UG10055

MCUXpresso IDE 24.12 User Guide

Rev. 4 — 14 January 2025

User guide

Document information

Information	Content
Keywords	MCUXpresso, MCUXpresso IDE
Abstract	MCUXpresso is a MCU development platform available from NXP. This document describes how to use MCUXpresso IDE.

1 Introduction to MCUXpresso IDE

MCUXpresso IDE is an easy-to-use Eclipse-based development environment for NXP MCUs based on Arm Cortex-M cores. It provides an end-to-end solution enabling engineers to develop embedded applications from initial evaluation to final production. The MCUXpresso IDE offers advanced editing, compiling, and debugging features with the addition of MCU-specific debugging views, code trace and profiling, multicore debugging, and integrated configuration tools.

The MCUXpresso platform ecosystem includes:

- MCUXpresso IDE - a software development environment for creating applications for NXP's ARM Cortex-M based MCUs including "LPC", "Kinetis" and i.MX RT" ranges
- MCUXpresso Config Tools comprising Pins, Clocks, and Peripherals Tools that are designed to work with SDK projects and are fully integrated and installed by default
- MCUXpresso SDKs each offering a package of device support and example software extending the capability and part knowledge of MCUXpresso IDE
- The range of LPCXpresso development boards, each of which includes a built-in "LPC-Link", "LPC-Link2", or CMSIS-DAP compatible debug probe. These boards are developed in collaboration with Embedded Artists.
- The range of Tower and Freedom development boards, most of which include an OpenSDA debug circuit supporting a range of firmware options
- The range of the i.MX RT Series EVK development board, which includes an OpenSDA debug circuit supporting a range of firmware options, or a high-performance FreeLink (LPC-Link2 compatible) debug probe
- The range of EVK development boards, which include an MCU-Link debug circuit
- The standalone "LPC-Link2" debug probe
- The standalone "MCU-Link" and "MCU-Link Pro" debug probes.

This guide is intended as an introduction to using MCUXpresso IDE. It assumes that you are familiar with MCUs and software development for embedded systems.

Note: MCUXpresso IDE incorporates technology and design from LPCXpresso IDE. This means that users familiar with LPCXpresso IDE find MCUXpresso IDE looks relatively familiar.

1.1 MCUXpresso IDE overview of features

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, Kinetis, and i.MX RT). The IDE combines the best of the widely popular LPCXpresso and Kinetis Design Studio IDEs, providing a common platform for all NXP Cortex-M microcontrollers.

MCUXpresso IDE is a free toolchain providing developers with no restrictions on code or debug sizes. It 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, EVK, LPCXpresso, and custom development boards with industry-leading open source and commercial debug probes including MCU-Link, MCU-Link Pro, LPC-Link2, PEmicro, and SEGGER.

The fully featured debugger supports both SWD and JTAG debugging and features direct download to on-chip and external flash memory.

For the latest details on new features and functionality visit:

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

1.1.1 Summary of features

Complete C/C++ integrated development environment

- Eclipse-based IDE with many ease-of-use enhancements
- The IDE installs with various Eclipse plugins including:
  – Git, support for PEmicro debug probes, ARM CMSIS-Pack
- It is possible to enhance the IDE with many other Eclipse plugins
- Command-line tools are included for integration into build, test, and manufacturing systems

Industry standard GNU toolchain including:

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

Advanced project wizards

- Simple creation of preconfigured applications for specific MCUs
  – Extendable with MCUXpresso SDKs
- Device-specific support for NXP's ARM-based MCUs (including LPC, Kinetis, and i.MX RT)
- 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
- **Note**: No assembler is 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 SWD and JTAG connection via LinkServer

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

Full install and integration of third-party debug solutions from:

- PEmicro
- SEGGER J-Link

Library support

- Redlib: a small-footprint embedded C library
  – RedLib-nf: a smaller footprint library offering reduced fprintf support
  – RedLib-mb: a library variant offers offering semihosting performance
- 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
- Extendible support per device via MCUXpresso SDKs

Trace functionality

- Instruction trace via Embedded Trace Buffer (ETB) on certain Cortex-M3/M4/M7/M33-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-M0+/M3/M4/M7/M33-based MCUs when debugging via MCU-Link, MCU-Link Pro and LPC-Link2, providing functionality including:
  - Profile tracing
  - Interrupt tracing
  - Datawatch tracing
  - Printf over ITM
    - **Note**: Now, extended to work with PEmicro and SEGGER J-Link, in addition to native LinkServer

LinkServer Energy Measurement

- On LPCXpresso boards, sample power usage at adjustable rates of up to 100 ksps; average power and energy usage display option
- MCU-Link Pro or built-in implementations provide extra features, like simultaneous target supply and current measurement, dynamic range switching for increased accuracy, analog signal input, and trigger-based measurements.
  - Power Profile view providing correlated energy and trace measurements.
- Explore detailed plots of collected data in the IDE
- Export and import data for offline analysis

RTOS Debug Awareness

- GDB thread awareness for various RTOS providers: FreeRTOS, Azure ThreadX, Zephyr, and MQX
- Views for different RTOS elements: Thread list, Message queues, Semaphores, Mutexes, Event flags, Timers, Memory block pools, Memory byte pools, and so on

MCUXpresso Configuration Tools

- MCUXpresso Config Tools designed to work with SDK projects are fully integrated and installed by default

1.1.2 Supported debug probes

MCUXpresso IDE installs with built-in support for 3 debug solutions. This support includes the installation of all necessary drivers and supporting software.

Note: Certain mbed boards require a serial port driver to be recognized and this one exception must be installed separately for each board. The driver is linked from *Help -> Additional Resources -> MBED Serial Port Driver Website*.

In normal use, MCUXpresso IDE presents a similar interface and array of features for each of the solutions listed below:

Native LinkServer (including CMSIS-DAP) as also used in LPCXpresso IDE

- It comes as a separate package that is silently installed by the MCUXpresso IDE installer
- This supports various debug probes including OpenSDA programmed with CMSIS-DAP firmware, LPC-Link2, MCU-Link, and so on.
- https://community.nxp.com/message/630896

PEmicro

- This supports various debug probes including OpenSDA programmed with PEmicro compatible firmware and MultiLink and Cyclone probes
- https://www.pemicro.com/

SEGGER J-Link

- This supports various debug probes including OpenSDA programmed with J-Link compatible firmware and J-Link debug probes
- https://www.segger.com/

See Debug Solutions Overview Chapter for more details.

Note: Kinetis Freedom and Tower boards typically provide an onboard OpenSDA debug circuit. You can program this with a range of debug firmware including:

- mBed CMSIS-DAP - supported by LinkServer connections
- DAP-Link - supported by LinkServer connections (DAP-Link is the preferred choice over mBed CMSIS-DAP, when available)
- J-Link - supported by SEGGER J-Link connections
- PEmicro - supported by PEmicro connections

It is possible to change the default firmware if required. For details of the procedure and the range of supported firmware options visit: https://www.nxp.com/opensda

Tip: *Under Windows 10, OpenSDA Bootloaders might experience problems and the OpenSDA LED blinks an error code. The following article discusses the problem and how to fix it:* *https://mcuoneclipse.com/2018/04/10/ recovering-opensda-boards-with-windows-10*

1.1.3 Development boards

NXP has a large range of development boards that work seamlessly with MCUXpresso IDE including:

1.1.3.1 LPCXpresso boards for LPC

These boards provide practical and easy-to-use development hardware to use as a starting point for your LPC Cortex-M MCU-based projects.
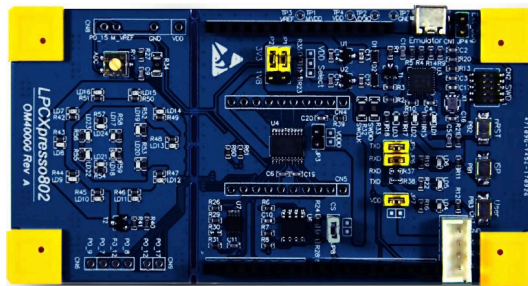

Figure 1. LPC800 series (LPCXpresso802)

Figure 2. LPCXpresso development board (LPCXpresso54608)

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

1.1.3.2 Freedom and Tower boards for Kinetis

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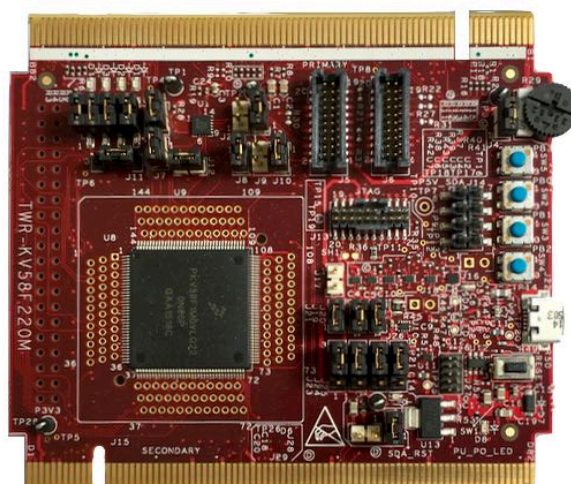

Figure 3. Tower (TWR-KV58F220M)

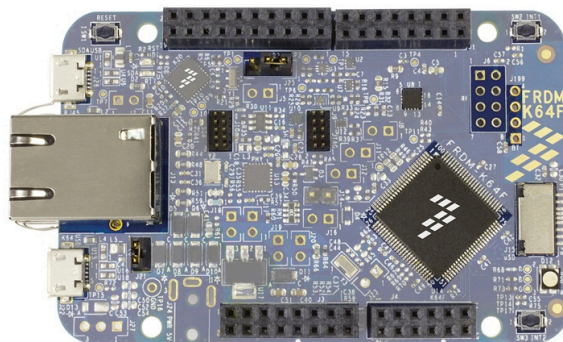For more information, visit: https://www.nxp.com/pages/:TOWER_HOME

Figure 4.   Freedom (FRDM-K64F)

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

1.1.3.3 i.MX RT Crossover processor boards

i.MX RT-based boards bring the convergence of low-power applications processors with high-performance microcontrollers.

