...
Import SDK example(s)...
Import project(s) from file system...
Build 'frdmk64f_driver_examples_gpio_...
Clean 'frdmk64f_driver_examples_gpio_...
Debug 'frdmk64f_driver_examples_gpio_...
Terminate, Build and Debug 'frdmk64f_...
Edit 'frdmk64f_driver_examples_gpio_le...
Quick Settings>>
Export project(s) to archive (zip)
NXP MK64FN1M0xxx12 (frdmk64f..._output)
1
2
3
4
5
6
7

```
frdmk64f_driver_examples_gpio_led_output LinkServer Debug [C/C++ (NXP Semiconductors) MCU Application]
  frdmk64f_driver_examples_gpio_led_output.axf [MK64FN1M0xxx12 (cortex-m4)]
    Thread #1 1 (Stopped) (Suspended : Breakpoint)
      main() at gpio_led_output.c:73 0x822
  arm-none-eabi-gdb (7.12.0.20161204)
```

```
frdmk64f_driver_examples_gpio_led_outp
  Binaries
  Includes
  CMSIS
  board
  drivers
  source
    gpio_led_output.c
  startup
  utilities
  Debug
  doc
  frdmk64f_driver_examples_gpio_led_ou
  frdmk64f_driver_examples_gpio_led_ou
lpc_board_nxp_lpcxpresso_11u68
lpc_chip_11u6x
periph_systick
  Includes
  example
  periph_systick JLink Debug.launch
  periph_systick JLink Release.launch
```

```c
    }
}

/*!
 * @brief Main function
 */
int main(void)
{
    /* Define the init structure for the output LED pin*/
    gpio_pin_config_t led_config = {
        kGPIO_DigitalOutput, 0,
    };

    /* Board pin, clock, debug console init */
    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();

    /* Print a note to terminal. */
    PRINTF("\r\n GPIO Driver example\r\n");
    PRINTF("\r\n The LED is taking turns to shine.\r\n");

    /* Init output LED GPIO. */
    GPIO_PinInit(BOARD_LED_GPIO, BOARD_LED_GPIO_PIN, &led_config);

    while (1)
    {
        delay();
        GPIO_TogglePinsOutput(BOARD_LED_GPIO, 1u << BOARD_LED_GPIO_PIN);
    }
```

To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view.

Name	Version	Location
SDK_2.x_FRDM-KL25Z	2.2.0	<Default Location>/SDK_2.x_FRDM-KL25Z
SDK_2.x_FRDM-KW41Z	2.2.0	<Default Location>/SDK_2.x_FRDM-KW41Z
SDK_2.x_TWR-KV11Z75M	2.2.0	<Default Location>/SDK_2.x_TWR-KV11Z75M
SDK_2.x_FRDM-KL28Z	2.2.0	<Default Location>/SDK_2.x_FRDM-KL28Z
SDK_2.x_LPCXpresso54114	2.2.0	<Default Location>/SDK_2.x_LPCXpresso541
SDK_2.x_FRDM-K82F	2.2.0	<Default Location>/SDK_2.x_FRDM-K82F
SDK_2.x_MK63FN1M0xxx12	2.2.0	<Default Location>/SDK_2.x_MK63FN1M0xxx

```
Boards
  LPCXpresso54114          1.0.0
    Examples
      cmsis_driver_examples
      demo_apps
      driver_examples
      multicore_examples
      rtos_examples
      usb_examples
Devices
```

Figure 2.3. Develop Perspective (whilst debugging)

1. Project Explorer / Peripherals / Registers Views
   - The **Project Explorer** gives you a view of all the projects in your current Workspace.
   - When debugging, the **Peripherals** view allows you to display a list of the MCU peripherals and project memory regions. Selecting a peripheral or memory region will spawn a new

window to display the detailed content. Note: depending on your MCUs configuration, some peripherals may not be powered/clocked and hence their content will not display.

   - When debugging, the **Registers** view allows you to display the registers and their content within the CPU of your MCU.
   - Not visible here is the **Symbol Viewer**; this view displays symbolic information from a referenced .axf file.

2. Editor
   - Centrally located is the **Editor**, which allows modification and saving of source code. When debugging, this is where you can see the code you are executing and can step from line to line. By pressing the `'i->'` icon at the top of the Debug view, you can switch to stepping by assembly instruction. Clicking in the left margin will set and delete breakpoints.
3. Console / Installed SDKs / Problems / Trace Views / Power Measurement
   - On the lower right are the Console, Installed SDK and Problems Views etc. The Console View displays status information on compiling and debugging, as well as semihosted program output.
   - The Installed SDK view enabled the management of installed SDKs. New SDKs can be added using drag and drop. Other SDK management features are also provided from this view including unzip, explore and delete.
   - The Problems View (available by changing tabs) shows all compiler errors and warnings and will allow easy navigation to the error location in the Editor View.
   - Sitting in parallel with the Console View are the various Views that make up the Trace functionality of MCUXpresso IDE. For more information on Trace functionality, please see the MCUXpresso IDE SWO Trace Guide and/or the MCUXpresso IDE Instruction Trace Guide.
     - The SWO trace Views allow you to gather and display runtime information using the SWO/SWV technology that is part of Cortex-M3/M4 based parts.
     - On some MCUs, you can also view instruction trace data downloaded from the MCU’s Embedded Trace Buffer (ETB) or Micro Trace Buffer (MTB).
   - Sitting in parallel with the Console View is the Power Measurement View, a dedicated trace View capable of displaying real-time target power usage. For more information please see the MCUXpresso IDE Power Measurement Guide.
4. Quickstart / Variables / Breakpoints / Outline Views
   - On the lower left of the window, the **Quickstart Panel View** has fast links to commonly used features. From here you can find various wizards including New Project, Import from SDK and Import from File System plus options such as Build, Debug, and Import.
     - **Note**: This Panel is essential to the operation of MCUXpresso IDE and so cannot be removed from the perspective.
   - Sitting in parallel to the Quickstart Panel, the **Global Variables** View allows you to see and edit the values of Global variables. Variables can be monitored while the target is running using the LinkServer Live Variables feature.
   - Sitting in parallel to the Quickstart Panel, the **Variables** View allows you to see and edit the values of local variables.
   - Sitting in parallel to the Quickstart Panel, the **Breakpoints** View allows you to see and modify currently set breakpoints.
   - Sitting in parallel to the Quickstart Panel, the **Outline** View allows you to quickly find components of the source file with input focus within the editor.
5. Debug View
   - The Debug View appears when you are debugging your application. This shows you the stack trace. In the “stopped” state, you can click on any function and inspect its local variables in the Variables tab (which is located parallel to the **Quickstart Panel View**).
6. Quick Access
   - allows quick access to features such as views, perspectives etc. for example enter ‘Error’ to view and open the IDE’s Error Log, or ‘Trace’ to view and open the various LinkServer Trace views.
7. Quick Links

- Various useful shortcuts, for example to open a project’s workspace.

3. Debug Solutions Overview

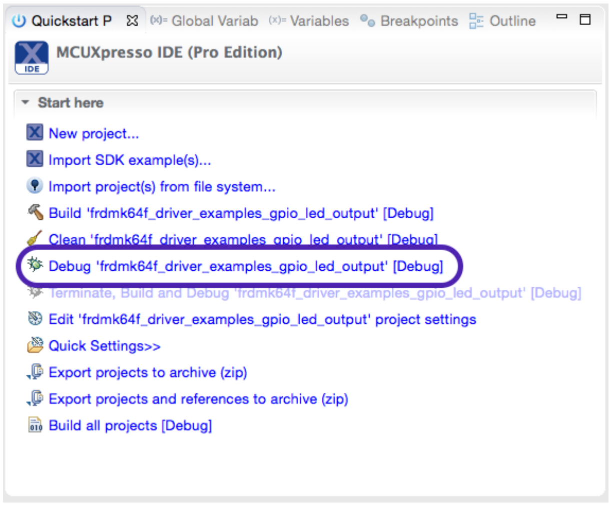
MCUXpresso IDE installs with built-in support for 3 debug solutions; comprising the Native LinkServer (including CMSIS-DAP) [15] as used in LPCXpresso IDE. Plus support for both P&E Micro [22] and SEGGER J-Link. [23]

This support includes the installation of all necessary drivers and supporting software.

The rest of this chapter discusses these different Debug solutions. For general information on debugging please see the chapter Debugging a Project [69]

Note: Within MCUXpresso IDE, the debug solution used has no impact on project setting or build configuration. Debug operations for basic debug are also identical.

To perform a debug operation:

1. select a project within the MCUXpresso IDE Project View
2. click **Debug** from within the MCUXpresso IDE **QuickStart** View
   - 
     Quickstart P | Global Variab | Variables | Breakpoints | Outline
     MCUXpresso IDE (Pro Edition)
     Start here
     New project...
     Import SDK example(s)...
     Import project(s) from file system...
     Build 'frdmk64f_driver_examples_gpio_led_output' [Debug]
     Clean 'frdmk64f_driver_examples_gpio_led_output' [Debug]
     Debug 'frdmk64f_driver_examples_gpio_led_output' [Debug]
     Terminate, Build and Debug 'frdmk64f_driver_examples_gpio_led_output' [Debug]
     Edit 'frdmk64f_driver_examples_gpio_led_output' project settings
     Quick Settings>>
     Export projects to archive (zip)
     Export projects and references to archive (zip)
     Build all projects [Debug]
   - A probe discovery operation is automatically performed to display the available debug connections, including LinkServer, P&E and J-Link compatible probes.
3. select the required debug probe and click **OK**
   - A project launch configuration is automatically created containing debug chain specific configurations
     - Launch configurations [12] are stored within a project and are different for each of the supported debug solutions

From this point onwards, the low level debug operations are controlled by one of the above debug solutions.

However, from the users point of view, most common debug operations within the IDE will appear the same (or broadly similar), for example:

- Automatic inheritance of part knowledge
- Automatic downloading of generated image to target Flash memory
- Setting breakpoints [81] and watchpoints [82]
- Stepping (single, step in step out etc.)
- Viewing and editing local variables, registers, peripherals [84], memory
- Viewing disassembly
- Semihosted IO

Note: In addition MCUXpresso IDE will dynamically manage each debug solutions connection requirements allowing multiple sessions to be started without conflict. Note: For debug of Multicore MCUs please refer to the section Debugging MultiCore Projects [156]

However, it is important to note that advanced operations such as the handling of launch configuration features may be very different for each debug solution. Furthermore, advanced debug features and capabilities may vary between solutions and even similar features may appear quite different within the IDE.

MCUXpresso IDE documentation will only describe the advanced features provided by native LinkServer debug connection. These include:

- Flash programming
  - please see the chapter Introduction to LinkServer Flash Drivers [90]
- Instruction Trace
  - please see LinkServer Instruction Trace Guide
- Live Global Variable display
  - described later in this chapter
- Power Measurement
  - please see LinkServer Power Measurement Guide
- FreeRTOS Debug
  - please see FreeRTOS Debug Guide
- SWO Trace (Profiling, Interrupts, Data Watch) - LPC-Link2 Only
  - please see LinkServer SWO Trace Guide

P&E Micro and SEGGER debug solutions also provide a number of advanced features, details can be found at their respective web sites.

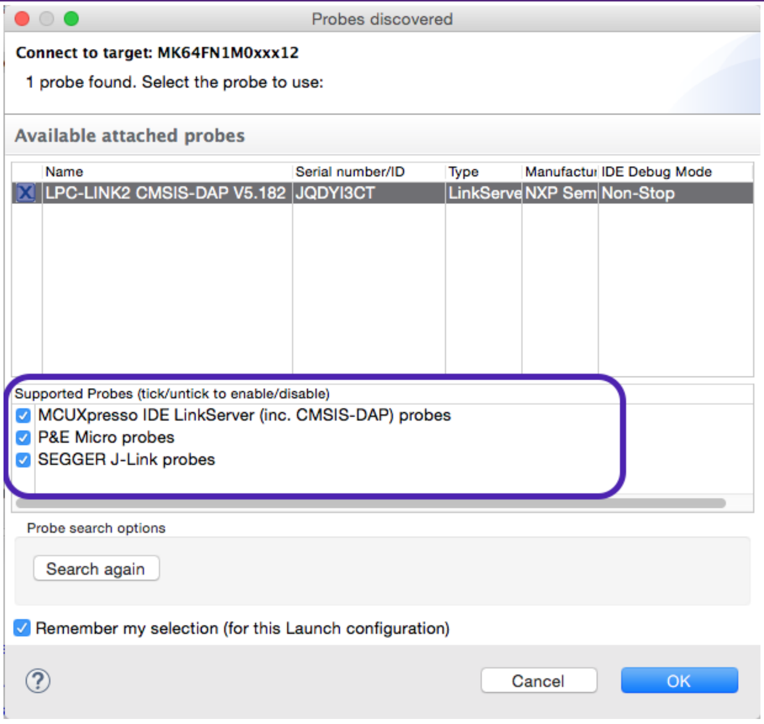
3.1 A note about Launch Configuration files

The debug properties of a project in MCUXpresso IDE are held locally within each project in **.launch** files (known as launch configuration files).

Launch configuration files are different for each debug solution (LinkServer, P&E, SEGGER) and contain the properties of the debug connection (SWD/JTAG, and various other configurations etc.) and can also include a debug probe identifier for automatic debug probe matching.

If a project has not yet been debugged, for example a newly imported or created project, then the project will not have a launch configuration associated with it.

When the user first tries to debug a project, MCUXpresso IDE will perform a **Debug Probe Discovery** operation and present the user with a list of debug probes found. **Note**: The Debug Solutions searched can be filtered from this dialogue as highlighted, removing options that are not required will speed up this process.


Probes discovered

Connect to target: MK64FN1M0xxx12

1 probe found. Select the probe to use:

Available attached probes

	Name	Serial number/ID	Type	Manufactu	IDE Debug Mode
☒	LPC-LINK2 CMSIS-DAP V5.182	JQDYI3CT	LinkServe	NXP Sem	Non-Stop

Supported Probes (tick/untick to enable/disable)
- ☑ MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes
- ☑ P&E Micro probes
- ☑ SEGGER J-Link probes

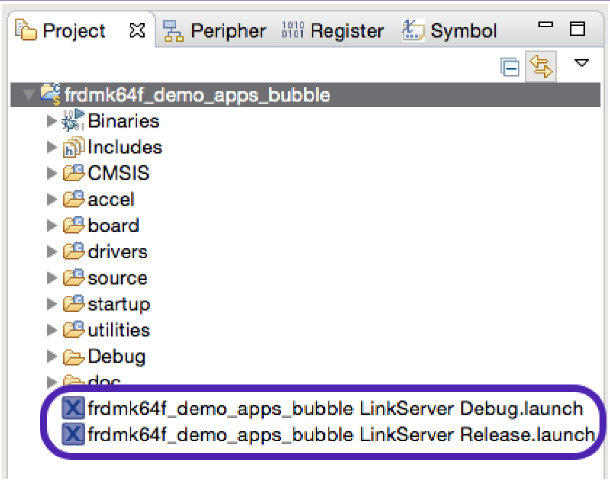
Probe search options

Search again

☑ Remember my selection (for this Launch configuration)

Figure 3.1. Debug Probe Discovery

Once the debug probe is selected and the user clicks ‘OK’, the IDE will automatically create a default launch configuration file for that debug probe (LinkServer launch configuration shown below).



```
frdmk64f_demo_apps_bubble
  Binaries
  Includes
  CMSIS
  accel
  board
  drivers
  source
  startup
  utilities
  Debug
  doc
  frdmk64f_demo_apps_bubble LinkServer Debug.launch
  frdmk64f_demo_apps_bubble LinkServer Release.launch
```

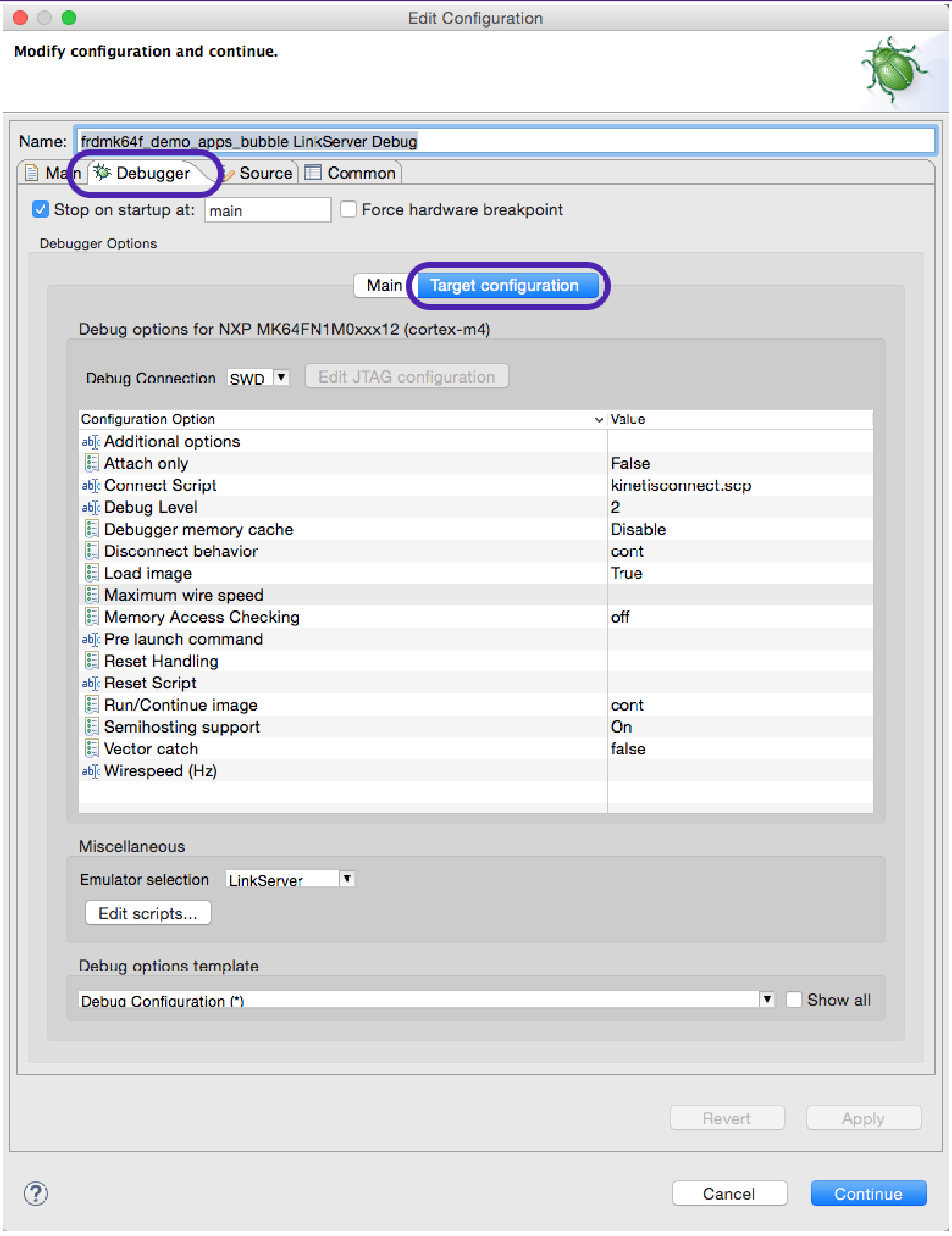
Figure 3.2. Launch Configuration Files

Note: a launch configuration will be created for each project build configuration.

For most debug operations, these files will not require any attention and can essentially be ignored. However, if changes are required, these files should not be edited manually, rather their properties should be explored within the IDE.

The simplest way to do this is to click to open the Project within the ‘Project Explorer’ pane, then simply double click any launch configuration file to automatically open the launch configuration *Edit Configuration* dialogue.

Note: This dialogue has a number of internal tabs, the *Debugger* tab (as shown below) contains the debug main settings.



Edit Configuration

Modify configuration and continue.

Name: frdmk64f_demo_apps_bubble LinkServer Debug

Tabs: Main | Debugger | Source | Common

[✓] Stop on startup at: main   [ ] Force hardware breakpoint

Debugger Options

Main | Target configuration

Debug options for NXP MK64FN1M0xxx12 (cortex-m4)

Debug Connection: SWD   Edit JTAG configuration

Configuration Option	Value
Additional options	
Attach only	False
Connect Script	kinetisconnect.scp
Debug Level	2
Debugger memory cache	Disable
Disconnect behavior	cont
Load image	True
Maximum wire speed	
Memory Access Checking	off
Pre launch command	
Reset Handling	
Reset Script	
Run/Continue image	cont
Semihosting support	On
Vector catch	false
Wirespeed (Hz)	

Miscellaneous

Emulator selection: LinkServer

Edit scripts...

Debug options template

Debug Configuration (*)   [ ] Show all

Figure 3.3. Launch Configuration

Some debug solutions support advanced operations (such as recovering of badly programmed parts) from this view.

Note: Once a launch configuration file has been created, it will be used for the projects future debug operations. If you wish to use the project with a different debug probe, then simply delete the existing launch configuration and allow a new one to be automatically used on the next debug operation.

Enhancement: New in MCUXpresso IDE version 10.1.0 – to simplify this operation, a probe discovery can be forced by holding the *SHIFT* key while launching a debug session from the Quickstart panel. If the new debug connection is completed, a new project launch configuration will created replacing any existing launch configurations.

Note: When exporting a project to share with others, launch configurations should usually be deleted before export (along with other IDE generated folders such as build configuration folders (Debug/Release if present)).

For further information please see the section Launch Configurations [73]

3.2 LinkServer Debug Connections

MCUXpresso IDE's native debug connection (known as LinkServer) supports debug operation through the following debug probes:

- LPC-Link2 with CMSIS-DAP firmware
- LPCXpresso V2/V3 Boards incorporating LPC-Link2 with CMSIS-DAP firmware
- CMSIS-DAP firmware installed onto on-board debug probe hardware (as shipped by default on LPCXpresso MAX and CD boards)
  - For more information on LPCXpresso boards see: http://www.nxp.com/lpcxpresso-boards
  - Additional driver may be required:
    - https://developer.mbed.org/handbook/Windows-serial-configuration
- CMSIS-DAP firmware installed onto on-board OpenSDA debug probe hardware (as shipped by default on certain Kinetis FRDM and TWR boards)
  - Known as DAP-Link and mBed CMSIS-DAP: http://www.nxp.com/opensda
  - Additional driver may be required:
    - https://developer.mbed.org/handbook/Windows-serial-configuration
- Other CMSIS-DAP probes such as Keil uLINK with CMSIS-DAP firmware: http://www2.keil.com/mdk5/ulink
- Legacy RedProbe+ and LPC-Link
- RDB1768 development board built-in debug connector (RDB-Link)
- RDB4078 development board built-in debug connector

Note: MCUXpresso IDE will automatically try to softload the latest CMSIS-DAP firmware onto LPC-Link2 or LPCXpresso V2/V3 boards. For this to occur, the DFU link on these boards must be set correctly. Please refer to the boards documentation for details.

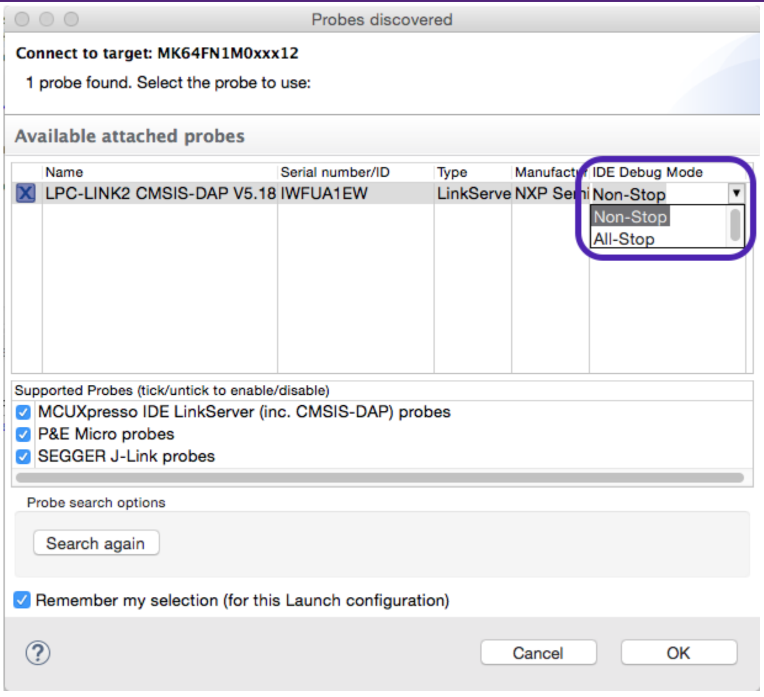
3.3 LinkServer Debug Operation

When the user first tries to debug a project, MCUXpresso IDE will perform a Debug Probe Discovery operation and present the user with a list of debug probes found.

Note: To perform a debug operation within MCUXpresso IDE, select the project to debug within the 'Project Explorer' view and then click Debug from the **QuickStart** View.

If more than one debug probe is presented, select the required probe. For LinkServer compatible debug probes, you can select from Non-Stop (the default) or All-Stop IDE debug mode.

Non-Stop uses GDB's "non-stop mode" and allows data to be read from the target while an application is running. Currently this mechanism is used to support the Live Variables feature within the Global Variables view.



Probes discovered

Connect to target: MK64FN1M0xxx12

1 probe found. Select the probe to use:

Available attached probes

	Name	Serial number/ID	Type	Manufact	IDE Debug Mode
☒	LPC-LINK2 CMSIS-DAP V5.18	IWFUA1EW	LinkServe	NXP Semi	Non-Stop

Supported Probes (tick/untick to enable/disable)

- ☑ MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes
- ☑ P&E Micro probes
- ☑ SEGGER J-Link probes

Probe search options

☑ Remember my selection (for this Launch configuration)

Figure 3.4. Debug Probe Discovery Non-Stop

Click ‘OK’ to start the debug session. At this point, the projects launch configuration files will be created. LinkServer Launch configuration files will contain the string ‘LinkServer’.

Note: If “Remember My Selection” is left ticked, then the probe details will be stored within the launch configuration file, and this probe will be automatically selected on subsequent debug operations for this project.

For a description of some common debugging operations using supported debug probes see Common Debugging Operations [76]

3.4 LinkServer Global and Live Global Variables

MCUXpresso IDE provides a new Global Variables view for displaying the values of global variables. This replaces the use of the “Expressions” view for displaying such variables, as used in LPCXpresso IDE (and KDS). This view defaults to be located within the **QuickStart** panel.

This view can be populated from a selection of a projects global variables. Simply click the “Add global” button to launch a dialogue:

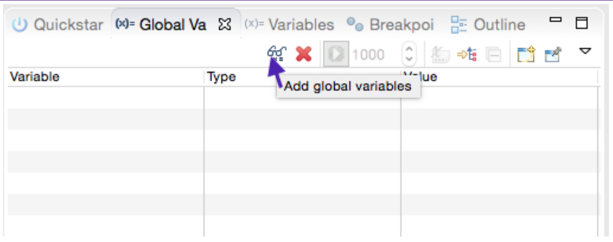
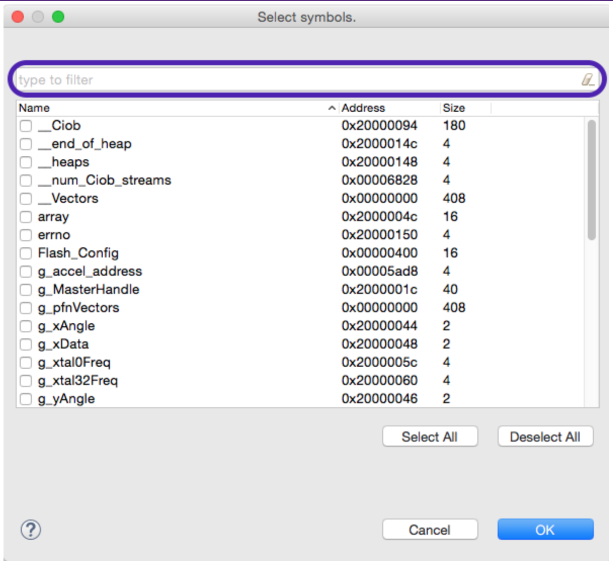


Figure 3.5. LinkServer Add Global Variables

This will then display a list of the global variables available in the image being debugged. Select the ones of interest via their checkboxes and click OK :



Name	Address	Size
__Ciob	0x20000094	180
__end_of_heap	0x2000014c	4
__heaps	0x20000148	4
__num_Ciob_streams	0x00006828	4
__Vectors	0x00000000	408
array	0x2000004c	16
errno	0x20000150	4
Flash_Config	0x00000400	16
g_accel_address	0x00005ad8	4
g_MasterHandle	0x2000001c	40
g_pfnVectors	0x00000000	408
g_xAngle	0x20000044	2
g_xData	0x20000048	2
g_xtal0Freq	0x2000005c	4
g_xtal32Freq	0x20000060	4
g_yAngle	0x20000046	2

Figure 3.6. LinkServer Global Variable Selector

Note: to simplify the selection of a variable, this dialogue supports the option to filter (highlighted) and sorts on each column.

Once selected, the chosen variables will be remembered for that occurrence of the dialogue.

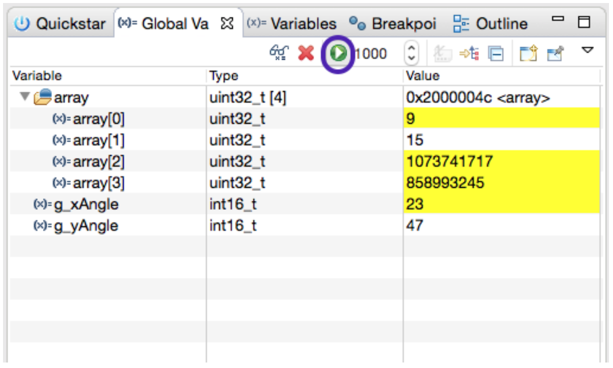
For “All-Stop” debug connections, the Global Variables view will be updated whenever the target is paused.

For “Non-Stop” debug connections, variables can be selected to be updated while the target is running. These are known as " **Live Variables**".

For variables to be “Live”:

- the target must be running
- the enable/disable (run) button clicked.

Once done, the display will update at the frequency selected (selectable from 500 ms to 10 s).



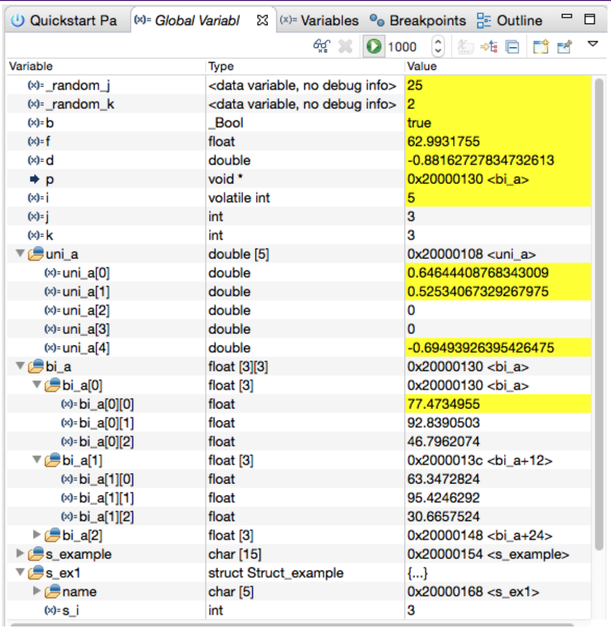
Variable	Type	Value
array	uint32_t [4]	0x2000004c <array>
array[0]	uint32_t	9
array[1]	uint32_t	15
array[2]	uint32_t	1073741717
array[3]	uint32_t	858993245
g_xAngle	int16_t	23
g_yAngle	int16_t	47

Figure 3.7. LinkServer Global Variable Display

Live Variables like normal Globals can also be edited in place. Simply click on the variable value and edit the contents. During the edit operation, the display will not update. This mechanism provides a powerful way of interacting with a running target without impacting on any other aspect of system performance.

Note: If you wish to have some global variables ‘Live’ and others not, then this can be achieved by spawning a second Globals display via the ‘New View’ button and populating this without enabling the ‘run’ feature for that view.

The usefulness of **Live Variables** reduces as the number of Globals monitored increases, and ultimately there will be a limit as to how many variables can be updated at the selected frequency. However, complex list of variables can be monitored if required. For example:

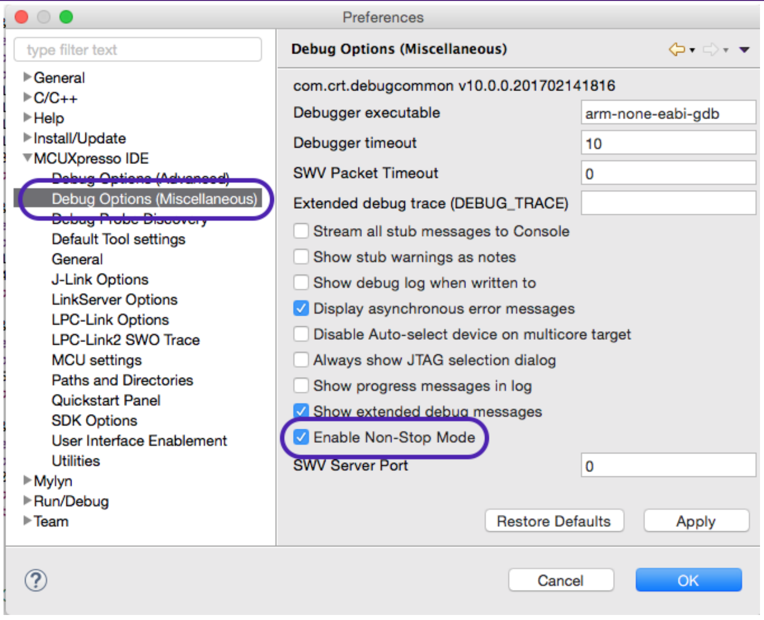


Variable	Type	Value
_random_j	<data variable, no debug info>	25
_random_k	<data variable, no debug info>	2
b	_Bool	true
f	float	62.9931755
d	double	-0.88162727834732613
p	void *	0x20000130 <bi_a>
i	volatile int	5
j	int	3
k	int	3
uni_a	double [5]	0x20000108 <uni_a>
uni_a[0]	double	0.64644408768343009
uni_a[1]	double	0.52534067329267975
uni_a[2]	double	0
uni_a[3]	double	0
uni_a[4]	double	-0.69493926395426475
bi_a	float [3][3]	0x20000130 <bi_a>
bi_a[0]	float [3]	0x20000130 <bi_a>
bi_a[0][0]	float	77.4734955
bi_a[0][1]	float	92.8390503
bi_a[0][2]	float	46.7962074
bi_a[1]	float [3]	0x2000013c <bi_a+12>
bi_a[1][0]	float	63.3472824
bi_a[1][1]	float	95.4246292
bi_a[1][2]	float	30.6657524
bi_a[2]	float [3]	0x20000148 <bi_a+24>
s_example	char [15]	0x20000154 <s_example>
s_ex1	struct Struct_example	{...}
name	char [5]	0x20000168 <s_ex1>
s_i	int	3

Figure 3.8. LinkServer Global Variable Display 1

MCUXpresso IDE defaults to the selection of “Non-Stop” mode when a LinkServer probe discovery operation is performed. This can be disabled from an MCUXpresso IDE Preference via:

Preferences -> Debug Options (Misc)


Preferences

type filter text

- General
- C/C++
- Help
- Install/Update
- MCUXpresso IDE
  - Debug Options (Advanced)
  - Debug Options (Miscellaneous)
  - Debug Probe Discovery
  - Default Tool settings
  - General
  - J-Link Options
  - LinkServer Options
  - LPC-Link Options
  - LPC-Link2 SWO Trace
  - MCU settings
  - Paths and Directories
  - Quickstart Panel
  - SDK Options
  - User Interface Enablement
  - Utilities
- Mylyn
- Run/Debug
- Team

Debug Options (Miscellaneous)

com.crt.debugcommon v10.0.0.201702141816

Debugger executable	arm-none-eabi-gdb
Debugger timeout	10
SWV Packet Timeout	0
Extended debug trace (DEBUG_TRACE)	

- [ ] Stream all stub messages to Console
- [ ] Show stub warnings as notes
- [ ] Show debug log when written to
- [✓] Display asynchronous error messages
- [ ] Disable Auto-select device on multicore target
- [ ] Always show JTAG selection dialog
- [ ] Show progress messages in log
- [✓] Show extended debug messages
- [✓] Enable Non-Stop Mode

SWV Server Port: 0

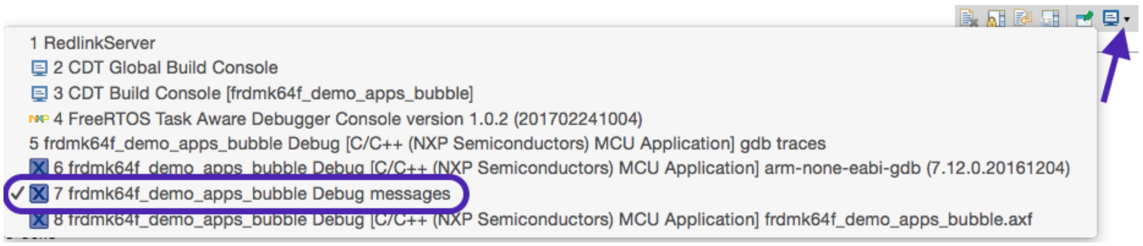
Figure 3.9. LinkServer Non Stop Preference

For a given project, the Non-Stop mode option is stored within the project's launch configuration. For projects that already have launch configurations, these will need to be deleted before proceeding.

3.5 LinkServer Troubleshooting

3.5.1 Debug Log

On occasion, it can be useful to explore the operations of a debug session in more detail. The steps are logged into a file known as the Debug log. This log will be displayed when a Debug operation begins, but by default, will be replaced by another view when execution starts. The debug log is a standard log within the IDE's Console view. To display this log, select the Console and then click to view the various options (as below):


1 RedlinkServer
2 CDT Global Build Console
3 CDT Build Console [frdmk64f_demo_apps_bubble]
4 FreeRTOS Task Aware Debugger Console version 1.0.2 (201702241004)
5 frdmk64f_demo_apps_bubble Debug [C/C++ (NXP Semiconductors) MCU Application] gdb traces
6 frdmk64f_demo_apps_bubble Debug [C/C++ (NXP Semiconductors) MCU Application] arm-none-eabi-gdb (7.12.0.20161204)
✓ 7 frdmk64f_demo_apps_bubble Debug messages
8 frdmk64f_demo_apps_bubble Debug [C/C++ (NXP Semiconductors) MCU Application] frdmk64f_demo_apps_bubble.axf

The debug log displays a large amount of information which can be useful in tracking down issues.

In the example debug log below, you can see that an initial Script file has been run. Connect scripts are required for debugging certain parts and are automatically added to launch configuration files by the IDE if required. Next, the hardware features of the MCU are captured and displayed, this includes the number of breakpoints and watchpoints available along with details of various hardware components indicating what debug features might be available, for example Instruction Trace.

Further down in this log you will see the selection of Flash driver (FTFE_4K), the identification of the part being debugged K64, and also the speed of Flash programming – in this case 80.59 KB/sec.

```
MCUXpresso RedlinkMulti Driver v10.1 (Sep 12 2017 14:57:42 - crt_emu_cm_redlink build 320)
Reconnected to existing redlink server (PID 4294967295)
Connecting to probe 2 core 0 (server PID unknown) gave 'OK'
============= SCRIPT: kinetisconnect.scp =============
Kinetis Connect Script
DpID = 2BA01477
Assert NRESET
Reset pin state: 00
Power up Debug
MDM-AP APID: 0x001C0000
MDM-AP System Reset/Hold Reset/Debug Request
MDM-AP Control: 0x0000001C
MDM-AP Status (Flash Ready) : 0x00000032
Part is not secured
MDM-AP Control: 0x00000014
Release NRESET
Reset pin state: 01
MDM-AP Control (Debug Request): 0x00000004
MDM-AP Status: 0x0001003A
MDM-AP Core Halted
============= END SCRIPT =============================
Probe Firmware: LPC-LINK2 CMSIS-DAP V5.183 (NXP Semiconductors)
Serial Number:  JQDYI3CT
VID:PID:  1FC9:0090
USB Path: \\?\hid#vid_1fc9&pid_0090&mi_00#9&23920c07&0&0000#{4d1e55b2-f16f-11cf-88cb-0011110030}
debug interface type      = <unknown> (DAP DP ID 2BA01477) over <error>
processor type            = Cortex-M4 (CPU ID 410FC240)
number of h/w breakpoints = 6
number of flash patches   = 2
number of h/w watchpoints = 4
Probe(0): Connected&Reset. DpID: 2BA01477. CpuID: 410FC240. Info: <None>
Debug protocol: SWD. RTCK: Disabled. Vector catch: Disabled.
Content of CoreSight Debug ROM(s):
RBASE E00FF000: CID B105100D PID 04000BB4C4 ROM dev (type 0x1)
ROM 1 E000E000: CID B105E00D PID 04000BB00C ChipIP dev SCS (type 0x0)
ROM 1 E0001000: CID B105E00D PID 04003BB002 ChipIP dev DWT (type 0x0)
ROM 1 E0002000: CID B105E00D PID 04002BB003 ChipIP dev FPB (type 0x0)
ROM 1 E0000000: CID B105E00D PID 04003BB001 ChipIP dev ITM (type 0x0)
ROM 1 E0040000: CID B105900D PID 04000BB9A1 CoreSight dev TPIU type 0x11 Trace Sink - TPIU
ROM 1 E0041000: CID B105900D PID 04000BB925 CoreSight dev ETM type 0x13 Trace Source - core
ROM 1 E0042000: CID B105900D PID 04003BB907 CoreSight dev ETB type 0x21 Trace Sink - ETB
ROM 1 E0043000: CID B105900D PID 04001BB908 CoreSight dev CSTF type 0x12 Trace Link - Trace/
funnel/router
Inspected v.2 On chip Kinetis Flash memory module FTFE_4K.cfx
Image 'Kinetis SemiGeneric Feb 17 2017 17:24:02'
Opening flash driver FTFE_4K.cfx
Sending VECTRESET to run flash driver
flash variant 'K 64 FTFE Generic 4K' detected (1MB = 256*4K at 0x0)
Closing flash driver FTFE_4K.cfx
NXP: MK64FN1M0xxx12
Connected: was_reset=true. was_stopped=true
MCUXpressoPro Full License - Unlimited
Awaiting telnet connection to port 3330 ...
GDB nonstop mode enabled
Opening flash driver FTFE_4K.cfx (already resident)
Sending VECTRESET to run flash driver
```

```
Writing 26324 bytes to address 0x00000000 in Flash
Erased/Wrote page  0-6 with 26324 bytes in 319msec
Closing flash driver FTFE_4K.cfx
Flash Write Done
Flash Program Summary: 26324 bytes in 0.32 seconds (80.59 KB/sec)
System reset and halt target
Stopped: Breakpoint #1
```

3.5.2 Flash Programming

Most debug sessions begin with a Flash programming operation, if this should fail, then the debug operation will be aborted.

Flash programming common operations:

1. Mass Erase: a mass erase will reset all the bytes in Flash (usually to 0xff). Such an operation may clear any internal low level structuring such as protection of Flash areas (from programming).
2. Sector Erase: internally Flash devices are divided into a number of sectors, where a sector is the smallest size of Flash that can be erased in a single operation. A sector will be larger than a page (see below). Sectors are usually the same size for the whole Flash device, however this is not always the case. A sector base address will be aligned on a boundary that is a multiple of its size.
3. Page Program: internally Flash devices are divided into a number of pages, where a page is the smallest size of Flash that can be programmed in a single operation. A page will be smaller than a sector. A page base addresses will be aligned on a boundary that is a multiple of its size.

A programming operation comprises repeated operations of sector erase followed by a set of program page operations; until the sector is fully programmed or there is no more data to program.

One of the common problems when programming Kinetis parts relates to their use of Flash configuration block at offset 0x400. For more information please see: Kinetis MCUs Flash Configuration Block [119] . Flash sector sizes on Kinetis MCUs range from 1KB to 8KB, therefore the first Sector Erase performed will clear the value of this block to all 0xFFs, if this is not followed by a successful program operation and the part is reset, then it will likely report as ‘Secured’ and subsequent debugging will not be possible until the part is recovered.

Such an event can occur if a debug operation is accidently performed to the ‘wrong board’ and a wrong Flash programmer is invoked.

To Recover a ‘locked’ part please see the section LinkServer GUI Flash programmer [93]

3.5.3 LinkServer executables

LinkServer debug operations rely on 3 main debug executables.

- **arm-none-eabi-gdb** – this is a version of GDB built to target ARM based MCUs
- **crt_emu_cm_redlink** – this executable (known as the debug stub) communicates with GDB and passes low level commands to the LinkServer executable (also known as redlink server)
- **redlinkserv** – this is the LinkServer executable and takes stub operations and communicates directly with the ARM Cortex debug hardware via the debug probe.

If a debug operation fails, or a crash occurs, it is possible that one or more of these processes will fail to shut down. Therefore, if the IDE has no active debug connection but is experiencing problems making a new debug connection, ensure that none of these executables is running.

3.6 P&E Debug Connections

P&E Micro software and drivers are automatically installed when MCUXpresso IDE installs. There is no need to perform any additional setup to use P&E Micro debug connections.

Currently we have tested using:

- Multilink Universal (FX)
- Cyclone Universal (FX) (USB and Ethernet)
- P&E firmware installed into on-board OpenSDA debug probe hardware (as shipped by default on certain Kinetis FRDM and TWR boards)

Note: Some Kinetis boards ship with OpenSDA supporting P&E VCOM but with no debug support. To update this firmware visit the OpenSDA Firmware Update pages linked at: *Help -> Additional Resources -> OpenSDA Firmware Updates*

3.7 P&E Debug Operation

The process to debug via a P&E compatible debug probe is exactly the same as for a native LinkServer (CMSIS-DAP) compatible debug probe. Simply select the project via the 'Project Explorer' view then click Debug from the **QuickStart** panel and select the P&E debug probe from the Probe Discovery Dialogue.

If more than one debug probe is presented, select the required probe and then click 'OK' to start the debug session. At this point, the projects launch configuration files will be created. **Note**: P&E Launch configuration files will contain the string 'PE'.

MCUXpresso IDE stores the probe information, along with its serial number in the projects launch configuration. This mechanism is used to match any attached probe when an existing launcher configuration already exits.

To simplify debug operations, MCUXpresso IDE will automatically start P&E's GDB Server and select and dynamically assign the various ports needed as required. This means that multiple P&E debug connections can be started, terminated, restarted etc. all without the need for any user connection configuration. These options can be controlled if required by editing the P&E launch configuration file.

For more information see Common Debugging Operations [76]

Note: If the project already had a P&E launch configuration, this will be selected and used. If they are no longer appropriate for the intended connection, simply delete the files and allow new launch configuration files to be created.

Important Note: Low level debug operations via P&E debug probes are supported by P&E software. This includes, Part Support handling, Flash Programming, and many other features. If problems are encountered, P&E Micro maintain a range of support forums at http://www.pemicro.com/forums/

3.7.1 P&E Differences from LinkServer Debug

MCUXpresso IDE core technology is intended to provide a seamless environment for code development and debug.

When used with P&E debug probes, the debug environment is provided by the P&E debug server. This debug server does not 100% match the features provided by native LinkServer connections. However basic debug operations will be very similar to LinkServer debug.

For a description of some common debugging operations using supported debug probes see Common Debugging Operations [76]

Note: LinkServer advanced features such as Instruction Trace, SWO Trace, Power Measurement, Live Global Variables etc. will not be available via a P&E debug connection.

3.7.2 P&E Micro Software Updates

P&E Micro support within MCUXpresso IDE is via an Eclipse Plugin. The P&E update site is automatically added to the list of Available Software Update sites.

To check whether an update is available, please select:

Help -> Check for Updates

Any available updates from P&E will then be listed for selection and installation.

Note: P&E Micro may provide news and additional information on their website, for details see https://www.pemicro.com

3.8 SEGGER Debug Connections

SEGGER J-Link software and documentation pack is installed automatically with the MCUXpresso IDE Installation for each host platform. No user setup is required to use the SEGGER debug solution within MCUXpresso IDE.

Currently we have tested using:

- J-Link debug probes (USB and Ethernet)
- J-Link firmware installed into on-board OpenSDA debug probe hardware (as shipped by default on certain Kinetis FRDM and TWR boards)
- J-Link firmware installed onto LPC-Link2 debug hardware and LPCXpresso V2/V3 boards
  - for details see https://www.segger.com/lpc-link-2.html
  - also for firmware programming see http://www.nxp.com/LPCSCRYPT

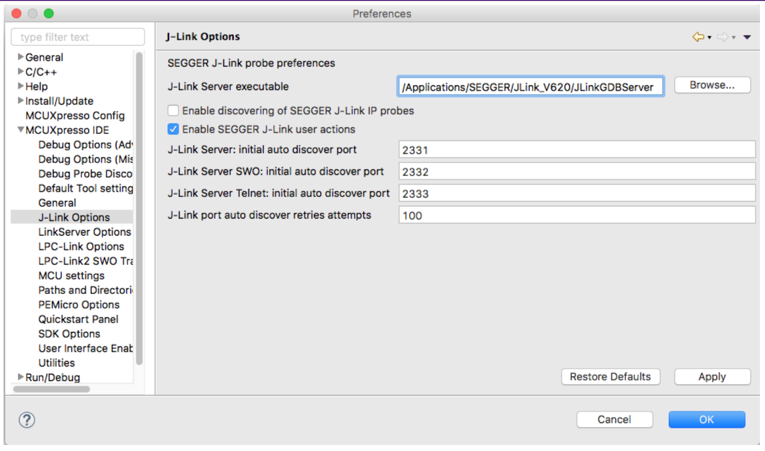
3.8.1 SEGGER software installation

Unlike other debug solutions supplied with MCUXpresso IDE, the SEGGER software installation is not integrated into the IDE installation, rather it is a separate SEGGER J-Link installation on your host.

The installation location will be similar to:

```
On Windows: C:/Program Files (x86)/SEGGER/JLink_V620
On Mac: /Applications/SEGGER/JLink_V620
On Linux: /opt/SEGGER/JLink
```

MCUXpressoIDE automatically locates the required executable and it is remembered as a Workspace preference. This can be viewed or edited within the MCUXpresso IDE preferences as below.



J-Link Options

SEGGER J-Link probe preferences

J-Link Server executable: /Applications/SEGGER/JLink_V620/JLinkGDBServer

[ ] Enable discovering of SEGGER J-Link IP probes

[✓] Enable SEGGER J-Link user actions

J-Link Server: initial auto discover port: 2331

J-Link Server SWO: initial auto discover port: 2332

J-Link Server Telnet: initial auto discover port: 2333

J-Link port auto discover retries attempts: 100

Figure 3.10. Segger Preferences

Note: this preference also provides the option to enable scanning for SEGGER IP probes (when a probe discovery operation is performed). By default, this option is disabled.

From time to time, SEGGER may release later versions of their software, which the user could choose to manually install. For details see https://www.segger.com/downloads/jlink

MCUXpresso IDE will continue to use the SEGGER installation path as referenced in a projects workspace unless the required executable cannot be found (for example, the referenced installation has been deleted). If this occurs:

1. The IDE will automatically search for the latest installation it can find. If this is successful, the Workspace preference will automatically be updated
2. If a SEGGER installation cannot be found, the user will be prompted to located an installation

To force a particular workspace to update to use a newer installation location simply click the *Restore Default* button.

To permanently select a particular SEGGER installation version, the location of the SEGGER GDB Server can be stored in an environment variable.

For example, under Windows you could set:

```
MCUX_SEGGER_SERVER="C:/Program Files (x86)/SEGGER/JLink_V612i/jLinkGDBServerCL.exe"
```

This location will then be used, overriding any workspace preference that maybe set.

SEGGER software un-installation

If MCUXpresso IDE is uninstalled, it will not remove the SEGGER J-Link installation. If this is required, then the user must manually uninstall the SEGGER J-Link tools.

Note: If for any reason MCUXpresso IDE cannot locate the SEGGER J-Link software, then the IDE will prompt the user to either manually locate an installation or disable the further use of the SEGGER debug solution.

3.9 SEGGER Debug Operation

The process to debug via a J-Link compatible debug probe is exactly the same as for a native LinkServer (CMSIS-DAP) compatible debug probe. Simply select the project via the 'Project

Explorer' view then click Debug from the **QuickStart** Panel and select the SEGGER Probe from the Probe Discovery Dialogue.

If more than one debug probe is presented, select the required probe and then click 'OK' to start the debug session. At this point, the projects launch configuration files will be created. **Note**: SEGGER Launch configuration files will contain the string 'JLink'.

To simplify debug operations, MCUXpresso IDE will automatically start SEGGER's GDB Server and select and dynamically assign the various ports needed as required. This means that multiple SEGGER debug connections can be started, terminated, restarted etc. all without the need for any user connection configuration. These options can be controlled if required by editing the SEGGER launch configuration file.

In MCUXpresso IDE, SEGGER Debug operations default to using the SWD Target Interface. When debugging certain multicore parts such as the LPC43xx Series, the JTAG Target Interface must be used to access the internal Slave MCUs. To select JTAG as the Target Interface, simply edit the SEGGER launch configuration file and select JTAG.

For more information see Common Debugging Operations [76]

Note: If the project already had a SEGGER launch configuration, this will be selected and used. If an existing launch configuration file is no longer appropriate for the intended connection, simply delete the files and allow new launch configuration files to be created.

Important Note: Low level debug operations via SEGGER debug probes are supported by SEGGER software. This includes, Part Support handling, Flash Programming, and many other features. If problems are encountered, SEGGER's provide a range of support forums at http://forum.segger.com/

3.9.1 SEGGER Differences from LinkServer Debug

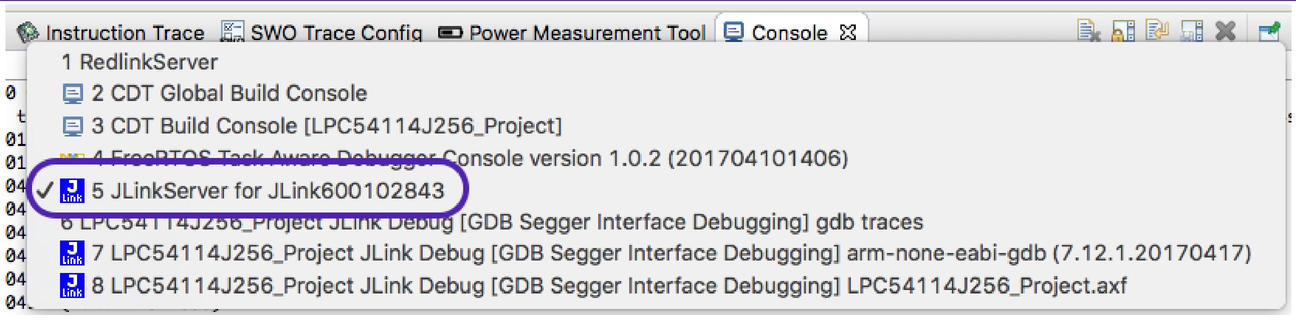
MCUXpresso IDE core technology is intended to provide a seamless environment for code development and debug. When used with SEGGER debug probes, the debug environment is provided by the SEGGER debug server. This debug server does not 100% match the features provided by native LinkServer connections. However basic debug operations will be very similar to LinkServer debug.

For a description of some common debugging operations using supported debug probes see Common Debugging Operations [76]

Note: LinkServer features such as Instruction Trace, SWO Trace, Power Measurement, Live Global Variables etc. will not be available via a SEGGER debug connection. However, additional functionality maybe available using external SEGGER supplied applications.

3.10 SEGGER Troubleshooting

When a debug operation to a SEGGER debug probe is performed, the SEGGER GDB server is called with a set of arguments provided by the launch configuration file. The command and resulting output is logged within the IDE SEGGER Debug Console. The console can be viewed as below:



1 RedlinkServer
2 CDT Global Build Console
3 CDT Build Console [LPC54114J256_Project]
4 FreeRTOS Task Aware Debugger Console version 1.0.2 (201704101406)
✓ 5 JLinkServer for JLink600102843
6 LPC54114J256_Project JLink Debug [GDB Segger Interface Debugging] gdb traces
7 LPC54114J256_Project JLink Debug [GDB Segger Interface Debugging] arm-none-eabi-gdb (7.12.1.20170417)
8 LPC54114J256_Project JLink Debug [GDB Segger Interface Debugging] LPC54114J256_Project.axf

Figure 3.11. Segger Server

The command can be copied and called independently of the IDE to start a debug session and explore connection issues.

Below is the shortened output of a successful debug session to an LPC54114 LPCXpresso Board.

```
[19-9-2017 04:25:13] Executing Server: /Applications/SEGGER/JLink_V618a/
JLinkGDBServer  -nosilent -swoport 2355 -select USB=600102843 -telnetport 2356/
-singlerun -endian little -noir -speed auto   -port 2357  -vd /
-device LPC54114J256_M4 -if SWD  -halt -reportuseraction

SEGGER J-Link GDB Server V6.18a Command Line Version

JLinkARM.dll V6.18a (DLL compiled Aug 11 2017 17:53:42)

Command line: -nosilent -swoport 2355 -select USB=600102843 -telnetport 2356 -singlerun/
-endian little -noir -speed auto -port 2357 -vd -device LPC54114J256_M4 -if SWD -halt /
-reportuseraction

-----GDB Server start settings-----
GDBInit file:                  none
GDB Server Listening port:     2357
SWO raw output listening port: 2355
Terminal I/O port:             2356
Accept remote connection:      yes
Generate logfile:              off
Verify download:               on
Init regs on start:            off
Silent mode:                   off
Single run mode:               on
Target connection timeout:     0 ms
------J-Link related settings------
J-Link Host interface:         USB
J-Link script:                 none
J-Link settings file:          none
------Target related settings------
Target device:                 LPC54114J256_M4
Target interface:              SWD
Target interface speed:        auto
Target endian:                 little

Connecting to J-Link...
J-Link is connected.
Device "LPC54114J256_M4" selected.
```

```
Firmware: J-Link V10 compiled Sep  8 2017 17:24:33
Hardware: V10.10
S/N: 600102843
Feature(s): RDI, FlashBP, FlashDL, JFlash, GDB
Checking target voltage...
Target voltage: 3.28 V
Listening on TCP/IP port 2357
...
Connected to target
Waiting for GDB connection...Connected to 127.0.0.1
Reading all registers
Read 4 bytes @ address 0x0000046E (Data = 0x4A023301)
Received monitor command: reset
Reset: Halt core after reset via DEMCR.VC_CORERESET.
Reset: Reset device via AIRCR.SYSRESETREQ.
Resetting target
Downloading 13796 bytes @ address 0x00000000 - Verified OK
Downloading 4 bytes @ address 0x000035E4 - Verified OK
J-Link: Flash download: Bank 0 @ 0x00000000: 1 range affected (32768 bytes)
J-Link: Flash download: Total time needed: 0.358s (Prepare: 0.065s, Compare: 0.056s,/
Erase: 0.111s, Program: 0.117s, Verify: 0.002s, Restore: 0.004s)
Writing register (PC = 0x00000160)
...
Read 4 bytes @ address 0x00000160 (Data = 0x480F2501)
...
Reset: Reset device via AIRCR.SYSRESETREQ.
Resetting target
Received monitor command: semihosting enable
Semi-hosting enabled (Handle on BKPT)
Received monitor command: exec SetRestartOnClose=1
Executed SetRestartOnClose=1
Setting breakpoint @ address 0x00000454, Size = 2, BPHandle = 0x0001
Starting target CPU...
...Breakpoint reached @ address 0x00000454
Reading all registers
Read 4 bytes @ address 0x00000454 (Data = 0xF8D9F000)
Removing breakpoint @ address 0x00000454, Size = 2
```

Note: If a SEGGER debug operation is not successful, the IDE will generate an error dialogue, the *'Details'* button can be clicked to display a copy of the SEGGER server log. One possible reason for a SEGGER debug operation to fail is due to a Device name mismatch. MCUXpresso IDE will try to supply the expected Device name to SEGGER server, however on rare occasions this may not be the name expected. The SEGGER launch configuration Device entry can be populated via a drop down list or via a user supplied device name.

If required, additional server options can be set within the SEGGER launch configuration. For example to capture logging information to a file, you can set the additional server option:

```
-log $(CWD)/my.log
```

where *$(CWD)* represents the current working directory of the debug connection, i.e. the dynamically created project build configuration folder.

4. SDKs and Preinstalled Part Support Overview

To support a particular MCU (or family of MCUs), a number of elements are required. These break down into:

- Startup code
  - This code will handle specific features required by the MCU
- Memory Map knowledge
  - The addresses and sizes of all memory regions
- Peripheral knowledge
  - Detailed information allowing the MCUs peripherals registers to be viewed and edited
- Flash Drivers
  - Routines to program the MCU’s on and off chip Flash devices as efficiently as possible
- Debug capabilities
  - Knowledge of the MCU debug interfaces and features (e.g. SWO, ETB)
- Example Code
  - Code to demonstrate the features of the particular MCU and supporting drivers

MCUXpresso IDE uses these data elements for populating its wizards, and for built in intelligence features, such as the automatic generation of linker scripts etc.

MCUXpresso IDE delivers its part support through an extensible scheme.

4.1 Preinstalled Part Support

Firstly the IDE installs with an enhanced version of the part support as provided with LPCXpresso IDE v8.2.2. This provides support for the majority of LPC Cortex-M based parts ‘out of the box’. This is known as preinstalled part support.

Example code for these preinstalled parts is provided by sophisticated LPCOpen packages (and Code Bundles). Each of these contains code libraries to support the MCU features, LPCXpresso boards (and some other popular ones), plus a large number of code examples and drivers. Version of these are installed by default at:

```
<install dir>/ide/Examples/LPCOpen
<install dir>/ide/Examples/CodeBundles
```

Further information can be founds at:

http://www.nxp.com/lpcopen

http://www.nxp.com/LPC800-Code-Bundles

4.2 SDK Part Support

Secondly, MCUXpresso IDE’s part support can be extended using freely available MCUXpresso SDK v2.x packages. These can be installed via a simple ‘drag and drop’ and automatically extend the IDE with new part knowledge and examples.

SDKs for MCUXpresso IDE can be generated and downloaded as required using the SDK Builder on the MCUXpresso Config Tools website at:

http://mcuxpresso.nxp.com/

Support for Kinetis devices is delivered by SDK 2.x packages, In addition, this mechanism will be used to offer support for new LPC MCUs from NXP such as the LPC54608J512.

Once an SDK has been installed, the included part support becomes available through the New Project Wizard and also the SDK example import Wizard.

Important Note: Only SDKs built specifically for **MCUXpresso IDE** are compatible with MCUXpresso IDE. **SDKs created for any other toolchain will not work!** Therefore, when generating an SDK be sure that MCUXpresso IDE is specified as the Toolchain.

4.2.1 Differences in Preinstalled and SDK part handling

Since SDKs bundle part (MCU) and board support into a single package, MCUXpresso IDE is able to provide linkage between SDK installed MCUs and their related boards when creating or importing projects.

For preinstalled parts, the board support libraries are provided within LPCOpen packages and Code Bundles. It is the responsibility of user to match an MCU with its related LPCOpen board and chip library when creating or importing projects.

Creating and importing project using Preinstalled and SDK part support is described in the following chapters.

4.3 Viewing Preinstalled Part Support

When MCUXpresso IDE is installed, it will contain preinstalled part support for most LPC based MCUs.

To explore the range of preinstalled MCUs simply click ‘New project’ in the **QuickStart panel**. This will open a page similar to the image below:

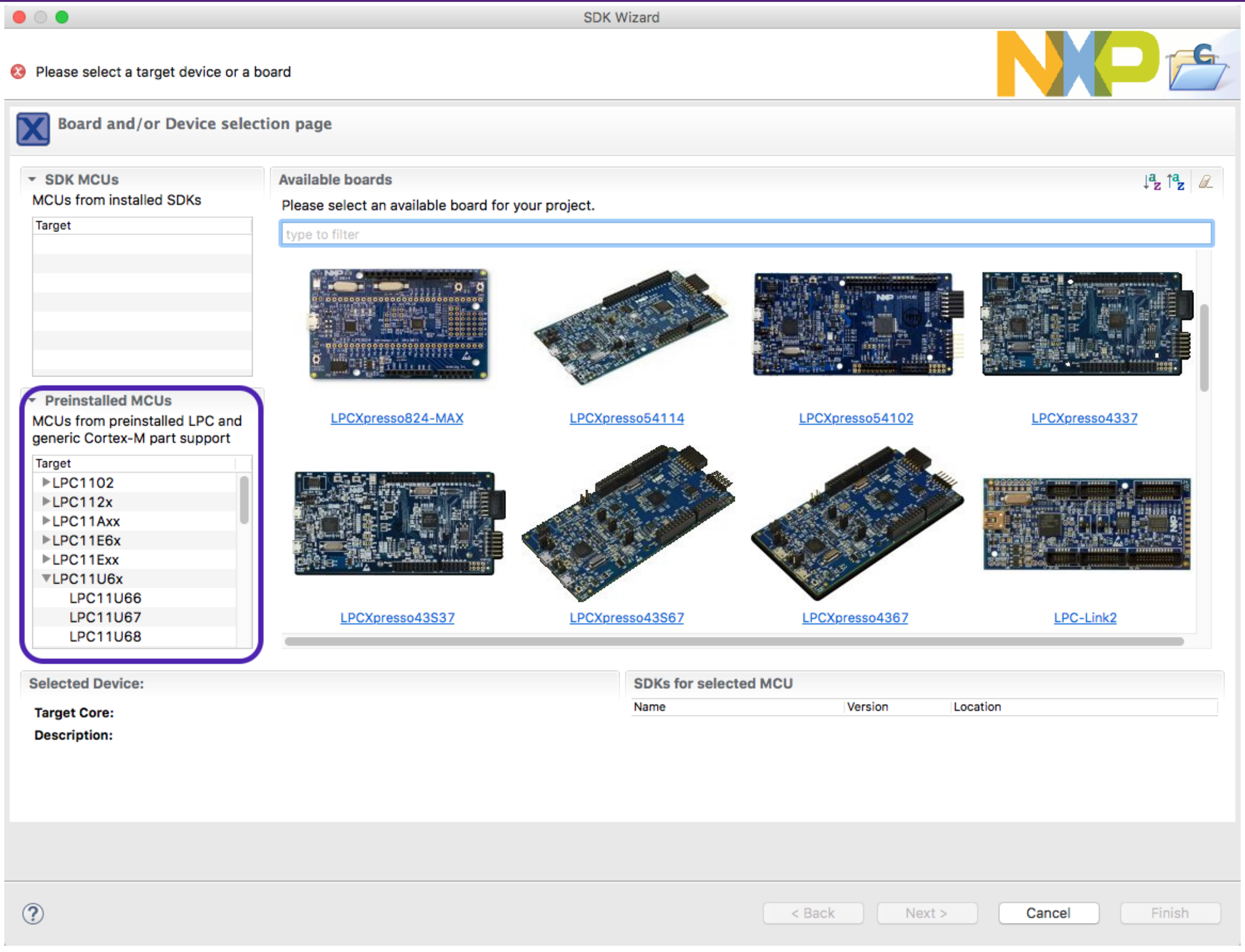


Figure 4.1. New Project Wizard

The list of preinstalled parts is presented on the bottom left of this window.

You will also see a range of related development boards indicating whether a matching LPCOpen Library or Code Bundle is available.

For creating project with preinstalled part support please see: Creating Projects with Preinstalled part support [57]

If you intend to work on an MCU that is not available from the range of preinstalled parts, for example a Kinetis MCU, then you must first extend the part support of MCUXpresso IDE by installing the appropriate MCU SDK.

4.4 Installing an SDK

The process to follow is simple, first download the SDK package, then install this into MCUXpresso IDE.

The easiest way to do this is to switch to the “Installed SDKs” view within the MCUXpresso IDE console view (highlighted below).

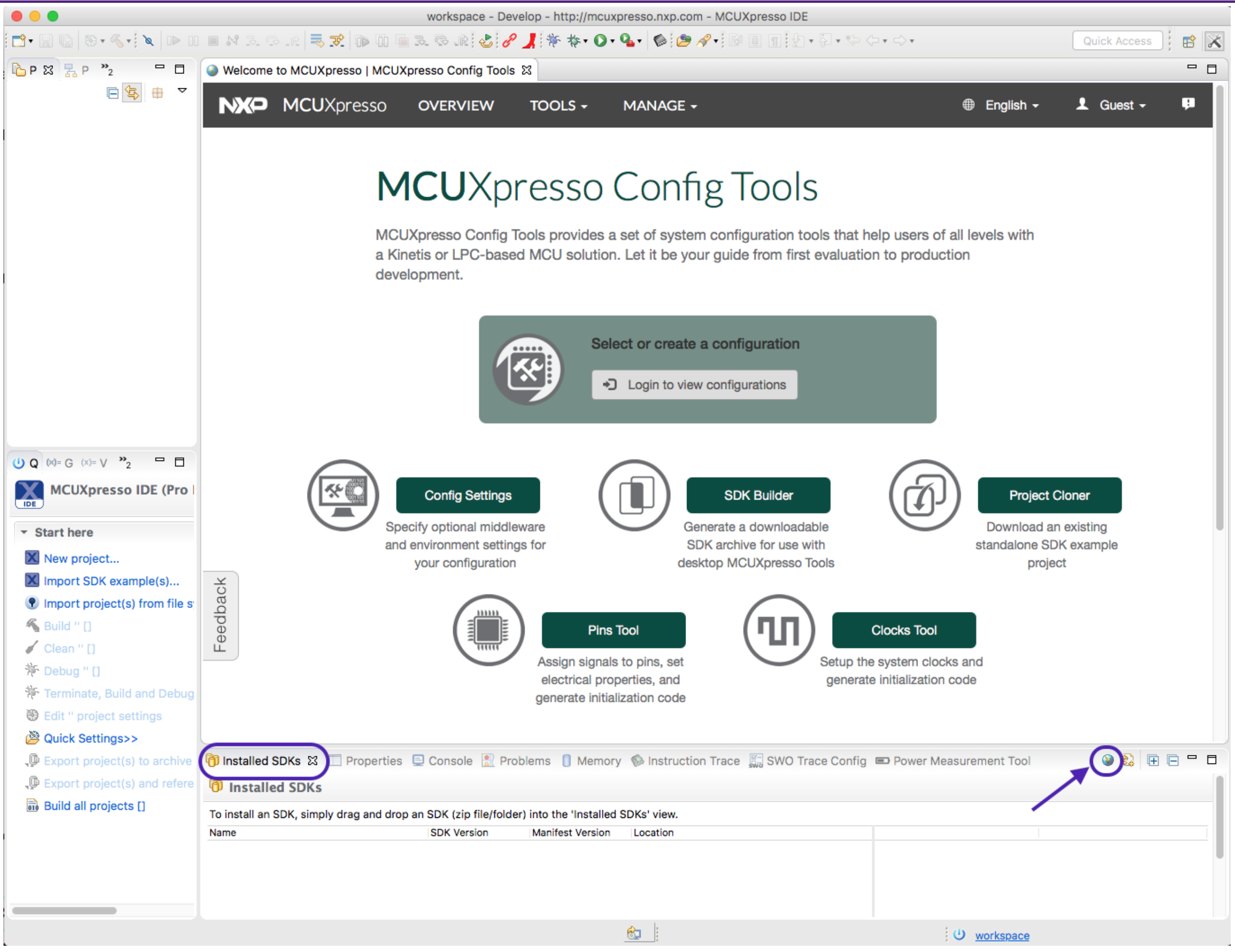

workspace - Develop - http://mcuxpresso.nxp.com - MCUXpresso IDE
Welcome to MCUXpresso | MCUXpresso Config Tools
MCUXpresso
OVERVIEW
TOOLS
MANAGE
English
Guest
MCUXpresso Config Tools
MCUXpresso Config Tools provides a set of system configuration tools that help users of all levels with a Kinetis or LPC-based MCU solution. Let it be your guide from first evaluation to production development.
Select or create a configuration
Login to view configurations
Config Settings
Specify optional middleware and environment settings for your configuration
SDK Builder
Generate a downloadable SDK archive for use with desktop MCUXpresso Tools
Project Cloner
Download an existing standalone SDK example project
Pins Tool
Assign signals to pins, set electrical properties, and generate initialization code
Clocks Tool
Setup the system clocks and generate initialization code
Feedback
MCUXpresso IDE (Pro
Start here
New project...
Import SDK example(s)...
Import project(s) from file s
Build '' []
Clean '' []
Debug '' []
Terminate, Build and Debug
Edit '' project settings
Quick Settings>>
Export project(s) to archive
Export project(s) and refere
Build all projects []
Installed SDKs
Properties
Console
Problems
Memory
Instruction Trace
SWO Trace Config
Power Measurement Tool
Installed SDKs
To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view.
Name
SDK Version
Manifest Version
Location
workspace

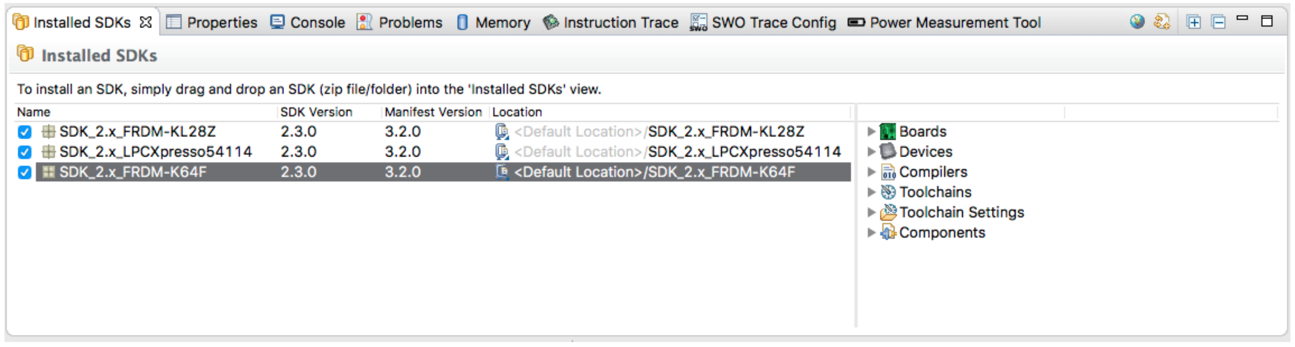
Figure 4.2. SDK Import

SDKs are free to download (login is required); MCUXpresso IDE offers a link to the SDK portal from the Installed SDK Console view (as indicated above). If required, the necessary SDK can be downloaded onto the host machine.

