



MCUXpresso IDE User Guide

Rev. 10.0 — 22 February, 2017

User guide



22 February, 2017

Copyright © 2017 NXP Semiconductors

All rights reserved.

1. Introduction to MCUXpresso IDE	1
1.1. MCUXpresso IDE Overview of Features	1
1.1.1. Summary of Features	1
1.1.2. Supported Debug Probes	3
1.1.3. Development Boards	3
2. IDE Overview	6
2.1. Documentation and Help	6
2.2. Workspaces	6
2.3. Perspectives and Views	7
2.4. Major Components of the Develop Perspective	8
3. Debug Solutions Overview	10
3.1. A note about Launch Configuration files	11
3.2. LinkServer Debug Connections	13
3.3. LinkServer Debug Operation	14
3.4. LinkServer Global and Live Global Variables	15
3.5. LinkServer Troubleshooting	18
3.5.1. Debug Log	18
3.5.2. Flash Programming issues	19
3.5.3. LinkServer executables	20
3.6. P&E Debug Connections	20
3.7. P&E Debug Operation	20
3.7.1. P&E Differences from LinkServer Debug	21
3.7.2. P&E Micro Software Updates	21
3.8. SEGGER Debug Connections	21
3.9. SEGGER Debug Operation	22
3.9.1. SEGGER Differences from LinkServer Debug	23
3.10. SEGGER Troubleshooting	23
4. SDKs and Pre-Installed Part Support Overview	25
4.1. Pre-installed Part Support	25
4.2. SDK Part Support	25
4.2.1. Important notes for SDK users	26
4.2.2. Differences in Pre-installed and SDK part handling	26
4.3. Viewing Pre-installed Part Support	27
4.4. Installing an SDK	27
4.4.1. "Power User" SDK Importing and Configuration	30
5. Creating New Projects using SDKs	32
5.1. New Project Wizard	32
5.1.1. SDK New Project Wizard: Basic Project Creation and Settings	34
5.1.2. SDK New Project Wizard: Advanced Project Settings	36
5.2. SDK Build Project	39
6. Importing Example Projects (from SDKs)	40
6.1. SDK Example Import Wizard	41
6.1.1. SDK Example Import Wizard: Basic Selection	41
6.1.2. SDK Example Import Wizard: Advanced options	44
6.1.3. SDK Example Import Wizard: Import from XML fragment	45
7. Creating New Projects using Pre-Installed Part Support	47
7.1. New Project Wizard	47
7.2. Creating a Project using a Wizard	48
7.2.1. Selecting the Wizard Type	49
7.2.2. Configuring the Project	50
7.3. Wizard Options	50
7.3.1. LPCOpen Library Project Selection	50
7.3.2. CMSIS-CORE Selection	51
7.3.3. CMSIS DSP Library Selection	52
7.3.4. Peripheral Driver Selection	52
7.3.5. Code Read Protect	52
7.3.6. Enable use of Floating Point Hardware	52

7.3.7. Enable use of Romdivide Library	52
7.3.8. Disable Watchdog	53
7.3.9. LPC1102 ISP Pin	53
7.3.10. Redlib Printf Options	53
7.3.11. Project Created	53
8. Importing Example Projects (from the filesystem)	54
8.1. Code Bundles for LPC800 Family devices	54
8.2. LPCOpen Software Drivers and Examples	54
8.3. Importing an Example Project	55
8.3.1. Importing Examples for the LPCXpresso4337 Development Board	56
8.4. Building Projects	57
8.4.1. Build Configurations	57
9. Debugging a Project	59
9.1. Debugging overview	59
9.1.1. Debug Probe Selection Dialog	60
9.1.2. Controlling Execution	61
10. C/C++ Library Support	63
10.1. Overview of Redlib, Newlib and NewlibNano	63
10.1.1. Redlib extensions to C90	63
10.1.2. Newlib vs NewlibNano	63
10.2. Library variants	64
10.3. Switching the selected C library	65
10.3.1. Manually switching	65
10.4. What is Semihosting?	66
10.4.1. Background to Semihosting	66
10.4.2. Semihosting implementation	66
10.4.3. Semihosting Performance	66
10.4.4. Important notes about using semihosting	66
10.4.5. Semihosting Specification	67
10.5. Use of printf	67
10.5.1. Redlib printf variants	67
10.5.2. NewlibNano printf variants	67
10.5.3. Newlib printf variants	68
10.5.4. Printf when using LPCOpen	68
10.5.5. Printf when using SDK	68
10.5.6. Retargeting printf/scanf	68
10.5.7. How to use ITM Printf	69
10.6. itoa() and uitoa() in Redlib	70
10.6.1.	71
10.6.2. Newlib/NewlibNano	71
10.7. Libraries and linker scripts	71
11. Memory Configuration and Linker Scripts	73
11.1. Introduction	73
11.2. Managed Linker Script Overview	74
11.3. How are managed linker scripts generated?	75
11.4. Default image layout	75
11.5. Examining the layout of the generated image	77
11.5.1. Linker --print-memory-usage	77
11.5.2. arm-none-eabi-size	77
11.5.3. Linker map files	78
11.5.4. Symbol Viewer	78
11.6. Other options affecting the generated image	80
11.6.1. LPC MCUs – Code Read Protection	80
11.6.2. Kinetis MCUs – Flash Config blocks	81
11.6.3. Placement of USB data	82
11.7. Modifying the generated linker script / memory layout	82

11.8. Using the Memory Configuration Editor	83
11.8.1. Editing a Memory Configuration	83
11.8.2. Device specific vs Default Flash Drivers	86
11.8.3. Restoring a Memory Configuration	87
11.8.4. Copying Memory Configurations	87
11.9. More advanced heap/stack placement	87
11.9.1. MCUXpresso style heap and stack	88
11.9.2. LPCXpresso style heap and stack	89
11.9.3. Reserving RAM for IAP Flash Programming	89
11.9.4. Stack checking	90
11.9.5. Heap Checking	91
11.9.6. Placement of specific code/data items	91
11.10. Freemaker Linker Script Templates	96
11.10.1. Basics	96
11.10.2. Reference	96
11.11. Freemaker Linker Script Template Examples	100
11.11.1. Relocating code from FLASH to RAM	100
11.11.2. Configuring projects to span multiple flash devices	102
11.12. Disabling managed linker scripts	103
12. Multicore Projects	105
12.1. LPC43xx Multicore Projects	105
12.2. LPC541xx Multicore Projects	105
A. Glossary	106

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:

- The MCUXpresso IDE, a software development environment for creating applications for NXP's ARM-based "LPC" and "Kinetis" range of MCUs.
- MCUXpresso SDKs, 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.

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 affordable professional editions, 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, visit <http://www.nxp.com/mcuxpresso/ide>

1.1.1 Summary of Features

- Complete C/C++ integrated development environment
 - Latest Eclipse-based IDE with many ease-of-use enhancements
 - Eclipse Neon (v4.6) and CDT (9.2)
- 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 (v5 update 3) 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 (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 support
 - SEGGER J-Link support
- 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 CAN
- LinkServer Power Measurement
 - On enabled boards, sample power usage at adjustable rates of up to 200 ksp/s; average

- Explore detailed plots of collected data in the IDE
- Export data for analysis with other tools

1.1.2 Supported Debug Probes

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

- **Native LinkServer** (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 [10] for more details.

Note: Kinetis Freedom and Tower boards typically provide an onboard 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 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/pages/:OPENSDA>

1.1.3 Development Boards

NXP Development board come in 3 families:

LPCXpresso Boards for LPC

A major part of the LPCXpresso platform is 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.

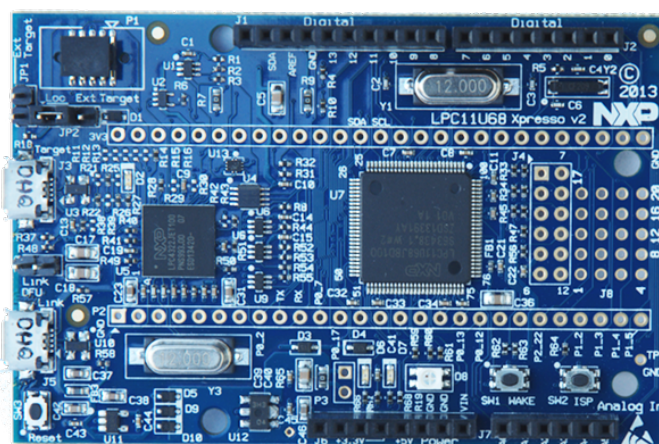


Figure 1.1. LPCXpresso V2 Board (LPCXpresso11U68)

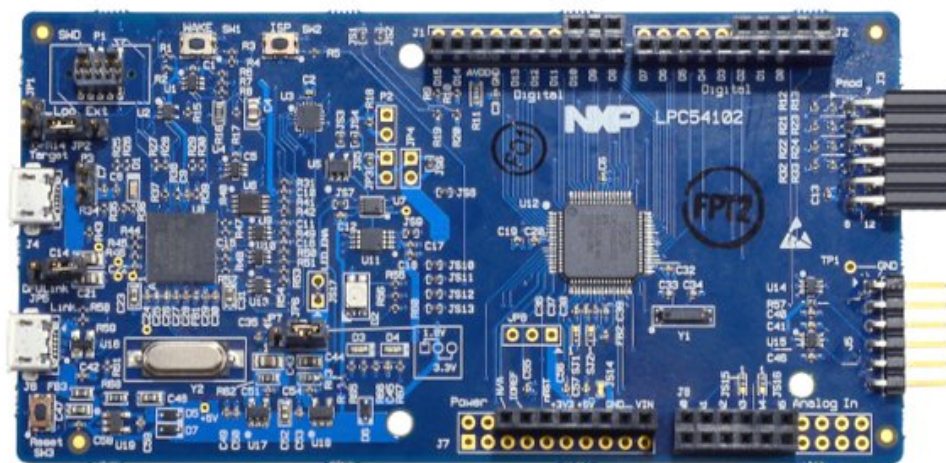


Figure 1.2. LPCXpresso V3 Board (LPCXpresso54102)

For more information, visit: <http://www.nxp.com/pages/LPCXPRESSO-BOARDS>

Freedom and Tower Boards for Kinetis

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

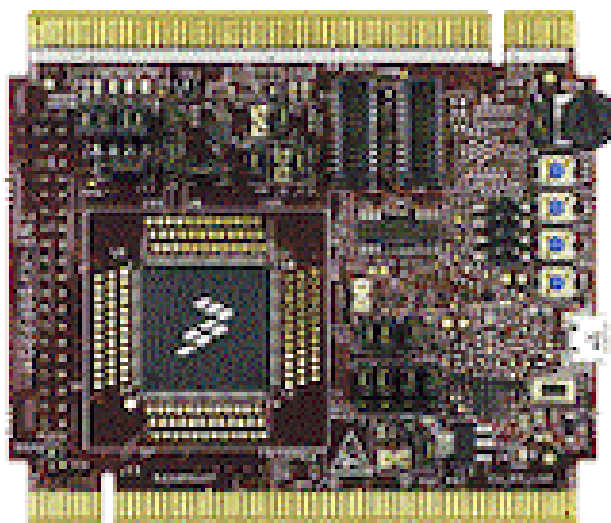


Figure 1.3. Tower (TWR-KV58F220M)

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

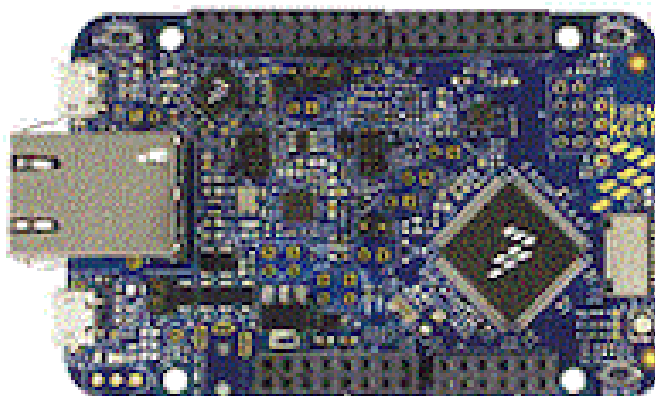


Figure 1.4. Freedom (FRDM-K64F)

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. That 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 and Licensing 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

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

Commonly used

2.2 Workspaces

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

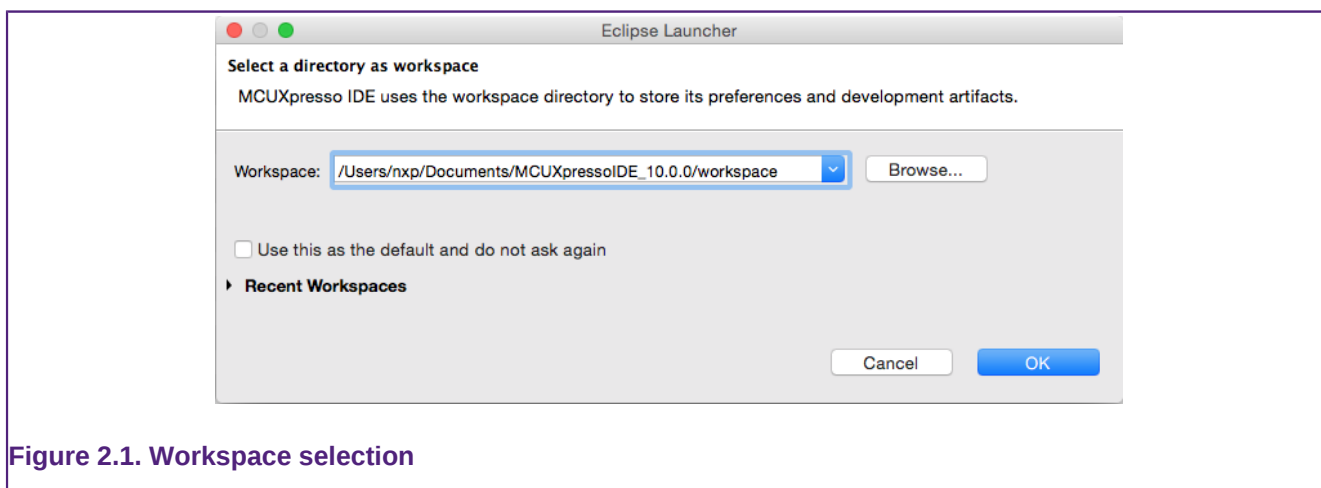


Figure 2.1. Workspace selection

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.

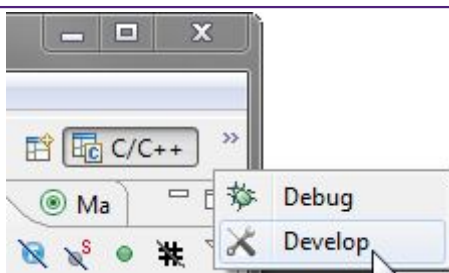


Figure 2.2. Perspective selection

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 -> Reset Perspective**.

2.4 Major Components of the Develop Perspective

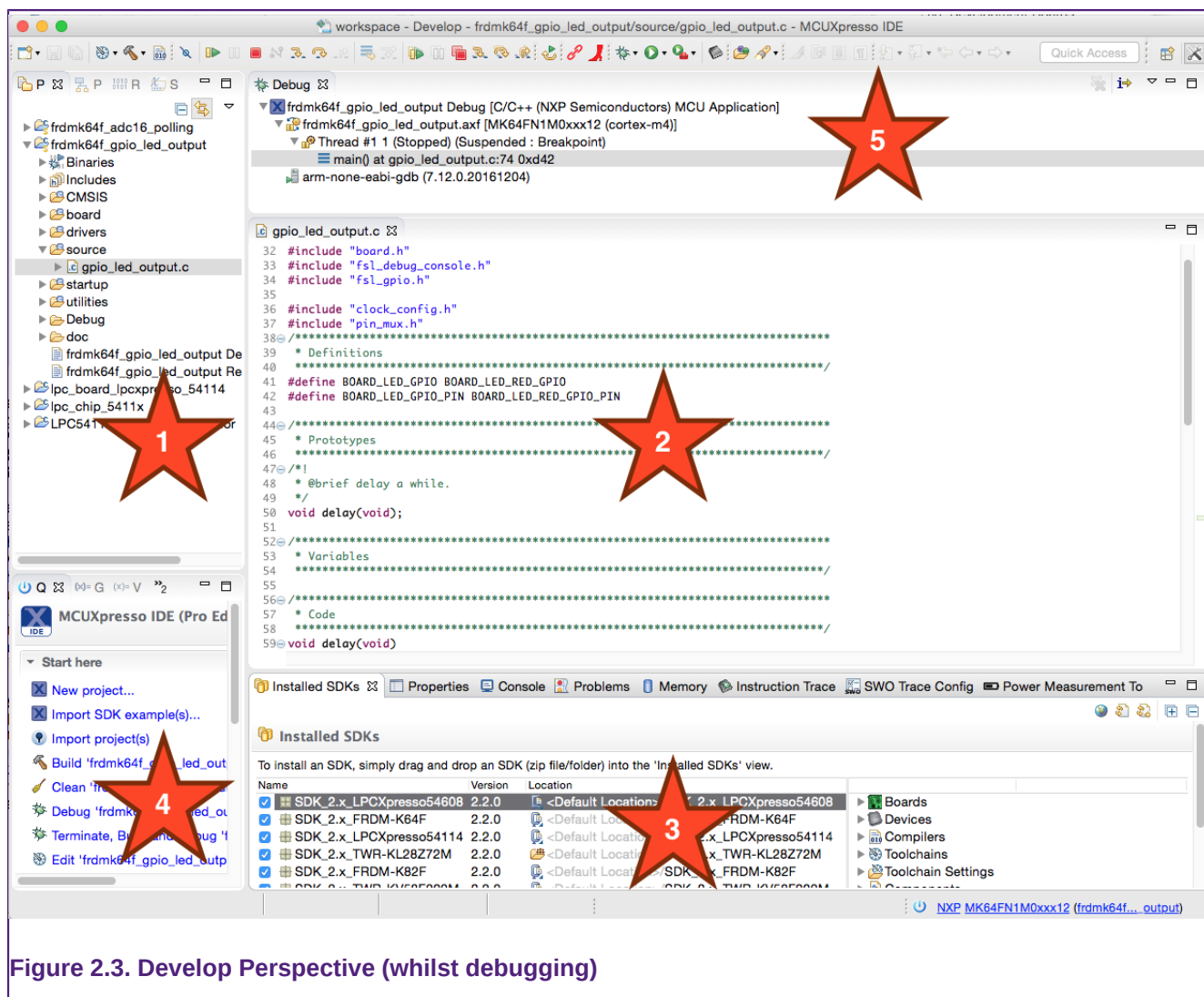


Figure 2.3. Develop Perspective (whilst debugging)

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 the registers within Peripherals.
- When debugging, the **Registers** view allows you to display the registers 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

- On the upper right 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 / Install SDK/ 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 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 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 / Outlines / Expressions 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.
 - 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 **Outlines** 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**).

3. Debug Solutions Overview

MCUXpresso IDE installs with built-in support for 3 debug solutions; comprising the Native LinkServer (CMSIS-DAP) [13] as used in LPCXpresso IDE. Plus support for both P&E Micro [20] and SEGGER J-Link. [21]

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 [59]

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. The user selects a project within the MCUXpresso IDE Project View
2. The user clicks Debug from within the MCUXpresso IDE **QuickStart** View
 - A probe discovery operation is automatically performed to display the available debug connections, including native, P&E and J-Link compatible probes.
3. The user selects the required debug probe and clicks OK
 - A project launch configuration is automatically created containing debug chain specific configurations
 - Launch configurations [11] 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 and watchpoints
- Stepping (single, step in step out etc.)
- Viewing and editing local variables, registers, peripherals, memory
- Viewing disassembly
- Semihosted IO

It is important to note that advanced operations such as the handling of launch configuration features will 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 TBD: Link
- 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 LinkServer 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, SEGGER, P&E) 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.

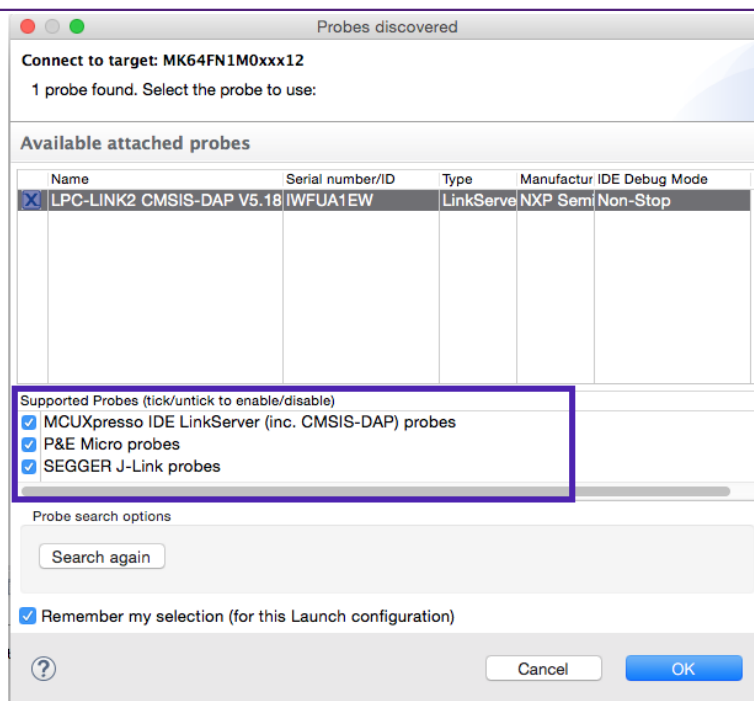


Figure 3.1. Debug Probe Discovery

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 (as shown below).

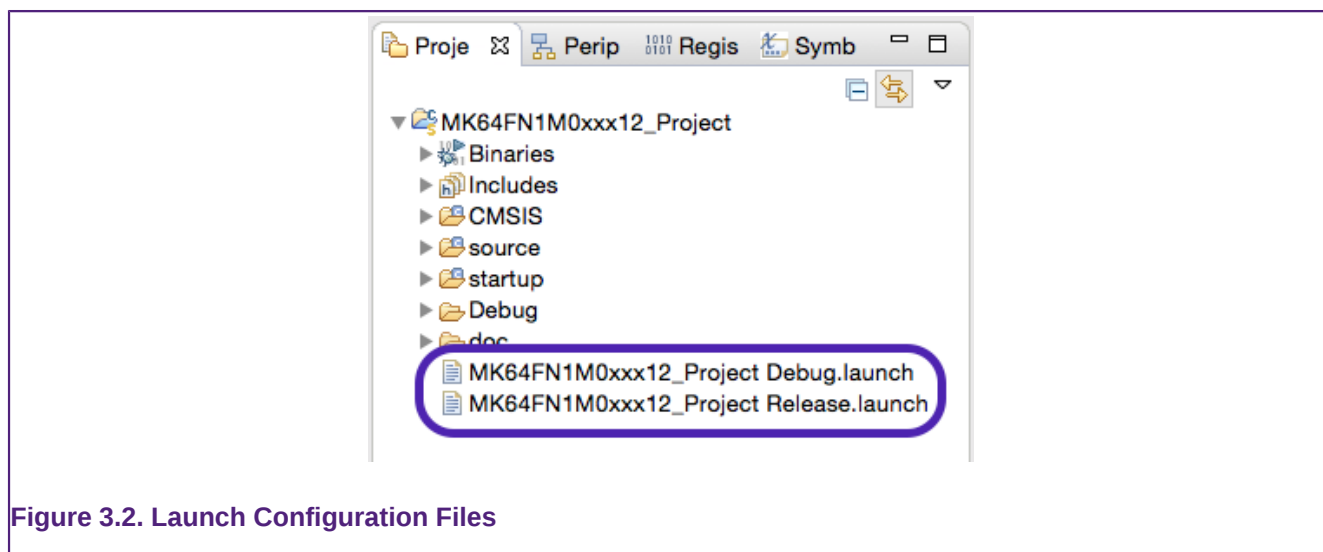


Figure 3.2. Launch Configuration Files

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.