Figure 5.   i.MX RTxxxx series (MIMXRT1050-EVK)

For more information, visit: https://www.nxp.com/pages/:IMX-RT-SERIES

Figure 6.   i.MX RTxxx series (MIMXRT600-EVK)

For more information, visit: https://www.nxp.com/pages/:IMX-RT-SERIES

2   New features in MCUXpresso IDE version 24.12

The new MCUXpresso IDE product comes with a set of improvements and bug fixes including:

Product

- Eclipse version 2023.12 (Eclipse Platform 4.30.0 / CDT 11.4.0).
- GNU ARM Embedded Toolchain version 13.2.Rel1.
  Note1: Debugging info is enforced to DWARF version 4 ("-gdwarf-4").
  Note2: The IDE does not use the "-ffreestanding" compiler flag when creating projects with NPW. GNU GCC 13 does not allow anymore the inclusion of non-freestanding library headers in freestanding environments.
- ARM CMSIS-Pack Eclipse Plug-ins 2.9.0.
- xPack Windows Build Tools (v4.4.1-2).
- Upgraded: Newer LinkServer software (v24.12.21).
- Upgraded: Newer SEGGER J-Link software (v8.12a).
- Upgraded: Newer PEmicro plugins (v5.9.5).
- Upgraded: Version v24.12 of MCUXpresso Config Tools.
- Automatic probe selection and device/board verification when using CMSIS-DAP-based probes with target information available.

SDK

- Synchronization with SDK v24.12.
  Note: Only MCUXpresso IDE packages downloaded from mcuxpresso.nxp.com are supported. GitHub SDK format is currently not supported.

2.1 Feature highlights from previous releases of MCUXpresso IDE

Product

- New IDE build for macOS with native Apple silicon support. See the download section for the new product.
- GNU Make 4.4 is now integrated into the IDE on all OSes.
- The IDE does not internally use the WMI command-line utility anymore (deprecated as of Windows 10, version 21H1). It uses PowerShell instead.
- New Welcome View designed to provide a dramatically improved out-of-box experience for new users
- Improved IDE Update capability simplifying the update procedure for all supported hosts
- Scripts to create a command line environment now supplied in DOS and Bash versions
  – Description of the use of these scripts is available in the Installation Guide
- SDK installation options improved, see SDK importing and configuration
- Windows version now uses *Busybox* (from the GNU MCU Eclipse Windows Build Tools project) to provide a Unix-like layer for GCC tools
- All previous Pro Edition features are now part of the standard Free edition, leading to the discontinuation of the Pro edition

IDE

- Support for displaying markdown (.md) files.
- Other Symbols and Images tab on a launch configuration also allows specifying extra images to be downloaded on target.
- Application Code Hub integration inside the IDE
- FreeRTOS v11 support in TAD views and in LinkServer GDB thread awareness.
- Zephyr RTOS TAD views for timers, mutexes, events and semaphores.
- Speed-up automatic generation of Makefiles
- Pass arguments to archiver and linker using response files on Windows when the 32K command line limit is exceeded
- Import ELF binary/executable. This is available from File -> Import -> C/C++ -> MCUXpresso Executable Importer.
- Added Power Profile feature aiming to correlate energy/power measurement with SWO trace. The view displays the SWO-based trace information (similar to the SWO Profile view), plus each function shows various energy consumption information. The feature is available from Eclipse Menu -> Analysis -> Power Profile.
- Added analog data traffic statistics information for energy/power-based views.
- Added FreeRTOS Task Notifications view to display task notification list for each task, including status and value properties. This included synchronization with FreeRTOS v10.4.3. For more information, refer to *MCUXpresso_IDE_FreeRTOS_Debug_Guide.pdf* documentation.
- Azure RTOS ThreadX debug awareness:
  – Ability to export trace that can be further used within TraceX Microsoft tool
  – GDB thread awareness
  – Views, similar to the FreeRTOS, for: Thread list, Message queues, Semaphores, Mutexes, Event flags, Timers, Memory block pools, and Memory byte pools
- Added offline peripheral view ("Offline Peripherals"). With this view, it is possible to inspect the peripheral registers outside of a debug session. Inspecting the reset value is possible as well, together with the rest of the register elements.
- Peripherals+ view design was changed to support register group expansion directly into the Peripherals+ view, with no extra Memory View usage. Consequently, all elements shown before in Memory View are now available directly in the Peripherals+ view: values, bitfields, and details.
- Added Energy Measurement view for energy consumption measurement.
  – Various measurement channels: voltage, current, power, and so on, at various sampling speeds, up to 100 ksps
  – Import/Export measurement
  – Zooming, panning, and annotation capabilities on graph

- – Unlimited sampling time, depending on the available space on the disk
- – Data gathering enabled/disabled by trigger signal MCU-Link Pro and on-board probes with energy measurement circuitry can use a GPIO signal as a trigger/enable (input) for the energy measurement data gathering such that data capture commences and stops based on trigger signal transitions. The Energy Measurement view includes a trigger configuration section to use this probe capability. The supported modes of operation include level-based and pulse-based transitions of the trigger signal, with additional configuration of start and stop conditions. **Note**. This feature requires firmware version MCU-LINK CMSIS-DAP v2.249 (or greater).
- Community forum accessible now from the main toolbar too (together with the older link from **Help -> MCUXpresso IDE support forum**).
- Expressions added in Global Variables are now persistent between debug sessions.
- Added new control to manage the maximum number of child expressions that are evaluated in advance by the Live Variables service. This improves the Global Variables window responsiveness, for instance, when displaying large structures. New control available on Eclipse Preferences -> MCUXpresso IDE -> Debug Options -> "Number of subexpressions proactively evaluated by Live Variables service". The default is 2 set as depth.
- Added Save info for support option to help report an issue by gathering MCU IDE environment information.
- New Plugin SDK mechanism that provides a simpler flow for the selection and installation of MCUXpresso IDE SDKs
- New Dark theme provides a low-light interface that displays mostly dark surfaces that may be more relaxing on the eye
- Improved Image information view
- Improved Installed SDK operations
- Improved Code size
  - – The code size of debug builds of SDK projects has been reduced by decreasing the overhead of the assert() function, which is commonly called by SDK functions.
- Added support for handling more complex specifications of dependencies between SDK components.
- Heap and Stack view for all debug solutions
  - – shows usage against allocated managed linker script RAM allocation for bare metal projects
  - – Live Heap updates and stack when paused
- Image information view extends and replaces the Symbol Browser
  - – incorporating detailed memory usage plus hyperlinked Memory Content and Static Call Graph display
- Revamped Develop perspective
- Editor Syntax highlighting for linker scripts, linker templates and debug map files
  - – Providing linked navigation of file contents
- Redesigned Quickstart panel
  - – Quick Start panel -> Quick Settings now displays the current settings for Library
  - – Links for Dedicated debug operations for all supported Debug Solutions
- Faults view automatically displayed (for LinkServer) should a CPU fault occur
- Improved Registers view with enhanced display and grouping options
- Launch configurations are now only automatically generated for the selected build configuration
- Project Memory configuration can now be edited *in place* for settings and wizards
- Project Explorer view enhanced to display current project build configuration for the selected project (also displayed in Quickstart view)
- Support for new MCUs based on the ARM Cortex M33

Projects

- A specific SVD file can be assigned to a project that can be used afterward within OfflinePeripherals or Peripherals+ views.

- Imported or new [projects](#) now expand to show the source file containing the main function and also open this file within the editor
- Improved display of Components in [New Project Wizard](#)
- Quick Start panel -> [Quick Settings](#) now displays the current settings
- Project association with an SDK (MCU) can now be flexibly managed, maintaining existing memory configuration if desired see [Project configuration](#)
- Many enhancements for improved [Project sharing](#) including:
  – Drag and Drop of projects for import and export
  – Options for project local inclusion of: SDK part support, flash drivers, and LinkServer connect and reset scripts
- [Project virtual nodes](#) introduced to enable easy visibility and editing of project configurations
- [Project GUI Flash Tool](#) for all debug solutions delivered via project launch configurations