To install an SDK, simply open a Windows Explorer / filer onto the directory containing the SDK package(s), then select the ZIP file(s) and drag them into the “Installed SDKs” view.

You will then be prompted with a dialog asking you to confirm the import – click OK. The SDK or SDKs will then be automatically installed into MCUXpresso IDE part support repository.

Once complete the “Installed SDKs” view will update to show you the package(s) that you have just installed.



Installed SDKs | Properties | Console | Problems | Memory | Instruction Trace | SWO Trace Config | Power Measurement Tool

Installed SDKs

To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view.

Name	SDK Version	Manifest Version	Location
✓ SDK_2.x_FRDM-KL28Z	2.3.0	3.2.0	<Default Location>/SDK_2.x_FRDM-KL28Z
✓ SDK_2.x_LPCXpresso54114	2.3.0	3.2.0	<Default Location>/SDK_2.x_LPCXpresso54114
✓ SDK_2.x_FRDM-K64F	2.3.0	3.2.0	<Default Location>/SDK_2.x_FRDM-K64F

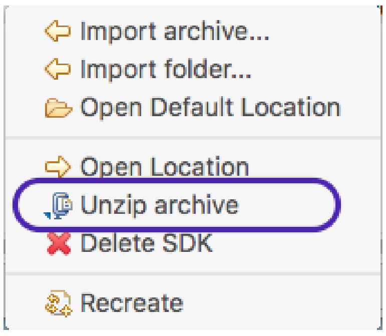
Figure 4.3. SDK Import View

Notes:

- Released in parallel with MCUXpresso IDE version 10.1.0 are updated SDKs (MCUXpressoSDK v2.3.0). These are indicated by their version 2.3.0 and a manifest version 3.2.0 in the Installed SDK view. While older SDKs are still compatible with MCUXpresso IDE version 10.1.0, it is recommended that users check they are using the latest available SDK package.
- MCUXpresso IDE can import an SDK as a zipped package or unzipped folder. Typically importing as a zipped package is expected.
  - The main consequence of leaving SDKs zipped is that you will not be able to create (or import projects) into a workspace with linked references back to the SDK source files.
- When an SDK is imported via drag and drop, required files are copied and the original file/folder is unaffected. This installs data into a default location allowing imported SDKs to be shared among different IDE instances/installations and workspaces.
  - By default, SDKs (like workspaces) are located in user local storage, this means they will only be available to the user who performed the installation. The next section shows how a public location could be used if needed.

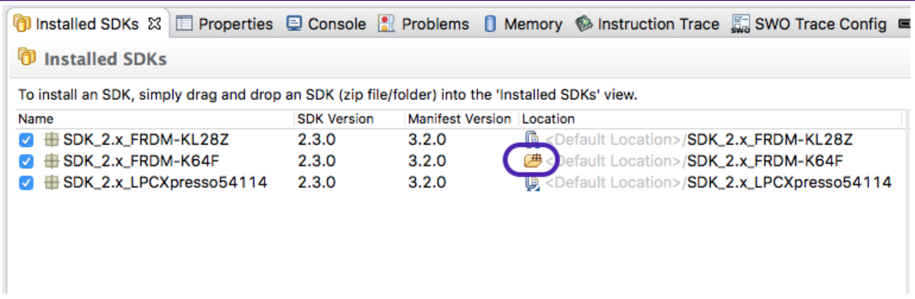
4.4.1 Installed SDKs Operations

The Installed SDKs display will show whether the SDKs are stored as zipped archives or regular folders. MCUXpresso IDE offers the option to unzip an archive in place via a right click option onto the selected SDK (as below).



Note: Unzipping an SDK may take some time and is generally not needed unless you wish to make use of referenced files or perform many example imports (where some speed improvement will be seen).

Once an SDK has been unzipped, its icon will be updated to reflect that it is now stored internally as a folder.


Installed SDKs | Properties | Console | Problems | Memory | Instruction Trace | SWO Trace Config

Installed SDKs

To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view.

Name	SDK Version	Manifest Version	Location
✓ SDK_2.x_FRDM-KL28Z	2.3.0	3.2.0	<Default Location>/SDK_2.x_FRDM-KL28Z
✓ SDK_2.x_FRDM-K64F	2.3.0	3.2.0	<Default Location>/SDK_2.x_FRDM-K64F
✓ SDK_2.x_LPCXpresso54114	2.3.0	3.2.0	<Default Location>/SDK_2.x_LPCXpresso54114

Figure 4.4. SDK Unzipped

Deleting an Installed SDK

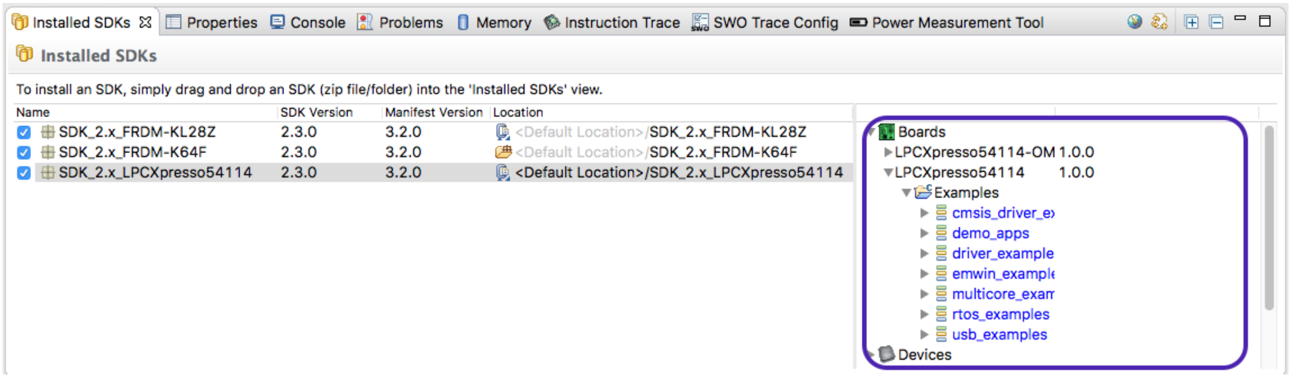
If an SDK has been installed by the ‘Drag and Drop’ method, then a copy of the SDK will have been installed into the *Default Location*. SDKs installed into this location can be deleted via a right click option. Once an SDK has been deleted, then part support will automatically be recreated for the remaining SDKs. Pleae see Uninstallation Considerations [36] for more information.

Along side each installed SDK is a check box, if this is unchecked the SDK will be hidden from MCUXpresso IDE until re-checked. If multiple SDKs are installed that contain shared part support, then this feature may be useful to force the selection of part support from a particular SDK. Please see Shared Part Support Handling [35] for more information.

SDKs installed into non default locations, must be manually deleted or hidden if they are no longer required. Note: you may have to quit MCUXpresso IDE to delete these SDKs. Please see SDK Importing and Configuration [33] for more information.

4.4.2 Installed SDKs Features

You can explore each of the SDKs within the “Installed SDKs” view to examine its contents as below:


Installed SDKs | Properties | Console | Problems | Memory | Instruction Trace | SWO Trace Config | Power Measurement Tool

Installed SDKs

To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view.

Name	SDK Version	Manifest Version	Location
✓ SDK_2.x_FRDM-KL28Z	2.3.0	3.2.0	<Default Location>/SDK_2.x_FRDM-KL28Z
✓ SDK_2.x_FRDM-K64F	2.3.0	3.2.0	<Default Location>/SDK_2.x_FRDM-K64F
✓ SDK_2.x_LPCXpresso54114	2.3.0	3.2.0	<Default Location>/SDK_2.x_LPCXpresso54114

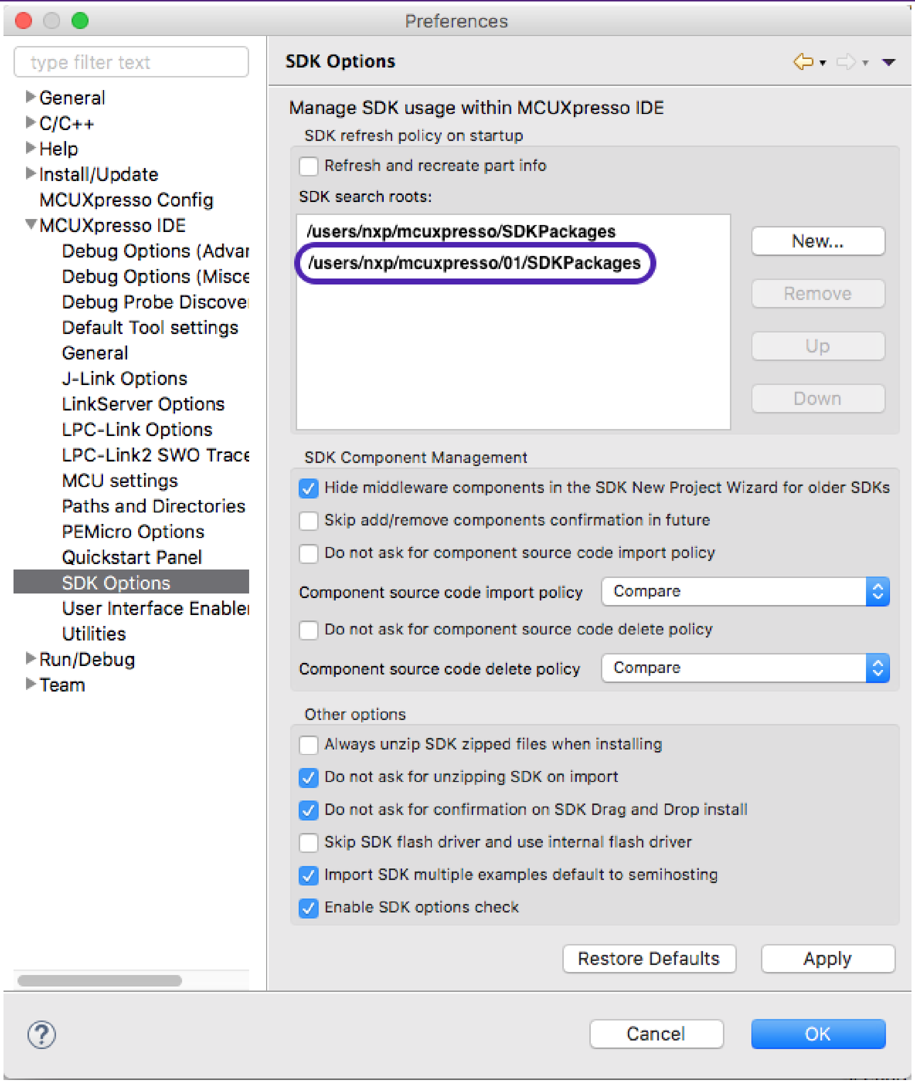
```
Boards
  ▶LPCXpresso54114-OM 1.0.0
  ▼LPCXpresso54114     1.0.0
    ▼Examples
       ▶ cmsis_driver_e
       ▶ demo_apps
       ▶ driver_example
       ▶ emwin_example
       ▶ multicore_exam
       ▶ rtos_examples
       ▶ usb_examples
Devices
```

Figure 4.5. SDK Explore

4.4.3 Advanced Use: SDK Importing and Configuration

Although using the “Installed SDKs” view offers the most straight forward way of importing SDKs, MCUXpresso IDE also provides additional capabilities for importing and configuring its SDK usage.

If you go to *Preferences->MCUXpresso IDE->SDK Options* then the following window will appear:



Preferences

SDK Options

Manage SDK usage within MCUXpresso IDE

SDK refresh policy on startup

- [ ] Refresh and recreate part info

SDK search roots:

/users/nxp/mcuxpresso/SDKPackages
/users/nxp/mcuxpresso/01/SDKPackages

SDK Component Management

- [x] Hide middleware components in the SDK New Project Wizard for older SDKs
- [ ] Skip add/remove components confirmation in future
- [ ] Do not ask for component source code import policy

Component source code import policy: Compare

- [ ] Do not ask for component source code delete policy

Component source code delete policy: Compare

Other options

- [ ] Always unzip SDK zipped files when installing
- [x] Do not ask for unzipping SDK on import
- [x] Do not ask for confirmation on SDK Drag and Drop install
- [ ] Skip SDK flash driver and use internal flash driver
- [x] Import SDK multiple examples default to semihosting
- [x] Enable SDK options check

Figure 4.6. SDK Preferences

Note: from here you can see there are actually two default locations specified but the highlighted location will be used for new installations, see SDK compatibility with earlier versions of MCUXpresso IDE [35] for more information.

From here you can add paths to other folders where you have stored or plan to store SDK folders/zips. Those SDKs will appear in the Installed SDKs View along with those from the default location.

The main differences between having SDKs in the default location or leaving them in other folders are:

- "Delete SDK" function is disabled when using non-default locations
  - since these SDKs are not imported, they may be original files
- The knowledge of the SDKs and their part support is per-workspace

The order of the SDKs in the SDK location list may be important on occasion: if you have multiple SDKs for the same part in various locations, you can choose which to load by reordering. If multiple SDK are found, a warning is displayed into the Installed SDK view.

Note: Only the default SDK location(s) is persistent between workspaces. Any other locations must be created for each Workspace as required.

Other SDK options

Also from the previous dialogue, are two distinct sets of options; the first to control the handling of adding (and removing) SDK components. Please see the section SDK Project Component Management [53] for more information. The second set of 'other options' are a self describing set of features.

4.4.4 Important notes for SDK users

Only SDKs created for MCUXpresso IDE can be used

If an error of the form *MCUXpresso IDE was unable to load one or more SDKs* is seen, the most likely reason is that the SDK was not built for MCUXpresso IDE. Within the SDK Builder, verify that the Toolchain is set to MCUXpresso IDE. If necessary, reset the toolchain to MCUXpresso IDE and rebuild the SDK.

SDK compatibility with earlier versions of MCUXpresso IDE

As mentioned earlier, a new SDK version 2.3.0 has been released in parallel with MCUXpresso IDE version 10.1.0 however, this SDK format includes features that are not compatible with earlier versions of MCUXpresso IDE. As a result, these new SDKs will fail to install into older versions of MCUXpresso IDE.

To support users who might have both this and also older versions of MCUXpresso IDE installed on their system, we have adopted a new default SDK installation location but also maintained support for the default used by older versions (now effectively Read Only from version 10.1.0).

The result of this is that MCUXpresso IDE version 10.1.0 will automatically inherit any SDKs installed into the (old) default location by previous versions of the IDE. While older versions of the IDE will not 'see' any SDKs installed with MCUXpresso IDE version 10.1.0 or later.

Note: If there is no need to maintain compatibility with older versions of the IDE, it is recommended that users migrate to using these new SDKs where available.

Shared Part Support handling

Each SDK package will contain part support for one or more MCUs, therefore it is possible to have two (or more) SDK packages containing the same part support. For example, a user might request a Tower K64 SDK and later a Freedom K64 SDK that both target the same MK64FN1M0xxx12 MCU. If both SDKs are installed into the IDE, both sets of examples and board drivers will be available, but the IDE will select the most up to date version of part support specified within these SDKs. This means the various wizards and dialogues will only ever present a single instance of an MCU, but may offer a variety of compatible boards and examples. **Note**: If a board is selected (from one SDK) and part support is provided by another SDK, a message will displayed within the project wizard to show this has occurred but no user action is required.

If two SDKs with matching part support are installed, and the SDK providing part support later deleted, then part support will automatically be used from the remaining SDK.

Building a Fat SDK

An SDK can be generated for a selected part (processor type/MCU) or a board. If just a part is selected, then the generated SDK will contain both part support and also board support data for the closest matching development board.

Therefore, to obtain an SDK with both Freedom and Tower board support for say the Kinetis MK64... part, simply select the part and the board support will be added automatically.

If a part is chosen that has no directly matching board, say the Kinetis MK63… then the generated SDK will contain:

- part support for the requested part i.e. MK63...
- part support for the recommended closest matching part that has an associated development board i.e. MK64...
- board support packages for the above part i.e. Freedom and/or Tower MK64...

Uninstallation Considerations

MCUXpresso IDE allows SDKs to be installed and uninstalled as required (although for most users there is little benefit in uninstalling an SDK). However, since the SDK provides part support to the IDE, if an SDK is uninstalled, part support will also be removed. Any existing project built using part support from an uninstalled SDK will no longer build or debug. Such a situation can be remedied by re-installing the missing SDK. **Note**: if there is another SDK installed capable of providing the 'missing' part support, then this will automatically used.

Sharing Projects

If a project built using part support from an SDK and is then exported – for example to share the project with a colleague who also uses MCUXpresso IDE, then the colleague must also install an SDK providing part support for the projects MCU. **Note**: it is recommended that any required SDKs are installed before a project requiring SDK part support is imported. However, if this is not done, simply select the imported project in the project explorer and right click and select: *C/C++ Build -> MCU settings* ensure the correct MCU is selected and click **Refresh MCU Cache**. Please see the section Importing Example Projects [64]

5. Creating New Projects using installed SDK Part Support

For creating project using Preinstalled part support please see: Creating Projects using Preinstalled Part Support [57]

From the **QuickStart** Panel in the bottom left of the MCUXpresso IDE window there are two options:

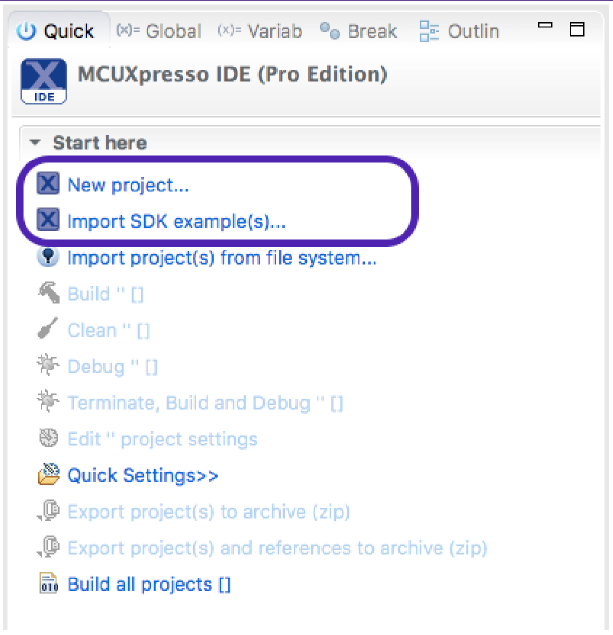

Quick
Global
Variab
Break
Outlin
MCUXpresso IDE (Pro Edition)
Start here
New project...
Import SDK example(s)...
Import project(s) from file system...
Build '' []
Clean '' []
Debug '' []
Terminate, Build and Debug '' []
Edit '' project settings
Quick Settings>>
Export project(s) to archive (zip)
Export project(s) and references to archive (zip)
Build all projects []

Figure 5.1. SDK Projects

The first will invoke the **New Project Wizard**, that guides the user in creating new projects from the installed SDKs (and also from preinstalled part support – which will be discussed in a later chapter).

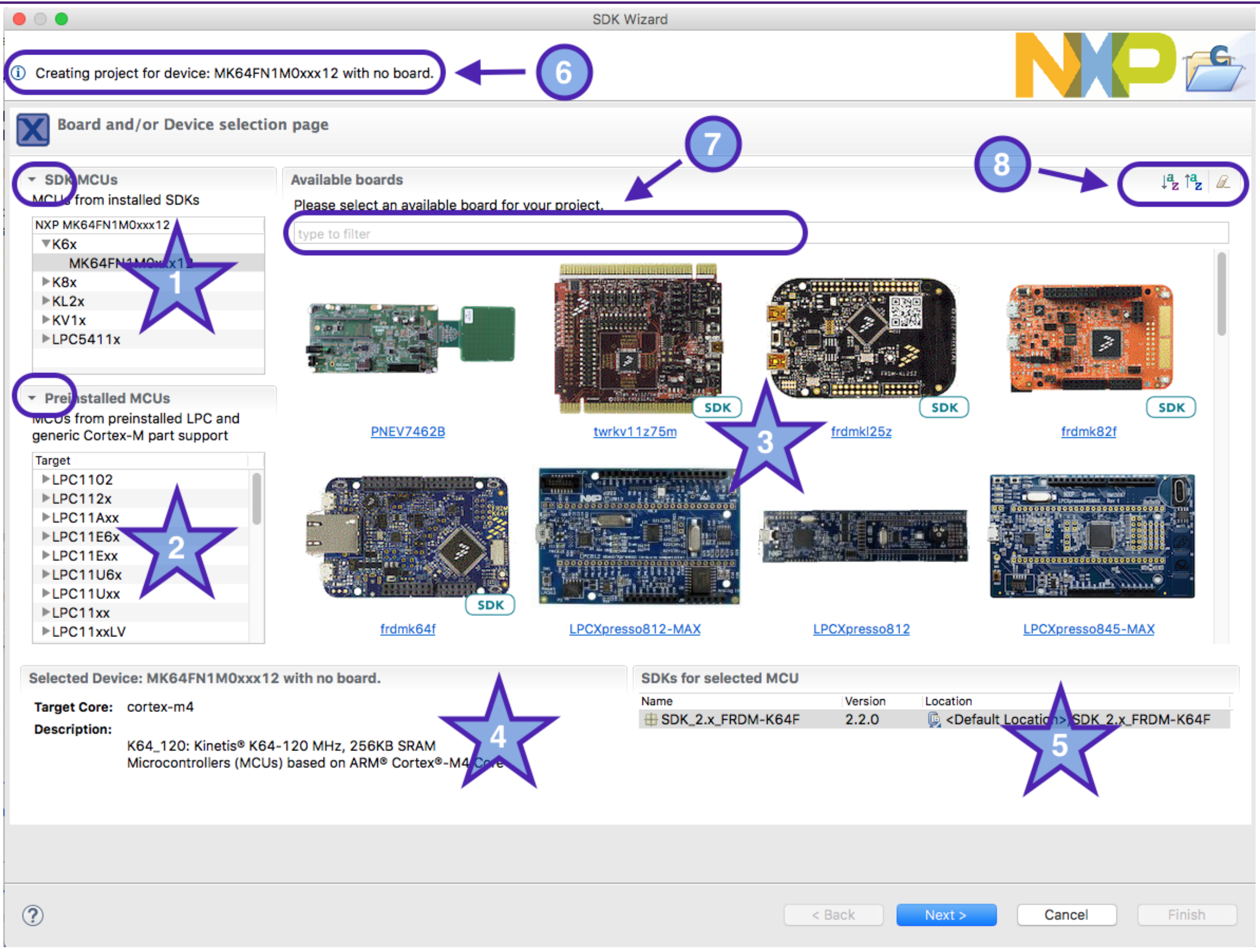
The second option invokes the **Import SDK Example Wizard** that guides the user to import SDK example projects from installed SDKs.

This option will be explored in the next chapter.

Click New project to launch the New Project Wizard.

5.1 New Project Wizard

The New Project Wizard will begin by opening the “Board and/or device selection” page, this page is populated with a range of features described below:



Creating project for device: MK64FN1M0xxx12 with no board.

Selected Device: MK64FN1M0xxx12 with no board.

Target Core: cortex-m4

Description: K64_120: Kinetis® K64-120 MHz, 256KB SRAM Microcontrollers (MCUs) based on ARM® Cortex®-M4 Core

Name	Version	Location
SDK_2.x_FRDM-K64F	2.2.0	<Default Location>/SDK_2.x_FRDM-K64F

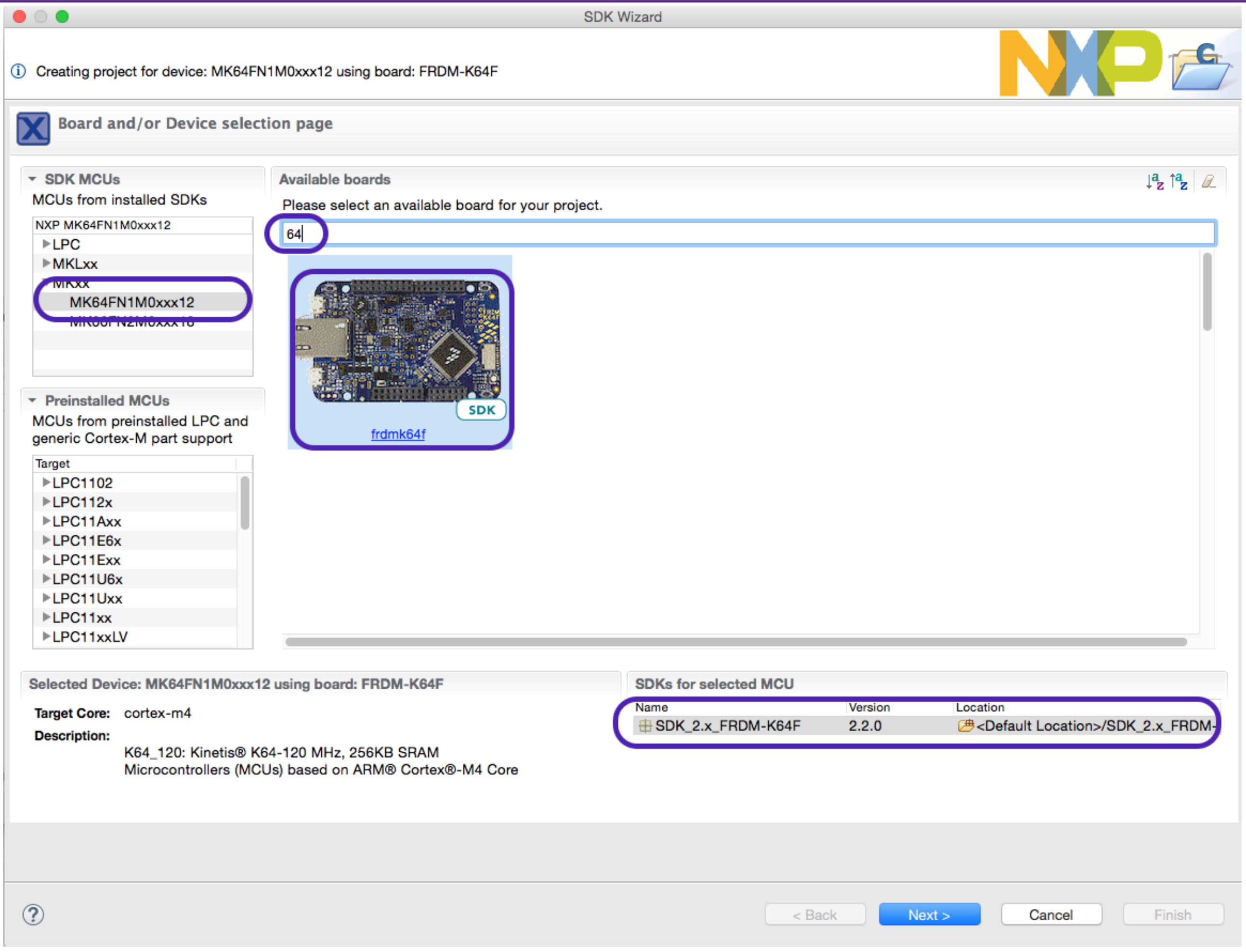
Figure 5.2. New Project Wizard first page

1. A display of all parts (MCUs) installed via SDKs. Click to select the MCU and filter the available matching boards. SDK part support can be hidden by clicking on the triangle (highlighted in the blue oval)
2. A display of all preinstalled parts (these are all LPC or Generic M parts). Click to select the MCU and filter the available matching boards (if any). Preinstalled part support can be hidden by clicking on the triangle (highlighted in blue)
3. A display of all boards from both SDKs or matching LPCOpen packages. Click to select the board and its associated MCU.
   - Boards from SDK packages will have **SDK** superimposed onto their image.
4. Some description relating to the users selection
5. A display to show the matching SDK for a chosen MCU or Board. If more than one matching SDK is installed, the user can select the SDK to use from this list
6. Any Warning, Error or Information related to the current selection
7. An input field to filter the available boards e.g. enter ‘64’ to see matching MK64… Freedom or Tower boards available
8. 3 options: to Sort boards from A-Z, Z-A or clear any filter made through the input field or a select click. **Note**: once a project has been created, the filter settings will be remembered the next time the Wizard is entered (unless cleared by an external event such as the installation of a new SDK).

This page provides a number of ways of quickly selecting the target for the project that you want to create.

In this description, we are going to create a project for a Freedom MK64xxx board (The required SDK has already been imported).

First, to reduce the number of boards displayed, we can simply type ‘64’ into the filter (7). Now only boards with MCUs matching ‘64’ will be displayed.



Creating project for device: MK64FN1M0xxx12 using board: FRDM-K64F

Filter: 64

Selected Device: MK64FN1M0xxx12 using board: FRDM-K64F

Target Core: cortex-m4

Description: K64_120: Kinetis® K64-120 MHz, 256KB SRAM Microcontrollers (MCUs) based on ARM® Cortex®-M4 Core

Name	Version	Location
SDK_2.x_FRDM-K64F	2.2.0	<Default Location>/SDK_2.x_FRDM-

Figure 5.3. New Project Wizard selection

When the (SDK) board is selected, you can see highlighted in the above figure that the matching MCU (part) and SDK are also selected automatically.

With a chosen board selected, now click ‘Next’...

5.1.1 SDK New Project Wizard: Basic Project Creation and Settings

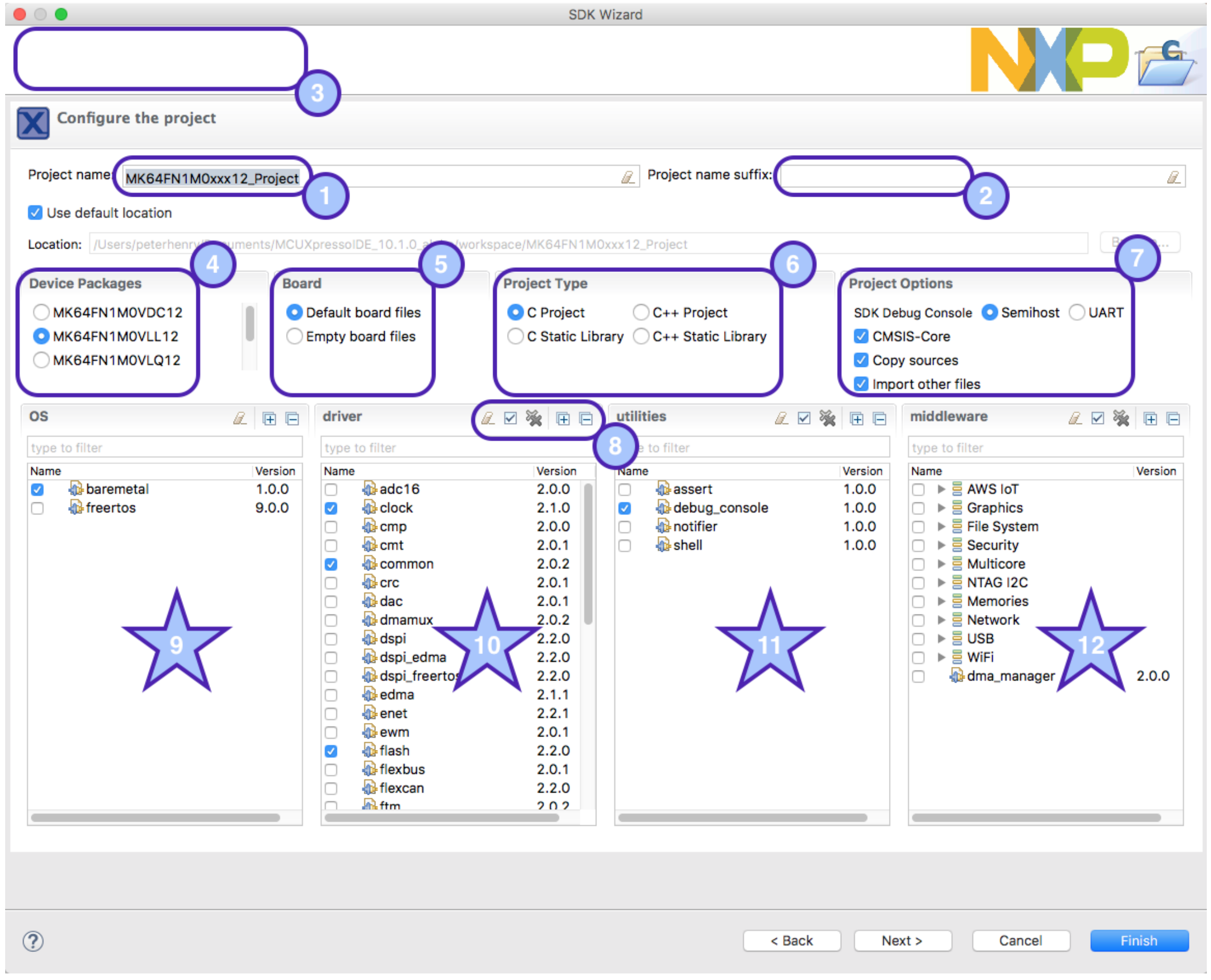
The SDK New Project Wizard consists of two pages offering basic and advanced configuration options. Each of these pages is preconfigured with default options (the default options offered on the advanced page may be set based on chosen settings from the basic page).

Therefore, to create a simple ‘Hello World’ C project for the Freedom MK64… board we selected, all that is required is simply click ‘Finish’.

Note: The project will be given a default name based on the MCU name. If this name matches a project within the workspace e.g. the wizard has previously been used to generate a project with the default name, then the error field will show a name clash and the ‘next’ and ‘finish’ buttons will be ‘greyed out’. To change the new project’s name; the blank ‘Project Name Suffix’ field can be used to quickly create a unique name but retain the original prefix.

This will create a project in the chosen workspace taking all the default Wizard options for our board.

However, the wizard offers the flexibility to select/change many build, library and source code options. These options and the components of this first Wizard page are described below.



Configure the project

Project name: MK64FN1M0xxx12_Project

Project name suffix:

[✓] Use default location

Location: /Users/peterhenry/Documents/MCUXpressoIDE_10.1.0_a/workspace/MK64FN1M0xxx12_Project

Device Packages
- ( ) MK64FN1M0VDC12
- (●) MK64FN1M0VLL12
- ( ) MK64FN1M0VLQ12

Board
- (●) Default board files
- ( ) Empty board files

Project Type
- (●) C Project
- ( ) C++ Project
- ( ) C Static Library
- ( ) C++ Static Library

Project Options
- SDK Debug Console (●) Semihost ( ) UART
- [✓] CMSIS-Core
- [✓] Copy sources
- [✓] Import other files

OS

	Name	Version
✓	baremetal	1.0.0
	freertos	9.0.0

driver

	Name	Version
	adc16	2.0.0
✓	clock	2.1.0
	cmp	2.0.0
	cmt	2.0.1
✓	common	2.0.2
	crc	2.0.1
	dac	2.0.1
	dmamux	2.0.2
	dspi	2.2.0
	dspi_edma	2.2.0
	dspi_freertos	2.2.0
	edma	2.1.1
	enet	2.2.1
	ewm	2.0.1
✓	flash	2.2.0
	flexbus	2.0.1
	flexcan	2.2.0
	ftm	2.0.2

utilities

	Name	Version
	assert	1.0.0
✓	debug_console	1.0.0
	notifier	1.0.0
	shell	1.0.0

middleware

	Name	Version
	AWS IoT	
	Graphics	
	File System	
	Security	
	Multicore	
	NTAG I2C	
	Memories	
	Network	
	USB	
	WiFi	
	dma_manager	2.0.0

Figure 5.4. New Project Wizard basic SDK settings

1. Project Name: The default project name prefix is automatically selected based on the part selected on the previous screen
   - Note: Due to restrictions in the length of filenames accepted by the Windows version of the underlying GCC toolchain, it is recommended that the length of project names is kept to 56 characters or less. Otherwise you may see project build error messages regarding files not being found, particularly during the link step.
2. Project Suffix: An optional suffix to append to a project name can be entered here
3. Errors and Warnings: Any error or warning will be displayed here. The ‘Next’ option will not be available until any error is handled – for example, a project name has been selected that matches an existing project name in your workspace. The suffix field (2) allows a convenient way of updating a project name

4. MCU Package: The device package can be selected from the range contained with the SDK. The package relates to the actual device packaging and typically has no meaning for project creation
5. Board files: This field allows the automatic selection of a default set of board support files, else empty files will be created. If a part rather than a board had been selected on the previous screen, these options will not be displayed.
    - If you intend to use board specific features such as output over UART, you should ensure Default board files are selected
6. Project Type: C or C++ projects or libraries can be selected. Selecting 'C' will automatically select RedLib libraries, selecting C++ will select NewlibNano libraries. See C/C++ Library Support [103]
7. Project Options:
    - Enable Semihost: will cause the Semihosted variant of the chosen library to be selected. For C projects this will default to be Redlib Semihost-nf. Semihosting allows IO operations such as printf and scanf to be emulated by the debug environment.
    - CMSIS-Core: will cause a CMSIS folder containing a variety of support code such as Clock Setup, header files to be created. It is recommended to leave this options ticked
    - Copy Sources: For zipped SDKs, this option will be ticked and greyed out. For unzipped SDKs, projects can be created that use linked references back to the original SDK folder. This feature is recommended for 'Power Users' only
8. Each set of components support a filter and check boxes for selection. These icons allow filters to be cleared, all check boxes to be set, all check boxes to be cleared
9. OS: This provides the option to pull in and link against Operating System resources such as FreeRTOS.
10. driver: enables the selection of supporting driver software components to support the MCU peripheral set.
11. utilities: a range of optional supporting utilities.
    - For example select the debug_console to make use of the SDK Debug Console handling of IO
    - Selecting this option will cause the wizard to substitute the (SDK) PRINTF() macro for C Library printf() within the generated code
    - The debug console option relies on the OpenSDA debug probe communicating to the host via VCOM over USB.
12. middleware: enables the selection of various middleware components

Finally, if there is no error condition displayed, 'Finish' can be selected to finish the wizard, alternatively, select 'Next' to proceed to the Advanced options page (described next).

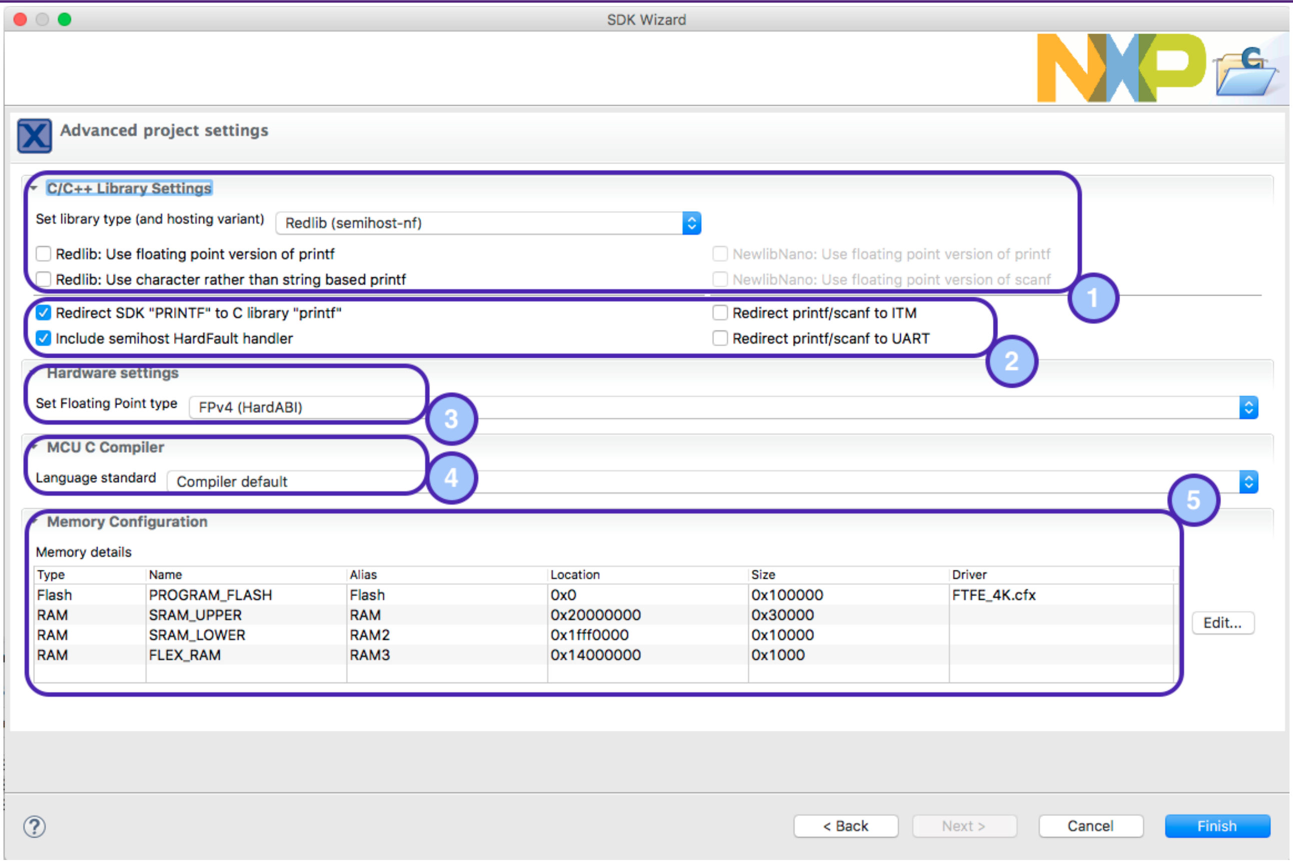
Important Note: Any components (OS, driver, utilities, middleware) selected by default within this wizard will be linked into the final image. However, any additional components selected by the user will only bring the corresponding sources into the project, these will only be linked into the final image if subsequently referenced. Additionally, selecting a component will automatically select any dependencies. Finally, please note that this is an additive process, **removing components may leave unresolved dependencies resulting in a project that will not build**.

Note: Some middleware components such as USB, are not currently compatible with the New project wizard functionality and so will be hidden. The recommended approach if such components are required is to import an example including the component and then modify this as required. Please see SDK Project Component Management [53] for details of how this might be done.

Note: By default, new project files are stored within the current MCUXpresso IDE workspace, **this is recommended since the workspace then contains both the sources and project descriptions.** However, the New Project Wizard allows a non default location to be specified if required. To ensure that each project's sources and local configuration are self contained when using non standard locations, the IDE will automatically create a sub directory inside the specified location using the *Project name prefix* setting. The newly created project files will then be stored within this location.

5.1.2 SDK New Project Wizard: Advanced Project Settings

The advanced configuration page will take certain default options based on settings from the first wizard project page, for example a C project will pre-select Redlib libraries, where as a C++ project will pre-select NewlibNano.



Set library type (and hosting variant): Redlib (semihost-nf)
- [ ] Redlib: Use floating point version of printf
- [ ] Redlib: Use character rather than string based printf
- [ ] NewlibNano: Use floating point version of printf
- [ ] NewlibNano: Use floating point version of scanf
- [x] Redirect SDK "PRINTF" to C library "printf"
- [x] Include semihost HardFault handler
- [ ] Redirect printf/scanf to ITM
- [ ] Redirect printf/scanf to UART

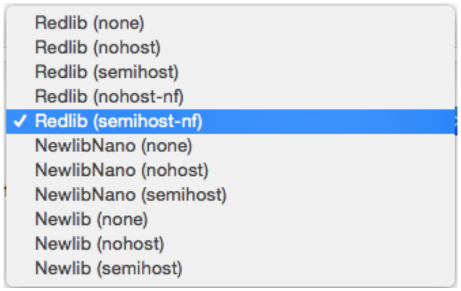
Set Floating Point type: FPv4 (HardABI)

Language standard: Compiler default

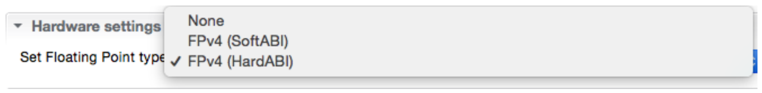
Memory details

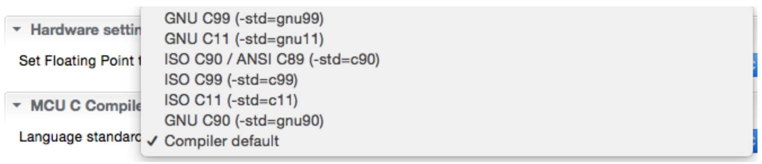
Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x100000	FTFE_4K.cfx
RAM	SRAM_UPPER	RAM	0x20000000	0x30000	
RAM	SRAM_LOWER	RAM2	0x1fff0000	0x10000	
RAM	FLEX_RAM	RAM3	0x14000000	0x1000	

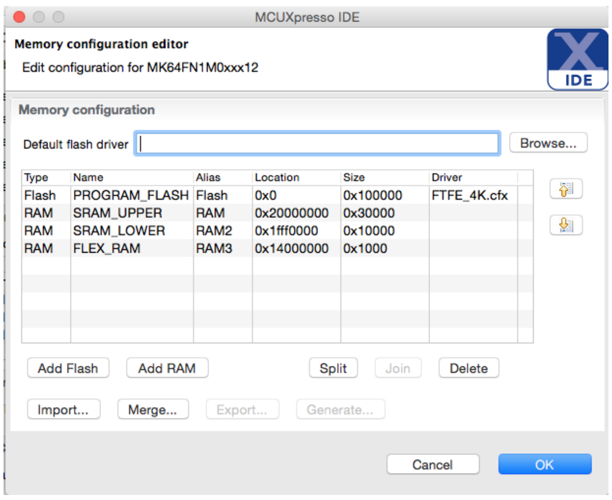
Figure 5.5. New Project Wizard advanced SDK settings

1. This panel allows the selection of library variants. See C/C++ Library Support [103]. **Note**: if a C++ project was selected on the previous page, then the Redlib options will be Greyed out.
   - 

     Redlib (none)
     Redlib (nohost)
     Redlib (semihost)
     Redlib (nohost-nf)
     ✓ Redlib (semihost-nf)
     NewlibNano (none)
     NewlibNano (nohost)
     NewlibNano (semihost)
     Newlib (none)
     Newlib (nohost)
     Newlib (semihost)
   - Also, based on the selection, a number of options may be chosen to modify the capability (and size) of printf support
   - Redlib Floating Point printf: If this option is ticked, floating point support for printf will automatically be linked in. This will allow printf to support the printing out of floating point variables at the expense of larger library support code. Similarly for Newlib.
   - Redlib use Character printf: selecting this option will avoid heap usage and reduce code size but make printf operations slower.

2. This panel allows options to be set related to Input/Output. See C/C++ Library Support [103].
   - Redirect SDK “PRINTF”: many SDK examples use a PRINTF macro, selecting this option causes redirection to C library IO rather than options provided by the SDK debug console.
   - Redirect printf/scanf to ITM: causes a C file 'retarget_itm.c to be pulled into your project. This then enables printf/scanf I/O to be sent over the SWO channel. The benefit of this is that I/O operations can be performed with little performance penalty. Furthermore, these routines do not require debugger support and for example could be used to generate logging that would effectively go to Null unless debug tools were attached. **Note**: This feature is not available on Cortex M0 and M0+ parts.
     - More information can be found in the MCUXpresso IDE SWO Trace Guide.
   - Redirect printf/scanf to UART: Sets the define SDK_DEBUGCONSOLE_UART causing the C libraries printf functions to re-direct to the SDKs debug console UART code.
3. Hardware Settings: from this drop down you can set options such as the type of floating point support available/required. This will default to an appropriate value for your MCU.
   - 

4. MCU C Compiler: from this drop down you can set various compiler options that can be set for the GNU C/C++ compiler.
   - 

5. Memory Configuration: This panel shows the Flash and RAM memory layout for the MCU project being created. The pre-selected LinkServer Flash driver is also shown. **Note**: this Flash driver will only be used for LinkServer (CMSIS-DAP) debug connections.
   - Clicking Edit invokes the IDE’s memory configuration editor. From this dialogue, the project’s default memory setting and hence automatically generated linker settings can be changed. See Memory Configuration and Linker Scripts [113]
   - 

MCUXpresso IDE

Memory configuration editor

Edit configuration for MK64FN1M0xxx12

Memory configuration

Default flash driver

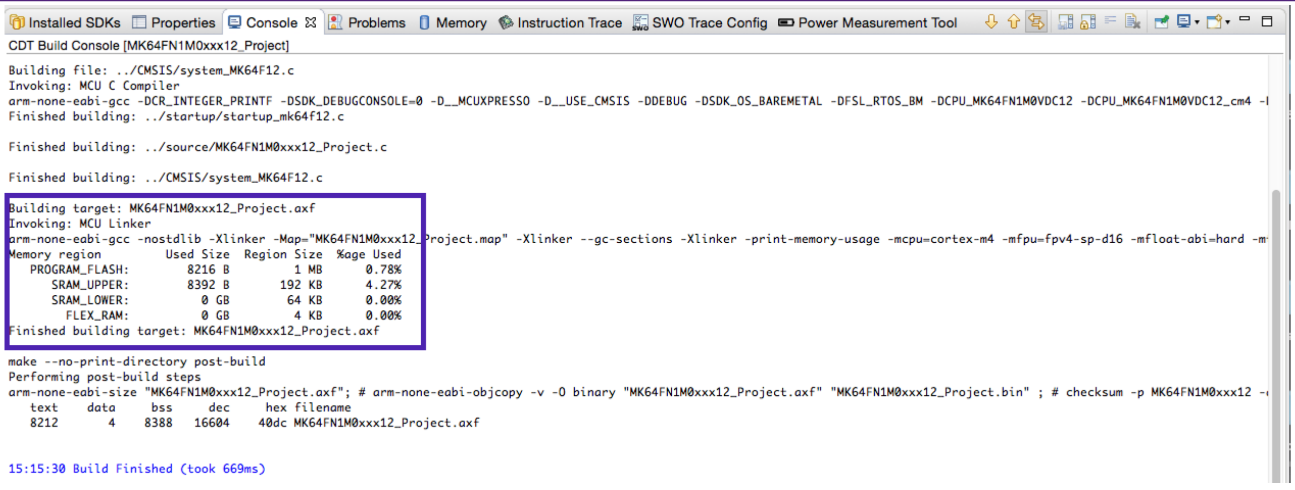
Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x100000	FTFE_4K.cfx
RAM	SRAM_UPPER	RAM	0x20000000	0x30000	
RAM	SRAM_LOWER	RAM2	0x1fff0000	0x10000	
RAM	FLEX_RAM	RAM3	0x14000000	0x1000	

5.2 SDK Build Project

To build a project created by the SDK New Project Wizard, simply select the project in the ‘Project Explorer’ view, then go to the ' **QuickStart**' Panel and click on the **build button** to build the project. This will build the project for the default projects ‘Debug’ configuration.

Note: MCUXpresso IDE projects are created with two build configurations, Debug and Release (more can be added if required). These differ in the default level of compiler optimisation. Debug projects default to None (-O0), and Release projects default to (-Os). For more information on switching between build configurations, see How do I switch between Debug and Release builds? [163]

The build log will be displayed in the console view as below.



CDT Build Console [MK64FN1M0xxx12_Project]

```
Building file: ../CMSIS/system_MK64F12.c
Invoking: MCU C Compiler
arm-none-eabi-gcc -DCR_INTEGER_PRINTF -DSDK_DEBUGCONSOLE=0 -D__MCUXPRESSO -D__USE_CMSIS -DDEBUG -DSDK_OS_BAREMETAL -DFSL_RTOS_BM -DCPU_MK64FN1M0VDC12 -DCPU_MK64FN1M0VDC12_cm4 -I
Finished building: ../startup/startup_mk64f12.c

Finished building: ../source/MK64FN1M0xxx12_Project.c

Finished building: ../CMSIS/system_MK64F12.c

Building target: MK64FN1M0xxx12_Project.axf
Invoking: MCU Linker
arm-none-eabi-gcc -nostdlib -Xlinker -Map="MK64FN1M0xxx12_Project.map" -Xlinker --gc-sections -Xlinker -print-memory-usage -mcpu=cortex-m4 -mfpu=fpv4-sp-d16 -mfloat-abi=hard -m
Memory region         Used Size  Region Size  %age Used
   PROGRAM_FLASH:        8216 B         1 MB      0.78%
      SRAM_UPPER:        8392 B       192 KB      4.27%
      SRAM_LOWER:           0 GB        64 KB      0.00%
        FLEX_RAM:           0 GB         4 KB      0.00%
Finished building target: MK64FN1M0xxx12_Project.axf

make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "MK64FN1M0xxx12_Project.axf"; # arm-none-eabi-objcopy -v -O binary "MK64FN1M0xxx12_Project.axf" "MK64FN1M0xxx12_Project.bin" ; # checksum -p MK64FN1M0xxx12 -
   text    data     bss     dec     hex filename
   8212       4    8388   16604    40dc MK64FN1M0xxx12_Project.axf


15:15:30 Build Finished (took 669ms)
```

Figure 5.6. New Project Wizard Build

The projects memory usage as highlighted above is shown below:

```
Memory region    Used Size  Region Size %age Used

  PROGRAM_FLASH:    8216 B       1 MB      0.78%
  SRAM_UPPER:       8392 B     192 KB      4.27%
  SRAM_LOWER:          0 GB     64 KB      0.00%
  FLEX_RAM:            0 GB      4 KB      0.00%

Finished building target: MK64FN1M0xxx12_Project.axf
```

By default, the application will build and link against the first Flash memory found within the devices memory configuration. For most MCUs there will only be one Flash device available. In this case our project requires 8216 bytes of Flash memory storage, 0.78% of the available Flash storage.

RAM will be used for global variable, the heap and the stack. MCUXpresso IDE provides a flexible scheme to reserve memory for Stack and Heap. The above example build has reserved 4KB each for the stack and the heap. Please See Memory Configuration and Linker Scripts [113] for detailed information.

6. Importing Example Projects (from installed SDKs)

In addition to drivers and part support, SDKs also deliver many example projects for the target MCU.

To import examples from an installed SDK, go to the **QuickStart** panel and select **Import SDK example(s)**.

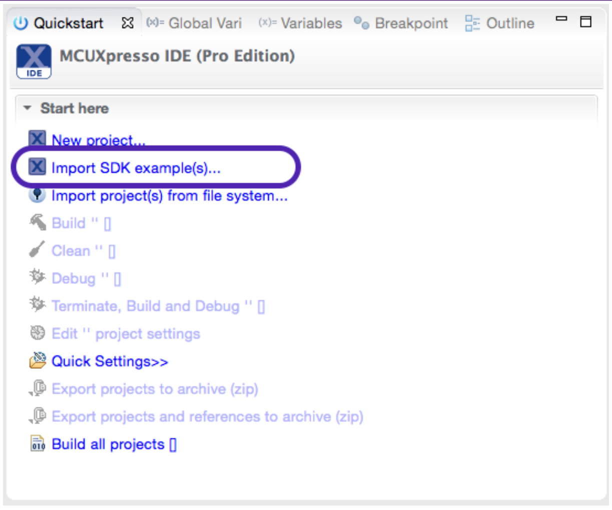
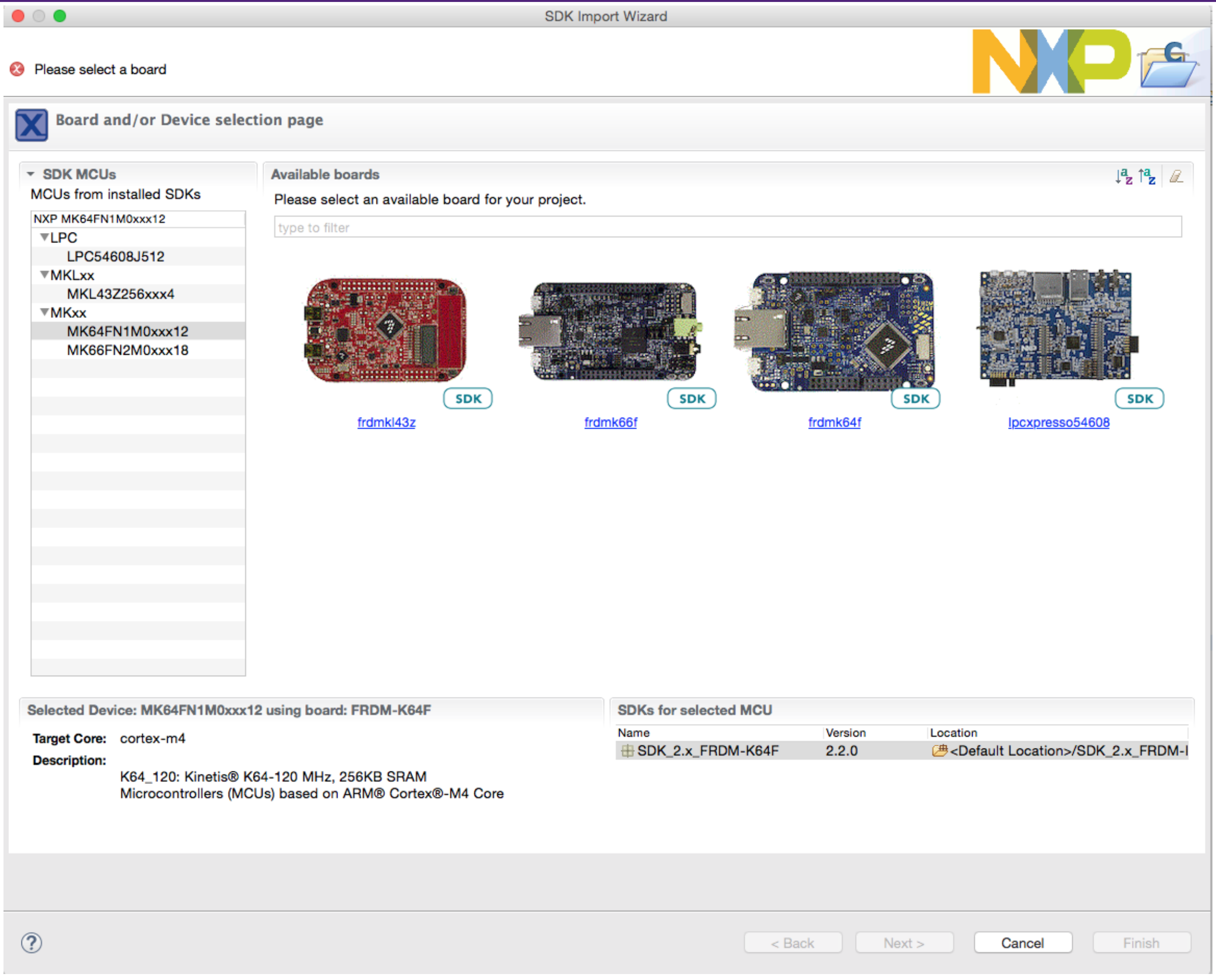


Figure 6.1. SDK Example

This option invokes the **Import SDK Example Wizard** that guides the user to import SDK example projects from installed SDKs.

Like the New Project wizard, this will initially launch a page allowing MCU/board selection. However now, only SDK supported parts and boards will be presented.


SDK Import Wizard

Please select a board

Board and/or Device selection page

SDK MCUs
MCUs from installed SDKs

NXP MK64FN1M0xxx12
- LPC
  - LPC54608J512
- MKLxx
  - MKL43Z256xxx4
- MKxx
  - MK64FN1M0xxx12
  - MK66FN2M0xxx18

Available boards
Please select an available board for your project.

type to filter

Selected Device: MK64FN1M0xxx12 using board: FRDM-K64F

Target Core: cortex-m4

Description:
K64_120: Kinetis® K64-120 MHz, 256KB SRAM Microcontrollers (MCUs) based on ARM® Cortex®-M4 Core

SDKs for selected MCU

Name	Version	Location
SDK_2.x_FRDM-K64F	2.2.0	<Default Location>/SDK_2.x_FRDM-I

Figure 6.2. SDK Example Board

6.1 SDK Example Import Wizard

Selection and filtering work in the same way as for the New Project Wizard [37] but note, examples are created for particular development boards, therefore a board must be selected to move to the ‘Next’ page of the wizard.

6.1.1 SDK Example Import Wizard: Basic Selection

The SDK Example Import Wizard consists of two pages offering basic and advanced configuration and selection options. The second configuration page is only available when a single example is selected for import. This is because examples may set specific options, and therefore changing settings globally is not sensible.

The first page offers all the available examples in various categories. These can be expanded to view the underlying hierarchical structure. The various settings and options are explained below: **Note**: The project will be given a default name based on the MCU name, Board name and Example name. If this name matches a project within the workspace e.g. the wizard has previously been use to generate an example with the default name, then the error field will show a name clash and the ‘next’ and ‘finish’ buttons will be greyed out. To change the new example name, the blank ‘Project Name Suffix’ field can be used to quickly create a unique name but retain the original prefix e.g. add ‘1’.

MCUXpresso IDE will create a project with common default settings for your chosen MCU and board. However, the wizard offers the flexibility to select/change many build, library and source code options. These options and the components of this first Wizard page are described below.

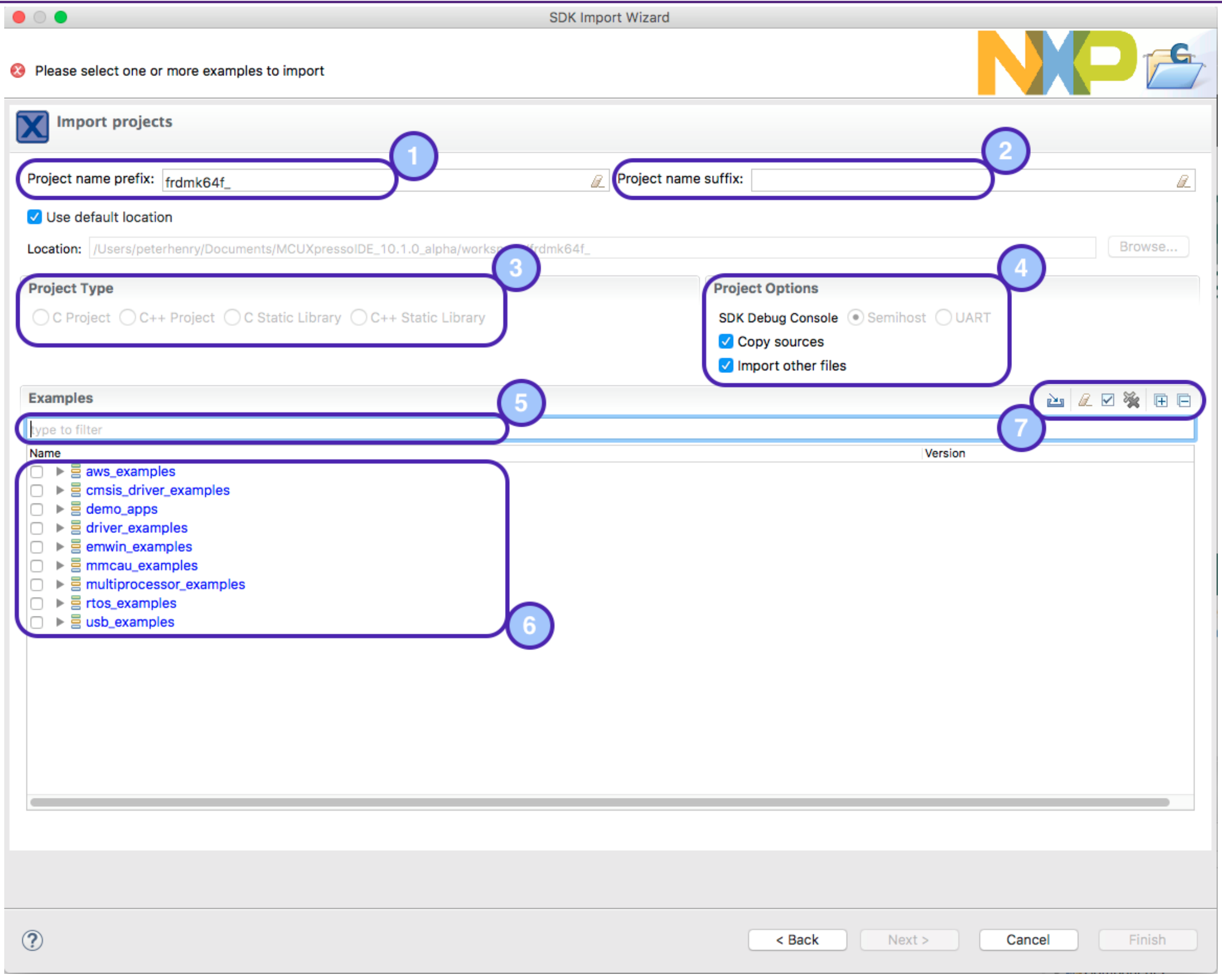


Figure 6.3. SDK Example Selection

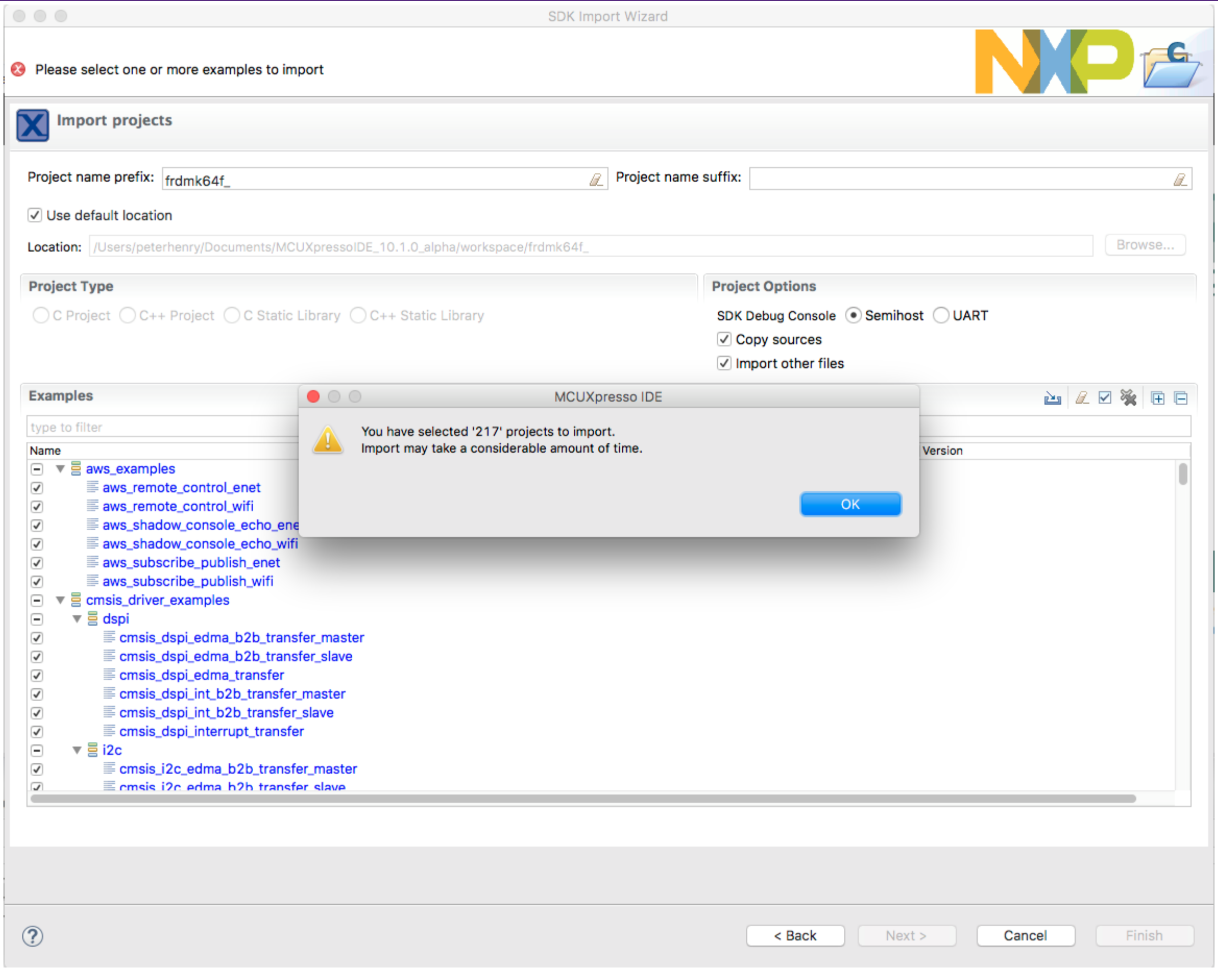
1. Project Name: A project name is automatically created with a name of the form: *prefix_SDK example path_example name_suffix*.
2. Project Suffix: An optional suffix to append to a project name can be entered here. This is particularly useful if you are repeating an import of one or more projects since an entry here can make all auto generated names unique for the current workspace... **Note**: Changing the default name of Imported SDK **Multicore examples** may cause linkage to fail.
3. Project Type: These will be set by the pre-set type of the example being imported. If more than one example is imported, then these options will appear greyed out.
4. Project Options:
   - 'SDK Debug Console': Once an example(s) is selected, this option can be used to control IO between semihost console or UART.
   - 'Copy sources': For unzipped SDKs, you can untick this option to create project containing source links to the original SDK files. This option should only be unticked with care, since editing linked example source will overwrite the original files!
   - 'Import other files': By default non source files such as graphics are filtered out during import, check this box to import all files.

5. Examples Filter: Enter text into this field to find possible matches, for example enter ‘LED’ or ‘bubble’ to find examples present in many SDKs. This filter is case insensitive.
6. Examples: The example list broken into categories. **Note**: for some parts there will be many potential examples to import
7. Various options (from left to right):
    - Opens a filer window to allow an example to be imported from an XML description. This is intended as a developer feature and is described in more detail below.
    - Clear any existing filter
    - Select (tick) all Examples
    - Clear all ticked examples
    - Open the example structure
    - Close the example structure

Finally, if there is no error condition displayed, ‘Finish’ can be selected to finish the wizard, alternatively if only one example has been selected the option to select ‘Next’ to proceed to the Advanced options page is available (described in the next section).

Note: SDKs may contain many examples, 217 is indicated for the FRDM MK64 SDK example shown below. Importing many examples will take time ... Consider that each example may consist of many files and associated description XML. A single example import may only take a few seconds, but this time is repeated for each additional example. Furthermore, the operation of the IDE maybe impacted by a large number of project in a single workspace, therefore it is suggested that example imports be limited to sensible numbers.

Note: Due to restrictions in the length of filenames accepted by the Windows version of the underlying GCC toolchain, it is recommended that the length of project names is kept to 56 characters or less. Otherwise you may see project build error messages regarding files not being found, particularly during the link step.