One way to do this is as follows:

Select the Project (with a launch configuration file) within the 'Project Explorer' pane, then Right click on the Project and select: *Launch Configurations -> Edit Current*

To view the configuration items, be sure to select the *Debugger* tab as shown below:

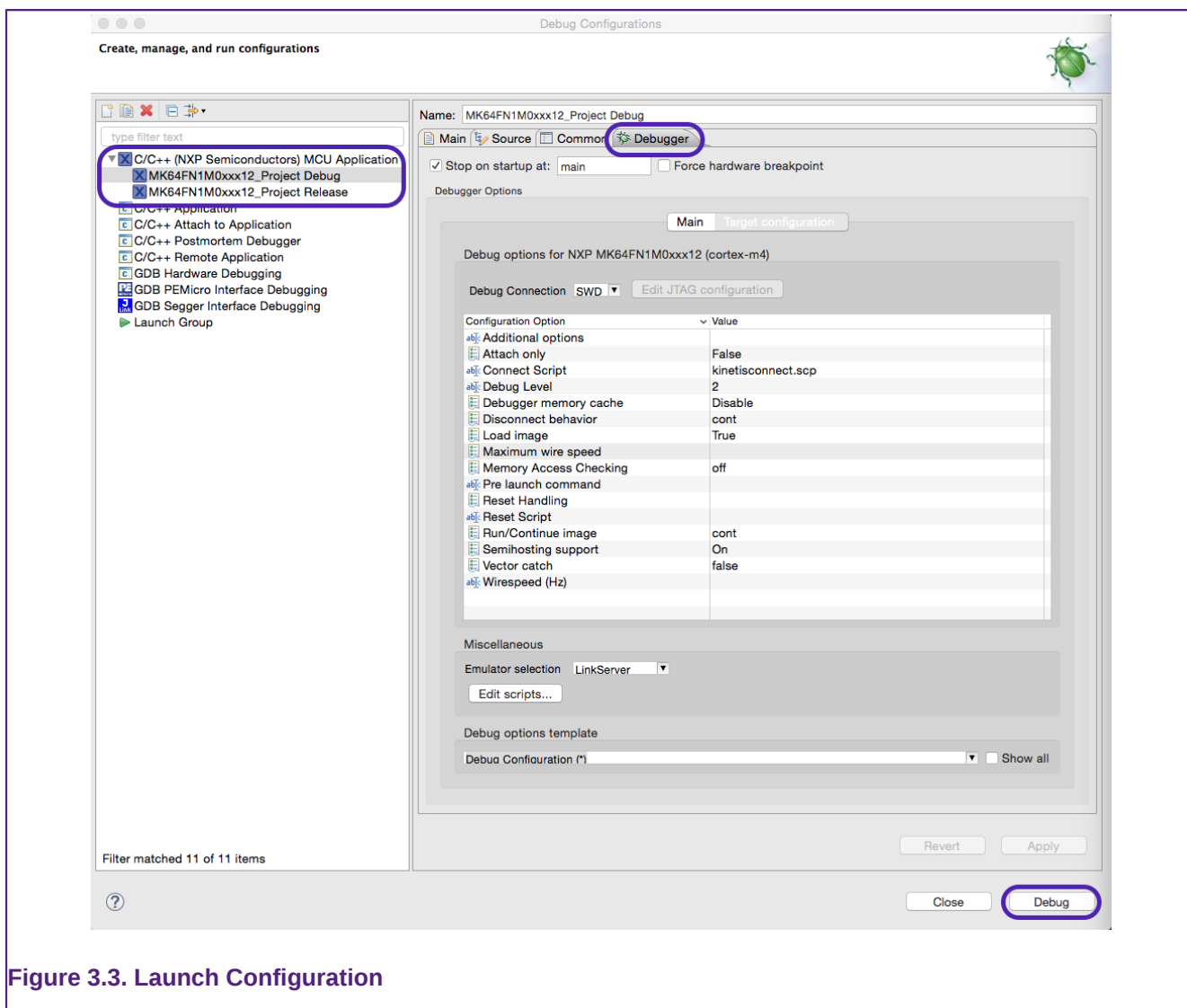


Figure 3.3. Launch Configuration

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.

Note: When exporting 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)).

Further information regarding Launch Configuration files can be found in the LPCXpresso IDE FAQ 'Launch Configuration Menu': <https://community.nxp.com/message/630714>

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 onboard debug probe hardware (as shipped by default on LPCXpresso MAX and CD boards)
 - For more information on LPCXpresso boards see: <http://www.nxp.com/pages/LPCXPRESSO-BOARDS>
- CMSIS-DAP firmware installed onto onboard 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/pages/OPENSDA>
 - Additional driver may be required:
 - <https://developer.mbed.org/handbook/Windows-serial-configuration>
- Keil uLINK with CMSIS-DAP formware: <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.

For LinkServer compatible debug probes, you can select from Non-Stop (the default) or All-Stop IDE debug mode.

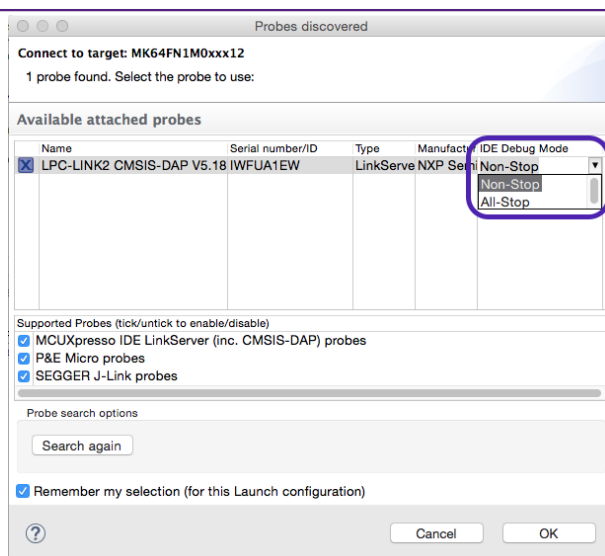


Figure 3.4. Debug Probe Discovery Non-Stop

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.

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 New Global Variables view.

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:

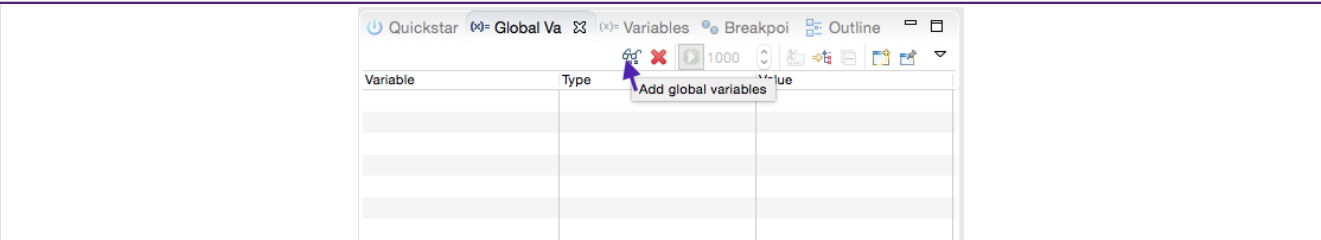


Figure 3.5. LinkServer Populate Globals

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 :

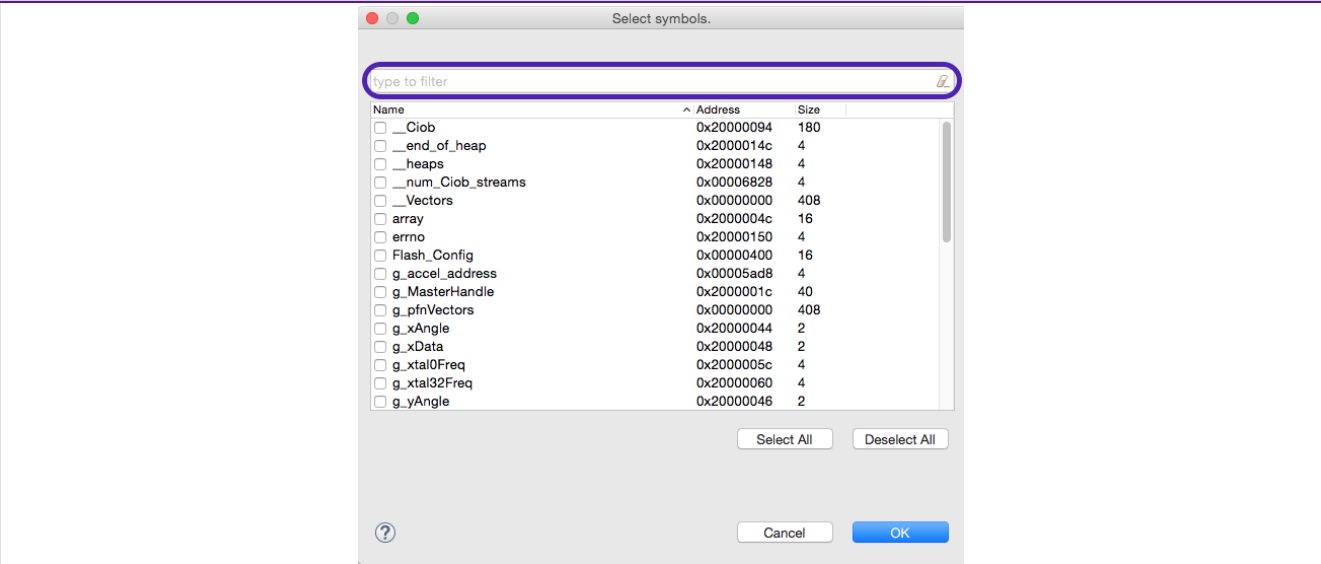


Figure 3.6. LinkServer Global Selector

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.

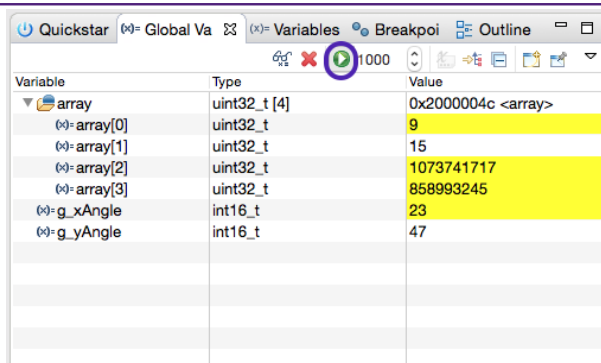
For “All-Stop” debug connections, the Global Variables view will be updated whenever the target is paused.

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

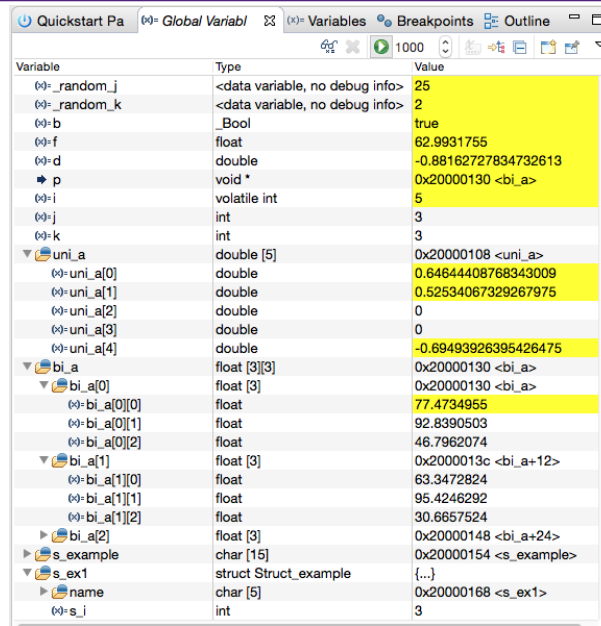
Figure 3.7. LinkServer Globals Display

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.

MCUXpresso IDE defaults to the selection of “Non-Stop” mode when a probe discovery operation is performed.

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

The usefulness of **Live Variables** reduces as the number 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]	0x20000130 <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 Globals Display Many

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

Preferences -> Debug Options (Misc)

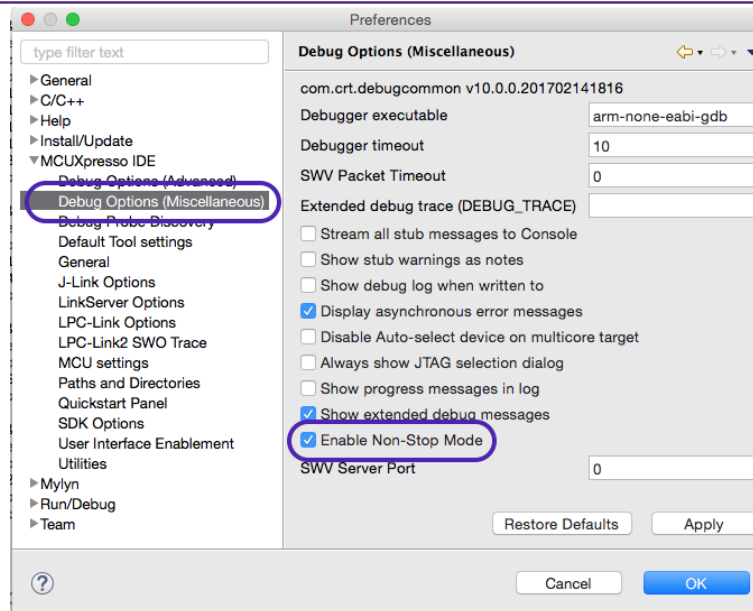


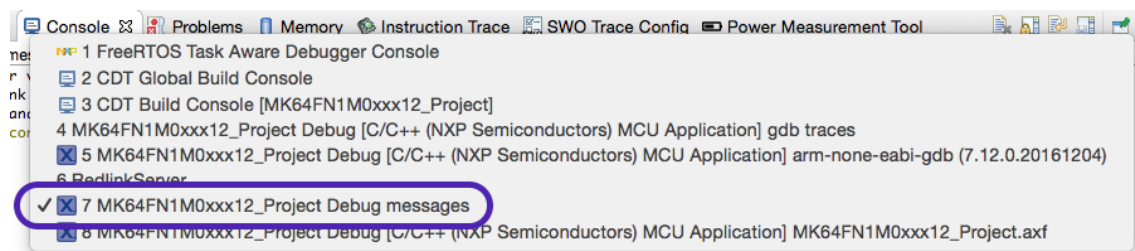
Figure 3.9. LinkServer Non Stop Preference

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. For more details of launch configurations, see <https://community.nxp.com/message/630714>.

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 with 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):



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. These Connect scripts are required for debugging certain parts and are automatically added to launch configuration files by the IDE if required.

Further down in this file 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 (81.97 KB/sec).

```
MCUXpresso RedlinkMulti Driver v10.0 (Feb 16 2017 18:37:08 - crt_emu_cm_redlink
build 175)
Reconnected to existing redlink server (PID -1)
Connecting to core 0 (probe handle 1, PID -1) 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 V5.181 (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 Jan 13 2017 16:14:07'
Opening flash driver FTFE_4K.cfx
flash variant 'K 64 FTFE Generic 4K' detected (1MB = 256*4K at 0x0)
Closing flash driver FTFE_4K.cfx
NXP: MK64FN1M0xxx12
( 65) Chip Setup Complete
Connected: was_reset=true. was_stopped=true
MCUXpressoPro Full License - Unlimited
Awaiting telnet connection on port 3330 ...
GDB nonstop mode enabled
Opening flash driver FTFE_4K.cfx (already resident)
Writing 10744 bytes to address 0x00000000 in Flash
Erased/Wrote page 0-2 with 10744 bytes in 128msec
Closing flash driver FTFE_4K.cfx
Flash Write Done
Flash Program Summary: 10744 bytes in 0.13 seconds (81.97 KB/sec)
Stopped: Breakpoint #1

```

3.5.2 Flash Programming issues

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

Flash programming common programming 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).
1. 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.
1. 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 related to their use of flash configuration block at offset 0x400. For more information please see: Kinetis MCUs Flash Configuration Block [81] . Sector sizes on Kinetis MCUs range from 1KB to 8KB, therefore the first Sector Erase performed will clear the value of this block to 0xFFFFFFFFFFFFFFFF, if this is not followed by a successful program operation and the part is reset, then it will likely report as 'Secured' and programming will not be possible unless the part is recovered.

Such an event can occur if a debug operation is accidentally performed to the 'wrong board'. Where the wrong flash programmer has been invoked.

TBD: The Kinetis Mass Erase Script can be used as a connect script to deal with locked parts. This is harmless but does have the consequence of erasing the entire flash when a connection is made.

TBD: The GUI flash programmer running the Kinetis Recover Operaiton

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 CPUs
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 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 onboard OpenSDA debug probe hardware (as shipped by default on certain Kinetis FRDM and TWR boards)

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.

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 exists.

Note: If the project already had a 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, it may be difficult for the user to determine whether the problem relates to the IDE or to P&E lower level code.

See TBD: CUSTOMER SUPPORT HANDLING

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.

Note: LinkServer 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.

TBD: Do we want to do any more for P&E

3.8 SEGGER Debug Connections

SEGGER J-Link software and documentation pack is installed automatically with the MCUXpresso IDE Installation for each host platform. Unlike other debug solutions, this is not integrated into the IDE installation, rather a separate SEGGER J-Link installation will be installed onto your host.

MCUXpressoIDE automatically locates the required executables and so no further setup is required to debug with SEGGER J-Link.

Currently we have tested using:

- J-Link debug probes (USB and Ethernet)
- J-Link firmware installed into onboard OpenSDA debug probe hardware (as shipped by default on certain Kinetis FRDM and TWR boards)

From time to time, SEGGER may release later versions of their software. For MCUXpresso to use a different version of software from that originally installed, the user can update the internal linkage.

The installation location will be similar to:

On Windows: *'C:/Program Files (x86)/SEGGER/JLink_V612c/jLinkGDBServerCL.exe'* On Mac: *'/Applications/SEGGER/JLink_V612c/JLinkGDBServer'*

Updating this path allows the SEGGER tools to perform a probe discovery operation i.e. check whether a SEGGER compatible debug probe is connected with an updated installation.

Once this has been set, it will be remembered as a Workspace preference. This can be viewed or edited within the MCUXpresso IDE preferences as below.

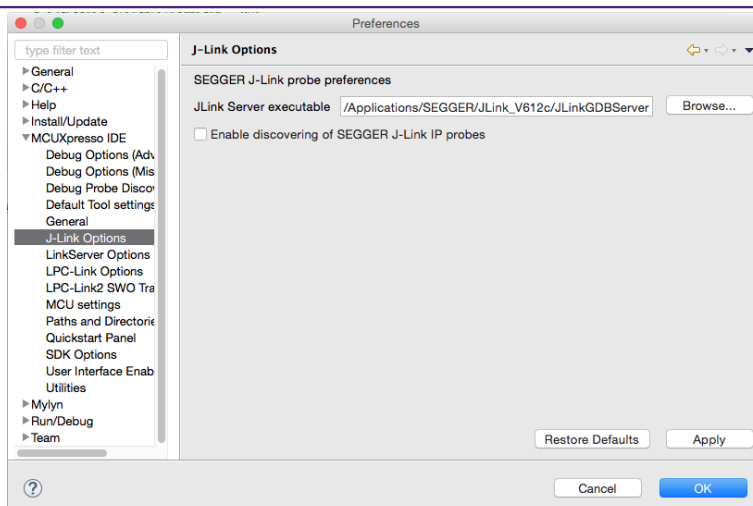


Figure 3.10. Segger Preferences

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.

To avoid having to reselect this new path every time a new workspace is chosen, 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_V612c/jLinkGDBServerCL.exe"
```

This value will then automatically populate the workspace settings for each new workspace overriding the installation default.

3.9 SEGGER Debug Operation

The process to debug via a J-Link 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.

Note: MCUXpresso IDE can only discover a single J-Link probe connected over USB. TBD: is this still true?

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: If the project already had a 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, it may be difficult for the user to determine whether the problem relates to the IDE or to SEGGER lower level code.

See TBD: CUSTOMER SUPPORT HANDLING

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.

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

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:

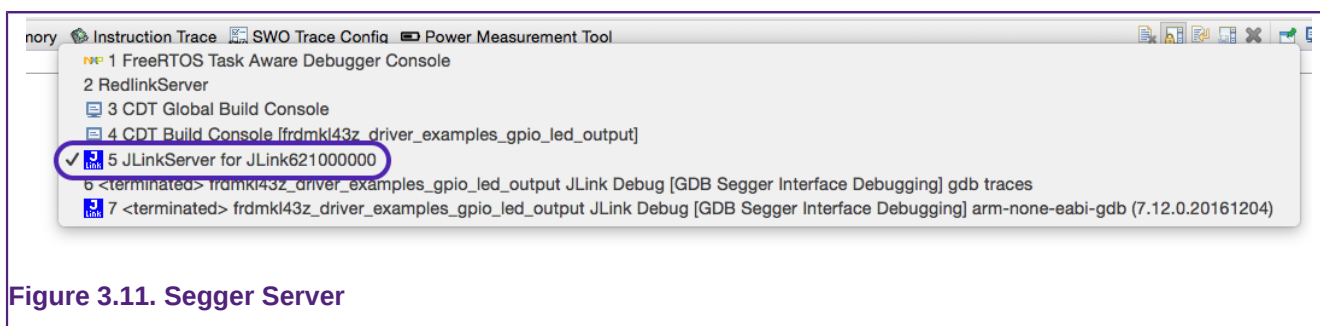


Figure 3.11. Segger Server

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

Below is the output of a successful debug session to a Kinetis KL43 Freedom Board.

```
Executing Server: /Applications/SEGGER/JLink_V612c/JLinkGDBServer -nosilent
-swoport 2332 -select USB -telnetport 2333 -endian little -noir -speed auto
-port 2331 -vd -device MKL43Z256xxx4 -if SWD -nohalt -reportuseraction

SEGGER J-Link GDB Server V6.12c Command Line Version

JLinkARM.dll V6.12c (DLL compiled Dec 16 2016 17:26:51)

-----GDB Server start settings-----
GDBInit file:                none
GDB Server Listening port:    2331
SWO raw output listening port: 2332
Terminal I/O port:           2333
Accept remote connection:    yes
Generate logfile:             off
Verify download:              on
Init regs on start:           off
Silent mode:                  off
Single run mode:              off
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:                 MKL43Z256xxx4
Target interface:              SWD
Target interface speed:        auto
Target endian:                 little