Debug

- MCU-Link probes firmware can now be updated automatically from within Probes Discovered dialog.
- The LinkServer debug solution is now installed as a separate package, incorporating all the LinkServer-specific support files that used to be part of the MCUXpresso IDE installation directory.
- Also see the supplied Readme document for further information and details of bug fixes and improvements. This document is located within the MCUXpresso IDE installation folder.
- LinkServer LPC-Link2 firmware version being softloaded is v5.460, which offers support for powering RT1xxx EVK boards (that incorporate on-board debug probes based on LPC-Link2 hardware) through the USB debug connection.
- Added Zephyr RTOS Awareness:
  – Added GDB thread awareness for LinkServer debug connection.
  – Added Threads view.
- The target configuration for SWO trace is now optional. The default setting is to have the IDE perform the necessary configuration for the SWO trace. However, the user can choose to disable this functionality and rely on the target configuration performed by the application. The "SWO configured by IDE" (depending on the debug probe) checkbox in the "SWO Trace Config" view -> "Change" button -> "Clock speed configuration" dialog controls the behavior. For more information, refer to *MCUXpresso_IDE_SWO_Trace.pdf* documentation.
- A new console named [SWO and Trace console](#) displays all configuration and register settings performed while configuring SWO.
- [J-Link] Added the possibility to connect to a remote gdb server. In launch configuration -> J-Link Debugger tab -> GDB Server Settings, use the Server execution option to set a remote server.
- UART console is the default debug console when importing a project.
- The IDE displays inside the Probes Discovered dialog the nickname assigned to a PEmicro or J-Link debug probe. Each solution offers a specific procedure for assigning a nickname, therefore it is necessary to follow the appropriate documentation.
- Auto-debug secondary project(s) for multicore projects option becomes the default option for multicore debug purpose for LinkServer debug connection. That means, in the case of multicore projects in which the primary project refers to one or several secondary projects, initiating debugging with the primary project results in the automatic start of debug sessions for secondary projects.
  – Option is set by default on: **Window -> Preferences -> MCUXpresso IDE -> Debug Options -> LinkServer Options -> Miscellaneous -> Enable auto-debug secondary project(s) for multicore projects**
  – If you don't want to have this feature enabled (so if you want to start debug sessions for each core independently), uncheck this option.
- Similar, auto-debug secondary project(s) for multicore projects option becomes the default option for multicore debug purpose for PEmicro too. The option is enabled by default on: **Window -> Preferences ->**

MCUXpresso IDE -> Debug Options -> PEMicro Options -> Enable auto-debug secondary project(s) for multicore projects and also for J-Link debug session, set by default on: **Window -> Preferences -> MCUXpresso IDE -> Debug Options -> J-Link Options -> Enable Auto-debug secondary project(s) for multicore projects**.

- Firmware version check on MCU-Link probes
- Most LinkServer Flash programming now implements a *Verify Same* operation for any flash sector that is unchanged from previous debug operations
- LinkServer multicore debug operations can now be started via a single click
- Reworked Live global variables graphing offering improvements to variable selection and display
- Reworked SWO Interrupt trace
- LinkServer LPC-Link2 firmware now softloaded as v5.361, which offers improved debug control through target reset
- Redesigned LinkServer Launch configuration dialog offering improved functionality and ease of use
  - This is reflected in a new LinkServer Launch configuration icon
- New launch configuration tab for all debug solutions to allow the loading of Debug symbols from additional images
- Improved performance for Single Stepping LinkServer debug connections
- Implemented support for SWO Trace on Cortex-M33-based MCUs
- Live global variables are now available for SEGGER JLINK and PEmicro debug probes in addition to LinkServer LPC-Link2
- LinkServer internal flash drivers prioritized over supplied SDK drivers
- Debug shortcut buttons now multicore aware ensuring secondary project attach settings are observed
- Improved Faults view now displays Fault Address when available
- SWO trace features are now available for SEGGER JLINK and PEmicro debug probes in addition to LinkServer LPC-Link2 and MCU-Link
- LinkServer debug probes now support selection via their serial number (for command line use)
- Increased integration of our supported debug solutions including:
  - GUI Flash Tool is re-architected to provide support for LinkServer, PEmicro, and SEGGER debug solutions
    - Offering binary flash programming and erase capability for all supported debug solutions
    - With a feature set integrated into the Quickstart panel, project Launch Configurations, and from the IDE as before
  - Instruction trace is seamlessly supported by LinkServer, PEmicro, and SEGGER debug solutions
- LinkServer semihosted operations including printf are further optimized to deliver approximately double the performance of the previous release
- Re-architected semihosting mechanism via new library variant *Redlib MB* and LinkServer, which can deliver both a further increase in performance and no disruption to code executing with time critical interrupts
- LinkServer Graphing of global variable values
  - *Live* global variable values can now be traced both in graphical and tabular forms
- Peripheral display filtering to simplify complex peripheral views

LinkServer Flash Programming

- External flash drivers for RT116x, RT117x, RT500 and RT600 available as examples in <install_dir>/ide/Examples/Flashdrivers/NXP/iMXRT.
- SFDP Flash drivers extended to support i.MX RT MCUs
- Programming of data flash regions on certain Kinetis parts is now supported
- Improved flash programming performance and reliability
- LinkServer Enhanced external SPIFI/QSPI programming via self-configuring flash drivers
  - using JEDEC SFDP (Serial Flash Discovery Protocol) available for LPC18/43, LPC546xx, LPC540xx, i.MX RTxxx, i.MX RTxxxx

UG10055

User guide **Rev. 4 — 14 January 2025** Document feedback

12 / 316

SDK

- Extended integration with ARM CMSIS-Pack Eclipse Plug-ins. Now, the ARM CMSIS-Pack Eclipse Plug-in manages the addition of a new Open-CMSIS-Pack component. This brings support for:
  - Components dependency
  - Multiple component selection
  - Automatically check dependencies in the new multiple-component selection view
  - Copy configuration and template files to the project
- Added support for selecting library type in SDK CLI. Now redlib, newlib, and newlib_nano can be selected as options when generating a project. Check MCUXpresso_IDE_Command_Line_User_Guide.pdf for details.
- Provide CLI utility to merge sub-manifest files: added the manifest.merge command:
  - Running the headless mode with -help manifest.merge generates a template property file, which contains the following:
    - manifest.xml (location of the manifest containing references to sub-manifests)
    - repo.location (repository where the manifest specified in the manifest.xml property is located).
    - merged.manifest.xml (location of the result manifest file).
  - Specifying all properties from the template file is required for the command to run.
  - The manifest specified in the manifest.xml file must be inside the specified repository.
- Complex dependencies: support for < not > operator in the dependency conditions.
- Added the possibility to explore Open-CMSIS packs and import (middleware) components into an Eclipse project. In the current version of the feature, users shall manually add component dependencies. A future version will automatically resolve dependencies and add them to the project:
  - CMSIS-Pack Management for Eclipse created by ARM plugin included in the product for packs management: Perspective -> Open Perspective -> Other -> CMSIS-Pack Manager. From the Packs view (toolbar) you can: Reload, Check for updates on Web, Import Packs from disk, and so on.
  - Once the desired packs are available, you can add them to the Eclipse project by right-clicking the project entry in Project Explorer -> SDK Management -> Add components from Open-CMSIS-Pack and select the desired one from the "Add Open-CMSIS component to project" wizard. The component is then available in the Project Explorer view (the sources being linked to the original pack location), and also in the Project Settings with details about its hierarchical path.
  - You can delete components from the project by selecting the component from Project Explorer -> <select project> -> Project Settings -> Open-CMSIS components, right-click it, and choose "Delete Open-CMSIS component".
- Support to allow sub-manifest under the same SDK.
  - Adapted SDK Creator for creating split manifests.
  - Updated "Contribute project to SDK Git repository" feature to work with the new sub-manifests.
- A GitHub SDK repository can be imported and managed within IDE, integrating the existing SDK management functionality and Git capability of MCUXpresso IDE.
  - Support to import an already cloned repository;
  - Support to install/clone a remote repository (using west init and west update);
  - Ability to contribute a project back to SDK GitHub repository;
- Improved SDK installation and refresh time.
- Redesigned New and Import SDK example wizard.
  - incorporating Error Decorators
- SDK part support is now generated within the current workspace eliminating issues that could arise if launching multiple IDEs.
  - Part support is intelligently regenerated when required, avoiding unnecessary delays
- SDK drag-and-drop location can now be set via a workspace preference.
- Installed SDK view improved to display version information and enhanced tooltips.
- SDK Manifest Analyser to provide visibility of SDK XML description.

- Easy access to <u>Embedded Documentation</u>.
- Extension of SDK Component Management to allow <u>Project Refresh</u>.
  - Improved SDK Component Management
- General Improvements in SDK Handling including:
  - SDK version string now present and reported in SDK view
  - User selection of versioned internal XML descriptions (enabled via preference)
  - Better automatic support for SDKs with overlapping capabilities

3 IDE overview

The following chapter provides a high-level overview of the features offered by MCUXpresso IDE (often referred to as the IDE).

3.1 Workspaces

MCUXpresso IDE prompts you to select a workspace when it is launched for the first time, as indicated in Figure 7.
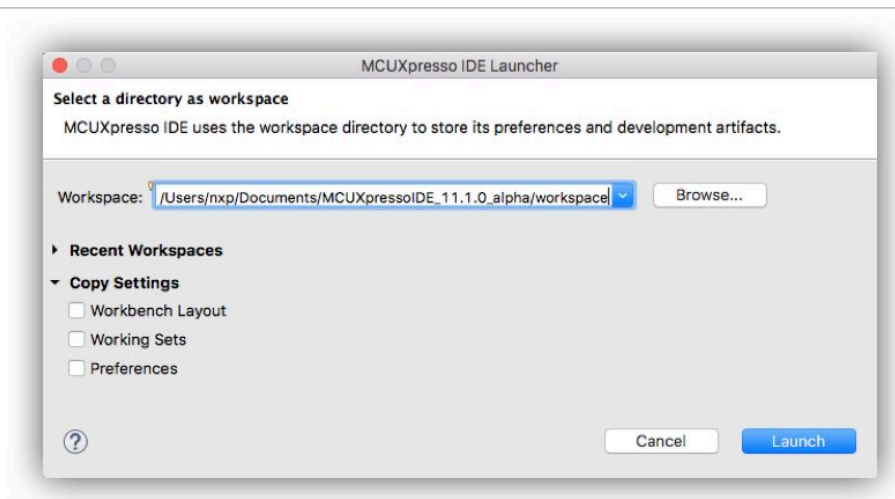
Figure 7. Workspace selection

A workspace is simply a filing system directory used to store projects and data, and for new installations, it is typically recommended to accept the default location. If you tick the **Use this as the default and do not ask again** option, then MCUXpresso IDE always starts up with the chosen workspace opened; otherwise, a prompt to choose a workspace always appear.