SDK Import Wizard

Please select one or more examples to import

Import projects

Project name prefix: frdmk64f_ Project name suffix:

☑ Use default location

Location: /Users/peterhenry/Documents/MCUXpressoIDE_10.1.0_alpha/workspace/frdmk64f_

Project Type: ○ C Project ○ C++ Project ○ C Static Library ○ C++ Static Library

Project Options: SDK Debug Console [✓ Semihost] ○ UART; ☑ Copy sources; ☑ Import other files

Examples

- aws_examples
  - ☑ aws_remote_control_enet
  - ☑ aws_remote_control_wifi
  - ☑ aws_shadow_console_echo_enet
  - ☑ aws_shadow_console_echo_wifi
  - ☑ aws_subscribe_publish_enet
  - ☑ aws_subscribe_publish_wifi
- cmsis_driver_examples
  - dspi
    - ☑ cmsis_dspi_edma_b2b_transfer_master
    - ☑ cmsis_dspi_edma_b2b_transfer_slave
    - ☑ cmsis_dspi_edma_transfer
    - ☑ cmsis_dspi_int_b2b_transfer_master
    - ☑ cmsis_dspi_int_b2b_transfer_slave
    - ☑ cmsis_dspi_interrupt_transfer
  - i2c
    - ☑ cmsis_i2c_edma_b2b_transfer_master
    - ☑ cmsis_i2c_edma_b2b_transfer_slave

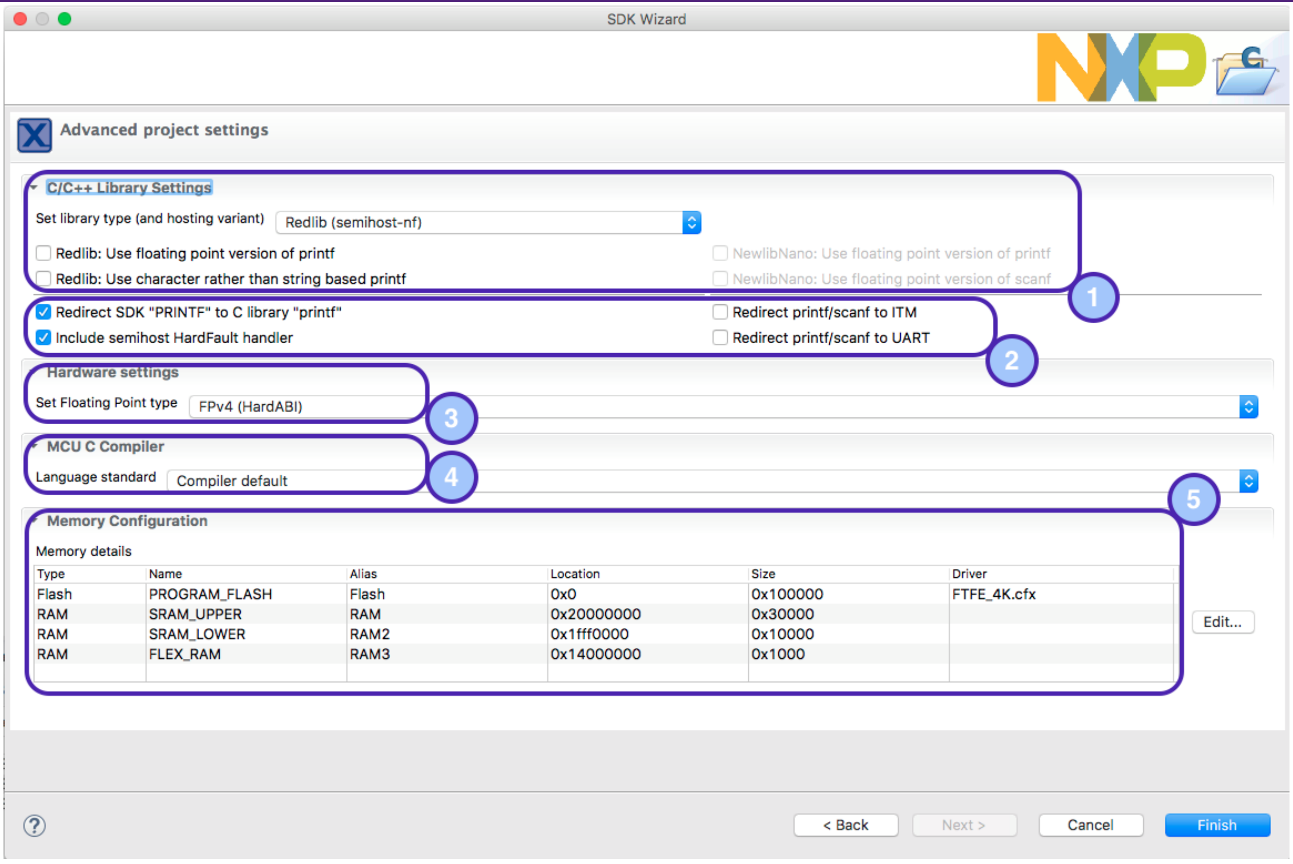
MCUXpresso IDE

You have selected '217' projects to import.
Import may take a considerable amount of time.

Figure 6.4. SDK Example Selection Many

6.1.2 SDK Example Import Wizard: Advanced options

The advanced configuration page (shown below) will take certain default options based on the example’s selected, for example a C project will pre-select Redlib libraries, where as a C++ project will pre-select NewlibNano.


SDK Wizard
Advanced project settings
C/C++ Library Settings
Set library type (and hosting variant) Redlib (semihost-nf)
☐ Redlib: Use floating point version of printf
☐ Redlib: Use character rather than string based printf
☐ NewlibNano: Use floating point version of printf
☐ NewlibNano: Use floating point version of scanf
☑ Redirect SDK "PRINTF" to C library "printf"
☑ Include semihost HardFault handler
☐ Redirect printf/scanf to ITM
☐ Redirect printf/scanf to UART
Hardware settings
Set Floating Point type FPv4 (HardABI)
MCU C Compiler
Language standard Compiler default
Memory Configuration
Memory details
Edit...
< Back
Next >
Cancel
Finish
1
2
3
4
5

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x100000	FTFE_4K.cfx
RAM	SRAM_UPPER	RAM	0x20000000	0x30000	
RAM	SRAM_LOWER	RAM2	0x1fff0000	0x10000	
RAM	FLEX_RAM	RAM3	0x14000000	0x1000	

Figure 6.5. New Project Wizard advanced SDK settings

These settings closely match those in SDK New Project Wizard description. Therefore see SDK New Project Wizard:Advanced Options [42] for a description of these options. **Note**: Changing these advanced options may prevent an example from building or executing.

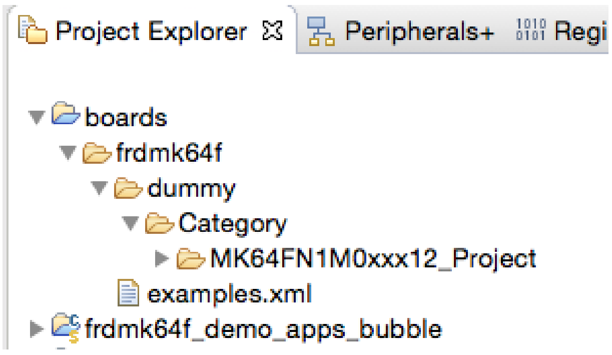
6.1.3 SDK Example Import Wizard: Import from XML fragment

This option works in conjunction with the *'Project Explorer' -> Tools -> Generate Example XML* (and is also used to import project created by the MCUXpresso Config Tools Project Generator).

The functionality here is to merge existing sources within a selectable board package framework.

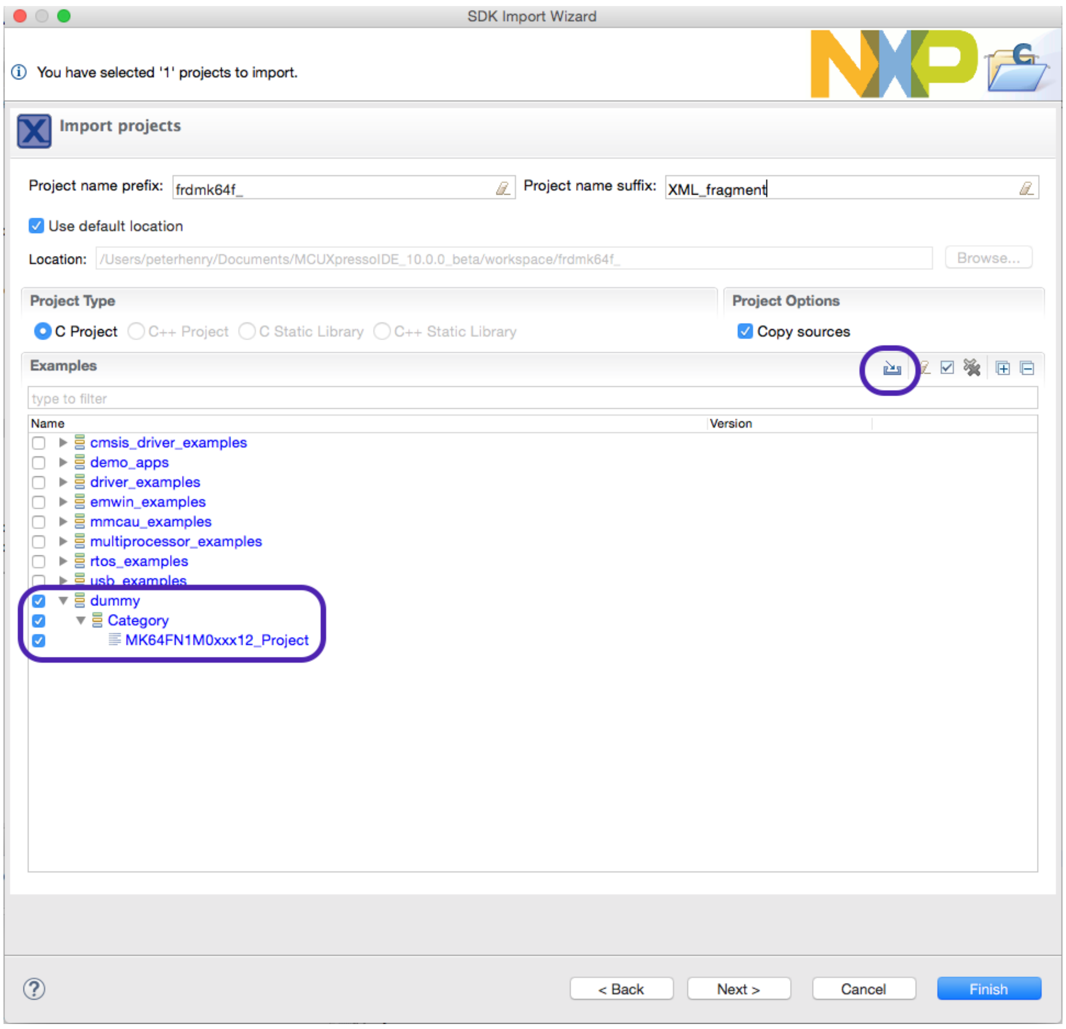
To create an XML "fragment" for an existing project in your workspace, right click on the project in the 'Project Explorer' (or just in the 'Project Explorer' view with no project selected) and choose *Tools->Generate examples.xml file*

The selected project or all the projects in the workspace (if no projects are selected) will be converted into a fragment within a new folder created in the workspace itself:



To create a project from a fragment, click on "Import SDK examples…" in the **QuickStart** Panel view:

Then select a board and then click on the button "Import from XML…" (highlighted below and described in the previous section). You will see the examples definitions from the external fragment in list of examples as shown and selected below.


SDK Import Wizard

You have selected '1' projects to import.

Import projects

Project name prefix: frdmk64f_ Project name suffix: XML_fragment

Use default location

Location: /Users/peterhenry/Documents/MCUXpressoIDE_10.0.0_beta/workspace/frdmk64f_

Project Type: C Project, C++ Project, C Static Library, C++ Static Library

Project Options: Copy sources

Examples

Name	Version
cmsis_driver_examples	
demo_apps	
driver_examples	
emwin_examples	
mmcau_examples	
multiprocessor_examples	
rtos_examples	
usb_examples	
✓ dummy	
✓ Category	
✓ MK64FN1M0xxx12_Project	

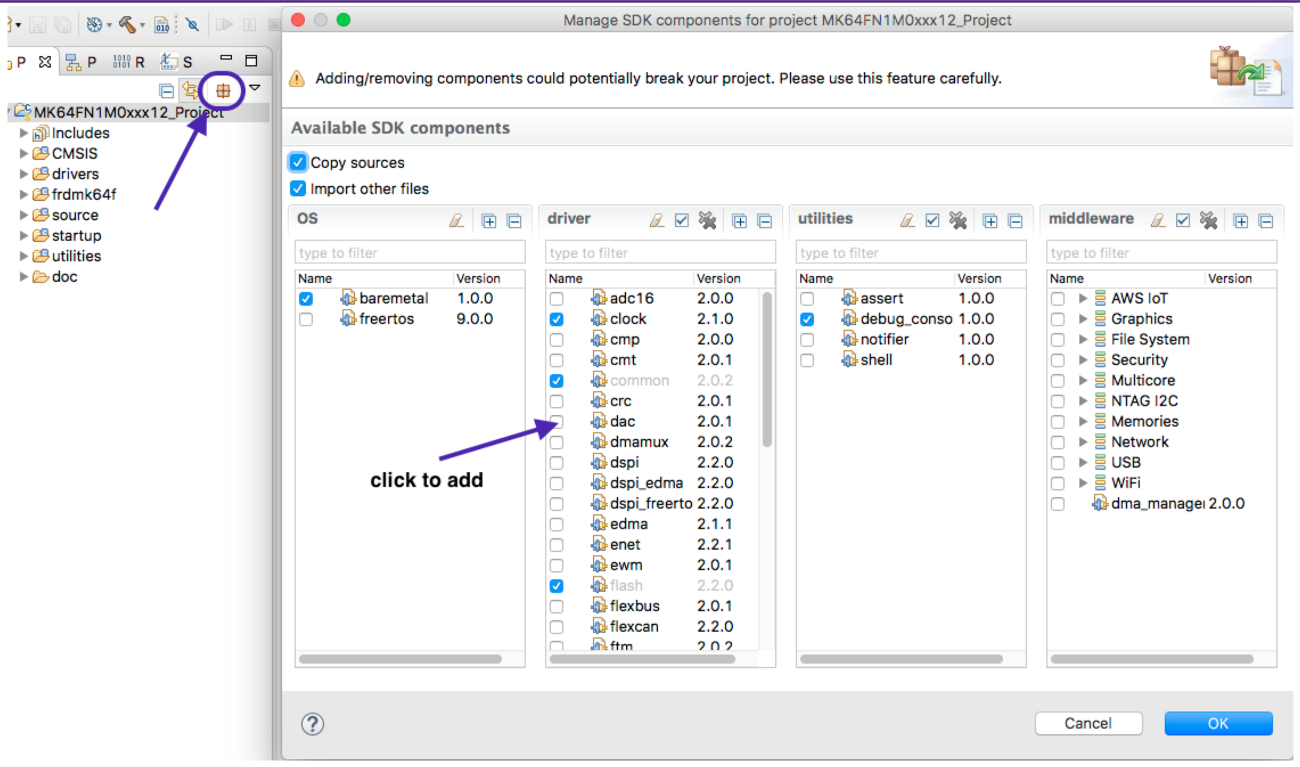
Select the external examples you want to re-create and click on "Finish". The project(s) will be created in the workspace.

6.1.4 Importing Examples to non default locations

By default, imported example sources will be stored within the current MCUXpresso IDE workspace, **this is recommended since the workspace then contains both the sources and project descriptions**. However, the Import SDK Example Wizard allows a non default location to be specified if required. To ensure that each project's sources and local configuration are self contained when using non standard locations, the IDE will automatically create a sub directory inside the specified location using the *Project name prefix* setting. Single or multiple imported projects will then be stored within this location.

7. SDK Project Component Management

Project and examples created from SDKs contain a number of software components such as peripheral drivers or middleware. In previous versions of MCUXpresso IDE, the option to add components was only available when creating a new project and not possible for imported examples. New in MCUXpresso IDE version 10.1.0 is the ability to easily add (or remove) SDK components to a previously created or imported example project via a new *Manage SDK components* wizard. To launch the Manage SDK Components wizard, simply select the chosen project in the Project explorer and then click the *package* icon as indicated below:



Manage SDK components for project MK64FN1M0xxx12_Project

Adding/removing components could potentially break your project. Please use this feature carefully.

Available SDK components

☑ Copy sources

☑ Import other files

OS

	Name	Version
✓	baremetal	1.0.0
	freertos	9.0.0

driver

	Name	Version
	adc16	2.0.0
✓	clock	2.1.0
	cmp	2.0.0
	cmt	2.0.1
✓	common	2.0.2
	crc	2.0.1
	dac	2.0.1
	dmamux	2.0.2
	dspi	2.2.0
	dspi_edma	2.2.0
	dspi_freerto	2.2.0
	edma	2.1.1
	enet	2.2.1
	ewm	2.0.1
✓	flash	2.2.0
	flexbus	2.0.1
	flexcan	2.2.0
	ftm	2.0.2

utilities

	Name	Version
	assert	1.0.0
✓	debug_conso	1.0.0
	notifier	1.0.0
	shell	1.0.0

middleware

	Name	Version
	AWS IoT	
	Graphics	
	File System	
	Security	
	Multicore	
	NTAG I2C	
	Memories	
	Network	
	USB	
	WiFi	
	dma_manage	2.0.0

Figure 7.1. Manage SDK Components

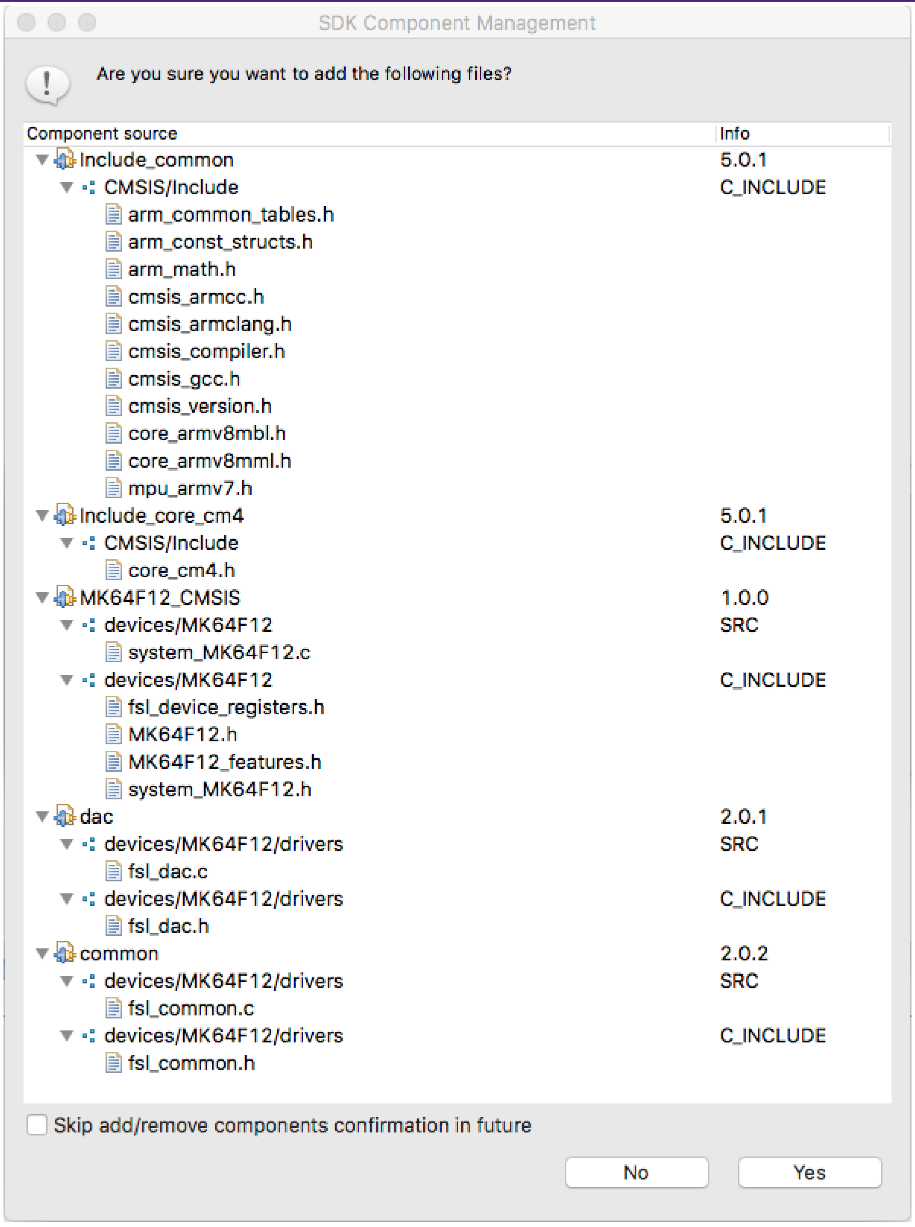
NOTE: This powerful feature can add (or remove) SDK components (**and their dependencies**) at a source file level, relying on meta data contained within the SDK. However the following points should be noted:

- The IDE can only maintain dependencies between SDK components. SDK component functions referenced from user created files or from sources such as an SDK example's main() function will not be taken into account when determining the sage removal of components. Therefore, the IDE cannot always prevent users removing components that may actually be required for a successful project build.
- Defined symbols will not be removed when components are removed, therefore users should ensure only required symbols are present if components are removed. Failing to do this may lead to project build failures.

7.1 SDK Project Component Management example

To demonstrate the use of this feature, the *dac* driver will be added. To do this, launch the Manage SDK components wizard, and click on the *dac* driver component then click 'OK'.

Next, a dialogue will be presented listing all of the source files required by this component – as below.


SDK Component Management

Are you sure you want to add the following files?

Component source	Info
Include_common	5.0.1
CMSIS/Include	C_INCLUDE
arm_common_tables.h	
arm_const_structs.h	
arm_math.h	
cmsis_armcc.h	
cmsis_armclang.h	
cmsis_compiler.h	
cmsis_gcc.h	
cmsis_version.h	
core_armv8mbl.h	
core_armv8mml.h	
mpu_armv7.h	
Include_core_cm4	5.0.1
CMSIS/Include	C_INCLUDE
core_cm4.h	
MK64F12_CMSIS	1.0.0
devices/MK64F12	SRC
system_MK64F12.c	
devices/MK64F12	C_INCLUDE
fsl_device_registers.h	
MK64F12.h	
MK64F12_features.h	
system_MK64F12.h	
dac	2.0.1
devices/MK64F12/drivers	SRC
fsl_dac.c	
devices/MK64F12/drivers	C_INCLUDE
fsl_dac.h	
common	2.0.2
devices/MK64F12/drivers	SRC
fsl_common.c	
devices/MK64F12/drivers	C_INCLUDE
fsl_common.h	

☐ Skip add/remove components confirmation in future

No Yes

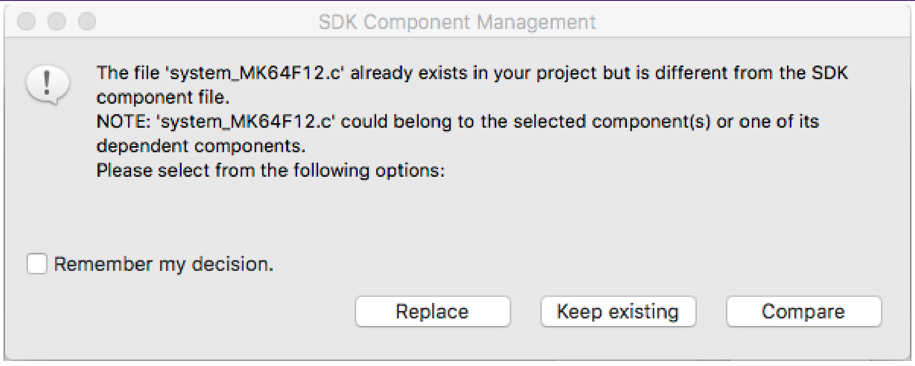
Figure 7.2. SDK Component Management

Note: Many of these files may already be included within your project.

Click ‘Yes’ to add these source files to your project.

Important Note: Since your project may contain edited or entirely new versions of the required source files, MCUXpresso IDE will perform a comparison between the new files to be included and any existing files already within the selected project.

Should a source file difference be found, then a dialogue as below will be launched:



SDK Component Management

The file 'system_MK64F12.c' already exists in your project but is different from the SDK component file.
NOTE: 'system_MK64F12.c' could belong to the selected component(s) or one of its dependent components.
Please select from the following options:

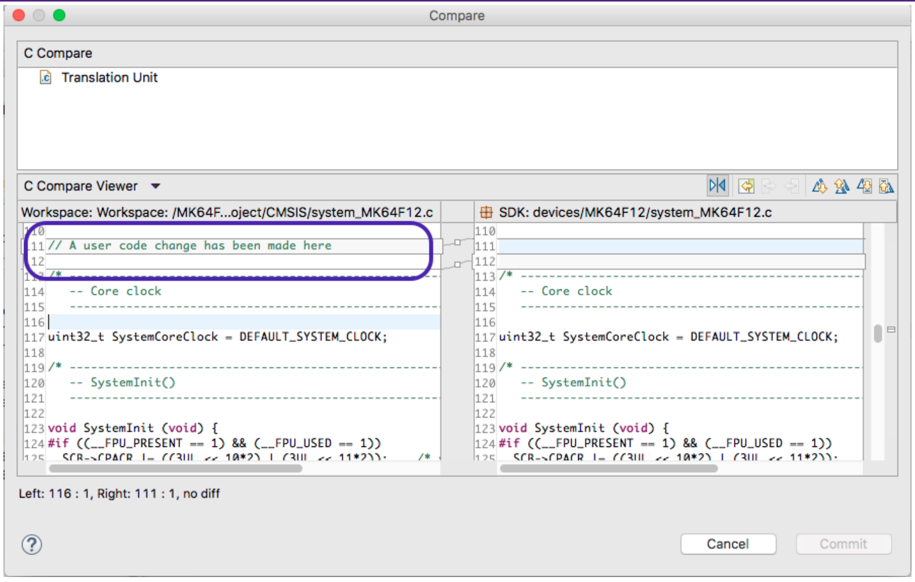
Figure 7.3. SDK Component Management file difference

From here you can choose from the following options:

Replace click to overwrite the projects file from the SDK version **Keep Existing** click to keep the existing project file unchanged **Compare** click to compare the two files – this will launch the Eclipse file compare utility so the new SDK file can be compared with the projects copy.

In this example, we will click ‘Compare’ ...

Below, you can see that a user project source modification has been found:



Workspace: Workspace: /MK64F...oject/CMSIS/system_MK64F12.c

```
// A user code change has been made here

/* ------------------------------------------------
   -- Core clock
   ------------------------------------------------

uint32_t SystemCoreClock = DEFAULT_SYSTEM_CLOCK;

/* ------------------------------------------------
   -- SystemInit()
   ------------------------------------------------

void SystemInit (void) {
#if ((__FPU_PRESENT == 1) && (__FPU_USED == 1))
```

SDK: devices/MK64F12/system_MK64F12.c

```


/* ------------------------------------------------
   -- Core clock
   ------------------------------------------------

uint32_t SystemCoreClock = DEFAULT_SYSTEM_CLOCK;

/* ------------------------------------------------
   -- SystemInit()
   ------------------------------------------------

void SystemInit (void) {
#if ((__FPU_PRESENT == 1) && (__FPU_USED == 1))
```

Left: 116 : 1, Right: 111 : 1, no diff

Figure 7.4. SDK Component Management file compare

The Compare utility allows any change to be examined and a decision made regarding which code lines to choose or ignore. When the decisions have been made, click ‘Commit’ to use these changes or ‘Cancel’ to leave the project file unchanged.

Finally please note the application build sizes before the addition:

```
Memory region         Used Size  Region Size  %age Used
   PROGRAM_FLASH:       13348 B         1 MB      1.27%
      SRAM_UPPER:        8444 B       192 KB      4.29%
      SRAM_LOWER:          0 GB        64 KB      0.00%
        FLEX_RAM:          0 GB         4 KB      0.00%
```

```
Finished building target: MK64FN1M0xxx12_Project.axf
```

Followed by the application sizes after the addition.

```
Memory region         Used Size  Region Size  %age Used
   PROGRAM_FLASH:       13348 B         1 MB      1.27%
      SRAM_UPPER:        8444 B       192 KB      4.29%
      SRAM_LOWER:           0 GB        64 KB      0.00%
        FLEX_RAM:           0 GB         4 KB      0.00%
Finished building target: MK64FN1M0xxx12_Project.axf
```

These are exactly the same, because although new source files have been added to the project, they will not be referenced by any existing code and so not included in the final image. To make use of any new component, some functionality must of course be referenced.

Note: Some middleware components such as USB, are not compatible with the Add/Remove component functionality and so will be hidden from the Add/Remove dialogue. The recommended approach if such components are required is to import an example including the component and modify as required. This restriction will be addressed in a future release.

8. Creating New Projects using Preinstalled Part Support

For Creating project using SDKs please see Creating New Projects using installed SDK Part Support [37]

To explore the range of preinstalled parts/MCUs simply click ‘New project’ in the **QuickStart panel**. This will open a page similar to the image below:

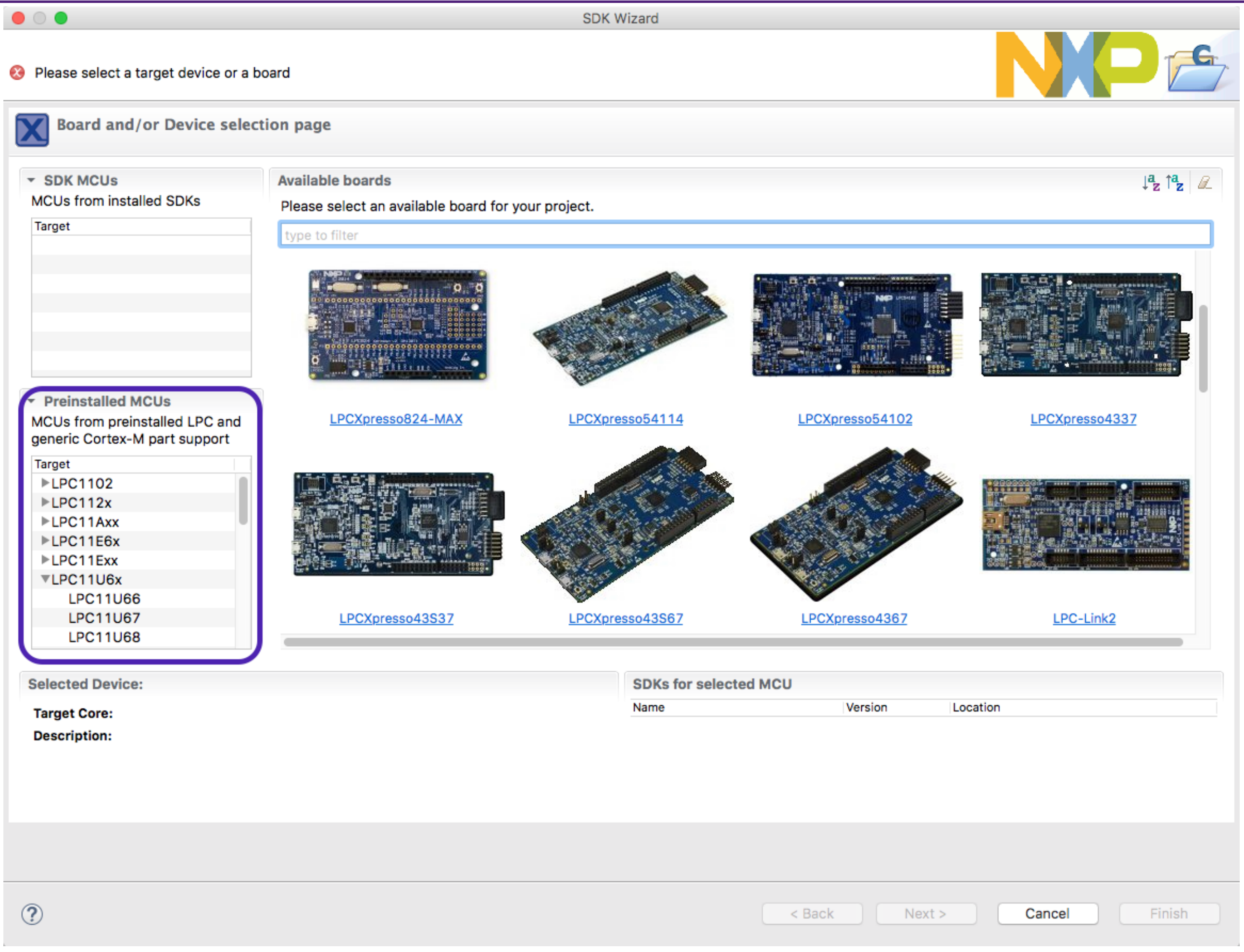


Figure 8.1. New Project Wizard Preinstalled

The list of preinstalled parts is presented on the bottom left of this window.

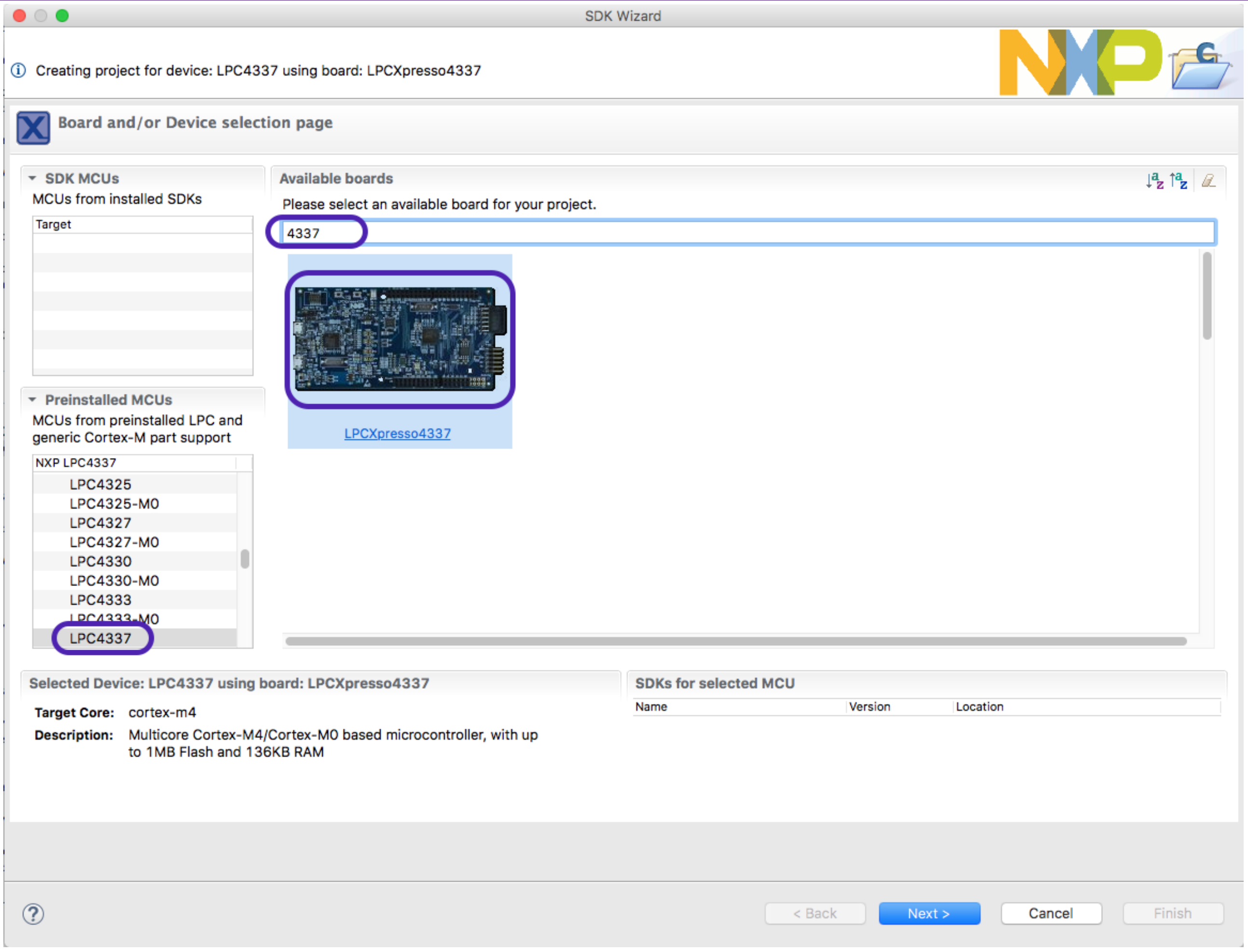
You will also see a range of related development boards indicating whether a matching LPCOpen Library is available.

For details of this page see: New Project Wizard details [37]

8.1 New Project Wizard

This wizard page provides a number of ways of quickly selecting the target for the project that you want to create.

In this description, we are going to create a project for an LPC4337 MCU (for this MCU an LPCOpen project exists), so we can locate this MCU using the board filter. **NOTE**: Boards will be displayed where either LPCOpen or CodeBundle projects exist, To reduce the number of boards displayed, we can simply type ‘4337’ into the filter so only boards with MCUs containing ‘4337’ will be displayed.



SDK Wizard

Creating project for device: LPC4337 using board: LPCXpresso4337

Board and/or Device selection page

SDK MCUs
MCUs from installed SDKs

Target

Available boards
Please select an available board for your project.

4337

LPCXpresso4337

Preinstalled MCUs
MCUs from preinstalled LPC and generic Cortex-M part support

NXP LPC4337
LPC4325
LPC4325-M0
LPC4327
LPC4327-M0
LPC4330
LPC4330-M0
LPC4333
LPC4333-M0
LPC4337

Selected Device: LPC4337 using board: LPCXpresso4337

Target Core: cortex-m4
Description: Multicore Cortex-M4/Cortex-M0 based microcontroller, with up to 1MB Flash and 136KB RAM

SDKs for selected MCU

Name	Version	Location

Figure 8.2. New Project Wizard selection for Preinstalled MCUs

When a board is selected as highlighted in the above figure, the matching MCU (part) is also selected automatically.

Note: if no matching board is available, the required MCU can be selected from the list of Preinstalled MCUs.

Note: Boards added to MCUXpresso IDE from SDKs will have an ‘SDK’ graphic superimposed on the board image. Boards without the SDK graphic indicate that a matching LPCOpen package (or Code bundle) is available for that board and associated MCU.

LPCOpen is described in section LPCOpen Software Drivers and Examples [64]

With a chosen board selected, now click ‘Next’...

The wizards for Preinstalled MCUs are very similar to those featured in LPCXpresso IDE.

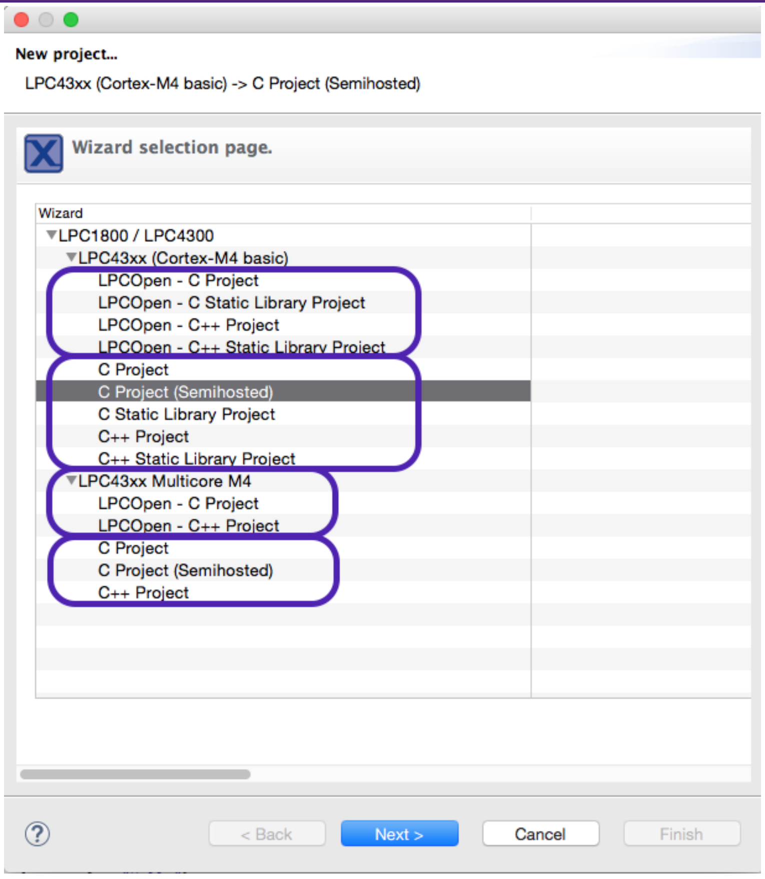
8.2 Creating a Project

The MCUXpresso IDE includes many project templates to allow the rapid creation of correctly configured projects for specific MCUs.

This New Project wizard supports 2 types of projects:

- Those targeting LPCOpen libraries
- Standalone projects

In addition, certain MCUs like the LPC4337 support multiple core internally, for these MCUs, Multicore options will also be presented (as below):


New project...

LPC43xx (Cortex-M4 basic) -> C Project (Semihosted)

Wizard selection page.

Wizard
- LPC1800 / LPC4300
  - LPC43xx (Cortex-M4 basic)
    - LPCOpen - C Project
    - LPCOpen - C Static Library Project
    - LPCOpen - C++ Project
    - LPCOpen - C++ Static Library Project
    - C Project
    - C Project (Semihosted)
    - C Static Library Project
    - C++ Project
    - C++ Static Library Project
  - LPC43xx Multicore M4
    - LPCOpen - C Project
    - LPCOpen - C++ Project
    - C Project
    - C Project (Semihosted)
    - C++ Project

Figure 8.3. New project: wizard selection

You can now select the type of project that you wish to create (see below for details of Wizard types).

In this case, we will show the steps in creating a simple C ‘Hello World’ example project.

8.2.1 Selecting the Wizard Type

For most MCU families the MCUXpresso IDE provides wizards for two forms of project: LPCOpen and non-LPCOpen. For more details on LPCOpen, see Software drivers and examples[64] . For both kinds, the main wizards available are:

C Project

- Creates a simple C project, with the `main()` routine consisting of an infinite `while(1)` loop that increments a counter.
- For LPCOpen projects, code will also be included to initialize the board and enable a LED.

C++ Project

- Creates a simple C++ project, with the `main()` routine consisting of an infinite `while(1)` loop that increments a counter.
- For LPCOpen projects, code will also be included to initialize the board and enable a LED.

C Static Library Project

- Creates a simple static library project, containing a source directory and, optionally, a directory to contain include files. The project will also contain a “liblinks.xml” file, which can be used by the smart update wizard on the context-sensitive menu to create links from application projects to this library project. For more details, please see the FAQ at

https://community.nxp.com/message/630594

C++ Static Library Project

- Creates a simple (C++) static library project, like that produced by the C Static Library Project wizard, but with the tools set up to build C++ rather than C code.

The non-LPCOpen wizard families also include a further wizard:

Semihosting C Project

- Creates a simple “Hello World” project, with the `main()` routine containing a `printf()` call, which will cause the text to be displayed within the Console View of the MCUXpresso IDE. This is implemented using “semihosting” functionality. See the section on Semihosting [106] for more information.

8.2.2 Configuring the Project

Once you have selected the appropriate project wizard, you will be able to enter the name of your new project, this must be unique for the current workspace.

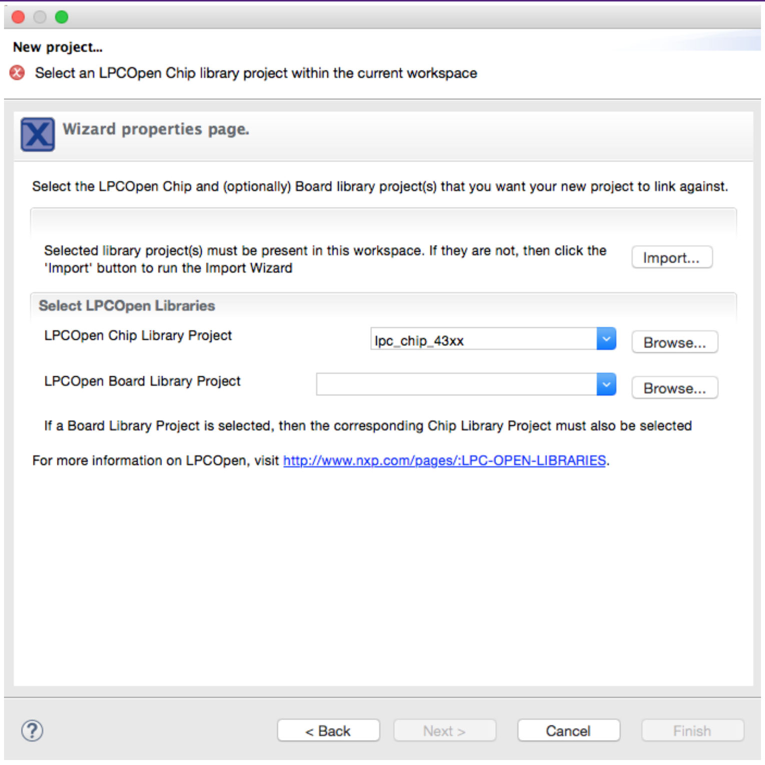
Finally you will be presented with one or more “Options” pages that provide the ability to set a number of project-specific options. The choices presented will depend upon which MCU you are targeting and the specific wizard you selected, and may also change between versions of the MCUXpresso IDE. **Note**: if you have any doubts over any of the options, then we would normally recommend leaving them set to their default values.

The following sections detail some of the options that you may see when running through a wizard.

8.3 Wizard Options

8.3.1 LPCOpen Library Project Selection

When creating an LPCOpen-based project, the first option page that you will see is the LPCOpen library selection page.



New project...

Select an LPCOpen Chip library project within the current workspace

Wizard properties page.

Select the LPCOpen Chip and (optionally) Board library project(s) that you want your new project to link against.

Selected library project(s) must be present in this workspace. If they are not, then click the 'Import' button to run the Import Wizard

Select LPCOpen Libraries

LPCOpen Chip Library Project: lpc_chip_43xx

LPCOpen Board Library Project:

If a Board Library Project is selected, then the corresponding Chip Library Project must also be selected

For more information on LPCOpen, visit http://www.nxp.com/pages/:LPC-OPEN-LIBRARIES.

Figure 8.4. LPCOpen library selection

This page allows you to run the “Import wizard” to download the LPCOpen bundle for your target MCU/board from http://www.nxp.com/lpcopen and import it into your Workspace, if you have not already done so.

You will then need to select the LPCOpen Chip library for your MCU using the Workspace browser (and for some MCUs an appropriate value will also be available from the dropdown next to the Browse button). **Note** that the wizard will not allow you to continue until you have selected a library project that exists within the Workspace.

Finally, you can optionally select the LPCOpen Board library for the board that your MCU is fitted to, using the Workspace browser (and again, in some cases an appropriate value may also be available from the dropdown next to the Browse button). Although selection of a board library is optional, it is recommended that you do this in most cases.

8.3.2 CMSIS-CORE Selection

For backwards compatibility reasons, the non-LPCOpen wizards for many parts provide the ability to link a new project with a CMSIS-CORE library project. The CMSIS-CORE portion of ARM’s **Cortex Microcontroller Software Interface Standard** (or **CMSIS**) provides a defined way of accessing MCU peripheral registers, as well as code for initializing an MCU and accessing various aspects of functionality of the Cortex CPU itself. The MCUXpresso IDE typically provides support for CMSIS through the provision of CMSIS library projects. CMSIS-CORE library projects can be found in the Examples directory of your MCUXpresso IDE installation.

Generally, if you wish to use CMSIS-CORE library projects, you should use `CMSIS_CORE_<partfamily>` (these projects use components from ARM’s CMSIS v3.20 specification). The MCUXpresso IDE does in some cases provide libraries based on early versions of the CMSIS specification with names such as `CMSISv1p30_<partfamily>`, but these are not recommended for use in new projects.

The CMSIS library option within the MCUXpresso IDE allows you to select which (if any) CMSIS-CORE library you want to link to from the project you are creating. **Note** that you will need to import the appropriate CMSIS-CORE library project into the workspace before the wizard will allow you to continue.

For more information on CMSIS and its support in the MCUXpresso IDE, please see the FAQ at

https://community.nxp.com/message/630589

Note: The use of LPCOpen instead of CMSIS-CORE library projects is recommended in most cases for new projects. (In fact LPCOpen actually builds on top of many aspects of CMSIS-CORE.) For more details see Software drivers and examples [64]

8.3.3 CMSIS DSP Library Selection

ARM's **Cortex Microcontroller Software Interface Standard** (or **CMSIS**) specification also provides a definition and implementation of a DSP library. The MCUXpresso IDE provides prebuilt library projects for the CMSIS DSP library for Cortex-M0/M0+, Cortex-M3 and Cortex-M4 parts, although a source version of it is also provided within the MCUXpresso IDE Examples.

Note: The CMSIS DSP library can be used with both LPCOpen and non-LPCOpen projects.

8.3.4 Peripheral Driver Selection

For some parts, one or more peripheral driver library projects may be available for the target MCU from within the Examples area of your MCUXpresso IDE installation. The non-LPCOpen wizards allow you to create appropriate links to such library projects when creating a new project. You will need to ensure that you have imported such libraries from the Examples before selecting them in the wizard.

Note: The use of LPCOpen rather than these peripheral driver projects is recommended in most cases for new projects.

8.3.5 Enable use of Floating Point Hardware

Certain MCUs may include a hardware floating point unit (for example NXP LPC32xx, LPC407x_8x, and LPC43xx parts). This option will set appropriate build options so that code is built to use the hardware floating point unit and will also cause startup code to enable the unit to be included.

8.3.6 Code Read Protect

NXP's Cortex based LPC MCUs provide a "Code Read Protect" (CRP) mechanism to prevent certain types of access to internal Flash memory by external tools when a specific memory location in the internal Flash contains a specific value. The MCUXpresso IDE provides support for setting this memory location. See the section on Code Read Protection [118] for more information.

8.3.7 Enable use of `Romdivide` Library

Certain NXP Cortex-M0 based MCUs, such as LPC11Axx, LPC11Exx, LPC11Uxx, and LPC12xx, include optimized code in ROM to carry out divide operations. This option enables the use of these Romdivide library functions. For more details see the FAQ at

https://community.nxp.com/message/630743

8.3.8 Disable Watchdog

Unlike most MCUs, NXP's LPC12xx MCUs enable the watchdog timer by default at reset. This option disables that default behavior. For more details, please see the FAQ at https://community.nxp.com/message/630654

8.3.9 LPC1102 ISP Pin

The provision of a pin to trigger entry to NXP's ISP bootloader at reset is not hardwired on the LPC1102, unlike other NXP MCUs. This option allows the generation of default code for providing an ISP pin. For more information, please see NXP's application note, AN11015, "Adding ISP to LPC1102 systems".

8.3.10 Memory Configuration Editor

For certain MCUs such as the LPC18xx and LPC43xx, the wizard will present the option to edit the target memory configuration. This is because these parts may make use of external SPIFI Flash memory and hence this can be described here if required. For more information please see: LinkServer Flash Support [90] and also Memory Configuration and Linker Scripts [113]

Note: Memory configuration can of course also be edited after a project has been created.

8.3.11 Redlib Printf Options

The "Semihosting C Project" wizard for some parts provides two options for configuring the implementation of printf family functions that will get pulled in from the Redlib C library:

- Use non-floating-point version of printf
  - If your application does not pass floating point numbers to `printf()` family functions, you can select a non-floating-point variant of printf. This will help to reduce the code size of your application.
  - For MCUs where the wizard does not provide this option, you can cause the same effect by adding the symbol `CR_INTEGER_PRINTF` to the project properties.
- Use character- rather than string-based printf
  - By default `printf()` and `puts()` make use of `malloc()` to provide a temporary buffer on the heap in order to generate the string to be displayed. Enable this option to switch to using "character-by-character" versions of these functions (which do not require heap space). This can be useful, for example, if you are retargeting printf() to write out over a UART – since in this case it is pointless creating a temporary buffer to store the whole string, only to print it out over the UART one character at a time.
  - For MCUs where the wizard does not provide this option, you can cause the same effect by adding the symbol `CR_PRINTF_CHAR` to the project properties.

Note: if you only require the display of fixed strings, then using `puts()` rather than `printf()` will noticeably reduce the code size of your application.

For more information see C/C++ Library Support [103]

8.3.12 Project Created

Having selected the appropriate options, you can then click on the Finish button, and the wizard will create your project for you, together with appropriate startup code and a simple `main.c` file. Build options for the project will be configured appropriately for the MCU that you selected in the project wizard.

You should then be able to build and debug your project, as described in Section 9.5 and Chapter 10.

9. Importing Example Projects (from the file system)

MCUXpresso IDE supports two schemes for importing examples:

- From SDKs – see the **QuickStart** Panel -> Import SDK example(s). See Importing Examples Projects (from SDK) [45]
- From the filing system – see the **QuickStart** Panel -> Import project(s) from file System
  - this option is discussed below:

Note: This option can also be used to import projects exported from MCUXpresso IDE. See Exporting Projects [67]

MCUXpresso IDE installs with a large number of example projects for preinstalled parts, that can be imported directly into a workspace: These are located at:

```
<install_dir>\ide\Examples
```

and consist of:

- CMSIS-DSPLIB
  - a suite of common signal processing functions for use on Cortex-M processor based devices.
- CodeBundles for LPC800 family
  - which consist of software examples to teach users how to program the peripherals at a basic level.
- FlashDrivers
  - example projects to create Flash driver used by LinkServer
- Legacy
  - a range of historic examples and drivers including CMSIS / Peripheral Driver Library
- LPCOpen
  - High quality board and chip support libraries for LPC MCUs, plus example projects

9.1 Code Bundles for LPC800 Family devices

The LPC800 Family of MCUs are ideal for customers who want to make the transition from 8 and 16-bit MCUs to the Cortex M0/M0+. For this purpose, we’ve created Code Bundles which consist of software examples to teach users how to program the peripherals at a basic level. The examples provide register level peripheral access, and direct correspondence to the memory map in the MCU User Manual. Examples are concise and accurate explanations are provided within the readmes and source file comments. Code Bundles for LPC800 family devices are made available at the time of the series product launch, ready for use with a range of tools including MCUXpresso IDE.

More information on code bundles together with latest downloads can be found at:

https://www.nxp.com/LPC800-Code-Bundles

9.2 LPCOpen Software Drivers and Examples

LPCOpen is an extensive collection of free software libraries (drivers and middleware) and example programs that enable developers to create multifunctional products based on LPC microcontrollers. Access to LPCOpen is free to all LPC developers.

Amongst the features of LPCOpen are:

- MCU peripheral device drivers with meaningful examples
- Common APIs across device families
- Commonly needed third party and open source software ports
- Support for Keil, IAR and LPCXpresso/MCUXpresso IDE toolchains

LPCOpen is thoroughly tested and maintained. The latest LPCOpen software now available provides:

- MCU family-specific download package
- Support for USB ROM drivers
- Improved code organization and drivers (efficiency, features)
- Improved support for the MCUXpresso IDE

CMSIS / Peripheral Driver Library / code bundle software packages are still available, from within your MCUXpresso IDE install directory in **\ide\Examples\Legacy** . But generally, these should only be used for existing development work. When starting a new evaluation or product development, we would recommend the use of LPCOpen if available.

More information on LPCOpen together with package downloads can be found at:

http://www.nxp.com/lpcopen

9.3 Importing an Example Project

To import an example project from the file system, locate the **QuickStart** panel and select 'Import projects from Filesystem'

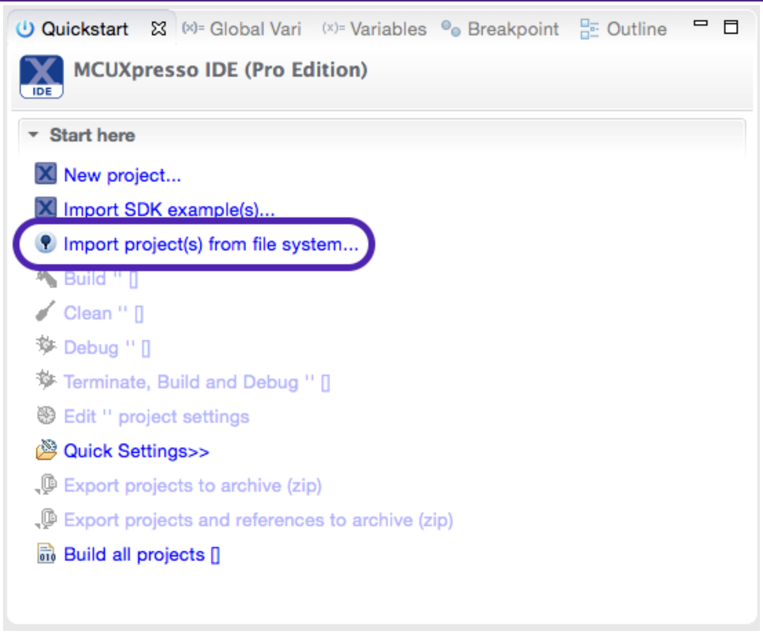
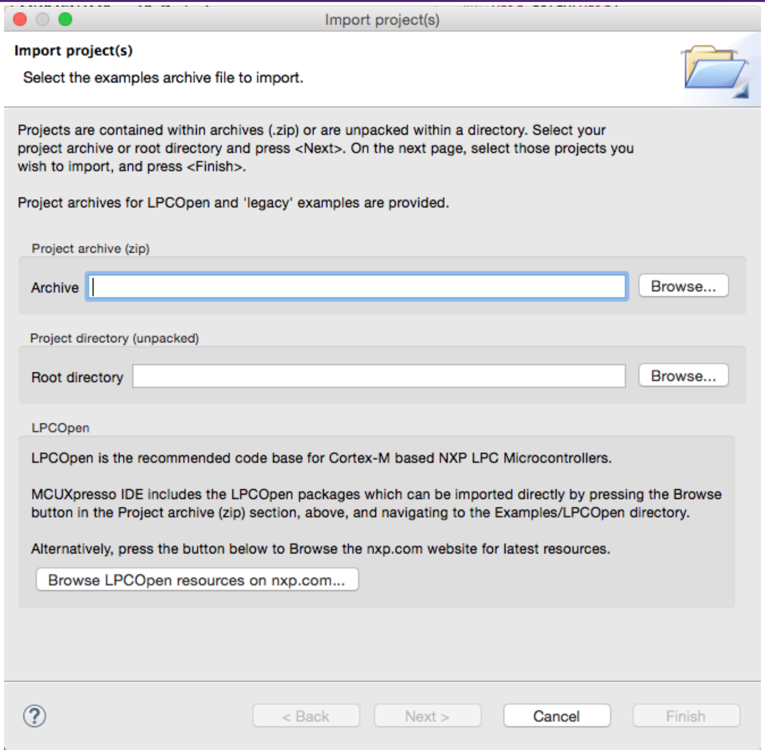


Figure 9.1. Importing project(s)

From here you can browse the file system.


Import project(s)

Import project(s)

Select the examples archive file to import.

Projects are contained within archives (.zip) or are unpacked within a directory. Select your project archive or root directory and press <Next>. On the next page, select those projects you wish to import, and press <Finish>.

Project archives for LPCOpen and 'legacy' examples are provided.

Project archive (zip)

Archive [ ] Browse...

Project directory (unpacked)

Root directory [ ] Browse...

LPCOpen

LPCOpen is the recommended code base for Cortex-M based NXP LPC Microcontrollers.

MCUXpresso IDE includes the LPCOpen packages which can be imported directly by pressing the Browse button in the Project archive (zip) section, above, and navigating to the Examples/LPCOpen directory.

Alternatively, press the button below to Browse the nxp.com website for latest resources.

Figure 9.2. Importing examples

- **Browse** to locate Examples stored in zip archive files on your local system. These could be archives that you have previously downloaded (for example LPCOpen packages from http://www.nxp.com/lpcopen or the supplied, but deprecated, sample code located within the Examples/Legacy subdirectory of your MCUXpresso IDE installation).
- **Browse** to locate projects stored in directory form on your local system (for example, you can use this to import projects from a different Workspace into the current Workspace).
- **Browse LPCOpen resources** to visit http://www.nxp.com/lpcopen and download an appropriate LPCOpen package for your target MCU. This option will automatically open a web browser onto a suitable links page.

To demonstrate how to use the Import Project(s) functionality, we will now import the LPCOpen examples for the LPCXpresso4337 development board.

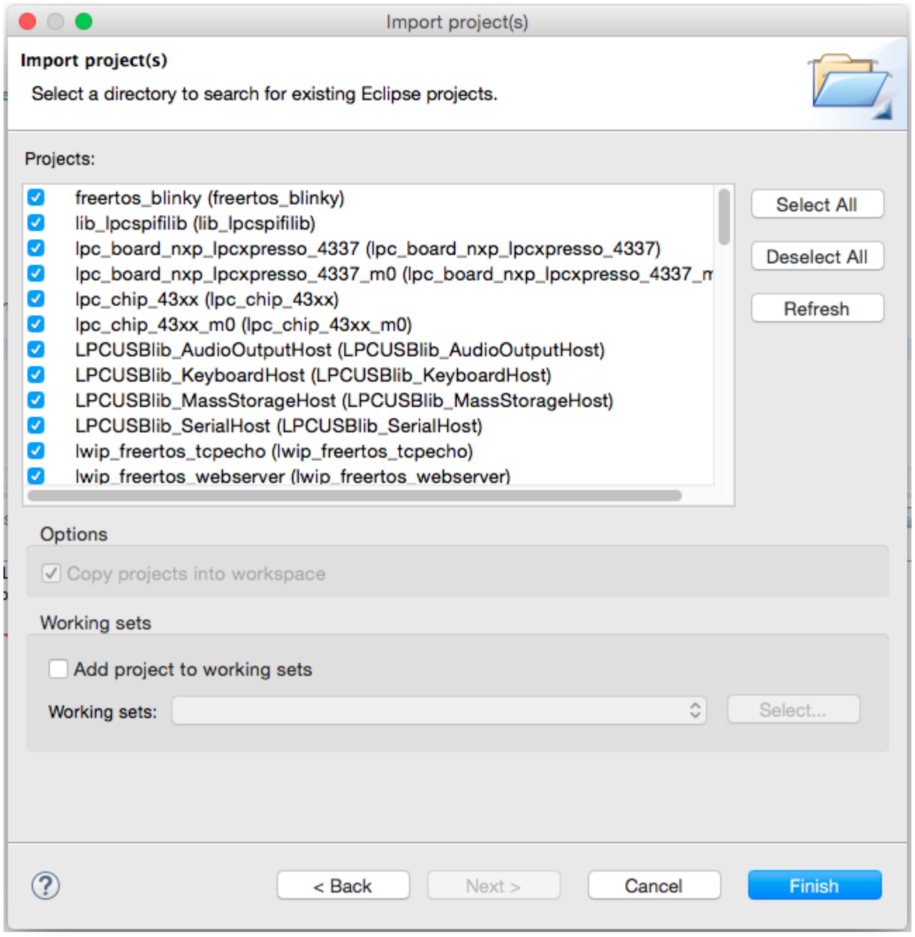
9.3.1 Importing Examples for the LPCXpresso4337 Development Board

First of all, assuming that you have not previously downloaded the appropriate LPCOpen package, click on **Browse LPCOpen Resources**, which will open a web browser window. Click on **LPC4300 Series**, and then locate **NXP LPCXpresso4337**, and then download **2.xx** version for LPCXpresso Toolchain (LPCOpen packages created for LPCXpresso IDE are compatible with MCUXpresso IDE).

Note: LPCOpen Packages for the LPC4337 are preinstalled and located at:

```
<install_dir>\ide\Examples\LPCOpen\...
```

Once the package has downloaded, return to the Import Project(s) dialog and click on the **Browse** button next to **Project archive (zip)**; then locate the LPCOpen LPCXpresso4337 package archive previously downloaded. Select the archive, click **Open** and then click **Next**. You will then be presented with a list of projects within the archive, as shown in Figure 9.3.



Import project(s)

Import project(s)

Select a directory to search for existing Eclipse projects.

Projects:

- [✓] freertos_blinky (freertos_blinky)
- [✓] lib_lpcspifilib (lib_lpcspifilib)
- [✓] lpc_board_nxp_lpcxpresso_4337 (lpc_board_nxp_lpcxpresso_4337)
- [✓] lpc_board_nxp_lpcxpresso_4337_m0 (lpc_board_nxp_lpcxpresso_4337_m
- [✓] lpc_chip_43xx (lpc_chip_43xx)
- [✓] lpc_chip_43xx_m0 (lpc_chip_43xx_m0)
- [✓] LPCUSBlib_AudioOutputHost (LPCUSBlib_AudioOutputHost)
- [✓] LPCUSBlib_KeyboardHost (LPCUSBlib_KeyboardHost)
- [✓] LPCUSBlib_MassStorageHost (LPCUSBlib_MassStorageHost)
- [✓] LPCUSBlib_SerialHost (LPCUSBlib_SerialHost)
- [✓] lwip_freertos_tcpecho (lwip_freertos_tcpecho)
- [✓] lwip_freertos_webserver (lwip_freertos_webserver)

Options

[✓] Copy projects into workspace

Working sets

[ ] Add project to working sets

Working sets:

Figure 9.3. Selecting projects to import

Select the projects you want to import and then click **Finish**. The examples will be imported into your Workspace.

Note: generally, it is a good idea to leave all projects selected when doing an import from a zip archive file of examples. This is certainly true the first time you import an example set, when you will not necessarily be aware of any dependencies between projects. In most cases, an archive of projects will contain one or more library projects, which are used by the actual application projects within the examples. If you do not import these library projects, then the application projects will fail to build.

9.4 Exporting Projects

MCUXpresso IDE provides the following export options from the **QuickStart** panel:

- Export project(s) to archive (zip)
- Export project(s) and references to archive (zip)
    - choose this option to export project(s) and automatically also export referenced libraries

To export one or more projects, first select the project(s) in the **Project Explorer** then from the **QuickStart** Panel -> Export project(s) to archive (zip). This will launch a filer window. Simply select the destination and enter a name for the archive to be exported then click ‘OK’.

9.5 Building Projects

Building the projects in a workspace is a simple case of using the **Quickstart Panel** to “Build all projects”. Alternatively, a single project can be selected in the ‘Project Explorer’ View and built. **Note** that building a single project may also trigger a build of any associated library projects.

9.5.1 Build Configurations

By default, each project will be created with two different “build configurations”: **Debug** and **Release**. Each build configuration will contain a distinct set of build options. Thus a **Debug** build will typically compile its code with optimizations disabled (`-O0`) and **Release** will compile its code optimizing for minimum code size (`-Os`). The currently selected build configuration for a project will be displayed after its name in the **QuickStart** Panel’s Build/Clean/Debug options.

For more information on switching between build configurations, see How do I switch between Debug and Release builds? [163]

10. Debugging a Project

This chapter describes many of the common debug features supported by the debug solutions within MCUXpresso IDE. Please also refer to the chapter Debug Solutions Overview [11] for more details of supported debug solutions and management of debug operations.

10.1 Debugging overview

A debug operation requires a physical connection between the host computer and the target MCU via a debug probe. The debug probe translates the high level commands provided by the host into the appropriate low level operations supported on the target MCU.

This connection to the debug probe is usually made via USB to the host computer (although IP probes from P&E and SEGGER are also supported). Some debug probes such as LPC-Link2 or SEGGER J-Link *Plus* are separate physical devices, however many LPCXpresso, Freedom and Tower boards also incorporate a built in debug probe accessed by one of the development boards USB connections.

Note: If a separate debug probe is used, you must ensure that the appropriate cables are used to connect the debug probe to the target development board, and that the target is powered.

Typically, an on board debug probe connection will also provide power to the development board and target MCU. In contrast, an external debug probe will not usually power the target, and a second connection (often USB) will be required to provide power to the board and MCU. Some external debug probes such as the LPC-Link2 can also provide power to the target board – this is enabled by connecting the link *JP2*. For other debug probes, refer to their supplied documentation.

External debug probes will usually provide superior features and performance compared to on-board debug probes, however please note that LPCXpresso V2 and V3 boards incorporate a full featured LPC-Link2 debug probe.

Note: Some LPCXpresso development boards have two USB connectors fitted. Make sure that you have connected the lower connector marked DFU-Link.

Note: Many Freedom and Tower boards also have two USB connectors fitted. Make sure that you have connected to the one marked ‘OpenSDA’ - this is usually (but not always) marked on the board. If in doubt, the debug processor used on these designs is a Kinetis K20 MCU, it is approximately 6mm square. The USB nearest this MCU will be the OpenSDA connection.

To start debugging a project on your target MCU, simply highlight the appropriate project in the ‘Project Explorer’, and then in the **Quickstart Panel** click on **Debug 'Project Name'**, as in Figure 10.1, alternatively click the blue bug icon to perform the same action.

Note: The green bug icon should not be used because this invokes the standard Eclipse debug operation and so skips certain essential MCUXpresso IDE debug steps.

By default, this operation will first build the selected project and (assuming there are no build errors) launch a debug probe discovery operation (see next section).

Note: This default behaviour can be changed by editing the Workspace preference located at *Preferences -> Run/Debug -> Launching -> Build (if required) before launching*. For individual projects, the *Main* tab of the launch configuration allows the workspace preference to be overridden.

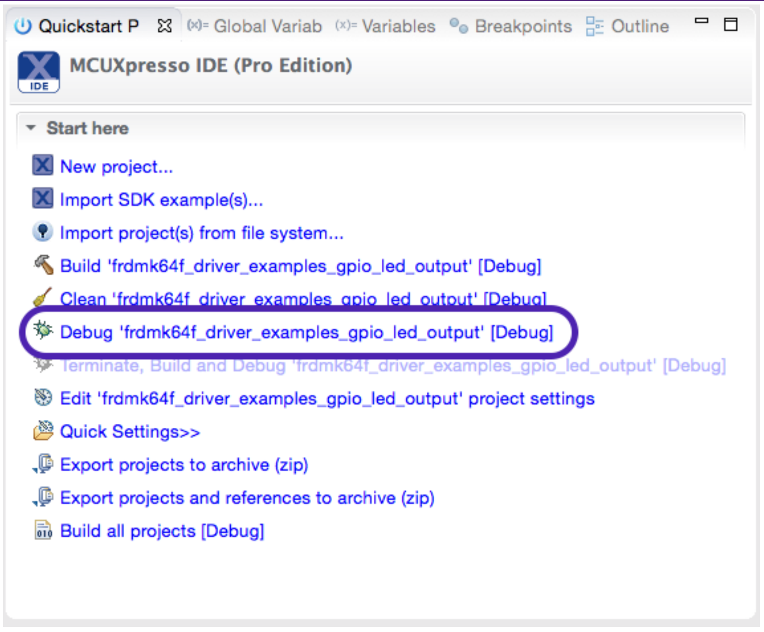

Quickstart P
Global Variab
Variables
Breakpoints
Outline
MCUXpresso IDE (Pro Edition)
Start here
New project...
Import SDK example(s)...
Import project(s) from file system...
Build 'frdmk64f_driver_examples_gpio_led_output' [Debug]
Clean 'frdmk64f_driver_examples_gpio_led_output' [Debug]
Debug 'frdmk64f_driver_examples_gpio_led_output' [Debug]
Terminate, Build and Debug 'frdmk64f_driver_examples_gpio_led_output' [Debug]
Edit 'frdmk64f_driver_examples_gpio_led_output' project settings
Quick Settings>>
Export projects to archive (zip)
Export projects and references to archive (zip)
Build all projects [Debug]

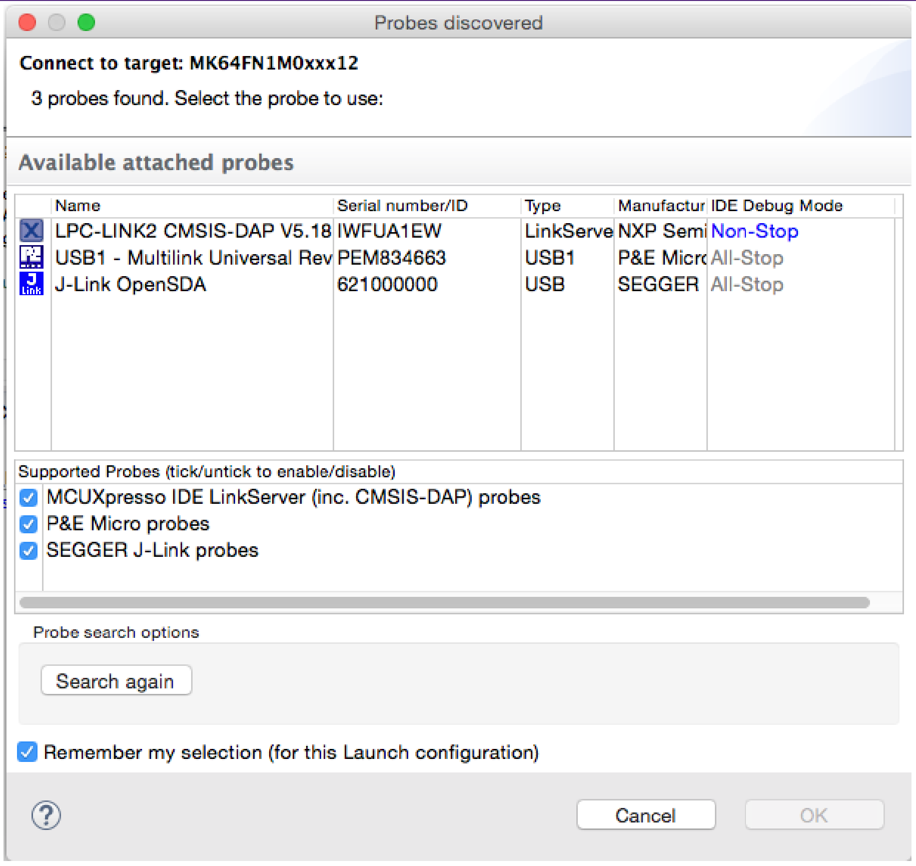
Figure 10.1. Launching a debug session

Note: Previously debugged projects will contain launch configuration files. Please see the section A note about Launch Configuration files [12] for more information.