Connecting to J-Link...
J-Link is connected.
Firmware: J-Link OpenSDA compiled Nov 16 2016 09:42:59
Hardware: V1.00
S/N: 621000000
Checking target voltage...
Target voltage: 3.30 V
Listening on TCP/IP port 2331
Connecting to target...Connected to target
Waiting for GDB connection...Connected to 127.0.0.1
Reading all registers
Read 4 bytes @ address 0x00002DCC (Data = 0xB004BEAB)
Read 2 bytes @ address 0x00002DCC (Data = 0xBEAB)
Read 2 bytes @ address 0x00002DC6 (Data = 0xB40F)
Read 2 bytes @ address 0x00002DC8 (Data = 0x4669)
Read 2 bytes @ address 0x00002DCA (Data = 0x2005)
Resetting target
Semi-hosting enabled (Handle on BKPT)
Downloading 14792 bytes @ address 0x00000000 - Verified OK
Downloading 4 bytes @ address 0x000039C8 - Verified OK
Writing register (PC = 0x14010000)
Read 4 bytes @ address 0x00000114 (Data = 0xB672B510)
Read 2 bytes @ address 0x0000064E (Data = 0x1D3B)
Read 2 bytes @ address 0x0000064E (Data = 0x1D3B)
Read 2 bytes @ address 0x0000064E (Data = 0x1D3B)
Setting breakpoint @ address 0x0000064E, Size = 2, BPHandle = 0x0001
Starting target CPU...
...Breakpoint reached @ address 0x0000064E
Reading all registers
Read 4 bytes @ address 0x0000064E (Data = 0x22011D3B)
Removing breakpoint @ address 0x0000064E, Size = 2
```

4. SDKs and Pre-Installed 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 MCUs 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 Pre-installed Part Support

Firstly the IDE installs with an enhanced version of the part support as provided with LPCXpresso IDE v 8.2.2. This provides support for the majority of LPC parts 'out of the box'. This is known as pre-installed part support.

Example code for these pre-installed 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. The latest version of these are installed by default at:

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

4.2 SDK Part Support

Secondly, MCUXpresso IDE's part support can be extended using freely available MCUXpresso SDK2.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 all Kinetis parts is delivered by SDK2.x packages, in addition this mechanism will be used to offer support for new LPC MCUs from NXP such as the LPC5640x.

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

4.2.1 Important notes for SDK users

Only SDKs created for MCUXpresso IDE can be used

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

Shared Part Support handling

Due to the nature of SDK packages, it is possible to have different packages containing identical 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 only select the most up to date version of part support specified within these SDKs. This means the various wizards and dialogues will present a single MCU, but offer a variety of compatible boards and examples.

Building a 'Fat' SDK

An SDK can be generated for a selected part 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 board i.e. MK64...
- board support packages for the above part i.e. Freedom and 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.

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.

4.2.2 Differences in Pre-installed and SDK part handling

Since SDKs bundle part and board support into a single package, MCUXpresso IDE is able to provide graphical linkage between SDK MCUs and their related boards when creating or importing projects. For pre-installed 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 Pre-Installed and SDK part support is described in the following chapters.

4.3 Viewing Pre-installed Part Support

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

To explore the range of pre-installed MCUs simply click 'New project' in the **QuickStart panel**. This will open a page similar to the image below:

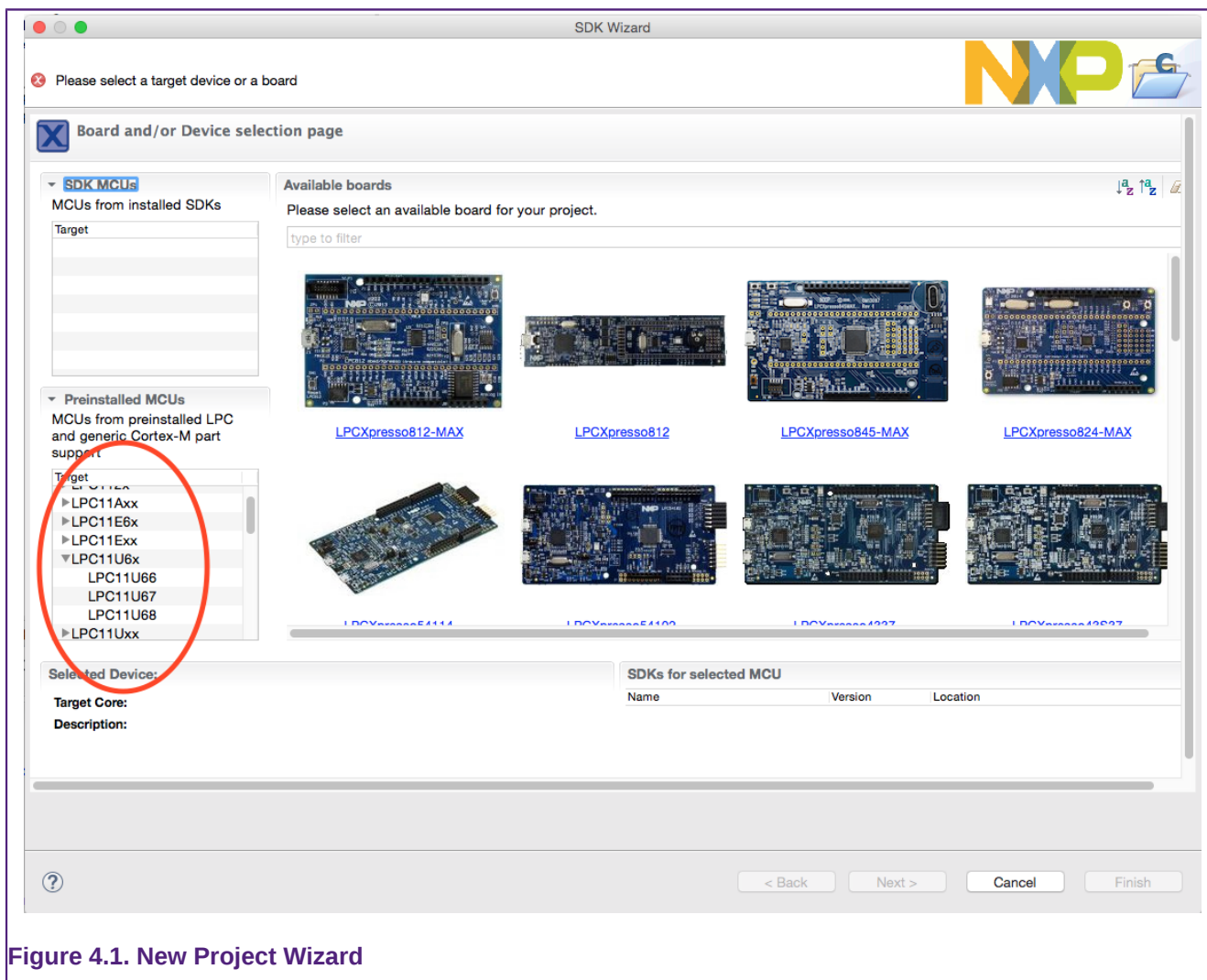


Figure 4.1. New Project Wizard

The list of pre-installed 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 creating project with Pre-Installed part support please see: Creating Projects with Pre-Installed part support) [47]

If you intend to work on an MCU that is not available from the range of Pre-Installed parts for example a Kinetis MCU then you must first extend the part support with MCUXpresso IDE with the required 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).

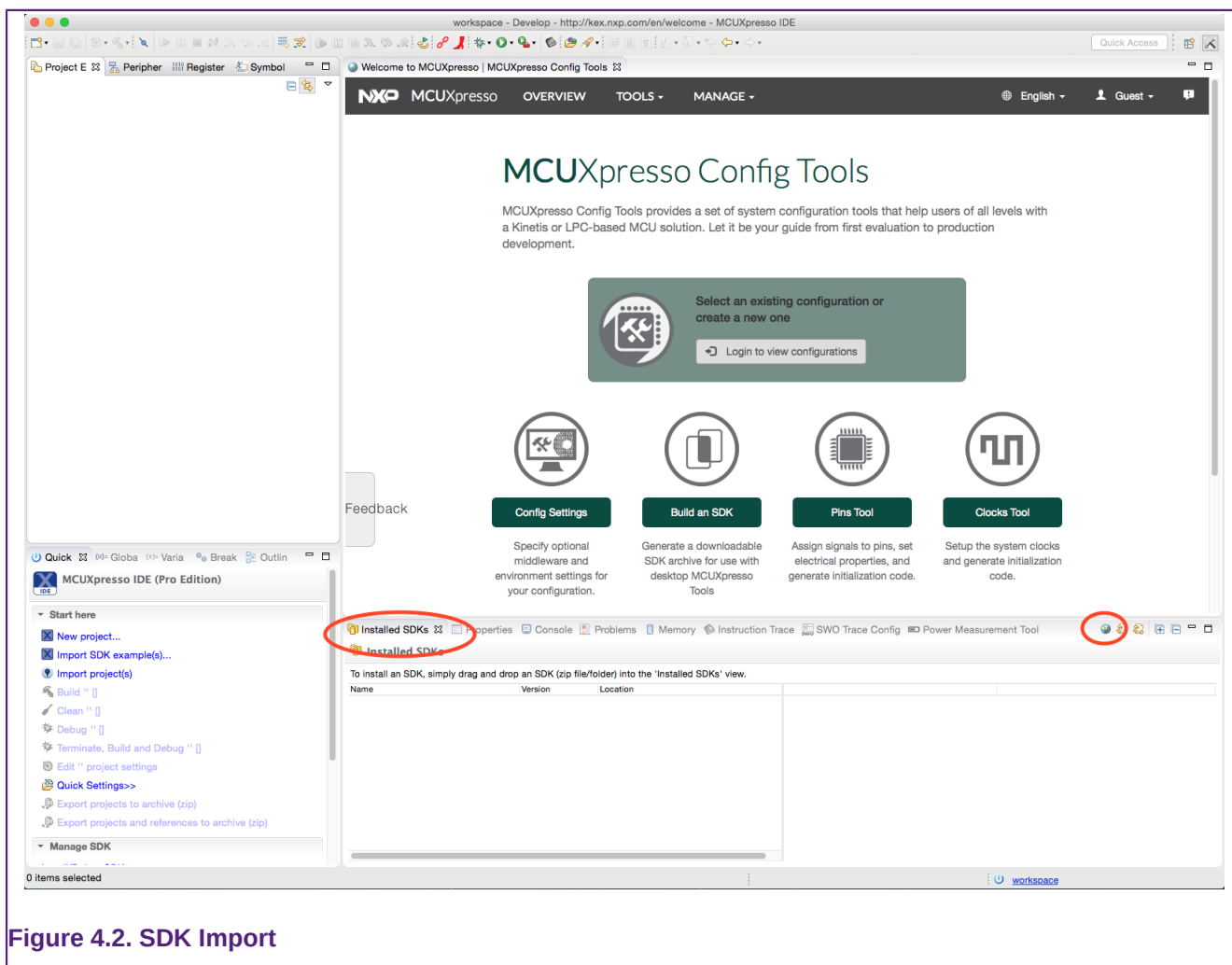


Figure 4.2. SDK Import

SDKs are free to download; 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 the SDK, simply open a Windows Explorer / file explorer 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 will then be automatically installed into MCUXpresso IDE part support repository.

Notes:

- 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.
- By using the Drag & Drop mechanism for installing the SDK into the default location, you make a copy of that SDK that can be shared among different IDE instances/installations and workspaces.

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

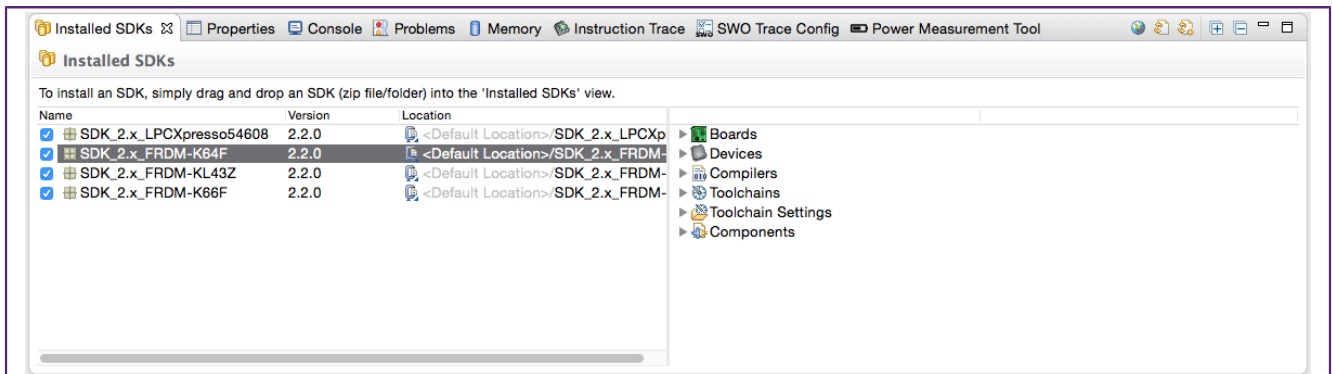
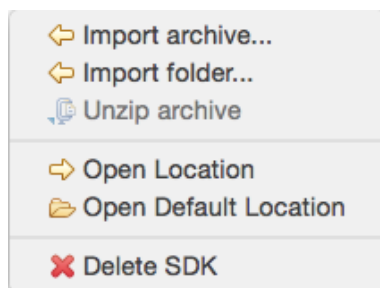


Figure 4.3. SDK Import View

The display will show whether the SDKs are stored as zipped folders or not. 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 reference files or perform many example imports (where some speed improvement will be seen). See

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

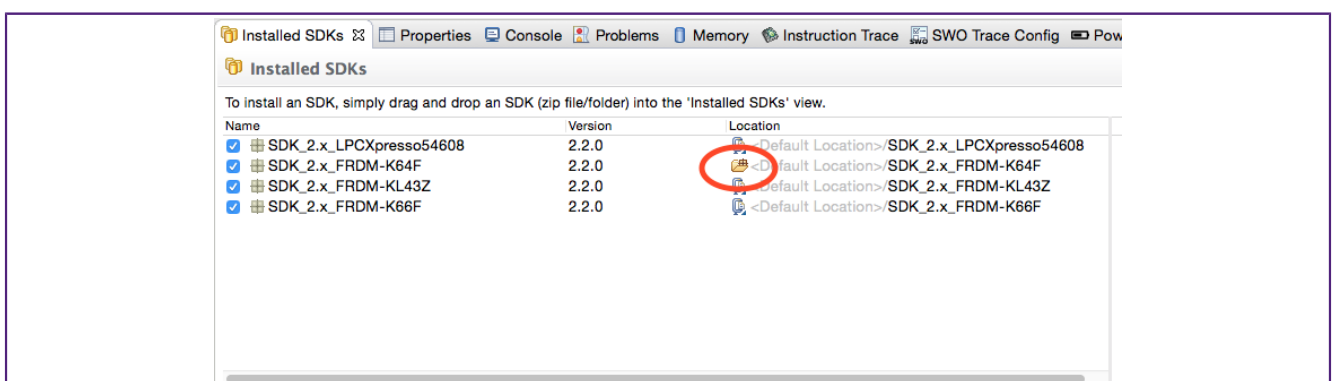


Figure 4.4. SDK Unzipped

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

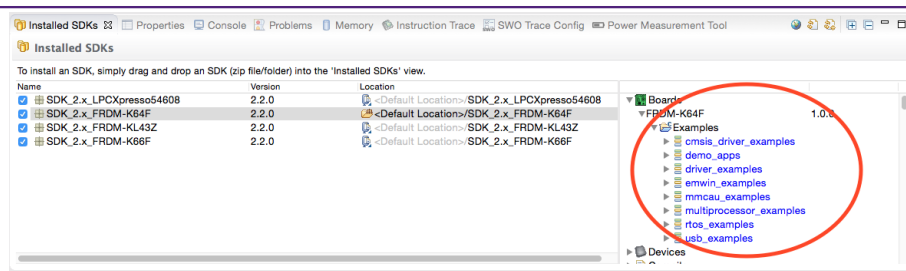


Figure 4.5. SDK Explore

4.4.1 “Power User” 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 Location* or click on the shortcut “Use existing SDK...” in the **QuickStart** view then the following window will appear:

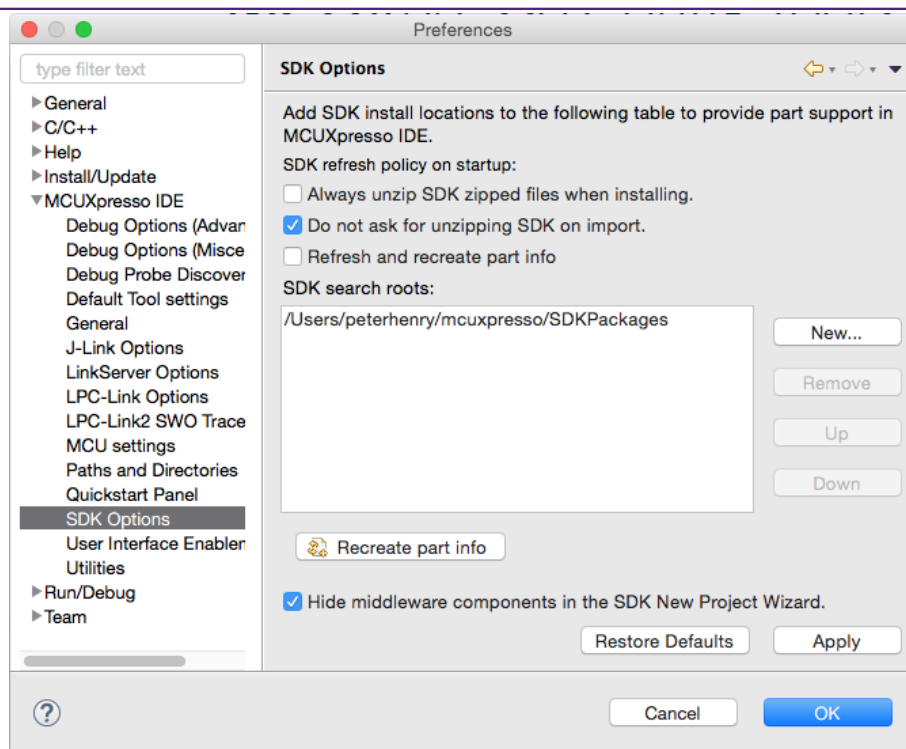


Figure 4.6. SDK Preferences

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 (folders are scanned recursively):

- “Delete SDK” function is disabled when using non-default locations
 - to avoid accidentally deleting an original file
- 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 more 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.

5. Creating New Projects using SDKs

For creating project with Pre-Installed part support please see: Creating Projects with Pre-Installed part support [47]

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

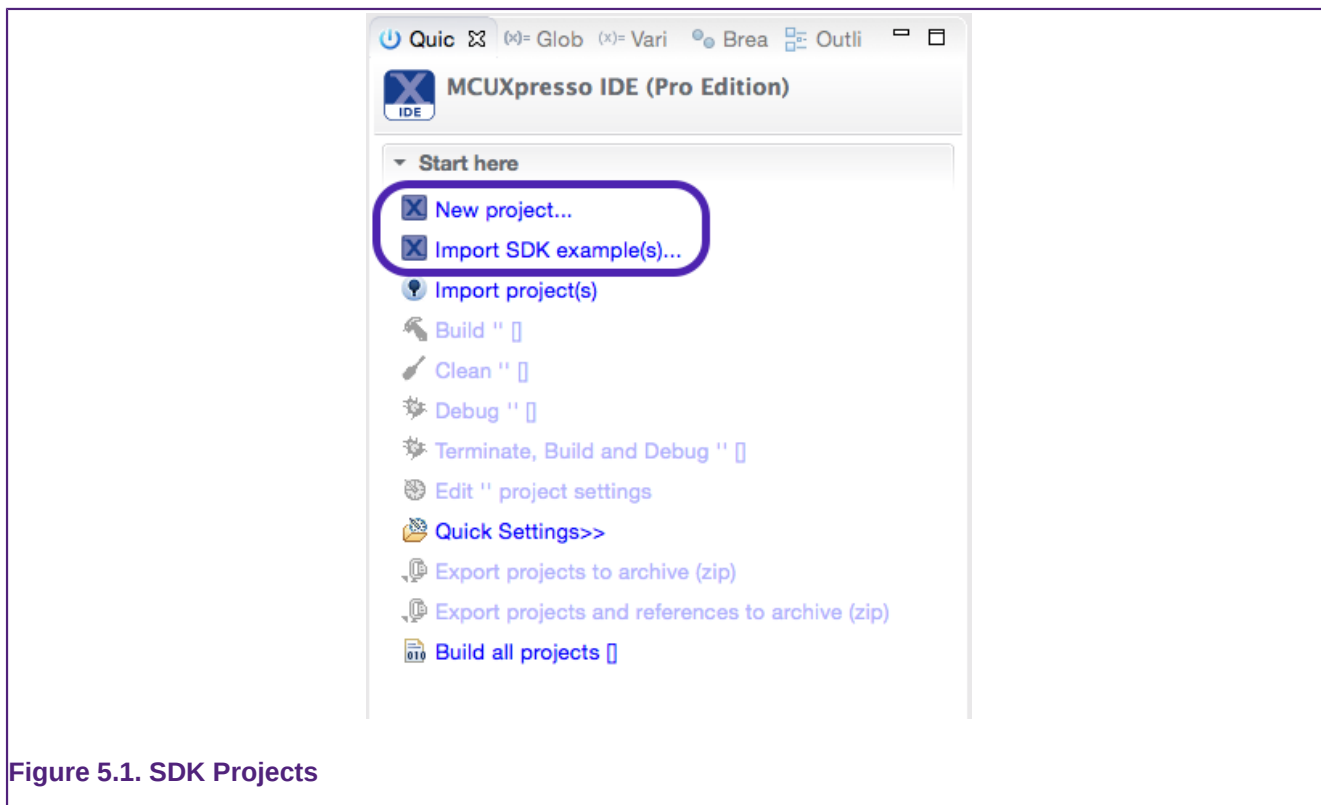


Figure 5.1. SDK Projects

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

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

TDB: 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:

1. A display of all parts (MCUs) installed via *SDK*. Click to select the MCU and filter the available matching boards. SDK part support can be hidden by clicking on the triangle (highlighted in blue)
2. A display of all pre-installed parts (these are all LPC or Generic M parts). Click to select the MCU and filter the available matching boards (if any). Pre-Installed part support can be hidden by clicking on the triangle (highlighted in blue)

3. A display of all boards from both *SDK* 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 or Error 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).

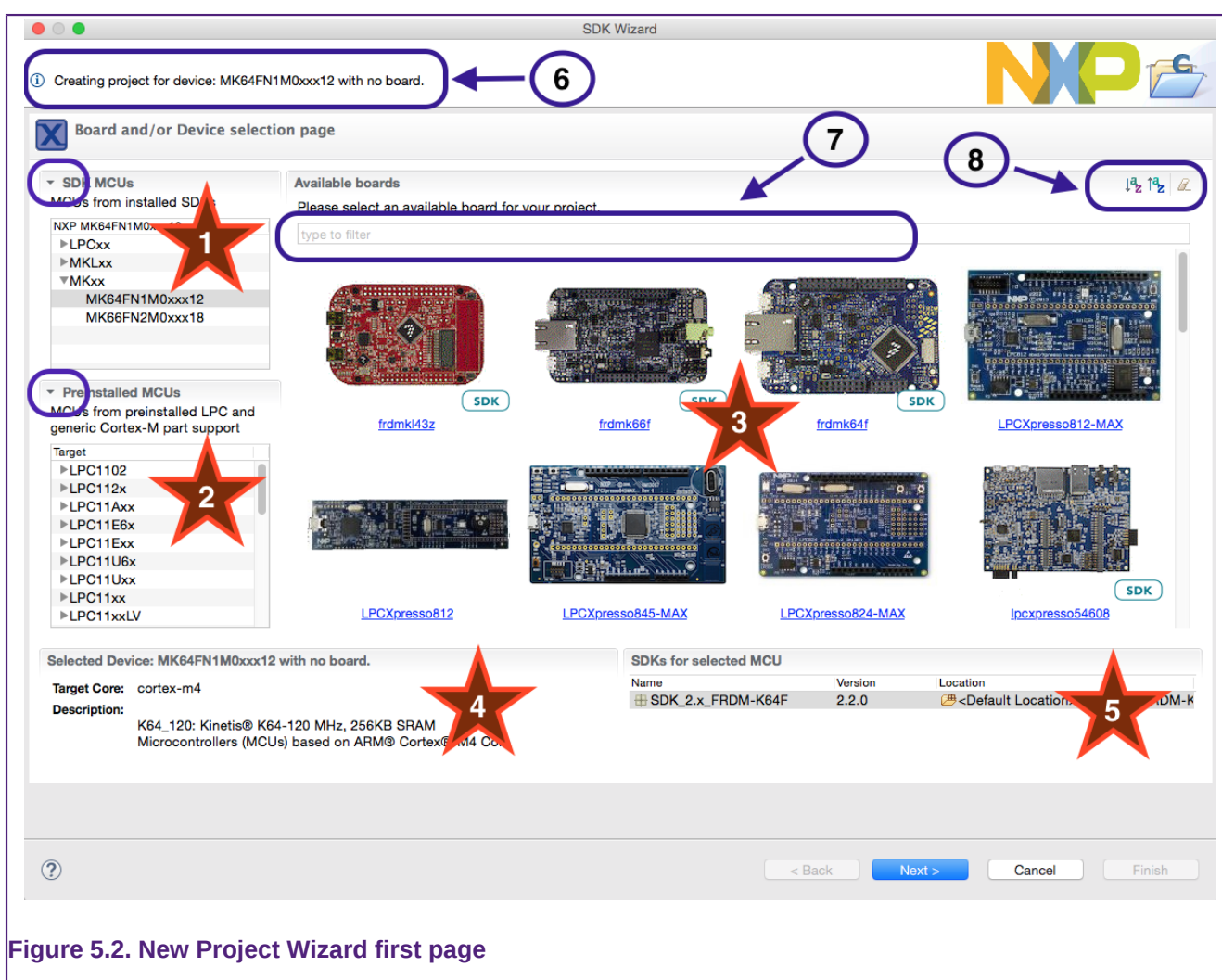


Figure 5.2. New Project Wizard first page

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 MK64.... 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.

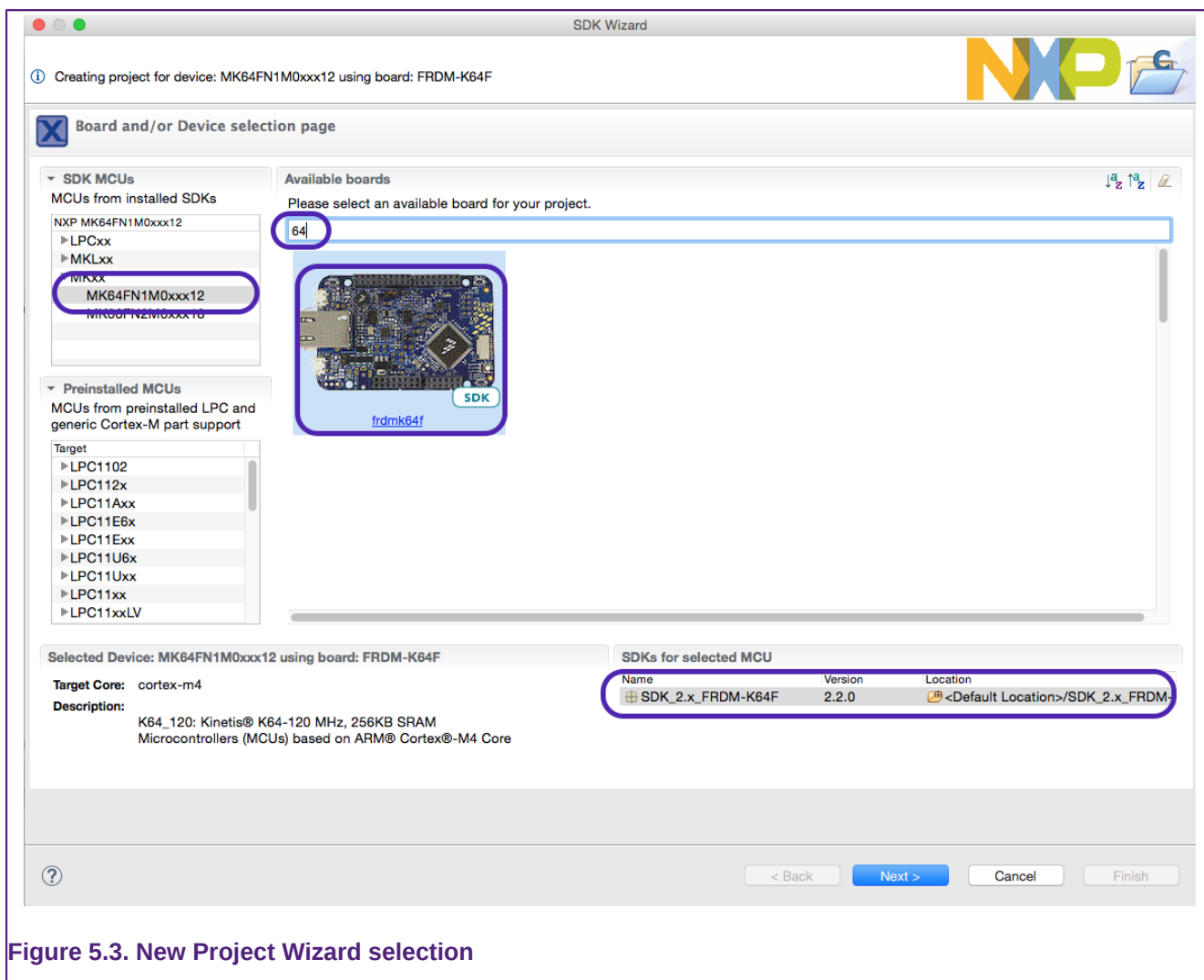


Figure 5.3. New Project Wizard selection

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 sensible default options (the default options offered on the advanced page may be set based on user 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 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.

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.

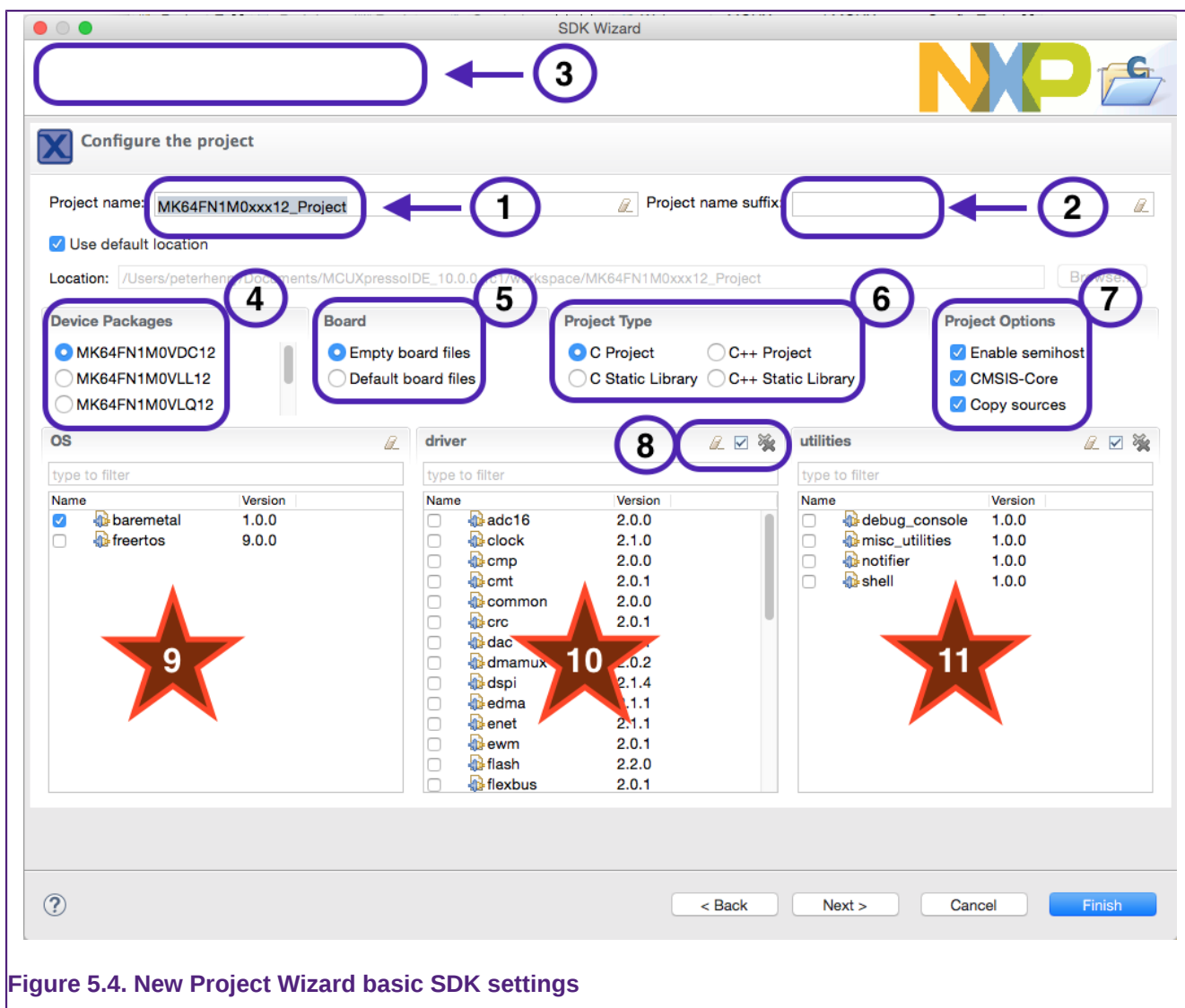


Figure 5.4. New Project Wizard basic SDK settings

1. Project Name: The default project name prefix is automatically selected based on the part selected on the previous screen
2. Project Suffix: An optional suffix to append to a project name can be entered here
3. Error and Warnings: Any error or warning such as will be displayed here. The 'Next' option will not be available until any warning or 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 be 'Greyed' out.
6. Project Type: C or C++ projects or libraries can be selected. Selecting 'C' will automatically select RedLib libraries, selecting C++ will select Newlib-Nano libraries. TDB: link to later

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 sources such as FreeRTOS. TDB: better description
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 direct program output via a UART.
- the debug console option relies on the OpenSDA debug probe communicating to the host via VCOM over USB. TBD: can we do something better here....

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).

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 configure project page, for example a C project will pre-select Redlib, where as a C++ project will pre-select Newlib.

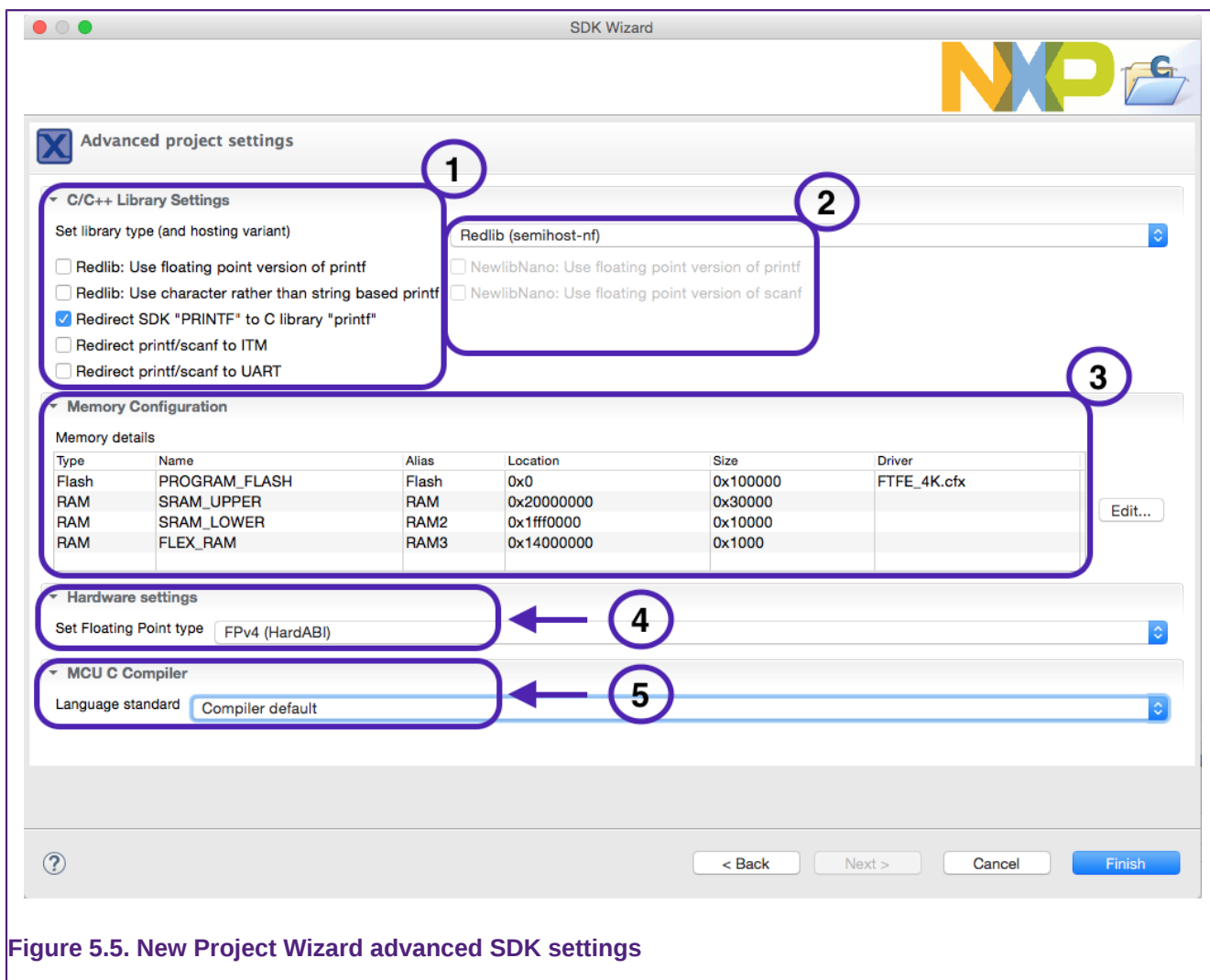
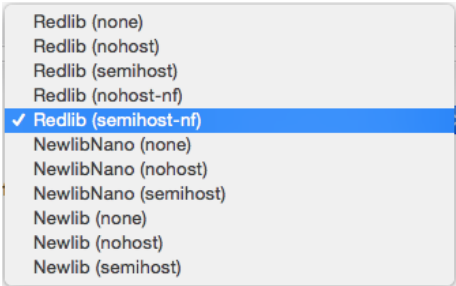


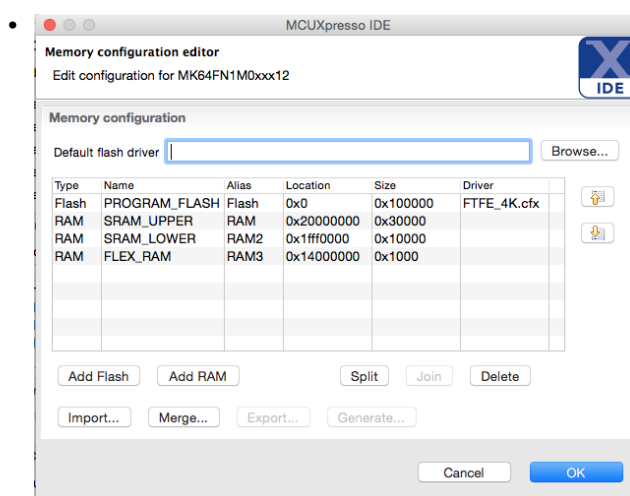
Figure 5.5. New Project Wizard advanced SDK settings

1. This panel allows options to be set related to Input/Output. **Note** if a C++ project was selected on the previous page, then the Redlib options will be Greyed out. See C/C++ Library Support [63]
 - 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 printing out of floating point variables at the expense of larger library support code.
 - Redlib use Character printf: selecting this option will avoid heap usage and reduce code size but make printf operations slower
 - Redirect SDK "PRINTF": many SDK examples use a PRINTF macro, selecting this option select 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. TBD: link to more info
 - Redirect printf/scanf to UART: Sets the define SDK_DEBUGCONSOLE_UART causing the SDK debug console routines to output via a real UART rather than over VCOM.
2. This panel allows the selection of various library variants. See C/C++ Library Support [63]

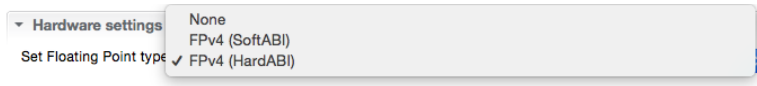
- 

3. **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.

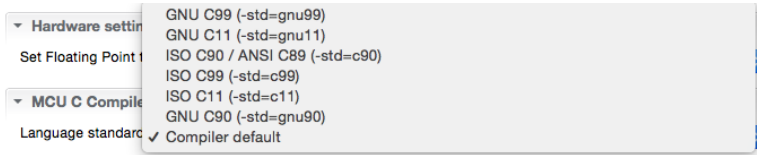
- Clicking Edit invokes the IDE's memory configuration editor. From this dialogue, the project's default memory setting and hence automatically generated linker settings can be changed. See Memory Configuration and Linker Scripts [73]



4. **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.

- 

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

- 

Notes:

- It is recommended that you always select the “CMSIS-Core” project option. Without this being included in your generated project, other SDK components that you select may not successfully build.
- Copy sources is selected by default; this ensures that the newly created project will be standalone i.e. there will be no linked references to the original files within the SDK.

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 the FAQ at

<https://community.nxp.com/message/630628>

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

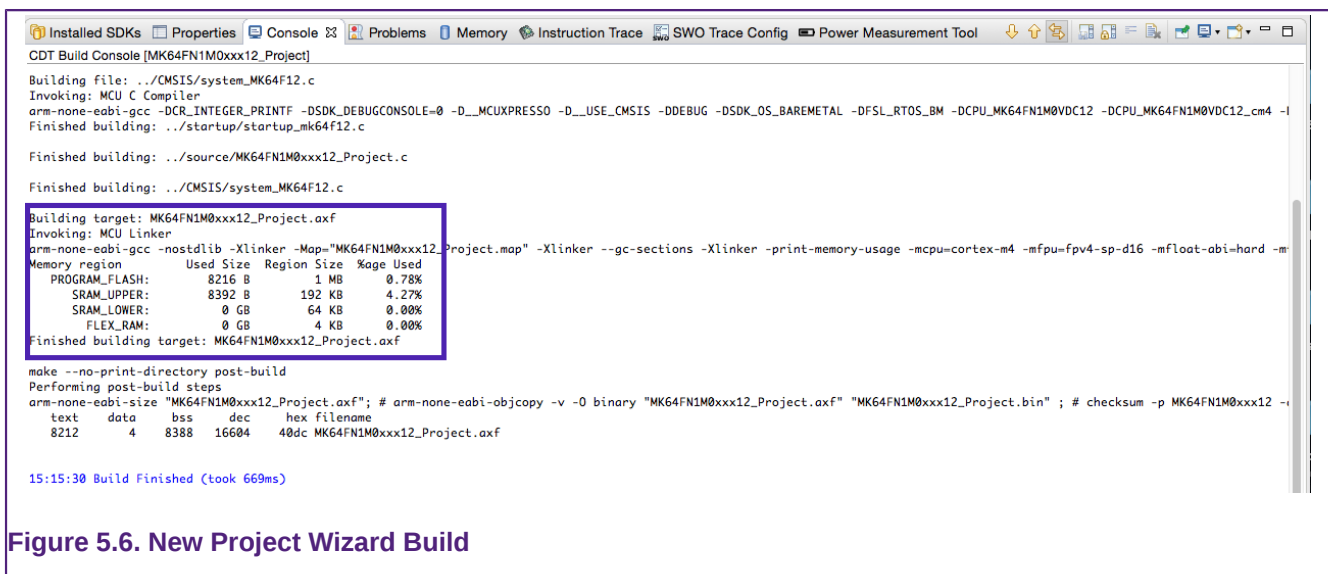


Figure 5.6. New Project Wizard Build

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 1 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 default for this build has reserved 4KB each for the stack and the heap. Please See Memory Configuration and Linker Scripts [73] for detailed information.

6. Importing Example Projects (from SDKs)

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

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

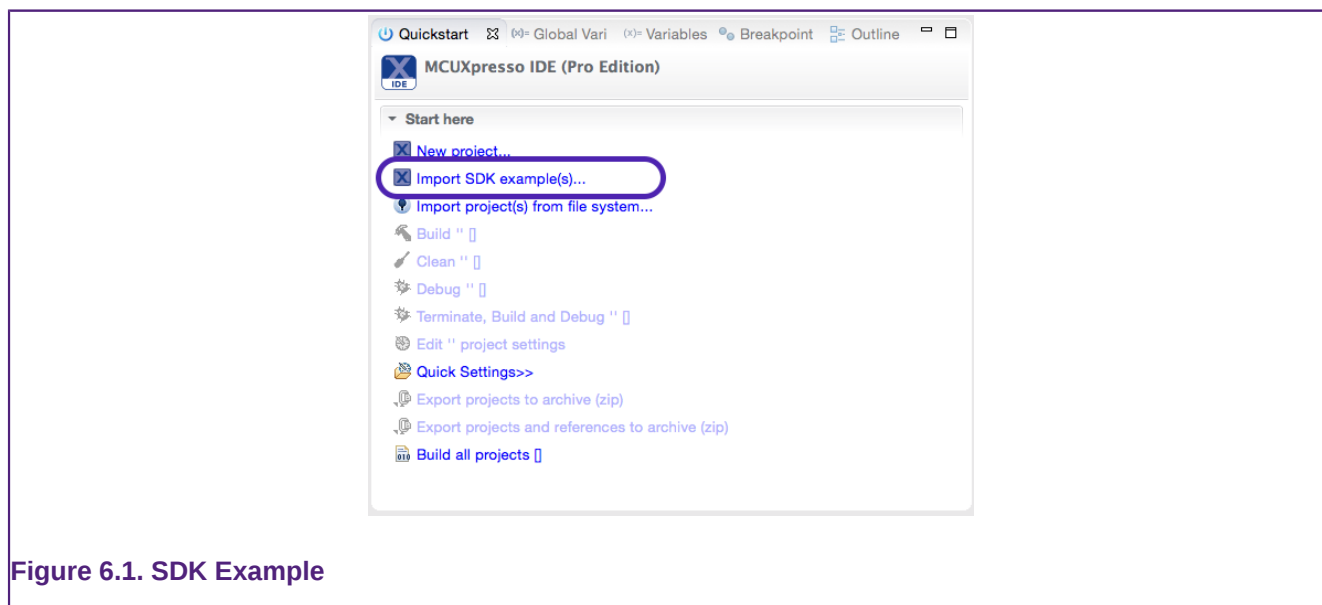


Figure 6.1. SDK Example

This option invokes the **SDK Import 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.

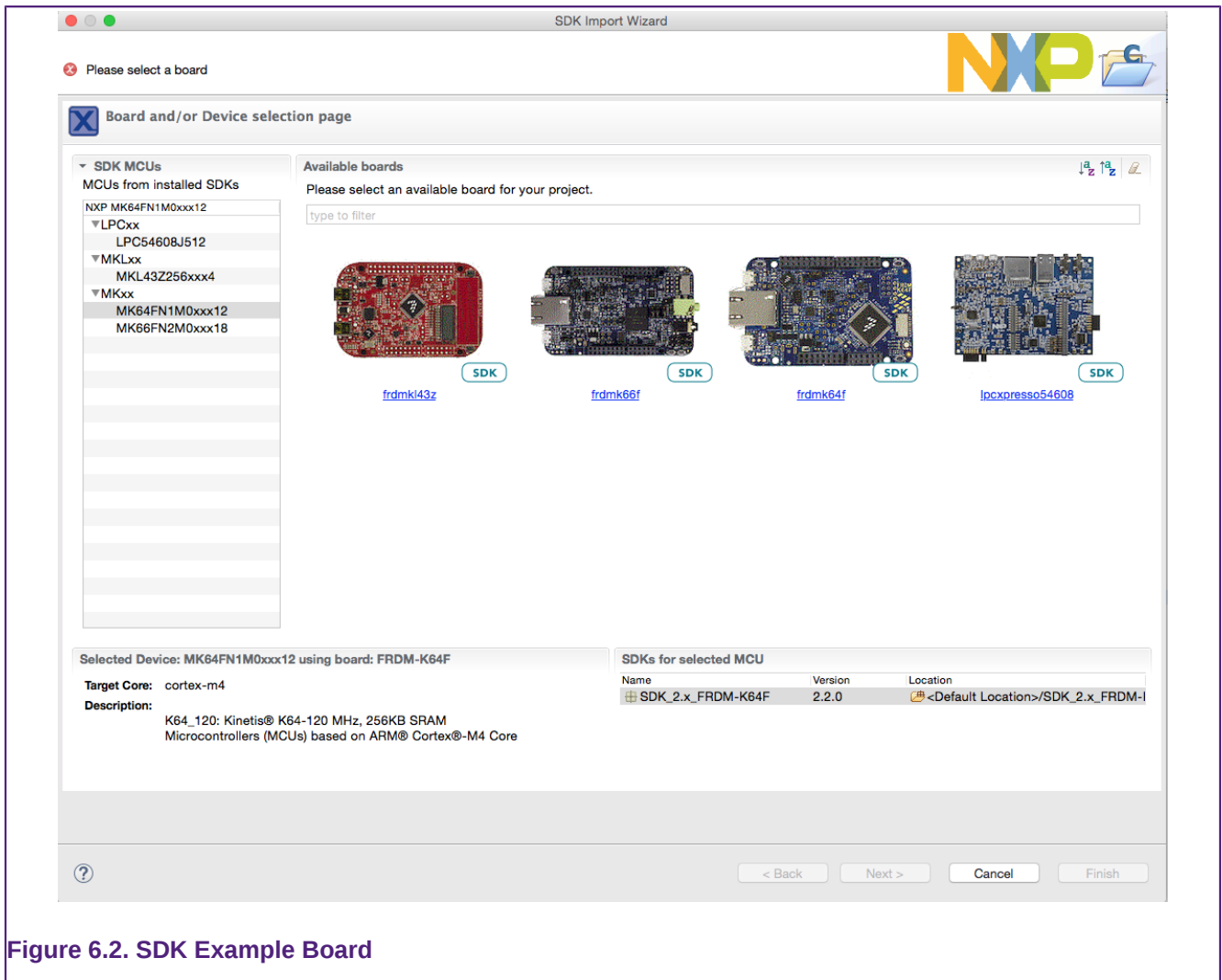


Figure 6.2. SDK Example Board

6.1 SDK Example Import Wizard

Selection and filtering work in the same way as for the “New Project Wizard”:#newprojectwizard but note, examples are created for particular development boards 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 setting 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 setting 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.

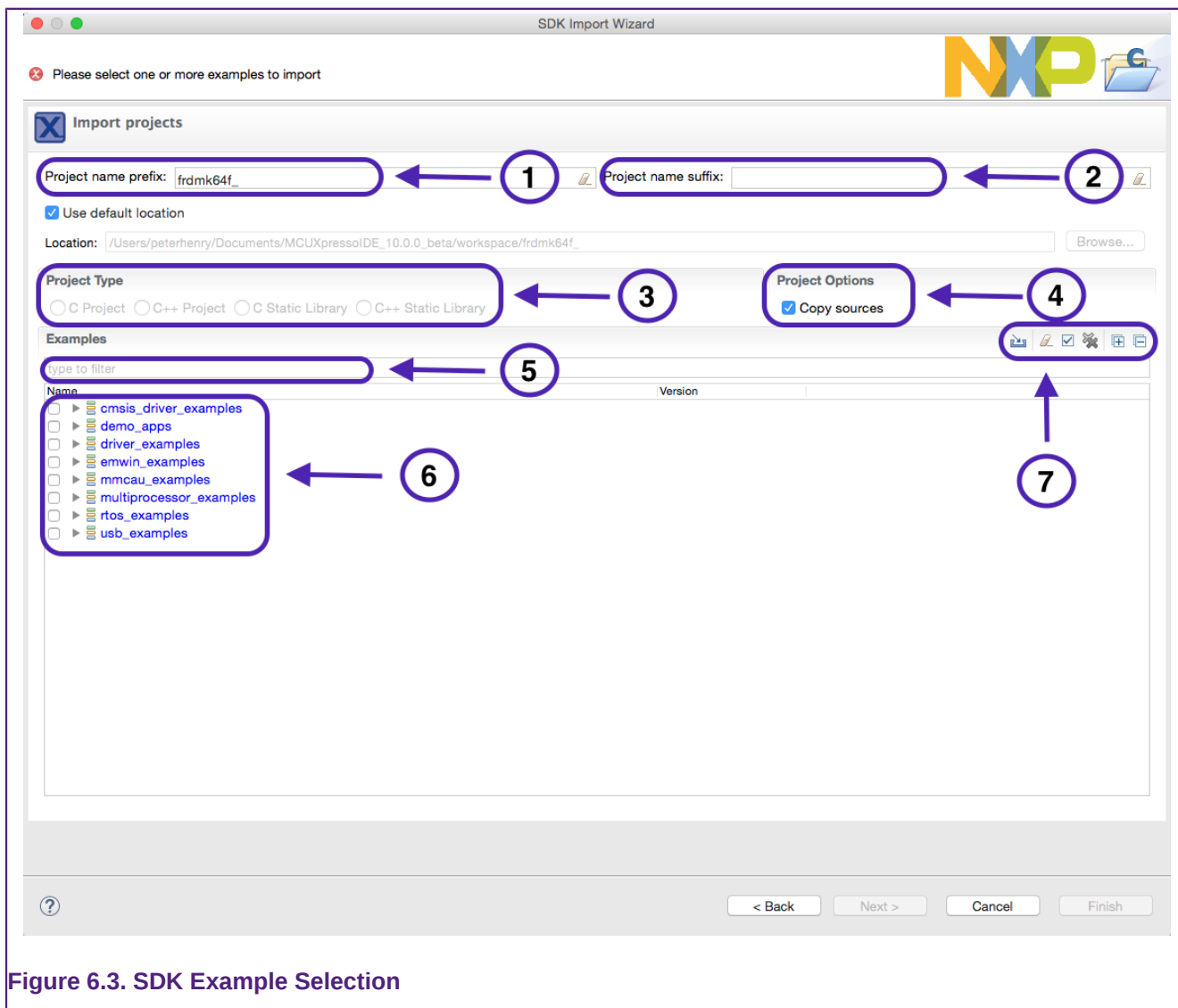


Figure 6.3. SDK Example Selection

1. Project Name: A project name is automatically created with a name of the form: `\prefix_\SDK example path_\example name_\suffix`. The ability to add a suffix allows an import to be repeated but project names easily made unique.
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...
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: For unzipped SDKs, you can untick this option to create project containing links to the original SDK files. This option should only be unticked with care, since editing linked example source will overwrite the original files.
5. Examples Filter: Enter text into this field to select matches for example 'LED', 'bubble' will select common examples from the set. 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:

- Opens a filter 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 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, 185 is indicated for the FRDM MK64 SDK example shown below. Importing many examples will take time ... Consider that each example may consist of 20 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, Eclipse workspaces do not cope well with 100+ projects, therefore it is suggested that example imports are limited to sensible numbers.

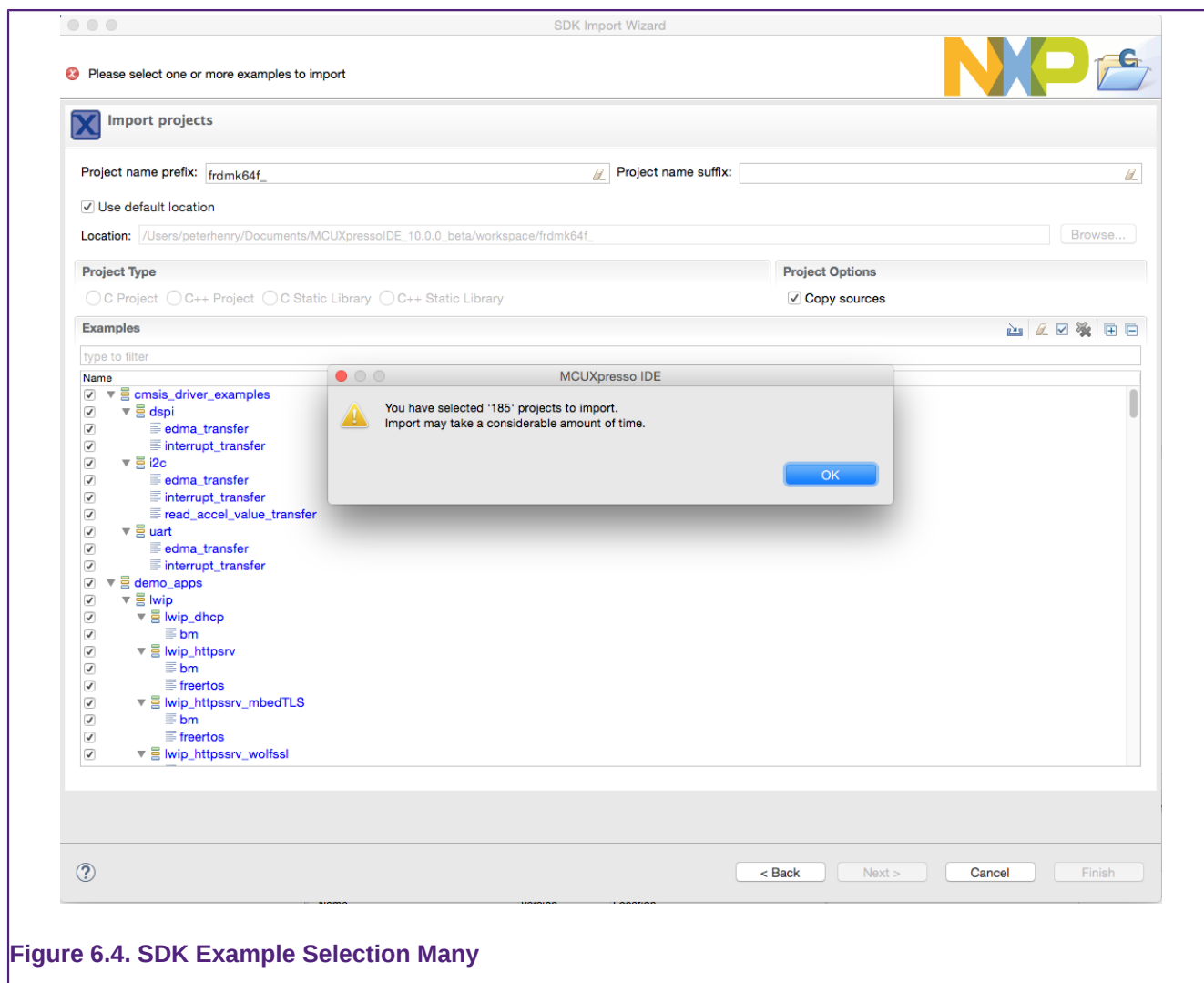


Figure 6.4. SDK Example Selection Many

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 Newlib.

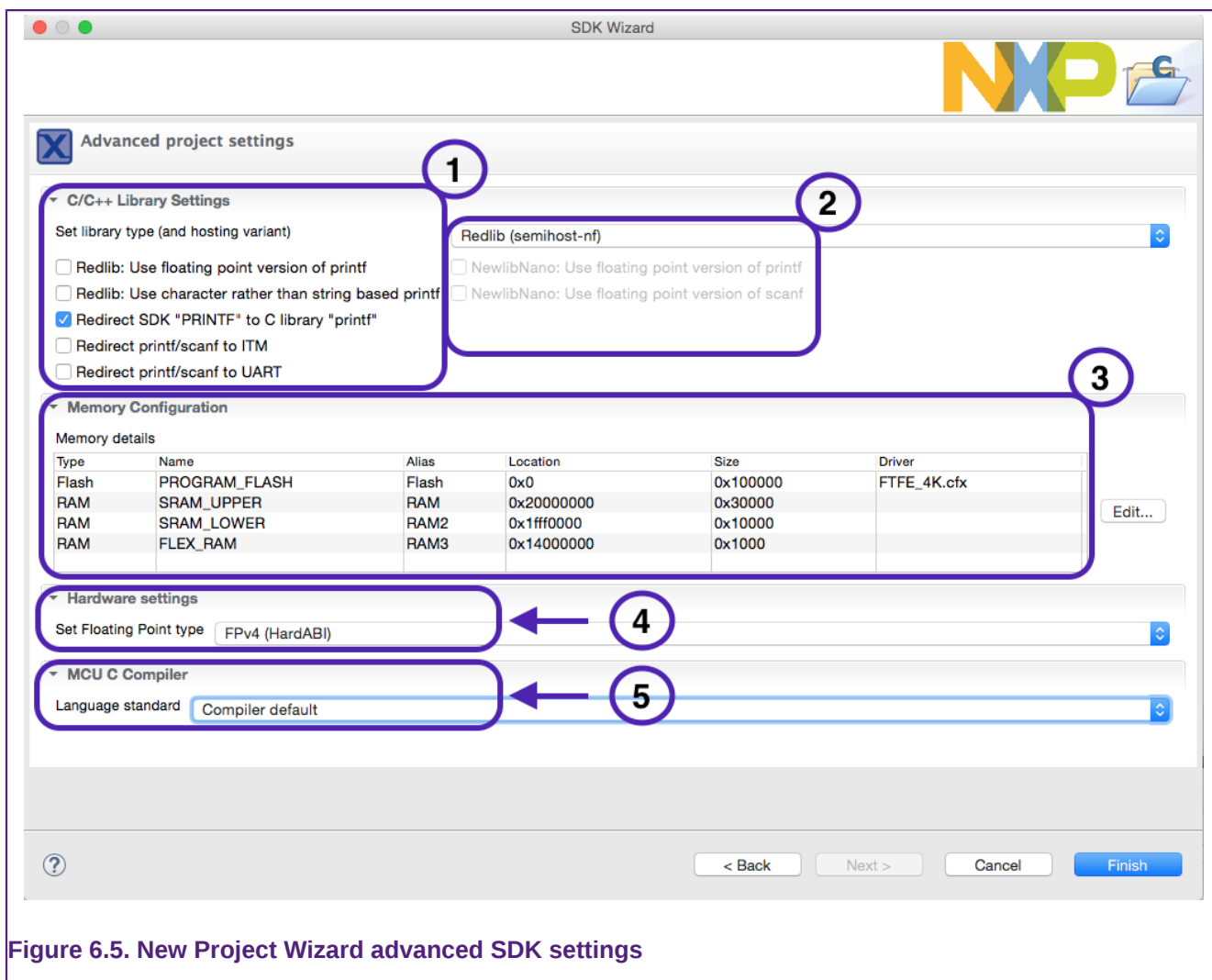


Figure 6.5. New Project Wizard advanced SDK settings

These settings closely match those in SDK New Project Wizard description. Therefore See SDK New Project Wizard:Advanced Options [36] for a description of these options. **Note:** Changing these advanced options may prevent an example from building or executing.

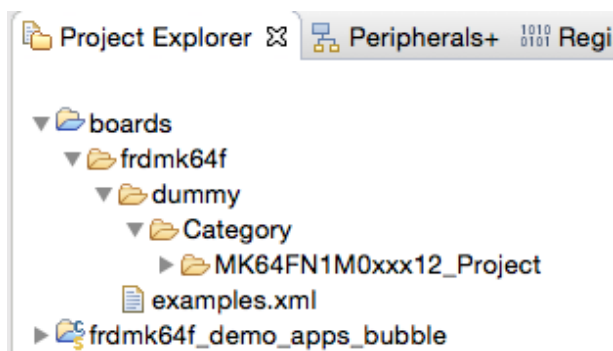
6.1.3 SDK Example Import Wizard: Import from XML fragment

This option works in conjunction with the 'Project Explorer' -> Tools -> Generate Example XML

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

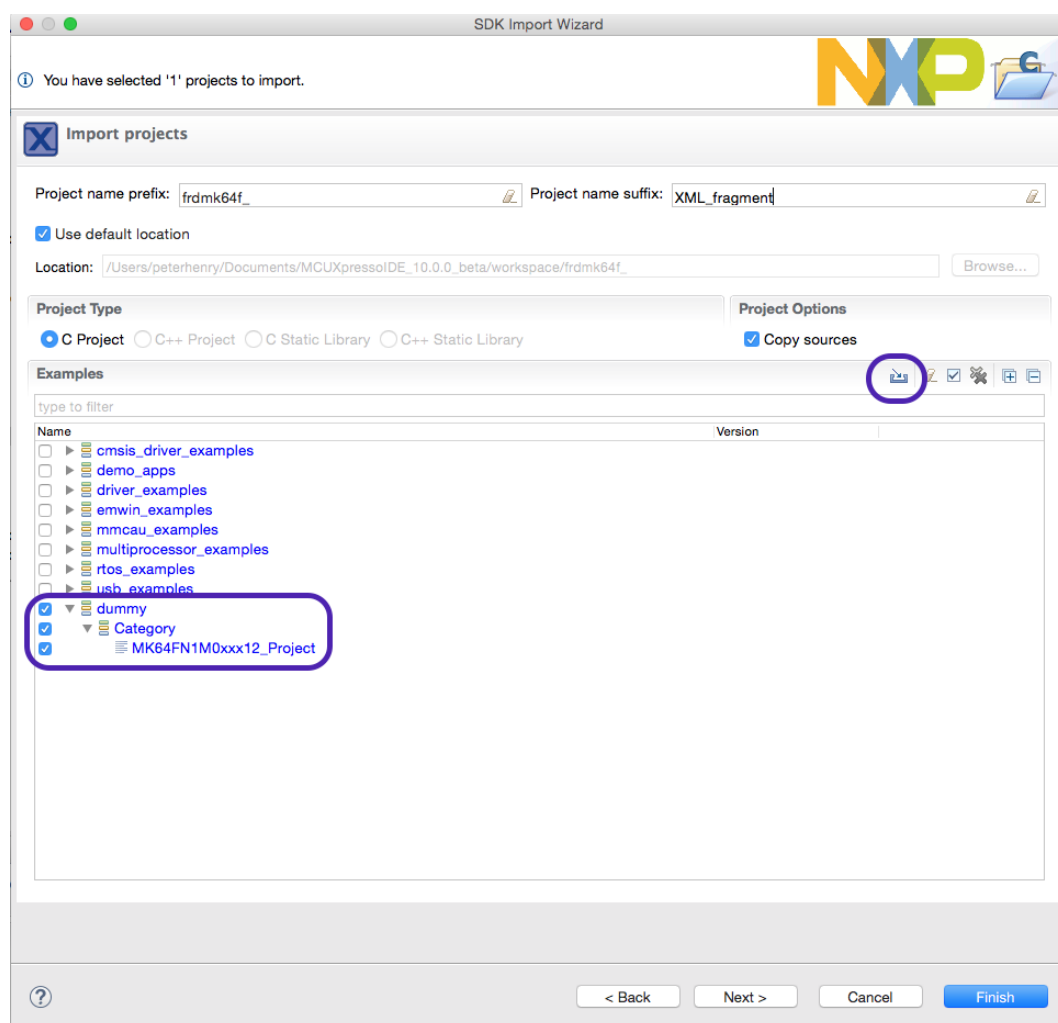
To create an XML "fragment" for an existing project in your workspace, right click on the project in the 'Project Explorer' (or just in the 'Project Explorer' view with no project selected) and choose *Tools->Generate examples.xml file*

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.



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

7. Creating New Projects using Pre-Installed Part Support

For Creating project using SDKs please see Creating Projects with SDKs) [32]

To explore the range of pre-installed parts/MCUs simply click 'New project' in the **QuickStart panel**. This will open a page similar to the image below:

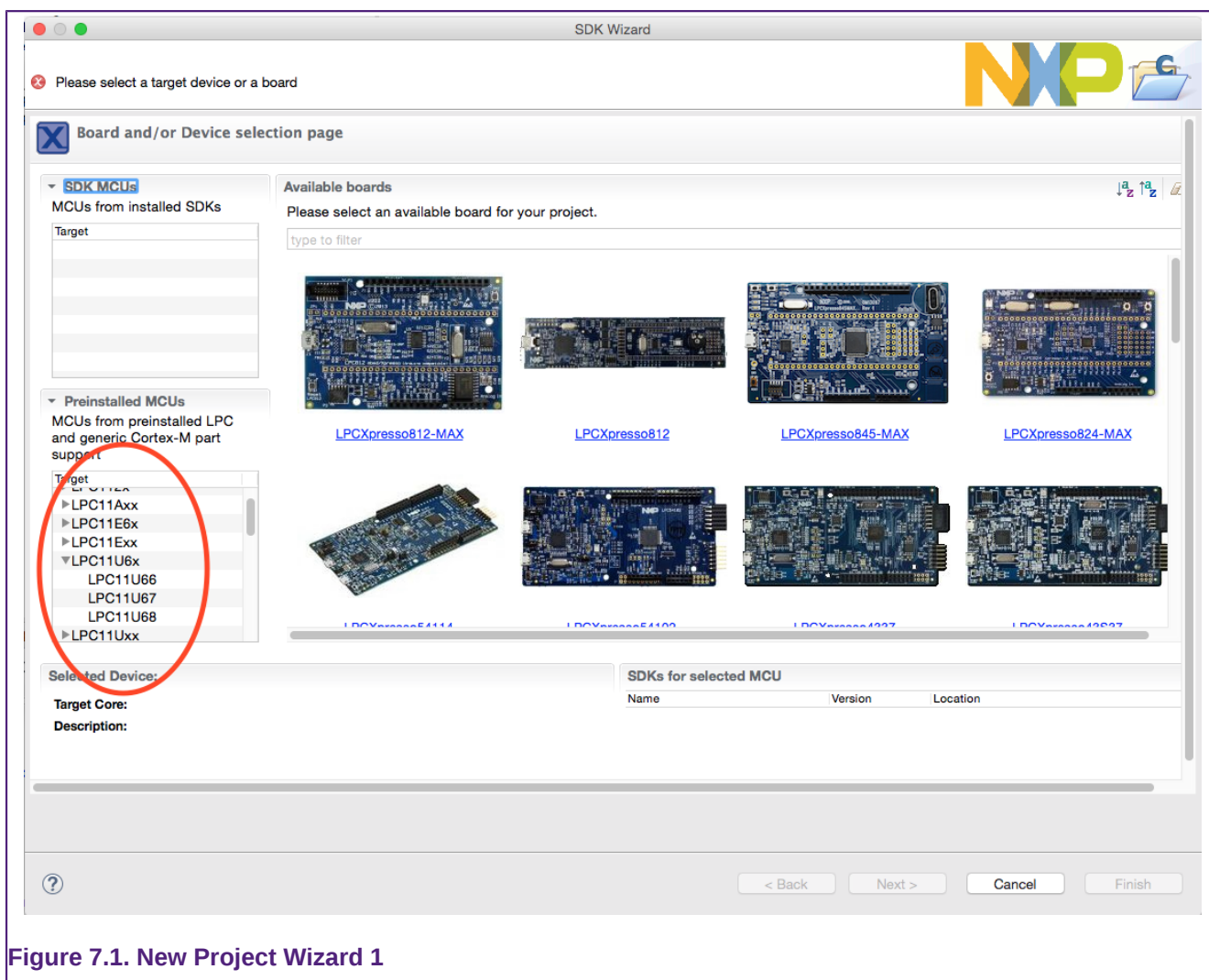


Figure 7.1. New Project Wizard 1

The list of pre-installed 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 [32]

7.1 New Project Wizard

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 an LPC4337 MCU (for this MCU an LPCOpen project exists), so we can locate the MCU using the board filter.

To reduce the number of boards displayed, we can simply type '4337' into the filter. Now only boards with MCUs matching '4337' will be displayed.

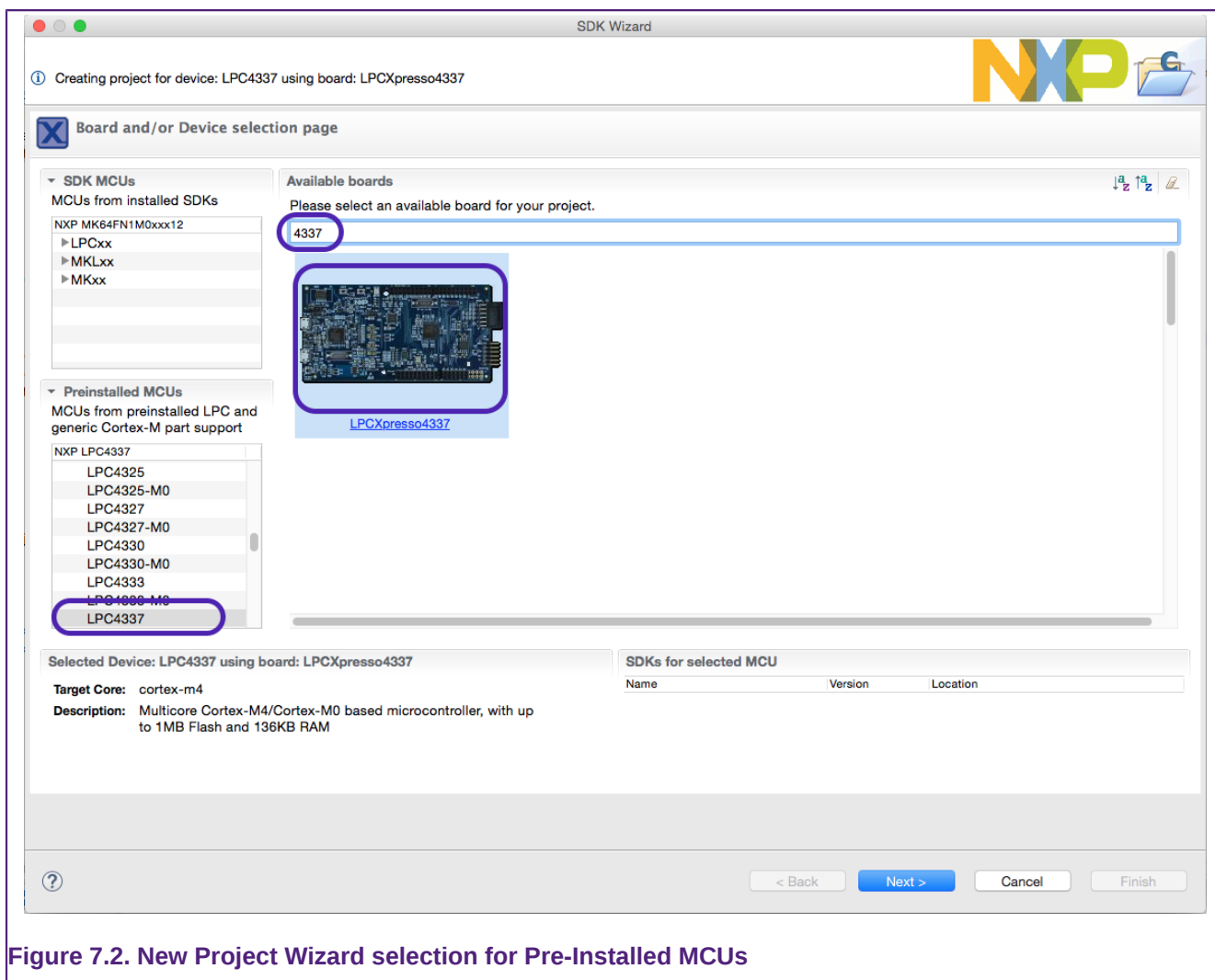


Figure 7.2. New Project Wizard selection for Pre-Installed MCUs

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

Note: if no matching board is available, the required MCU can be selected from the list of Pre-Installed 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 is available for that board and associated MCU.

LPCOpen is described in section LPCOpen Software Drivers and Examples [54]

With a chosen board selected, now click 'Next'...

The wizards for Pre-Installed MCUs are very similar to those featured in LPCXpresso IDE.

7.2 Creating a Project using a Wizard

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):

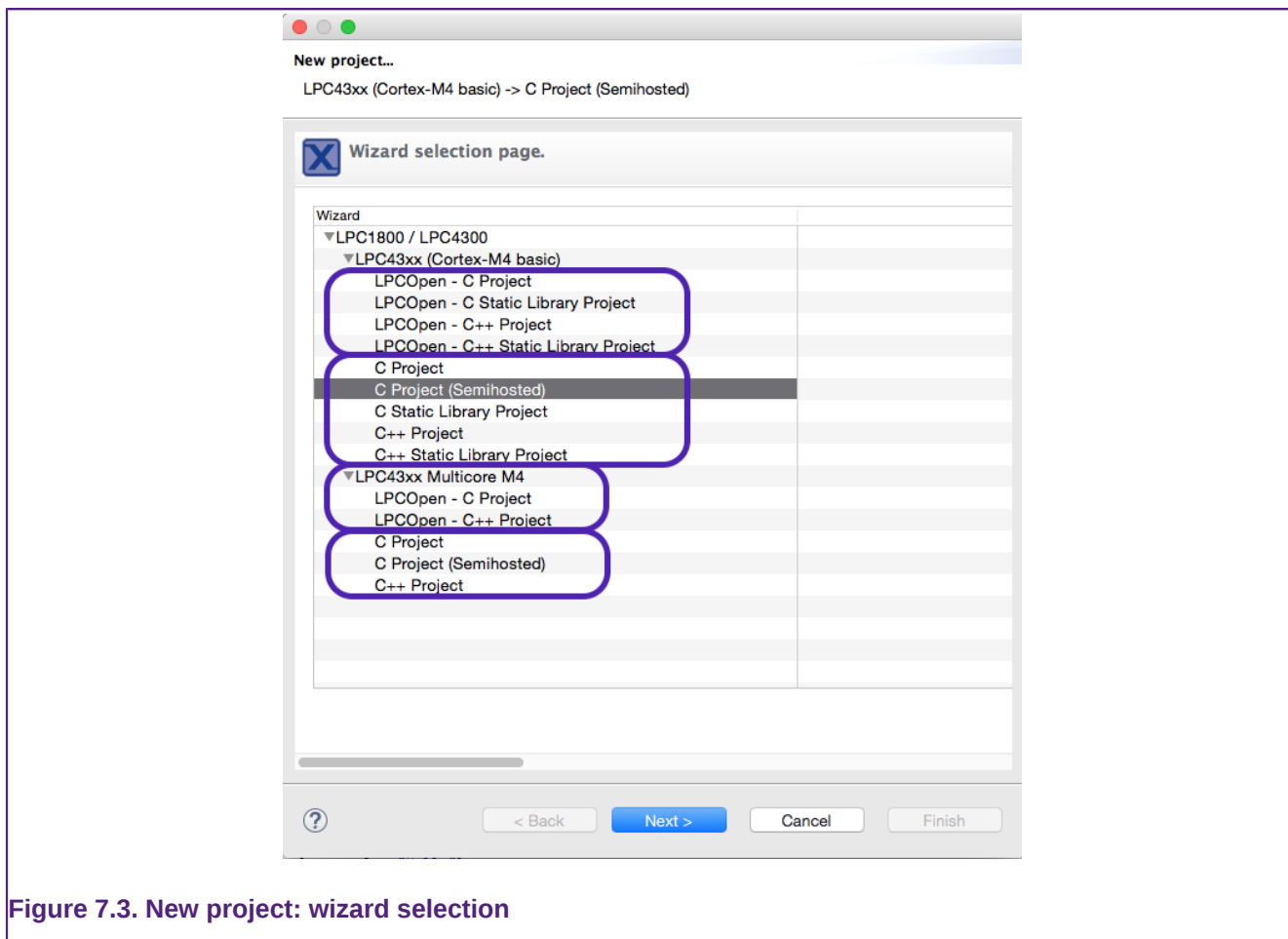


Figure 7.3. New project: wizard selection

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.

7.2.1 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 [54]. 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. For more details, please see the FAQ at

<https://community.nxp.com/message/630846>

7.2.2 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** that 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.

7.3 Wizard Options

7.3.1 LPCOpen Library Project Selection

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

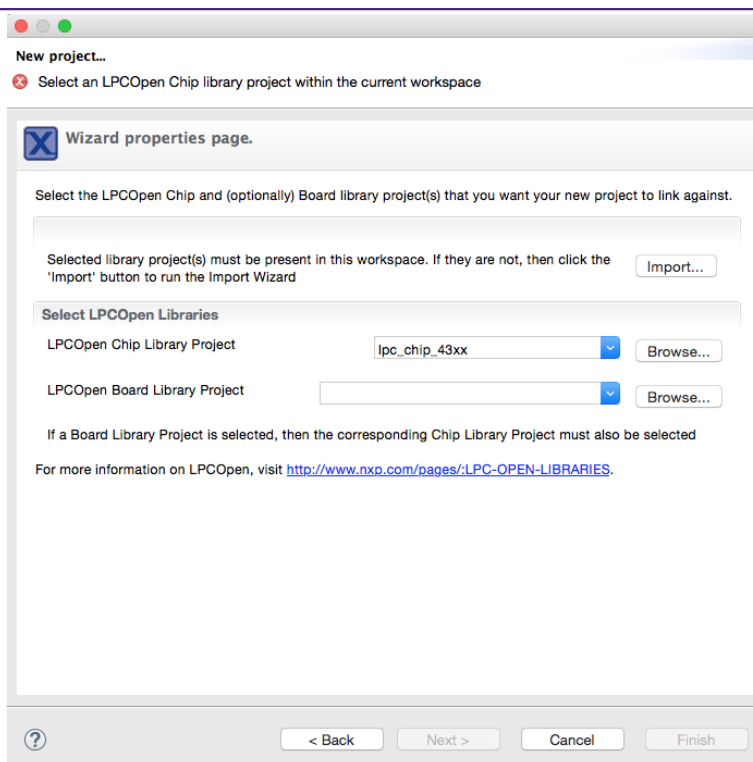


Figure 7.4. LPCOpen library selection

This page allows you to run the “Import wizard” to download the LPCOpen bundle for your target MCU/board from <http://www.nxp.com/pages/LPC-OPEN-LIBRARIES> 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.

7.3.2 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 [54]

7.3.3 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.

7.3.4 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.

7.3.5 Code Read Protect

NXP's Cortex and ARM7 based 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. For more details see the FAQ at

<https://community.nxp.com/message/630586>

7.3.6 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.

7.3.7 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>

7.3.8 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>

7.3.9 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".

7.3.10 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 additional 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 [63]

7.3.11 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 8.4 and ???.

8. Importing Example Projects (from the filesystem)

MCUXpresso IDE supports two schemes for importing examples:

- From SDKs – see the **QuickStart** Panel -> Import SDK example(s) See Importing Examples Projects (from SDK) [40]
- From the filing system – see the **QuickStart** Panel -> Import project(s)
 - this option is discussed below:

MCUXpresso IDE installs with a large number of example projects, 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 use to create flash driver used by LinkServer
- Legacy
 - a range if historic examples and drivers including CMSIS / Peripheral Driver Library
- LPCOpen
 - High quality board and chip support libraries for LPC MCUs, plus example projects

8.1 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 comments in source files. 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>

8.2 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 `lib\Examples\NXP`. 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.

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

<http://www.nxp.com/pages/LPC-OPEN-LIBRARIES>

8.3 Importing an Example Project

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

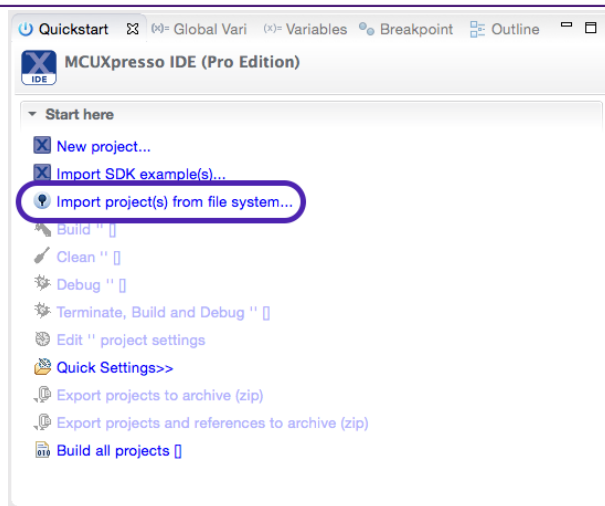


Figure 8.1. Importing project(s)

From here you can browse the filing system.

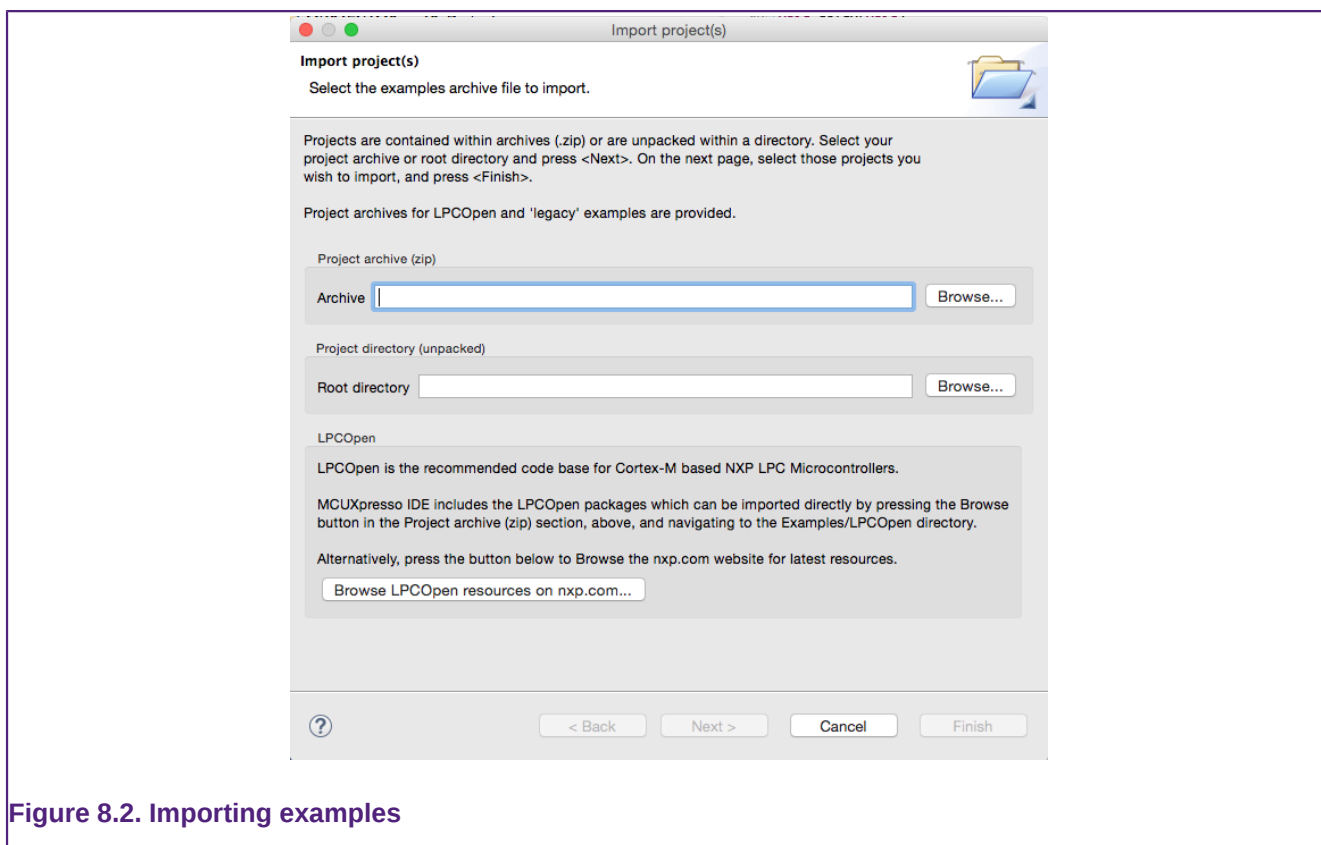


Figure 8.2. Importing examples

- **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/pages/LPC-OPEN-LIBRARIES> or the supplied, but deprecated, sample code bundles located within the Examples 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** packages to visit <http://www.nxp.com/pages/LPC-OPEN-LIBRARIES> 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.