MCUXpresso IDE can only access a single workspace at a time but many Workspaces may be used. You may change the workspace that MCUXpresso IDE uses, via the **File -> Switch workspace** option.

Tip: *It is possible to run multiple instances of the IDE in parallel, with each instance accessing a different workspace.*

Note: when changing workspaces, you may choose to copy settings (preferences) from an existing workspace to the new workspace using the various *Copy Settings* tick box options.

3.2 Welcome view

MCUXpresso IDE version 11.1.0 launches with a new Welcome View. This View is intended to help reduce the learning curve for new users by offering links and help for common tasks and IDE operations.
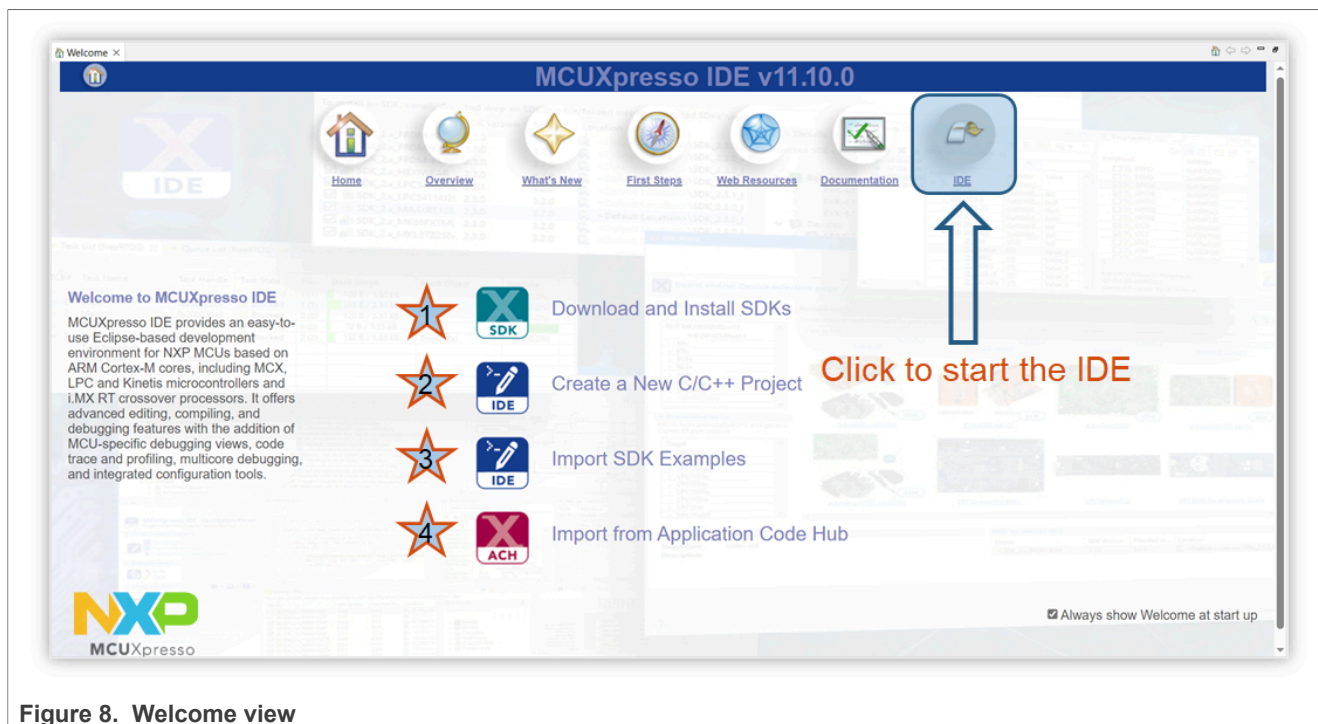
Figure 8. Welcome view

1. Click to select, download, and install a Plugin SDK
2. Click to be guided through Creating a new project
3. Click to be guided through Importing an example
4. Click to open the Application Code Hub wizard

Since the installation of an SDK adds support for most NXP MCUs to the IDE, the first option is to guide the user to a new Plugin SDK installation view. From this view, they can select, download, and install an SDK for a required MCU or (development board) with just a few clicks. This screen also contains guided workflows for creating New Projects and Installing SDK Examples.

Across the top of this View are links to Features and Resources including a *jump to* IDE link (highlighted above) which takes the user directly to the main development view (Perspective) of the IDE.

Note: This Welcome View is provided by and so incorporates standard icons to maximize, minimize, restore, and so on, like all Eclipse views. Since this view is intended to be used *full screen*, minimizing or restoring may lead to a poor screen layout. The recommended way to switch back to the main IDE Develop view is via the IDE link or by closing this Welcome Screen. You can restore the Welcome View at any time by clicking the Home Icon within the main Eclipse Icon view.

It is also possible to disable the Welcome view from appearing at startup by unchecking the box at the lower right of the view.

3.3 Documentation and help

In addition to the help features offered from the Welcome View are a comprehensive suite of Guides.

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

MCUXpresso IDE documentation comprises a suite of documents including:

- MCUXpresso IDE Installation Guide
- MCUXpresso IDE User Guide (this document)
- MCUXpresso IDE SWO Trace Guide
- MCUXpresso IDE Instruction Trace Guide
- MCUXpresso IDE LinkServer Energy Measurement Guide
- MCUXpresso IDE FreeRTOS Debug Guide
- MCUXpresso IDE Azure RTOS ThreadX Debug Guide
- MCUXpresso IDE Zephyr RTOS Debug Guide
- MCUXpresso IDE MQX RTOS Debug Guide
- MCUXpresso (IDE) Config Tools User's Guide

The installation folder of MCUXpresso IDE includes these guides in *PDF* format as well.

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

You can also find related web links at *Help -> Additional resources*, as shown below:
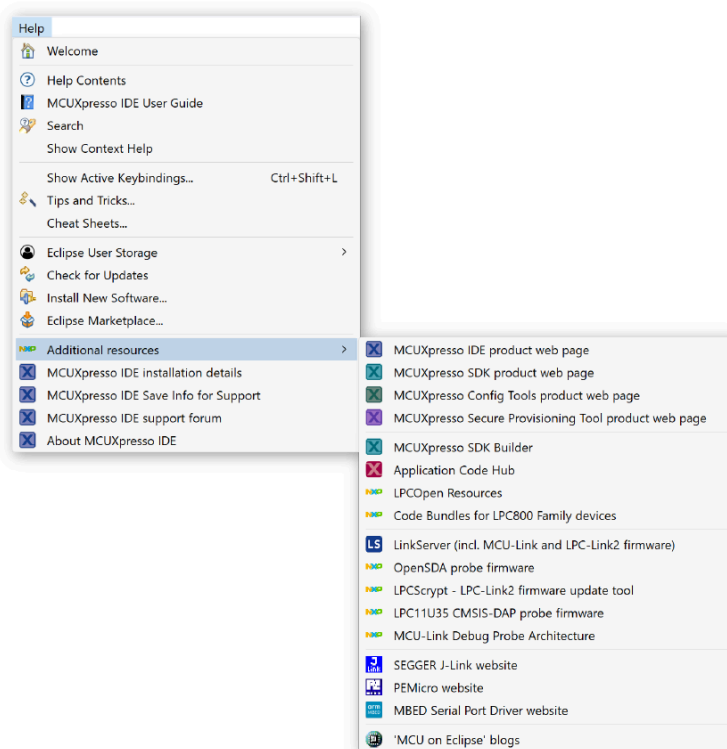
Figure 9. Additional resources

3.4 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. It is possible to open, move (drag), dock, and close the Views, and also to save and restore the layout of the currently displayed Views.

Typically, MCUXpresso IDE operates using the single **Develop Perspective**, under which both code development and debug sessions operate as shown in Figure 12. This single perspective simplifies the Eclipse environment but at the cost of slightly reducing the amount of information displayed on screen.

Alternatively, 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.

Note: when within the debug perspective, the concept of a selected project remains. The *Blue Debug* button tooltip displays this selected project. Also, if you start a debug operation within the Debug perspective and then you make a switch to the Develop perspective, the IDE automatically opens a debug stack view to display the active debug connection.

You can manually switch between Perspectives using the Perspective icons in the top right of the MCUXpresso IDE window, as shown in Figure 10.

Figure 10. Perspective selection

The user can select new perspectives by clicking the view+ icon. After selecting a view, its icon appears within the horizontal section as highlighted above.

You can also rearrange all Views in a Perspective to match your specific requirements by dragging and dropping. If you accidentally close a View, you can restore it by selecting it from the **Window -> Show View** dialog. It is also possible to restore the default layout for a perspective at any time via **Window -> Perspective -> Reset Perspective**.

Commonly used Views for Analysis (Trace) and RTOS debugging have been made more readily available via top-level dropdown menus as shown below:
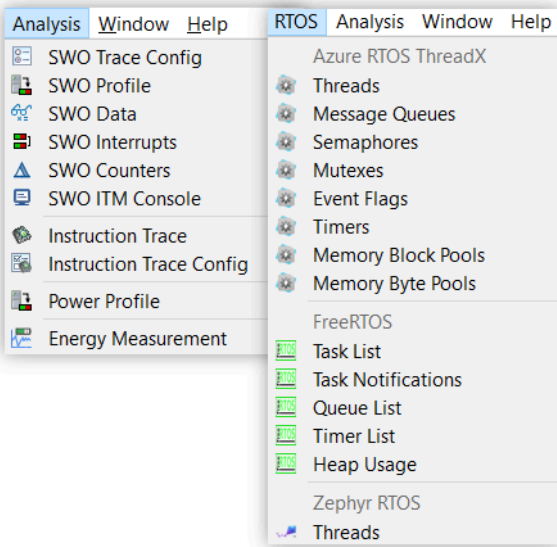
Figure 11. Additional views

Once selected, these additional views appear alongside the Console view but can be relocated as desired.