Once a debug probe has been selected (and 'OK' clicked) the binary contents of the .axf file will automatically be downloaded to the target via the debug probe connection. Typically, projects are built to target MCU Flash memory, and in these cases, a suitable Flash driver will automatically be selected to perform the Flash programming operation. Next a default breakpoint will be set on the first instruction in `main()`, the application will be started (by simulating a processor reset), and code will be executed until the default breakpoint is hit.

10.1.1 Debug Probe Selection Dialog (Probe Discovery)

The first time you debug a project, the IDE will perform a probe discovery operation and display the discovered Debug Probes for selection. This will show all supported probes that are attached to your computer. In the example shown in Figure 10.2, a LinkServer (LPC-Link2), a P&E Micro Multilink and also a J-Link (OpenSDA) probe have been found.



Probes discovered

Connect to target: MK64FN1M0xxx12

3 probes found. Select the probe to use:

Available attached probes

Name	Serial number/ID	Type	Manufactur	IDE Debug Mode
LPC-LINK2 CMSIS-DAP V5.18	IWFUA1EW	LinkServe	NXP Semi	Non-Stop
USB1 - Multilink Universal Rev	PEM834663	USB1	P&E Micrc	All-Stop
J-Link OpenSDA	621000000	USB	SEGGER	All-Stop

Supported Probes (tick/untick to enable/disable)

- ✓ MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes
- ✓ P&E Micro probes
- ✓ SEGGER J-Link probes

Probe search options

✓ Remember my selection (for this Launch configuration)

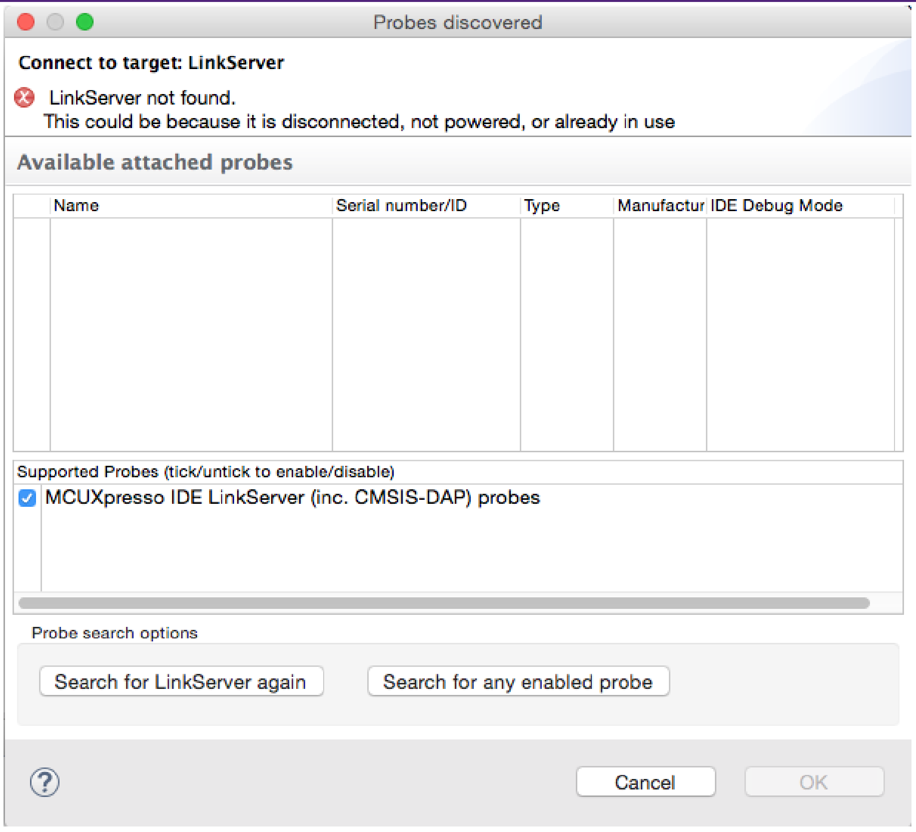
Figure 10.2. Attached probes: debug emulator selection

MCUXpresso IDE supports unique debug probe association.

Debug probes can return an ID (Serial number) that is used to associate a particular debug probe with a particular project. Some debug probes will always return the same ID, however debug probes such as the LPC-Link2 will return a unique ID for each probe – in our example **IWFUA1EW**.

For any future debug sessions, the stored probe selection will be automatically used to match the project being debugged with the previously used debug probe. This greatly simplifies the case where multiple debug probes are being used.

However, if a debug operation is performed and the previously remembered debug probe cannot be found, then a debug probe discovery operation will be performed from within the same family e.g. LinkServer, P&E or SEGGER.


Probes discovered

Connect to target: LinkServer

LinkServer not found.
This could be because it is disconnected, not powered, or already in use

Available attached probes

	Name	Serial number/ID	Type	Manufactur	IDE Debug Mode

Supported Probes (tick/untick to enable/disable)

☑ MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes

Probe search options

Figure 10.3. LPC-Link2 no longer connected

This might have been because you had forgotten to connect the probe, in which case simply connect it to your computer and select **Search again**. If you are using a different debug probe from the same family of debug probes, simply select the new probe and this will replace the previously selected probe.

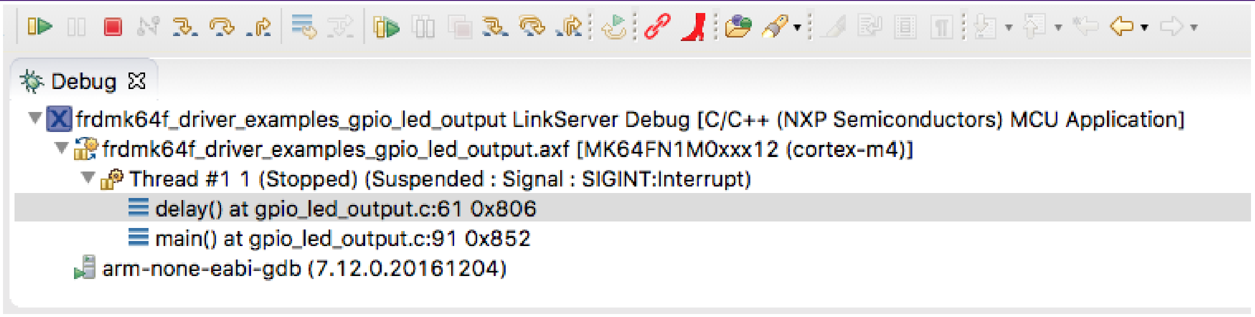
Notes:

- The “Remember my selection” option is enabled by default in the Debug Emulator Selection Dialog, and will cause the selected probe to be stored in the launch configuration for the current configuration (typically Debug or Release) of the current project. You can thus remove the probe selection at any time by simply deleting the launch configuration.
- You will need to select a probe for each project that you debug within a Workspace (as well as for each configuration within a project).
- If you wish to debug a project using a different family of debug probe, then the simplest option is to delete the launch configuration files associated with the project and start a debug operation. Please see the section A note about Launch Configuration files [12] for more information.

10.1.2 Controlling Execution

When you have started a debug session a default breakpoint is set on the first instruction in `main()`, the application is started (by simulating a processor reset), and code is executed until the default breakpoint is hit.

Program execution can now be controlled using the common debug control buttons, as listed in Table 10.1, which are displayed on the global toolbar. The call stack is shown in the Debug View, as in Figure 10.4.



Debug

frdmk64f_driver_examples_gpio_led_output LinkServer Debug [C/C++ (NXP Semiconductors) MCU Application]
frdmk64f_driver_examples_gpio_led_output.axf [MK64FN1M0xxx12 (cortex-m4)]
Thread #1 1 (Stopped) (Suspended : Signal : SIGINT:Interrupt)
delay() at gpio_led_output.c:61 0x806
main() at gpio_led_output.c:91 0x852
arm-none-eabi-gdb (7.12.0.20161204)

Figure 10.4. Debug controls and Debug Call Stack

Table 10.1. Program execution controls

Button	Description	Keyboard Shortcut
	Restart program execution (from reset)	
	Run/Resume the program	F8
	Pause Execution of the running program	
	Terminate the debug Session	Ctrl + F2
	Run, Pause, Terminate all debug sessions	
	Step over a C/C++ line	F6
	Step into a function	F5
	Return from a function	F7
	Step in, over, out all debug sessions	
	Show disassembled instructions	

Note: The debug controls for ‘all’ debug sessions will perform identically to their single session counterparts if only one debug session exists.

Note: Typically a user will only have a single active debug session. However if there is more than one debug session, the active session can be chosen by clicking within the debug call stack within the Debug view.

Setting a breakpoint

To set a breakpoint, simply double-click on the left margin area of the line on which you wish to set the breakpoint (before the line number).

Restarting the application

If you hit a breakpoint or pause execution and want to start execution of the application from the beginning again, you can do this using the **Restart** button.

Stopping debugging

To stop debugging just press the **Stop** button.

If you are debugging using the **Debug Perspective**, then to switch back to the **C/C++ Perspective** when you stop your debug session, just click on the **C/C++** tab in the upper right area of the MCUXpresso IDE (as shown in Figure 2.2).

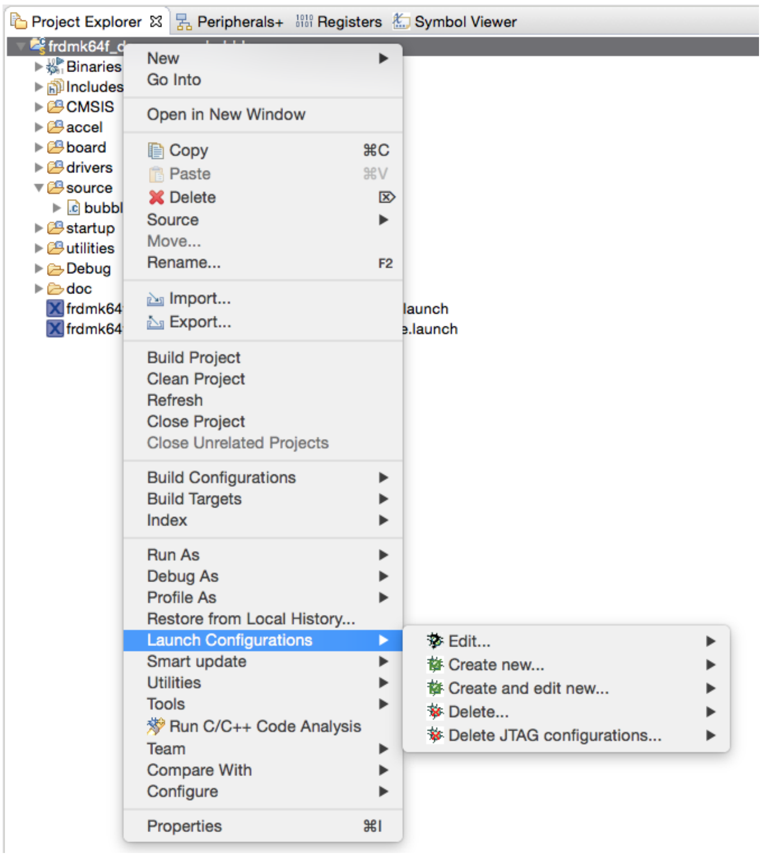
10.2 Launch Configurations

Launch Configuration files will be automatically created within the root directory of a project the first time a debug operation is performed. They will typically be named:

```
{projname}{debug solution}Debug.launch
{projname}{debug solution}Release.launch
```

A file will be created for each build variant, and used to store the settings for the debug connection for that build configuration.

Normally, there is no need to edit launch configurations, as the default settings created by the IDE will be suitable. However, in some circumstances, you may need to manage them – typically under direction from an FAQ. In such cases this can be done via the “Launch Configurations” entry on the context sensitive menu available from the Project Explorer view...



Note: to view the contents or edit an existing launch configuration file, you can also simply double click it.

A number of options are available here:

Edit...

- Allows various debug settings to be modified
  - Typically not required since the default options will be correct for most debug operations

Create new...

- Create a launch configurations for a particular debug solution, if they do not already exist.
  - Normally you will not need this option as it is carried out automatically the first time that you debug your project. However, if you want the flexibility to debug a project with different debug solutions for example, LinkServer and SEGGER, then both sets of launch configurations can be created. On the next debug operation, the user can select the launch configuration to use for that session.

Create and edit new...

- Allows new launch configurations to be created and immediately opened for editing.

Delete...

- Allows the launch configurations for the selected project (or projects) to be deleted.
- This can be useful as it allows you to put the debug connection settings back to the default after making modifications for some reason, or if you are moving your project to a new version of the tools, and want to ensure that your debug settings are correct for this version of the tools.

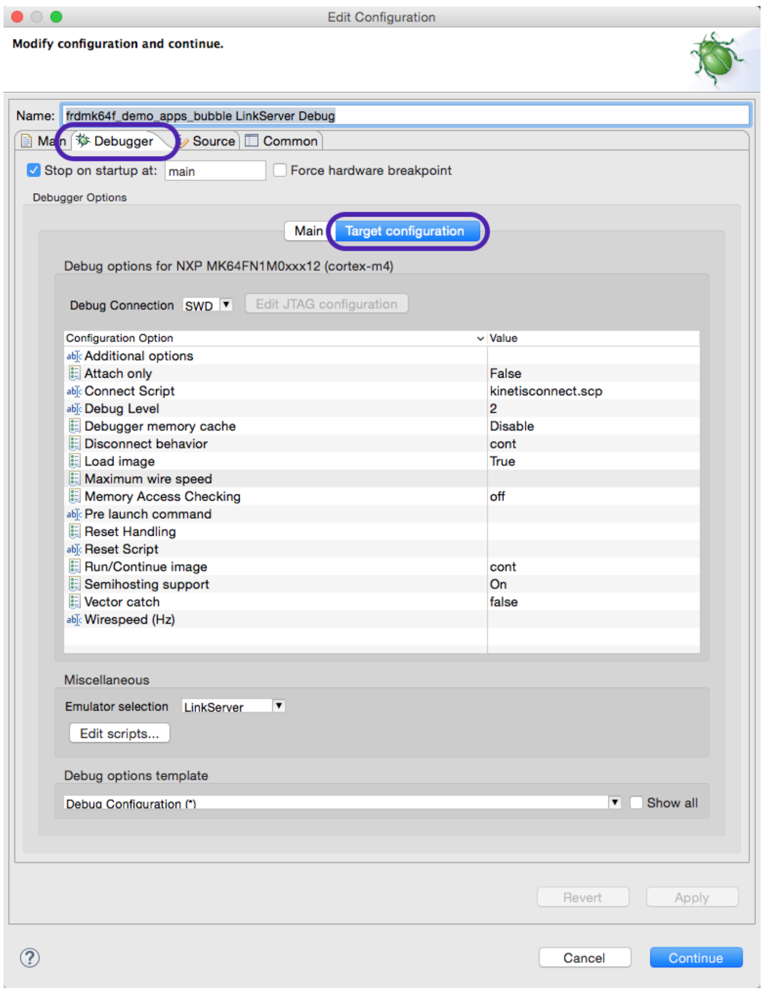
Delete JTAG Configuration...

- Allows the JTAG configuration files for the selected project (or projects) to be deleted. These files are stored in the Debug/Release subdirectories.

10.2.1 Editing a Launch Configuration (LinkServer)

WARNING: - Modifying the default settings for a launch configuration can prevent a successful debug connection from being made. Make changes with care!

After selecting the "Edit..." or "Create and edit New" launch configuration menu entry, you will then see a new dialog box pop up, which looks like the following...



Modify configuration and continue.

Name: frdmk64f_demo_apps_bubble LinkServer Debug

Stop on startup at: main — Force hardware breakpoint

Debugger Options

Debug options for NXP MK64FN1M0xxx12 (cortex-m4)

Debug Connection: SWD

Configuration Option	Value
Additional options	
Attach only	False
Connect Script	kinetisconnect.scp
Debug Level	2
Debugger memory cache	Disable
Disconnect behavior	cont
Load image	True
Maximum wire speed	
Memory Access Checking	off
Pre launch command	
Reset Handling	
Reset Script	
Run/Continue image	cont
Semihosting support	On
Vector catch	false
Wirespeed (Hz)	

Miscellaneous

Emulator selection: LinkServer

Debug options template

Debug Configuration (*) — Show all

Most settings that you may need to modify can be found in the Debugger tab, in the Target configuration sub-tab (as shown in the above screenshot).

Some examples of modifications that you may need to make in particular circumstances are:

- Changing the initial breakpoint on debug startup
  - When the debugger starts, it automatically sets an initial (temporary) breakpoint on the first statement in main(). If desired, you can change where this initial breakpoint is set, or even remove it completely.
- Modifying the Debugger connect behavior
  - via a Connect Script e.g. kinetisconnect.scp
- Connecting to a target via JTAG rather than SWD
  - if supported by the target, you can edit the Debug type
- Connecting to a running target
  - set Attach only to True

10.3 Common Debug Operations and Launch Configurations

Where possible MCUXpresso IDE attempts to provide a common debug experience regardless of the debug solution being used. However some debug tasks require launch configuration modifications and these will be different for each debug solution. In this section, some common debug operations are discussed for each debug solution.

10.3.1 Connecting to a running target (attach)

A typical debug session will begin by downloading code to Flash and then debugging from main() onwards. However, to explore an already running system a debug connection can be made to the target MCU without affecting the code execution (at least until the user chooses to halt the system!).

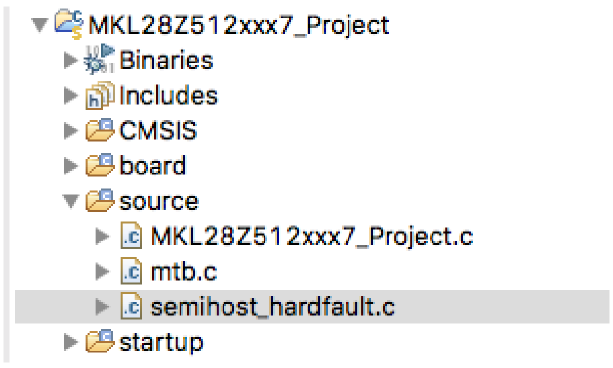
Note : Source level debug of a running target is only possible if the sources of the project to be attached exactly match the binary code running on the target.

Semihosted printf

Projects linked against semihosting libraries that perform semihosted operations e.g. printf, can not execute without a debugger connected. This is because semihosted operations make use of a BreakPoint instruction that is intercepted by the debug tools to trigger the desired behaviour (typically the printf string appearing within the IDE console). Without a debug connection, these BreakPoint instructions will not be trapped and a Hard Fault exception will occur. By default, the supplied Hard Fault handler implementation will be an infinite loop. Therefore if an ‘attach’ is performed to such a target, the user will observe the code running within the hard fault handler. To avoid this occurring, ensure that the project makes no use of semihosted operations via sending output to a UART, using the ITM feature, commenting out semihosted operations etc.

In consequence, if for example a user had created an LED blinky application that also performed semihosted printf operations, then without a debug connection the blinky would stop when the first printf was executed.

New in MCUXpresso IDE version 10.1.0: New projects and newly imported SDK example projects will automatically include a semihost hardfault handler (as can be seen in the image below). The purpose of this handler is to prevent the problem described above. Now, if a semihosted operation is performed without debug tools attached, the new semihost hardfault handler will be entered. The handler will check to see if a semihosted operation cause it to be entered and if so, simply return.



In consequence, if the user creates an LED blinky application that also performs semihosted printf operations, then without a debug connection the blinky will continue regardless of any printf operation that may occur.

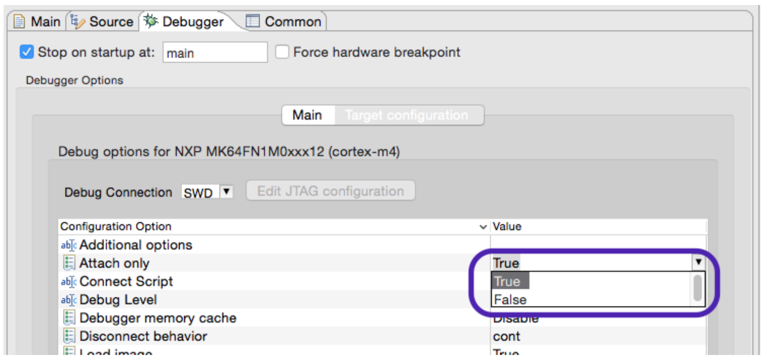
This functionality can be disabled if required by either simply deleting the handler file, or by defining a symbol:

```
__SEMIHOST_HARDFAULT_DISABLE
```

Note: Previously created projects imported into MCUXpresso IDE (such as LPCOpen projects), will not inherit this feature.

LinkServer

Edit the project launch configuration by double clicking on the launch config file, select the Debugger tab and Target configuration view, then set the 'Attach only' setting to True as below:



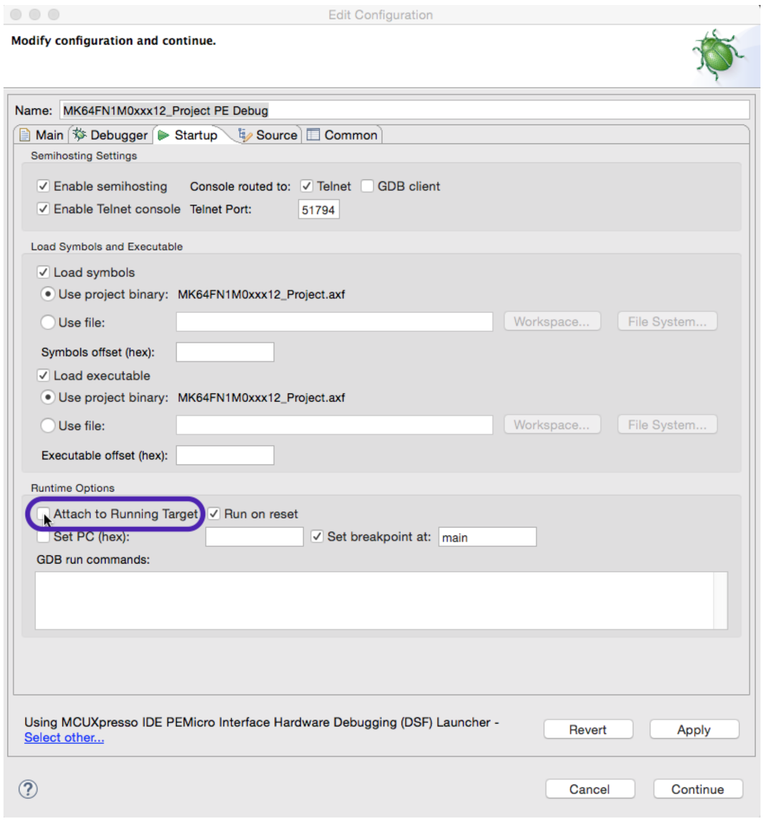
Configuration Option	Value
Additional options	
Attach only	True
Connect Script	True / False
Debug Level	
Debugger memory cache	Disable
Disconnect behavior	cont
Load image	True

When a debug connection is made, the target will continue running until it is paused. However, if the IDE Debug Mode is set to Non-Stop (the default) then Global variables values can be explored and displayed.

Other operations such as ITM console IO will also function. See the LinkServer SWO Trace Guide for further information.

P&E

Edit the project launch configuration by double clicking on the launch config file, select the Startup tab, then set the 'Attach to a running target' check box as below:



Edit Configuration

Modify configuration and continue.

Name: MK64FN1M0xxx12_Project PE Debug

Semihosting Settings

- [✓] Enable semihosting Console routed to: [✓] Telnet [ ] GDB client
- [✓] Enable Telnet console Telnet Port: 51794

Load Symbols and Executable

- [✓] Load symbols
- (●) Use project binary: MK64FN1M0xxx12_Project.axf
- ( ) Use file:
- Symbols offset (hex):
- [✓] Load executable
- (●) Use project binary: MK64FN1M0xxx12_Project.axf
- ( ) Use file:
- Executable offset (hex):

Runtime Options

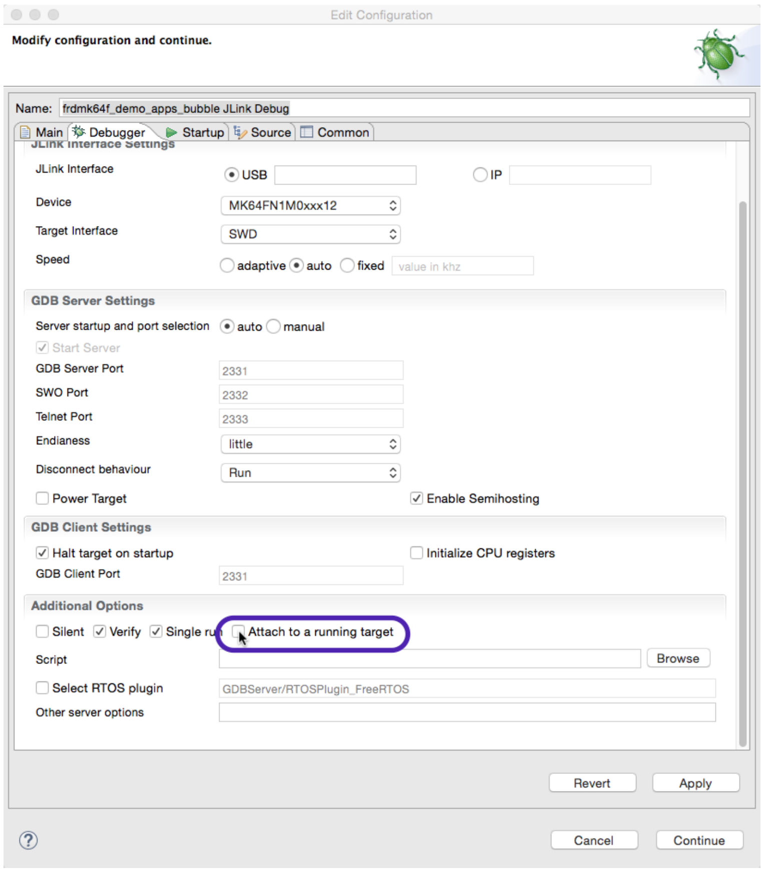
- [ ] Attach to Running Target [✓] Run on reset
- [ ] Set PC (hex): [✓] Set breakpoint at: main
- GDB run commands:

Using MCUXpresso IDE PEMicro Interface Hardware Debugging (DSF) Launcher - Select other...

When a debug connection is made, the target will continue running until it is paused.

SEGGER JLink

Edit the project launch configuration by double clicking on the launch config file, select the Debugger tab, then set the ‘Attach to a running target’ check box as below:



Modify configuration and continue.

Name: frdmk64f_demo_apps_bubble JLink Debug

JLink Interface Settings

JLink Interface: (● USB) ○ IP
Device: MK64FN1M0xxx12
Target Interface: SWD
Speed: ○ adaptive (● auto) ○ fixed value in khz

GDB Server Settings

Server startup and port selection: (● auto) ○ manual
☑ Start Server
GDB Server Port: 2331
SWO Port: 2332
Telnet Port: 2333
Endianess: little
Disconnect behaviour: Run
☐ Power Target
☑ Enable Semihosting

GDB Client Settings

☑ Halt target on startup
☐ Initialize CPU registers
GDB Client Port: 2331

Additional Options

☐ Silent ☑ Verify ☑ Single run ☐ Attach to a running target
Script
☐ Select RTOS plugin: GDBServer/RTOSPlugin_FreeRTOS
Other server options

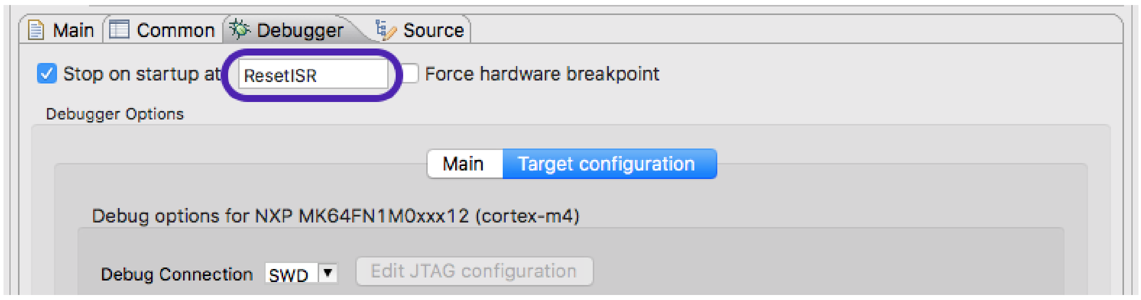
When a debug connection is made, the target will continue running until it is paused.

10.3.2 Controlling the initial breakpoint (on main)

When the debugger starts, it automatically sets an initial (temporary) breakpoint on the first statement in main(). If desired, you can change where this initial breakpoint is set, or even remove it completely. One common requirement is to debug an application from startup. The entry point (startup) in an standard example application can be identified by a symbol called ResetISR, a breakpoint can be set on this symbol to halt execution at the first instruction within an application.

LinkServer

To debug from the start of the image, edit the project launch configuration by double clicking on the launch config file, select the Debugger tab, replace main with ResetISR



☑ Stop on startup at: ResetISR ☐ Force hardware breakpoint

Debugger Options

Debug options for NXP MK64FN1M0xxx12 (cortex-m4)

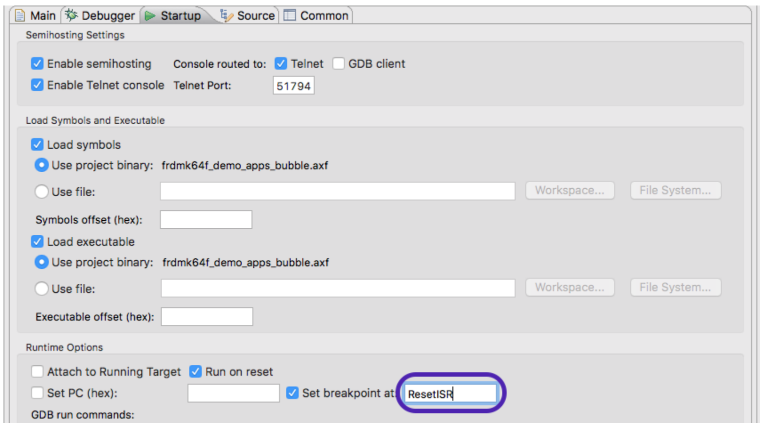
Debug Connection: SWD

When a debug connection is made, the target should halt at this symbol.

To disable the initial breakpoint, uncheck the option ‘Stop on startup at...’. To restore the original behaviour, replace the symbol ResetISR with main, and check the option ‘Stop on startup at...’. Alternatively, you could delete the launch configuration and allow the IDE to create a new one.

P&E

Edit the project launch configuration by double clicking on the launch config file, select the Startup tab, replace main with ResetISR


Main | Debugger | Startup | Source | Common

Semihosting Settings
[✓] Enable semihosting   Console routed to: [✓] Telnet [ ] GDB client
[✓] Enable Telnet console   Telnet Port: 51794

Load Symbols and Executable
[✓] Load symbols
[✓ Use project binary: frdmk64f_demo_apps_bubble.axf]
Use file: Workspace...   File System...
Symbols offset (hex):
[✓] Load executable
[✓ Use project binary: frdmk64f_demo_apps_bubble.axf]
Use file: Workspace...   File System...
Executable offset (hex):

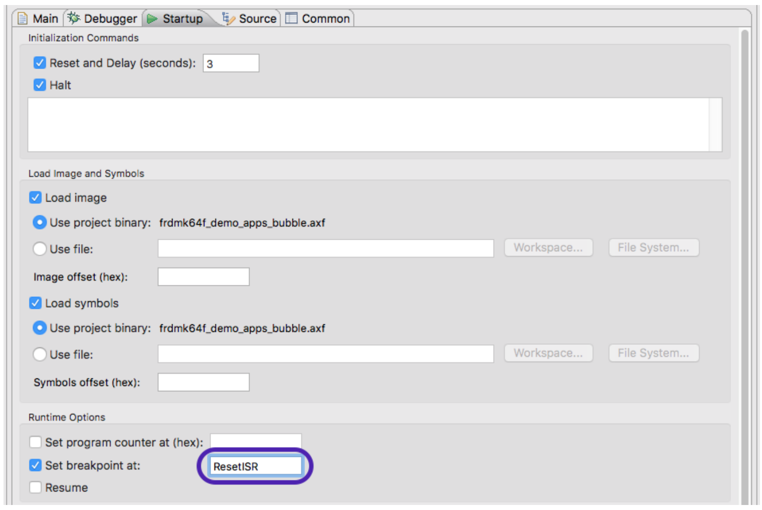
Runtime Options
[ ] Attach to Running Target   [✓] Run on reset
[ ] Set PC (hex):   [✓] Set breakpoint at: ResetISR
GDB run commands:

When a debug connection is made, the target should halt at this symbol.

To disable the intial breakpoint, uncheck the option ‘Set breakpoint at...’. To restore the original behaviour, replace the symbol ResetISR with main, and check the option ‘Set breakpoint at...’. Alternatively, you could delete the launch configuration and allow the IDE to create a new one.

SEGGER JLink

Edit the project launch configuration by double clicking on the launch config file, select the Startup tab, replace main with ResetISR


Main | Debugger | Startup | Source | Common

Initialization Commands
[✓] Reset and Delay (seconds): 3
[✓] Halt

Load Image and Symbols
[✓] Load image
[✓ Use project binary: frdmk64f_demo_apps_bubble.axf]
Use file: Workspace...   File System...
Image offset (hex):
[✓] Load symbols
[✓ Use project binary: frdmk64f_demo_apps_bubble.axf]
Use file: Workspace...   File System...
Symbols offset (hex):

Runtime Options
[ ] Set program counter at (hex):
[✓] Set breakpoint at: ResetISR
[ ] Resume

When a debug connection is made, the target should halt at this symbol.

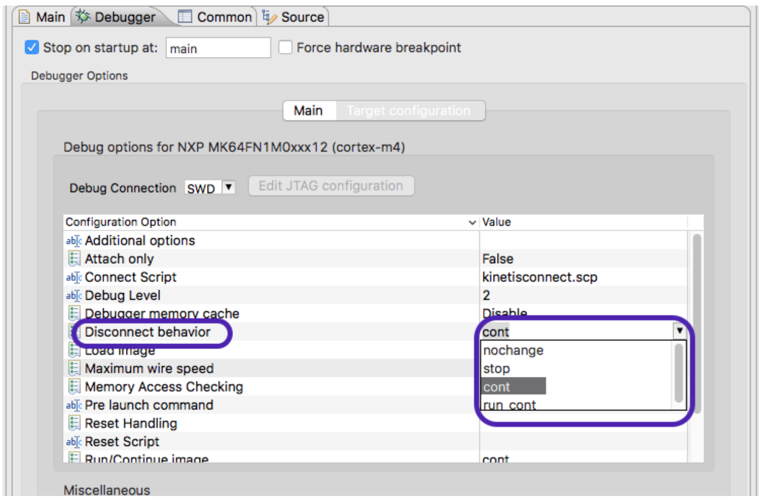
To disable the initial breakpoint, uncheck the option ‘Set breakpoint at...’. To restore the original behaviour, replace the symbol ResetISR with main, and check the option ‘Set breakpoint at...’. Alternatively, you could delete the launch configuration and allow the IDE to create a new one.

10.3.3 Disconnect behaviour

Once the user has completed a debug session, the debugger connection can be terminated via the IDE’s Terminate button! The exact behaviour of the target will depend on the particular debug solution.

LinkServer

For LinkServer, the launch configuration contains a set of options to control what the target should do when terminated. The default option is for the target to continue running from the current PC value, however this can be changed by selecting a new setting within the launch configuration.



[✓] Stop on startup at: main

Force hardware breakpoint

Debugger Options

Main | Target configuration

Debug options for NXP MK64FN1M0xxx12 (cortex-m4)

Debug Connection: SWD

Configuration Option	Value
Additional options	
Attach only	False
Connect Script	kinetisconnect.scp
Debug Level	2
Debugger memory cache	Disable
Disconnect behavior	cont
Load image	
Maximum wire speed	
Memory Access Checking	
Pre launch command	
Reset Handling	
Reset Script	
Run/Continue image	cont

nochange
stop
cont
run cont

Miscellaneous

Where:

- **nochange** - will leave the target in its current state
- **stop** - will leave the target in debug state i.e. halted
- **cont** - the default, will either start the image from its current PC value or leave it running
- **run cont** - will reset the target and let it run

P&E

The Terminate button will force the target to halt. Alternatively, for P&E debug the IDE supports another option – to disconnect and force the target to run. This can be achieved via the IDE's disconnect button.

SEGGER JLink

The target will *Run* on disconnect by default. The launch configuration option, *Disconnect behaviour* can be changed to *Halt* causing the target to halt on disconnect.

10.4 Breakpoints

When viewing source (or disassembly) during a debug session, you can toggle breakpoints by simply clicking/double clicking in the left most side of the source view, typically shown as a light blue column. This is also where the breakpoint symbol is shown when one is set. This can be done when the target is paused or running.

Breakpoints (and Watchpoints) are also displayed, and can be deleted or disabled in the Breakpoints View. If you are using the "Develop" perspective, then by default it will be in the bottom left of the MCUXpressoIDE window tabbed with the Quickstart and other views

If you have closed the Breakpoint view at some point, then you can reopen it using the "Window -> Show view" menu or 'Window -> Reset Perspective".

10.4.1 Breakpoints Resources

When debugging code running from Flash memory, the debugger is limited on how many breakpoints it can set at any time by the number of hardware breakpoint units provided by the ARM CPU within the MCU.

Note: Code located in RAM can use a different breakpoint mechanism offering the capability of essentially unlimited breakpoints.

Typically, the number of hardware breakpoints/watchpoints that can be set are as follows:

```
Cortex-M0/M0+ (LPC)   - 4 breakpoints, 2 watchpoints
Cortex-M0/M0+ (Kinetis) - 2 breakpoints, 1 watchpoints
Cortex-M3/M4/M7    - 6 breakpoints, 4 watchpoints
```

ARM does provide a level of implementation flexibilty, so always consult your MCU documentation.

If you try to set too many breakpoints/watchpoints when debugging, then the precise behaviour will depend on the debug solution you are using. For LinkServer an error of the form below will be generated.

```
15: Target error from Set break/watch
Unable to set an execution break - no resource available.
```

To fix the problem, simply remove the excess breakpoint(s).

Also remember that a breakpoint will be (temporarily) required for the initial breakpoint set by default on the function main() when you debug your application. A breakpoint may also be required (temporarily) when single stepping code.

Note: When the target is paused, any number breakpoints may be set within the source or disassembly views of the IDE, however only when the target is Resumed (Run) will the low level debug hardware attempt to set the required breakpoints. Therefore it is possible to request many more breakpoints that are supported by the target MCU leading to the error described above.

10.4.2 Skip All Breakpoints

You can use the “Skip all breakpoints” button in the Breakpoints view (or on the main toolbar) to temporarily disable all breakpoints. This can be particularly useful on parts with only a few breakpoints available, particularly when you want to reload your image, which will typically cause the default breakpoint on main() to be temporarily set again automatically by the tools.

10.5 Watchpoints

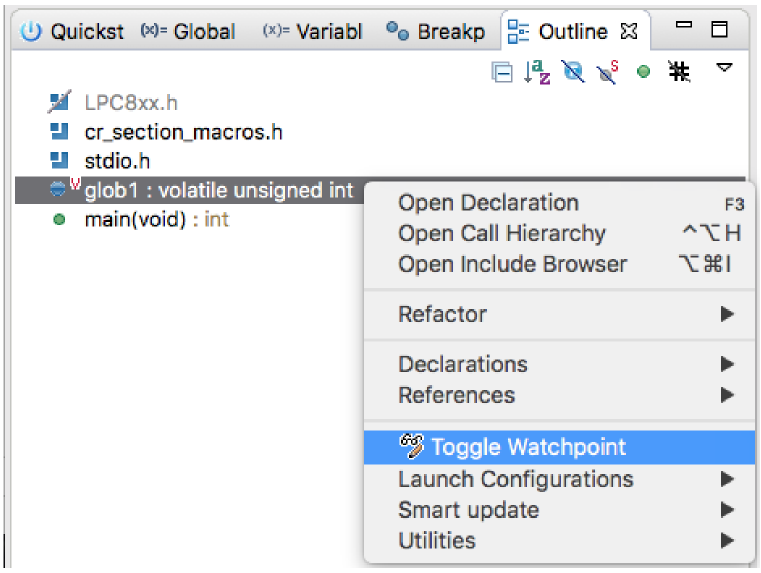
Watchpoints are Breakpoints for Data and are often referred to as Data Breakpoints. **Note**: a key difference from breakpoints is that the data event must first occur before the core will be halted.

Watchpoints are a powerful aid to debugging allowing the monitoring of global variables, peripheral accesses, stack depth etc. The number of watchpoints that can be set varies with the MCU family and implementation.

Watchpoints are implemented using watchpoints units which are data comparators within the debug architecture of an MCU/CPU and sit close to the processor core. When programmed they will monitor the processor’s address lines and other signals for the specific event of interest. This hardware is able to monitor data accesses performed by the CPU and force it to halt when a particular data event has occurred.

The method for setting Watchpoints is rather more hidden within the IDE than some other debugging features. One of the easiest ways to set a Watchpoint is to use the Outline View, which by default this will be located within the IDE Quickstart panel.

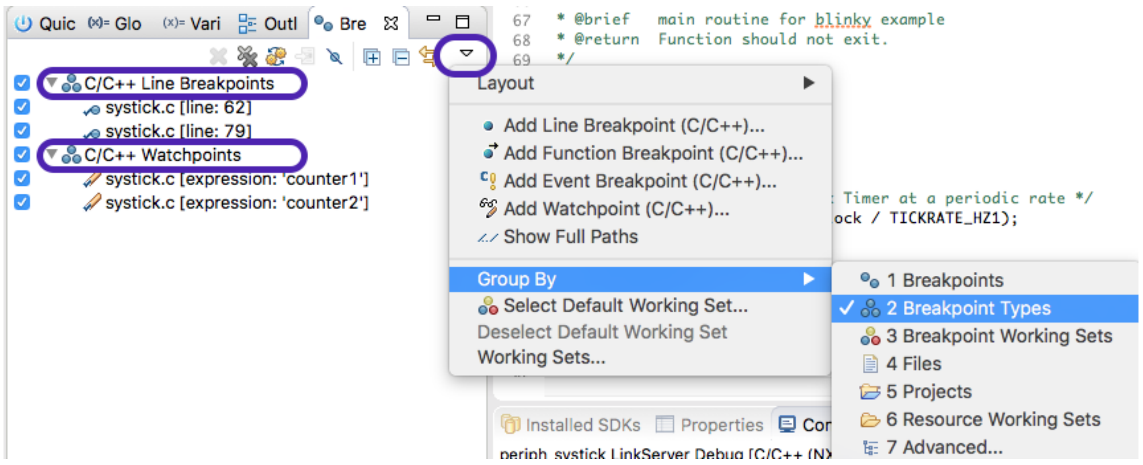
From this view you can locate global and static variables then simply select **Toggle Watchpoints**.



Once set, they will appear within the Breakpoint pane alongside any breakpoints that have been set.

Watchpoints can be configured to halt the CPU on a Read (or Load), Write (or Store), or both. Since watchpoints 'watch' accesses to memory, they are suitable for tracking accesses to global or static variables, and any data accesses to memory including those to memory mapped peripherals.

Note : To easily distinguish between Breakpoints and Watchpoints within the Breakpoint view, you can choose to group entries by Breakpoint type. From within the Breakpoints view, click the Eclipse Down Arrow Icon Menu, then you can select to Group By Breakpoint Types as shown below:


Quic
Glo
Vari
Outl
Bre
C/C++ Line Breakpoints
systick.c [line: 62]
systick.c [line: 79]
C/C++ Watchpoints
systick.c [expression: 'counter1']
systick.c [expression: 'counter2']
* @brief main routine for blinky example
* @return Function should not exit.
*/
Layout
Add Line Breakpoint (C/C++)...
Add Function Breakpoint (C/C++)...
Add Event Breakpoint (C/C++)...
Add Watchpoint (C/C++)...
Show Full Paths
Group By
Select Default Working Set...
Deselect Default Working Set
Working Sets...
1 Breakpoints
2 Breakpoint Types
3 Breakpoint Working Sets
4 Files
5 Projects
6 Resource Working Sets
7 Advanced...
Timer at a periodic rate */
ock / TICKRATE_HZ1);
Installed SDKs
Properties
Cor
periph_systick LinkServer Debug [C/C++ (NX

As you can see from the above graphic, the option to set a Watchpoint is also available directly from the Breakpoint view. When set from here, you will be offered an unpopulated dialogue – simply entering an address will cause a watchpoint to be created, monitoring accesses to that location.

Another place to set Watchpoints within the IDE is from the context sensitive menu within a Memory view.

Note: Watchpoint resources are shared with other debug features, in particular an SWO Data Watch item will require a dedicated watchpoint unit to monitor the value.

Note: Due to the way watchpoints are implemented, any monitored access will be performed by the CPU before a halts occurs (unlike instruction breakpoints – which halt the CPU before the underlying instruction executes). When a watchpoint is hit you will see some 'skid' beyond the instruction that performed the watched data access. If the instruction after the data access changes program flow (e.g. a branch or function return), then the IDE may not show the instruction or statement that caused the CPU to halt.

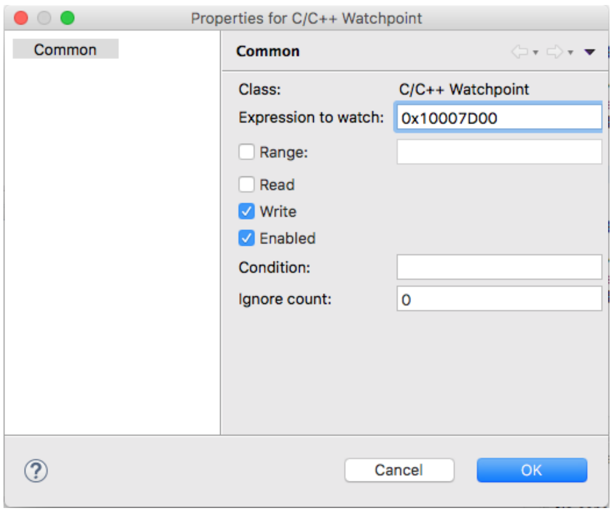
Note: Application initialisation performed by the C library may write to monitored memory locations, therefore you may see your application halting during startup if watchpoints have been set on initialised global data.

10.5.1 Using Watchpoints to monitor stack depth

Watchpoints provide a very simply way of monitoring stack depth when an application is running.

Stacks on ARM based processors use a Full Descending scheme and so have the potential to descend into areas of memory used for other purposes (typically holding global data). Establishing the maximum depth of an applications stack can be a challenge especially since any memory corruption due to excessive stack use may not be immediately apparent. Watchpoints may be used to monitor and trap the stack exceeding a particular depth and then review the area of code that caused the occurrence.

The graphic below shows the use of the breakpoint view feature *Add Watchpoint (C/C++) ...* where an address has been selected to watch for the Stack reaching 0x10007D00.



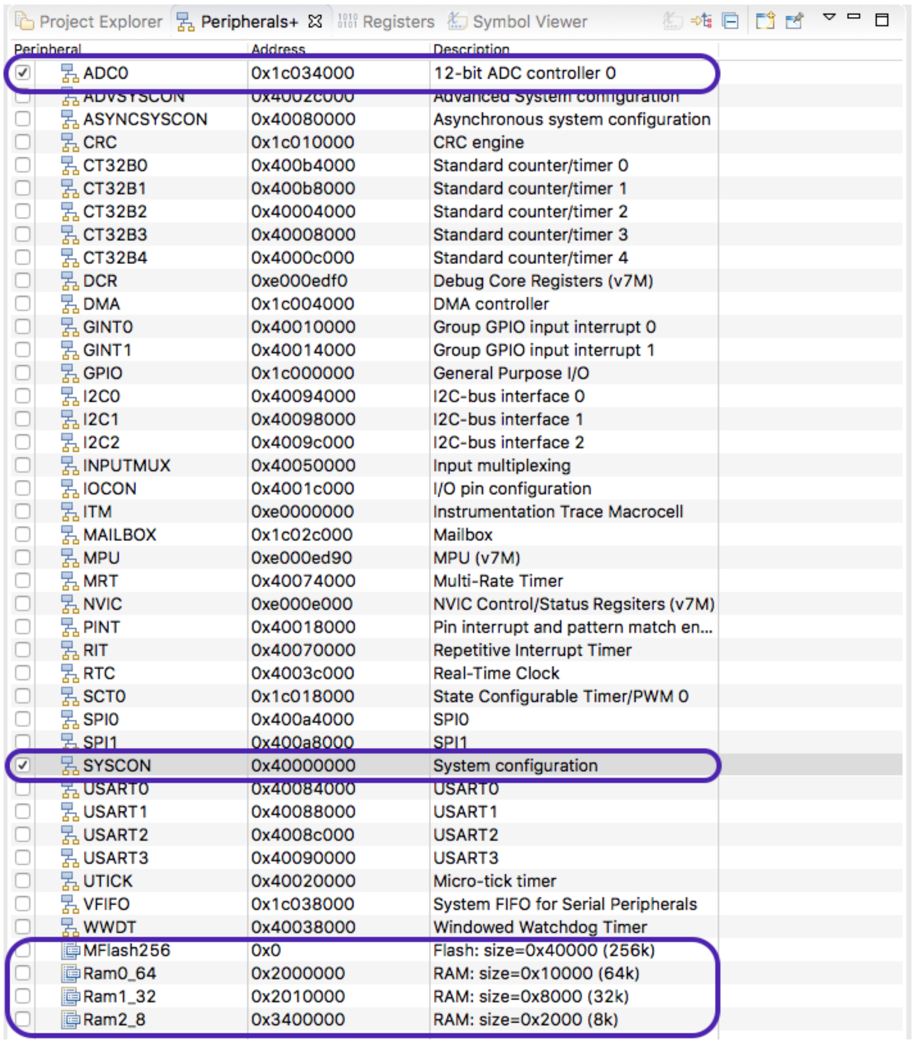
Class: C/C++ Watchpoint
Expression to watch: 0x10007D00
☐ Range:
☐ Read
☑ Write
☑ Enabled
Condition:
Ignore count: 0

10.6 Peripherals

Peripherals is a generic term referring to both core peripherals, for example the System Timer (SysTick) and SOC/MCU peripherals such as an ADC or UART. In both instances these hardware blocks are exposed within the MCUs address space (known as memory mapped peripherals) and so can be interrogated by accesses to their specific memory locations.

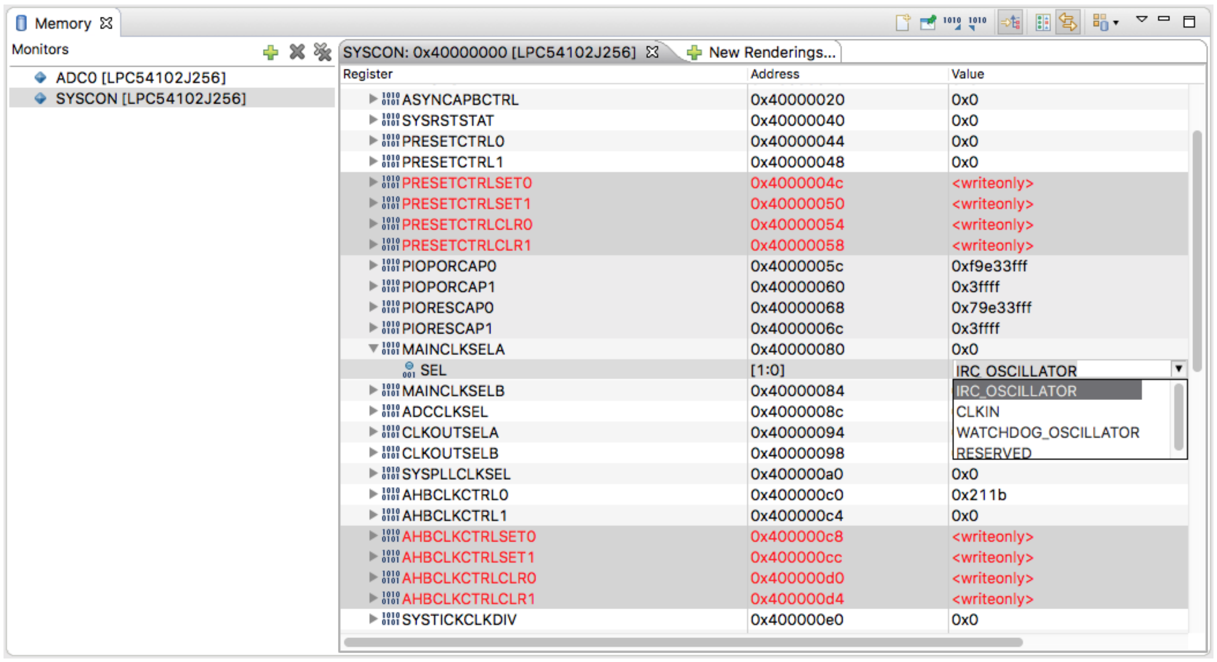
MCUXpresso IDE's debug support (whether built in or provided by an SDK) includes knowledge of an MCU's peripheral set, this is available via the Peripherals tab within the Project Explorer pane (once a debug connection is made).

Highlighted in the view below are two peripherals that have been selected for detailed display. Also highlighted are the device memory regions, if these memory regions are selected a standard hex memory display will be created. Memory regions are not peripherals in the normal sense but are included here so their memory space can be easily displayed.



	Peripheral	Address	Description
✓	ADC0	0x1c034000	12-bit ADC controller 0
☐	ADVSYSCON	0x4002c000	Advanced System configuration
☐	ASYNCSYSCON	0x40080000	Asynchronous system configuration
☐	CRC	0x1c010000	CRC engine
☐	CT32B0	0x400b4000	Standard counter/timer 0
☐	CT32B1	0x400b8000	Standard counter/timer 1
☐	CT32B2	0x40004000	Standard counter/timer 2
☐	CT32B3	0x40008000	Standard counter/timer 3
☐	CT32B4	0x4000c000	Standard counter/timer 4
☐	DCR	0xe000edf0	Debug Core Registers (v7M)
☐	DMA	0x1c004000	DMA controller
☐	GINT0	0x40010000	Group GPIO input interrupt 0
☐	GINT1	0x40014000	Group GPIO input interrupt 1
☐	GPIO	0x1c000000	General Purpose I/O
☐	I2C0	0x40094000	I2C-bus interface 0
☐	I2C1	0x40098000	I2C-bus interface 1
☐	I2C2	0x4009c000	I2C-bus interface 2
☐	INPUTMUX	0x40050000	Input multiplexing
☐	IOCON	0x4001c000	I/O pin configuration
☐	ITM	0xe0000000	Instrumentation Trace Macrocell
☐	MAILBOX	0x1c02c000	Mailbox
☐	MPU	0xe000ed90	MPU (v7M)
☐	MRT	0x40074000	Multi-Rate Timer
☐	NVIC	0xe000e000	NVIC Control/Status Regsiters (v7M)
☐	PINT	0x40018000	Pin interrupt and pattern match en...
☐	RIT	0x40070000	Repetitive Interrupt Timer
☐	RTC	0x4003c000	Real-Time Clock
☐	SCT0	0x1c018000	State Configurable Timer/PWM 0
☐	SPI0	0x400a4000	SPI0
☐	SPI1	0x400a8000	SPI1
✓	SYSCON	0x40000000	System configuration
☐	USART0	0x40084000	USART0
☐	USART1	0x40088000	USART1
☐	USART2	0x4008c000	USART2
☐	USART3	0x40090000	USART3
☐	UTICK	0x40020000	Micro-tick timer
☐	VFIFO	0x1c038000	System FIFO for Serial Peripherals
☐	WWDT	0x40038000	Windowed Watchdog Timer
☐	MFlash256	0x0	Flash: size=0x40000 (256k)
☐	Ram0_64	0x2000000	RAM: size=0x10000 (64k)
☐	Ram1_32	0x2010000	RAM: size=0x8000 (32k)
☐	Ram2_8	0x3400000	RAM: size=0x2000 (8k)

From this view each peripheral is listed along with its base address and brief description. If selected from the associated check box, a detailed memory view will be launched. This view exposes the inner peripheral registers and offers bit field enumerations to greatly simplify both reading existing configurations and setting new values.



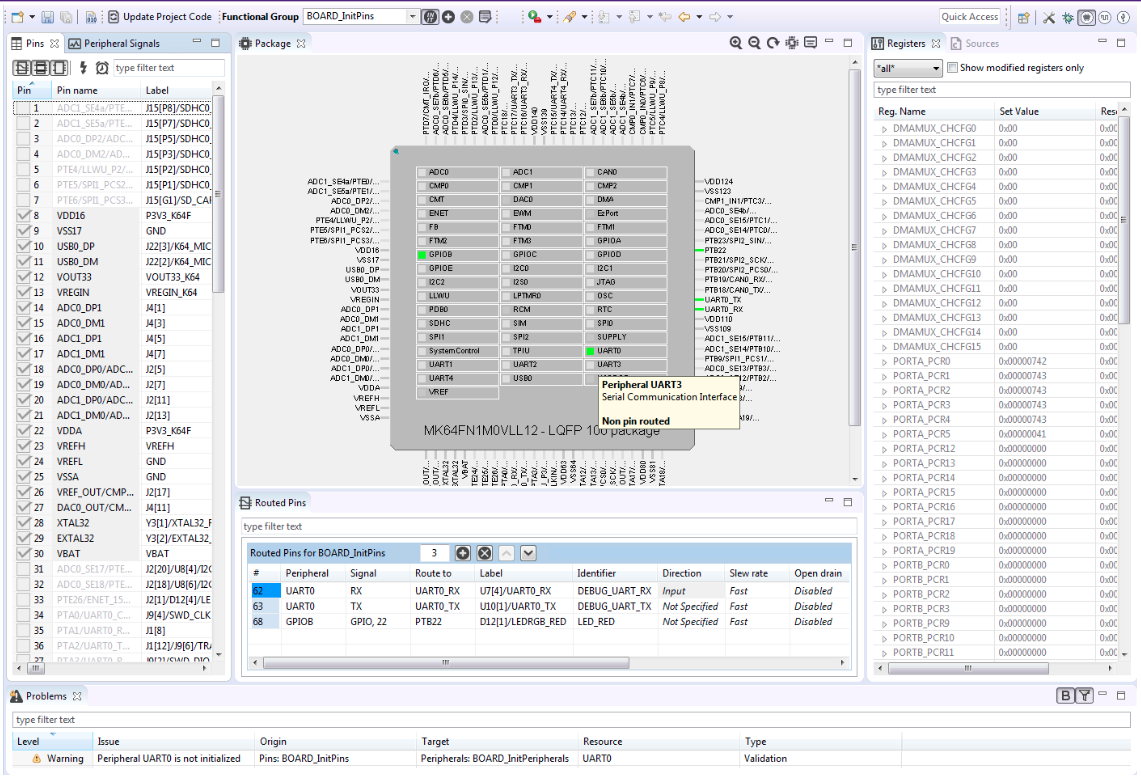
Register	Address	Value
ASYNCAPBCTRL	0x40000020	0x0
SYSRSTSTAT	0x40000040	0x0
PRESETCTRL0	0x40000044	0x0
PRESETCTRL1	0x40000048	0x0
PRESETCTRLSET0	0x4000004c	<writeonly>
PRESETCTRLSET1	0x40000050	<writeonly>
PRESETCTRLCLR0	0x40000054	<writeonly>
PRESETCTRLCLR1	0x40000058	<writeonly>
PIOPORCAP0	0x4000005c	0xf9e33fff
PIOPORCAP1	0x40000060	0x3ffff
PIORESCAP0	0x40000068	0x79e33fff
PIORESCAP1	0x4000006c	0x3ffff
MAINCLKSELA	0x40000080	0x0
SEL	[1:0]	IRC_OSCILLATOR
MAINCLKSELB	0x40000084	
ADCCLKSEL	0x4000008c	
CLKOUTSELA	0x40000094	
CLKOUTSELB	0x40000098	
SYSPLLCLKSEL	0x400000a0	0x0
AHBCLKCTRL0	0x400000c0	0x211b
AHBCLKCTRL1	0x400000c4	0x0
AHBCLKCTRLSET0	0x400000c8	<writeonly>
AHBCLKCTRLSET1	0x400000cc	<writeonly>
AHBCLKCTRLCLR0	0x400000d0	<writeonly>
AHBCLKCTRLCLR1	0x400000d4	<writeonly>
SYSTICKCLKDIV	0x400000e0	0x0

Important Note: When an MCU powers up, many peripherals will be unavailable because they are unpowered/not clocked. Attempting to access a peripheral in this state will result in failure, and the detailed peripheral view will simply display their base address in red. Certain peripherals may be partially available, unavailable sections will again display in red. Entries that have changed, will display in yellow.

Warning: It is **strongly** advised that only peripherals that are well understood are accessed in this manner since attempting to view certain peripherals can break a debug connection or perform other unexpected actions. MCUXpresso IDE's debug features cannot offer protection from such occurrances.

11. MCUXpresso Config Tools

This chapter provides an introduction to the features of the MCUXpresso Config Tools installed by default with MCUXpresso IDE. The Config Tools present new perspectives in addition to the IDE's Develop and Debug perspectives.



Pin	Pin name	Label
1	ADC1_SE4a/PTE...	J15[P8]/SDHC0_
2	ADC1_SE5a/PTE...	J15[P7]/SDHC0_
3	ADC0_DP2/ADC...	J15[P5]/SDHC0_
4	ADC0_DM2/AD...	J15[P3]/SDHC0_
5	PTE4/LLWU_P2/...	J15[P2]/SDHC0_
6	PTE5/SPI1_PCS2...	J15[P1]/SDHC0_
7	PTE6/SPI1_PCS3...	J15[G1]/SD_CAI
✓ 8	VDD16	P3V3_K64F
✓ 9	VSS17	GND
✓ 10	USB0_DP	J22[3]/K64_MIC
✓ 11	USB0_DM	J22[2]/K64_MIC
✓ 12	VOUT33	VOUT33_K64
✓ 13	VREGIN	VREGIN_K64
✓ 14	ADC0_DP1	J4[1]
✓ 15	ADC0_DM1	J4[3]
✓ 16	ADC1_DP1	J4[5]
✓ 17	ADC1_DM1	J4[7]
✓ 18	ADC0_DP0/ADC...	J2[5]
✓ 19	ADC0_DM0/AD...	J2[7]
✓ 20	ADC1_DP0/ADC...	J2[11]
✓ 21	ADC1_DM0/AD...	J2[13]
✓ 22	VDDA	P3V3_K64F
✓ 23	VREFH	VREFH
✓ 24	VREFL	GND
✓ 25	VSSA	GND
✓ 26	VREF_OUT/CMP...	J2[17]
✓ 27	DAC0_OUT/CM...	J4[11]
✓ 28	XTAL32	Y3[1]/XTAL32_F
✓ 29	EXTAL32	Y3[2]/EXTAL32_
✓ 30	VBAT	VBAT
31	ADC0_SE17/PTE...	J2[20]/U8[4]/I2C
32	ADC0_SE18/PTE...	J2[18]/U8[6]/I2C
33	PTE26/ENET_15...	J2[1]/D12[4]/LE
34	PTA0/UART0_C...	J9[4]/SWD_CLK
35	PTA1/UART0_R...	J1[8]
36	PTA2/UART0_T...	J1[12]/J9[6]/TRA

Routed Pins for BOARD_InitPins 3

#	Peripheral	Signal	Route to	Label	Identifier	Direction	Slew rate	Open drain
62	UART0	RX	UART0_RX	U7[4]/UART0_RX	DEBUG_UART_RX	*Input*	*Fast*	*Disabled*
63	UART0	TX	UART0_TX	U10[1]/UART0_TX	DEBUG_UART_TX	*Not Specified*	*Fast*	*Disabled*
68	GPIOB	GPIO, 22	PTB22	D12[1]/LEDRGB_RED	LED_RED	*Not Specified*	*Fast*	*Disabled*

all Show modified registers only

Reg. Name	Set Value	Res
DMAMUX_CHCFG0	0x00	0x0C
DMAMUX_CHCFG1	0x00	0x0C
DMAMUX_CHCFG2	0x00	0x0C
DMAMUX_CHCFG3	0x00	0x0C
DMAMUX_CHCFG4	0x00	0x0C
DMAMUX_CHCFG5	0x00	0x0C
DMAMUX_CHCFG6	0x00	0x0C
DMAMUX_CHCFG7	0x00	0x0C
DMAMUX_CHCFG8	0x00	0x0C
DMAMUX_CHCFG9	0x00	0x0C
DMAMUX_CHCFG10	0x00	0x0C
DMAMUX_CHCFG11	0x00	0x0C
DMAMUX_CHCFG12	0x00	0x0C
DMAMUX_CHCFG13	0x00	0x0C
DMAMUX_CHCFG14	0x00	0x0C
DMAMUX_CHCFG15	0x00	0x0C
PORTA_PCR0	0x00000742	0x0C
PORTA_PCR1	0x00000743	0x0C
PORTA_PCR2	0x00000743	0x0C
PORTA_PCR3	0x00000743	0x0C
PORTA_PCR4	0x00000743	0x0C
PORTA_PCR5	0x00000041	0x0C
PORTA_PCR12	0x00000000	0x0C
PORTA_PCR13	0x00000000	0x0C
PORTA_PCR14	0x00000000	0x0C
PORTA_PCR15	0x00000000	0x0C
PORTA_PCR16	0x00000000	0x0C
PORTA_PCR17	0x00000000	0x0C
PORTA_PCR18	0x00000000	0x0C
PORTA_PCR19	0x00000000	0x0C
PORTB_PCR0	0x00000000	0x0C
PORTB_PCR1	0x00000000	0x0C
PORTB_PCR2	0x00000000	0x0C
PORTB_PCR3	0x00000000	0x0C
PORTB_PCR9	0x00000000	0x0C
PORTB_PCR10	0x00000000	0x0C
PORTB_PCR11	0x00000000	0x0C

Level	Issue	Origin	Target	Resource	Type
Warning	Peripheral UART0 is not initialized	Pins: BOARD_InitPins	Peripherals: BOARD_InitPeripherals	UART0	Validation

Figure 11.1. Config Tools Showing Pins Perspective

Please refer to the MCUXpresso IDE Config Tools User Manual for detailed information.

11.1 Using the Config Tools

MCUXpresso IDE includes the following Config Tools:

- Pins Tool
  - allows you to configure pin routing and generates 'pin_mux.c & .h' source files
- Clocks Tool
  - allows you to configure system clocks and generates 'clock_config.c & .h' source files
- Peripherals Tool
  - allows you to configure other peripherals and generates 'peripherals.c & .h' source files

MCUXpresso Config Tools can be used to review or modify the configuration of SDK example projects or new projects based on SDK 2.x. To open the tool, simply right click on the project in Project Explorer and select the appropriate Open command:

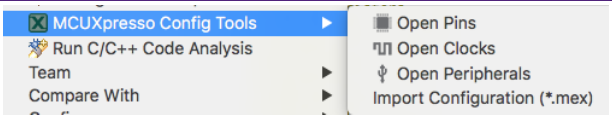
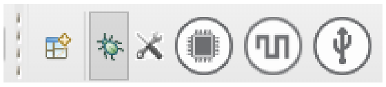


Figure 11.2. Config Tools Launch

If the project does not contains any configuration file (.mex) yet, it is automatically created by importing the existing source files (from YAML comments from pin_mux.c, clock_config.c and/or peripherals.c). If there are no source files in the project, a default configuration is created. The configuration is stored in the root of project folder with ".mex" file extension.

11.1.1 Tool Perspectives

Each tool is displayed in separate perspective. Once the configuration is opened, you can switch between perspectives to review/modify configuration of each tool – using the toolbar on the upper right part of the IDE window:



If your workspace contains multiple projects, please be aware that the MCUXpresso Config Tools only support one configuration to be opened at a time and that configuration must be opened explicitly for each project using the Open command from the popup menu. Switching perspectives does not switch the selected configuration.

11.1.2 Pins Tool

The Pins Tool allows you to display and configure the pins of the MCU. Basic configuration can be done in either of these views Pins, Peripheral Signals or Package. More advanced settings (pin electrical features) can be adjusted in Routed Pins view.

11.1.3 Clocks Tool

The Clocks Tool allows you to display and modify clock sources and outputs settings in Table view. More advanced settings can be adjusted via Diagram view and Details view. Global settings of the clocking environment such as run modes, MCG modes and SCG modes can be modified via main application toolbar.

11.1.4 Peripherals Tool

You can use the Peripherals tool to configure initialization of selected peripherals and generate code for them. In the Peripherals view, select the peripheral to configure and confirm addition of the configuration component. Then you can select the mode of the peripheral and configure the settings within the settings editor.

11.1.5 Generate Code

To update sources in the project, simply hit "Update Project Code" button on the toolbar. The command opens dialog with list of files that will be re-generated and allows to select which tools will generate the code.

Alternatively, it is also possible to export select source file by hitting export button in the Sources view.

11.1.6 SDK Components

Generated code uses the API of the SDK components to configure peripherals. SDK components missing in the IDE project are reported in problems view. It is possible to add component into IDE project by right click on the reported problem and selecting the proposed quick fix.

12. LinkServer Flash Support

MCUXpresso IDE's LinkServer based debug connections makes use of a RAM loadable Flash driver mechanism. Such a Flash driver contains the knowledge required to program the internal Flash on a particular MCU (or potentially, family of MCUs). This knowledge may be either prebuilt into the driver, or some of it may be determined by the driver as it starts up (typically known as a "generic" Flash driver).

At the time a debug connection is started from within the MCUXpresso IDE, the LinkServer debug session running on the host will typically download a Flash driver into RAM on the target MCU. It will then communicate with the downloaded Flash driver via the debug probe in order to program the required code and data into the Flash memory.

In addition, the loadable Flash driver mechanism also provides the ability to produce Flash drivers which can be used to program external Flash memory (for instance via the SPIFI Flash memory interface on LPC18x, LPC40xx, LPC43xx and LPC5460x families). The sources for some of these drivers is provided in the *Examples/Flashdrivers* subdirectory within the MCUXpresso IDE installation directory.

LinkServer Flash drivers have a .cfx file extension. For Preinstalled MCUs, the Flash driver used for each part/family will be located in the /bin/Flash subdirectory of the MCUXpresso IDE installation. For SDK installed MCUs, the Flash driver will generally be supplied within the SDK, though versions are also provided in the /bin/Flash subdirectory too.

12.1 Default vs Per-Region Flash drivers

By default, for legacy reasons, Preinstalled MCUs are configured to use what is called a "Default" Flash driver. This means that this Flash driver will be used for all Flash memory blocks that are defined for that MCU (i.e. as displayed in the Memory Configuration Editor).

For most users, there is never any need to change the automatically selected Flash driver for the MCU being programmed.

However, MCUXpresso IDE also supports the creation and programming of projects that span multiple Flash devices. In order to allow this to work, Flash drivers can also be specified per-region.