8.3.1 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 Packages**, which will open a web browser window. Click on **Download LPCOpen Packages**, and then the link to **LPCOpen v2.xx for LPC43xx family devices**, and then choose the download for the LPCXpresso4337 board.

Note: LPCOpen Packages for the LPC4337 are pre-installed 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 8.3.

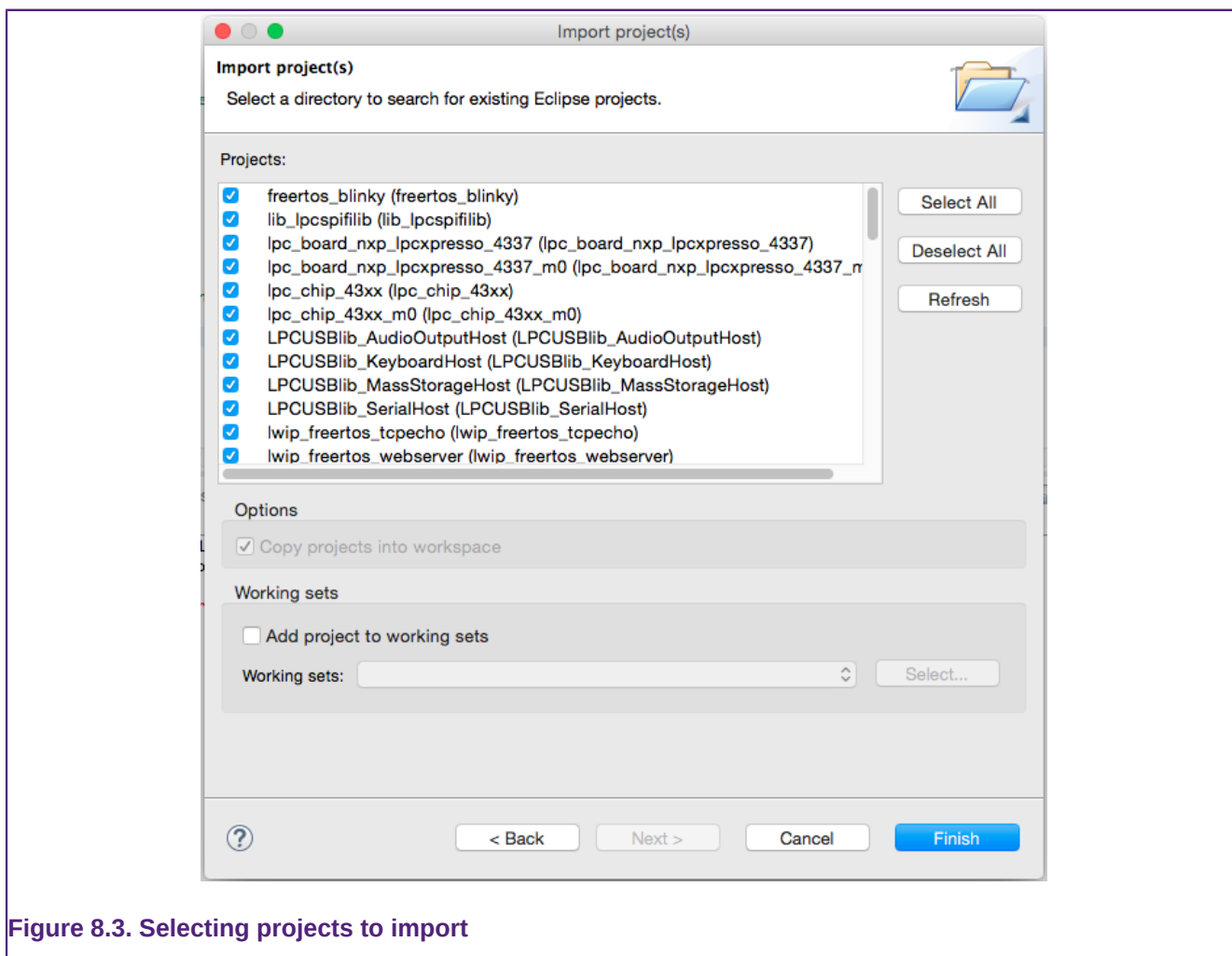


Figure 8.3. Selecting projects to import

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.

8.4 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.

8.4.1 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 the FAQ at

<https://community.nxp.com/message/630628>

9. Debugging a Project

This chapter shows how a simple debug session should be performed on an example application/project. The details below are common to all supported debug solutions. Refer to the chapter Debug Solutions Overview [10] for more details of supported debug solutions and management of debug operations.

9.1 Debugging overview

The debug chain usually starts with a debug probe USB connection 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.

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

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, simply highlight the project in the 'Project Explorer', and then in the **Quickstart Panel** click on **Debug 'Project Name'**, as in Figure 9.1. By default, this operation will first build the project and (assuming there is no build error), launch a debug probe discovery operation (see next section).

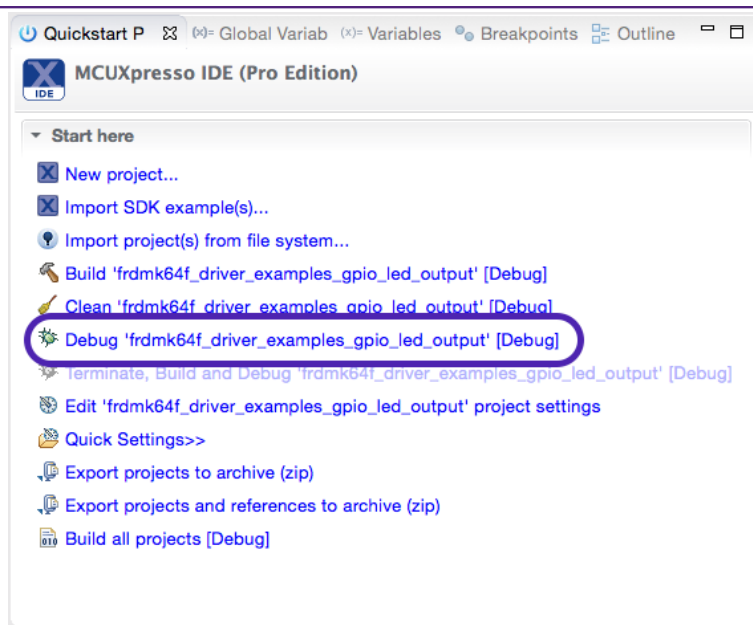


Figure 9.1. Launching a debug session

Note: Previously debugged projects will contain launch configuration files. Please see the section A note about Launch Configuration files [11] 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.

9.1.1 Debug Probe Selection Dialog

The first time you debug a project, the Debug Probe Discovery Dialogue will be displayed. This will show all supported probes that are attached to your computer. In the example shown in Figure 9.2, a LinkServer (LPC-Link2), a P&E Micro Multilink and also a J-Link (OpenSDA) probe have been found.

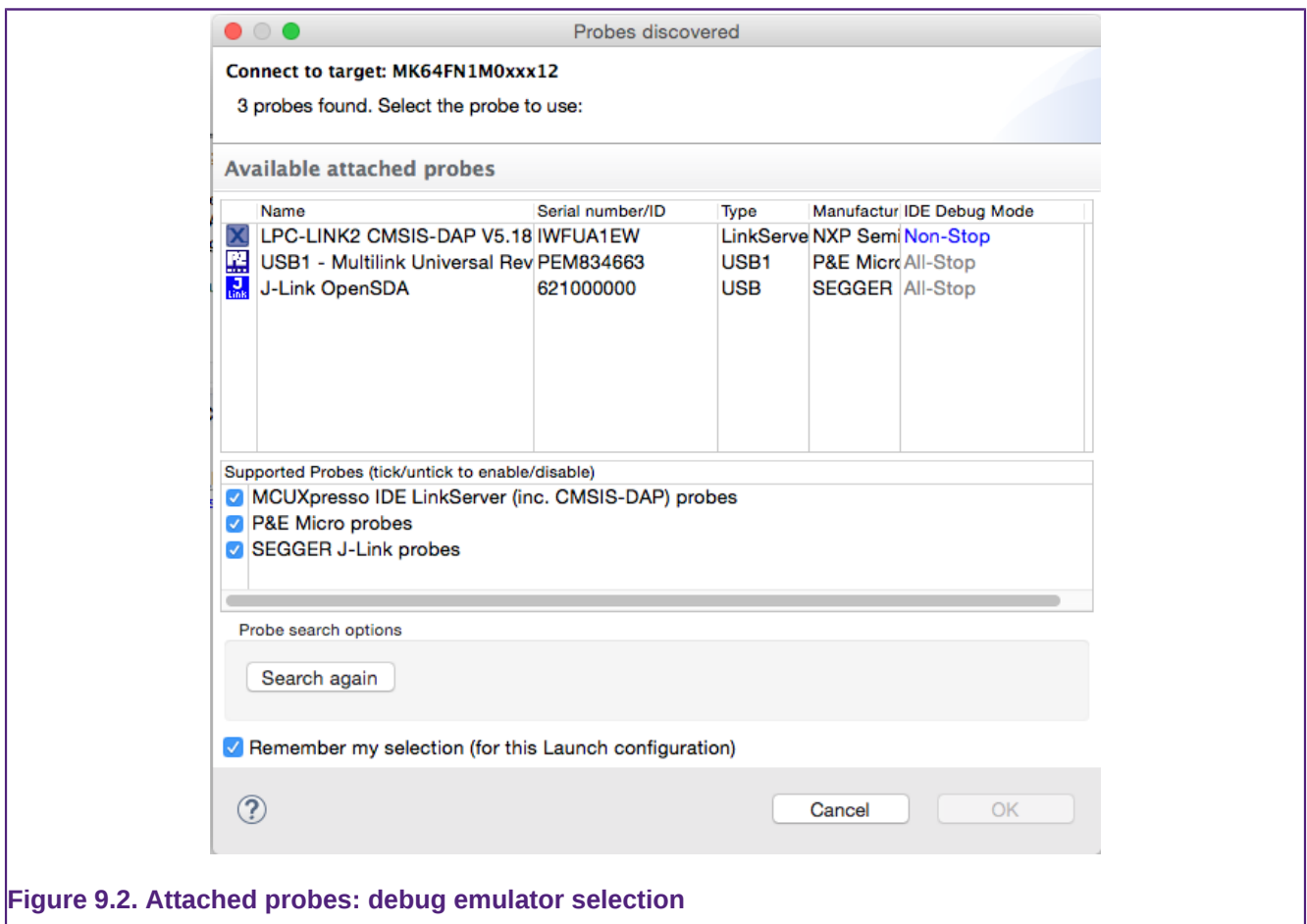


Figure 9.2. Attached probes: debug emulator selection

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.

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

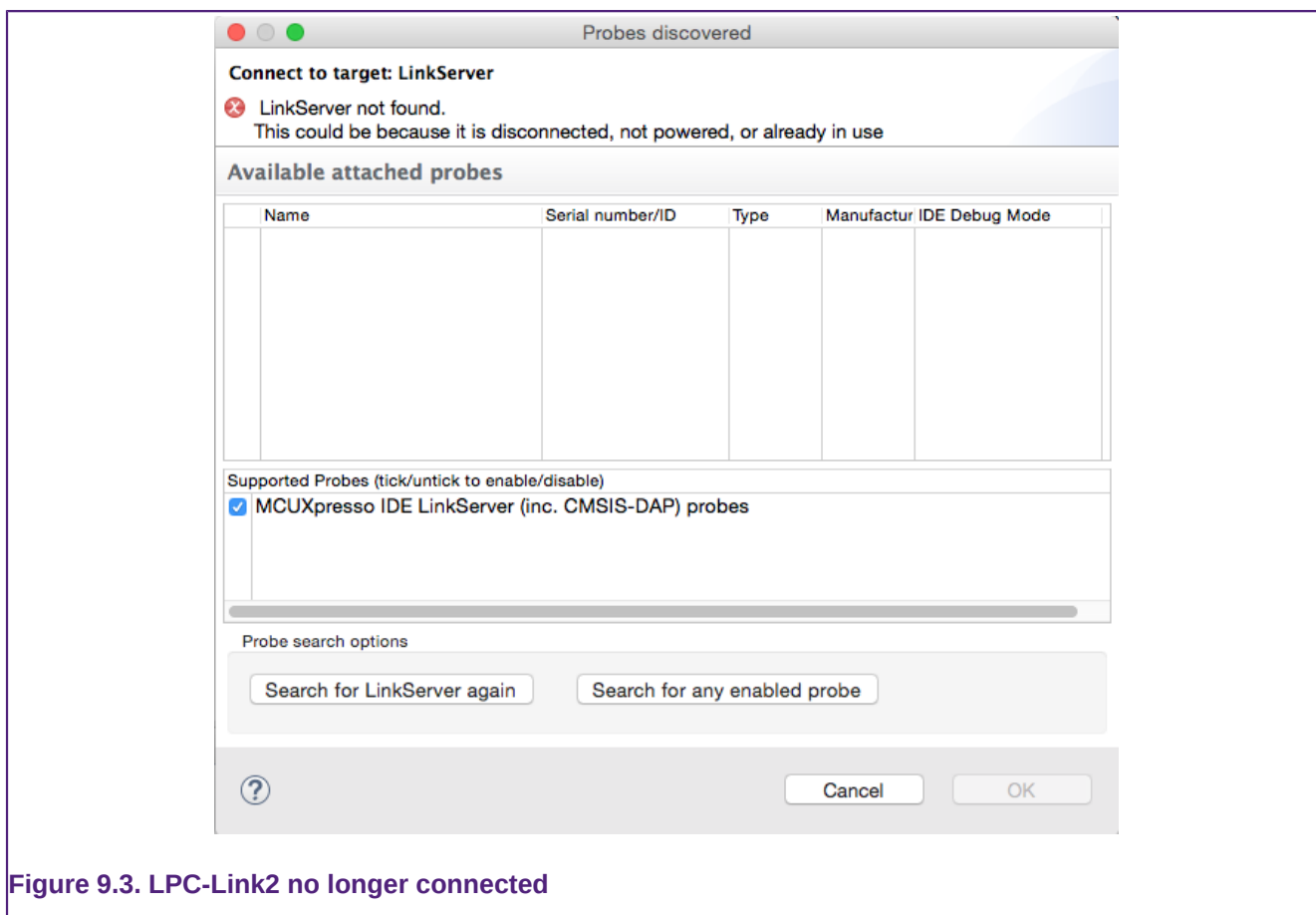


Figure 9.3. LPC-Link2 no longer connected

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 [11] for more information.

9.1.2 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 9.1, which are displayed on the global toolbar. The call stack is shown in the Debug View, as in Figure 9.4.

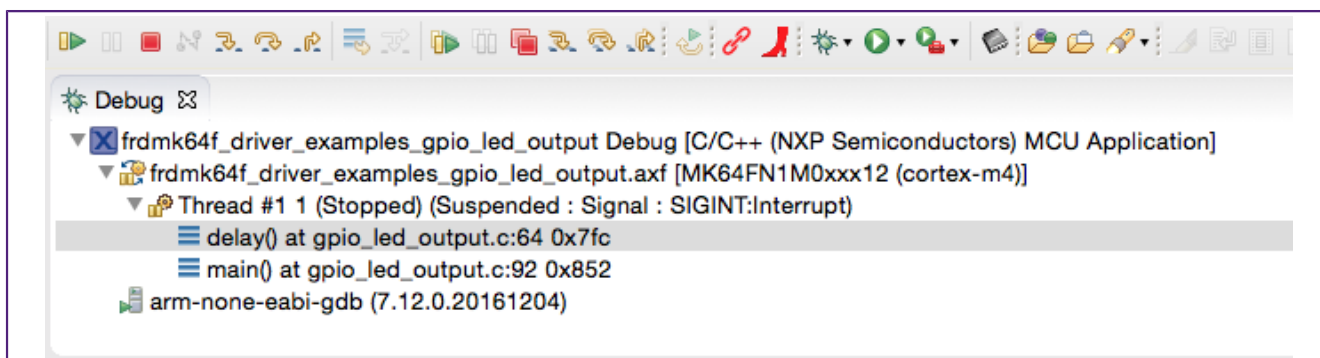


Figure 9.4. Debug controls

Table 9.1. Program execution controls

Button	Description	Keyboard Shortcut
	Restart program execution (from reset)	
	Run/Resume the program	F8
	Pause Execution of the running program	
	Stop the debugger	Ctrl + F2
	Run, Pause, Stop 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	

Setting a breakpoint

To set a breakpoint, simply double-click on the 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. C/C++ Library Support

See C/C++ Library Support [63]

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

10.1 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, MUCXpresso 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 Redlib, which can often produce much smaller applications.

10.1.1 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.

10.1.2 Newlib vs NewlibNano

Differences between Newlib and Newlib-Nano include:

- Newlib-Nano 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, as described above. TBD: Where is it described?
- 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 (`iprntf/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

10.2 Library variants

Each C library family is provided in a number of different variants : None, Nohost, 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 that this emulated I/O is relatively slow and can only be used when debugging.
- **Semihost-nf (no files)**
 - Redlib only. Similar to Semihost, 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**
 - 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(fprintf.o): In function `printf':
fprintf.c:(.text.printf+0x38): undefined reference to `__sys_write'
fprintf.c:(.text.printf+0x4c): undefined reference to `__Clob'
... 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-isatty.o): In function `_isatty_r':
isatty.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

10.3 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.

10.3.1 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 that Redlib is not available for C++ projects
5. In Use 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 that Redlib is not available for C++ projects.

10.4 What is Semihosting?

Semihosting is a term to describe application I/O via the debug probe. For this to operate, library code and debug support are required.

10.4.1 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.

10.4.2 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.

10.4.3 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.

10.4.4 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.

10.4.5 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

10.5 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.

10.5.1 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 additional 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.

10.5.2 NewlibNano `printf` variants

By default, Newlib-Nano 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.

10.5.3 Newlib printf variants

Newlib provides an "iprintf" function which implements integer only printf

10.5.4 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`

10.5.5 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 onboard 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.

10.5.6 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.

TBD: Link 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: <http://sourceware.org/newlib/libc.html#Syscalls>

10.5.7 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"`

10.6 itoa() and uitoa() in Redlib

itoa() is non-standard library function which is provided in many other toolchains to convert an integer to a string. 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()` / `uito()` 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
```