Note: The rest of this guide assumes that the user uses the default Develop Perspective.

3.5 Major components of the Develop perspective
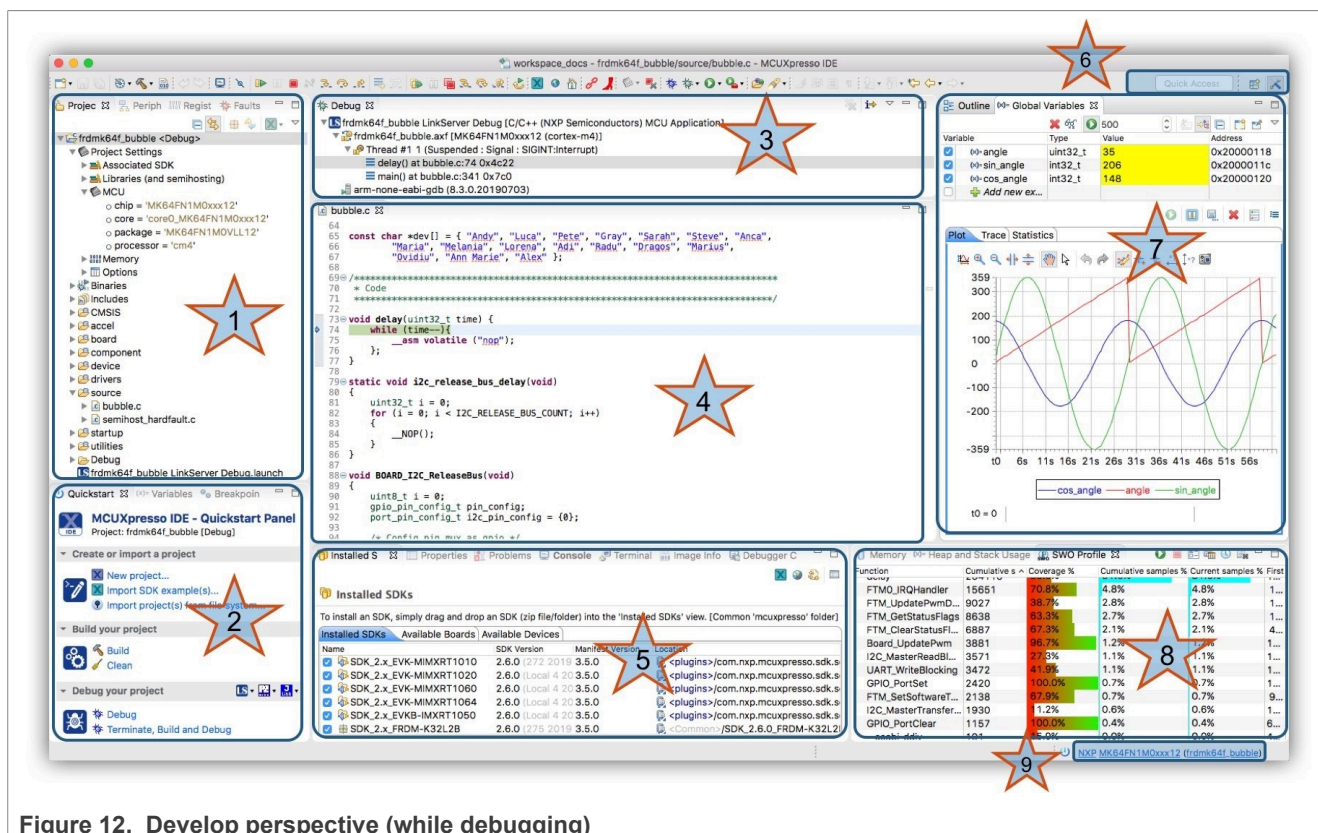
Figure 12. Develop perspective (while debugging)

1. **Project Explorer** / **Peripherals** / **Registers** / **Faults**
   - The **Project Explorer view** (shown) gives you a view of all the projects within your current Workspace.
     – Many editing and configuration features are available from this view including new Project sharing options and Virtual nodes
   - When debugging, the **Peripherals** view allows you to display a list of the MCU Peripherals and project memory regions. **Note**: depending on your MCUs configuration, some peripherals may not be powered/clocked, and therefore the view does not display their content.
   - When debugging, the improved **Registers** view allows you to view the Registers and their content within the CPU of your MCU.
     – The view also displays pseudo-registers such as 'cycle delta', which shows the calculated number of cycles since the last pause
   - Also displayed here is the **Faults** view, which appears automatically if a CPU Fault (such as hard fault) occurs. This view decodes CPU registers to provide detailed information indicating the reason for the fault occurring.

2. **Quickstart** / **Variables** / **Breakpoints**
   - On the lower left of the window, the **Quickstart Panel View** (shown) has fast links to commonly used features. From here you can launch various wizards including New Project, Import projects from SDK, and also from the file system plus options such as Build, Debug, Export, and so on. The large icon in each section performs the first option in the group, that is, New project, Build, Debug. Also, the Debug group contains debug solution-specific Debug shortcut buttons
     – **Note**: This Panel is essential to the operation of MCUXpresso IDE and so it is not possible to remove it from the perspective.
   - 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](#) and [Watchpoints](#)

3. **Debug**
   - The Debug View appears when you are [Debugging](#) your projects. This view shows you the debug stack, in the "stopped/paused" state you can click within the stack and inspect items in scope such as local variables.

4. **Editor**
   - Centrally located is the **Editor**, which allows the creation and editing of source code and other text files. When debugging, this is where you can see the code that 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 from source to assembly instructions. Clicking in the left margin sets and deletes [Breakpoints](#)
     - [Enhanced editors](#) provides structure, keyword, and linkage for debug Map files, Linker Script, and Linker Template files.

5. **Console** / **Installed SDKs** / **Problems** / **Trace Views** / **Power Measurement**
   - On the mid-lower of the window are Console, Installed SDK, Problems Views, and so on. The Console View displays status information on compilation and debug operations, and displaying semihosted program output.
   - The [Installed SDK](#) view (shown) enables the management of installed SDKs. You can also add new SDKs as Plugins, via Drag and Drop, or Copy and Paste. This view also provides other SDK management features including unzip, explore, and delete. Use the Outline view to view details of any selected SDK.
     - The user can browse and extract SDK Documentation
   - The **Problems View** shows all compiler errors and warnings and allows easy navigation to the error location in the Editor View.
   - The **Image Information View**
     - This [Image Information](#) view provides detailed information on an image (or object) static memory footprint (usage and content).

6. **Quick Access** / **Perspective Selection**
   - Enables quick access to features such as views, perspectives, and so on. For example, enter 'Error' to view and open the Error Log of the IDE, or 'Trace' to view and open the various LinkServer Trace views.
   - Perspective Selection allows you to switch between the various defined perspectives.

7. **Outline** / **Global Variables**
   - The **Outline** View allows you to quickly locate symbols, declarations, and functions within the editor view. This view can also display details of any SDK selected in the Installed SDK view.
   - Sitting in parallel is the **Global Variables View** (shown) which allows you to see and edit the values of Global variables.
     - Use the [Live variables](#) and [Variable graphing](#) features to monitor variables while the target is running.

8. **Memory** / **Heap and Stack** / **Trace**
   - The **Memory View** provides a range of options for viewing target memory
   - The **Heap and Stack View** enables easy monitoring of [Heap and Stack](#) values for bare metal projects.
     - Warnings are given when preset limits are approached or exceeded
   - **Trace Views**
     - Trace Views including SWO Trace (Profiling shown), Instruction Trace, and Power are not shown on this screenshot. However, you can select these views when required from the Analysis Menu. For more information on Trace functionality, see the *MCUXpresso IDE SWO Trace Guide* and/or the *MCUXpresso IDE Instruction Trace Guide* and/or the *MCUXpresso IDE LinkServer Power Measurement Guide*.
     - The **SWO Trace Views** allow you to gather and display runtime information using the SWO/SWV technology that is part of Cortex-M0+/M3/M4/M7/M33-based parts.
     - The **Instruction Trace view** on certain MCUs, you can capture and view instruction trace data downloaded from the Embedded Trace Buffer (ETB) or Micro Trace Buffer (MTB) of the MCU.

– The **Power Measurement View**, this view can display real-time target power usage. For more information, see the *MCUXpresso IDE Power Measurement Guide*.

9. **Status Bar Shortcuts**
   • Various useful shortcuts, for example, to open the workspace of a project or to open a terminal at the location of the project with the environment of the IDE. Hover here to see tooltips that explain the various options.