For example, this allows a project based on an LPC43xx device with internal Flash to also make use of an external SPIFI Flash device. This is achieved by removing the default Flash driver from the memory configuration and instead explicitly specifying the Flash driver to use for each Flash memory block (per-region Flash drivers). A typical use case could be to create an application to run from the MCU's internal Flash that makes use of static constant data (e.g. for graphics) stored in external SPIFI device.

Note: SDK installed MCUs are always defined using Per-Region Flash drivers.

12.2 Special case Flash drivers for LPC MCUs

For most projects, the selection of a Flash driver is automatically performed by the Project wizard, however for some MCUs some user intervention may be required.

12.2.1 LPC18xx / LPC43xx Internal Flash Drivers

A number of LPC18/43 parts provide dual banks of internal Flash, with bank A starting at address 0x1A000000, and bank B starting at address 0x1B000000.

```
* LPC18x3/LPC43x3 : Flash = 2x 256KB (512 KB total)
* LPC18x5/LPC43x5 : Flash = 2x 384KB (768 KB total)
* LPC18x7/LPC43x7 : Flash = 2x 512KB ( 1 MB total)
```

When you create a new project using the New Project Wizard for one of these parts, an appropriate default Flash driver (from *LPC18x3_43x3_2x256_BootA.cfx / LPC18x5_43x5_2x384_BootA.cfx /LPC18x7_43x7_2x512_BootA.cfx*) will be selected which after programming the part will also configure it to boot from Bank A Flash.

If you wish to boot from Bank B Flash instead, then you will need to manually configure the project to use the corresponding “BootB” Flash driver (*LPC18x3_43x3_2x256_BootB.cfx / LPC18x5_43x5_2x384_BootB.cfx / LPC18x7_43x7_2x512_BootB.cfx*). This can be done by selecting the appropriate driver file in the “Flash driver” field of the Memory Configuration Editor. **Note**: you will also need to delete Flash Bank A from the list of available memories (or at least reorder so that Flash Bank B is first).

12.2.2 SPIFI Flash Drivers

A number of LPC parts provide support for external SPIFI Flash, sometimes in addition to internal Flash. Programming these Flash memories provides a number of challenges because the size of memory (if present) is unknown, and the actual memory device is also unknown. These issues are handled using *Generic Drivers* which can interrogate the memory device to find its size and programming requirements.

At the time of writing, these LPC devices comprise:

Table 12.1. SPIFI details

LPC Part	SPIFI Address	Bootable	Flash Driver
LPC18xx/LPC43xx	0x14000000	Yes	LPC18_43_SPIFI_GENERIC.cfx
LPC40xx	0x28000000	No	LPC40xx_SPIFI_GENERIC.cfx
LPC5460x	0x10000000	No	LPC546x_SPIFI_GENERIC.cfx

During a programming operation, the Flash driver will interrogate the SPIFI Flash device to identify its configuration. If the device is recognised, its size and name will be reported in the MCUXpresso IDE Debug log - as below:

```
...
Inspected v.2 External Flash Device on SPI using SPIFI lib LPC18_43_SPIFI_GENERIC.cfx
Image 'LPC18/43 Generic SPIFI Mar  7 2017 13:14:25'
Opening flash driver LPC18_43_SPIFI_GENERIC.cfx
flash variant 'MX25L8035E' detected (1MB = 16*64K at 0x14000000)
...
```

Note: Although the Flash driver reports the size and location of the SPIFI device, the IDE’s view of the world is determined by the project memory configuration settings. It remains the users responsibility to ensure these setting match the actual device in use.

Flash devices supported by our SPIFI Flash Drivers

Below is a list of SPIFI Flash devices supported by our supplied SPIFI Flash drivers. **Note**: additional devices which identify as one of the devices below are also expected to work. However if a device is not supported by our supplied Flash Drivers, sources to generate these drivers are supplied in the *Examples/Flashdrivers* subdirectory within the MCUXpresso IDE installation directory. Users may thus add support for new SPIFI devices if needed.

```
GD25Q32C
MT25QL128AB
```

```
MT25Q512A
MT25Q256A
N25Q256
N25Q128
N25Q64
N25Q32
PM25LQ032C
MX25L1606E
MX25L1635E
MX25L3235E
MX25R6435F
MX25L6435E
MX25L12835E
MX25V8035F
MX25L8035E
S25FL016K
S25FL032P
S25FL064P
S25FL129P 64kSec
S25FL129P 256kSec
S25FL164K
S25FL256S 64kSec
S25FL256S 256kSec
S25FL512S
W25Q40CV
W25Q32FV
W25Q64FV
W25Q128FV
W25Q256FV_Untested
W25Q80BV
```

12.3 Configuring projects to span multiple Flash devices

https://community.nxp.com/thread/388979

12.4 Kinetis Flash Drivers

Kinetis MCUs make use of a range of generic drivers, which are supplied as part of the SDK part support package. When a project is created or imported, the appropriate Flash driver is automatically selected and associated with the project.

Kinetis Flash drivers generally follow a simple namimg convention i.e. **FTFx_nK_xx** where:

- FTFx is the Flash module name of the MCU, where x can take the value E, A or L
- nK represents the Flash sector size the Flash device supports, where n can take the value 1, 2, 4, 8
  - a sector size is the smallest amount of Flash that can be erased on that device
- xx is an optional suffix for special case drivers e.g. __Tiny for use on parts with a small quantity of RAM

So for example a K64F MCU's Flash driver will be called *FTFE_4K*, because the K64F MCU uses the FTFE Flash module type and support a 4KB Flash sector size.

When a debug session is started that programs data into Flash memory, the IDE's debug log file will report the Flash driver used and parameters it has read from the MCU. Below we can see the driver identified a K64 part and the size of the internal Flash available. It also reports the programming speed achieved when programming this device. These logs can be useful when problems are encountered.

Note: when the Flash driver starts up, it will interrogate the MCU and report a number of data items. However, due to the nature of internal registers with the MCU, these many not exactly match the MCU being debugged.

```
Probe Firmware: LPC-LINK2 CMSIS-DAP V5.181 (NXP Semiconductors)
Serial Number:  IWFUA1EW
VID:PID:  1FC9:0090
USB Path: USB_1fc9_0090_14131100_ff00
Probe(0): Connected&Reset. DpID: 2BA01477. CpuID: 410FC240. Info: <None>
Debug protocol: SWD. RTCK: Disabled. Vector catch: Disabled.
Inspected v.2 On chip Kinetis Flash memory module FTFE_4K.cfx
Image 'Kinetis SemiGeneric Feb 17 2017 17:24:02'
Opening flash driver FTFE_4K.cfx
flash variant 'K64 FTFE Generic 4K' detected (1MB = 256*4K at 0x0)
Closing flash driver FTFE_4K.cfx
NXP: MK64FN1M0xxx12
Connected: was_reset=true. was_stopped=true
MCUXpressoPro Full License - Unlimited
Awaiting telnet connection on port 3331 ...
GDB nonstop mode enabled
Opening flash driver FTFE_4K.cfx (already resident)
Writing 26732 bytes to address 0x00000000 in Flash
Erased/Wrote page  0-6 with 26732 bytes in 285msec
Closing flash driver FTFE_4K.cfx
Flash Write Done
Flash Program Summary: 26732 bytes in 0.28 seconds (91.60 KB/sec)
```

Flash drivers for a number of Kinetis MCUs are listed below:

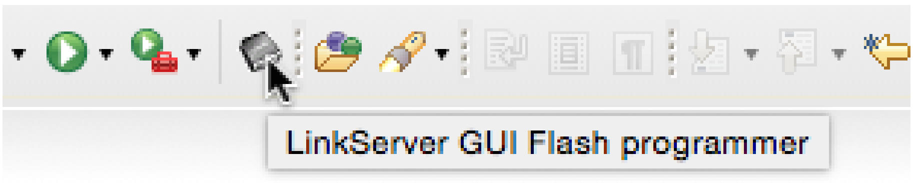
```
K64F  FTFE_4K (1MB)
K22F  FTFA_2K (512KB)
KL43  FTFA_1K (256KB)
KL27  FTFA_1K (64KB)
K40   FTFL_2K (256KB)
```

12.5 Using the LinkServer Flash programmer

As well as supporting the programming of Flash when starting a debug session, the Flash programming capabilities of LinkServer can also be accessed directly, both via the GUI and from the command line. This might be useful, for instance, in carrying out small production runs.

12.5.1 The LinkServer GUI Flash programmer

The Flash programming utility, which is invoked automatically when you launch a debug session, can also be accessed at other times within the MCUXpresso IDE environment by clicking on the “Program Flash” icon on the toolbar at the top of the IDE window....



This button provides access to 3 distinct flash programming operations:

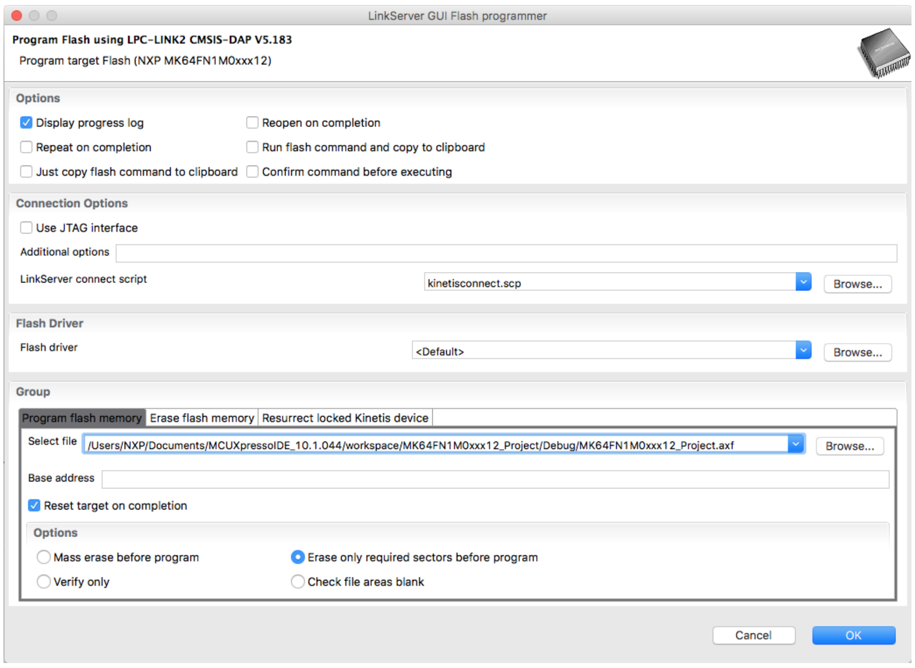
1. Programming an .axf or .bin file
2. Flash Mass Erase
3. Kinetis Flash Recovery

The behaviour of these 3 operations can be modified by a common set of self describing check boxes.

Before clicking on the “Program Flash” icon, ensure that you have a project selected in the Project Explorer pane which is configured for the MCU that you are going to program. This will ensure that appropriate configuration options for the Flash programmer are automatically set correctly. Additionally, if you intend to program a .bin file or the binary contents of a .axf file, you can directly select these from within a project, the Flash programming tool will then also pick up the appropriate file to program.

Note: Each use of the GUI Flash programmer will invoke a Debug Probe Discovery operation, the Probes discovered dialogue will only show LinkServer compatible debug probes and the option to specify the IDE Debug Mode will be hidden since it has no meaning for this operation. The GUI Flash programmer can only be used with projects with no launch configurations or LinkServer launch configurations.

Programming an .axf or .bin file


LinkServer GUI Flash programmer

Program Flash using LPC-LINK2 CMSIS-DAP V5.183
Program target Flash (NXP MK64FN1M0xxx12)

Options
- [✓] Display progress log
- [ ] Reopen on completion
- [ ] Repeat on completion
- [ ] Run flash command and copy to clipboard
- [ ] Just copy flash command to clipboard
- [ ] Confirm command before executing

Connection Options
- [ ] Use JTAG interface
- Additional options:
- LinkServer connect script: kinetisconnect.scp

Flash Driver
- Flash driver: <Default>

Group
Program flash memory | Erase flash memory | Resurrect locked Kinetis device
- Select file: /Users/NXP/Documents/MCUXpressoIDE_10.1.044/workspace/MK64FN1M0xxx12_Project/Debug/MK64FN1M0xxx12_Project.axf
- Base address:
- [✓] Reset target on completion

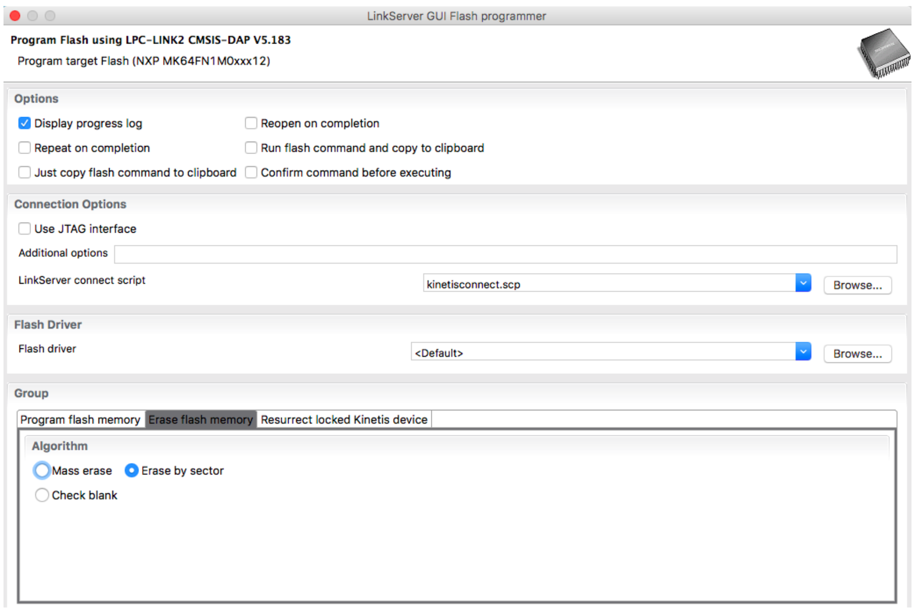
Options
- ○ Mass erase before program
- ● Erase only required sectors before program
- ○ Verify only
- ○ Check file areas blank

From this view you can select a .axf or .bin file to be programmed. Note: for a bin file you must also provide an appropriate base address. The utility will inherit the Flash driver from the projects configuration or alternatively a different Flash driver can be selected.

There are 4 options that affect the operation:

- Mass erase before program – will cause the Flash to be erased before it is programmed
- Erase only required sectors – will ensure that only the sections of the Flash to be programmed are first erase.
- Verify only – will cause the selected image to be compared with the contents of Flash
- Check file areas blank – will check to see if the area of Flash to be programmed is already erased

Flash Mass Erase


LinkServer GUI Flash programmer

Program Flash using LPC-LINK2 CMSIS-DAP V5.183
Program target Flash (NXP MK64FN1M0xxx12)

Options
[✓] Display progress log
[ ] Reopen on completion
[ ] Repeat on completion
[ ] Run flash command and copy to clipboard
[ ] Just copy flash command to clipboard
[ ] Confirm command before executing

Connection Options
[ ] Use JTAG interface
Additional options
LinkServer connect script: kinetisconnect.scp

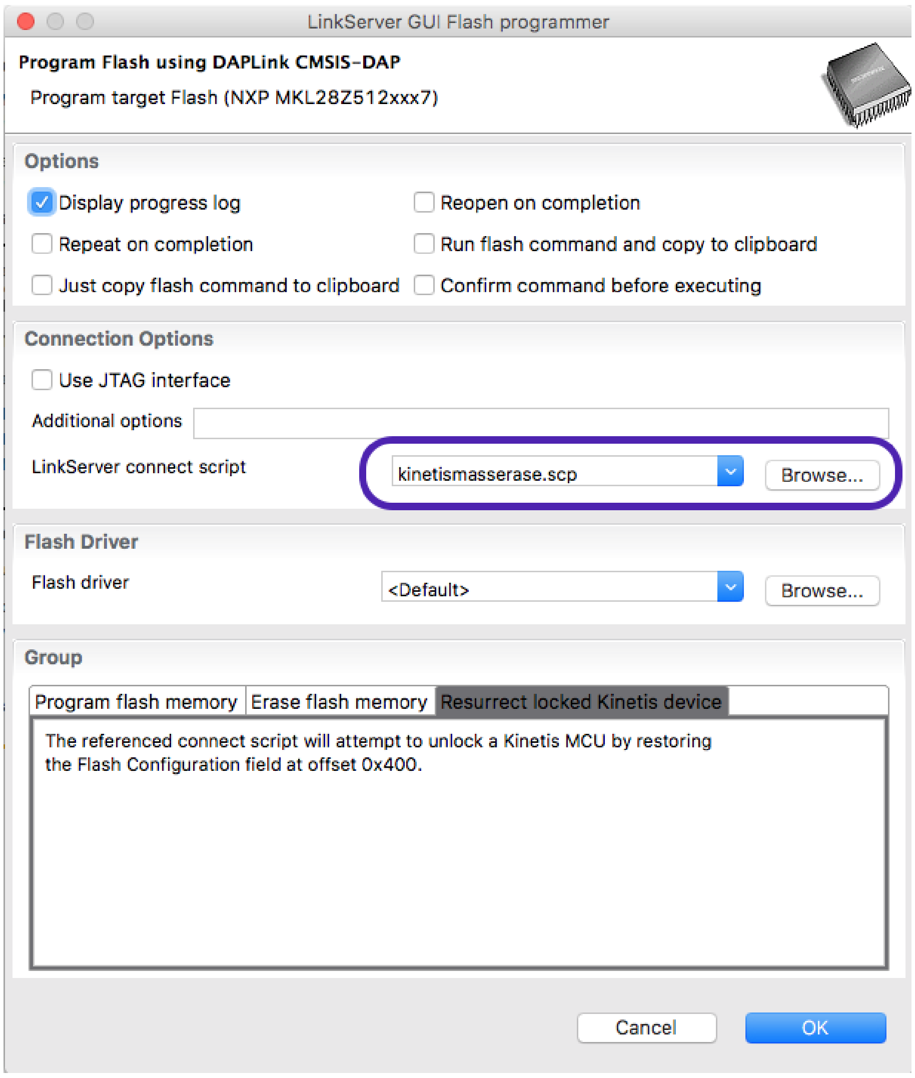
Flash Driver
Flash driver: <Default>

Group
Program flash memory | Erase flash memory | Resurrect locked Kinetis device

Algorithm
Mass erase [✓ Erase by sector]
Check blank

On occasion it can be useful to completely erase the memory on a Flash device. The utility will inherit the Flash driver from the projects configuration or alternatively a different Flash driver can be selected. The mass erase feature within the Flash driver will be used to perform the erase operation. You can also perform a non destructive operation to determine whether the Flash is blank. **Note**: Kinetis Flashes are typically not mass erased to fully blank since doing so would enable their Flash security feature.

Kinetis Flash Recovery


LinkServer GUI Flash programmer
Program Flash using DAPLink CMSIS-DAP
Program target Flash (NXP MKL28Z512xxx7)
Options
Display progress log
Reopen on completion
Repeat on completion
Run flash command and copy to clipboard
Just copy flash command to clipboard
Confirm command before executing
Connection Options
Use JTAG interface
Additional options
Flash Driver
Flash driver
Browse...
Group
Program flash memory
Erase flash memory
Resurrect locked Kinetis device
Cancel
OK

- [✓] Display progress log
- [ ] Reopen on completion
- [ ] Repeat on completion
- [ ] Run flash command and copy to clipboard
- [ ] Just copy flash command to clipboard
- [ ] Confirm command before executing
- [ ] Use JTAG interface

LinkServer connect script: kinetismasserase.scp

Flash driver: <Default>

Resurrect locked Kinetis device:

The referenced connect script will attempt to unlock a Kinetis MCU by restoring the Flash Configuration field at offset 0x400.

This operation is provided to recover a Kinetis MCUs whose Flash device has become 'secured'. A secured MCU cannot be programmed by a normal Flash programming (or debug) operation. Should this occur, simply select the 'Resurrect Kinetis device' tab, this will also automatically populate the *Connect Script* field with the *kinetismasserase.scp* script by default (although this choice may be overridden by SDK settings). Click 'OK' to run this script and attempt to recover the Flash device.

Note: Should this process fail to recover the part, an alternate script called *kinetisunlock.scp* may be successful. This alternate script must be manually selected via the *Connect Script Browse button*.

12.5.2 The Command Line Flash programmer

Flash programming is usually invoked automatically when you launch a debug session from within the MCUXpresso IDE, but can also be accessed directly using a command line utility (also known as the LinkServer debug stub). This can be useful for things like programming the Flash for devices with limited production runs.

The MCUpresso IDE Flash programming utility is located at:

```
<install_dir>/ide/bin/
```

To run a Flash programming operation from the command line, the correct Flash utility stub for your part should be called with appropriate options. For boards containing Cortex-M MCUs the utility is called *crt_emu_cm_redlink*.

For example:

```
crt_emu_cm_redlink -p LPC11U68 --flash-load "LPC11U68_App.axf"
```

will load the AXF file LPC11U68_App.axf into Flash on an LPC11U68.

Note: typically, LPC-Link2 or LPCXpresso V2 and V3 boards have debug probe firmware soft loaded automatically by the IDE when a debug operation is first performed. Therefore to use these debug probes from the command line they must either have their firmware softloaded or have probe firmware programmed into the Flash. Probe firmware can be soft-loaded from the command line by use of scripts *boot_link1* for LPC-Link and *boot_link2* for LPC-Link2, these are located at *mcuxpresso_install_dir/ide/bin*. To program debug probe firmware into the Flash memory of an LPC-Link2 debug probe, please see: http://www.nxp.com/LPCSCRYPT

Programming an image into Flash

In the simplest case the Flash programming utility takes the following options if the file to be flashed is an AXF (or ELF) file:

```
crt_emu_cm_redlink -p target --flash-load "filename" [--flash-driver "flashdriver"]
```

it is also possible to flash binary files using:

```
crt_emu_cm_redlink -p target --flash-load "filename" --load-base base_address [--flash-driver /
  "flashdriver"]
```

Where:

- crt_emu_cm_redlink is the name of the Flash utility
- target is the target chip name. For example LPC1343, LPC1114/301, LPC1768 etc. (see ‘Finding Correct Parameters...’ below)
- --flash-load can actually be one of a few different options. Use:
  - --flash-load to write the file to Flash,
  - --flash-load-exec to write it to Flash and then cause it to start running,
  - --flash-mass-load to erase the Flash and then write the file to the Flash, and
  - --flash-mass-load-exec to erase the Flash, write the file to Flash and then cause it to start running.
- filename is the file to Flash program. It may be an executable (axf) or a binary (bin) file. If using a binary file, the base_address also must be specified. Using enclosing quotes is optional unless the name includes unusual characters or spaces.
- base_address is the address where the binary file will be written. It can be specified as a hex value with a leading 0x.

If you are using Flash memory that is external to the main chip you will need to specify an appropriate Flash driver that supports the device. This usually takes the name of a .cfx file held in a default location. In unusual circumstances it is possible to specify an absolute file system name of a file. Using enclosing quotes is optional unless the name includes unusual characters or spaces (see ‘Finding Correct Parameters...’ below).

WARNING: When crt_emu_cm_redlink Flash drivers program data that they believes will form the start of an execute-in-place image they determine where the image’s vector table is and automatically inserts a checksum of the initial few vectors, as required in many LPC parts. This may not be the value held in that location by the file from which the Flash was programmed. This means that if the content of the Flash were to be compared against the file a difference at that specific location may be found.

WARNING: Flash is programmed in sectors. The sizes and distributions of Flash sectors is determined by the Flash device used. Data is programmed in separate contiguous blocks – there may be many contiguous blocks of data specified in an EFL (.AXF) file but there is only one in a binary file. When a contiguous data block is programmed into Flash data preceding the block start in its Flash sector is preserved. Data following data in the block in the final sector, however is erased.

Programming Flash with SDK Part Support

The above method works for parts supported with preinstalled part support. If SDK part support is required, then additional options must be passed to the utility.

- *sdk_parts_directory* - the place where the utility can find SDK part information; and
- *sdk_flash_directory* - the place where the utility can find Flash drivers provided by the SDK.

These are supplied to the utility by adding the following two options

```
-x "sdk_parts_directory" --flash-dir "sdk_flash_directory"
```

on to the command line already described. For example:

```
crt_emu_cm_redlink -p LPC54018 --flash-load "LPC54018_app.axf" \
  -x ~/mcuxpresso/01/.mcuxpressoide_packages_support/LPC54018_support \
  --flash-dir ~/mcuxpresso/01/.mcuxpressoide_packages_support/LPC54018_support/Flash
```

Since this is quite a lot to type you might wish to put the location of your SDK support directory into an environment variable as follows:

Windows:

```
set DIR_SDK ...\mcuxpresso\01\.mcuxpressoide_packages_support\LPC54018_support
crt_emu_cm_redlink -p LPC54018 --flash-load "LPC54018_app.axf" -x %DIR_SDK% --flash-dir %DIR_SDK%\Flash
```

MacOS or Linux:

```
export DIR_SDK="~/.mcuxpresso/01/.mcuxpressoide_packages_support/LPC54018_support"
crt_emu_cm_redlink -p LPC54018 --flash-load "LPC54018_app.axf" -x $DIR_SDK --flash-dir \
  $DIR_SDK/Flash
```

Use “Finding Correct Parameters from MCUXpresso IDE”, below, to determine what values you require for these options.

Programming Flash taking MCUXpresso IDE project Memory edits into Account

MCUXpresso IDE allows the user to modify the default definition of the memory areas (including the specification of different named Flash regions) used in a hardware using the Edit... button found in the project’s properties at C/C++Build > MCU Settings under the heading “Memory details”. The editor can create multiple named Flash regions.

In order to use these updates to the project’s part information the utility must use the directory where MCUXpresso IDE stores the project’s products for whatever configuration has been modified (typically the configuration will be called ‘Debug’) as the source of its part information.

To find the location of this directory in MCUXpresso expand the project in the Project Editor view, select the directory with the required configuration name (e.g. ‘Debug’), right click on it to bring up its properties and see the ‘Resource’ heading.

Supply this directory name as the sdk_parts_directory to the utility by adding the options:

```
-x "sdk_parts_directory"
```

Even if the part is supported by an SDK this will be the correct option to use for -x.

Programming Flash for complex debug connections

Some boards or chips occasionally need additional steps to occur before a stable debug connection can be established. Such debug connections are set up by small BASIC like programs called Connect Scripts. A good indication as to whether your chip or board normally requires a connect script can be discovered when “Finding Correct Parameters from MCUXpresso IDE” (see below).

Connect scripts are distributed within the product and do not normally need to be written from scratch.

If a connect script is required it can be supplied by adding the following option to the command line already described:

--connectscript "connectscript"

If you are using --flash-load-exec rather than --flash-load you may also find that the part that you are using requires its own “reset script” to replace the standard means of starting the execution of the flashed image. Again you may discover whether one is necessary as below. When required it can be supplied by adding the following option to the command line:

--resetscript "resetscript"

(As usual the quotes are required only if the script file name contains a space or other unprintable character.)

Finding the correct parameters from MCUXpresso IDE

Note: A simple way of finding the correct command and options is to use the GUI Flash Programmer described above, the completion dialog shows the exact command line invoked by the GUI. On this line the IDE will have chosen the correct
target name default Flash driver, flashdriver a connectscript to be run, if needed a resetscript to be run, if needed with --flash-load-exec an sdk_parts_directory where XML information about the part being used (if it is provided via an SDK) can be found an sdk_flash_directory where flash drivers supporting the part being used (if it is provided via an SDK) can be found

Note that the details will only appear and be relevant only if a project supporting the relevant chip or board is selected in the project explorer view.

For example the command line produced might be

```
crt_emu_cm_redlink  "/Workspace/frdmk64f_driver_examples_blinky.axf" -g --debug 2 --vendor NXP \
  -p MK64FN1M0xxx12 -ProbeHandle=1 -CoreIndex=0 --ConnectScript kinetisconnect.scp   -x \
  /Users/nxp/mcuxpresso/01/.mcuxpressoide_packages_support/MK64FN1M0xxx12_support  --flash-dir \
  /Users/nxp/mcuxpresso/01/.mcuxpressoide_packages_support/MK64FN1M0xxx12_support/Flash
```

Looking at this the *target name* follows *-p*; the *flashdriver* follows *--flash-driver*; a *connectscript* follows *--connectscript*; a *resetscript* follows *--resetscript*; any *sdk_flash_directory* is provided following *--flash-dir* and any *sdk_parts_directory* is provided following *-x*.

If the target does not require a connect script or reset script the relevant options will not appear. If the project is not based on an SDK -x and --flash-dir do not appear.

Dealing with Errors during Flash operations

If your board requires a connect script to be run in order to provide a stable environment for Flash drivers you may see errors when you undertake a Flash operation without using it. You can use Finding Correct Parameters from MCUXpresso IDE", above, to check whether a connect script is required.

On some boards it is possible to run an image which is incompatible with the Flash driver (which crt_emu_cm_redlink runs on the target to help it manipulate a Flash device). This incompatibility is likely to show in the form of programming errors signalled as the operation progresses. Often they are due to unmaskable exceptions (such as watchdog timers) being used by the previous image that interfere with a Flash driver's operation.

There are a number of ways to address this situation:

- Does your board support In System Processing (ISP) Reset? Using it will usually reset the hardware and stop in the Boot ROM, thus ensuring a stable environment for Flash drivers. If present it can usually be activated with one or more on-board swtiches. You may have to refer to the board's documentation.
- Use the --vc option with crt_emu_cm_redlink. This option causes a reset when the utility's connection to the board's debug port is established. Most chips will be left having executed part of the Boot ROM and usually the resulting state is suitable for running a Flash driver. (There are exceptions however.)
- Erase the contents of Flash (see below) or program a (e.g. small) image that ensures no non-maskable exceptions are involved. Naturally these solutions have the problem that they are as likely to fail (and for the same reason) as the programming operation. It is sometimes the case that an incompatible image will allow the Flash drivers to operate for a short period in which there is a chance that one of these 'solutions' can be used.

Validating the Content of Flash

The Flash programming utility can validate the content of Flash programmed as an AXF (or ELF) file:

```
crt_emu_cm_redlink -p target --flash-verify "filename" [--flash-driver "flashdriver"]
```

it is also possible to verify binary files using:

```
crt_emu_cm_redlink -p target --flash-verify "filename" --load-base base_address \
  [--flash-driver "flashdriver"]
```

Where target and Flash driver have the same meaning as above.

For example:

```
crt_emu_cm_redlink -p LPC11U68 --flash-verify "LPC11U68_App.axf"
```

Note: the issues described in 'Dealing with Errors During Flash Operation' still apply when executing this command.

Erasing the Flash

The Flash programming utility can also delete the content of Flash. To do so it takes the following options:

```
crt_emu_cm_redlink -p target --flash-mass-erase [--flash-driver "flashdriver"]
```

Where target and Flash driver have the same meaning as above.

For example:

```
crt_emu_cm_redlink -p LPC11U68 --flash-mass-erase
```

Note: the issues described in 'Dealing with Errors During Flash Operation' still apply when executing this command.)

Validating that Flash has been Erased

The Flash programming utility can validate that the content of Flash has been erased:

```
crt_emu_cm_redlink -p target --flash-check --area flash " [--flash-driver "flashdriver"]
```

For example:

```
crt_emu_cm_redlink -p LPC11U68 --flash-check --area flash
```

It is also possible to check that just the specific areas that would have been programmed by a given AXF or binary file are blank.

```
crt_emu_cm_redlink -p target --flash-check-file "filename" [--flash-driver "flashdriver"]
```

it is also possible to verify binary files using:

```
crt_emu_cm_redlink -p target --flash-check-file "filename" --load-base base_address \
  [--flash-driver "flashdriver"]
```

Where target and Flash driver have the same meaning as above.

For example:

```
crt_emu_cm_redlink -p LPC11U68 --flash-check-file "LPC11U68_App.axf"
```

Note: the issues described in 'Dealing with Errors During Flash Operation' still apply when executing this command.)

Examples

To load the binary executable file app.bin at location 0 on an LPC54113J128 target using LPC-Link2, use the following command line:

```
crt_emu_cm_redlink -p LPC54113J128 --load-base 0 --flash-load-exec app.bin
```

To load the executable file app.axf and start it executing on an LPC1768 target using LPC-Link2, use:

```
crt_emu_cm_redlink -p LPC1768 --flash-load-exec "app.axf"
```

To erase Flash, program the executable app.axf into an LPC18S37 board, which has no internal Flash but supports external Flash on the board, and then run it:

```
crt_emu_cm_redlink -p LPC18S37 --flash-mass-load-exec "app.axf" --flash-driver \
  LPC18x7_43x7_2x512_BootA.cfx
```

To erase then program app.axf into a Kinetis MK64FN1M0xxx12, which is supported through an SDK, and requirings a connect script (on MacOS/Linux):

```
crt_emu_cm_redlink -p MK64FN1M0xxx12 --flash-mass-load "app.axf" \
   --connectscript kinetisconnect.scp \
   -x ~/mcuxpresso/01/.mcuxpressoide_packages_support/MK64FN1M0xxx12_support \
   --flash-dir ~/mcuxpresso/01/.mcuxpressoide_packages_support/MK64FN1M0xxx12_support/Flash
```

To delete the Flash on an LPC1343:

```
crt_emu_cm_redlink -p LPC1343 --flash-mass-erase
```

To delete the Flash on an LPC54113J128 using vector catch to ensure that the currently booted code does not interfere with the Flash driver:

```
crt_emu_cm_redlink -p LPC54113J128 --flash-erase --vc
```

To check that the Flash is blank on an LPC54018 which is supported by an SDK and which has modified its memory layout stored in the MCUXpresso SDK example project held at ~/ws/lpcxpresso54018_driver_examples_gpio_gpio_led_output:

```
crt_emu_cm_redlink -p LPC54018 --flash-check -x \
  ~/ws/lpcxpresso54018_driver_examples_gpio_gpio_led_output/Debug \
  --flash-dir ~/mcuxpresso/01/.mcuxpressoide_packages_support/LPC54018_support/Flash
```

13. C/C++ Library Support

MCUXpresso IDE ships with three different C/C++ library families. This provides the maximum possible flexibility in balancing code size and library functionality.

13.1 Overview of Redlib, Newlib and NewlibNano

- **Redlib** Our own (non-GNU) ISO C90 standard C library, with some C99 extensions.
- **Newlib** GNU C/C++ library
- **NewlibNano** a version of the GNU C/C++ library optimized for embedded.

By default, MCUXpresso IDE will use Redlib for C projects, NewlibNano for SDK C++ projects, and Newlib for C++ projects for preinstalled MCUs.

Newlib provides complete C99 and C++ library support at the expense of a larger (in some cases, much larger) code size in your application.

NewlibNano was produced as part of ARM's "GNU Tools for ARM Embedded Processors" initiative in order to provide a version of Newlib focused on code size. Using NewlibNano can help dramatically reduce the size of your application compared to using the standard version of Newlib – for both C and C++ projects.

If you need a smaller application size and don't need the additional functionality of the C99 or C++ libraries, we recommend the use of Redlib, which can often produce much smaller applications.

13.1.1 Redlib extensions to C90

Although Redlib is basically a C90 standard C library, it does implement a number of extensions, including some from the C99 specification. These include:

- Single precision math functions
  - Single precision implementations of some of the math.h functions such as sinf() and cosf() are provided.
- stdbool.h
  - An implementation of the C99 stdbool.h header is provided.
- itoa
  - itoa() is non-standard library function which is provided in many other toolchains to convert an integer to a string. To ease porting, an implementation of this function is provided, accessible via stdlib.h. More details can be found later in this chapter.

13.1.2 Newlib vs NewlibNano

Differences between Newlib and NewlibNano include:

- NewlibNano is optimized for size.
- The printf and scanf family of routines have been re-implemented in NewlibNano to remove a direct dependency on the floating-point input/output handling code. Projects that need to handle floating-point values using these functions must now explicitly request the feature during linking.
- The printf and scanf family of routines in NewlibNano support only conversion specifiers defined in C89 standard. This provides a good balance between small memory footprint and full feature formatted input/output.
- NewlibNano removes the now redundant integer-only implementations of the printf/scanf family of routines (iprintf/iscanf, etc). These functions now alias the standard routines.
- In NewlibNano, only unwritten buffered data is flushed on exit. Open streams are not closed.

- In NewlibNano, the dynamic memory allocator has been re-implemented

13.2 Library variants

Each C library family is provided in a number of different variants : None, Nohost and Nohost-nf, Semihost and Semihost-nf (Redlib only). These variants each provide a different set of ‘stubs’ that form the very bottom of the C library and include certain low-level functions used by other functions in the library.

Each variant has a differing set of these stubs, and hence provides differing levels of functionality:

- **Semihost**
  - This library variant provides implementation of all functions, including file I/O. The file I/O will be directed through the debugger and will be performed on the host system (semihosting). For example, printf/scanf will use the debugger console window and fread/fwrite will operate on files on the host system. **Note**: emulated I/O is relatively slow and can only be used when debugging.
- **Semihost-nf (no files)**
  - Redlib only. Similar to Semhost, but only provides support for the 3 standard built-in streams – stdin, stdout, stderr. This reduces the memory overhead required for the data structures used by streams, but means that the user application cannot open and use files, though generally this is not a problem for embedded applications.
- **Nohost and Nohost-nf**
  - This library variant provides the string and memory handling functions and some file-based I/O functions. However, it assumes that you have no debugging host system, thus any file I/O will do nothing. However, it is possible for the user to provide their own implementations of some of these I/O functions, for example to redirect output to the UART.
- **None**
  - This has literally no stub and has the smallest memory footprint. It excludes low-level functions for all file-based I/O and some string and memory handling functions.

In many embedded microcontroller applications it is possible to use the None variant by careful use of the C library, for instance avoiding calls to printf().

If you are using the wrong library variant, then you will see build errors of the form:

- Linker error "Undefined reference to ‘xxx’ "

For example for a project linking against Redlib(None) but using printf() :

```
… libcr_c.a(fpprintf.o): In function `printf':
fpprintf.c:(.text.printf+0x38): undefined reference to `__sys_write'
fpprintf.c:(.text.printf+0x4c): undefined reference to `__Ciob'
… libcr_c.a(_deferredlazyseek.o): In function `__flsbuf':
_deferredlazyseek.c:(.text.__flsbuf+0x88): undefined reference to `__sys_istty'
… libcr_c.a(_writebuf.o): In function `_Cwritebuf':
_writebuf.c:(.text._Cwritebuf+0x16): undefined reference to `__sys_flen'
_writebuf.c:(.text._Cwritebuf+0x26): undefined reference to `__sys_seek'
_writebuf.c:(.text._Cwritebuf+0x3c): undefined reference to `__sys_write'
… libcr_c.a(alloc.o): In function `_Csys_alloc':
alloc.c:(.text._Csys_alloc+0xe): undefined reference to `__sys_write'
alloc.c:(.text._Csys_alloc+0x12): undefined reference to `__sys_appexit'
… libcr_c.a(fseek.o): In function `fseek':
fseek.c:(.text.fseek+0x16): undefined reference to `__sys_istty'
fseek.c:(.text.fseek+0x3a): undefined reference to `__sys_flen'
```

Or if linking against NewlibNano(None):

```
… libc_nano.a(lib_a-writer.o): In function `_write_r':
writer.c:(.text._write_r+0x10): undefined reference to `_write'
… libc_nano.a(lib_a-closer.o): In function `_close_r':
closer.c:(.text._close_r+0xc): undefined reference to `_close'
… libc_nano.a(lib_a-lseekr.o): In function `_lseek_r':
lseekr.c:(.text._lseek_r+0x10): undefined reference to `_lseek'
… libc_nano.a(lib_a-readr.o): In function `_read_r':
readr.c:(.text._read_r+0x10): undefined reference to `_read'
… libc_nano.a(lib_a-fstatr.o): In function `_fstat_r':
fstatr.c:(.text._fstat_r+0xe): undefined reference to `_fstat'
… libc_nano.a(lib_a-isattyr.o): In function `_isatty_r':
isattyr.c:(.text._isatty_r+0xc): undefined reference to `_isatty'
```

In such cases, simply change the library hosting being used (as described below), or remove the call to the triggering C library function.

13.3 Switching the selected C library

Normally the library variant used by a project is set up when the project is first created by the New Project Wizard. However it is quite simple to switch the selected C library between Redlib, Newlib and NewlibNano, as well as switching the library variant in use.

To switch, highlight the project in the Project Explorer view and go to:

Quickstart -> Quick Settings -> Set library/header type

and select the required library and variant.

13.3.1 Manually switching

Alternatively, you can make the required changes to your project properties manually as follows...

When switching between Newlib(Nano) and Redlib libraries you must also switch the headers (since the 2 libraries use different header files). To do this:

1. Select the project in Project Explorer
2. Right-click and select Properties
3. Expand C/C++ Build and select Settings
4. In the Tools settings tab, select Miscellaneous under MCU C Compiler. **Note**: Redlib is not available for C++ projects
5. In Library headers, select Newlib or Redlib
6. In the Tools setting tab, select Architecture & Headers under MCU Assembler
7. In Library headers, select Newlib or Redlib

Repeat the above sequence for all Build Configurations (typically Debug and Release).

To then change the libraries actually being linked with (assuming you are using Managed linker scripts):

1. Select the project in Project Explorer
2. Right-click and select Properties
3. Expand C/C++ Build and select Settings
4. In the Tools settings tab, select Managed Linker Script under MCU Linker
5. In the Library drop-down, select the Newlib, NewlibNano or Redlib library variant that you require (None, Nohost, Semihost, Semihost-nf).

Again repeat the above sequence for all Build Configurations (typically Debug and Release). **Note**: Redlib is not available for C++ projects.

13.4 What is Semihosting?

Semihosting is a term to describe application IO via the debug probe. For this to operate, library code and debug support are required.

13.4.1 Background to Semihosting

When creating a new embedded application, it can sometimes be useful during the early stages of development to be able to output debug status messages to indicate what is happening as your application executes.

Traditionally, this might be done by piping the messages over, a serial cable connected to a terminal program running on your PC. The MCUXpresso IDE offers an alternative to this scheme, called semihosting. Semihosting provides a mechanism for code running on the target board to use the facilities of the PC running the IDE. The most common example of this is for the strings passed to a printf being displayed in the IDE's console view.

The term "semihosting" was originally termed by ARM in the early 1990s, and basically indicates that part of the functionality is carried out by the host (the PC with the debug tools running on it), and partly by the target (your board). The original intention was to provide I/O in a target environment where no real peripheral-based I/O was available at all.

13.4.2 Semihosting implementation

The way it is actually implemented by the tools depends upon which target CPU you are running on. With Cortex-M based MCUs, the bottom level of the C library contains a special BKPT instruction. The execution of this is trapped by the debug tools which determine what operation is being requested – in the case of a printf, for example, this will effectively be a "write character to stdout". The debug tools will then read the character from the memory of the target board – and display it in the console window within the IDE.

Semihosting also provides support for a number of other I/O operations (though this relies upon your debug probe also supporting them)... For example it provides the ability for scanf to read its input from the IDE console. It also allows file operations, such that fopen can open a file on your PC's hard drive, and fscanf can then be used to read from that file.

13.4.3 Semihosting Performance

It is fair to say that the semihosting mechanism does not provide a high performance I/O system. Each time a semihosting operation takes place, the processor is basically stopped whilst the data transfer takes place. The time this takes depends somewhat on the target CPU, the debug probe being used, the PC hardware and the PC operating system. But it takes a definite period of time, which may make your code appear to run more slowly.

13.4.4 Important notes about using semihosting

When you have linked with the semihosting library, your application will no longer work standalone – it will only work when connected to the debugger.

Semihosting operations cause the CPU to drop into "debug state", which means that for the duration of the data transfer between the target and the host PC no code (including interrupts) will get executed on the target. Thus if your application uses interrupts, then it is normally advisable to avoid the use of semihosting whilst interrupts are active – and certainly within interrupt handlers themselves. If you still need to use printf, then you can retarget the bottom level of the C library to use an alternative communication channel, such as a UART or the Cortex-M CPU's ITM channel.

New in MCUXpresso IDE version 10.1.0: Is support to allow semihosted operation to continue without a debug connection. Please see the section Semihosted printf [76] for more information.

13.4.5 Semihosting Specification

The semihosting mechanism used within MCUXpresso IDE is based on the specification contained in the following document available from ARM's website... => ARM Developer Suite (ADS) v1.2 Debug Target Guide, Chapter 5. Semihosting

13.5 Use of printf

By default, the output from printf() (and puts()) will be displayed in the debugger console via the semihosting mechanism. This provides a very easy way of getting basic status information out from your application running on your target.

For printf() to work like this, you must ensure that you are linking with a "semihost" or "semihost-nf" library variant.

Note that if you only require the display of fixed strings, then using puts() rather than printf() will noticeably reduce the code size of your application.

13.5.1 Redlib printf variants

Redlib provides the following two variants of printf. Many of the MCUXpresso New project wizards provide options to select which of these to use when you create a new project.

Character vs String output

By default printf() and puts() functions will output the generated string at once, so that a single semihosted operation can output the string to the console of the debugger. Note that these versions of printf() /puts() make use of malloc() to provide a temporary buffer on the heap in order to generate the string to be displayed.

It is possible to switch to using "character-by-character" versions of these functions (which do not require heap space) by specifying the build define "CR_PRINTF_CHAR" (which should be set at the project level). This can be useful, for example, if you are retargeting printf() to write out over a UART (as detailed below)- as in this case it is pointless creating a temporary buffer to store the whole string, only to then print it out over the UART one character at a time

Integer only vs full printf (including floating point)

The printf() routine incorporated into Redlib is much smaller than that in Newlib. Thus if code size is an issue, then always try to use Redlib if possible. In addition, if your application does not pass floating point numbers to printf, you can also select a "integer only" (non-floating point compatible) variant of printf. This will reduce code size further.

To enable the "integer only" printf from Redlib, define the symbol "CR_INTEGER_PRINTF" (at the project level). This is done by default for projects created from the SDK new project wizard.

13.5.2 NewlibNano printf variants

By default, NewlibNano uses non-floating point variants of the printf and scanf family of functions, which can help to dramatically reduce the size of your image if only integer values are used by such functions.

If your codebase does require floating point variants of printf/scanf, then these can be enabled by going to:

Project -> Properties -> C/C++ Build -> Settings -> MCU Linker -> Managed Linker Script and selecting the " *Enable printf/scanf float*" tick box.

13.5.3 Newlib printf variants

Newlib provides an “iprintf” function which implements integer only printf

13.5.4 Printf when using LPCOpen

If you are building your application against LPCOpen, you may find that printf output does not get displayed in the MCUXpresso IDE’s debug console by default. This is due to many LPCOpen board library projects by default redirecting printf to a UART output.

If you want to direct printf output to the debug console instead, then you will need to modify your projects so that:

1. Your main application project is linked against the “semihost” variant of the C library, and
2. You disable the LPCOpen board library’s redirection of printf output by either:
   - locating the source file board.c within the LPCOpen board library and comment out the line: #include "retarget.h, or
   - locating the file board.h and enable the line: #define DEBUG_SEMIHOSTING

13.5.5 Printf when using SDK

The MCUXpresso SDK codebase provides its own printf style functionality through the macro PRINTF. This is set up in the header file fsl_debug_console.h such that it can either point to the printf function provided by the C library itself, or can be directly to the SDK function pseudo-printf function : DbgConsole_Printf() . This will typically cause the output to be sent out via a UART (which may be connected to an on-board debug probe which will sent it back to the host over a USB VCOM channel). This is controlled by the macro **SDK_DEBUGCONSOLE** thus:

- If SDK_DEBUGCONSOLE == 0
  - PRINTF is directed to C library printf()
- If SDK_DEBUGCONSOLE == 1
  - PRINTF is directed to SDK DbgConsole_Printf()

The Advanced page of the SDK new project wizard and Import SDK examples wizard offer the option to configure a project so that PRINTF is directed to C library printf() by setting **SDK_DEBUGCONSOLE** appropriately.

In addition, if PRINTF is being directed to the C library printf(), then if **SDK_DEBUGCONSOLE_UART** is also defined, then printf output will still be directed to the UART. Again the Advanced page of the SDK new project wizard and Import SDK examples wizard offer an option to control this.

13.5.6 Retargeting printf/scanf

By default, the printf function outputs text to the debug console using the “semihosting” mechanism.

In some circumstances, this output mechanism may not be suitable for your application. Instead, you may want printf to output via an alternative communication channel such as a UART or – on Cortex-M3/M4 – the ITM channel of SWO Trace. In such cases you can retarget the appropriate portion of the bottom level of the library.

The section “How to use ITM Printf” below provides an example of how this can be done.

Note: when retargeting these functions, you can typically link against the “nohost” variant of the C Library, rather than the “semihost” one.

Redlib

To retarget Redlib’s printf(), you need to provide your own implementations of the function __sys_write():

```
int __sys_write(int iFileHandle, char *pcBuffer, int iLength)
```

Function returns number of unwritten bytes if error, otherwise 0 for success

Similarly if you want to retarget scanf(), you need to provide your own implementations of the function __sys_readc():

```
int __sys_readc(void)
```

Function returns character read

Note: these two functions effectively map directly onto the underlying “semihosting” operations.

Newlib / NewlibNano

To retarget printf(), you will need to provide your own implementation of the Newlib system function _write():

```
int _write(int iFileHandle, char *pcBuffer, int iLength)
```

Function returns number of unwritten bytes if error, otherwise 0 for success

To retarget scanf, you will need to provide your own implementation of the Newlib system function _read():

```
int _read(int iFileHandle, char *pcBuffer, int iLength)
```

Function returns number of characters read, stored in pcBuffer

More information on the Newlib system calls can be found at: https://sourceware.org/newlib/libc.html#Syscalls

13.5.7 How to use ITM Printf

ITM Printf is a scheme to achieve application IO via a debug probe without the usual semihosting penalties.

ITM Overview

As part of the Cortex-M3/M4 SWO Trace functionality available when using an LPC-Link2 (with NXP’s CMSIS-DAP firmware), MCUXpresso IDE provides the ability to make use of the ITM : The Instrumentation Trace Macrocell (ITM) block provides a mechanism for sending data from your target to the debugger via the SWO trade stream. This communication is achieved through a memory-mapped register interface. Data written to any of 32 stimulus registers is forwarded to the SWO stream. Unlike other SWO functionality, using the ITM stimulus ports requires changes to your code and so should not be considered non-intrusive.

Printf operations can be carried out directly by writing to the ITM stimulus port. However the stimulus port is output only. And therefore scanf functionality is achieved via a special global variable, which allows the debugger to send characters from the console to the target (using the trace interface). The debugger writes data to the global variable named ITM_RxBuffer to be picked up by scanf.

Note: MCUXpresso IDE currently only supports ITM via stimulus port 0.

Note: For more information on SWO Trace, please see the MCUXpresso IDE LinkServer SWO Trace Guide.

ITM printf with SDK

The Advanced page of the SDK new project wizard and Import SDK examples wizard offer the option to configure a project so as to redirect printf/scanf to ITM. Selecting this option will cause the file retarget_itm.c to be generated in your project to carry out the redirection.

ITM printf with LPCOpen

To use this functionality with an LPCOpen project you need to: Include the file retarget_itm.c in your project – available from the Examples subdirectory of your IDE installation Ensure you are using a semihost, semihost-nf, or nohost C library variant. Then simply add calls to printf and scanf to your code.

If you just linking against the LPCOpen Chip library, then this is all you need to do. However if you are also linking against an LPCOpen board library then you will likely see build errors of the form:

```
../src/retarget.h:224: multiple definition of `__sys_write'
../src/retarget.h:240: multiple definition of `__sys_readc'
```

locating the file board.h and enable the line: *#define DEBUG_SEMIHOSTING*, or locating the source file board.c within the LPCOpen board library and comment out the line: *#include "retarget.h*

13.6 itoa() and uitoa()

itoa() is non-standard library function which is provided in many other toolchains to convert an integer to a string.

13.6.1 Redlib

To ease porting, the MCUXpresso IDE provides two variants of this function in the Redlib C library....

```
char * itoa(int value, char *vstring, unsigned int base);
char * uitoa(unsigned int value, char *vstring, unsigned int base);
```

which can be accessed via the system header....

```
#include <stdlib.h>
```

itoa() converts an integer value to a null-terminated string using the specified base and stores the result in the array pointed to by the vstring parameter. Base can take any value between 2 and 16; where 2 = binary, 8 = octal, 10 = decimal and 16 = hexadecimal.

If base is 10 and the value is negative, then the resulting string is preceded with a minus sign (-). With any other base, value is always considered unsigned. The return value to the function is a pointer to the resulting null-terminated string, the same as parameter vstring.

uitoa() is similar but treats the input value as unsigned in all cases.

Note: the caller is responsible for reserving space for the output character array – the recommended length is 33, which is long enough to contain any possible value regardless of the base used.

Example invocations

```
char vstring [33];
itoa (value,vstring,10); // convert to decimal
itoa (value,vstring,16); // convert to hexadecimal
itoa (value,vstring,8);; // convert to octal
```

Standards compliance

As noted above, itoa() / uitoa() are not standard C library functions. A standard-compliant alternative for some cases may be to use sprintf() - though this is likely to cause an increase in the size of your application image:

```
sprintf(vstring,"%d",value); // convert to decimal
sprintf(vstring,"%x",value); // convert to hexadecimal
sprintf(vstring,"%o",value); // convert to octal
```

13.6.2 Newlib/NewlibNano

Newlib and NewlibNano now also provide similar functionality though with slightly different naming **- itoa()** and **utoa()**.

13.7 Libraries and linker scripts

When using the managed linker script mechanism, as described in the chapter “Memory configuration and Linker Script Generation”, then the appropriate settings to link against the required library family and variant will be handled automatically.

However if you are not using the managed linker script mechanism, then you will need to define which library files to use in your linker script. To do this, add one of the following entries before the SECTION line in your linker script:

- Redlib (None), add
  - [C project only]: GROUP (libcr_c.a libcr_eabihelpers.a)
- Redlib (Nohost), add
  - [C projects only]: GROUP (libcr_nohost.a libcr_c.a libcr_eabihelpers.a)
- Redlib (Semihost-nf), add
  - [C projects only]: GROUP (libcr_semihost_nf.a libcr_c.a libcr_eabihelpers.a)
- Redlib (Semihost), add
  - [C projects only]: GROUP (libcr_semihost.a libcr_c.a libcr_eabihelpers.a)

- NewlibNano (None), add
  - [C projects]: GROUP (libgcc.a libc_nano.a libm.a libcr_newlib_none.a)
  - [C++ projects]: GROUP (libgcc.a libc_nano.a libstdc++_nano.a libm.a libcr_newlib_none.a)
- NewlibNano (Nohost), add
  - [C projects]: GROUP (libgcc.a libc_nano.a libm.a libcr_newlib_nohost.a)
  - [C++ projects]: GROUP (libgcc.a libc_nano.a libstdc++_nano.a libm.a libcr_newlib_nohost.a)
- NewlibNano (Semihost), add

  - [C projects]: GROUP (libgcc.a libc_nano.a libm.a libcr_newlib_semihost.a)
  - [C++ projects]: GROUP (libgcc.a libc_nano.a libstdc++_nano.a libm.a libcr_newlib_semihost.a)

- Newlib (None), add
  - [C projects]: GROUP (libgcc.a libc.a libm.a libcr_newlib_none.a)
  - [C++ projects]: GROUP (libgcc.a libc.a libstdc++.a libm.a libcr_newlib_none.a)
- Newlib (Nohost), add
  - [C projects]: GROUP (libgcc.a libc.a libm.a libcr_newlib_nohost.a)
  - [C++ projects]: GROUP (libgcc.a libc.a libstdc++.a libm.a libcr_newlib_nohost.a)
- Newlib (Semihost), add
  - [C projects]: GROUP (libgcc.a libc.a libm.a libcr_newlib_semihost.a)
  - [C++ projects]: GROUP (libgcc.a libc.a libstdc++.a libm.a libcr_newlib_semihost.a)

In addition, if using NewlibNano, then tick box method of enabling printf/scanf floating point support in the Linker pages of Project Properties will also not be available. In such cases, you can enabling floating point support manually by going to:

Project Properties -> C/C++ Build -> Settings -> MCU Linker -> Miscellaneous

and entering *-u _printf_float* and/or *-u _scanf_float* into the “Linker flags” box.

A further alternative is to put an explicit reference to the required support function into your project codebase itself. One way to do this is to add a statement such as:

asm (“.global _printf_float”);

to one (or more) of the C source files in your project.

14. Memory Configuration and Linker Scripts

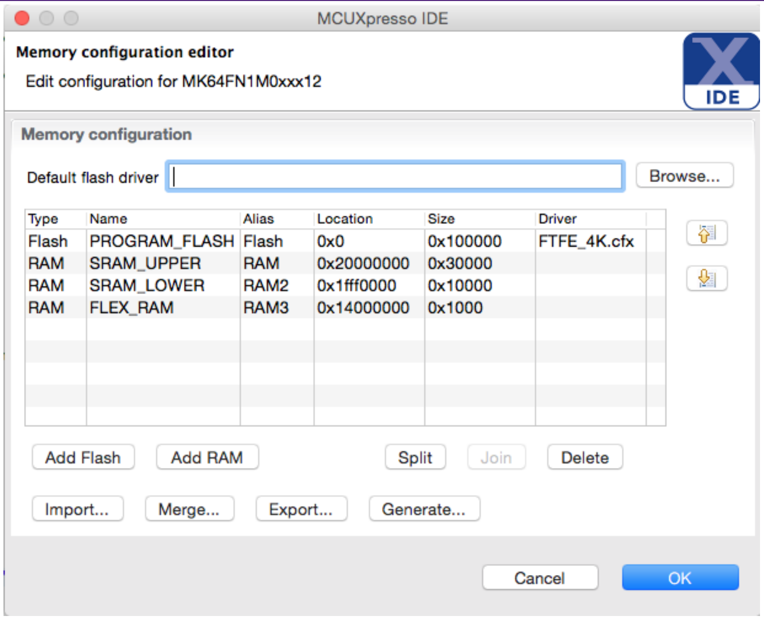
14.1 Introduction

A key part of the core technology within MCUXpresso IDE is the principle of a default defined memory map for each MCU. For devices with internal Flash, this will also specify a Flash driver to be used to program that Flash memory (for use with LinkServer “native” debug probes).

For preinstalled MCUs, the definition of the memory map is contained within the MCU part knowledge that is built into the product. For MCUs installed into MCUXpresso IDE from an SDK, the definition of the memory map is loaded from manifest file within the SDK structure.

But in both cases, the defined memory map is used by the MCUXpresso IDE to drive the “managed linker script” mechanism. This auto-generates a linker script to place the code and data from your project appropriately in memory, as well as being made available to the debugger.

A projects memory map can be viewed and modified by the user to add, remove (split/join) or reorder blocks using the Memory Configuration Editor. For example, if a project targets an MCU that supports external Flash (e.g. SPIFI), then its memory map can be easily extended to define the SPIFI memory region (base and size). In addition, an appropriate Flash driver can be associated with the newly defined region.



Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x100000	FTFE_4K.cfx
RAM	SRAM_UPPER	RAM	0x20000000	0x30000	
RAM	SRAM_LOWER	RAM2	0x1fff0000	0x10000	
RAM	FLEX_RAM	RAM3	0x14000000	0x1000	

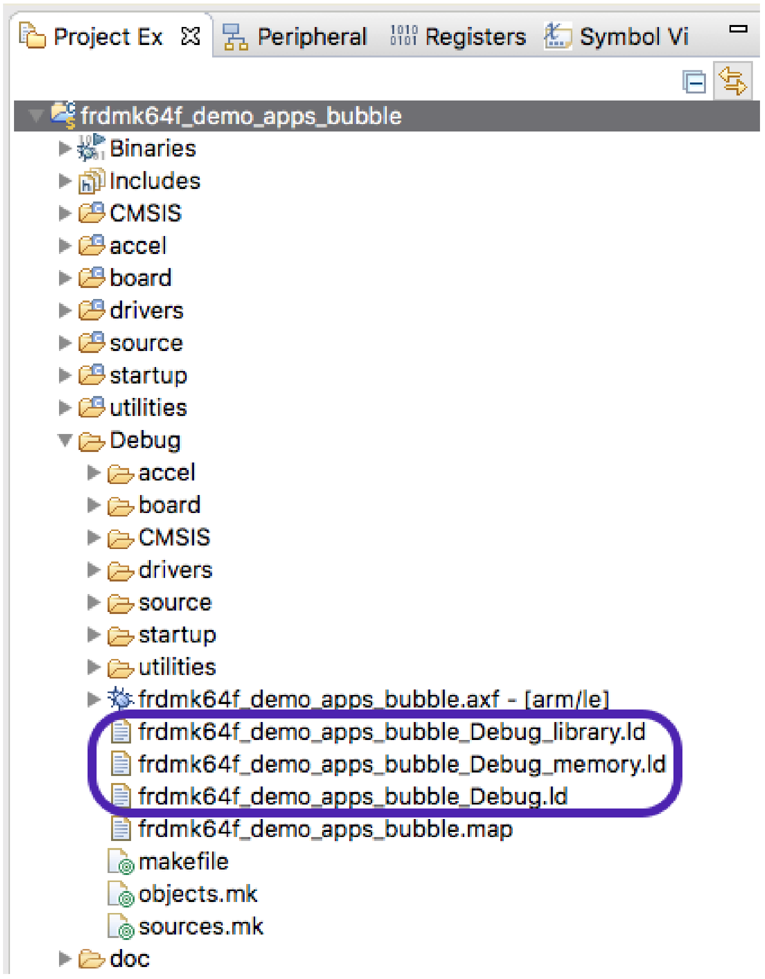
Figure 14.1. Memory Configuration

14.2 Managed Linker Script Overview

By default, the use of “managed linker scripts” is enabled for projects. This mechanism allows the MCUXpresso IDE to automatically create a script for each build configuration that is suitable for the MCU selected for the project, and the C libraries being used. It will create (and at times modify) three linker script files for each build configuration of your project:

```
<projname>_<buildconfig>_lib.ld
<projname>_<buildconfig>_mem.ld
<projname>_<buildconfig>.ld
```

This set of hierarchical files are used to define the C libraries being used, the memory map of the system and the way your code and data is placed into the memory map. These files will be located in the build configuration subdirectories of your project (typically – Debug and Release).



```
frdmk64f_demo_apps_bubble
  Binaries
  Includes
  CMSIS
  accel
  board
  drivers
  source
  startup
  utilities
  Debug
    accel
    board
    CMSIS
    drivers
    source
    startup
    utilities
    frdmk64f_demo_apps_bubble.axf - [arm/le]
    frdmk64f_demo_apps_bubble_Debug_library.ld
    frdmk64f_demo_apps_bubble_Debug_memory.ld
    frdmk64f_demo_apps_bubble_Debug.ld
    frdmk64f_demo_apps_bubble.map
    makefile
    objects.mk
    sources.mk
  doc
```

Figure 14.2. Project Explorer Debug folder Linker Scripts

The managed linkers script mechanism also automatically takes into account memory map changes made in Memory Configuration Editor as well as other configuration changes, such as C/C++ library setting.

14.3 How are managed linker scripts generated?

The MCUXpresso IDE passes a set of parameters into the linker script generator (based on the “Freemarker” scripting engine) to create an appropriate linker script for your project. This generator uses a set of conditionally parsed template files, each of which control different aspects of the generated linker script.

It is possible to modify certain aspects of the generated linker script by providing one or more modified template files locally within \linkscripts subdirectory of project directory structure. Any such templates that you provide locally will then override the default ones built into MCUXpresso. A full set of the default linker templates (.ldt) files are provided inside *\Wizards\linker* subdirectory of your IDE install.

14.4 Default image layout

Code and initial values of initialised data items are placed into first bank of Flash (as show in memory configuration editor). During startup, the MCUXpresso IDE startup code copies the data into the first bank of RAM (as show in memory configuration editor), and zero initializes the BSS data directly after this in memory. This process uses a global section table generated into the image from the linker script.

Other RAM blocks can also have data items placed into them under user control, and the startup code will also initialize these automatically. See later in this chapter for more details.

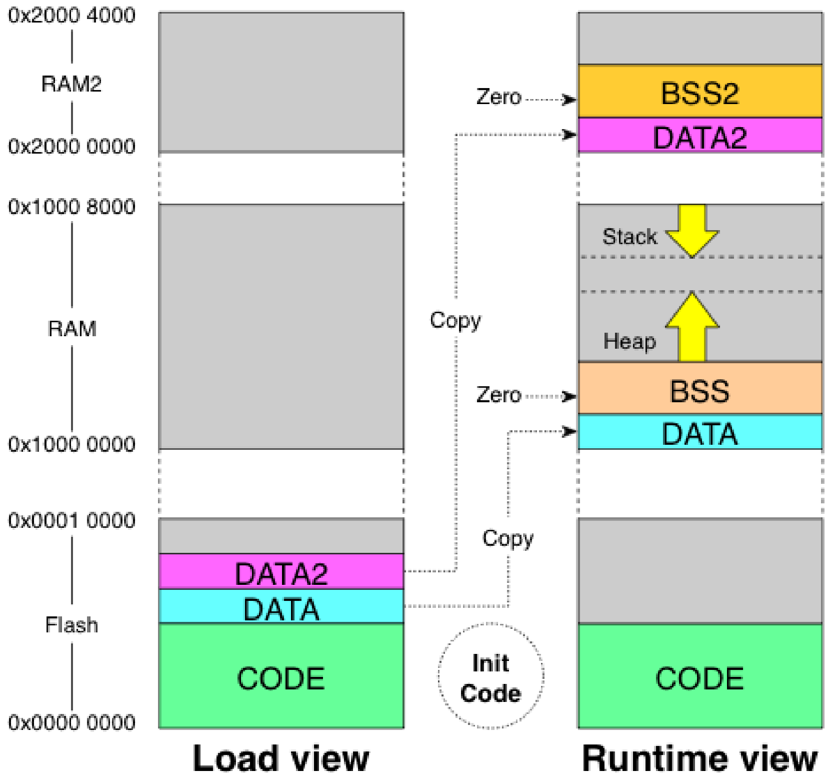


Figure 14.3. Default Memory Layout

Note: The above memory layout is simply the default used by the IDE's managed linker script mechanism. There are a number of mechanisms that can be used to modify the layout according to the requirements of your actual project – such as simply editing the order of the RAM banks in the Memory Configuration Editor. These various methods are described later in this chapter.

The default memory layout will also locate the heap and stack in the first RAM bank, such that:

- the heap is located directly after the BSS data, growing upwards through memory
- the stack located at the end of the first RAM bank, growing down towards the heap

Again this heap and stack placement is a default and it is very easy to modify the locations for a particular project, as will be described later in this chapter.

Note: When you import a project, you may find that the defaults have already been modified. Check the Project Properties to confirm the exact details.

14.5 Examining the layout of the generated image

Looking at the size of the AXF file generated by building your project on disk does not provide any information as to how much Flash/RAM space your application will occupy when downloaded to your MCU. The AXF file contains a lot more information than just the binary code of your application, for example the debug data used to provide source level information when debugging, that is never downloaded to your MCU.

Looking at the size of the AXF file generated by building your project on disk does not provide any information as to how much Flash/RAM space your application will occupy when downloaded to your MCU. The AXF file contains a lot more information than just the binary code of your application, for example the debug data used to provide source level information when debugging, that is never downloaded to your MCU.

14.5.1 Linker --print-memory-usage

MCUXpresso IDE projects use the --print-memory-usage option on the link step of a build to display memory usage information in the build console of the following form:

```
Memory region      Used Size   Region Size     %age Used
PROGRAM_FLASH:        26764 B         1 MB         2.55%
SRAM_UPPER:            8532 B       192 KB         4.34%
SRAM_LOWER:              0 GB        64 KB         0.00%
FLEX_RAM:                0 GB         4 KB         0.00%
Finished building target: frdmk64f_demo_apps_bubble.axf
```

The memory regions displayed here will match up to the memory banks displayed in the memory configuration editor when the managed linker script mechanism is being used.

By default, the application will build and link against the first Flash memory found within the devices memory configuration. For most MCUs there will only be one Flash device available. In this case our project requires 26764 bytes of Flash memory storage, 2.55% of the available Flash storage.

RAM will be used for global variable, the heap and the stack. MCUXpresso IDE provides a flexible scheme to reserve memory for Stack and Heap. This build has reserved 4KB each for the stack and the heap contributing 8KB to the overall 8532 bytes reported.

If using the LPCXpresso style of heap and stack placement (described later in this chapter), the RAM consumption provided by this is only that of your global data. It will not include any memory consumed by your stack and heap when your application is actually executing.

Note: project imported into MCUXpresso IDE may not have been created with this option. To add this, right click on the project and select *C/C++ Build ->Settings -> MCU Linker -> Miscellaneous then click '+' and add --print-memory-usage*

14.5.2 arm-none-eabi-size

In addition, a post-build step will normally invoke the arm-none-eabi-size utility to provide this information in a slightly different form....

```
  text    data    bss    dec     hex   filename
  2624     524     32    3180     c6c   LPCXpresso1768_systick_twinkle.axf
```

- **text -** shows the code and read-only data in your application (in decimal)
- **data** - shows the read-write data in your application (in decimal)
- **bss -** show the zero initialized ('bss' and 'common') data in your application (in decimal)

- **dec** - total of ‘text’ + ‘data’ + ‘bss’ (in decimal)
- **hex** - hexadecimal equivalent of 'dec'

Typically:

- the Flash consumption of your application will then be text + data
- the RAM consumption of your application will then be data + bss

Again if using the LPCXpresso style of heap and stack placement (described later in this chapter), the RAM consumption will not include any memory consumed by your stack and heap when your application is actually executing.

You can also manually run the arm-none-eabi-size utility on both your final application image, or on individual object files within your build directory by right clicking on the file in Project Explorer and selecting the Binary Utilities -> Size option.

14.5.3 Linker Map files

The linker option “-map” option, which is enabled by default by the project wizard when a new project is created, allows you to analyse in more detail the contents of your application image. When you do a build, this will cause a file called *projectname.map* to be created in the Debug (or Release) subdirectory, which can be loaded into the editor view. This contains a large amount of information, including:

- A list of archive members (library objects) included with details
- A list of discarded input sections (because they are unused and the linker option --gc-sections is enabled).
- The location, size and type of all code, data and bss items that have been placed in the image

14.5.4 Symbol Viewer

The Symbol Viewer provides a simple way of displaying the symbols in an object, library archive or executable. By default, this is located in the top left of the MCUXpresso IDE window, in parallel with the Project Explorer view.

Viewing Symbols in the Viewer

To open an image in the Symbol Viewer, either highlight it in the Project Explorer Views and use the context sensitive menu ‘Tools->View Symbols’ menu, or use the Browse button on the Toolbar within the Symbol Viewer windows itself

The Symbol Viewer can display object files (.o), libraries (.lib) and executables (.axf or .elf)

The image will be processed and displayed in the Symbol Viewer as shown in the next section.

It is possible to open multiple Symbol Viewers by pressing the ‘Green +’ icon in the toolbar. The symbols for different images can then be displayed simultaneously.

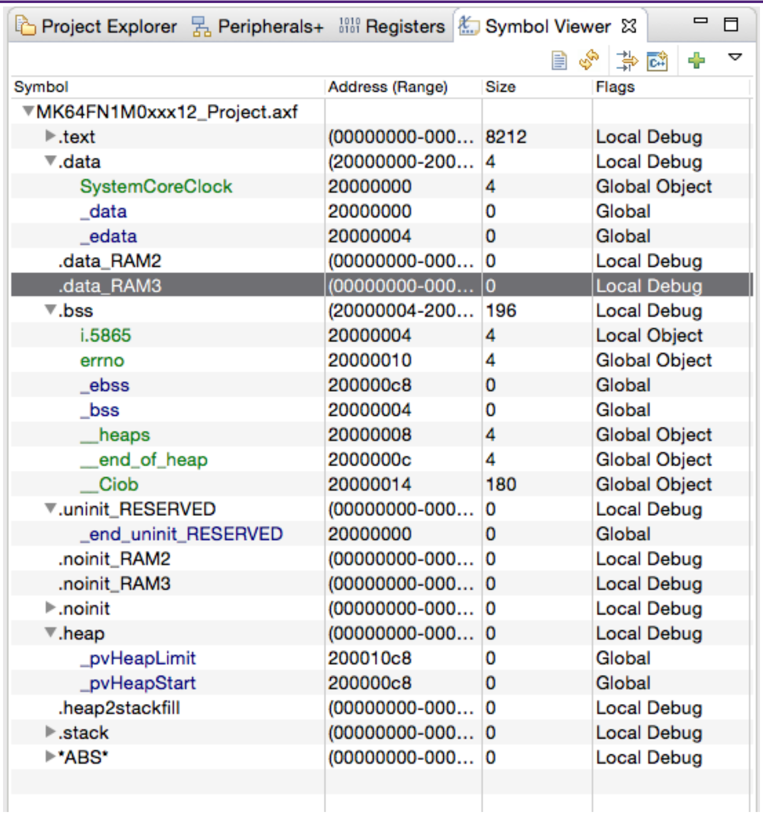
Using the Symbol Viewer

When first opening a file, the viewer will display the sections found in the file (e.g. .text, .bss etc). Expanding a section will show the symbols within that section. Clicking on the symbol name will open the source file in an editor window at the symbol definition (if source is available).