10.6.2 Newlib/NewlibNano

Newlib and NewlibNano now also provide similar functionality though with slightly different naming - **itoa()** and **utoa()**.

10.7 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*
- Redlib (Nohost), add
 - [C projects only]: *GROUP*
- Redlib (Semihost-nf), add
 - [C projects only]: *GROUP*
- Redlib (Semihost), add
 - [C projects only]: *GROUP*
- NewlibNano (None), add
 - [C projects]: *GROUP*
 - [C++ projects]: *GROUP*
- NewlibNano (Nohost), add
 - [C projects]: *GROUP*

- [C++ projects]: *GROUP*
- NewlibNano (Semihost), add
 - [C projects]: *GROUP*
 - [C++ projects]: *GROUP*
- Newlib (None), add
 - [C projects]: *GROUP*
 - [C++ projects]: *GROUP*
- Newlib (Nohost), add
 - [C projects]: *GROUP*
 - [C++ projects]: *GROUP*
- Newlib (Semihost), add
 - [C projects]: *GROUP*
 - [C++ projects]: *GROUP*

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.

11. Memory Configuration and Linker Scripts

See Memory Configuration and Linker Scripts [73]

See Creating Projects with SDKs [32] See Creating Projects with Pre-Installed part support [47] See New Project Wizard details [32]

See Importing Projects from SDKs [40] See Importing Projects from the filesystems [54] See Debug Solutions Overview [10] Figure 2.1

11.1 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 – typically CMSIS-DAP / LPC-Link2).

For pre-installed 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 associated with the newly defined region.

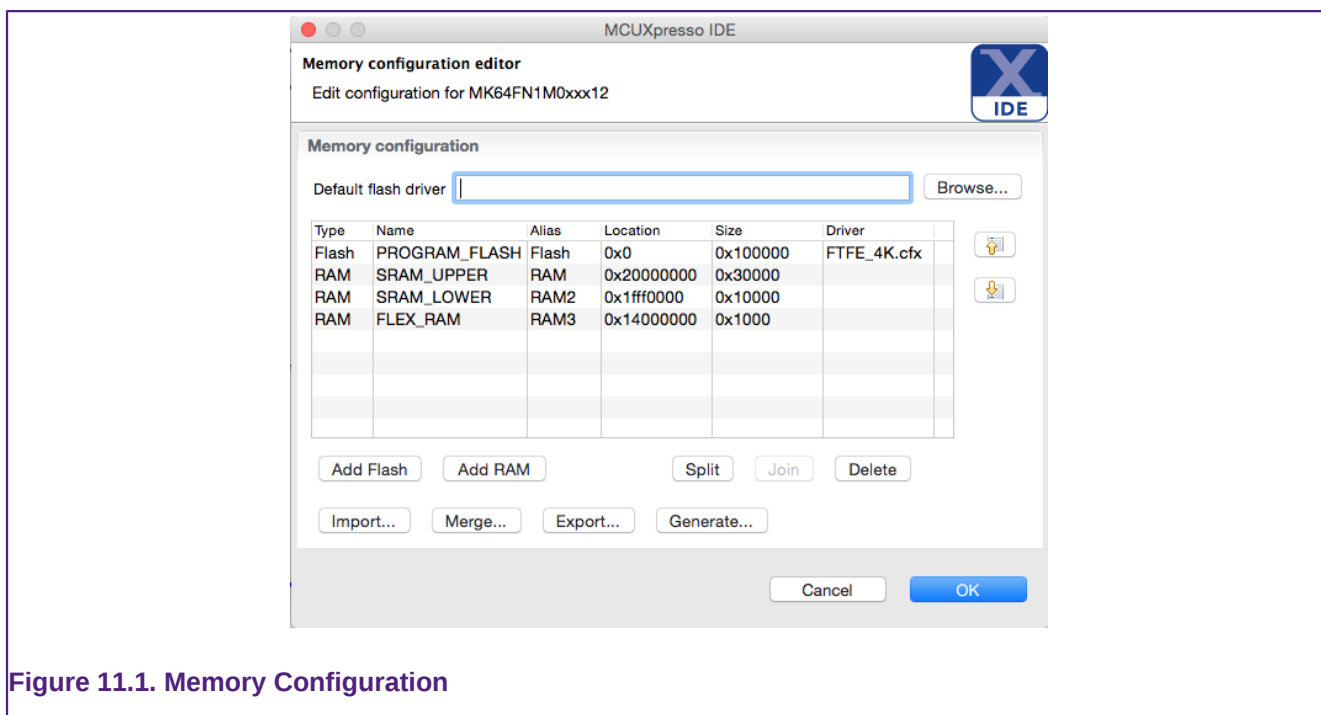


Figure 11.1. Memory Configuration

11.2 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).

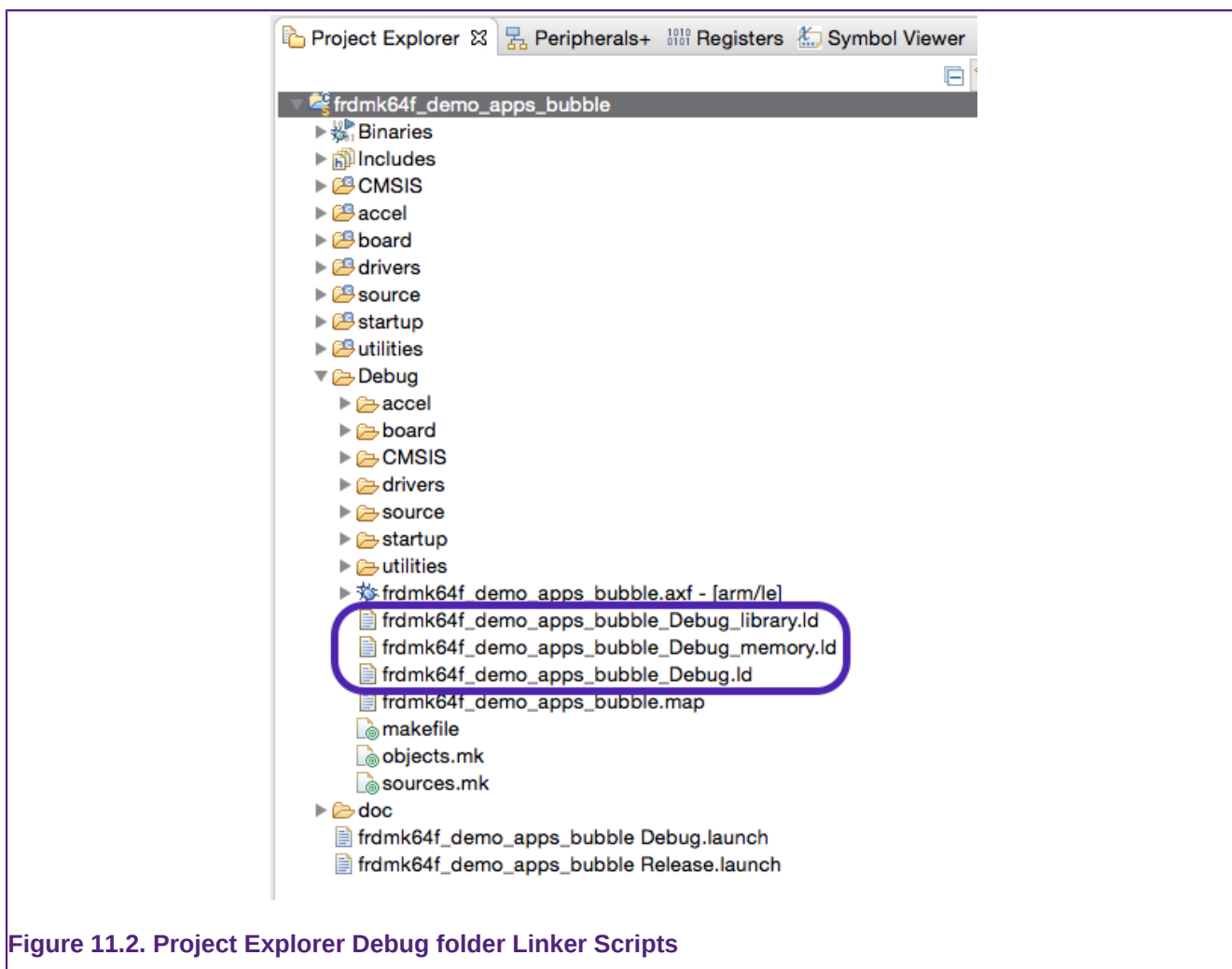


Figure 11.2. Project Explorer Debug folder Linker Scripts

The managed linker 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.

11.3 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 the 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 the Wizards\linker subdirectory of your IDE install.

TBD: More details and examples

11.4 Default image layout

Code and initial values of initialised data items are placed into the first bank of flash (as shown in memory configuration editor). During startup, the MCUXpresso IDE startup code copies the data into the first bank of RAM (as shown in memory configuration editor), and zero initializes

the BSS data directly after this in memory. This process uses a data 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.

TBD: In order to initialize additional 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 initialize them.

The MCUXpresso IDE generated startup code will then initialize memory region(s) by entering a loop reading this global section table, and calling a subroutine to carry out the initialization of each region it finds in the Global Section Table.

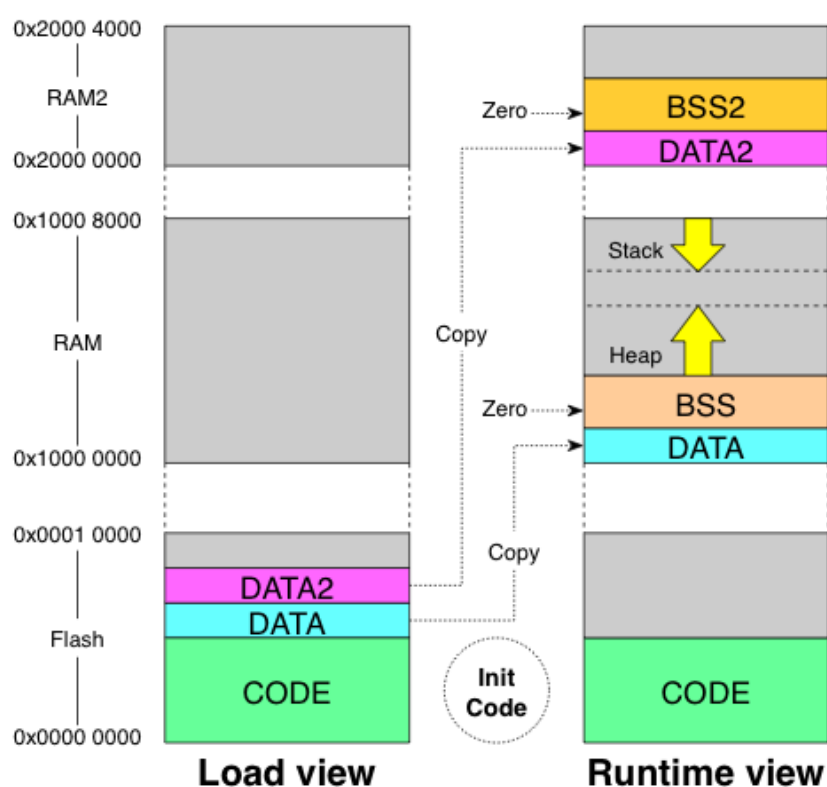


Figure 11.3. Default Memory Layout

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.

11.5 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.

11.5.1 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			

TBD: Links to styles of memory layout and usage – below is the text from earlier when build is mentioned.

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 1 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 default for this build has reserved 4KB each for the stack and the heap.

TBD: sort out Please See Memory Configuration and Linker Scripts[73] for detailed information.

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.

11.5.2 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.

11.5.3 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 \<application\>.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 and why
- 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.

11.5.4 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)

TBD: LPCXpressoViewSymbolsMenu.preview.jpg

The image will be processed and displayed in the Symbol Viewer as seen below:

TBD: LPCXpressoSymbolsViewer.jpg

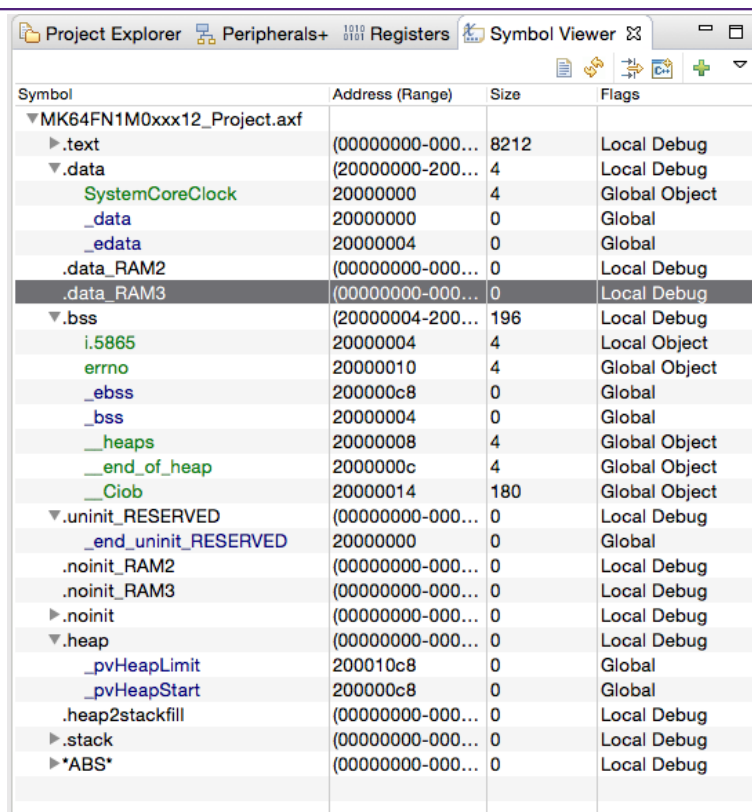
It is possible to open multiple Symbol Viewers by pressing the 'Green +' icon in the toolbar. The symbols for different images can then be displayed simultaneously.

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 appropriate).

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
l.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 11.4. 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.

11.6 Other options affecting the generated image

11.6.1 LPC MCUs – Code Read Protection

Most of NXP's LPC Cortex-M based MCUs which contain 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.

TBD : Screenshot of linker properties with CRP box

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.