3.5.1 Dark theme

MCUXpresso IDE contains support for a *Dark Theme*. Dark Theme is a Workspace preference that the user can select from *Window -> Preferences -> Appearance -> Theme* followed by a selection from the dropdown menu.
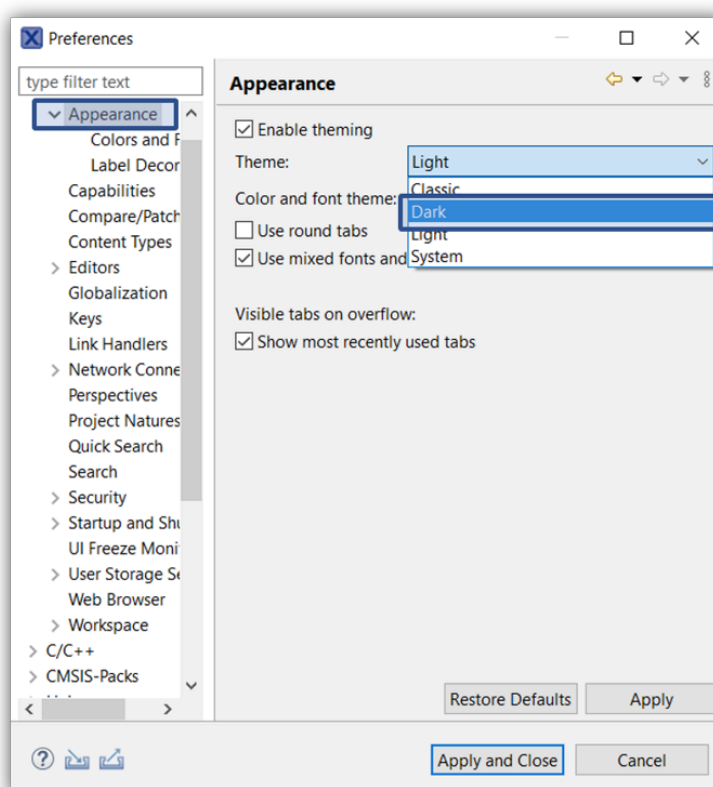
Figure 13. Appearance preference

When selected, a Dark theme is used to render the perspective and appears similar to the image below:
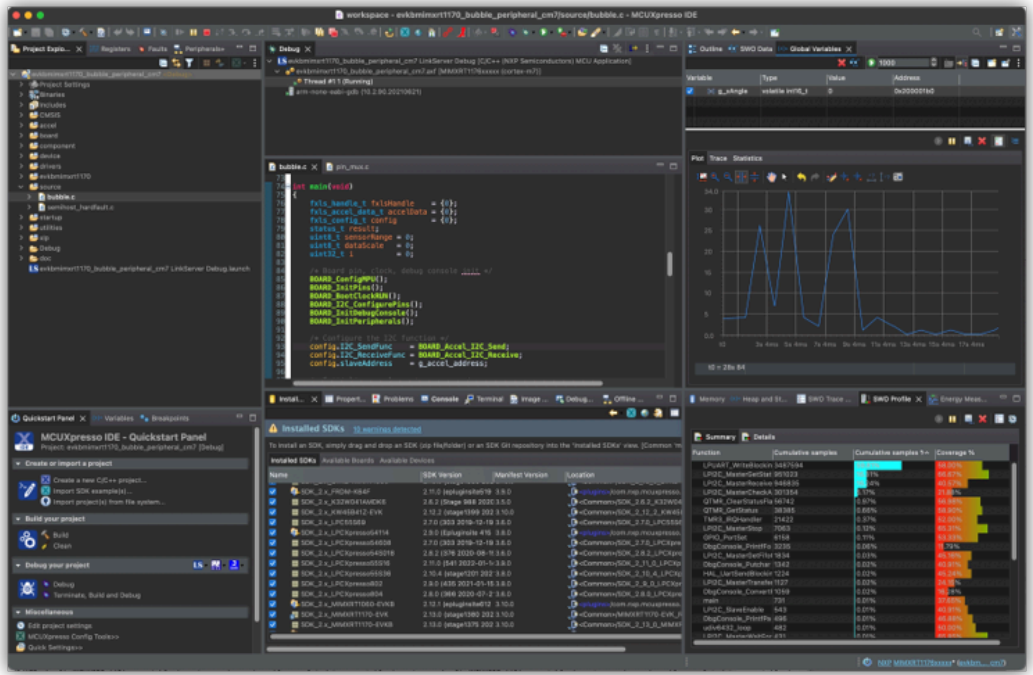
Figure 14. Develop perspective dark

Note: An IDE restart *File -> Restart* is required for the perspective to display correctly.

3.6 The Quickstart Panel

A key feature of MCUXpresso IDE is the **Quickstart** Panel - which is frequently referenced in this document. The Quickstart panel is designed to bring together many of the common IDE features and operations including links to Project Creation, Project Building, Project Debug, and Miscellaneous common Project operations.

It is **strongly recommended** that this panel be used to perform the supported MCUXpresso IDE operations described below since many underlying Eclipse features are enhanced when accessed in this way to improve and simplify the user experience.

Features of the Panel are highlighted and described below:
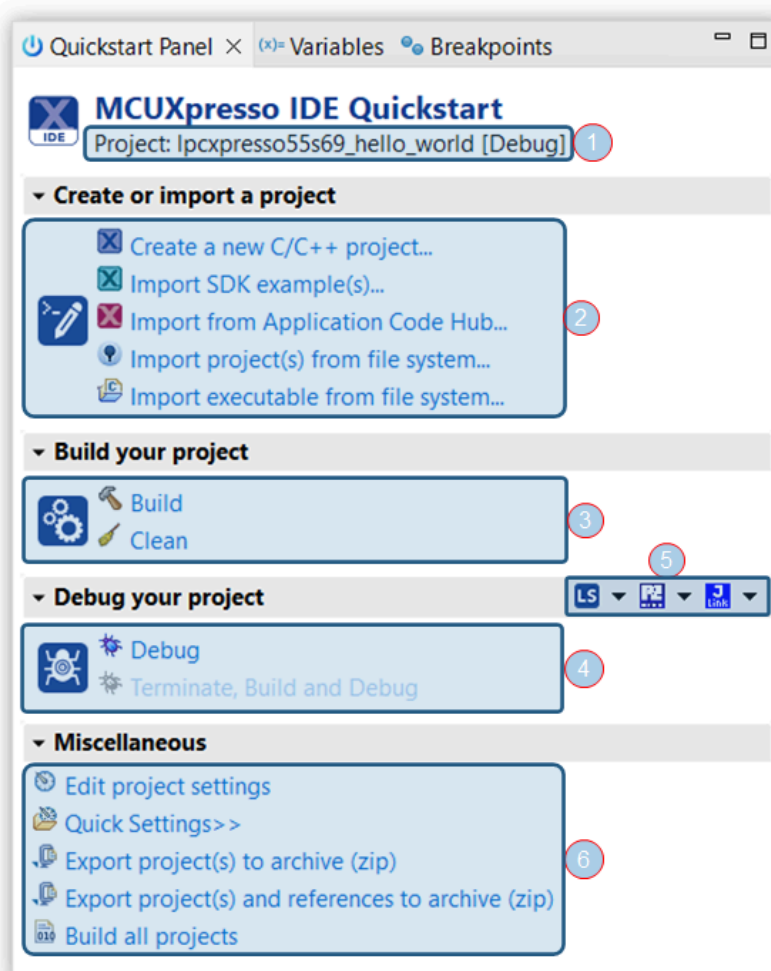
Figure 15. The Quickstart panel

Tip: *The Large Icon performs the action of the first button in the group.*

Where:

1. Shows the Project currently selected within the *Project Explorer* view. Build, Debug, and Miscellaneous operations are performed on this Project
2. Links to Create new project, Import SDK example, Import from Application Code Hub, Import from file system and Import executable
3. Build (or Clean) the currently selected Project
   - See progress and results within the Console view
4. Debug the currently selected Project
   - Clicking Debug will by default Build the project (if necessary), perform a Debug Probe Discovery, create a default Launch Configuration (if necessary) and if successful, begin the debug session.
   - Terminate, Build and Debug terminates the existing Debug session for the selected project, and then performs another debug operation. It is intended to be used for iterative source code fixes and debug retry operations
5. Debug shortcuts offer a range of debug operations for specific vendor Debug Solutions
6. The Miscellaneous section offers a range of options and shortcuts
   - Edit project settings is a shortcut equivalent to a right click a project and then selecting *Properties*

- **MCUXpresso Config** Tools offers shortcuts to launch one of the Config tools for the selected project
- **Quick Settings** offers a range of options for the currently selected project
- Export the selected Project (and References) to the file system. See also additional information on **Sharing projects**
  - **–** This feature requires that selecting the project at the top level within Project Explorer
- Build the *Active* Build Configuration of all projects within the current workspace.