The columns of the symbol viewer show information about the symbols:

- **Symbol Name:**
- **Address:** The address (or value) of the Symbol
- **Size:** The size of the symbol, in bytes. For functions this would be the size of the function. For variables, this would be the size occupied by the variable

- **Flags**: The type of the Symbol. Typically this would be Local or Global and Function or Object (data variable)



Symbol	Address (Range)	Size	Flags
▼MK64FN1M0xxx12_Project.axf			
▶.text	(00000000-000…	8212	Local Debug
▼.data	(20000000-200…	4	Local Debug
SystemCoreClock	20000000	4	Global Object
_data	20000000	0	Global
_edata	20000004	0	Global
.data_RAM2	(00000000-000…	0	Local Debug
.data_RAM3	(00000000-000…	0	Local Debug
▼.bss	(20000004-200…	196	Local Debug
i.5865	20000004	4	Local Object
errno	20000010	4	Global Object
_ebss	200000c8	0	Global
_bss	20000004	0	Global
__heaps	20000008	4	Global Object
__end_of_heap	2000000c	4	Global Object
__Ciob	20000014	180	Global Object
▼.uninit_RESERVED	(00000000-000…	0	Local Debug
_end_uninit_RESERVED	20000000	0	Global
.noinit_RAM2	(00000000-000…	0	Local Debug
.noinit_RAM3	(00000000-000…	0	Local Debug
▶.noinit	(00000000-000…	0	Local Debug
▼.heap	(00000000-000…	0	Local Debug
_pvHeapLimit	200010c8	0	Global
_pvHeapStart	200000c8	0	Global
.heap2stackfill	(00000000-000…	0	Local Debug
▶.stack	(00000000-000…	0	Local Debug
▶*ABS*	(00000000-000…	0	Local Debug

Figure 14.4. Symbol Viewer

Note: The symbols displayed are a snapshot of the symbols for a particular build, therefore these should be refreshed when a new build is performed. This can easily be done using the Reload icon in the Symbol Viewer window.

Other utilities

The arm-none-eabi-nm utility is effectively a command line version of the Symbol Browser. But it can sometime be useful when looking at the size of your application, as it can produce some of the information provided in the linker map file but in a more concise form. For example:

```
arm-none-eabi-nm -S --size-sort -s project.axf
```

produces a list of all the symbols in an image, their sizes and their addresses, listed in size order. For more information on this utility, please see the GNU binutils documentation.

Note: you can run arm-none-eabi-nm as a post-build step, or else open a command shell using the status bar shortcuts (at the bottom of the IDE window).

14.6 Other options affecting the generated image

14.6.1 LPC MCUs – Code Read Protection

Most of NXP's LPC Cortex-M based MCUs which have internal Flash memory contain "Code Read Protection" (CRP) support. This mechanism uses one of a number of known values being placed in a specific location in Flash memory to provide a number of levels of protection. When the MCU boots, this specific location in Flash memory is read and depending upon its value, the

MCU may prevent access to the Flash memory by external devices. This location is typically at 0x2FC though for LPC18xx/43xx parts with internal Flash, the CRP location is at an offset of 0x2FC from the start of the Flash bank being used.

CRP : Preinstalled MCUs

Support for setting up the CRP memory location is provided via a combination of the Project Wizard, a header file and a number of macros. This support allows specific values to be easily placed into the CRP memory location, based on the user's requirements.

The New Project wizard contains an option to allow linker support for placing a CRP word to be enabled when you create a new project. This is typically enabled by default. This wizard option actually then controls the "Enable CRP" checkbox of the Project Properties linker Target tab.

In addition, the wizard will create a file, 'crp.c' which defines the 'CRP_WORD' variable which will contain the required CRP value. A set of possible values are provided by the NXP/crp.h header file that this then includes. Thus for example 'crp.c' will typically contain:

```
#include <NXP/crp.h>
__CRP const unsigned int CRP_WORD = CRP_NO_CRP ;
```

which is then placed at the correct location in Flash by the linker script generated by the managed linker script mechanism:

```
. = 0x000002FC ;
KEEP(*(.crp))
```

Note: the value CRP_NO_CRP ensures that the Flash memory is fully accessible. When you reach the stage of your project where you want to protect your image, you will need to modify the CRP word to contain an appropriate value.

Important Note: You should take particular care when modifying the value placed in the CRP word, as some CRP settings can disable some or all means of access to your MCU (including debug). Before making use of CRP, you are strongly advised to refer to the User Manual for the LPC MCU that you are using.

CRP : MCUs installed by Importing an SDK

The support for CRP in LPC parts imported into MCUXpresso IDE from an SDK, is generally similar to the Preinstalled MCUs. However rather than having a separate crp.c file, the CRP_WORD variable definition is generally found within the startup code.

14.6.2 Kinetis MCUs – Flash Config blocks

Kinetis MCUs provides an alternative means of protecting the user's image in Flash using the Flash Configuration Block. The Flash Configuration Field is generally located at addresses 0x400-0x40F and unlike the LPC CRP mechanism only specific values give access, whereas any other values are likely to lock the part.

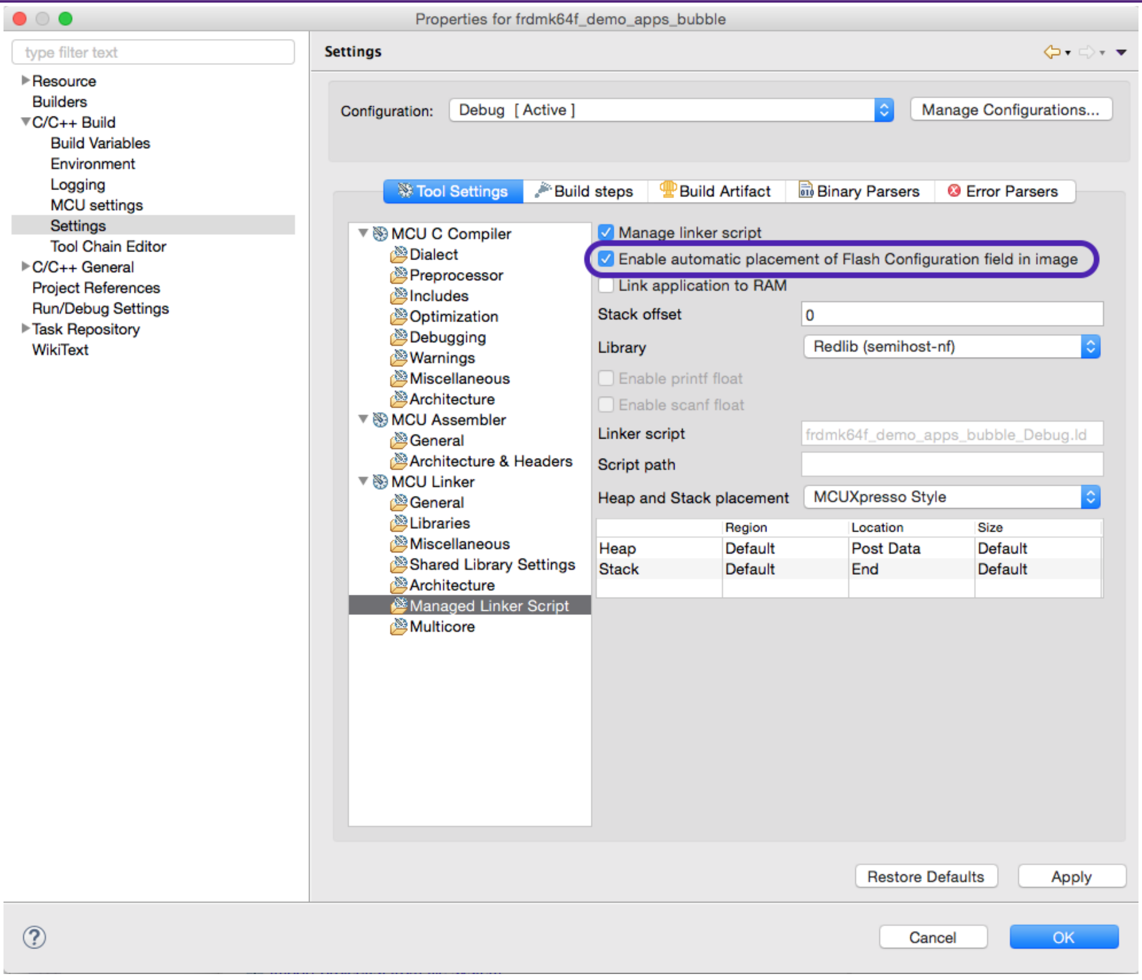
The value of the Flash Configuration block for a project is provided by the following structure which will be found in the startup code:

```
__attribute__ ((used,section(".FlashConfig"))) const struct {
    unsigned int word1;
    unsigned int word2;
    unsigned int word3;
    unsigned int word4;
} Flash_Config = {0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFE};
```

which is then placed appropriately by the linker script generated by the managed linker script mechanism.

```
/* Kinetis Flash Configuration data */
. = 0x400 ;
PROVIDE(__FLASH_CONFIG_START__ = .) ;
KEEP(*(.FlashConfig))
PROVIDE(__FLASH_CONFIG_END__ = .) ;
ASSERT(!(__FLASH_CONFIG_START__ == __FLASH_CONFIG_END__),
 "Linker Flash Config Support Enabled, but no .FlashConfig
 section provided within application");
/* End of Kinetis Flash Configuration data */
```

Important Note: The support for placing the Flash Configuration Block can be disabled by unticking a checkbox of the Project Properties linker Target tab. However this is generally not advisable as it is very likely to result in a locked MCU.



Configuration: Debug [Active]

- [✓] Manage linker script
- [✓] Enable automatic placement of Flash Configuration field in image
- [ ] Link application to RAM

Stack offset: 0
Library: Redlib (semihost-nf)
- [ ] Enable printf float
- [ ] Enable scanf float

Linker script: frdmk64f_demo_apps_bubble_Debug.ld
Script path:
Heap and Stack placement: MCUXpresso Style

	Region	Location	Size
Heap	Default	Post Data	Default
Stack	Default	End	Default

Figure 14.5. Linker Settings

14.6.3 Placement of USB data

For MCUs where part support is imported from an SDK, the managed linker script mechanism supports the automatic placement of USB global data (as used by the SDK USB Drivers), including for parts with dedicated USB_RAM (small or large variants).

14.7 Modifying the generated linker script / memory layout

The linker script generated by the managed linker script mechanism will be suitable for use, as is, for many applications. However in some circumstances you may need to make changes. MCUXpresso IDE provides a number of mechanisms to allow you to do this whilst still being able to use the managed linker script mechanism. These include:

- Changing the layout and order of memory using the Memory Configuration Editor
- Changing the size and location of the stack and heap using the Heap and Stack Editor
- Decorating the definitions of variables and functions in your source code with macros from the *cr_section_macros.h* to cause them to be placed into different memory blocks
- Providing project specific versions of Freemarker linker script templates to change particular aspects of how the managed linker script mechanism creates the final linker script

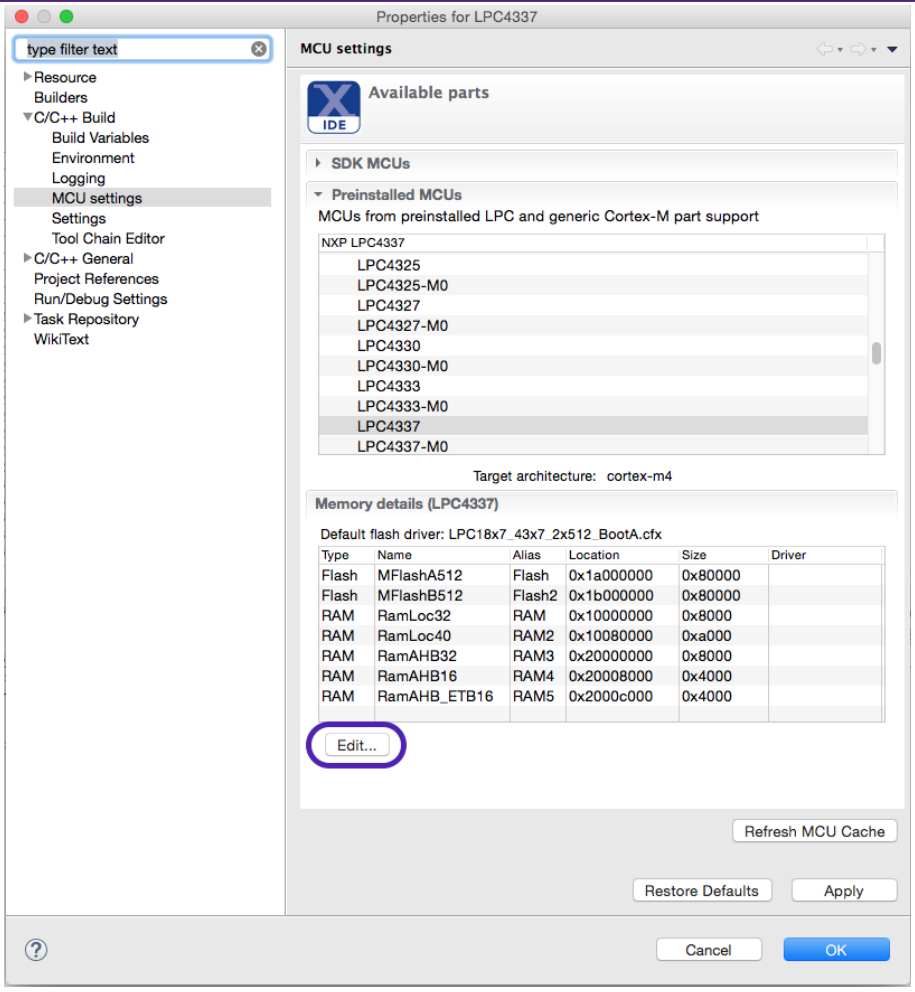
The following sections describe these in more detail.

14.8 Using the Memory Configuration Editor

The Memory Editor is accessed via the MCU settings dialog, which can be found at

Project Properties -> C/C++ Build -> MCU settings

This lists the memory details for the selected MCU, and will, by default, display the memory regions that have been defined by the MCUXpresso IDE itself.



MCUs from preinstalled LPC and generic Cortex-M part support

NXP LPC4337
- LPC4325
- LPC4325-M0
- LPC4327
- LPC4327-M0
- LPC4330
- LPC4330-M0
- LPC4333
- LPC4333-M0
- LPC4337
- LPC4337-M0

Target architecture: cortex-m4

Memory details (LPC4337)

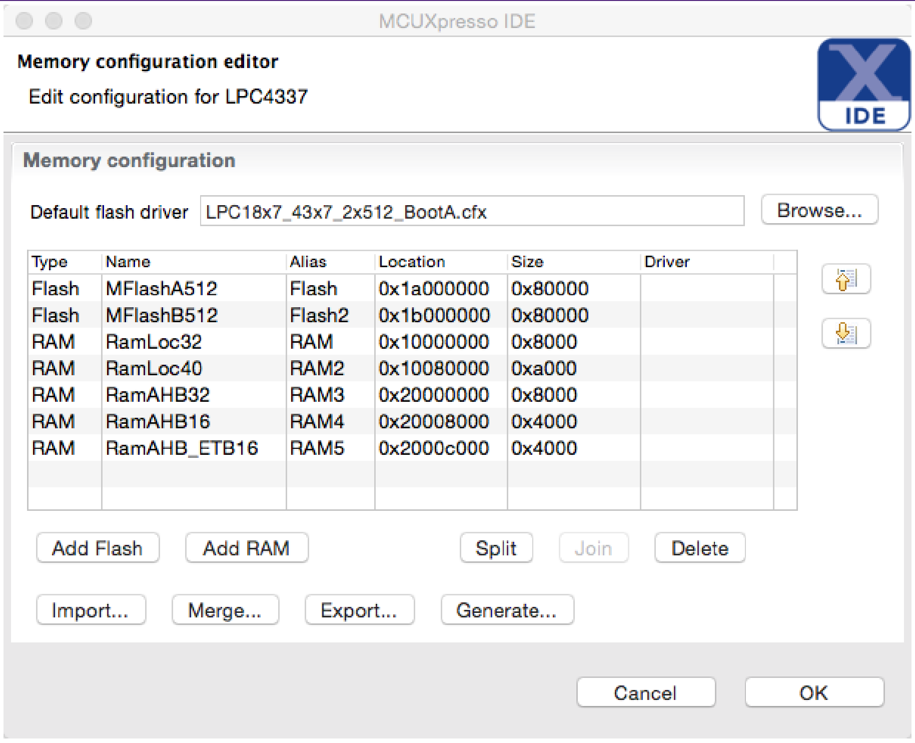
Default flash driver: LPC18x7_43x7_2x512_BootA.cfx

Type	Name	Alias	Location	Size	Driver
Flash	MFlashA512	Flash	0x1a000000	0x80000	
Flash	MFlashB512	Flash2	0x1b000000	0x80000	
RAM	RamLoc32	RAM	0x10000000	0x8000	
RAM	RamLoc40	RAM2	0x10080000	0xa000	
RAM	RamAHB32	RAM3	0x20000000	0x8000	
RAM	RamAHB16	RAM4	0x20008000	0x4000	
RAM	RamAHB_ETB16	RAM5	0x2000c000	0x4000	

Figure 14.6. LPC4337... default memory regions

14.8.1 Editing a Memory Configuration

In the example below, we will show how the default memory configuration for an LPC4337... can be changed. Selecting the **Edit...** button will launch the **Memory configuration editor** dialog — see Figure 14.7.


MCUXpresso IDE

Memory configuration editor

Edit configuration for LPC4337

Memory configuration

Default flash driver LPC18x7_43x7_2x512_BootA.cfx

Type	Name	Alias	Location	Size	Driver
Flash	MFlashA512	Flash	0x1a000000	0x80000	
Flash	MFlashB512	Flash2	0x1b000000	0x80000	
RAM	RamLoc32	RAM	0x10000000	0x8000	
RAM	RamLoc40	RAM2	0x10080000	0xa000	
RAM	RamAHB32	RAM3	0x20000000	0x8000	
RAM	RamAHB16	RAM4	0x20008000	0x4000	
RAM	RamAHB_ETB16	RAM5	0x2000c000	0x4000	

Figure 14.7. Memory configuration editor

Known blocks of memory, with their type, base location, and size are displayed. Entries can be created, deleted, etc by using the provided buttons.

For simplicity, the **additional** memory regions are given sequential aliases, starting from 2, so RAM2, RAM3 etc (as well as using their “formal” region name – for example RamAHB32).

Table 14.1. Memory editor controls

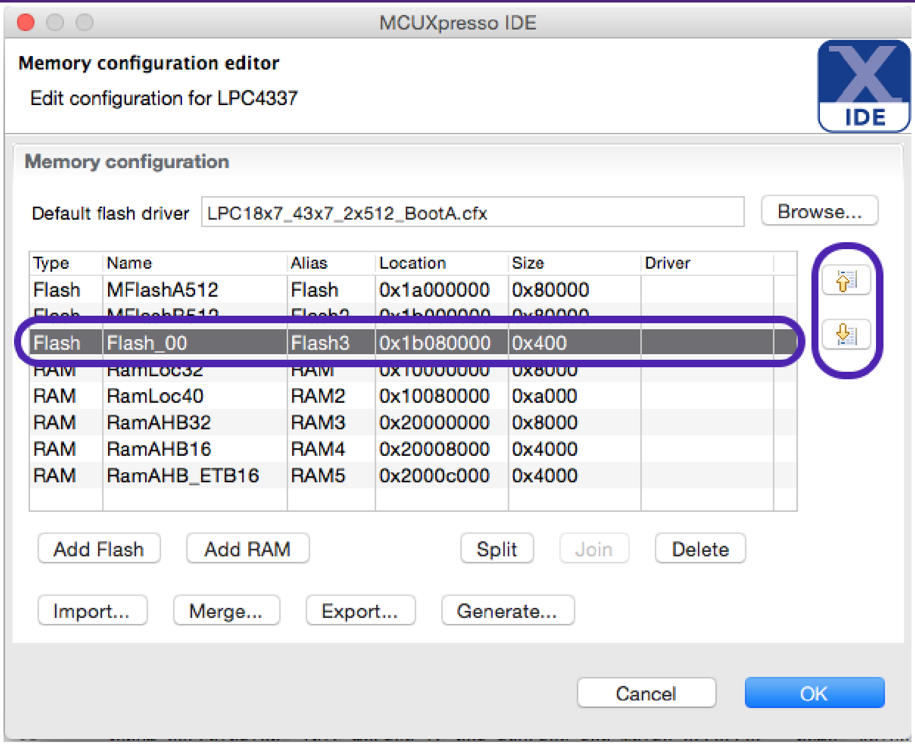
Button	Details
Add Flash	Add a new memory block of the appropriate type.
Add RAM	Add a new memory block of the appropriate type.
Split	Split the selected memory block into two equal halves.
Join	Join the selected memory block with the following block (if the two are contiguous).
Delete	Delete the selected memory block.
Import	Import a memory configuration that has been exported from another project, overwriting the existing configuration.
Merge	Import a partial memory configuration from a file, merging it with the existing memory configuration. This allows you, for example, to add an external Flash bank definition to an existing project.
Export	Export a memory configuration for use in another project.
Up / Down	Reorder memory blocks. This is important: if there is no Flash block, then code will be placed in the first RAM block, and data will be placed in the block following the one used for the code (regardless of whether the code block was RAM or Flash).
Generate	Generates local part support for the selected MCU.
Driver	Highlighted in blue, shows the selection of a per-Flash region Flash driver. Click this field to see a drop down of all available drivers. Please see: LinkServer Flash Support [90]
Browse(Flash driver)	Select the appropriate driver for programming the Flash memory specified in the memory configuration. This is only required when the Flash memory is external to the MCU. Flash drivers for external Flash must have a “.cfx” file extension and must be located in the ide\bin\Flash subdirectory of the MCUXpresso IDE installation.

The name, location, and size of this new region can be edited in place. **Note** that when entering the size of the region, you can enter full values in decimal or in hex (by prefixing with `0x`), or by specifying the size in kilobytes or megabytes. For example:

- To enter a region size of 32KB, enter `32768`, `0x8000` or `32k`.
- To enter a region size of 1MB, enter `0x100000` or `1m`.

Note: memory regions must be located on four-byte boundaries, and be a multiple of four bytes in size.

The screenshot below shows the dialog after the “Add Flash” button has been clicked. Use the highlighted up/down buttons to move this region to be top in the list.



MCUXpresso IDE

Memory configuration editor

Edit configuration for LPC4337

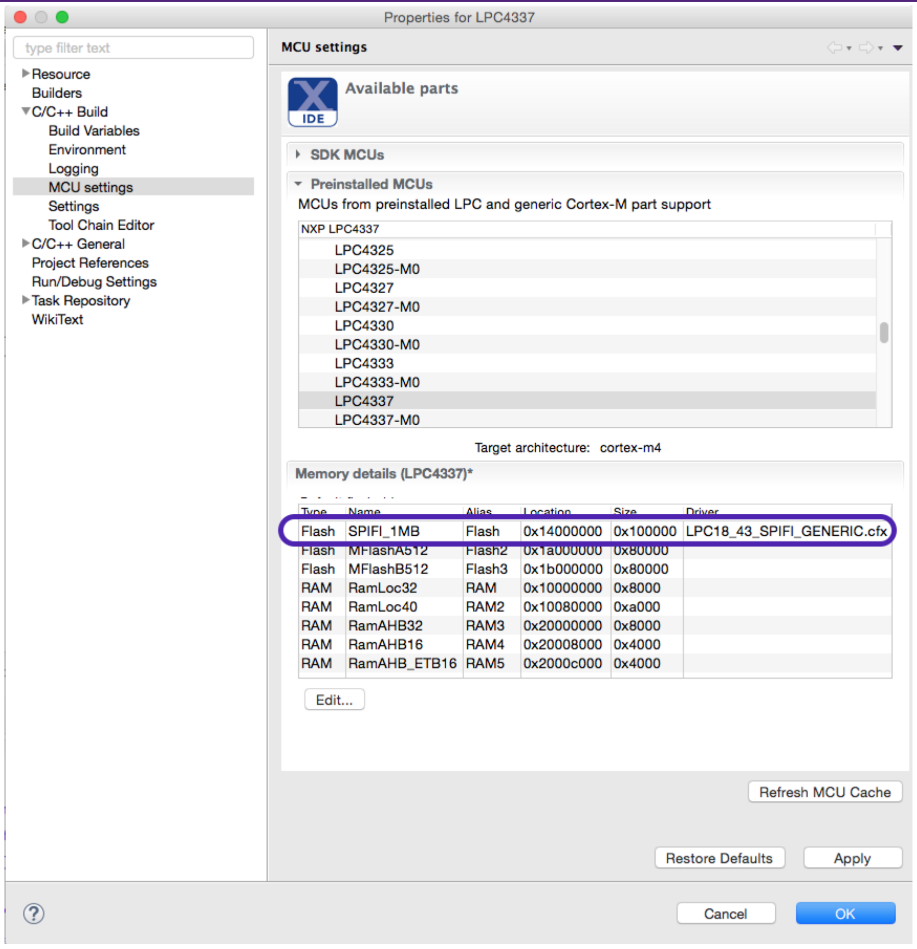
Memory configuration

Default flash driver: LPC18x7_43x7_2x512_BootA.cfx

Type	Name	Alias	Location	Size	Driver
Flash	MFlashA512	Flash	0x1a000000	0x80000	
Flash	MFlashB512	Flash2	0x1b000000	0x80000	
Flash	Flash_00	Flash3	0x1b080000	0x400	
RAM	RamLoc32	RAM	0x10000000	0x8000	
RAM	RamLoc40	RAM2	0x10080000	0xa000	
RAM	RamAHB32	RAM3	0x20000000	0x8000	
RAM	RamAHB16	RAM4	0x20008000	0x4000	
RAM	RamAHB_ETB16	RAM5	0x2000c000	0x4000	

Figure 14.8. Effect of Add Flash

After updating the new memory configuration, click **OK** to return to the MCU settings dialog, which will be updated to reflect the new configuration.



MCUs from preinstalled LPC and generic Cortex-M part support

NXP LPC4337
LPC4325
LPC4325-M0
LPC4327
LPC4327-M0
LPC4330
LPC4330-M0
LPC4333
LPC4333-M0
LPC4337
LPC4337-M0

Target architecture: cortex-m4

Memory details (LPC4337)*

Type	Name	Alias	Location	Size	Driver
Flash	SPIFI_1MB	Flash	0x14000000	0x100000	LPC18_43_SPIFI_GENERIC.cfx
Flash	MFlashA512	Flash2	0x1a000000	0x80000	
Flash	MFlashB512	Flash3	0x1b000000	0x80000	
RAM	RamLoc32	RAM	0x10000000	0x8000	
RAM	RamLoc40	RAM2	0x10080000	0xa000	
RAM	RamAHB32	RAM3	0x20000000	0x8000	
RAM	RamAHB16	RAM4	0x20008000	0x4000	
RAM	RamAHB_ETB16	RAM5	0x2000c000	0x4000	

Figure 14.9. Updated MCU settings

Here you can see that the region has been named SPIFI_1MB, and the default Flash driver has been deleted and the Generic SPIFI driver selected for the newly created SPIFI_1MB region.

MCUXpresso IDE provides extended support for the creation and programming of projects that span multiple Flash devices. In addition to a single default Flash driver, per region Flash drivers can also be specified (as above). Using this scheme projects can be created that span Flash regions and can be programmed in a single ‘debug’ operation.

Note: that once the memory details have been modified, the selected MCU as displayed on the “Status Bar” (at the bottom of the IDE window) will be displayed with an asterisk (*) next to it. This provides an indication that the MCU memory configuration settings for the selected project have been modified.

14.8.2 Device specific vs Default Flash Drivers

When a project is configured to use additional Flash devices via the Memory Configuration Editor, the Flash driver to be used for programming that Flash device has to be specified in the Driver column. Typically for a SPIFI device, this should be:

- *LPC18_43_SPIFI_GENERIC.cfx* (for *LPC18/LPC43* series MCUs)
- *LPC40xx_SPIFI_GENERIC.cfx* (for *LPC407x/8x* MCUs)
- *LPC5460x_SPIFI_GENERIC.cfx* (for *LPC5460x* MCUs).

14.8.3 Restoring a Memory Configuration

To restore the memory configuration of a project back to the default settings, simply reselect the MCU type, or use the “Restore Defaults” button, on the MCU Settings properties page.

14.8.4 Copying Memory Configurations

Memory configurations can be exported for import into another project. Use the Export and Import buttons for this purpose.

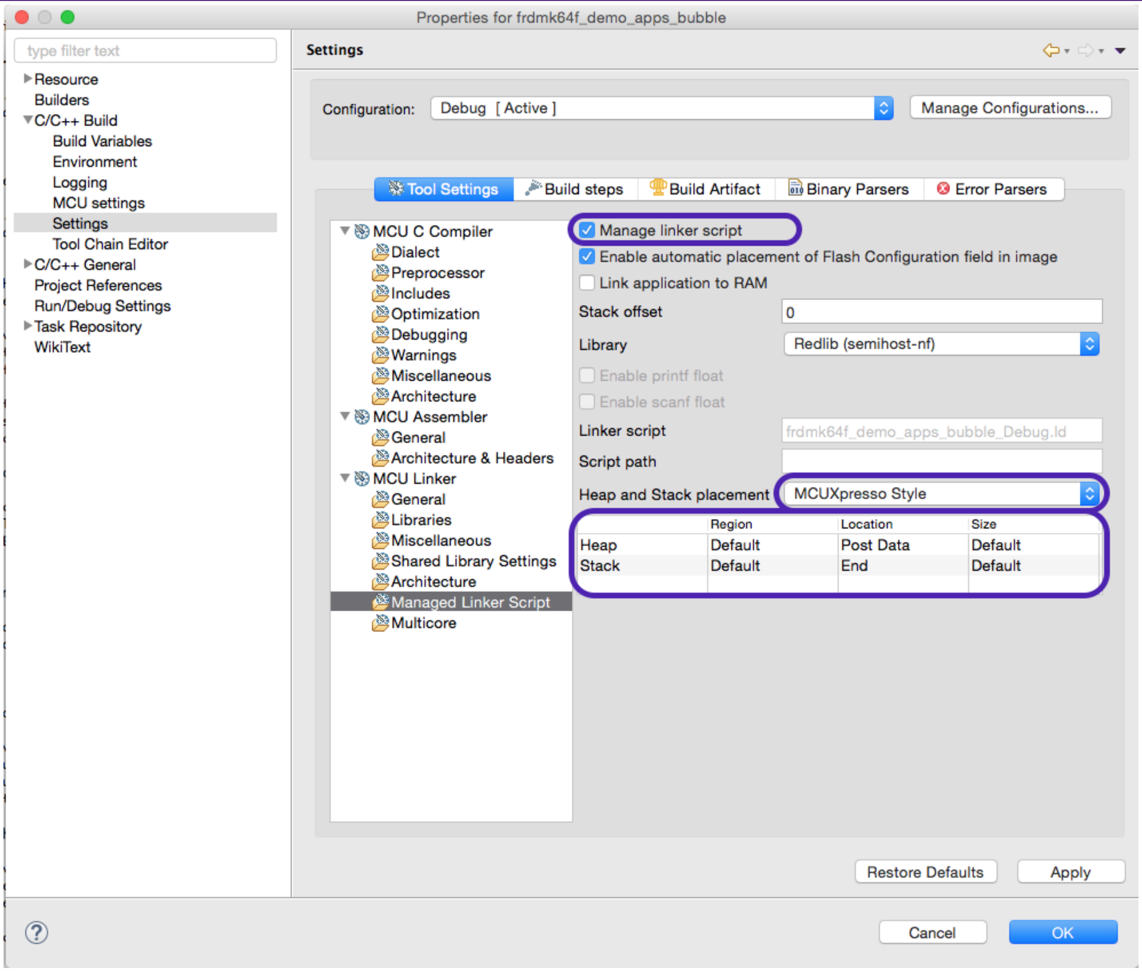
MCUXpresso IDE provides a standard memory layout for each known MCU. In addition, the MCUXpresso IDE supports the editing of the target memory layout used for a project. This allows for the details of external Flash to be defined or for the layout of internal RAM to be reconfigured. Also, it allows a Flash driver to be allocated for use with parts with no internal Flash, but where an external Flash part is connected.

14.9 More advanced heap/stack placement

MCUXpresso IDE provides two models of heap/stack placement. The first of these is the “LPCXpresso Style”, which is the mechanism provided by the previous generation LPCXpresso IDE. This is the default model used for projects created for Preinstalled MCUs. The second model is the “MCUXpresso style”. This is the default model used for projects created for MCUs imported from SDKs.

The heap/stack placement model being used for a particular project/build configuration can be modified by right clicking on the project and selecting:

Project Properties -> C/C++ Build -> Settings -> MCU Linker -> Managed Linker Scripts


Properties for frdmk64f_demo_apps_bubble

Settings

Configuration: Debug [Active]

Manage linker script [✓]

Enable automatic placement of Flash Configuration field in image [✓]

Link application to RAM [ ]

Stack offset: 0

Library: Redlib (semihost-nf)

Enable printf float [ ]

Enable scanf float [ ]

Linker script: frdmk64f_demo_apps_bubble_Debug.ld

Script path:

Heap and Stack placement: MCUXpresso Style

	Region	Location	Size
Heap	Default	Post Data	Default
Stack	Default	End	Default

Figure 14.10. MCUXpresso IDE Linker Settings

In the dialogue above, highlights show the managed linker script option along with the selection of the MCUXpresso Style scheme.

14.9.1 MCUXpresso style heap and stack

By default the heap and stack are placed in the “default” memory region (i.e. the first RAM block displayed in the memory configuration area), with the heap placed after the application’s data and the stack rooted at the top of this block.

However, using the Heap and Stack editor in Project Properties, it is very simple to individually change the stack and heap locations (both the memory block used, and the location within that block), and also the size of the memory to be used by each of them.

Region

- Default : Place into first RAM bank as shown in Memory Configuration Editor
- List of memory regions, and aliases, as show in Memory Configuration Editor

Location

- Start : Place at start of specified RAM bank.
- Post Data : Place after any data in specified RAM bank. Default for heap.
- End : Place at end of specified RAM bank. Default for stack.

Size

- Default: 1/16th of the memory region size, up to a maximum of 4KB (and a minimum of 128bytes). Hovering the cursor over the field will show the current value that will be used.
- Value : Specify exact required size. Must be a multiple of 4. Note that when entering the size of the region, you can enter full values in decimal or in hex (by prefixing with 0x), or by specifying the size in Kilobytes (or Megabytes). For example:
  - To enter a size of 32KB, enter 32768, 0x8000 or 32k.
  - A value of 0 can be entered to prevent any heap use by an application.
    - Note: For semihosted printf to operate without any heap space, you must enable the “character only” version. For Redlib, define the symbol “CR_PRINTF_CHAR” (at the project level) and remove other semhosting defines such as CR_INTEGER_PRINTF. Character only semihosted printf is significantly slower than the default version and may display differently depending on your debug solution.

Note: The MCUXpresso style of setting heap and stack has the advantage over the LPCXpresso style described below in that the memory allocated for heap/stack usage is also taken into account in the image size information displayed in the Build console when your project is built.

14.9.2 LPCXpresso style heap and stack

By default the heap and stack are still placed in the “default” memory region (i.e. the first RAM block displayed in the memory configuration area), with the heap placed after the application’s data and the stack rooted at the top of this block.

To relocate the stack or heap, or provide a maximum extent of the heap, then the linker “--defsym” option can be used to define one or more of the following symbols:

```
__user_stack_top
__user_heap_base
_pvHeapLimit
```

To do this, use the __MCU Linker – Miscellaneous – Other Options_ box in Project Properties.

For example:

--defsym=__user_stack_top=__top_RAM2

- Locate the stack at the top of the second RAM bank (as listed in the memory configuration editor)
- Note : The symbol __top_RAM2 is defined in the project by the managed linker script mechanism at:

```
<projname>_<buildconfig>_mem.ld
```

--defsym=__user_heap_base=__end_bss_RAM2

- Locate the start of the heap in the second RAM bank, after any data that has been placed there

--defsym=_pvHeapLimit=__end_bss_RAM2+0x8000

- Locate the end of the heap in the second RAM bank, offset by 32KB from the end of any data that has been placed there

--defsym=_pvHeapLimit=0x10004000

- Locate the end of the heap at the absolute address 0x10004000

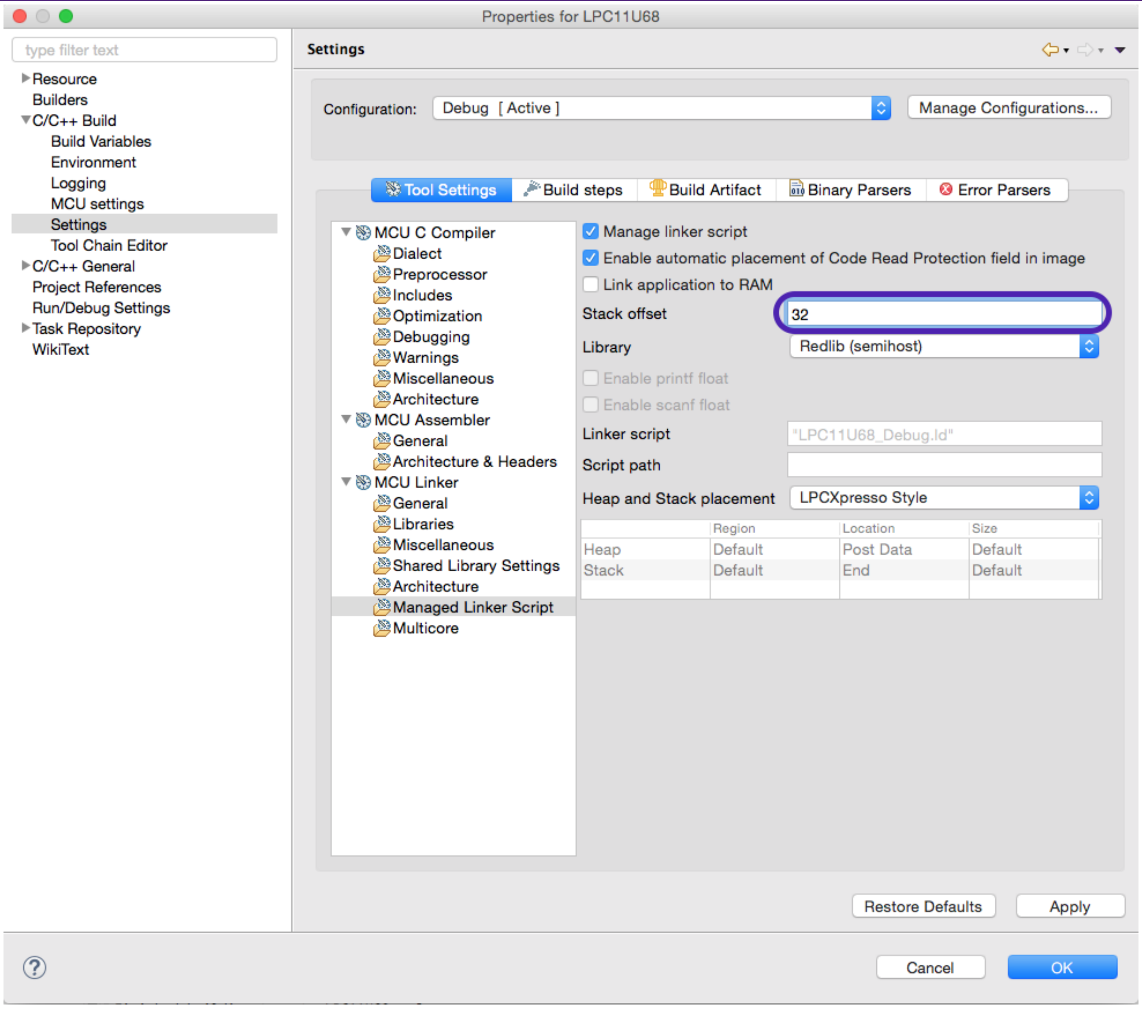
14.9.3 Reserving RAM for IAP Flash Programming

The IAP Flash programming routines available in NXP’s LPC MCUs generally make use of some of the onchip RAM when executed. For example on the LPC1343 the top 32 bytes of onchip RAM are used. Thus if you are calling the IAP routines from your own application, you need to ensure that this memory is not used by your main application – which typically means by the stack.

However, with the managed linker script mechanism, it is easy to modify the start position of the stack (remember that stacks grow down) to avoid this clash with the IAP routines. To do this go to:

Project Properties -> C/C++ Build -> Settings -> MCU Linker -> Manager Linker Script

and modify the value in the “Stack Offset” field from 0 to 32. This will work whether you are using LPCXpresso style or MCUXpresso style of heap/stack placement.


Properties for LPC11U68

Settings

Configuration: Debug [Active]

☑ Manage linker script
☑ Enable automatic placement of Code Read Protection field in image
☐ Link application to RAM

Stack offset: 32

Library: Redlib (semihost)

☐ Enable printf float
☐ Enable scanf float

Linker script: "LPC11U68_Debug.ld"

Script path:

Heap and Stack placement: LPCXpresso Style

	Region	Location	Size
Heap	Default	Post Data	Default
Stack	Default	End	Default

Figure 14.11. MCUXpresso IDE Linker Reserve Stack Space

The value you enter in this field must be a multiple of 4.

You are also advised to check the documentation for the actual MCU that you are using to confirm the amount of memory required by the IAP routines.

14.9.4 Stack checking

Although, as described above, it is possible to define a size of memory to be used for the stack, Cortex-M CPUs have no support for hardware stack checking. Thus if you want to automatically detect if the stack exceeds the memory set aside for it – other mechanisms must be used. For example:

- Locate stack to fall off start of memory block and trigger fault
- Include code that sets the stack to a known value, and periodically checks whether the lowest address has been overwritten.
- When debugging, set a watchpoint on the lowest address the stack is allowed to reach
- Use the Memory Protection Unit (MPU) to detect overflow, on parts which implement one

14.9.5 Heap Checking

By default, the heap used by the malloc() family of routines grows upwards from the end of the user data in RAM up towards the stack – a “one region memory model”.

When a new block of memory is requested, the memory allocation function _sbrk() will make a call to the following function to check for heap overflow:

```
unsigned __check_heap_overflow (void * new_end_of_heap)
```

This should return:

- **1** - If the heap will overflow
- **0** - If the heap is still OK

If 1 is returned, Redlib’s malloc() will set errno to ENOMEM and return a null pointer to the caller

The default version of **__check_heap_overflow()** built into the MUCXpresso IDE supplied C libraries carry out no checking unless the symbol “_pvHeapLimit” has been created in your image, to mark the end location of the heap.

This symbol will have been created automatically if you are using the MCUXpresso style of heap and stack placement described earlier in this chapter. Or alternatively if using the LPCXpresso style of heap and stack placements, you can use the --defsym option to set this.

If you wish to use a different means of heap overflow checking, then you can find a reference implementation of **__check_heap_overflow()** in the file **_cr_check_heap.c** that can be found in the Examples subdirectory of your IDE installation.

This file also provides functionality to allow simple heap overflow checking to be done by looking to see if the heap has reached the current location of the stack point, which of course assumes that the heap and stack are in the same region. This check is not enabled by default implementation within the C library as it can break in some circumstances – for example when the heap is being managed by an RTOS.

14.9.6 Placement of specific code/data items

It is possible to make small changes to the placement of specific code/data items within the final image without modifying the Freemarker linker script templates. Such placement can be controlled via macros provided in an MCUXpresso IDE supplied header file which can be pulled into your project using:

```
#include <cr_section_macros.h>
```

Placing data into different RAM blocks

Many MCUs provide more than one bank of RAM. By default the managed linker script mechanism will place all of the application data and bss (as well as the heap and stack) into the first bank of RAM.

However it is also possible to place specific data or bss items into any of the defined banks for the target MCU, as displayed in the Memory Configuration Editor, by decorating their definitions in your source code with macros from the cr_section_macros.h MCUXpresso IDE supplied header file

For simplicity, the **additional** memory regions are named sequentially, starting from 2, so RAM2, RAM3 etc (as well as using their “formal” region name – for example RamAHB32).

For example, the LPC1768 has a second bank of RAM at address 0x2007c000. The managed linker script mechanism creates a data (and equivalent bss) load section for this region thus:

```
.data_RAM2 : ALIGN(4)
{
    FILL(0xff)
    *(.data.$RAM2*)
    *(.data.$RamAHB32*)
} > RamAHB32 AT>MFlash512
```

To place data into this section, you can use the __DATA macro, thus:

```
// create an initialised 1k buffer in RAM2
__DATA(RAM2) char data_buffer[1024];
```

Or the __BSS macro:

```
// create a zero-init buffer in RAM2
__BSS(RAM2) char bss_buffer[128];
```

In some cases you might need a finer level of granularity than just placing a variable into a specific memory bank, and rather need to place it at a specific address. In such a case you could then edit the predefined memory layout for your particular project using the “Memory Configuration Editor” to divide up (and rename) the existing banks of RAM. This then allows you to provide a specific named block of RAM into which to place the variable that you need at a specific address, again by using the attribute macros provided by the “cr_section_macros.h” header file.

Noinit Memory Sections

Normally global variables in an application will end up in either a “.data” (initialized) or “.bss” (zero-initialized) data section within your linked application. Then when your application starts executing, the startup code will automatically copy the initial values of “.data” sections from Flash to RAM, and zero-initialize “.bss” data sections directly in RAM.

MCUXpresso IDE’s managed linker script mechanism also supports the use of “.noinit” data within your application. Such data is similar to “.bss” except that it will not get zero-initialized during startup.

Note: Great care must be taken when using “.noinit” data such that your application code makes no assumptions about the initial value of such data. This normally means that your application code will need to explicitly set up such data before using it – otherwise the initial value of such a global variable will basically be random (i.e. it will depend upon the value that happens to be in RAM when your system powers up).

One common example of using such .noinit data items is in defining the frame buffer stored in SDRAM in applications which use an onchip LCD controller (for example NXP LPC178x and LPC408x parts).

Making global variables noinit

The linker script generated by the MCUXpresso IDE managed linker script mechanism will contain a section for each RAM memory block to contain “.noinit” items, as well as the “.data” and “.bss” items. Note that for a particular RAM memory block, all “.data” items will be placed first, followed by “.bss” items, and then “.noinit” items.

However, normally for a particular RAM memory block where you are going to be put “.noinit” items, you would actually be making all of the data placed into that RAM “.noinit”.

The “cr_section_macros.h” header file then defines macros which can be used to place global variables into the appropriate “.noinit” section. First of all include this header file:

```
#include <cr_section_macros.h>
```

The __NOINIT macro can then be used thus:

```
// create a 128 byte noinit buffer in RAM2
__NOINIT(RAM2) char noinit_buffer[128];
```

And if you want “.noinit” items placed into the default RAM bank, then you can use the __NOINIT_DEF macro thus:

```
// create a noinit integer variable in the main block of RAM
__NOINIT_DEF int noinit_var ;
```

Placing code/rodata into different FLASH blocks

Most MCUs only have one bank of Flash memory. But with some parts more than one bank may be available – and in such cases, by default, the managed linker script mechanism will still place all of the application code and rodata (consts) into the first bank of Flash (as displayed in the Memory Configuration Editor).

For example:

- most of the LPC18 and LPC43xx parts containing internal Flash (such as LPC1857 and LPC4357) actually provide dual banks of Flash.
- some MCUs have the ability to access external Flash (typically SPIFI) as well as their built-in internal Flash (e.g. LPC18xx, LPC40xx, LPC43xx, LPC546xx).

However it is also possible to place specific functions or rodata items into the second (or even third) bank of Flash. This placement is controlled via macros provided in the *"cr_section_macros.h"* header file.

For simplicity, the **additional** Flash region can be referenced as Flash2 (as well as using its “formal” region name – for example MFlashB512 – which will vary depending upon part).

First of all include this header file:

```
#include <cr_section_macros.h>
```

Then, for example, to place a rodata item into this section, you can use the __RODATA macro, thus:

```
__RODATA(Flash2) const int roarray[] = {10,20,30,40,50};
```

Or to place a function into it you can use __TEXT macro:

```
__TEXT(Flash2) void systick_delay(uint32_t delayTicks) {
 :
}
```

In addition, the __RODATA_EXT and __TEXT_EXT macros can be used to place functions/ rodata into a more specifically named section, for example:

```
__TEXT_EXT(Flash2,systick_delay) void systick_delay(uint32_t delayTicks) {
 :
}
```

will be placed into the section ".text.$Flash2.systick_delay" rather than ".text.$Flash2".

Placing specific functions into RAM blocks

In most modern MCUs with built-in Flash memory, code is normally executed directly from Flash memory. Various techniques, such as prefetch buffering are used to ensure that code will execute with minimal or zero wait states, even a higher clock frequencies. Please see the documentation for the MCU that you are using for more details.

However it is also possible to place specific functions into any of the defined banks of RAM for the target MCU, as displayed in:

Project -> Properties -> C/C++ Build -> MCU settings

and sometimes there can be advantages in relocating small, time critical functions so that they run out of RAM instead of Flash.

For simplicity, the **additional** memory regions are named sequentially, starting from 2, (as well as using their "formal" region name – for example RamAHB32). So for a device with 3 RAM regions, alias names RAM, RAM2 and RAM3 will be available.

This placement is controlled via macros provided in a header file which can be pulled into your project using:

```
#include <cr_section_macros.h>
```

The macro __RAMFUNC can be used to locate a function into a specific RAM region.

For example, to place a function into the main RAM region, use:

```
__RAMFUNC(RAM) void fooRAM(void) {...
```

To place a function into the RAM2 region, use:

```
__RAMFUNC(RAM2) void fooRAM2(void) {...
```

Alternatively, RAM can be selected by formal name (as listed in the memory configuration editor), for example:

```
__RAMFUNC(RamAHB32) void HandlerRAM(void) {...
```

In order to initialize RAM based code (and data) into specified RAM banks, the managed linker script mechanism will create a “Global Section Table” in your image, directly after the vector table. This contains the addresses and lengths of each of the data (and bss) sections, so that the startup code can then perform the necessary initialization (copy code/data from Flash to RAM) .

Long branch veneers and debugging

Due to the distance in the memory map between Flash memory and RAM, you will typically require a “long branch veneer” between the function in RAM and the calling function in Flash. The linker can automatically generate such a veneer for direct function calls, or you can effectively generate your own by using a call via a function pointer.

One point to note is that debugging code with a linker generated veneer can sometimes cause problems. This veneer will not have any source level debug information associated with it, so that if you try to step in to a call to your code in RAM, typically the debugger will step over it instead.

You can work around this by single stepping at the instruction level, setting a breakpoint in your RAM code, or by changing the function call from a direct one to a call via a function pointer.

Reducing Code Size when support for LPC CRP or Kinetis Flash Config Block is enabled

One of the consequences of the way that LPC CRP and Kinetis Flash Configuration Blocks work is that the memory between the CPU’s vector table and the CRP word/ Flash Config Block is often left largely unused. This can typically increases the size of the application image by several hundred bytes (depending upon the MCU being used).

However this unused space can easily be reclaimed by choosing one or more functions to be placed into this unused memory. To do this, you simply need to decorate their definitions with the macro __AFTER_VECTORS which is supplied in the “cr_section_macros.h” header file

Obviously in order to do this effectively, you need to identify functions which will occupy as much of this unused memory as possible. The best way to do this is to look at the linker map file.

MCUXpresso IDE startup code already uses this macro to place the various initialization functions and default exception handlers that it contains into this space, thus reducing the ‘default’ unused space. But you can also place additional functions there by decorating their definitions with the macro, for example

```
__AFTER_VECTORS void myStartupFunction(void);
```

Note you will get a link error if the __AFTER_VECTORS space grows beyond the CRP/Flash Configuration Block (when this support is enabled):

```
myproj_Debug.ld:98 cannot move location counter backwards (from 00000334
to 000002fc)
collect2: ld returned 1 exit status
make: *** [myproj.axf] Error 1
```

In this case, you will need to remove the __AFTER_VECTORS macro from the definition of one or more of your functions.

14.10 Freemarker Linker Script Templates

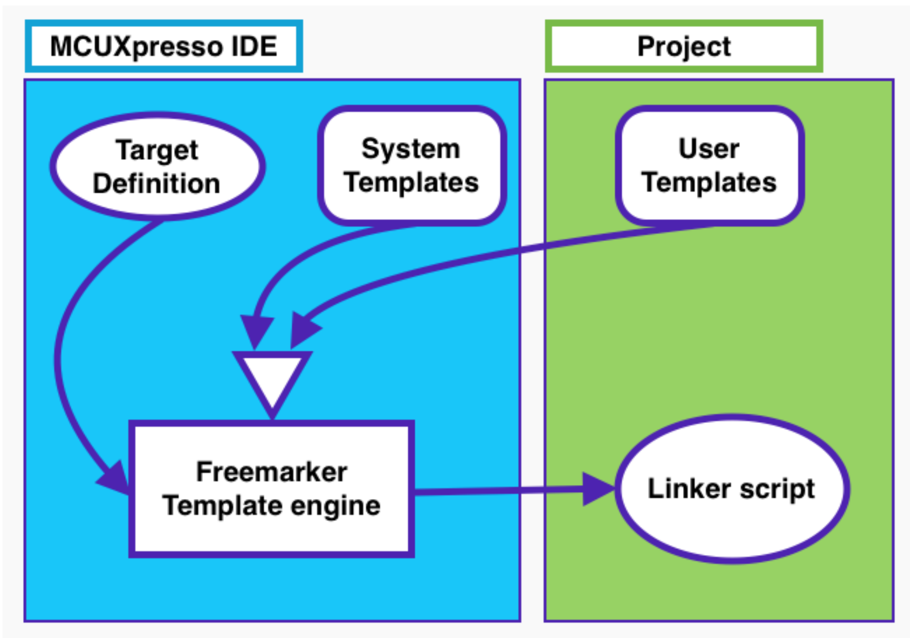
By default, MCUXpresso IDE projects use a managed linker script mechanism which automatically generates a linker script file without user intervention – allowing the project code and data to be laid out in memory based on the IDE's knowledge of the memory layout of the target MCU.

However sometimes the linker script generated in this way may not provide exactly the memory layout required. MCUXpresso IDE therefore provides a highly flexible and powerful linker script template mechanism to allow the user to change the content of the linker script generated by the managed linker script mechanism

14.10.1 Basics

FreeMarker is a template engine: a generic tool to generate text output (HTML web pages, e-mails, configuration files, source code, etc.) based on templates and changing data. Built into MCUXpresso IDE are a set of templates that are processed by the Freemarker template engine to create the linker script. Templates are written in the FreeMarker Template Language (FTL), which is a simple, specialized language, not a full-blown programming language like PHP. Full documentation for Freemarker can be found at :http://freemarker.org/docs/index.html":http://freemarker.org/docs/index.html .

MCUXpresso IDE automatically invokes Freemarker, passing it a data model that describes the memory layout of the target together with a 'root' template that is processed to create the linker script. This root template, #includes further 'component' templates. This structure allows a linker script to be broken down into various components, and allows a user to provide their own templates for a component, instead of having to (re-)write the whole template. For example, component templates are provided for text, data and bss sections, allowing the user to provide a different implementations as necessary, but leaving the other parts of the linker script untouched.



14.10.2 Reference

Freemarker reads input files, copying text and processing Freemarker directives and 'variables', and writes an output file. As used by the MCUXpresso IDE managed linker script mechanism, the input files describe the various components of a linker script which, together with variables defined by the IDE, are used to generate a complete linker script. Any of the component template input files may be overridden by providing a local version in the project.

The component template input files are provided as a hierarchy, shown below, where each file #includes those files nested below. This allows for individual components of the linker script to be overridden without having to supply the entire linker script, increasing flexibility, while maintaining the benefits of Managed Linker Scripts.

Linker script template hierarchy

linkscript.ldt (top level)

- user.ldt (an empty file designed to be overridden by users that is included in linkscript, memory and library templates)
- user_linkscript.ldt (an empty file designed to be overridden by users that is included in linkscript only)
- linkscript_common.ldt (root for main content)
  - header.ldt (the header for scripts)
    - listvars.ldt (a script to output a list of all predefined variables available to the template)
  - includes.ldt (includes the memory and library scripts)
  - section_top.ldt (top of the linker script SECTION directive)
  - text_section.ldt (text sections for each secondary Flash)
    - text_section_multicore.ldt (text sections for multicore targets)
    - text.ldt (for inserting *text)
    - rodata.ldt (for inserting rodata)
  - boot_hdr.ldt (allows placement of optional header before main code section)
    - boot_hdr_partfamily.ldt
  - main_text_section.ldt (the primary text section)
    - global_section_table.ldt (the global section table)
    - crp.ldt (the CRP information)
    - main_text.ldt (for inserting *text)
    - main_rodata.ldt (read-only data)
    - cpp_info.ldt (additional C++ requirements)
  - exdata.ldt (the exdata sections)
  - end_text.ldt (end of text marker)
  - usb_ram_section.ldt (placement of SDK USB data structures)
  - stack_heap_sdk_start.ldt (placement of MCUXpresso style heap/stack)
  - data_section.ldt (data sections for secondary ram)
    - data_section_multicore.ldt (data sections for multicore targets)
    - data.ldt (for inserting *data)
  - mtb_default_section.ldt (special section for MTB (cortex-m0+ targets)
  - uninit_reserved_section.ldt (uninitialised data)
  - main_data_section.ldt primary data section)
    - main_data.ldt (for inserting *data)
  - bss_section.ldt (secondary bss sections)
    - bss.ldt (for inserting *bss)
  - main_bss_section.ldt primary bss section)
    - main_bss.ldt (for inserting *bss)
  - noinit_section.ldt (no-init data)
  - noinit_noload_section.ldt (no-load data)
  - stack_heap_sdk_postdata.ldt (placement of MCUXpresso style heap/stack)
  - stack_heap_sdk_end.ldt (placement of MCUXpresso style heap/stack)
  - stack_heap.ldt (define the stack and heap)
  - checksum.ldt (create the LPC checksum)
  - image_size.ldt (provide basic symbols giving location and size of image)

- user.ldt (an empty file designed to be overridden by users that is included in linkscript, memory and library templates)
- user_library.ldt (an empty file designed to be overridden by users that is included in library only)

memory.ldt (the memory map)

- user.ldt (an empty file designed to be overridden by users that is included in linkscript, memory and library templates)
- user_memory.ldt (an empty file designed to be overridden by users that is included in memory only)

Linker script search paths

Whenever a linker script template is used, LPCXpresso will search in the following locations, in the order shown:

- *project/linkscripts*
- the searchPath global variable
  - The searchPath can be set in a script by using the syntax <#global searchPath="c:/windows/path;d:/another/windows/path">

each directory to search is separated by a semicolon ';'

- *mcuxpresso_install_dir/ide/Data/Linkscripts*
  - linker templates can be placed in this directory to override the default templates for an entire installation
- MCUXpresso IDE internally provided templates (not directly visible to users)

Thus, a project can simply override any template by simply creating a linkscript directory within the project and placing the appropriate template in there. Using the special syntax "super@" an overridden template can reference a file from the next level of the search path

e.g. *<#include "super@user.ldt">*

Linker script templates

Copies of the default linker script templates used within MCUXpresso IDE can be found in the Wizards/linker directory within the MCUXpresso IDE install.

Predefined variables (macros)

List (sequence) variables (used in *#list*)

- libraries[]
  - list of the libraries to be included in the "lib" script
  - for example (Redlib nohost)

```
libraries[0]=libcr_c.a
libraries[1]=libcr_eabihelpers.a
```

- configMemory[]
  - list of each memory region defined in the memory map for the project. Each entry has the following fields defined
    - name – the name of the memory region
    - alias – the alias of the memory region
    - location – the base address of the memory
    - size – the size of the memory region

    - sizek – the printable size of the memory region in k or M
    - mcuPattern
    - defaultRAM – boolean indicating if this is the default RAM region
    - defaultFlash – boolean indication if this is the default Flash region
    - RAM – boolean indicating if this is RAM
    - Flash – boolean indicating if this is Flash
  - for example

```
configMemory[0]= name=MFlashA512 alias=Flash location=0x1a000000
size=0x80000 sizek=512K bytes mcuPattern=Flash flash=true RAM=false
defaultFlash=true defaultRAM=false
```

```
configMemory[2]= name=RamLoc32 alias=RAM location=0x10000000
size=0x8000 sizek=32K bytes mcuPattern=RAM flash=false RAM=true
defaultFlash=false defaultRAM=true
```

- Slaves[]
  - list of the Slaves in a Multicore project. This variable is only defined in Multicore projects. Each entry has the following fields defined
    - name – name of the Slave
    - enabled – boolean indicating if this Slave is enabled
    - objPath – path to the object file for the Slave image
    - linkSection – name of the section this Slave is to be linked in
    - runtimeSection
    - textSection – name of the text section
    - textSectionNormalized – normalized name of the text section
    - dataStartSymbol – name of the Symbol defining the start of the data
    - dataEndSymbol – name of the Symbol defining the end of the data
  - for example

```
slaves[0] = name=M0APP objectPath=${workspace_loc:/MCB4357_Blinky_DualM0/Debug
/MCB4357_Blinky_DualM0.axf.o}linkSection=Flash2 runtimeSection= textSection=
.core_m0app textSectionNormalized=_core_m0appdata StartSymbol=__start_data
dataEndSymbol=__end_data enabled=true;</notextile>
```

Simple variables:

- CODE – name of the memory region to place the default code (text) section
- CRP_ADDRESS – location of the Code Read Protect value
- DATA – name of the memory region to place the default data section
- LINK_TO_RAM – value of the “Link to RAM” linker option
- STACK_OFFSET – value of the Stack Offset linker option
- FLASHn – defined for each FLASH memory
- RAMn – defined for each RAM memory
- basename – internal name of the process
- bss_align – alignment for .bss sections
- buildConfig – the name of the configuration being built
- chipFamily – the chip family
- chipName – name of the target chip
- data_align – alignment for .data section
- date – date string
- heap_symbol – name of the symbol used to define the heap

- isCppProject – boolean indicating if this is a C++ project
- isSlave – boolean indicating if this target is a slave – true iff is a slave core in a multicore system
- library_include – name of the library include file
- libtype – C library type
- memory_include – name of the memory include file
- mtb_supported – boolean indicating if mtb is supported for this target
- numCores – number of cores in this target
- procName – the name of the target processor
- project – the name of the project
- script – name of the script file
- slaveName – is the name of the slave (only present for slaves)
- stack_section – the name of the section where the stack is to be placed
- start_symbol – the name of the start symbol (entry point)
- scriptType – the type of script being generated (one of “script”, “memory”, or “library”)
- text_align – alignment for .text section
- version – product version string
- workspace_loc – workspace directory
- year – the year (extracted from the date)

Extended variables

Two ‘extended’ variables are available:

environment

- The environment variable makes the host Operating System environment variables available. For example, the Path variable is available as ${environment[“Path”]}. Note that environment variables are case sensitive.

systemProperties

- The Java system properties are available through the systemProperties variable. For example the “os.name” system property is available as ${systemProperties[“os.name”]}. Note that the system properties are case sensitive.

Outputting variables

A list of all predefined variables and their values can be output to the generated linker script by setting the Preference: *MCUXpresso IDE -> Default Tool settings ->* … and list predefined variables in the script

A list of extended variables and their values can be output to the generated linker script by creating a *linkscripts/user.ldt* file in the project with the content

```
<#assign listvarsext=true>
```

(This is likely to be used less often, hence the slightly longer winded method of specifying the option)

14.11 Freemarker Linker Script Template Examples

The use of Freemarker linker script templates allows more wide ranging changes to be made to the generated link script than is possible using the *cr_section_macros.h* macros. The following examples provide some examples of this.

14.11.1 Relocating code from FLASH to RAM

If you have specific functions in your code base that you wish to place into a particular block of RAM, then the simplest way to do this is to decorate the function definition using the macro __RAMFUNC described earlier in this chapter.

However once you want to relocate more than a few functions, or when you don't have direct access to the source code, this becomes impractical. In such case the use of Freemarker linker script templates will be a better approach. The following sections provide a number of such examples.

Relocating particular objects into RAM

In some cases, it may be required to relocate all of the functions (and rodata) from a given object file in your project into RAM. This can be achieved by providing three linker script template files into a linkscripts folder within your project. For example if it was required that all code/rodata from the files foo.c and bar.c were relocated into RAM, then this could be achieved using the following linker script templates:

```
main_text.ldt
 *(EXCLUDE_FILE(*foo.o *bar.o) .text*)
```

```
main_rodata.ldt
 *(EXCLUDE_FILE(*foo.o *bar.o) .rodata)
 *(EXCLUDE_FILE(*foo.o *bar.o) .rodata.*)
 *(EXCLUDE_FILE(*foo.o *bar.o) .constdata)
 *(EXCLUDE_FILE(*foo.o *bar.o) .constdata.*)
 . = ALIGN(${text_align});
```

```
main_data.ldt
 *foo.o(.text*)
 *foo.o(.rodata .rodata.* .constdata .constdata.*)
 *bar.o(.text*)
 *bar.o(.rodata .rodata.* .constdata .constdata.*)
 . = ALIGN(${text_align});
 *(.data*)
```

What each of these EXCLUDE_FILE lines (in main_text.ldt and main_rodata.ldt) is doing in pulling in all of the sections of a particular type (for example .text), except for the ones from the named object files. Then in main_data.ldt, we specify explicitly that the text and rodata sections should be pulled in from the named object files. Note that with the GNU linker, LD, the first match found in the final generated linker script is always used, which is why the EXCLUDE_FILE keyword is used in the first two template files.

Note: EXCLUDE_FILE only acts on the closest input section specified, which is why we have 4 separate EXCLUDE_FILE lines in the main_rodata.ldt file rather than just a single combined EXCLUDE_LINE.

Once you have built your project using the above linker script template files, then you can check the generated .ld file to see the actual linker script produced, together with the linker map file to confirm where the code and rodata have been placed.

Relocating particular libraries into RAM

In some cases, it may be required to relocate all of the functions (and rodata) from a given library in your project into RAM. One example of this might be if you are using a flashless LPC43xx

MCU with an external SPIFI Flash device being used to store and execute your main code from, but you need to actually update some data that you are also storing in the SPIFI Flash. In this case, the code used to update the SPIFI Flash cannot run from SPIFI Flash.

This can be achieved by providing three linker script template files into a linkscripts folder within your project. For example if it was required that all code/rodata from the library MYLIBRARYPROJ were relocated into RAM, then this could be achieved using the following linker script templates:

```
main_text.ldt
 *(EXCLUDE_FILE(*libMYLIBRARYPROJ.a:) .text*)
```

```
main_rodata.ldt
 *(EXCLUDE_FILE(*libMYLIBRARYPROJ.a:) .rodata)
 *(EXCLUDE_FILE(*libMYLIBRARYPROJ.a:) .rodata.*)
 *(EXCLUDE_FILE(*libMYLIBRARYPROJ.a:) .constdata)
 *(EXCLUDE_FILE(*libMYLIBRARYPROJ.a:) .constdata.*)
 . = ALIGN(${text_align});
```

```
main_data.ldt
 *libMYLIBRARYPROJ.a:(.text*)
 *libMYLIBRARYPROJ.a:(.rodata .rodata.* .constdata .constdata.*)
 . = ALIGN(${text_align});
 *(.data*)
```

Relocating majority of application into RAM

In some situations, you may wish to run the bulk of your application code from RAM – typically just leaving startup code and the vector table in Flash. This can be achieved by providing three linker script template files into a linkscripts folder within your project:

```
main_text.ldt
 *startup_*.o (.text.*)
 *(.text.main)
 *(.text.__main)
```

```
main_rodata.ldt
 *startup_*.o (.rodata .rodata.* .constdata .constdata.*)
 . = ALIGN(${text_align});
```

```
main_data.ldt
 *(.text*)
 *(.rodata .rodata.* .constdata .constdata.*)
 . = ALIGN(${text_align});
 *(.data*)
```

The above linker template scripts will cause the main body of the code to be relocated into the main (first) RAM bank of the target MCU, which by default will also contain data/bss, as well as the stack and heap.

If the MCU being targeted has more than one RAM bank, then the main body of the code could be relocated into another RAM bank instead. For example, if you wanted to relocate the code into the second RAM bank, then this could be done by providing the following data.ldt file instead of the main_data.ldt above:

```
data.ldt
 <#if memory.alias=="RAM2">
 *(.text*)
 *(.rodata .rodata.* .constdata .constdata.*)
 . = ALIGN(${text_align});
 </#if>
 *(.data.$${memory.alias}*)
 *(.data.$${memory.name}*)
```

Note: memory.alias value is taken from the Alias column of the Memory Configuration Editor.

14.11.2 Configuring projects to span multiple Flash devices

Most MCUs only have one bank of Flash memory. But with some parts more than one bank may be available – and in such cases, by default, the managed linker script mechanism will still place all of the application code and rodata (consts) into the first bank of Flash (as displayed in the Memory Configuration Editor)..

For example

- most of the LPC18 and LPC43xx parts containing internal Flash (such as LPC1857 and LPC4357) actually provide dual banks of Flash.
- some MCUs have the ability to access external Flash (typically SPIFI) as well as their built-in internal Flash (e.g. LPC18xx, LPC40xx, LPC43xx, LPC546xx).

The macros provided in the “cr_section_macros.h” header file provide some ability to control the placement of specific functions or rodata items into the second (or even third) bank of Flash. However the use of Freemarker linkers script templates allow this to be done in a much more powerful and flexible way.

One typical use case for this is a project which stores its main code and data in internal Flash, but additional rodata (for example graphics data for displaying on an LCD) in the external SPIFI Flash.

For instance, consider an example project where such rodata is all contained in a set of specific files, which we therefore want to place into the external Flash device. One very simple way to do this is to place such source files into a separate source folder within your project. You can then supply linker script templates to place the code and rodata from these files into the appropriate Flash.

For example, for a project using the LPC4337 with two internal Flash banks, plus external SPIFI Flash, if the source folder used for this purpose were called ‘spifidata’, then placing the following files into a ‘linkscripts’ directory within your project would have the desired effect:

```
text.ldt
 <#if memory.alias=="Flash3">
  *spifidata/*(.text*)
 </#if>
 *(.text_${memory.alias}*) /* for compatibility with previous releases */
 *(.text_${memory.name}*) /* for compatibility with previous releases */
 *(.text.$${memory.alias}*)
 *(.text.$${memory.name}*)
```

```
rodata.ldt
 <#if memory.alias=="Flash3">
 *spifidata/*(.rodata*)
```

```
</#if>
*(rodata.$${memory.alias}*)
*(rodata.$${memory.name}*)
```

Note: the check of the memory.alias being Flash3 is to prevent the code/rodata items from ending up in the BankB Flash bank (which is Flash2 by default).

14.12 Disabling Managed Linker Scripts

It is possible to disable the managed linker script mechanism if required and provide your own linker scripts, but this is not recommended for most users. In most circumstance, the facilities provided by the managed linker script mechanism, and its underlying Freemarker template mechanism should allow you to avoid the need for writing your own linker scripts. But if you do wish to do this, then untick the appropriate option at:

Properties -> C/C++ Build -> Settings -> MCU Linker -> Managed Linker Script

And then in the field *Script Path* provide the name and path (relative to the current build directory) of your own, manually maintained linker script.

In such cases you can either create your own linker script from scratch, or you can use the managed linker scripts as a starting point. One very important point though is that you are advised not to simply modify the managed linker scripts in place, but instead to copy them to another location and modify them there. This will prevent any chance of the tools accidentally overwriting them if at some point in the future you turn the managed make script mechanism back on.

Note: if your linker script includes additional files (as the managed linker scripts do), then you will also need to include the relative path information in the include inside the top level script file.