11.6.2 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, 0xFFFFFFFF};
```

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.

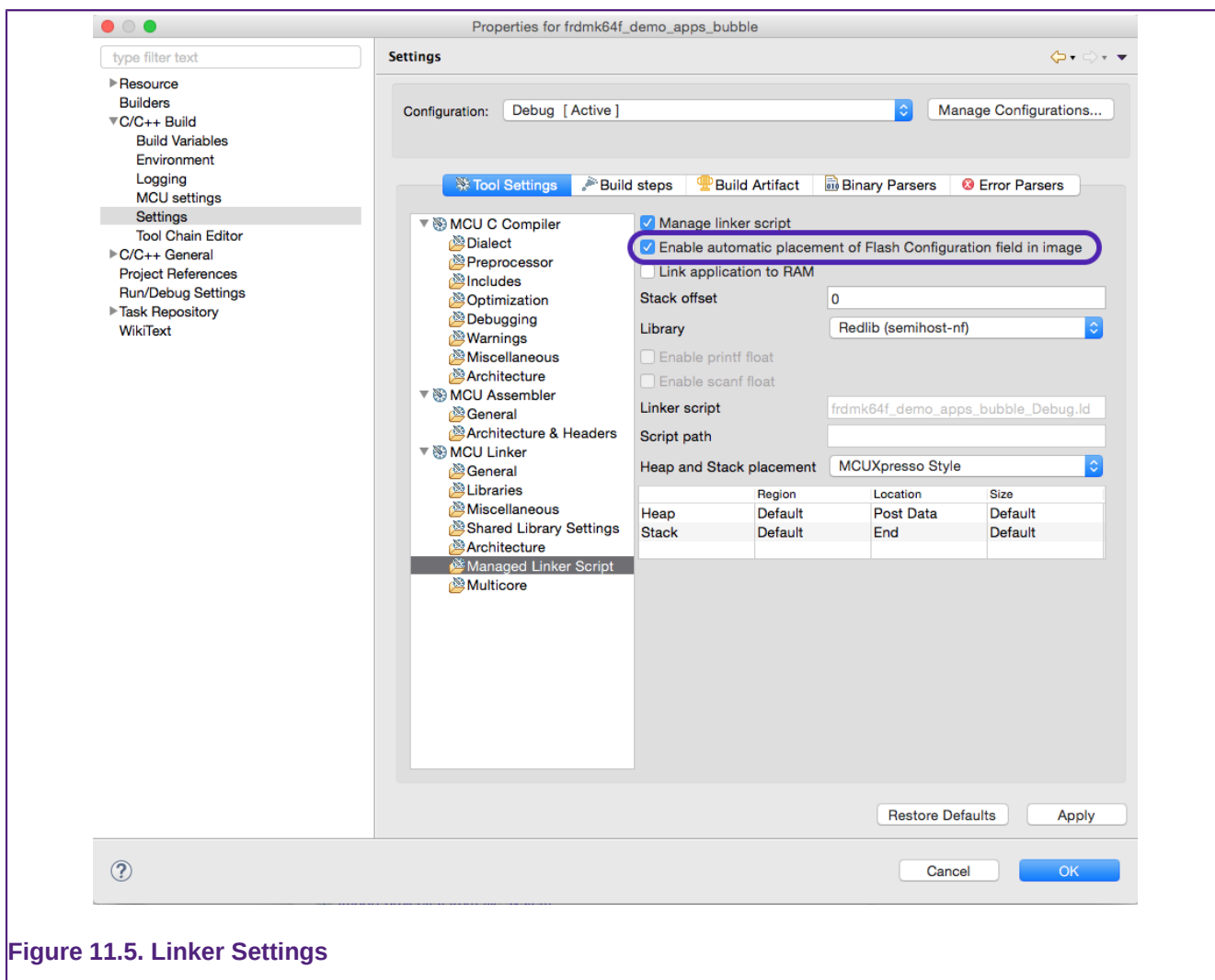


Figure 11.5. Linker Settings

11.6.3 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, including for parts with dedicated USB_RAM (small or large variants).

TBD: Need to say more here

11.7 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.

11.8 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.

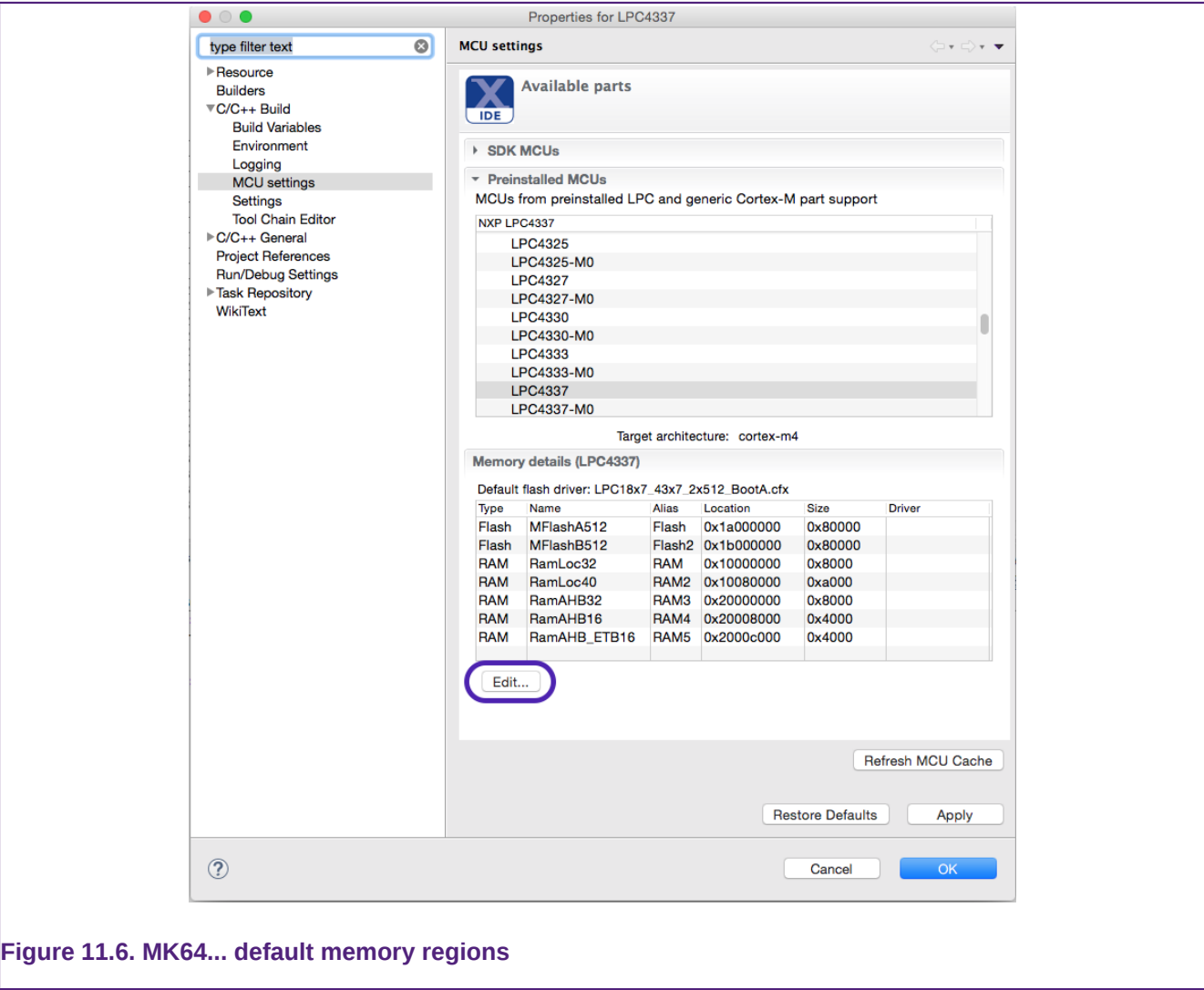


Figure 11.6. MK64... default memory regions

11.8.1 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 11.7.

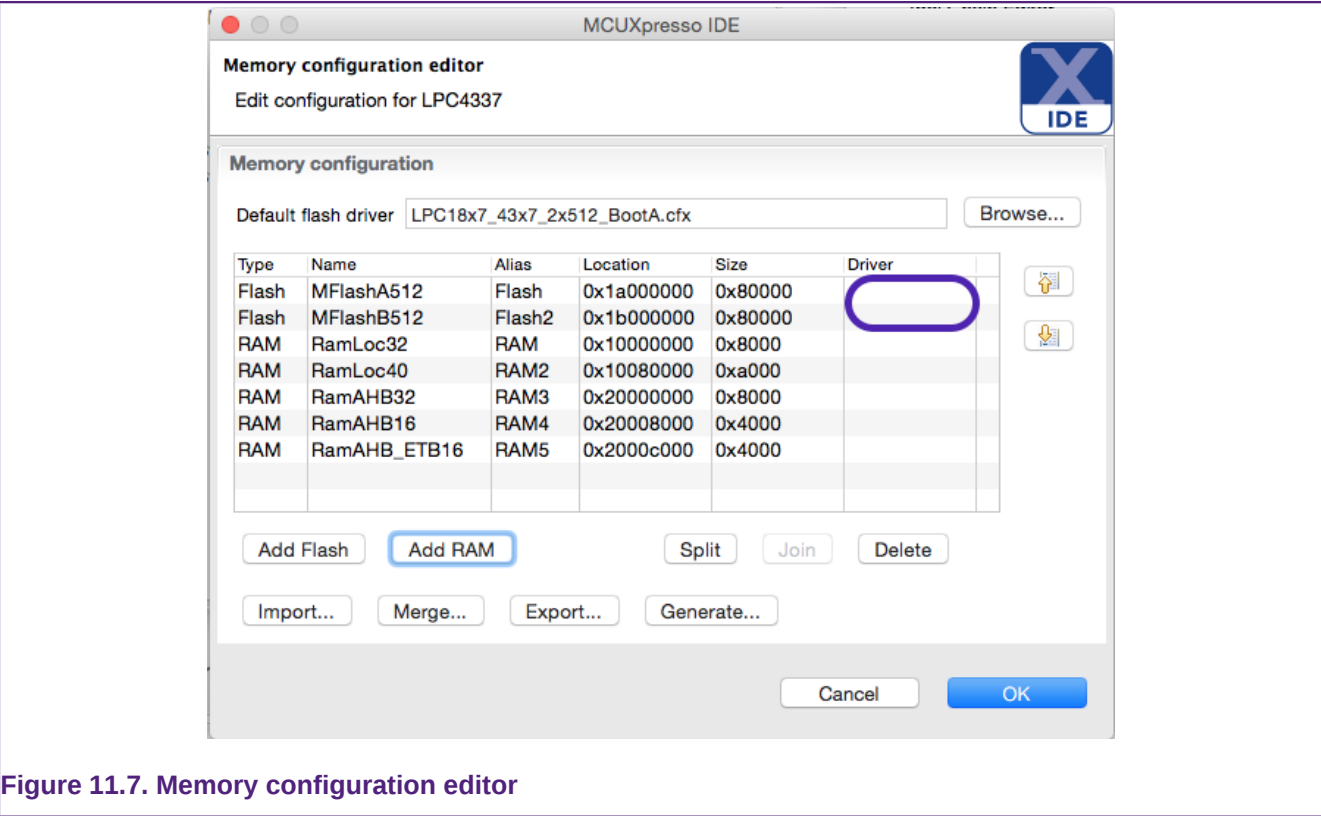


Figure 11.7. Memory configuration editor

Known blocks of memory, with their type, base location, and size are displayed. Entries can be created, deleted, etc by using the provided buttons — see ???.

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 11.1. 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. TBD: more?
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. TBD: more?
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 \bin\flash subdirectory of the MCUXpresso IDE installation. For more details see User loadable flash drivers TBD: two mechanisms .

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 in ??? shows the dialog after the “Add Flash” button has been clicked.

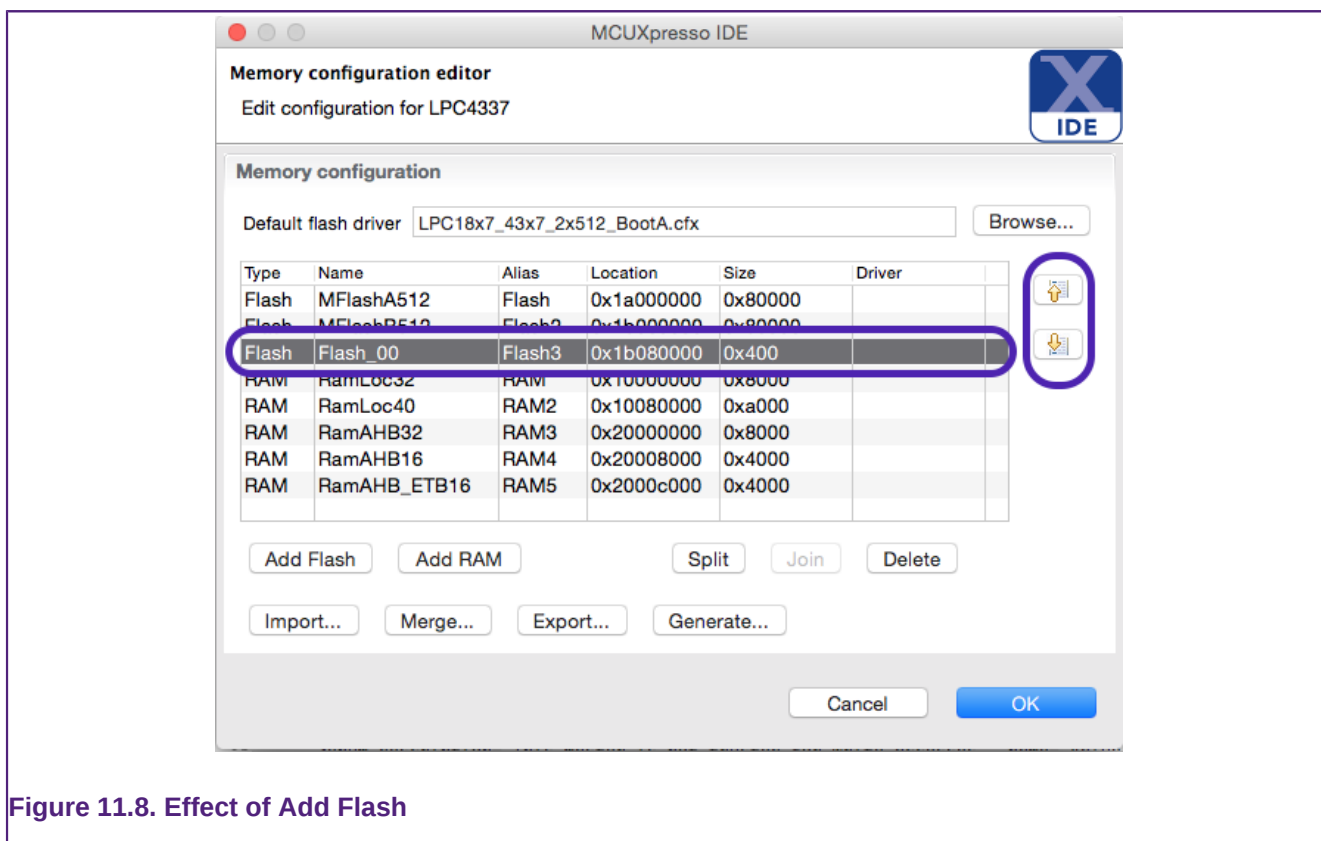


Figure 11.8. Effect of Add Flash

After updating the memory configuration, click **OK** to return to the MCU settings dialog, which will be updated to reflect the new configuration — see ???.

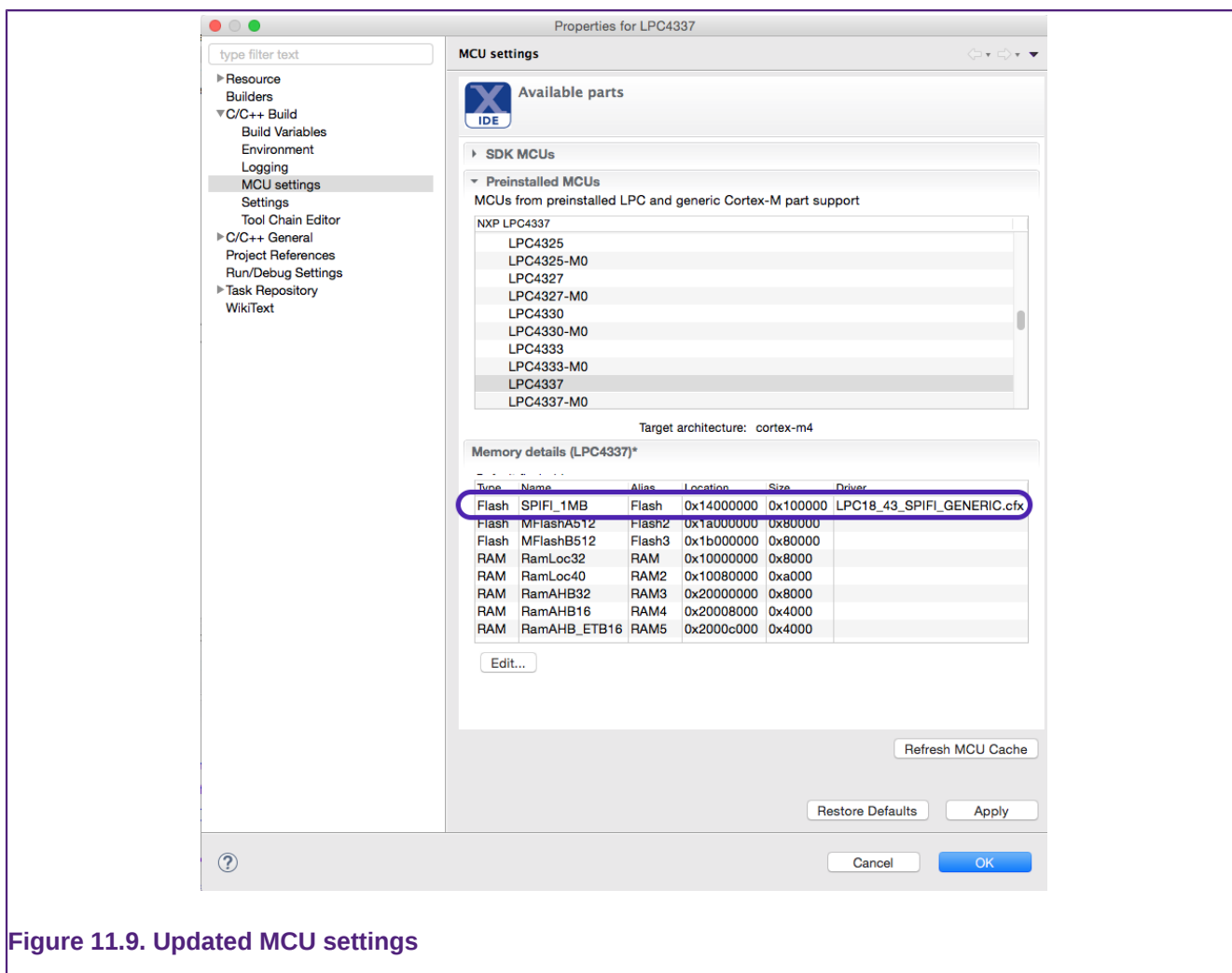


Figure 11.9. Updated MCU settings

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.

11.8.2 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 family MCUs) or LPC40xx_SPIFI_GENERIC.cfx (for LPC407x/8x MCUs, MCUXpresso IDE v8.2 onwards only).

11.8.3 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.

11.8.4 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 MCUs. TIn 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. In addition, it allows a flash driver to be allocated for use with parts with no internal flash, but where an external flash part is connected.

11.9 More advanced heap/stack placement

TBD: Link here from build

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 using the option...

Project -> Properties -> C/C++ Build -> Settings -> MCU Linker

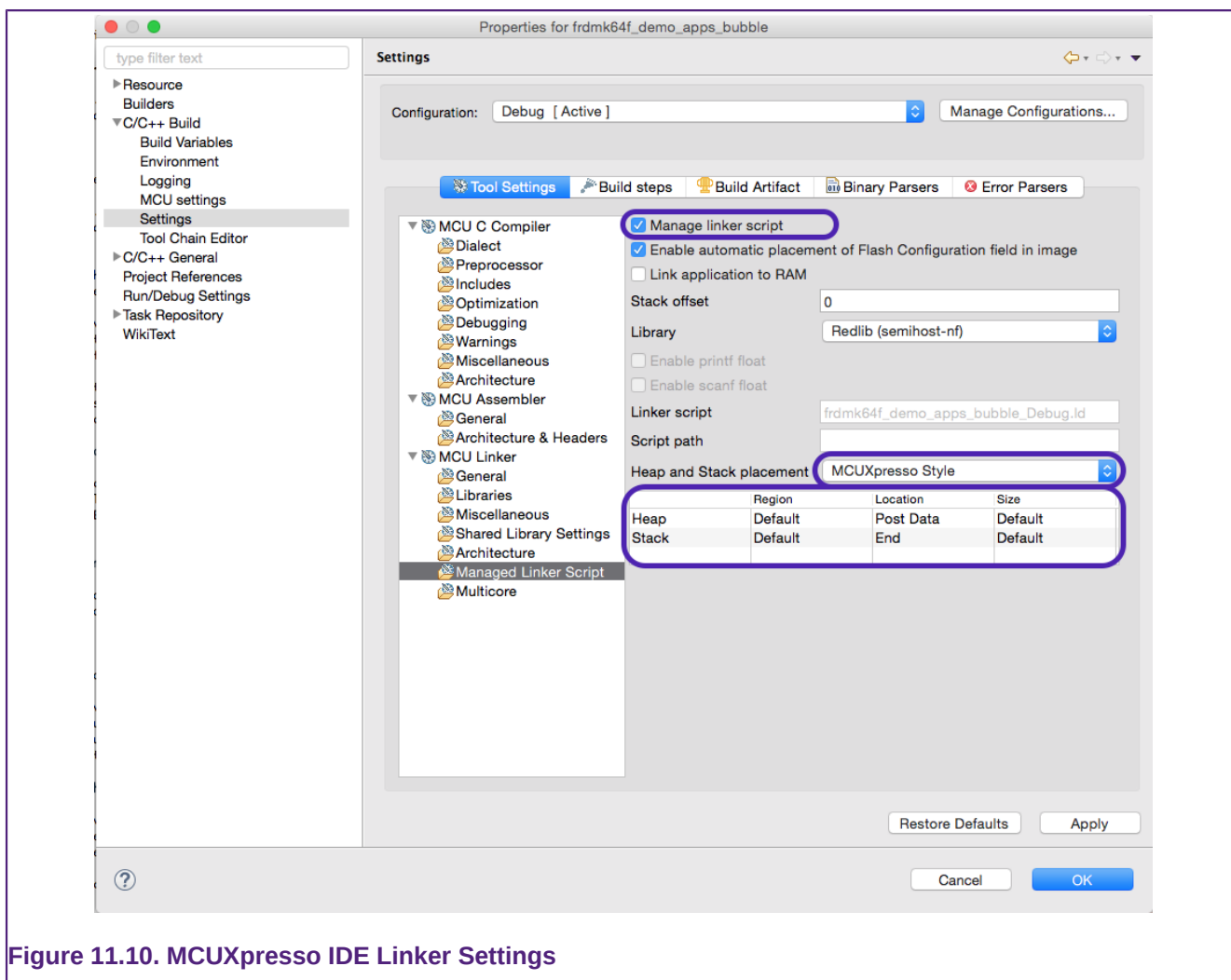


Figure 11.10. MCUXpresso IDE Linker Settings

In the dialogue above, highlights show the managed linker script option along with the selection of the MCUXpresso Style scheme.

11.9.1 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.

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.

11.9.2 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

11.9.3 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 -> Target

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.

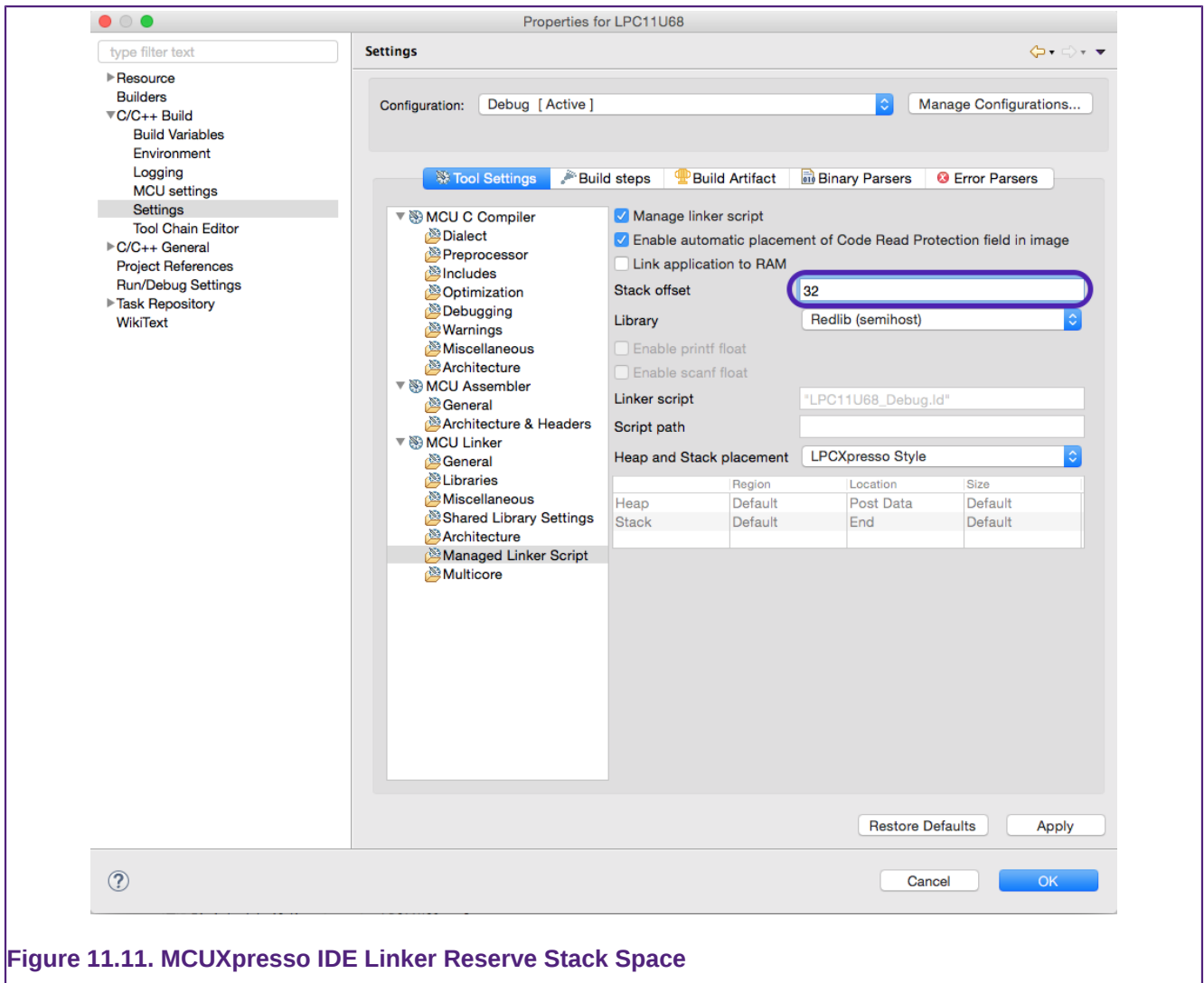


Figure 11.11. MCUXpresso IDE Linker Reserve Stack Space

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.

11.9.4 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

11.9.5 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 heap will overflow
- 0 – If heap 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 MCUXpresso 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.

11.9.6 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 Freemaker 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

TBD: <https://community.nxp.com/thread/389102>

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

TBD: <https://community.nxp.com/thread/389099>

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 at 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 an 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 increase 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.

11.10 Freemaker 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

11.10.1 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 Freemaker 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 Freemaker can be found at <http://freemarker.org/docs/index.html>.

MCUXpresso IDE automatically invokes Freemaker, 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, #include's 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.

TBD : Diagram from <https://community.nxp.com/message/630611>

TBD: Some example Freemaker linker script templates can be found later in this chapter

11.10.2 Reference

Freemaker reads input files, copying text and processing Freemaker 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 #include's 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)
- 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)
- section_tail.ldt (immediately before the send of linker SECTION directive)

library.ldt (the standard libraries used in the application)

- 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)
 - `<notextile> libraries0 = libcr_c.a </notextile>`
 - `<notextile> libraries1 = libcr_eabihelpers.a </notextile>`
- `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
 - `<notextile> configMemory0 = name=MFlashA512 alias=Flash location=0x1a000000 size=0x80000 sizek=512K bytes mcuPattern=Flash flash=true RAM=false defaultFlash=true defaultRAM=false </notextile>`
 - `<notextile> configMemory2 = name=RamLoc32 alias=RAM location=0x10000000 size=0x8000 sizek=32K bytes mcuPattern=RAM flash=false RAM=true defaultFlash=false defaultRAM=true </notextile>`
- `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
 - ```
<notextile>slaves 0 = name=M0APP objectPath=${workspace_loc:/
MCB4357_Blinky_DualM0/Debug/MCB4357_Blinky_DualM0.axf.o}
linkSection=Flash2 runtimeSection= textSection=.core_m0app
textSectionNormalized=_core_m0app dataStartSymbol=__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

### 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)

## 11.11 Freemaker Linker Script Template Examples

The use of Freemaker 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.

### 11.11.1 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.

TBD: Link

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 Freemaker 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 is 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.

## 11.11.2 Configuring projects to span multiple flash devices

TBD: <https://community.nxp.com/message/630582>

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 Freemarkers linker 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).

## 11.12 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 circumstances, the facilities provided by the managed linker script mechanism, and its underlying Freemarkers 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 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>

## 12. Multicore Projects

---

### 12.1 LPC43xx Multicore Projects

The LPC43xx family of MCUs contain a Cortex-M4 “master” core and one or more Cortex-M0 “slave” cores. After a power-on or Reset, the master core boots and is then responsible for booting the slave core(s). However, this relationship only applies to the booting process; after boot, your application may treat any of the cores as the master or a slave.

The MCUXpresso IDE allows for the easy creation of “linked” projects that support the targeting of LPC43xx Multicore MCUs. For more information on creating and using such multicore projects, please see the FAQ at

<https://community.nxp.com/message/637967>

### 12.2 LPC541xx Multicore Projects

Some members of the LPC541xx family of MCUs contain a Cortex-M4 core and a Cortex-M0+ core (with the Cortex-M4 being the master, and the M0+ the slave). 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.

The MCUXpresso IDE allows for the easy creation of “linked” projects that support the targeting of LPC541xx Multicore MCUs. For more information on creating and using such multicore projects, please see the FAQ at

<https://community.nxp.com/message/630715>

# Appendix A. Glossary

|       |                                                                                  |
|-------|----------------------------------------------------------------------------------|
| GROUP | libgcc.a    libc_nano.a    libstdc++_nano.a    libm.a<br>libcr_newlib_semihost.a |
| SDK   | s                                                                                |