Tip: *If the Quickstart panel has become hidden, then in the menu bar at the top of the IDE, select Window -> Show View -> MCUXpresso IDE and double-click Quickstart.*

The Quickstart panel is directly linked to active selection from Project Explorer, which controls the enablement state of various actions within the view. However, when the Quickstart panel is visible it also controls automatic selection of the debugged project when encountering various debug launch events. You can adapt the behavior by accessing the appropriate preference page: *Window -> Preferences -> MCUXpresso IDE -> Quickstart Panel*. The project associated with the active debug session can be autoselected in Project Explorer when adding, terminating, or removing a launch from the Debug view. If a child resource of the project is already selected when a launch event occurs, selection does not change.
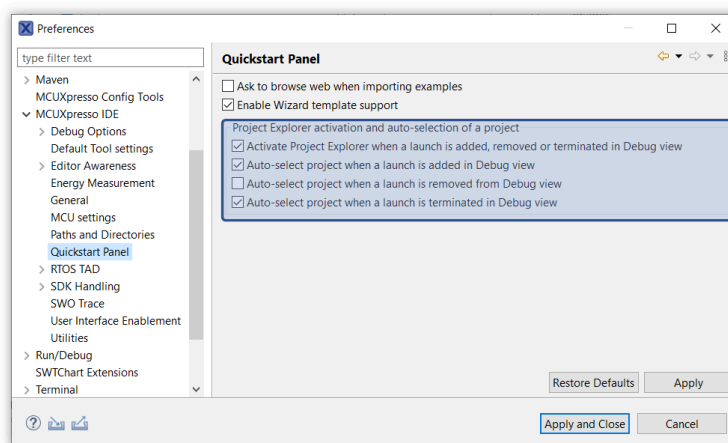
Figure 16. Quickstart panel preferences

3.7 Project Explorer and new projects

The version of Eclipse underlying MCUXpresso IDE incorporates some new **Project Explorer** functionality that is seen only when there are **no projects** within the chosen Workspace - as shown below:
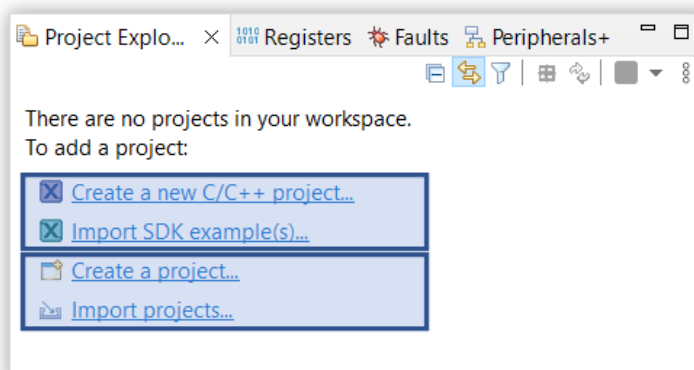
Figure 17. Project explorer empty

The first two options here are directly equivalent to the first two operations offered via the **Quickstart** panel. It is recommended to use the **Quickstart** in preference to the remaining options since this ensures that MCUXpresso IDE wizards and functionality are used.

Note: Due to this Eclipse feature, the Drag and Drop functionality to the Project Explorer view is unavailable until after creating or importing the first project.

New or Imported Projects appear in the Project Explorer view. A newly created project automatically expands to show the source file containing the main function. This source file is also opened into the editor for convenience as shown below.
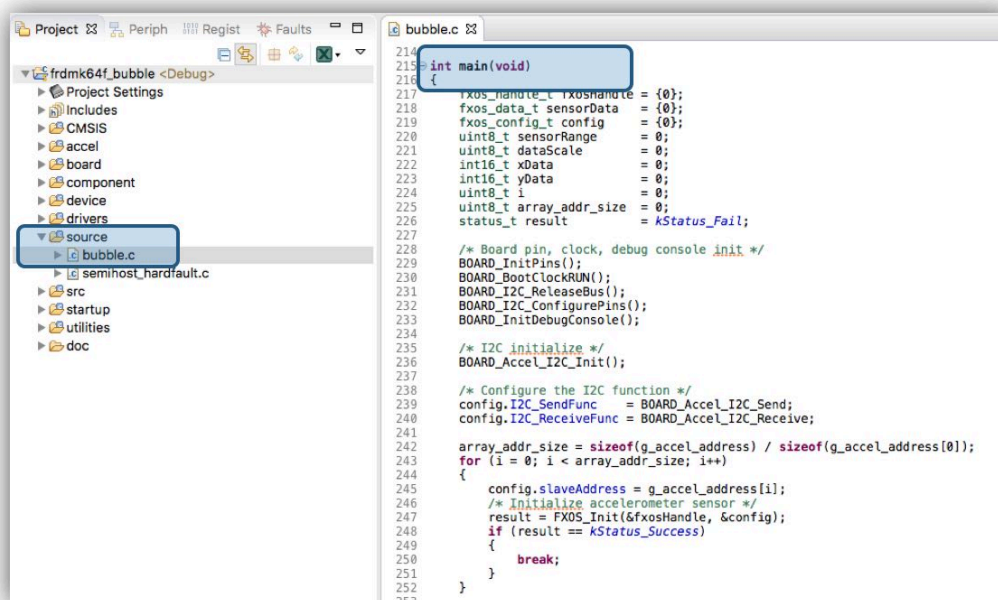
Figure 18. New or imported project

3.8 Updating MCUXpresso IDE

MCUXpresso IDE incorporates the facility to update an installation to add new features, updates, and/or to roll out bug fixes, and so on. To facilitate this mechanism, MCUXpresso IDE version internals locate key components with Eclipse-style plugins.

Tip: *Locating low-level components can be difficult due to both the complex directory structure but also because component locations may change after performing an update. Therefore, to simplify the experience, some soft links are available within the install_dir/ide as discussed in the section below "Locating IDE Components"*

By default, when NXP releases an update, a notification of the availability appears at the bottom of the screen.
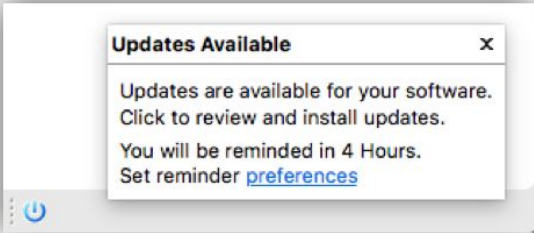
Figure 19. Update notification

Alternatively, you can check for updates via *Help -> Check for Updates*. If updates are available, a dialog similar to the one shown below appears:
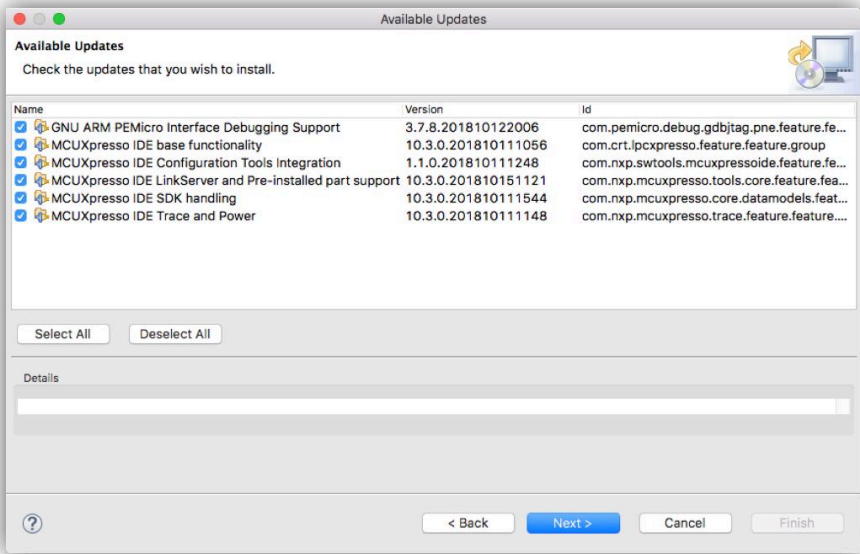
Figure 20. Updating MCUXpresso IDE components

Simply ensure that the required updates are checked and click *Next*. At this point, the components are downloaded and installed into MCUXpresso IDE. After installation, a restart is required before the new features are available.

Note: In addition to updates for MCUXpresso IDE, updates to the MCUXpresso Config tools and PEMicro debug solution are also delivered using this mechanism.

Major product releases are only delivered as full product installations since these are typically based on newer versions of Eclipse.

3.8.1 Locating IDE components

MCUXpresso IDE consists of many components, some of which may be used independently from the IDE. Also included are documents, examples, scripts, drivers, and so on, that may need to be referenced from within the IDE.

Due to the structural changes introduced in MCUXpresso IDE version 10.3.0, the paths for certain items may be different from previous releases and may change after a product update (and also be quite long). For example, the IDE binaries folder is now at a location of the form:

```
<install_dir>/ide//plugins/
com.nxp.mcuxpresso.tools.bin.macosx_11.1.0.201911211415/binaries
```

MCUXpresso IDE version 11.9.0 introduced another important change: LinkServer software debug probe support is now added in MCUXpresso IDE by installing [NXP LinkServer](#) product. As a result, LinkServer-specific support files are no longer in folders like IDE binaries. LinkServer is installed at the same folder level as the MCUXpresso IDE.

Therefore, to simplify the location of certain folders, shortcuts (or symbolic links) are installed into the *ide* directory within the installation directory of the product. You can use these links directly to locate components or items, or within script paths.

Shortcuts are available for the following directories:

- binaries -> install_dir/ide/binaries
- Examples -> install_dir/ide/Examples
- Wizards -> install_dir/ide/Wizards
- tools -> install_dir/ide/tools
- LinkServer -> install_dir/ide/LinkServer

In practice, these links allow paths to be used unchanged from earlier versions of MCUXpresso IDE, yet always reference the latest plugin components.

4 Part support overview (preinstalled and via SDKs)

To support a particular MCU (or family of MCUs) and any associated development boards, several elements are required. These break down into:

- Startup code
  - This code handles specific features required by the MCU
- Memory Map knowledge
  - The addresses, sizes, and types of all memory regions
- Peripheral knowledge
  - Detailed information allowing the MCUs peripherals registers to be viewed and edited
- Flash Drivers
  - Routines to program the on and off-chip Flash devices of the MCU as efficiently as possible
- Debug capabilities
  - Knowledge of the MCU debug interfaces and features (for example, SWO, ETB)
- *Example code* (this is not strictly required or a part support element)
  - Code to demonstrate the features of the particular MCU and supporting drivers

Collectively, this data is known as *Part Support*, MCUXpresso IDE uses these data elements for populating its wizards and for built-in intelligence features, such as the automatic generation of linker scripts, and so on.