For more details of writing your own linker scripts, please see the GNU Llinker (ld) documentation:

Help -> Help Contents -> Tools (Compilers, Debugger, Utilities) -> GNU Linker

There is also a good introduction to linker scripts available in Building Bare-Metal ARM Systems with GNU: Part 3 at:

http://www.embedded.com/design/mcus-processors-and-socs/4026080/Building-Bare-Metal-ARM-Systems-with-GNU-Part-3

15. Multicore Projects

15.1 Introduction

Multicore MCUs can be designed in many ways, however within MCUXpresso IDE there is an underlying expectation that one core (the Master) will control the execution (or at least the startup) of code running on other (Slave) core(s).

Multicore application projects as described below consists of two (or more) linked projects – one project containing the Master code and the other project(s) containing the Slave code. The ‘Master’ project contains a link to the ‘Slave’ project which will cause the output image from the ‘Slave’ to be included into the ‘Master’ image when the Master project is built. Note, building the Master project will trigger the Slave project to be built first.

After a power-on or Reset, the Master core boots and is then responsible for booting the Slave core. However, this relationship only applies to the booting process; after boot, your application may treat either of the cores as the Master or the Slave.

For this concept to work, the memory configurations of these related projects must be carefully managed to avoid unintended overlap or contention. One way this can be achieved is by linking the Slave application to execute entirely from a RAM bank unused by the Master. Our automatic linkerscript generation will then locate the Slave code within the Master’s generated image, this will then be relocated to the correct RAM location by the Master projects initialisation code at run time.

In practice, the master projects memory configuration will be the same as for a single core project, where as the Slave projects memory configuration will be set to use a ‘spare’ or dedicated Slave RAM region. In addition, a shared region may be used for communication between the CPUs

Note: MCUs supporting dedicated Flash regions for each core can also be supported by this scheme, in such a case the Slave project would be linked to the Slave’s Flash location.

To complete the story ... the Master project is debugged first, this will lead to the combined image being programmed into Flash and the Master code executed. The Master’s initialisation code will (in addition to other things) copy the Slave code into RAM (if appropriate) and then stop on Main. When the Slave project is debugged, the launch configuration will automatically be set to ‘Attach’ by the IDE since there is no need for this code to be programmed/downloaded. When the Master application is resumed it will release the Slave and both projects can be debugged as required.

Important Note: Multicore MCUs may offer significant flexibility in how they can be used. The mechanism described above and also used in example projects is not necessarily the only way (or even the best way) for a user application(s)/projects to be configured. However, it has been chosen as the simplest and safest way to demonstrate the concepts and issues involved.

The MCUXpresso IDE allows for the easy creation of “linked” projects that support the targeting of LPC541xx Multicore MCUs

The rest of this chapter will describe the use of the LPC5411x multicore MCU, however the concepts discussed will be the same (or similar) for other multicore MCUs such as the LPC43xx and LPC5410x.

15.2 Creating a Master / Slave project pair (using an SDK)

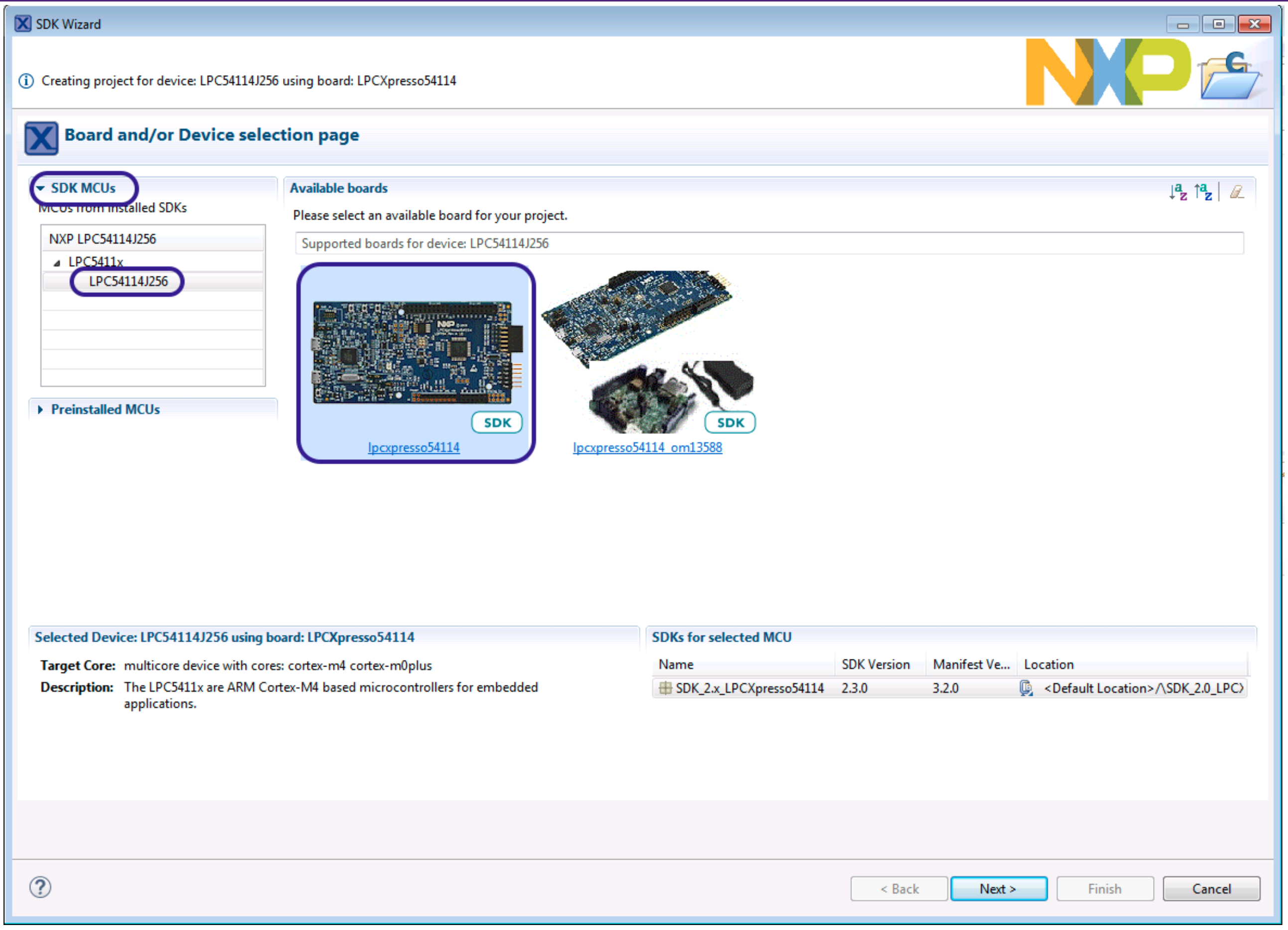
The example described below is base around the LPC5411x multicore MCU using the LPCXpresso54114 SDK.

Note: Be sure to have installed the LPCXpresso54114 SDK into MCUXpresso IDE to follow this example.

15.2.1 Creating the M0 Slave project

As discussed above, the Master projects configuration will need to reference the Slave project, therefore the Slave project should be created first.

Launch the New Project Wizard and select the LPCXpresso54114 SDK board. Entering 54114 into the boards filter will reduce the number of boards to help selection, then click Next.


SDK Wizard

Creating project for device: LPC54114J256 using board: LPCXpresso54114

Board and/or Device selection page

SDK MCUs

MCUs from installed SDKs

NXP LPC54114J256
- LPC5411x
  - LPC54114J256

Preinstalled MCUs

Available boards

Please select an available board for your project.

Supported boards for device: LPC54114J256

Selected Device: LPC54114J256 using board: LPCXpresso54114

Target Core: multicore device with cores: cortex-m4 cortex-m0plus

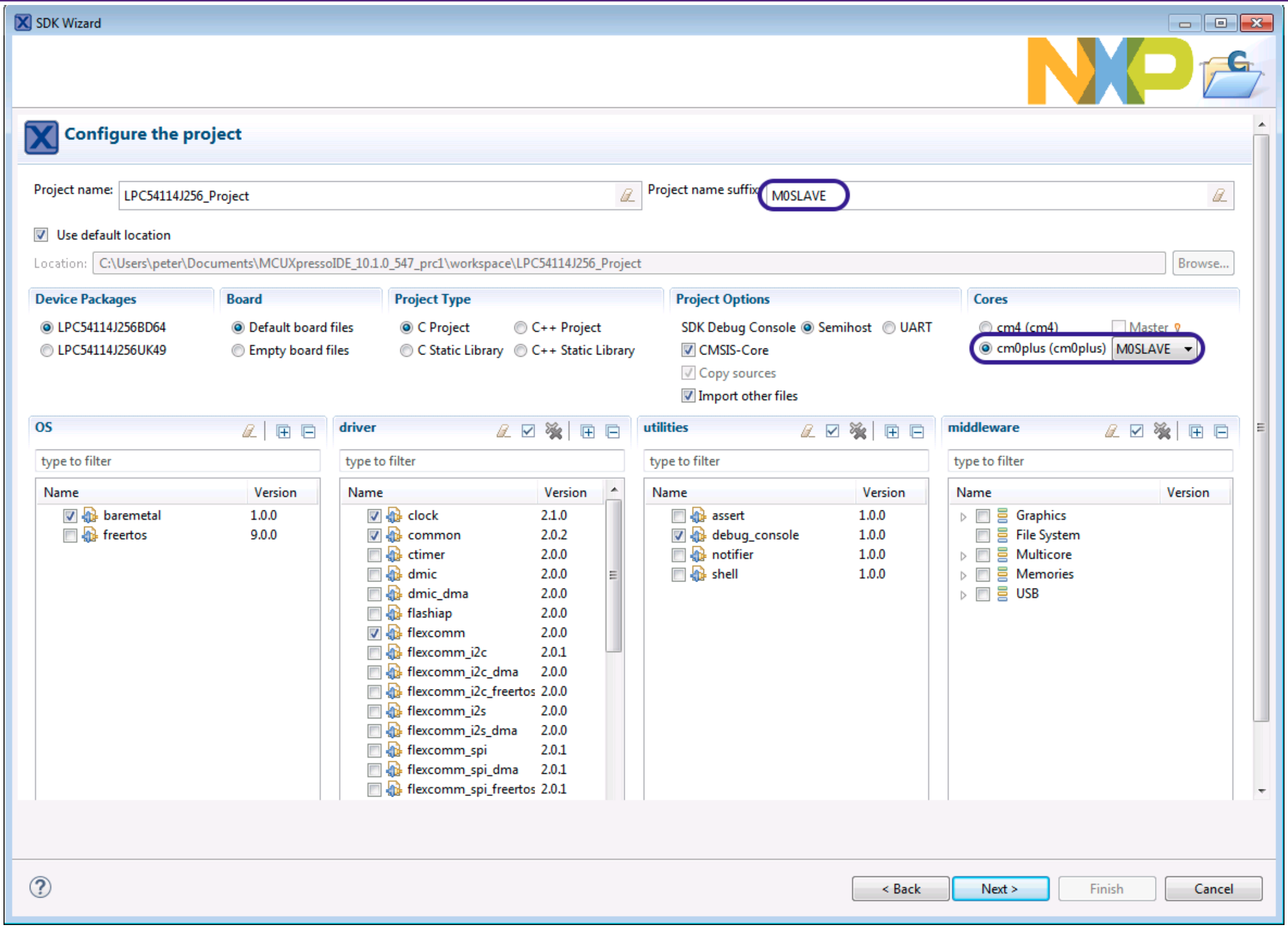
Description: The LPC5411x are ARM Cortex-M4 based microcontrollers for embedded applications.

SDKs for selected MCU

Name	SDK Version	Manifest Ve...	Location
SDK_2.x_LPCXpresso54114	2.3.0	3.2.0	<Default Location>/\SDK_2.0_LPC>

Figure 15.1. New Project Wizard SDK MultiCore M0

From the next wizard page, select the cm0plus Core, and see that the M0SLAVE is selected in the core options. Also note that the Project will automatically be given the suffix M0SLAVE. Drivers, utilities etc. can be selected at this stage for the Slave project if required.



Project name: LPC54114J256_Project
Project name suffix: M0SLAVE

[✓] Use default location
Location: C:\Users\peter\Documents\MCUXpressoIDE_10.1.0_547_prc1\workspace\LPC54114J256_Project

Device Packages: (●) LPC54114J256BD64 ( ) LPC54114J256UK49

Board: (●) Default board files ( ) Empty board files

Project Type: (●) C Project ( ) C++ Project ( ) C Static Library ( ) C++ Static Library

Project Options: SDK Debug Console (●) Semihost ( ) UART; [✓] CMSIS-Core; [✓] Copy sources; [✓] Import other files

Cores: ( ) cm4 (cm4) [ ] Master; (●) cm0plus (cm0plus) M0SLAVE

OS

Name	Version
[✓] baremetal	1.0.0
[ ] freertos	9.0.0

driver

Name	Version
[✓] clock	2.1.0
[✓] common	2.0.2
[ ] ctimer	2.0.0
[ ] dmic	2.0.0
[ ] dmic_dma	2.0.0
[ ] flashiap	2.0.0
[✓] flexcomm	2.0.0
[ ] flexcomm_i2c	2.0.1
[ ] flexcomm_i2c_dma	2.0.0
[ ] flexcomm_i2c_freertos	2.0.0
[ ] flexcomm_i2s	2.0.0
[ ] flexcomm_i2s_dma	2.0.0
[ ] flexcomm_spi	2.0.1
[ ] flexcomm_spi_dma	2.0.1
[ ] flexcomm_spi_freertos	2.0.1

utilities

Name	Version
[ ] assert	1.0.0
[✓] debug_console	1.0.0
[ ] notifier	1.0.0
[ ] shell	1.0.0

middleware

Name	Version
[ ] Graphics	
[ ] File System	
[ ] Multicore	
[ ] Memories	
[ ] USB	

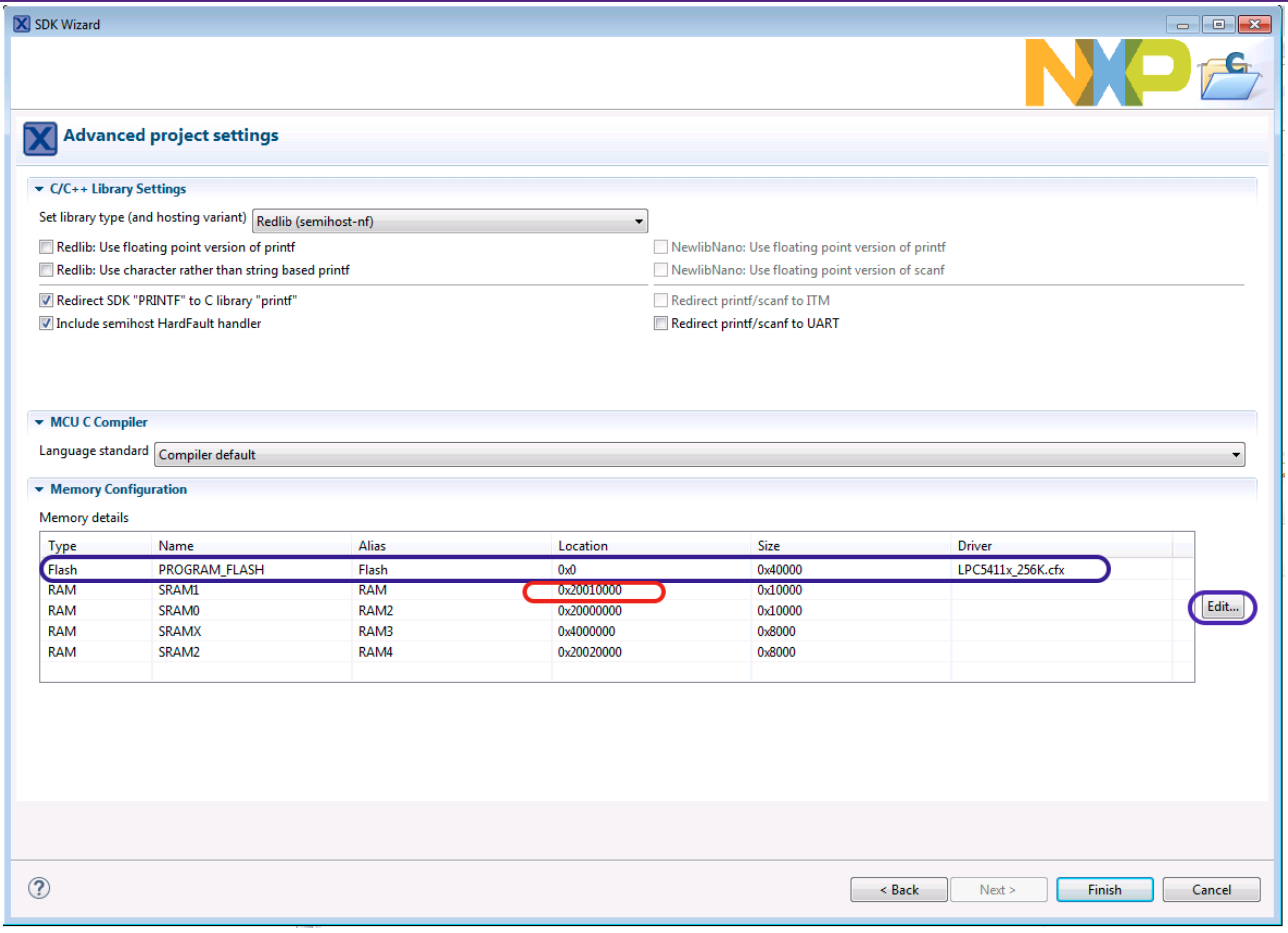
Figure 15.2. New Project Wizard SDK M0 Slave

Next, the M0 Slave memory configuration needs to be set.

Note: the MCUXpresso IDEs managed linker script mechanism will default to link code to the first Flash region in this view (if one exists) and use the first RAM region for data, heap and stack..

To force our project to link to a private area of RAM, we must ensure that the Flash region is removed and the chosen RAM bank is at the top of the list of memory regions. Note here that the SDK we are using has presented the RAM regions in a non sequential order. In this example we will configure the memory so that the M0 Slave project links to the RAM region starting at address 0x20010000 (the first region).

Click Edit... to launch the Memory configuration editor.



Set library type (and hosting variant): Redlib (semihost-nf)

- [ ] Redlib: Use floating point version of printf
- [ ] Redlib: Use character rather than string based printf
- [✓] Redirect SDK "PRINTF" to C library "printf"
- [✓] Include semihost HardFault handler
- [ ] NewlibNano: Use floating point version of printf
- [ ] NewlibNano: Use floating point version of scanf
- [ ] Redirect printf/scanf to ITM
- [ ] Redirect printf/scanf to UART

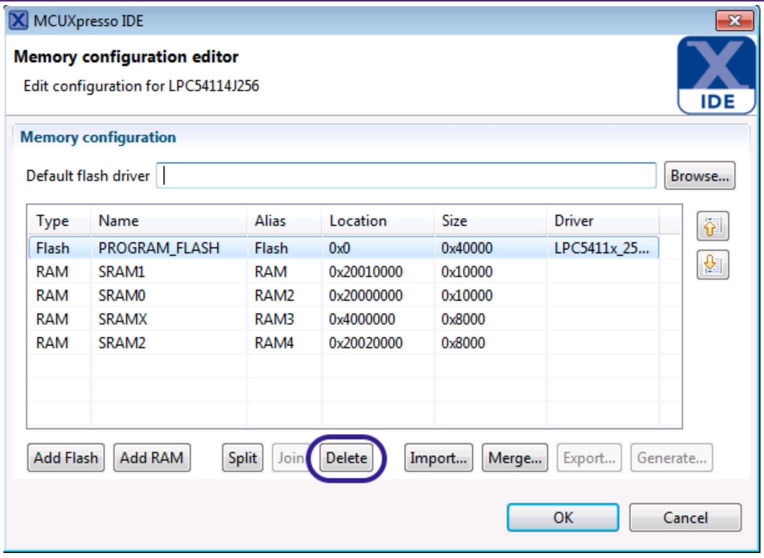
Language standard: Compiler default

Memory details

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x40000	LPC5411x_256K.cfx
RAM	SRAM1	RAM	0x20010000	0x10000	
RAM	SRAM0	RAM2	0x20000000	0x10000	
RAM	SRAMX	RAM3	0x4000000	0x8000	
RAM	SRAM2	RAM4	0x20020000	0x8000	

Figure 15.3. New Project Wizard SDK M0 Slave Memory

In the Memory Config Editor, select the PROGRAM_FLASH and click Delete to remove the region. Ensure that the top RAM region has the base address (location) 0x20010000, then click OK.



Memory configuration editor

Edit configuration for LPC54114J256

Memory configuration

Default flash driver

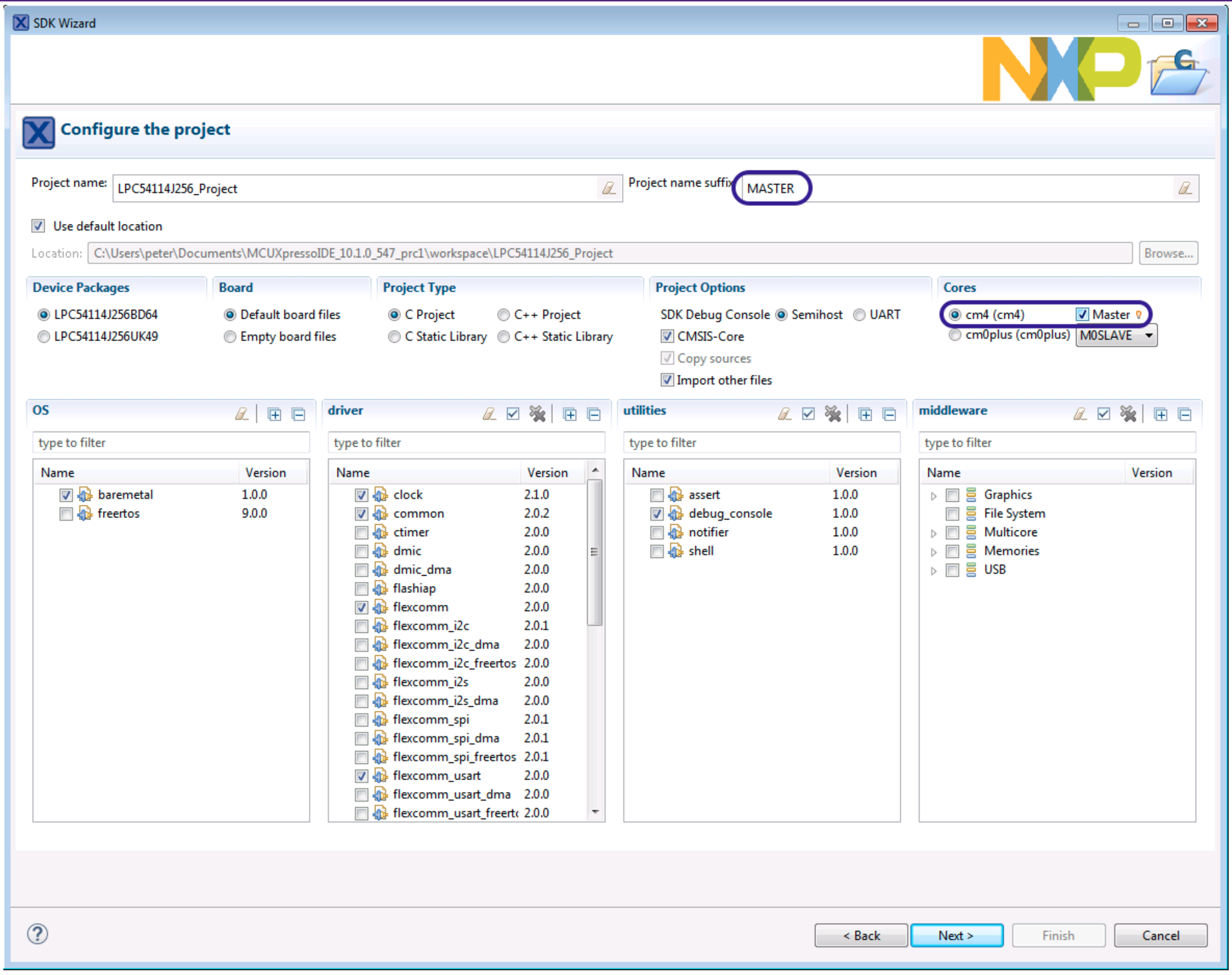
Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x40000	LPC5411x_25...
RAM	SRAM1	RAM	0x20010000	0x10000	
RAM	SRAM0	RAM2	0x20000000	0x10000	
RAM	SRAMX	RAM3	0x4000000	0x8000	
RAM	SRAM2	RAM4	0x20020000	0x8000	

Figure 15.4. New Project Wizard SDK M0 Slave Memory Edit

Finally, click Finish to complete the creation of the Slave project.

15.2.2 Creating the M4 Master project

To create the Master project, launch the New Project Wizard and again select the LPCXpresso54114 SDK board and click Next. This time, select the cm4 Core, and click the MASTER check box, this configures the wizard to create a Multicore project. Note that the Project will automatically be given the suffix MASTER. Drivers, utilities etc. can be selected at this stage for the Master project if required.


SDK Wizard

Configure the project

Project name: LPC54114J256_Project

Project name suffix: MASTER

☑ Use default location

Location: C:\Users\peter\Documents\MCUXpressoIDE_10.1.0_547_prc1\workspace\LPC54114J256_Project

Device Packages: [✓ LPC54114J256BD64] LPC54114J256UK49

Board: [✓ Default board files] Empty board files

Project Type: [✓ C Project] C++ Project C Static Library C++ Static Library

Project Options: SDK Debug Console [✓ Semihost] UART; ☑ CMSIS-Core; ☑ Copy sources; ☑ Import other files

Cores: [✓ cm4 (cm4)] ☑ Master; cm0plus (cm0plus) M0SLAVE

OS

	Name	Version
☑	baremetal	1.0.0
☐	freertos	9.0.0

driver

	Name	Version
☑	clock	2.1.0
☑	common	2.0.2
☐	ctimer	2.0.0
☐	dmic	2.0.0
☐	dmic_dma	2.0.0
☐	flashiap	2.0.0
☑	flexcomm	2.0.0
☐	flexcomm_i2c	2.0.1
☐	flexcomm_i2c_dma	2.0.0
☐	flexcomm_i2c_freertos	2.0.0
☐	flexcomm_i2s	2.0.0
☐	flexcomm_i2s_dma	2.0.0
☐	flexcomm_spi	2.0.1
☐	flexcomm_spi_dma	2.0.1
☐	flexcomm_spi_freertos	2.0.1
☑	flexcomm_usart	2.0.0
☐	flexcomm_usart_dma	2.0.0
☐	flexcomm_usart_freert	2.0.0

utilities

	Name	Version
☐	assert	1.0.0
☑	debug_console	1.0.0
☐	notifier	1.0.0
☐	shell	1.0.0

middleware

	Name	Version
☐	Graphics	
☐	File System	
☐	Multicore	
☐	Memories	
☐	USB	

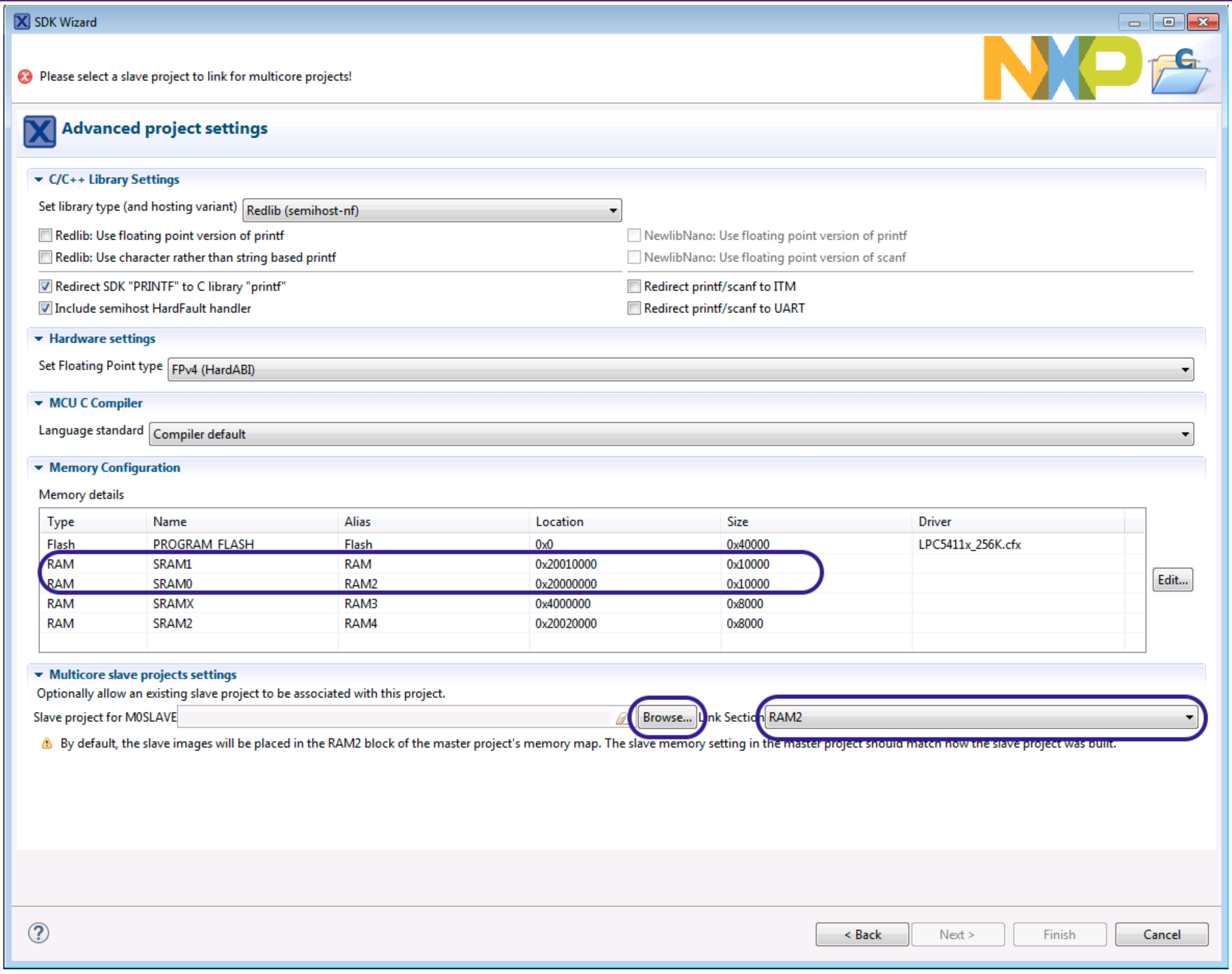
Figure 15.5. New Project Wizard SDK M4 Master

Next, the M4 Master memory configuration needs to be set. Typically we might leave the memory setting unaltered, however the SDK we are using presents the RAM regions in a non sequential way. In this example we wish to select the RAM region at 0x20000000 for the Master projects data and the Flash at 0x0 for the Master projects code (and also a copy of the Slave projects code)

Note: the MCUXpresso IDEs managed linker script mechanism will default to link code to the first Flash region in this view (if one exists) and use the first RAM region for data, heap and stack.

To adjust the memory layout, clicc Edit ... then the second RAM region (at location 0x20000000) and click the ‘Up’ arrow to move this to the top of the RAM regions.

Once this has been done, click ‘OK’.



Please select a slave project to link for multicore projects!

Advanced project settings

C/C++ Library Settings

Set library type (and hosting variant) Redlib (semihost-nf)

- [ ] Redlib: Use floating point version of printf
- [ ] Redlib: Use character rather than string based printf
- [ ] NewlibNano: Use floating point version of printf
- [ ] NewlibNano: Use floating point version of scanf
- [✓] Redirect SDK "PRINTF" to C library "printf"
- [✓] Include semihost HardFault handler
- [ ] Redirect printf/scanf to ITM
- [ ] Redirect printf/scanf to UART

Hardware settings

Set Floating Point type FPv4 (HardABI)

MCU C Compiler

Language standard Compiler default

Memory Configuration

Memory details

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x40000	LPC5411x_256K.cfx
RAM	SRAM1	RAM	0x20010000	0x10000	
RAM	SRAM0	RAM2	0x20000000	0x10000	
RAM	SRAMX	RAM3	0x4000000	0x8000	
RAM	SRAM2	RAM4	0x20020000	0x8000	

Multicore slave projects settings

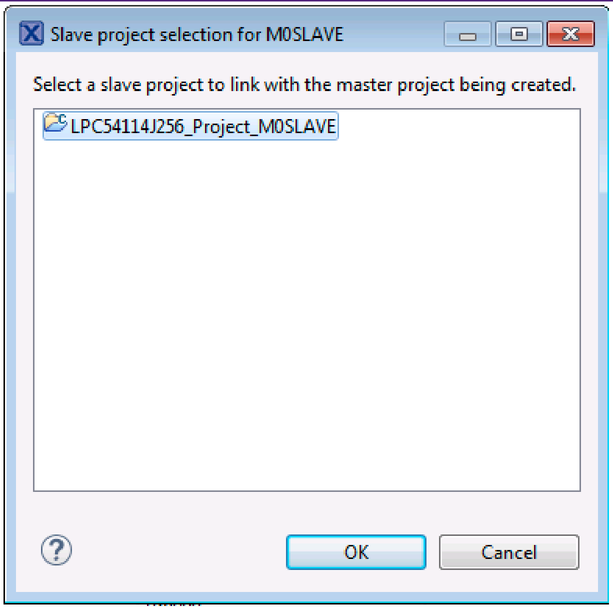
Optionally allow an existing slave project to be associated with this project.

Slave project for M0SLAVE Link Section RAM2

By default, the slave images will be placed in the RAM2 block of the master project's memory map. The slave memory setting in the master project should match how the slave project was built.

Figure 15.6. New Project Wizard SDK M4 Master Memory

Then click Browse to select the Slave project within the Workspace. Select the previously created Slave project and click ‘OK’.


Slave project selection for M0SLAVE

Select a slave project to link with the master project being created.

LPC54114J256_Project_M0SLAVE

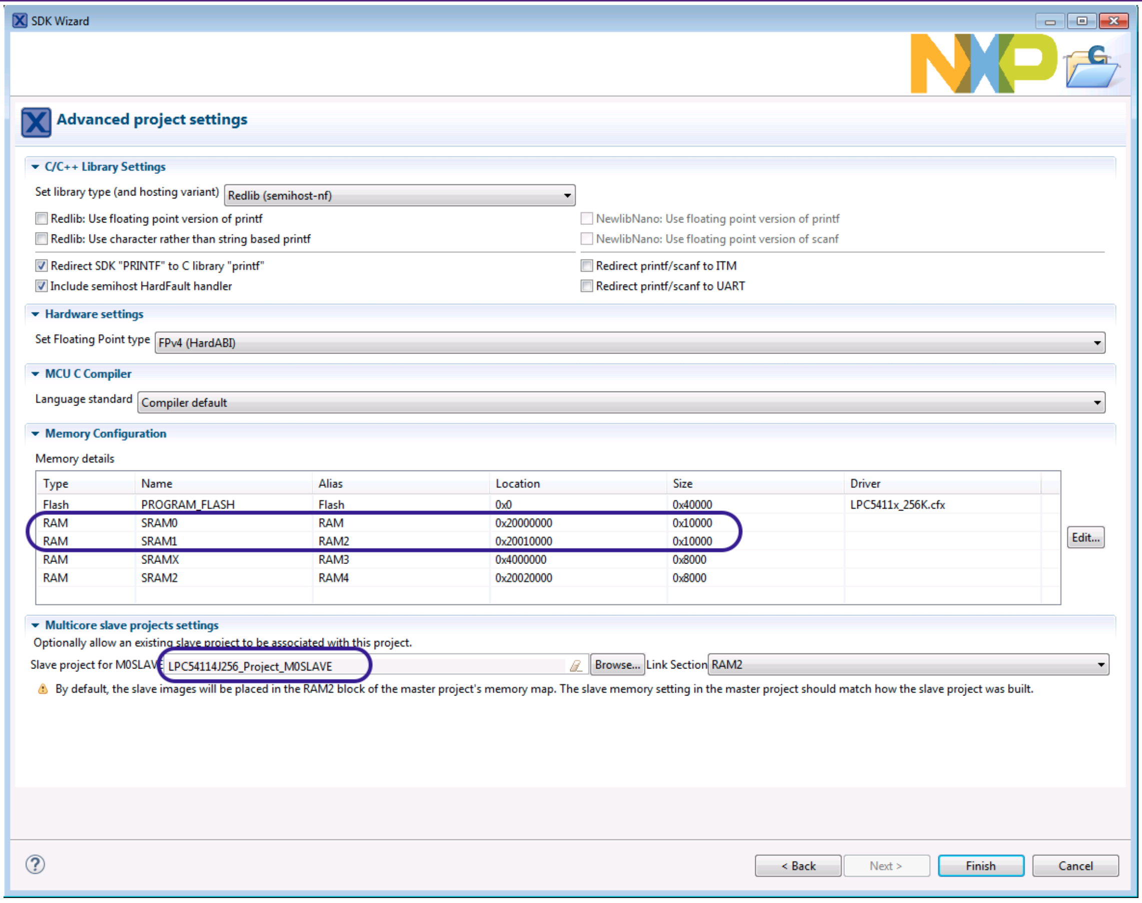
Figure 15.7. New Project Wizard SDK M4 Master Slave Selection

Also ensure the Link Section name (default of RAM2) selects a Master memory region that matches the linked address of the Slave project. In this case RAM2 should correspond to the address 0x2001000. If required, other memory regions can be selected here but please note: the first Flash Region, and the first RAM Region are not included in the drop down list because it is assumed that these will be used for the Master Project. If required, this setting can be changed later from:

Project Properties -> C/C++ Build -> Settings -> Multicore

Where all of the memory regions are available for selection.

Below we can see the edited project settings for the Master project.



SDK Wizard

Advanced project settings

C/C++ Library Settings

Set library type (and hosting variant): Redlib (semihost-nf)

- [ ] Redlib: Use floating point version of printf
- [ ] Redlib: Use character rather than string based printf
- [ ] NewlibNano: Use floating point version of printf
- [ ] NewlibNano: Use floating point version of scanf
- [x] Redirect SDK "PRINTF" to C library "printf"
- [x] Include semihost HardFault handler
- [ ] Redirect printf/scanf to ITM
- [ ] Redirect printf/scanf to UART

Hardware settings

Set Floating Point type: FPv4 (HardABI)

MCU C Compiler

Language standard: Compiler default

Memory Configuration

Memory details

Type	Name	Alias	Location	Size	Driver
Flash	PROGRAM_FLASH	Flash	0x0	0x40000	LPC5411x_256K.cfx
RAM	SRAM0	RAM	0x20000000	0x10000	
RAM	SRAM1	RAM2	0x20010000	0x10000	
RAM	SRAMX	RAM3	0x4000000	0x8000	
RAM	SRAM2	RAM4	0x20020000	0x8000	

Multicore slave projects settings

Optionally allow an existing slave project to be associated with this project.

Slave project for M0SLAVE: LPC54114J256_Project_M0SLAVE   Link Section: RAM2

By default, the slave images will be placed in the RAM2 block of the master project's memory map. The slave memory setting in the master project should match how the slave project was built.

Figure 15.8. New Project Wizard SDK M4 Master Project

Finally click Finish to generate the Master project.

15.3 Creating a Master / Slave project pair (using Preinstalled Part Support)

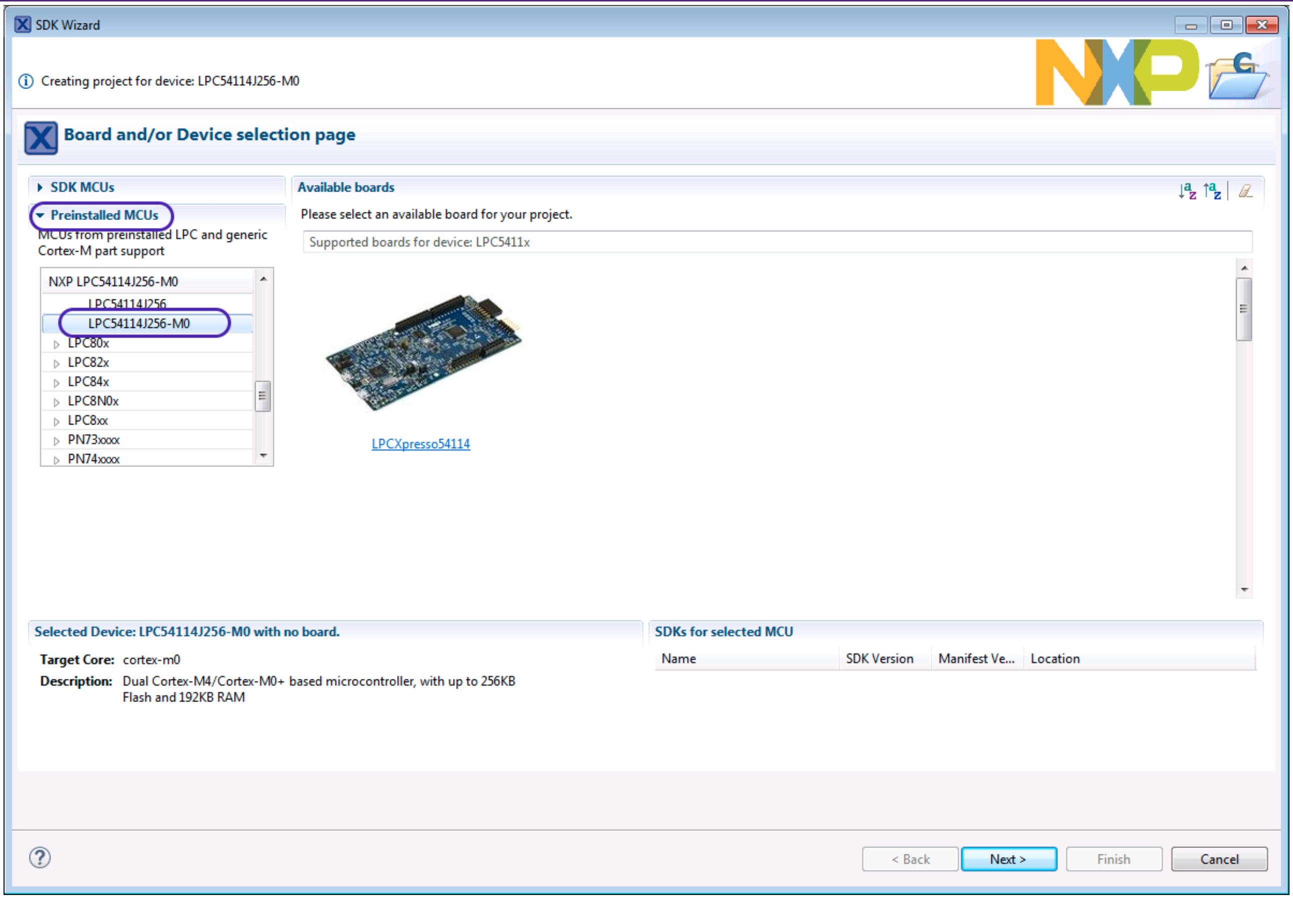
The example described below is base around the LPC5411x multicore MCU.

Note: It is recommended to create and build LPC541xx multicore projects which are linked to LPCOpen. Thus before you follow the below sequence, please ensure that you have imported the chip and (optionally) the board library projects (for both the M4 and M0+) from an LPCOpen package for the LPC5410x family or LPC5411x family (depending upon your target part).

15.3.1 Creating the M0 Slave project

As discussed above, the Master projects configuration will need to reference the Slave project, therefore the Slave project should be created first.

Launch the New Project Wizard and select the LPC54114-M0 from the Preinstalled MCUs.


SDK Wizard

Creating project for device: LPC54114J256-M0

Board and/or Device selection page

SDK MCUs

Preinstalled MCUs

MCUs from preinstalled LPC and generic Cortex-M part support

NXP LPC54114J256-M0
- LPC54114J256
- LPC54114J256-M0
- LPC80x
- LPC82x
- LPC84x
- LPC8N0x
- LPC8xx
- PN73xxxx
- PN74xxxx

Available boards

Please select an available board for your project.

Supported boards for device: LPC5411x

LPCXpresso54114

Selected Device: LPC54114J256-M0 with no board.

Target Core: cortex-m0

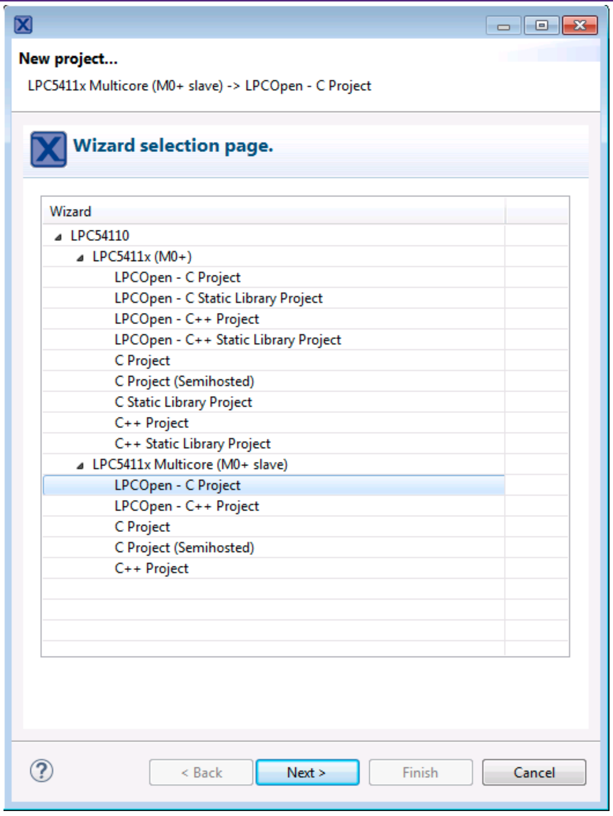
Description: Dual Cortex-M4/Cortex-M0+ based microcontroller, with up to 256KB Flash and 192KB RAM

SDKs for selected MCU

Name	SDK Version	Manifest Ve...	Location

Figure 15.9. New Project Wizard Preinstalled M0

Next, select a MultiCore M0 Slave project type, below we have selected an LPCOpen – C Project.



New project...

LPC5411x Multicore (M0+ slave) -> LPCOpen - C Project

Wizard selection page.

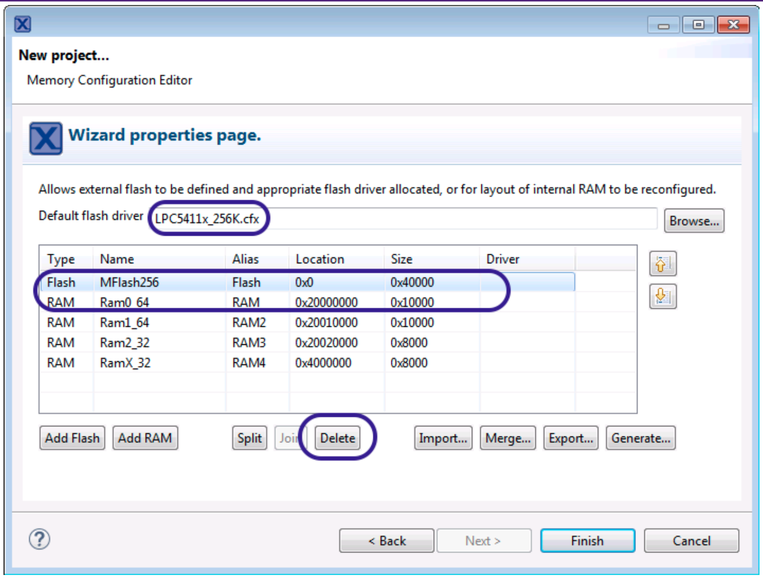
Wizard
- LPC54110
  - LPC5411x (M0+)
    - LPCOpen - C Project
    - LPCOpen - C Static Library Project
    - LPCOpen - C++ Project
    - LPCOpen - C++ Static Library Project
    - C Project
    - C Project (Semihosted)
    - C Static Library Project
    - C++ Project
    - C++ Static Library Project
  - LPC5411x Multicore (M0+ slave)
    - LPCOpen - C Project
    - LPCOpen - C++ Project
    - C Project
    - C Project (Semihosted)
    - C++ Project

Figure 15.10. New Project Wizard Preinstalled M0 C Project

Next, name the project, for example LPC54114_M0_Slave, then click next until the Memory Configuration page is reached. From here we can see the MCU memory regions.

Note: the MCUXpresso IDEs managed linker script mechanism will default to link code to the first Flash region in this view (if one exists) and use the first RAM region for data.

To force our project to link to a private area of RAM, simply delete the Flash and first RAM region (RAM0_64) from this view (since these will be use for the M4 Master project). To do this, just select the regions and click Delete. Since there will no longer be any Flash region, the default Flash driver can also be removed.



New project...

Memory Configuration Editor

Wizard properties page.

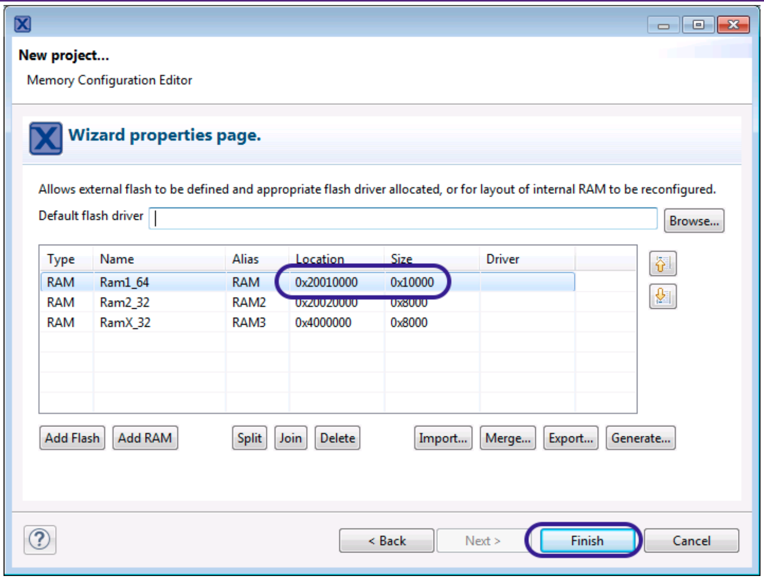
Allows external flash to be defined and appropriate flash driver allocated, or for layout of internal RAM to be reconfigured.

Default flash driver LPC5411x_256K.cfx

Type	Name	Alias	Location	Size	Driver
Flash	MFlash256	Flash	0x0	0x40000	
RAM	Ram0_64	RAM	0x20000000	0x10000	
RAM	Ram1_64	RAM2	0x20010000	0x10000	
RAM	Ram2_32	RAM3	0x20020000	0x8000	
RAM	RamX_32	RAM4	0x4000000	0x8000	

Figure 15.11. New Project Wizard Preinstalled M0 Memory

The memory setting should then look as below. In this case our Slave projects code and data will be linked to address 0x20010000 with the stack set to the top of this region.



New project...

Memory Configuration Editor

Wizard properties page.

Allows external flash to be defined and appropriate flash driver allocated, or for layout of internal RAM to be reconfigured.

Default flash driver

Type	Name	Alias	Location	Size	Driver
RAM	Ram1_64	RAM	0x20010000	0x10000	
RAM	Ram2_32	RAM2	0x20020000	0x8000	
RAM	RamX_32	RAM3	0x4000000	0x8000	

Figure 15.12. New Project Wizard Preinstalled M0 Memory edited

Now click Next -> Finish to complete the M0 Slave projects creation.

15.3.2 Creating the M4 Master project

To create the Master, Launch the New Project Wizard again and this time select the LPC54114 (M4) part and click Next. Select the matching ‘MultiCore M4 Master -> LPCOpen -C Project’ and click Next again. Now, name the new project, for example LPC54114_M4_Master and click next until the Multicore Project Setup page is reached.

Note: The wizard will present an identical memory configuration page, but on this occasion, no editing is required since the default Flash and RAM setting are

From here, click browse to select the previously created Slave project from the existing Workspace

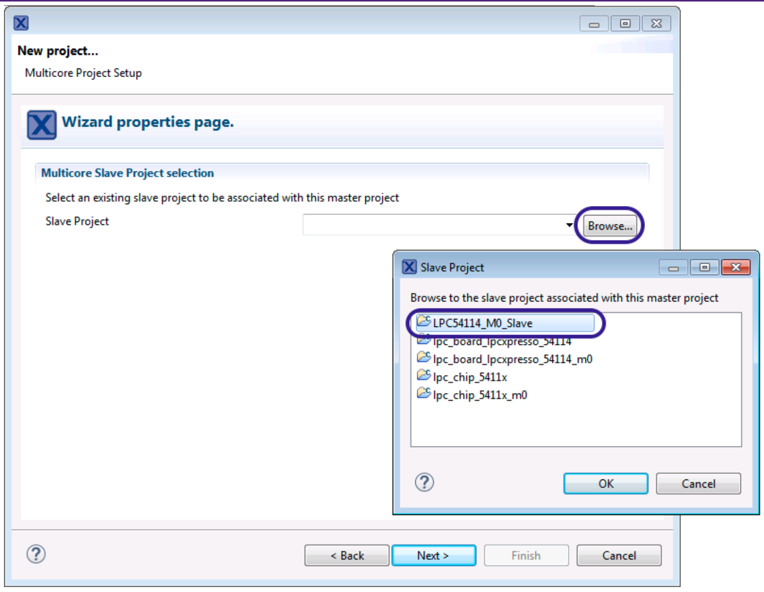

New project...
Multicore Project Setup
Wizard properties page.
Multicore Slave Project selection
Select an existing slave project to be associated with this master project
Slave Project
Browse...
Slave Project
Browse to the slave project associated with this master project
LPC54114_M0_Slave
lpc_board_lpcxpresso_54114
lpc_board_lpcxpresso_54114_m0
lpc_chip_5411x
lpc_chip_5411x_m0
OK
Cancel
< Back
Next >
Finish
Cancel

Figure 15.13. New Project Wizard Preinstalled M4 Select Slave

Now click Next -> Finish to complete the M4 Master projects creation.

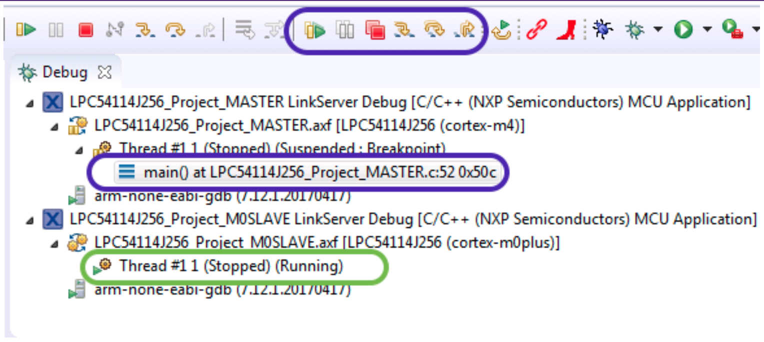
15.4 Debugging MultiCore Projects

The debug story for MultiCore MCUs can vary with their implementation and also the chosen debug solution.

Our MultiCore model as described above, assumes that the Master project will both copy the Slave MCUs code and data (into RAM) but also release the Slave from reset. Therefore the Master project should be run (debugged) first and typically run to main(). Once here, the instantiation of the Slave's code will be complete but the Slave will not have been released. On some MCUs, a debug connection can be made to the Slave before it has been released, but on others this will only be possible after they are released.

NOTE: Slave projects debug launch configurations may require user modification before a debug connection can be made. Please see the section Slave Project Debug [158]

In our example LPC54114, the Slave's debug connection can be made as soon as the Master reaches main(). The debug window will then look similar to that below.



Debug

LPC54114J256_Project_MASTER LinkServer Debug [C/C++ (NXP Semiconductors) MCU Application]
LPC54114J256_Project_MASTER.axf [LPC54114J256 (cortex-m4)]
Thread #1 1 (Stopped) (Suspended : Breakpoint)
main() at LPC54114J256_Project_MASTER.c:52 0x50c
arm-none-eabi-gdb (7.12.1.20170417)
LPC54114J256_Project_M0SLAVE LinkServer Debug [C/C++ (NXP Semiconductors) MCU Application]
LPC54114J256_Project_M0SLAVE.axf [LPC54114J256 (cortex-m0plus)]
Thread #1 1 (Stopped) (Running)
arm-none-eabi-gdb (7.12.1.20170417)

Figure 15.14. MultiCore Debug

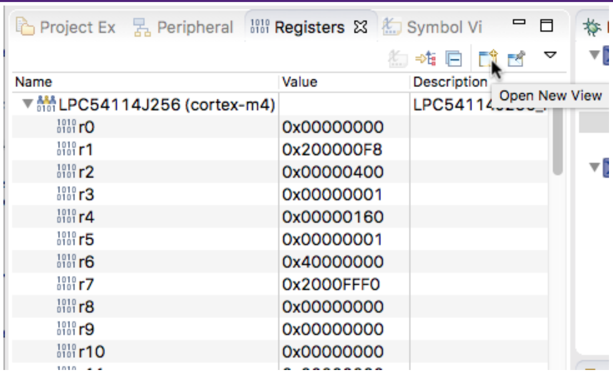
Note above: that the MultiCore debug controls have been highlighted, these controls differ from the standard controls in that they operate on all cores being debugged. Via these, the system to be stepped, run, paused, terminated etc.

In addition, the M4 Master debug stack (blue) is shown stopped at main, while the Slave stack (green) is waiting to be released by the Master; clicking between these stacks will change the IDE’s debug scope from one core to the other. The currently selected core will be the one used for displaying many of the debug related views, such as Registers and Locals.

15.4.1 Controlling Debug Views

It is also possible to create copies of many of the debug related views, and then lock each copy to a particular core (as described below).

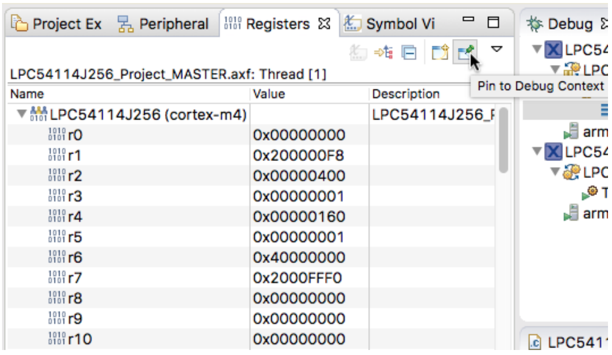
For example, to create two register views, one for the M4 and one for the M0+ ...First of all, use the “Open New view” button in the Registers view to create a second Registers view:



Name	Value	Description
LPC54114J256 (cortex-m4)		LPC54114
r0	0x00000000	
r1	0x200000F8	
r2	0x00000400	
r3	0x00000001	
r4	0x00000160	
r5	0x00000001	
r6	0x40000000	
r7	0x2000FFF0	
r8	0x00000000	
r9	0x00000000	
r10	0x00000000	

Figure 15.15. MultiCore Debug New View

Now pin the original view to the core currently selected in the Debug, using the “Pin to Debug context” button :

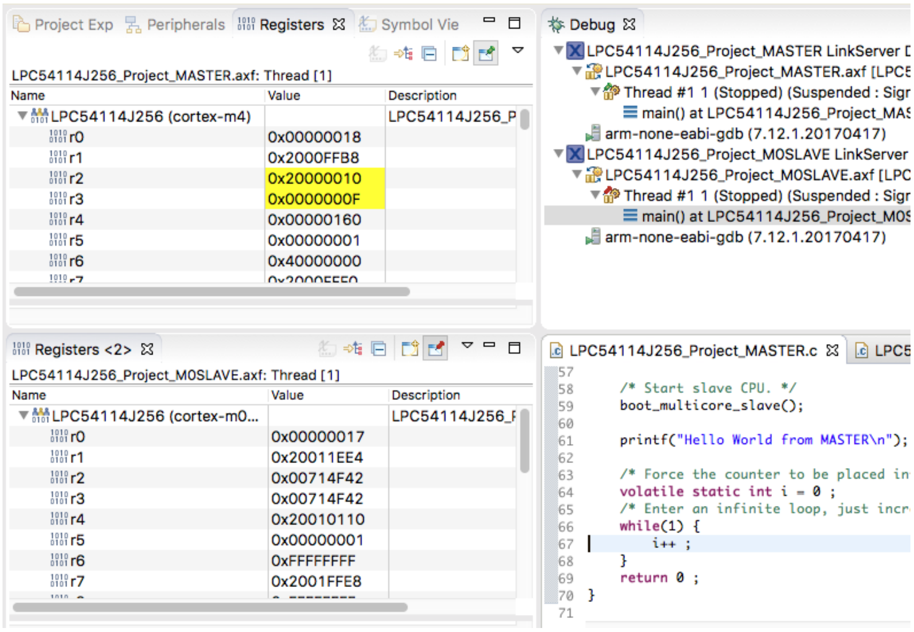


LPC54114J256_Project_MASTER.axf: Thread [1]

Name	Value	Description
LPC54114J256 (cortex-m4)		LPC54114J256_F
r0	0x00000000	
r1	0x200000F8	
r2	0x00000400	
r3	0x00000001	
r4	0x00000160	
r5	0x00000001	
r6	0x40000000	
r7	0x2000FFF0	
r8	0x00000000	
r9	0x00000000	
r10	0x00000000	

Figure 15.16. MultiCore Debug Pin View

Now select the other core in the Debug view, and go to the second Register view. Use this view's "Pin to Debug Context" button to lock this second Registers view to the selected core:



LPC54114J256_Project_MASTER.axf: Thread [1]

Name	Value	Description
LPC54114J256 (cortex-m4)		LPC54114J256_P
r0	0x00000018	
r1	0x2000FFB8	
r2	0x20000010	
r3	0x0000000F	
r4	0x00000160	
r5	0x00000001	
r6	0x40000000	

```
LPC54114J256_Project_MASTER LinkServer D
  LPC54114J256_Project_MASTER.axf [LPC5
    Thread #1 1 (Stopped) (Suspended : Sigr
      main() at LPC54114J256_Project_MAS
  arm-none-eabi-gdb (7.12.1.20170417)
LPC54114J256_Project_M0SLAVE LinkServer
  LPC54114J256_Project_M0SLAVE.axf [LPC
    Thread #1 1 (Stopped) (Suspended : Sigr
      main() at LPC54114J256_Project_M0S
  arm-none-eabi-gdb (7.12.1.20170417)
```

LPC54114J256_Project_M0SLAVE.axf: Thread [1]

Name	Value	Description
LPC54114J256 (cortex-m0...		LPC54114J256_F
r0	0x00000017	
r1	0x20011EE4	
r2	0x00714F42	
r3	0x00714F42	
r4	0x20010110	
r5	0x00000001	
r6	0xFFFFFFFF	
r7	0x2001FFE8	

```

    /* Start slave CPU. */
    boot_multicore_slave();

    printf("Hello World from MASTER\n");

    /* Force the counter to be placed in
    volatile static int i = 0 ;
    /* Enter an infinite loop, just incr
    while(1) {
        i++ ;
    }
    return 0 ;
}

```

Figure 15.17. MultiCore Debug Registers

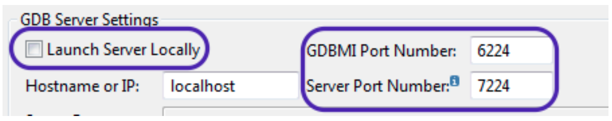
15.4.2 Slave Project Debug

Typically, the Master project will be debugged first in exactly the same way as a single CPU project. However the slave projects debug launch configuration may require special settings in order to establish a debug connection to the slave CPU.

MCUXpresso IDE will automatically configure the correct settings for LinkServer slave launch configurations however, for other debug solutions the slave debug settings may require modification, please see below:

- **Core Selection** - within a MultiCore MCU there will be more than one CPU (sometimes referred to as a device). The debug connection needs to be made to the appropriate internal CPU for both the Master and Slave Projects.
  - **LinkServer CMSIS-DAP Debug**: this process is **automatic** and hidden from the user. The selection details are stored within the projects build configuration folder(s) and will take the suffix .jtag or .swd

- **P&E Debug**: the Master CPU will be selected automatically for the Master project, however the launch configuration for the Slave project will need to be edited to select the required CPU. In our example this will be Core: M0
- **SEGGER Debug**: the Master CPU will be selected automatically for the Master project, however the launch configuration for the Slave project will need to be edited to select the required CPU. In our example this will be LPC54114J256_M0

- **Attach mode** for the Slave CPU – as described above, the debug connection to the slave(s) should be via an attach
  - **LinkServer CMSIS-DAP Debug**: this option is set **automatically** when the LinkServer debug launch configuration is created
  - **P&E Debug**: the launch configuration for the Slave project will need to be edited to force an attach operation
  - **SEGGER Debug**: the launch configuration for the Slave project will need to be edited to force an attach operation
- **Managing the Debug Server -** this is the low level interface between the debugger and target
  - **LinkServer CMSIS-DAP Debug**: the LinkServer launch configuration is **automatically** correctly configured when the debug connection is made
  - **P&E Debug**: the slave core is controlled via the master cores debug server, therefore the slave launch configuration must be edited to ensure that:
    - no Server is launched
    - the Port Numbers match those set in the Master Project’s launch configuration
    - 
      GDB Server Settings
      Launch Server Locally: unchecked
      Hostname or IP: localhost
      GDBMI Port Number: 6224
      Server Port Number: 7224
    - The Quickstart debug option cannot be used to make the slave debug connection. Instead, select *Run > Debug Configurations...* Next, select the appropriate slave launch configuration and then click *Debug*
    - See the P&E FAQ for more information http://www.pemicro.com/faqs/faq_view.cfm?ID=231
  - **SEGGER Debug**: the default settings can be used

15.5 MultiCore Projects additional information

15.5.1 Defines

A number of compiler defines are automatically created for LPC5410x projects to allow conditional compilation of certain blocks of code depending upon whether a specific project is configured to be a Slave, a Master or neither.

- __MULTICORE_MASTER
  - Defined automatically for a project that has been configured to be a Master project
- __MULTICORE_MASTER_SLAVE_M0SLAVE
  - Defined automatically for a project that has been configured to be a Master project and has had a Slave project associated with it (hence indicating to the Master project which cpu type the Slave project is for).
- __MULTICORE_M0SLAVE
  - Appropriate one defined automatically for a project that has been configured to be a Slave project
- __MULTICORE_NONE
  - Defined automatically for a project which has not been configured as either a Slave or Master project

Note: The multicore support within MCUXpresso IDE is highly flexible and provides functionality beyond that required for the LPC5411x family. Thus the symbols __MULTICORE_MASTER_SLAVE_M4SLAVE and __MULTICORE_M4SLAVE are also provided for completeness.

15.5.2 Slave Boot Code

boot_multicore_slave() is called by the Master project code created directly by the New project wizard to release the Slave core from sleep.

Note: the source files containing this function will be included in all LPC541xx projects, but will be conditionally compiled so that it is included only when required. This has been done to allow projects originally created, for example, as a Slave project, to be reconfigured (via the project properties – linker multicore tab) as a Master project.

15.5.3 Reset Handler code

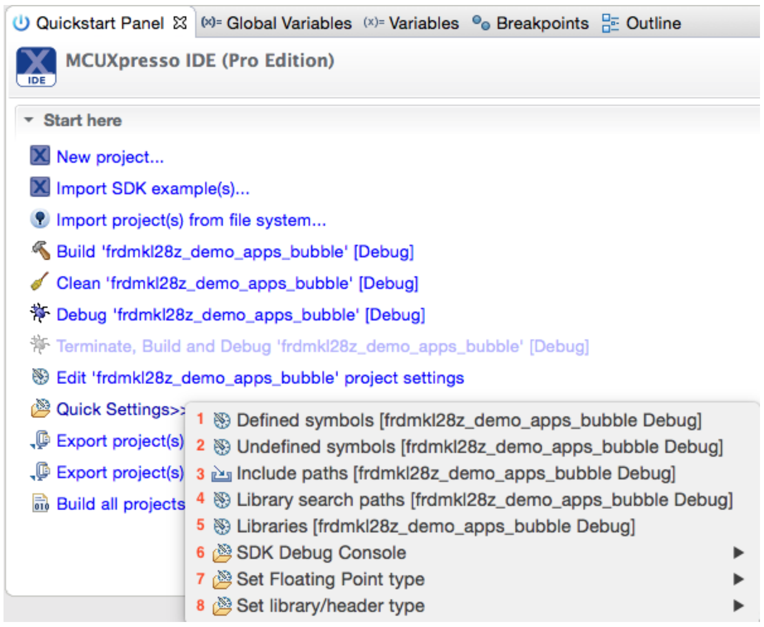
When configured as a Master project, the LPC541xx startup file will be built with additional (assembler) code at the beginning of the reset handler, ResetISR(), with the 'standard ' reset handler code moved to ResetISR2().

This additional code is required in order to allow correct booting of both the Master and Slave cores. It is written in assembler in order to force it to be 'Thumb1' code, and hence runnable by both cores.

16. Appendix – Additional Hints and Tips

16.1 Quick Settings

MCUXpresso IDE provides quick access to a range of project settings via the QuickStart Panel as shown below:


Quickstart Panel
Global Variables
Variables
Breakpoints
Outline
MCUXpresso IDE (Pro Edition)
Start here
New project...
Import SDK example(s)...
Import project(s) from file system...
Build 'frdmkl28z_demo_apps_bubble' [Debug]
Clean 'frdmkl28z_demo_apps_bubble' [Debug]
Debug 'frdmkl28z_demo_apps_bubble' [Debug]
Terminate, Build and Debug 'frdmkl28z_demo_apps_bubble' [Debug]
Edit 'frdmkl28z_demo_apps_bubble' project settings
Quick Settings>>
Export project(s)
Export project(s)
Build all projects
1 Defined symbols [frdmkl28z_demo_apps_bubble Debug]
2 Undefined symbols [frdmkl28z_demo_apps_bubble Debug]
3 Include paths [frdmkl28z_demo_apps_bubble Debug]
4 Library search paths [frdmkl28z_demo_apps_bubble Debug]
5 Libraries [frdmkl28z_demo_apps_bubble Debug]
6 SDK Debug Console
7 Set Floating Point type
8 Set library/header type

Note: These settings apply to the selected project's default build configuration only and simplify access to commonly used settings normally accessed from *Properties -> C/C++ Build -> Settings*

1. Defined symbols – select to edit the (-D) symbols
2. Undefined symbols – select to edit the (-U) symbols
3. Include paths – select to edit the (-I) the include paths
4. Library search paths – select to edit the (-L) the library
5. Libraries – select to edit the (-l) the linker libraries search
6. SDK Debug Console – select the SDK Debug Console's PRINTF output to be via UART or to redirect via the C libraries printf function
   - selecting printf will increase the size of the project binary compared to UART output
   - for semihosted printf output to be generated, the project must be linked against a suitable library.
   - for more information see the section on Semihosting and the use of printf [106]
7. Set Floating Point type – select to switch between the available Floating Point options
   - for more information see the section on Hardware Floating Point Support [167]
8. Set Library/Header type – select to switch the current C/C++ Library
   - for more information see the section on C/C++ Library Support [103]

16.2 Part Support Handling from SDKs

MCUXpresso IDE needs specific device information provided by the SDK in order to properly:

- Create/import projects
  - with part specific startup code
- Define memory layout

- Create debugging launch configuration
- Perform flash programming

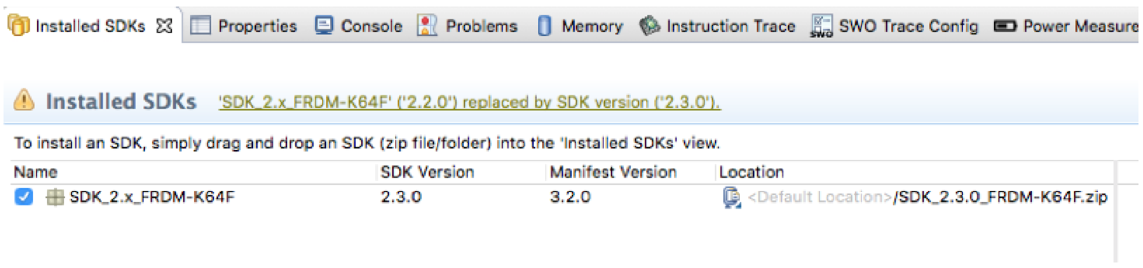
This detailed part knowledge is known as **Part Support**.

16.2.1 SDK Version control

MCUXpresso IDE obtains new Part Support from installed SDKs. The IDE's internal database only uses SDKs with the highest version number. For example, a user may have installed two SDKs for a single part:

- SDK_2.3.0_FRDM-K64F
- SDK_2.0.0_FRDM-K64F

The IDE loads only the 2.3.0 version of that SDK, and also provides a warning in the SDK View header:

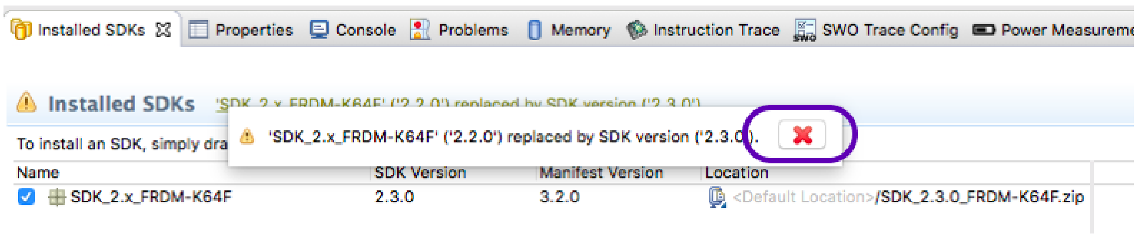


Installed SDKs 'SDK_2.x_FRDM-K64F' ('2.2.0') replaced by SDK version ('2.3.0').

To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view.

Name	SDK Version	Manifest Version	Location
✓ SDK_2.x_FRDM-K64F	2.3.0	3.2.0	<Default Location>/SDK_2.3.0_FRDM-K64F.zip

In this situation, it is likely that the user no longer needs the older version of the SDK. Therefore the IDE provides an option to delete this older SDK via clicking on the warning message, and clicking the 'X'.



Installed SDKs 'SDK_2.x_FRDM-K64F' ('2.2.0') replaced by SDK version ('2.3.0').

To install an SDK, simply dra

'SDK_2.x_FRDM-K64F' ('2.2.0') replaced by SDK version ('2.3.0').

Name	SDK Version	Manifest Version	Location
✓ SDK_2.x_FRDM-K64F	2.3.0	3.2.0	<Default Location>/SDK_2.3.0_FRDM-K64F.zip

Note: Once a new SDK for a part is installed, it will always replace any older installed SDK for that part, even if the new SDK is deactivated (by unchecking the associated tick box). The effect of deactivating an SDK is that part support and wizard will be removed from internal views. These will be restored if the SDK is activated again.

16.2.2 SDK Manifest versioning

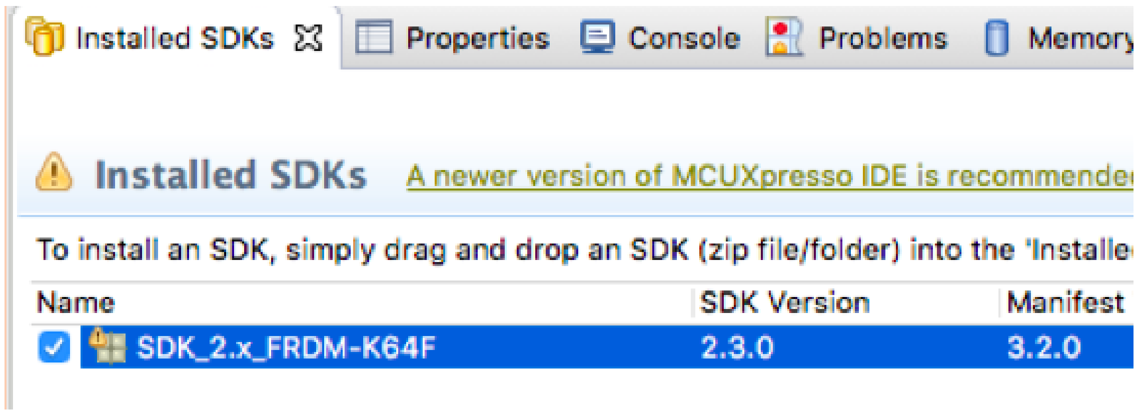
Along with SDK versioning, also the internal manifest in an SDK can have multiple versions. MCUXpresso IDE loads the manifest associated to its internal version head info. Thus, assuming an IDE with internal head version set to 3.2, we could have an SDK with the following manifests:

- Manifest version 3.2
- Manifest version 3.1
- Manifest version 3.0

In such case, the IDE will load the manifest version 3.2.

After loading, the IDE validates the manifest against the schema version head, and if for any reason this is not valid, it will try with the other schema versions. If the manifest 3.2 cannot be validated, then it tries with manifest 3.1, validating it, and so on. The manifest version is shown in the SDK View and any validation errors are shown in the Error Log.

In the case that the IDE loads an older manifest, or in the case the SDK contains a manifest 3.3 and the IDE manifest head is 3.2, the SDK image in the SDK view is decorated with a warning and, by clicking on the SDK, a message appears in the SDK view header:



Installed SDKs A newer version of MCUXpresso IDE is recommende

To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installe

Name	SDK Version	Manifest
SDK_2.x_FRDM-K64F	2.3.0	3.2.0

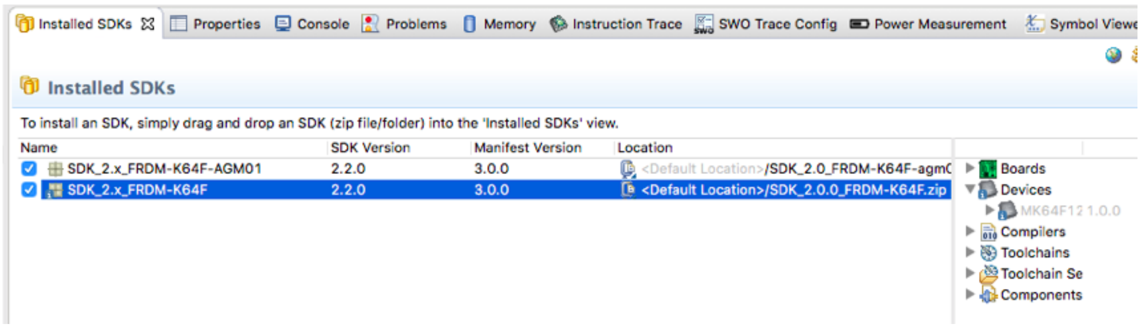
The full error will look like: *"A newer version of the MCUXpresso IDE is recommended for use with the selected SDK. Please update your MCUXpresso IDE in order to get full SDK features"*

At the time of the MCUXpresso IDE v10.1.0 release, such an error should never occur. However, in the future, newer SDKs may be released supporting features not understood by this version of the IDE. This will be used to warn users that there is a miss match between the SDK and IDE capabilities.

16.2.3 Device versions

If the user installs more than one SDK containing the same device (i.e. a device with the same identifier), the IDE loads the part support from the device with the highest version number, regardless of which SDK it is located within. If two or more SDKs have the same device with the same version number, then the order these are presented to the IDE by the host OS will determine which SDK is used.

If an SDK in the Installed SDK view contains a device that is not installed (because it is supplied by another SDK), its image (and the device in the SDK tree) will be decorated with an icon:



Installed SDKs

To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view.

Name	SDK Version	Manifest Version	Location
SDK_2.x_FRDM-K64F-AGM01	2.2.0	3.0.0	<Default Location>/SDK_2.0_FRDM-K64F-agm0
SDK_2.x_FRDM-K64F	2.2.0	3.0.0	<Default Location>/SDK_2.0.0_FRDM-K64F.zip

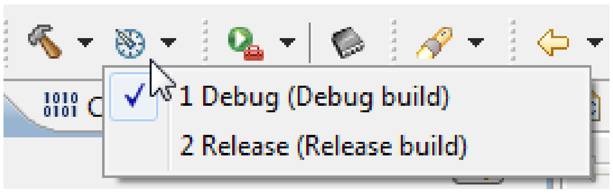
16.3 How do I switch between Debug and Release builds?

By default, MCUXpresso IDE projects will automatically have two build configurations, *Debug* and *Release*. Typically a project will be developed using the Debug build variant, but switched to Release late in the development cycle to benefit from more compilation optimisations.

16.3.1 Changing the build configuration of a single project

You can switch between Debug and Release build configurations by selecting the project you want to change the build configuration for in the Project Explorer view, then using one of the below methods:

- Select the menu item **Project->Build Configuration->Set Active** and select **Release** or **Debug** as necessary
- Use the drop down arrow next to the 'sundial' (Manage configurations for the current project) icon on the main toolbar (next to the 'hammer' icon) and select **Release** or **Debug** as necessary. Alternatively, you can use the drop down next to the 'hammer' icon to change the current configuration and then immediately trigger a build.



- Right click in the Project Explorer view to display the context sensitive menu and select **Build Configurations->Set Active** entry.

16.3.2 Changing the build configuration of multiple projects

It is also possible to set the build configuration of multiple projects at once. This may be necessary if you have a main application project linked with a library project, or you have linked projects for a multicore MCU such as an LPC43xx or LPC541xx (one project for the master Cortex-M4 CPU and another for a slave Cortex-M0/M0+ CPU).

To do this, you first of all you need to select the projects that you wish to change the build configuration for in the Project Explorer view – by clicking to select the first project, then use shift-click or control-click to select additional projects as appropriate. If you want to change all projects, then you can simply use Ctrl-A to select all of them.

Note it is important that when you select multiple projects, you should ensure that none of the selected projects are opened out – in other words, when you selected the projects, you must not have been able to see any of the files or the directory structure within them. If you do not do this, then some methods for changing the build configuration will not be available.

Once the required projects are selected, you then need to simply change the build configuration as you would do for a single project.

16.4 Editing Hints and Tips

The editor view within Eclipse, which sits under the MCUXpresso IDE, provides a large number of powerful features for editing your source files.

16.4.1 Multiple views onto the same file

The **Window -> Editor** menu provides several ways of looking at the same file in parallel.

- **Clone** : two editor views onto the same file
- **Toggle Split Editor** : splits the view onto the current file into two (either horizontally or vertically)

16.4.2 Viewing two edited files at once

To see more than one file at the same time, simply click the file tabs that you have open in the editor view, and then keep the mouse button held down and drag that file tab across to the right. After you've moved to the side, or below, an outline should appear showing you where that tab will be placed once you release the mouse button.

16.4.3 Source folding

Within the editor view, functions, structures etc may be folded to show the structure and hide the detail.

Folding is controlled via, right click in the margin of the editor view to bring up the context sensitive menu, then select **Folding-> <option required>**

Whne folding is enabled, you can then click on the **+** or **-** icon that now appear in the margin next to each function, structure, etc, to expand or collapse it, or use the **Folding->Expand all** and **Folding->Collapse all** options from the context sensitive menu

Various settings for Folding can also be controlled through:

Preferences -> C/C++ -> Editor -> Folding

16.4.4 Editor templates and Code completion

Within the editor, a number of related pieces of functionality allow you to enter code quickly and easily.

First of all, templates are fragments of code that can be inserted in a semi-automatic manner to ease the entering of repetitive code – such as blocks of code for C code structures such as for loops, if-then-else statements and so on.

Secondly, the indexing of your source code that is done by default by the tools, allows for auto completion of function and variable names. This is known as "content assist".

- Ctrl-Space at any point will list available editor template, function names etc.
- Ctrl-Shift-Space will display function parameters
- Alt-/ for word completion (press multiple times to cycle through multiple options).

In addition, the predefined templates are user extensible via:

Preferences -> C/C++ -> Editor -> Templates

16.4.5 Brace matching

The editor can highlight corresponding open and closing braces in a couple of ways.

First of all, if you place the cursor immediately to the right of a brace (either an opening or closing brace), then the editor will display a rectangle around the corresponding brace.

Secondly, if you double click immediately to the right of a brace, then the editor will automatically highlight all of the text between this brace and the corresponding one.

16.4.6 Syntax coloring

Syntax Coloring specifies how your source code is rendered in the editor view, with different colors used for different elements of your source code. The settings used can be modified in:

- Window -> Preferences -> C/C++ -> Editor -> Syntax Coloring*

Note that general text editor settings such as the background color can be configured in:

Window -> Preferences -> General -> Text Editors

Fonts may be configured in:

Window -> Preferences -> General -> Appearance -> Colors and Fonts

16.4.7 Comment/uncomment block

The editor offers a number of ways of comment in or out one or more lines of text. These can be accessed using the Source entry of the editor context-sensitive menu, or using the following keyboard shortcuts...

- Select the line(s) to comment, then hit Ctrl-/ to comment out using // at the start of the line, or uncomment if the line is currently commented out.
- Select the line(s) to comment, then hit Ctrl-Shift-/ to block comment out (placing /* at the start and */ at the end).
- To remove a block comment, hit Ctrl-Shift-\.

16.4.8 Format code

The editor can format your code to match the coding standards in use (**Window -> Preferences -> C/C++ -> Code Style**). This can automatically deal with layout elements such as indentation and where braces are placed. This can be carried out on the currently selected text using the Source->Format entry of the editor context-sensitive menu, or using the keyboard shortcuts Ctrl-Shift-F. If no text is selected, then the format will take place on the whole of the current file.

16.4.9 Correct Indentation

As you enter code in the editor, it will attempt to automatically indent your code appropriately, based on the code standards in use, and also the layout of the preceding text. However, in some circumstances, for example after manually laying text out, you may end up with incorrect indentation.

This can usually be corrected using the Source->Correct Indentation entry of the editor context-sensitive menu, or using the keyboard shortcuts Ctrl-I.

Alternatively, use the “Format code” option which will fix other layout issues in addition to indentation.

16.4.10 Insert spaces for tabs in editor

You can configure the IDE so that when editing a file, pressing the TAB key inserts spaces instead of tab characters. To do this go to

Window -> Preferences -> General -> Editors -> Text Editors

and tick the “Insert spaces for tabs” box.If you tick “Show white-space characters” you can see whether a tab character or space characters are being inserted when you press the TAB key

16.4.11 Replacing tabs with spaces

To replace existing tabs with spaces throughout the file, open the Code Style preferences:

Window -> Preferences -> C/C++ -> Code Style

- Select a Code Style profile and then select Edit...
- Choose the Indentation tab

- For the Tab policy, select Spaces only
- Apply the changes.
  - Note. If the Code Style has not been edited before, the Profile must be renamed before the change can be applied.
- The new style will be applied when the source is next formatted using Source -> Format

16.5 Hardware Floating Point Support

Most ARM-based systems – including those based on Cortex-M0, M0+ and M3, have historically not implemented any form of floating point in hardware. This means that any floating point operations contained in your code will be converted into calls to library functions that then implement the required operations in software.

However, many Cortex-M4 based MCUs do incorporate a single precision floating point hardware unit. Note that the optional Cortex-M4 floating point unit implements single precision operations (C/C++ float) only. Thus if your code makes use of double precision floating point (C/C++ double), then any such floating point operations contained in your code will still be converted into calls to library functions that then implement the required operations in software.

Similarly, Cortex-M7 based MCUs may incorporate a single precision or double precision floating point hardware unit.

16.5.1 Floating Point Variants

When a hardware floating point unit is implemented, ARM define that it may be used in one of two modes.

SoftABI

- Single precision floating point operations are implemented in hardware and hence provide a large performance increase over code that uses traditional floating point library calls, but when calls are made between functions any floating point parameters are passed in ARM (integer) registers or on the stack.
- SoftABI is the ‘most compatible’ as it allows code that is not built with hardware floating point usage enabled to be linked with code that is built using software floating point library calls.

HardABI

- Single precision floating point operations are implemented in hardware, and floating point registers are used when passing floating point parameters to functions.

HardABI will provide the highest absolute floating point performance, but is the ‘least compatible’ as it means that all of the code base for a project (including all library code) must be built for HardABI.

16.5.2 Floating point use – Preinstalled MCUs

When targeting preinstalled MCUs, MCUXpresso IDE generally assumes that when Cortex-M4 hardware floating point is being used, then the SoftABI will be used. Thus generally this is the mode that example code (including for example LPCOpen chip and board libraries) are compiled for. This is done as it ensures that components will tend to work out of the box with each other.

When you use a project wizard for a Cortex-M4 where a hardware floating point unit may be implemented, there will be an option to enable the use of the hardware within the wizard’s options. This will default to SoftABI – for compatibility reasons.

Selecting this option will make the appropriate changes to the compiler, assembler and linker settings to cause SoftABI code to be generated. It will also typically enable code within the startup code generated by the wizard that will turn the floating point unit on.

You can also select the use of HardABI in the wizards. Again this will cause appropriate tool settings to be used. But if you use this, you must ensure that any library projects used by your application project are also configured to use HardABI. If such projects already exist, then you can manually modify the compiler/assembler/linker settings in Project Properties to select HardABI.

Warning : Creating a project that uses HardABI when linked library projects have not been configured and built with this option will result in link time errors.

16.5.3 Floating point use – SDK installed MCUs

When targeting SDK installed MCUs, MCUXpresso IDE generally assumes that when hardware floating point is available, then the HardABI will be used. This will generally work without problem as generally projects for such MCUs contain all required code (with no use of library projects).

However, it is still possible to switch to using SoftABI using the “Advanced Properties settings” page of the |New project" and “Import SDK examples” wizards.

16.5.4 Modifying floating point configuration for an existing project

If you wish to change the floating point ABI for an existing project (for example to change it from using SoftABI to HardABI), then go to:

Quickstart -> Quick Settings -> Set Floating Point type

and choose the required option.

Alternatively, you can configure the settings manually by going to:

Project -> Properties -> C/C++ Build -> Settings -> Tool Settings

and changing the setting in **ALL** of the following entries:

- MCU C Compiler -> Architecture -> Floating point
- MCU Assembler -> Architecture & Headers -> Floating point
- MCU Linker -> Architecture -> Floating point

Note: For C++ projects, you will also need to modify the setting for the MCU C++ Compiler. **Warning:** Remember to change the setting for all associated projects, otherwise linker errors may result.

16.5.5 Do all Cortex-M4 MCUs provide floating point in hardware?

Not all Cortex-M4 based MCUs implement floating point in hardware, so please check the documentation provided for your specific MCU to confirm.

In particular with some MCU families, some specific MCUs may not provide hardware floating point, even though most of the members of the family do (for example the LPC407x_8x). Thus it is a good idea to double check the documentation, even if the project wizard in the MCUXpresso IDE for the family that you are targeting suggests that hardware floating point is available.

16.5.6 Why do I get a hard fault when my code executes a floating point operation?

If you are getting a hard fault when your application tries to execute a floating point operation, then you are almost certainly not enabling the floating point unit. This is normally done in the LPCOpen or SDK initialisation code, or else in the startup file that MCUXpresso IDE generates. But if there are configuration issues with your project, then you can run into problems.

For more information, please see the Cortex-M4 Technical Reference Manual, available on the ARM website.

16.6 LinkServer Scripts

The LinkServer debug server supports a Basic like programming language that can be used to script low level target operations. Within a LinkServer debug connection, we provide two call outs where scripts can be referenced (if required). The first call out is intended to assist with the initial debug connection, via a Connect Script, and the second is to assist with the targets reset via a Reset Script.

These scripts are specified within a LinkServer launch configuration file and will be preselected if needed for projects performing standard connections to known debug targets.

16.6.1 Supplied Scripts

A set of scripts are supplied within the MCUXpresso IDE installation at:

```
<install dir>/ide/bin/Scripts
```

These scripts will be used to prepopulate LinkServer launch configuration files when needed.

The purpose of certain scripts will be described below:

- **kinetismasserase.scp** - invoked by the GUI Flash Programmer to Ressurect locked Kinetis device
- **kinetisunlock.scp** - if for any reason the GUI Flash Programmer fails to resurrect a locked part (as above), this script can be specified in place of the above and the recovery attempt repeated
- **delayexample.scp** - an example script showing how a delay can be performed

16.6.2 Debugging code from RAM

[This section is deprecated – please see Converting Projects to Run from RAM with LinkServer [172] for details of the improved scheme]

MCUs have well defined boot strategies from reset, typically they will first run some internal manufacturer boot ROM code that performs some hardware setup and then control passes to code in flash (i.e. the users Application).

On occasion it can be useful to run and debug code directly from RAM. Since an MCU will not boot from RAM a scheme is needed to take control of the debuggers reset mechanism. This can be achieved the use of a LinkServer reset script.

Within MCUXpresso IDE, certain pre-created scripts are located at:

```
{install dir}/bin/Scripts
```

Contained in this directory is a script called *kinetisRamReset.scp* (see below).

```
10  REM Kinetis K64F Internal RAM (@ 0x20000000) reset script
20  REM Connect script is passed PC/SP from the vector table in the image by the debugger
30  REM For the simple use case we pass them back to the debugger with the location of the
45  REM reset context.
40  REM
50  REM Syntax here is that '~' commands a hex output, all integer variables are a% to z%
70  REM Find the probe index
80  p% = probefirstfound
90  REM Set the 'this' probe and core
100 selectprobecore p% 0
```

```
110 REM NOTE!! Vector table presumed RAM location is address 0x20000000
120 REM The script passes the SP (%b) and PC (%a) back to the debugger as the reset context.
130 b% = peek32 this 0x20000000
140 a% = peek32 this 0x20000004
150 print "Vector table SP/PC is the reset context."
160 print "PC = "; ~a%
170 print "SP = "; ~b%
180 print "XPSR = "; ~c%
190 end
```

This reset script makes an assumption that the user intends to run code from RAM at 0x20000000 – this is the value of the SRAM_Upper RAM block on Kinetis parts.

Note: To build a project to link against RAM, you can simply delete any flash entries within the projects memory configuration. If the MCUXpresso IDEs default linker settings are used then project will link to the first RAM block in the list. For many Kinetis parts, this address will match the expected address within the script. For some parts (for example KLxx) however, the first RAM block may take a different value. This problem can be resolved by editing the script or modifying the projects RAM addresses.

For users if LPC parts, the RAM addresses will be different but the principal remains the same. Within the *Scripts* directory, you will find an RAM reset script for the LPC18LPC43 parts, this script is identical to the one above apart from the assumed RAM address.

Finally, to use the script, simply edit the projects launch configuration for the ‘Reset Script’ entry, browse to the appropriate ‘RAMReset.scp’ script. For information about launch configurations please see the section "Launch Configuration Files::#launchconfig

Note: When executing code from RAM, the projects Vector table will also be located at the start of the RAM block. Cortex M MCUs can locate their vector table using an internal register called *VTOR* (the vector table offset register). Typically this register will be set automatically by a projects startup or init code. However, if execution fails when an interrupt occurs, check that this register is set the correct value.

16.6.3 LinkServer Scripting Features

LinkServer scripts are written in a simple version of the BASIC programming language. In this variant of BASIC, 26 variables are available (%a thru %z). On entry to the script some variable have assigned values:

```
    %a is the PC
    %b is the SP
    %c is the XPSR
```

On exit from the script %a is loaded into the PC and %b is loaded into the SP, thus providing a way for the script to change the startup behavior of the application.

They offer functionality as shown below:

Generic BASIC like functions that only work inside scripts

```
GOTO 'LineNumber'
IF 'relation' THEN 'statement'
REPEAT : Start of a repeat block
UNTIL 'relation' : End with condition of repeat block
BREAKREPEATTO 'LineNumber' : Premature end of a repeat loop
GOSUB 'LineNumber'
```

```
RETURN
TIME : Returns a 10ms incrementing count from the host
```

Generic BASIC like functions

```
PEEK8{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Address'
PEEK16{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Address'
PEEK32{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Address'
POKE8{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Address' 'Data'
POKE16{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Address' 'Data'
POKE32{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Address' 'Data'
QPOKE8  {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address> <Data>
QPOKE16 {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address> <Data>
QPOKE32{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Address' 'Data'
QSTARTTRANSFERS{[THIS] | ['ProbeIndex' 'CoreIndex']}
MEMSAVE{[THIS] | ['ProbeIndex' 'CoreIndex']} 'FileName' 'Byte StartAddress' 'Length in Bytes'
MEMLOAD{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Byte StartAddress' 'LengthLimit in Bytes'
MEMDUMP{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Byte StartAddress' 'Length in Bytes'
EXIT: Exit the server
LIST: Lists the script
NEW: Erases script from memory
RENUMBER: Renumbers in increments of 10
LOAD 'FILENAME': Loads a script from current directory
SAVE 'FILENAME': Saves a script to current directory
```

Probe related functions

```
PROBELIST : Creates and then returns an indexed list of the probes attached
PROBENUM : Returns the number of probes attached
PROBEOPENBYINDEX 'ProbeIndex' : Returns a unique probe handle
PROBECLOSE 'ProbeHandle'
PROBECLOSEBYINDEX 'ProbeIndex' : Returns an error code
PROBETIME 'ProbeIndex' : Returns time from firmware in the probe
PROBESTATUS : Returns an indexed summary of the status of the probes connected to the system
PROBEVERSION 'ProbeIndex': Returns version information about probe firmware
PROBEISOPEN 'ProbeIndex'
PROBEHASJTAG 'ProbeIndex'
PROBEHASSWD 'ProbeIndex'
PROBEHASSWV 'ProbeIndex'
PROBEHASETM 'ProbeIndex'
```

Core/TAP related functions

```
CORELIST 'ProbeIndex': Returns list of TAPs/Cores found connected to specified probe
CORECONFIG{[THIS] | ['ProbeIndex']}: Configures the scanchain
CORESCONFIGURED 'ProbeIndex'
COREREADID 'ProbeIndex' 'CoreIndex'
```

Wire related functions

```
WIRESWDCONNECT{[THIS] | ['ProbeIndex']}: Returns the DPID
WIREJTAGCONNECT{[THIS] | ['ProbeIndex']}:
WIRETIMEDRESET 'ProbeIndex' 'TimeIn_ms': pulls reset and returns the end state of the wire
WIREHOLDRESET 'ProbeIndex' 'State' : pulls reset and returns the end state of the wire
WIRESTATUS 'ProbeIndex' : Returns the status of the wire connection on the probe specified
```

```
WIRESETSPEED 'ProbeIndex' 'SpeedInHz': Requests a particular wire speed
WIREGETSPEED 'ProbeIndex' : Returns the current wire speed
WIRESETIDLECYCLES 'ProbeIndex' 'Cycles': Requests a specific number of idle cycles between debug
 transactions
WIREGETIDLECYCLES 'ProbeIndex' : Returns the current number of debug idle cycles WIREISCONNECTED
'ProbeIndex'
WIREGETPROTOCOL 'ProbeIndex'
SELECTPROBECORE 'ProbeIndex' 'CoreIndex' : sets up for use with following commands
THIS : displays the current Probe, Core pair
```

Cortex-M related functions

```
CMINITAPDP{[THIS] | ['ProbeIndex' 'CoreIndex']}: Initialize a CMx core ready for debug
connections
CMWRITEDP{[THIS] | ['ProbeIndex' 'CoreIndex']} 'REG' 'DATA': returns zero on success
CMWRITEAP{[THIS] | ['ProbeIndex' 'CoreIndex']} 'REG' 'DATA': returns zero on success
CMREADDP{[THIS] | ['ProbeIndex' 'CoreIndex']} 'REG': returns data
CMREADAP{[THIS] | ['ProbeIndex' 'CoreIndex']} 'REG': returns data (note this deals with RDBUF
on AP reads)
CMCLEARERRORS{[THIS] | ['ProbeIndex' 'CoreIndex']}
CMHALT{[THIS] | ['ProbeIndex' 'CoreIndex']}
CMRUN{[THIS] | ['ProbeIndex' 'CoreIndex']}
CMREGS{[THIS] | ['ProbeIndex' 'CoreIndex']}
CMWRITEREG{[THIS] | ['ProbeIndex' 'CoreIndex']} 'RegNumber' 'Value'
CMREADREG{[THIS] | ['ProbeIndex' 'CoreIndex']} 'RegNumber'
CMWATCHLIST{[THIS] | ['ProbeIndex' 'CoreIndex']}
CMWATCHSET{[THIS] | ['ProbeIndex' 'CoreIndex']} 'DWTIndex' 'Address' ['[RW]|R|W']
CMWATCHCLEAR{[THIS] | ['ProbeIndex' 'CoreIndex']} 'DWTIndex'
CMBREAKLIST{[THIS] | ['ProbeIndex' 'CoreIndex']} : List the hardware breakpoints
CMBREAKSET{[THIS] | ['ProbeIndex' 'CoreIndex']} 'Address' : Set an FPB
CMBREAKCLEAR{[THIS] | ['ProbeIndex' 'CoreIndex']} ['Address'] : Clear an FPB
CMSYSRESETREQ{[THIS] | ['ProbeIndex' 'CoreIndex']} : System reset request
CMVECTRESETREQ{[THIS] | ['ProbeIndex' 'CoreIndex']} : Core reset request
CMRESETVECTORCATCHSET{[THIS] | ['ProbeIndex' 'CoreIndex']} : Enable reset vector catch
CMRESETVECTORCATCHCLEAR{[THIS] | ['ProbeIndex' 'CoreIndex']} : Disable reset vector catch
```

Scripts can be specified within a LinkServer launch configuration to be run before a connection and/or before a reset.

16.7 RAM projects with LinkServer

MCUs have well defined boot strategies from reset, typically they will first run internal manufacturer boot ROM code to perform some hardware setup and then pass control to code in flash (i.e. the users Application).

Most examples and wizards create projects to run from MCU flash memory but on occasion it can be useful to debug code directly from RAM. There are two stages to such a task:

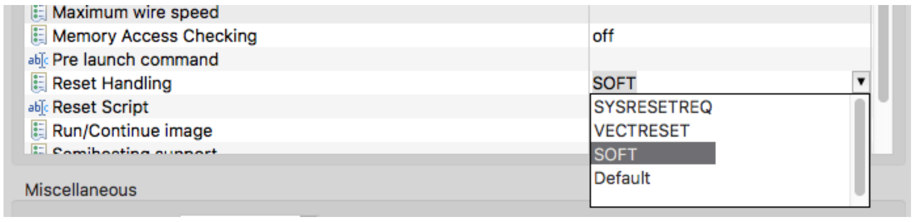
1. Modify a project to that it links to run from RAM
2. Modify the default reset mechanism to ensure that the RAM image is executed

To build a project to link against RAM, simply delete any flash entries within the projects memory configuration. If the MCUXpresso IDEs default linker settings are used then the project will then link against the first RAM block in the list (provided no Flash entry is present). Alternatively, from:

Project Properties -> C/C++ Build -> Settings -> MCU Linker -> Manager Linker Script, you can check the entry *Link application to RAM*.

Note: if the project has already been built to link to flash, then it should be cleaned before being rebuilt.

Since an MCU will not automatically boot from RAM, a scheme is needed to take control of the debuggers reset mechanism. This can be achieved via the use of a **SOFT** reset type. LinkServer launch configurations can take an additional option, add the line *--reset soft* to override the default reset type. Or preferably, set the reset type to *'SOFT'* as shown below.



Maximum wire speed	
Memory Access Checking	off
Pre launch command	
Reset Handling	SOFT
Reset Script	
Run/Continue image	

A soft reset is performed by setting the PC to the images resetISR() address, the stack pointer to the top of the first RAM region and VTOR (Vector Table Offset Register) to the base address of the first RAM region.

Note: Typically, MCU RAM sizes will be smaller that Flash sizes, therefore such a scheme may not be suitable for larger images.

16.7.1 Advantages of developing with RAM projects

There are a number of advantages when debugging from RAM:

- Breakpoints in RAM do not require dedicated HW resources, essentially there is no limit of the number of breakpoints that can be set.
- Flash programming step is not required, so the build and debug cycle will be faster.
- Development of secondary bootloaders is free from BootROM considerations
- No risk of accidently triggering Flash security features.
- No requirement to understand or have flash programming capability allowing code (including flash drivers) can be developed.
- Any flash contents are preserved while debugging
- Unit development of large applications

Note: It should be remembered that since the MCU will not undergo a true hardware reset, peripheral configurations will be inherited from one debug session to the next.

16.8 The Console View

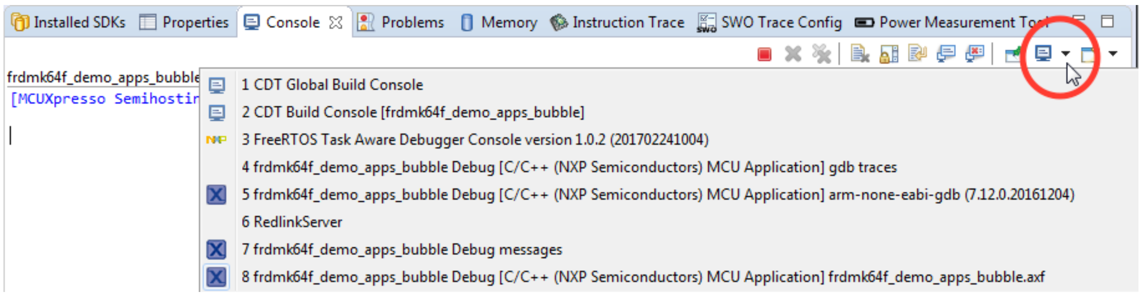
The Console View contains a number of different consoles providing textual information about the operation of various parts of MCUXpresso IDE. It is located by default in the bottom right of the Debug Perspective, in parallel with a number of other views – including the 'Installed SDKs' view.

The actual consoles available within the Console view will depend upon what operations are currently taking place – in particular a number of consoles will only become available once a debug session is started.

The currently displayed console will provide a local toolbar, with icons to do things like copying the contents of the console or clearing its contents.

To see the list of currently available consoles, and, if required, change to a different one..

1. Switch to the Console View
2. Using the toolbar within the Console View click on the drop-down arrow next to the **Display Selected Console** icon (which looks like a small monitor)
3. Select the require console from the drop down list


Installed SDKs | Properties | Console | Problems | Memory | Instruction Trace | SWO Trace Config | Power Measurement Tool

```
frdmk64f_demo_apps_bubble
[MCUXpresso Semihostin
```

1 CDT Global Build Console
2 CDT Build Console [frdmk64f_demo_apps_bubble]
3 FreeRTOS Task Aware Debugger Console version 1.0.2 (201702241004)
4 frdmk64f_demo_apps_bubble Debug [C/C++ (NXP Semiconductors) MCU Application] gdb traces
5 frdmk64f_demo_apps_bubble Debug [C/C++ (NXP Semiconductors) MCU Application] arm-none-eabi-gdb (7.12.0.20161204)
6 RedlinkServer
7 frdmk64f_demo_apps_bubble Debug messages
8 frdmk64f_demo_apps_bubble Debug [C/C++ (NXP Semiconductors) MCU Application] frdmk64f_demo_apps_bubble.axf

16.8.1 Console types

Consoles you will typically see include the following...

Build Console and Global Build Console

The Build Console (sometimes referred to as the *Build Log*) is used by the MCUXpresso IDE build tools (compiler,linker, etc.) to display output generated when building your project. In fact MCUXpresso IDE has two build consoles – one of which records the output from building the current project, and the second a global build console which will record the output from building all projects.

By default, the number of lines stored in the Build Console is limited to 500 lines. You can increase this to any reasonable number as follows:

1. Select the **Windows->Preferences** menu option
2. Now choose **C/C++ -> Build -> Console**
3. Increase the **"Limit Console out (number of lines)"** to a larger number, for instance 5000.

Note: This setting, like most within the MCUXpresso IDE is saved as part of your workspace. Thus you will need to make this change each time you create a new workspace.

Other options that can be set in Preferences include whether the console is cleared before a build, whether it should be opened when a build starts, and whether to bring the console to the top when building.

Once your build has completed, then if you have any build errors displayed in the console, clicking on them will, by default, cause the appropriate source file to be opened at the appropriate place for you to fix the error.

FreeRTOS Task Aware Debugger Console

This console displays status about the FreeRTOS TAD views. For more details, please see the *MCUXpresso IDE FreeRTOS Debug Guide*.

gdb traces and arm-none-eabi-gdb Consoles

These consoles give access to the GDB command line debugger, that sits underneath the MCUXpresso IDE's graphical debugging front end.

RedlinkServer/LinkServer Console

This console gives access to the server application that sits at the bottom of the debug stack when using a debug probe connected via the MCUXpresso IDEs native "LinkServer" debugging mechanism. LinkServer commands can be entered from this console.

Debug messages Console

The Debug Messages Console (sometimes referred to as the *Debug Log*) is used by the debug driver to display additional information that may be helpful in understanding connection issues when debugging your target MCU.

Semihosting Console

This console, generally displayed with *.axf*, allows semihosted output from the application running on the MCU target to be displayed, and potentially for input to be sent down to the target.

16.8.2 Copying the contents of a console

Occasionally, you may wish to copy out the contents of a console. For instance, the MCUXpresso IDE support team may ask you to provide the details of your Build Console in a forum thread. To do this:

1. Clean, then build your project.
2. Select the appropriate Build Console as above:
3. Select the contents (e.g. Ctrl-A)
4. Copy to the clipboard (e.g. Ctrl-C).
5. Paste from clipboard into forum thread (e.g. Ctrl-V). If there is a large amount of text in the build console, it is advisable to paste it into a text file, which can be ZIPed if appropriate.

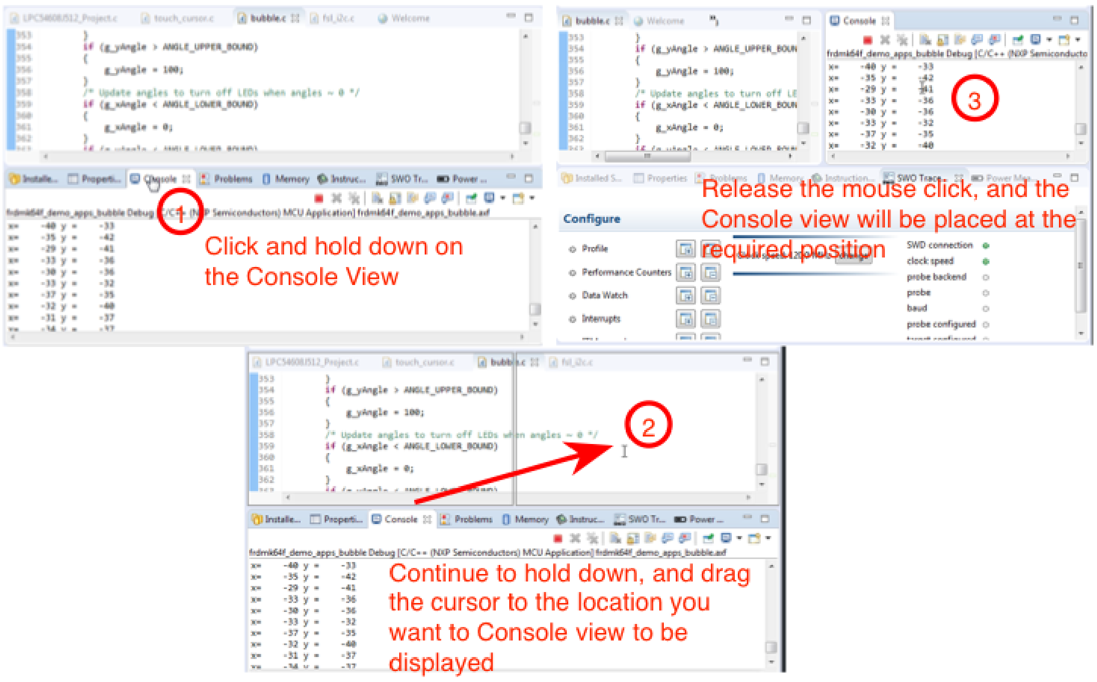
Note that some console will provide a button in their local toolbar to copy or save out their contents.

16.8.3 Relocating and duplicating the Console view

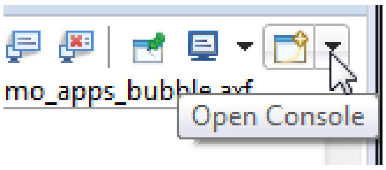
By default the Console View is positioned in parallel with a number of other views. This can mean, if a console is being regularly updated with new output (for instance the view displaying semihosted output from the application running on the target MCU), then by default this may cause the console to keep jumping to the foreground – hence hiding other views that you are using (for instance one of the SWO Trace views)

To avoid this you may wish to relocate the Console. To do this …

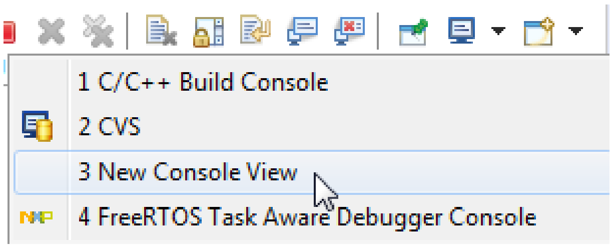
1. Click and hold down on the Console View
2. Continue to hold down, and drag the cursor to the location you want to Console view to be displayed
3. Then release the mouse click, and the Console view will be placed at the required position


1
Click and hold down on the Console View
2
Continue to hold down, and drag the cursor to the location you want to Console view to be displayed
3
Release the mouse click, and the Console view will be placed at the required position
Console
Configure

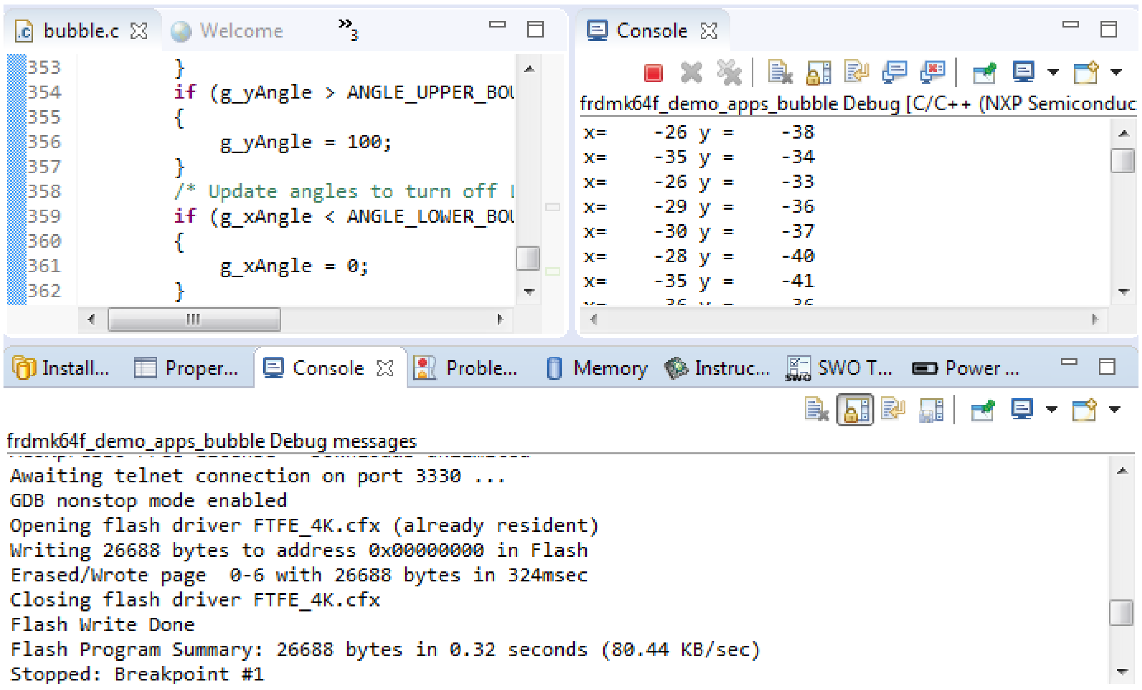
Another alternative is to spawn a duplicate instance of the Console view. This allows multiple consoles to be visible at the same time. To do this use the Open Console button on the Console view local toolbar



and then select "New Console View"



This will then display a second console view, which can be drag and dropped to a new location within in the Perspective, as shown for the single Console view case described above.

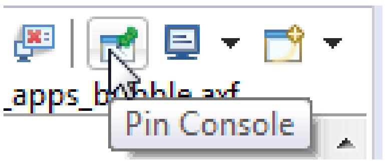

bubble.c
Welcome
Console
frdmk64f_demo_apps_bubble Debug [C/C++ (NXP Semiconduc
Install...
Proper...
Console
Proble...
Memory
Instruc...
SWO T...
Power ...
frdmk64f_demo_apps_bubble Debug messages

```
        }
        if (g_yAngle > ANGLE_UPPER_BOU
        {
            g_yAngle = 100;
        }
        /* Update angles to turn off L
        if (g_xAngle < ANGLE_LOWER_BOU
        {
            g_xAngle = 0;
        }
```

```
x=    -26 y =    -38
x=    -35 y =    -34
x=    -26 y =    -33
x=    -29 y =    -36
x=    -30 y =    -37
x=    -28 y =    -40
x=    -35 y =    -41
```

```
Awaiting telnet connection on port 3330 ...
GDB nonstop mode enabled
Opening flash driver FTFE_4K.cfx (already resident)
Writing 26688 bytes to address 0x00000000 in Flash
Erased/Wrote page  0-6 with 26688 bytes in 324msec
Closing flash driver FTFE_4K.cfx
Flash Write Done
Flash Program Summary: 26688 bytes in 0.32 seconds (80.44 KB/sec)
Stopped: Breakpoint #1
```

Having opened a second console view, select which console you want displayed in it, and then use the “Pin Console” button to ensure that it does not switch to one of the other consoles when output is displayed.



16.9 Using and troubleshooting LPC-Link2

16.9.1 LPC-Link2 hardware

LPC-Link2 is a powerful, low cost debug probe design from NXP Semiconductors based on the LPC43xx MCU. It has been implemented into a number of different systems, including:

- The standalone LPC-Link2 debug probe
- The debug probe built into the range of LPCXpresso V2/V3 boards.

For more details, see http://www.nxp.com/lpcxpresso-boards

16.9.2 Softloaded vs Pre-programmed probe firmware

One thing that most LPC-Link2 implementation offer is the ability to either softload the debug probe firmware (using USB DFU functionality) or to have the debug probe firmware pre-programmed into flash.

Programming the firmware into flash has some advantages, including:

- Allows the use of the LPC-Link2 with toolchains that, unlike MCUXpresso IDE, do not support sofloading of the probe firmware.
- Better supports the use of LPC-Link2 as a small production run programmer
- Allows the LPC-Link2 to be used with SEGGER J-Link firmware as an alternative to the normal CMSIS-DAP firmware. For more details please visit http://www.segger.com
- Avoids issues that the reenumeration of the LPC-Link2 can sometimes trigger as the firmware softloads (particularly where virtual machines are in use).

The recommended way to program the firmware into the flash of LPC-Link2 is NXP's LPCScrypt flash programming tool. For more details, see http://www.nxp.com/LPCSCRYPT

However, when used with MCUXpresso IDE, softloading the probe firmware is the recommended method of using LPC-Link2 in most circumstances.

This ensures that the firmware version matching the MCUXpresso IDE version can automatically be loaded when the first debug session is started (so normally the latest version). It also allows different probe firmware variants to be softloaded, depending on current user requirements.

For this to work, you need to make sure that the probe hardware is configured to allow DFU booting. To do this:

- For standalone LPC-Link2: remove the link from header JP1 (nearest USB)
- For LPCXpresso V2/V3: add a link to the header "DFU link"

16.9.3 LPC-Link2 firmware variants

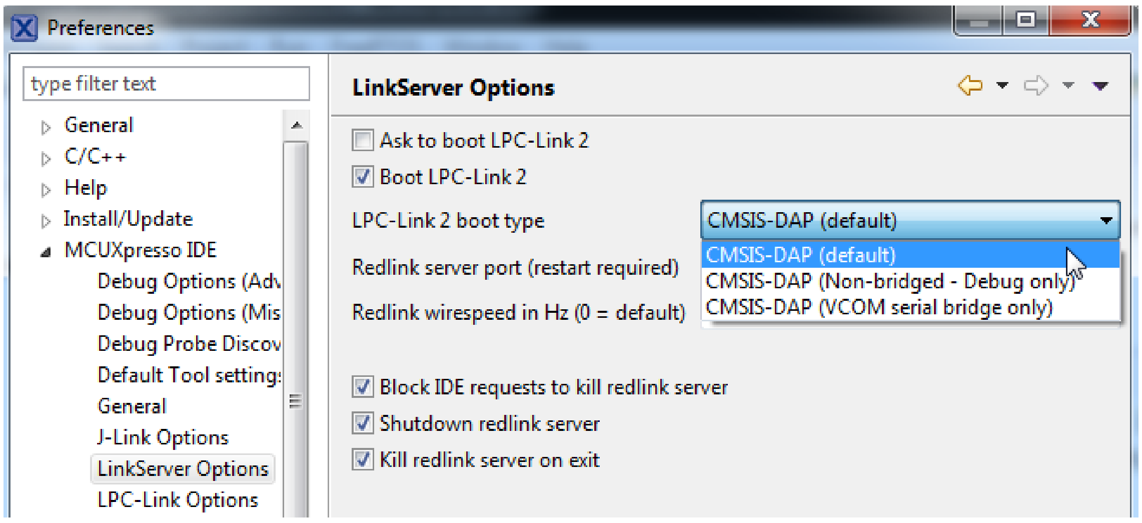
As well as providing debug probe functionality, NXP's CMSIS-DAP firmware for LPC-Link2 by default also includes bridge channels to provide:

- Support for SWO Trace capture from the MCUXpresso IDE
- Support for Power Measurement from the MCUXpresso IDE (certain LPCXpresso V3 boards only)
- Support for a UART VCOM port connected to the target processor (LPCXpresso V2/V3 boards only)
- Support for a LPCSIO bridge that provides communication to I2C and SPI slave devices (LPCXpresso V3 boards only)

However, two other variants of the CMSIS-DAP firmware are provided that remove some of these bridge channels.

- **"Non Bridged":** This version of firmware provides debug features only – removing the bridged channels such as trace, power measurement and VCOM. By removing the requirement for these channels, USB bandwidth is reduced, therefore this firmware may be preferable if multiple debug probes are to be used concurrently. The non-bridged build will also provide an increase in download and general debug performance.
- **"VCOM Only":** This version of firmware provides only debug and VCOM features. The removal of the other bridges allows better VCOM performance (though generally the bridged firmware provides more than good enough VCOM performance).

A particular workspace can be switched to softload a different firmware variant via **Preferences – MCUXpresso IDE – LinkServer Options – LPC-Link2 boot type**.



LinkServer Options

- [ ] Ask to boot LPC-Link 2
- [✓] Boot LPC-Link 2

LPC-Link 2 boot type: CMSIS-DAP (default)
- CMSIS-DAP (default)
- CMSIS-DAP (Non-bridged - Debug only)
- CMSIS-DAP (VCOM serial bridge only)

Redlink server port (restart required)

Redlink wirespeed in Hz (0 = default)

- [✓] Block IDE requests to kill redlink server
- [✓] Shutdown redlink server
- [✓] Kill redlink server on exit

Note: If a mix of bridged and unbridged debug probes is required, then it is recommended that these probes are pre-programmed with the required debug firmware. This can easily be done via LPCScrypt.

16.9.4 Manually booting LPC-Link2

The recommended way to use LPC-Link2 with the MCUXpresso IDE is to allow the GUI to boot and softload a debug firmware image at the start of a debug session.

Normally, LPC-Link2 is booted automatically (when configured to operate in DFU mode) however, under certain circumstances – such as when troubleshooting issues, or using the LinkServer command line flash utility, you may need to boot it manually.

LPC-Link2 USB Details

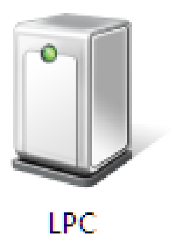
The standard utilities to explore USB devices on MCUXpresso IDE supported host platforms are:

- Windows – Device Manager
  - MCUXpressoIDE also provides a listusb utility in:
    - *{install_dir}*\ide\bin\Scripts
- Linux – terminal command: lsusb
- Mac OS X – terminal command: system_profiler SPUSBDataType

Before boot, LPC-Link2 appears as a USB device with details:

```
Device VendorID/ProductID:   0x1FC9/0x000C    (NXP Semiconductors)
```

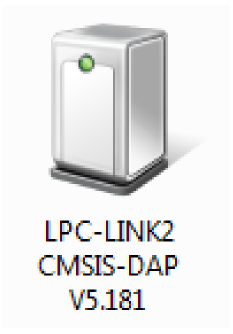
and will appear in Windows -> Devices and Printers, as below:



After boot, LPC-Link2 will by default appear as a USB device with details:

```
Device VendorID/ProductID:   0x1FC9/0x0090
```

and will appear in Windows -> Devices and Printers similar to below:



Note: Text details will vary depending on version number and which probe firmware variant is booted.

Booting from the command line

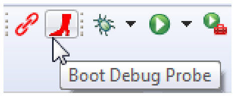
MCUXpresso IDE provides a boot script for all supported platforms. To make use of this script first of all connect the LPC-Link2 to your PC then enter the commands into a DOS command prompt (or equivalent):

```
cd {install_dir}\ide\bin
boot_link2
```

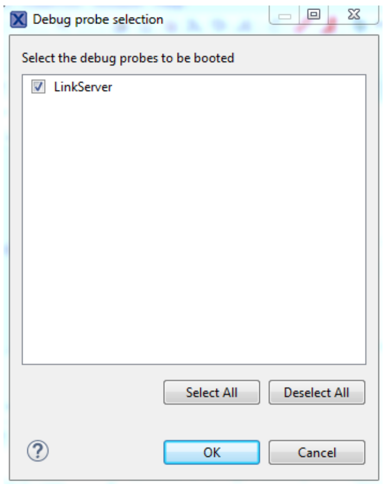
This will invoke the dfu-util utility to download the probe firmware into the RAM of the LPC-Link2's LPC43xx MCU and then reenumerate the probe.

Booting from the GUI

It is also possible to manually boot LPC-Link2 from the MCUXpresso IDE GUI, which may be a more convenient solution than using the command line. To do this, first of all connect the LPC-Link2 to your PC, then locate the red Boot icon on the Toolbar:



and then click OK in the dialog displayed :



16.9.5 LPC-Link2 windows drivers

The drivers for LPC-Link2 are installed as part of the main MCUXpresso IDE installation process.

- Note:* One thing to be aware of is that the first time you debug using a particular LPC-Link2 on a particular PC, the drivers will need to be loaded. This first time can take a variable period of time depending upon your PC and operating system version. This may mean that the first debug attempt fails, as the IDE may time out waiting for the booted LPC-Link2 to appear. In such as case, a second debug attempt should complete successfully. Otherwise, try booting the LPC-Link2 manually and checking the drivers load correctly.

If you need to reinstall the drivers, then the installer can be found at:

```
C:\nxp\{install_dir}\Drivers\lpc_driver_installer.exe
```

16.9.6 LPC-Link2 failing to enumerate

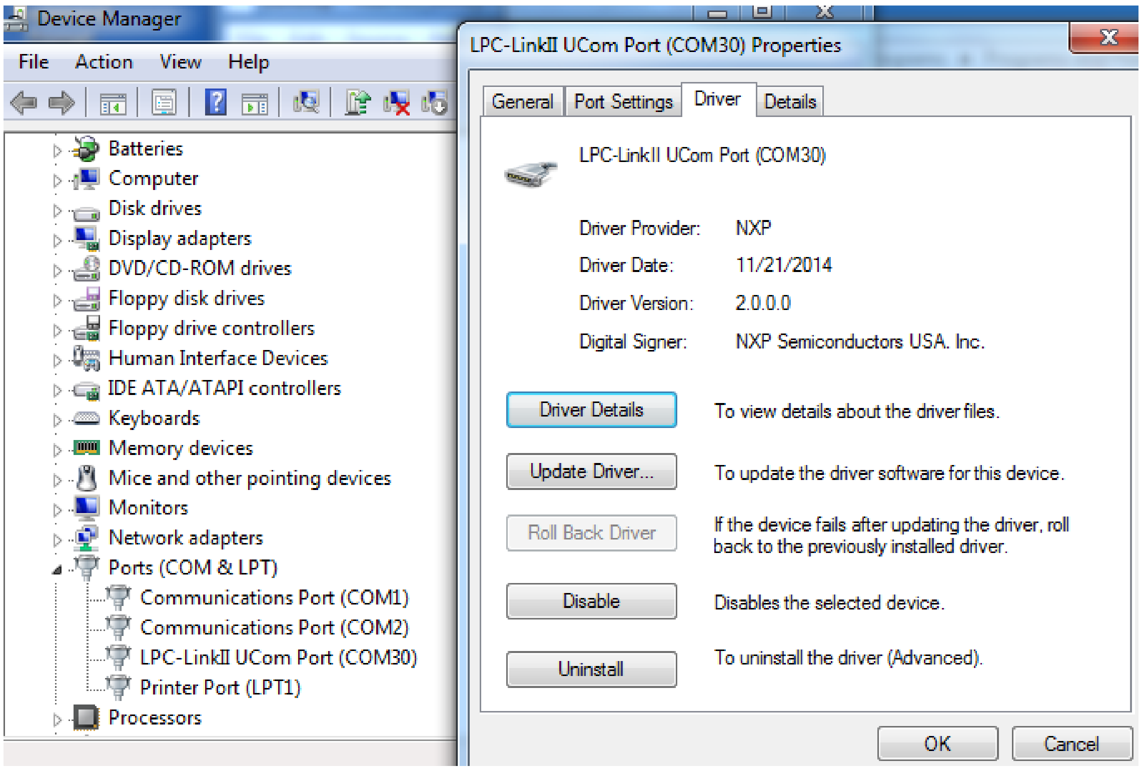
On some systems, after booting LPC-Link2 with CMSIS-DAP firmware, the booted debug probe does not enumerate correctly and the MCUXpresso IDE (or other toolchain) is unable to see the debug probe. This problem is normally caused by on old, obsolete, version of the VCOM driver being found by Windows instead of the the correct driver. To see if this is the cause of a problem on your computer, find the version number of the LPC-Link2 VCOM driver. The obsolete driver version is 1.0.0.0.

To find the version number of the LPC-Link2 VCOM driver

If you are using a soft-booted LPC-Link2 debug probe, start by booting your LPC-Link2, as described in Manually booting LPC-Link2 [178]. If your LPC-Link2 debug probe is booting from an image preprogrammed into the flash, you can skip this step.

Once your LPC-Link2 has booted, find the device in Device Manager and look at the driver version number.

- Open the Windows Device Manager
- Expand the “Ports (COM and LPT)” section
- Right-click on “LPC-LinkII UCom Port”, and select Properties
- Click on the Driver tab of the Properties dialog



LPC-LinkII UCom Port (COM30)

Driver Provider: NXP
Driver Date: 11/21/2014
Driver Version: 2.0.0.0
Digital Signer: NXP Semiconductors USA. Inc.

To view details about the driver files.
To update the driver software for this device.
If the device fails after updating the driver, roll back to the previously installed driver.
Disables the selected device.
To uninstall the driver (Advanced).

Note that this image shows the current correct version of the driver (2.0.0.0).

Removing the obsolete 1.0.0.0 LPC-LinkII UCOM driver

To remove the obsolete driver, perform the following actions:

1. In Device Manager, right-click on the LPC-LinkII UCOM device and select Uninstall
2. If there is an option to delete the driver software, make sure it is checked, and press OK
3. Select the menu item Action->Scan for hardware changes
4. In Windows Control Panel, select Add/Remove program or Uninstall a program option
5. Find the LPC Driver Installer, right-click on choose Uninstall
6. Let the uninstaller complete
7. Switch back to the Device Manager and Scan for hardware changes again
8. If the LPC-LinkII UCOM driver version is still present, Uninstall it again (steps 1 through 3) and repeat until the LPC-LinkII UCOM driver no longer appears
9. Now run the lpc_driver_installer.exe found in the MCUXpresso IDE “Drivers” directory

Note: A reboot is recommended after running the lpc_driver_installer.exe installer.

Now manually reboot the probe again (if softloading) and check **Windows – Devices and Printers** to see if the device now appears correctly as an LPC-Link2 CMSIS-DAP Vx.xxx.

If this fails to correct the problem, there is one final thing to try:

- Open a Command Prompt as the Administrative user and run the following commands

```
cd %temp%
pnputil -e >devices.txt
notepad devices.txt
```

- Search devices.txt for an entry similar to this, and note down the Published name (oemXX.inf)

```
Published name :            oem38.inf
Driver package provider :   NXP
Class :                     Ports (COM & LPT)
Driver date and version :   09/12/2013 1.0.0.0
Signer name :               NXP Semiconductors USA. Inc.
```

- Using the name notes above, run the following command (replacing XX with the number found above)

```
pnputil -f -d oemXX.inf
```

16.9.7 Troubleshooting LPC-Link2

If you have been able to use LPC-Link2 in a debug session but now see issues such as “No compatible emulator available” or “Priority 0 connection to this core already taken” when trying to perform a debug operation ...

- Ensure you have shut down any previous debug session
  - You must close a debug session (press the Red ‘terminate’ button) before starting another debug session
- It is possible that the debug driver is still running in the background. Use the task manager or equivalent to kill any tasks called:
  - redlinkserv
  - arm-none-eabi_gdb*
  - crt_emu_*

If your host has never worked with LPC-Link2, then the following may help to identify the problem:

- Try manually booting your LPC-Link2 as per Manually booting LPC-Link2, and ensure that the drivers have installed correctly.
- Try a different USB cable!
- Try a different USB port. If your host has USB3 and USB2, then try a USB2 port
  - there are known issues with motherboard USB3 firmware, ensure your host is using the latest driver from the manufacturer. Note, this is not referencing the host OS driver but the motherboard firmware of the USB port
- If using a USB hub, first try a direct connection to the host computer
- If using a USB hub, try using one with a separate power supply – rather than relying on the supply over USB from your PC.
- Try completely removing and re-installing the host device driver. See also LPC-Link2 fails to enumerate [180] above.
- If using Windows 8.1 or later, then sometimes the Windows USB power settings can cause problems. For more details use your favourite search engine to search for “windows 8 usb power settings” or similar.

16.10 Make fails with Virtual Alloc pointer is null error

Very rarely, building a project on Windows may result in an error similar to this:

```
0 [main] us 0 init_cheap: VirtualAlloc pointer is null, Win32 error 487
AllocationBase 0x0, BaseAddress 0x71110000, RegionSize 0x350000, State 0x10000
\msys\bin\make.exe: *** Couldn't reserve space for cygwin's heap, Win32 error 0
```

This is a problem that affects a tiny minority of customers, and depends on what other applications they are running at the same time. This is caused by a feature in the MSYS binaries that we use to provide the the build environment for the MCUXpresso IDE on Windows.

If this happens, you can replace the file **\ide\msys\bin\msys-1.0.dll** within your MCUXpresso IDE install directory with the **msys-1.0-alternate.dll** file in the same directory (i.e. do a rename)

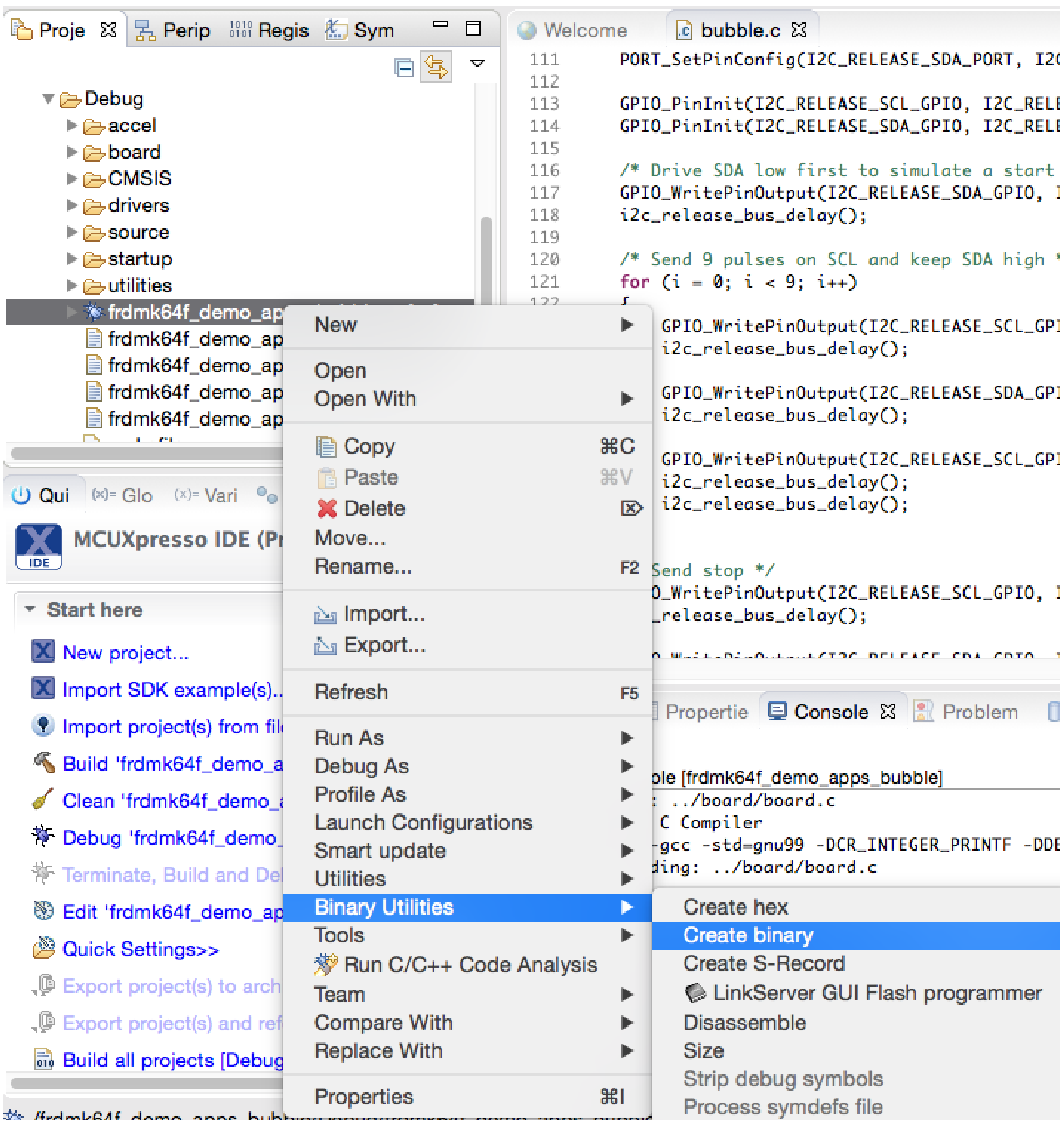
Note that this does not fix the problem, rather it moves DLL base address. Unfortunately, it is possible the error may occur with this replacement DLL too, again depending on what other applications are running. In which case you will need to revert to the original DLL again.

16.11 Creating bin, hex or S-Record files

When building a project, the MCUXpresso IDE tools create an ARM executable format (AXF) file – which is actually standard ELF/DWARF file. This file can be programmed directly down to your target using the MCUXpresso IDE debug functionality, but it may also be converted into a variety of formats suitable for use in other external tools.

16.11.1 Simple conversion within the IDE

The simplest way to create a one-off binary or hex file is to open up the Debug (or Release) folder in Project Explorer right click on the .axf file, and **"Binary Utilities->Create binary"** (or Create hex, S-Record).



```
    PORT_SetPinConfig(I2C_RELEASE_SDA_PORT, I2(

    GPIO_PinInit(I2C_RELEASE_SCL_GPIO, I2C_RELE
    GPIO_PinInit(I2C_RELEASE_SDA_GPIO, I2C_RELE

    /* Drive SDA low first to simulate a start
    GPIO_WritePinOutput(I2C_RELEASE_SDA_GPIO, 1
    i2c_release_bus_delay();

    /* Send 9 pulses on SCL and keep SDA high
    for (i = 0; i < 9; i++)
    {
        GPIO_WritePinOutput(I2C_RELEASE_SCL_GPI
        i2c_release_bus_delay();

        GPIO_WritePinOutput(I2C_RELEASE_SDA_GPI
        i2c_release_bus_delay();

        GPIO_WritePinOutput(I2C_RELEASE_SCL_GPI
        i2c_release_bus_delay();
        i2c_release_bus_delay();

    Send stop */
    O_WritePinOutput(I2C_RELEASE_SCL_GPIO, 1
    _release_bus_delay();
```

```
ole [frdmk64f_demo_apps_bubble]
: ../board/board.c
 C Compiler
-gcc -std=gnu99 -DCR_INTEGER_PRINTF -DDE
ding: ../board/board.c
```

You can also change the underlying commands and options that are called by these menu entries from the " **Preferences->MCUXpresso IDE ->Utilites**" preference page.

16.11.2 From the command line

The above “Binary Utilities” option within the IDE GUI is simply invoking the command line objcopy tool (arm-none-eabi-objcopy). Objcopy can convert into the following formats:

- srec (Motorola S record format)
- binary
- ihex (Intel hex)
- tekhex

For example, to convert example.axf into binary format, use the following command:

arm-none-eabi-objcopy -O binary example.axf example.bin

If you ctrl-click on the project name on the right hand side of the bottom bar of the IDE, this will launch a command prompt in the project directory with appropriate tool paths set up. You can also use the Project Explorer right-click “Utilities->Open command prompt here” option to do this.

All you need to do before running the objcopy command is change into the directory of the required Build configuration.

16.11.3 Automatically converting the file during a build

Objcopy may be used to automatically convert an axf file during a build. To do this, create an appropriate Post-build step

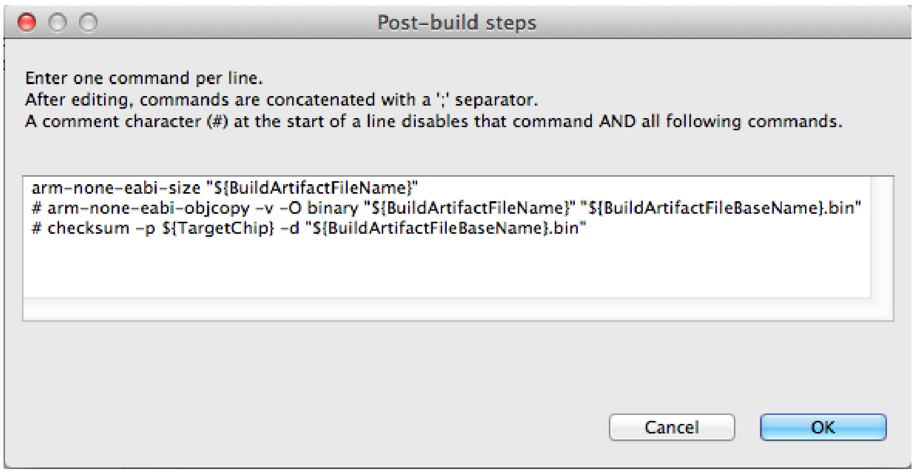
16.11.4 Binary files and checksums

When creating a binary file for most LPC MCUs, you also need to ensure that you apply a checksum to it – so that the LPC bootloader sees the image as being valid. Generally the linker script will do this if the managed linker script mechanism is used. Otherwise the “checksum” utility found in the \ide\bin subdirectory of your MCUXpresso IDE installation can be used.

16.12 Post-build (and Pre-build) steps

It is sometimes useful to be able to automatically post-process your linked application, typically to run one or more of the GNU ‘binutils’ on the generated AXF file.

For example, any application project that you create using the Project wizard will have least one such “post-build step” - typically to display the size of your application.



Enter one command per line.
After editing, commands are concatenated with a ';' separator.
A comment character (#) at the start of a line disables that command AND all following commands.

```
arm-none-eabi-size "${BuildArtifactFileName}"
# arm-none-eabi-objcopy -v -O binary "${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.bin"
# checksum -p ${TargetChip} -d "${BuildArtifactFileBaseName}.bin"
```

Note: Additional commands may also be listed (for example to create a binary and to run a checksum command), but be commented out by use of a # character and hence not executed. Any commands following a comment #command will be ignored.

Adding additional steps is very simple. In the below example we are going to carry out three post-link steps:

- displaying the size of the application
- generate an interleaved C / assembler listing
- create a hex version of the application image

To do this:

- Open the Project properties. There are a number of ways of doing this. For example, make sure the Project is highlighted in the Project Explorer view then open the menu “Project -> Properties”.
- In the left-hand list of the Properties window, open “C/C++ Build” and select “Settings”.
- Select the “Build steps” tab
- In the “Post-build steps - Command” field, click 'Edit...'
  - Paste in the lines below and click 'OK'

```
arm-none-eabi-size ${BuildArtifactFileName};
arm-none-eabi-objdump -S ${BuildArtifactFileName} > ${BuildArtifactFileBaseName}.lss;
arm-none-eabi-objcopy -O ihex ${BuildArtifactFileName} ${BuildArtifactFileBaseName}.hex;
```

- Click apply
- Repeat for your other Build Configurations (Debug/Release)

Next time you do a build, this set of post-build steps will now run, displaying the application size in the console, creating you an interleaved C/assembler listing file called *.lss* and a hex file called *hex*.

Note: Pre-build steps can be added to a project in exactly the same way if required.

Temporarily removing post-build steps

If you want to temporarily remove a step from your post-build process, rather than deleting it completely – move that entry to the end of the line and pre-fix it with a “#” (hash) character. This acts as a comment, causing the rest of the post-build steps to be ignored.