MCUXpresso IDE installs with a base set of parts support primarily for older LPC Devices (Preinstalled). Knowledge of later devices such as the LPC5xxxx, Kinetis, i.MX RTxxx, and so on, must be provided to the IDE via the installation of an SDK

4.1 Preinstalled part support

The IDE installs with an enhanced version of the part support as provided with the older NXP IDE *LPCXpresso IDE v8.2.2*. This provides support for most LPC Cortex-M-based parts 'out of the box'. This is known as preinstalled part support. In general, SDKs are not available for these older parts. However, you can use the LPC5410x and LPC5411x part families with either Preinstalled Part Support or SDK Part support.

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

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

Find further information at:

https://www.nxp.com/lpcopen

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

4.1.1 Differences in preinstalled and SDK part handling

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

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

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

Note: When exporting or sharing projects created with Preinstalled part support, no special actions are required, since other installations of MCUXpresso IDE also contain the required part support. For sharing projects created from SDKs, see Sharing projects.

4.1.2 Viewing preinstalled part support

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

To explore the range of preinstalled MCUs simply click 'Create a new C/C++ project' in the **Quickstart** panel. This opens a page similar to the image below:
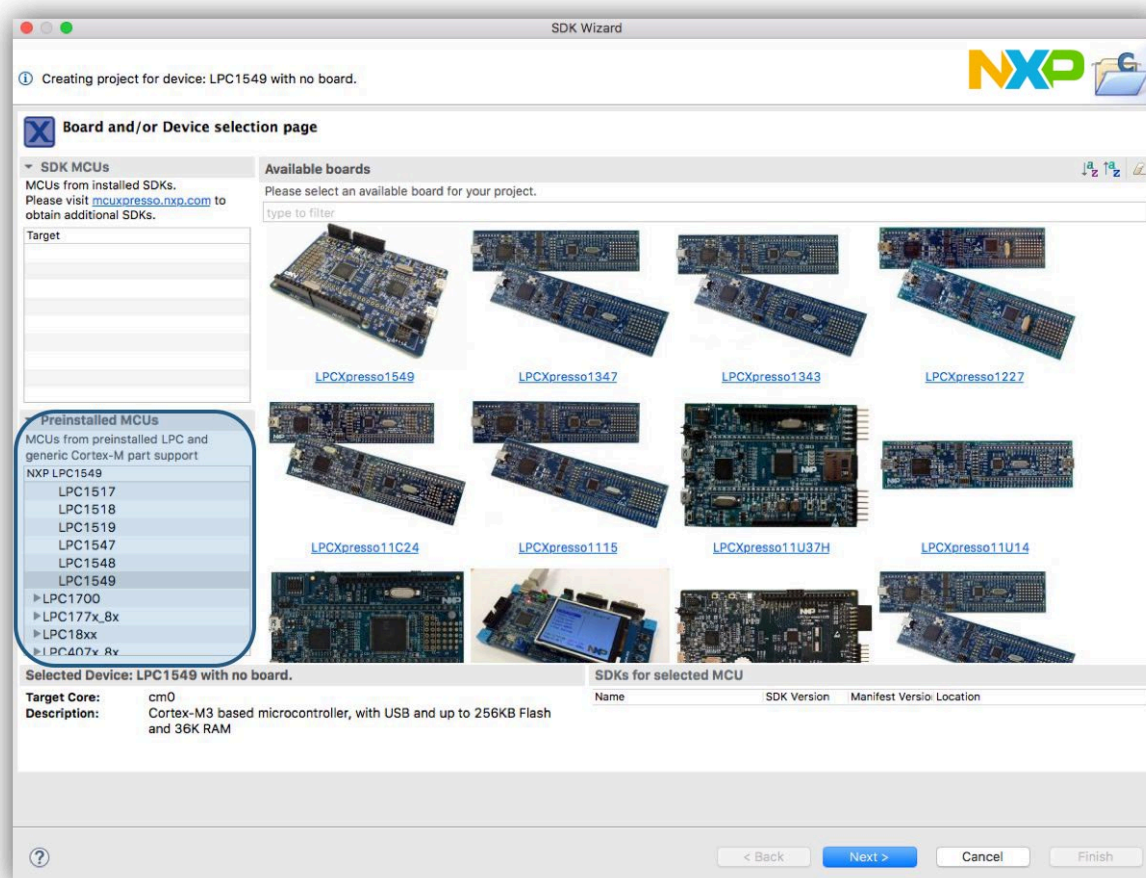
Figure 21. New Project Wizard

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

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

For creating projects with preinstalled part support, see: Creating Projects with Preinstalled part support

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

4.2 SDK part support

Extend the Part Support of the IDE by using freely available MCUXpresso SDK v2.x packages. SDK 2.x packages are used to add support for all Kinetis, i.MX RT, and newer LPC MCUs, and so on.

Starting with MCUXpresso IDE version 11.1.0 there is a streamlined approach to the supply and installation of SDKs - these SDKs are known as **Plugin SDKs**. Plugin SDKs are pre-built SDKs hosted on NXP's servers that you can browse, download, and install directly from within the IDE when required. See Obtaining and installing a Plugin SDK

Each SDK installs as an Eclipse plugin and so benefit from the standard Eclipse management and update mechanisms. MCUXpresso IDE Plugin SDKs are available for a wide range of NXP's MCUs. Like all Eclipse plugins, once Plugin SDKs are installed, they become part of the product itself. Management of a Plugin SDK can be performed using the standard Eclipse mechanisms

The previous *Classic* method of SDK installation and handling is still available. See SDK part support via SDK Builder

After installing an SDK, the included part support becomes available through the New Project Wizard and also the SDK example import Wizard, and for use by imported projects.

4.2.1 Obtaining and installing a Plugin SDK

SDKs are installed and managed via the *Installed SDKs* view, which is located by default as the first tab within the Consoles view. See Major components item 3 for more information. You can also start a Plugin SDK installation via the New Welcome system and via the *Download and Install SDKs* icon on the main IDE icon bar.

Once launched, a dialog similar to the one shown below appears:
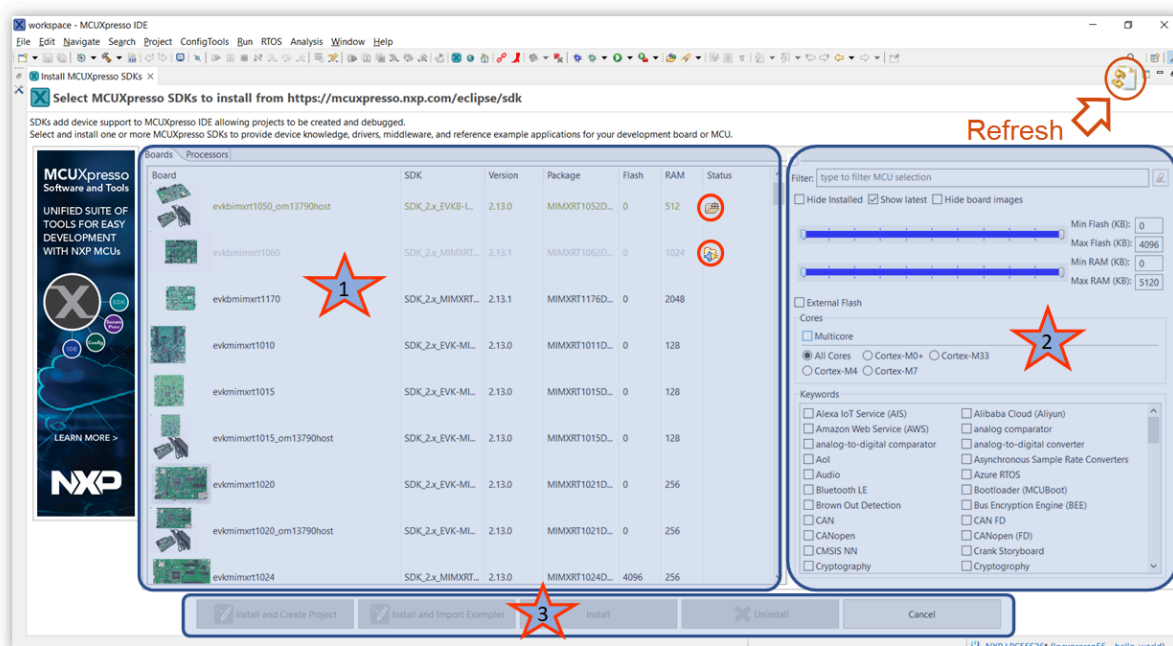
Figure 22. Plugin SDK installation

1. From this section, you can select the SDK for the desired Board (or Processor) for installation. Column sorting is supported to help location and options for filtering the list are discussed below.
   - By default, SDKs that are already installed are hidden from this view
   - If the *Hide Installed* is unchecked, installed SDKs are also shown along with a Status indication for the SDKs already installed (shown as a red circle)
2. The user may select a range of filtering options to reduce the list of displayed SDKs. These filters allow them to explore the capabilities of the MCUs and Boards.
3. After selecting an SDK, it can be installed (with options)
   - *Install and Create Project* Downloads, Installs, and launches the New Project Wizard with the chosen board selected
   - *Install and Import Example* Downloads, Installs, and launches the Import SDK Example Wizard with the chosen board selected
   - *Install* Downloads and Install s
   - *Uninstall* removes the Plugin SDK from the IDE

Note: On rare occasions, it may be necessary to force manually a refresh of the cached contents of the remote repository. You can perform this via the button highlighted above.

Once an SDK (or SDKs) is selected and an install operation begins, you are presented with an option to accept the SDK license condition as below:
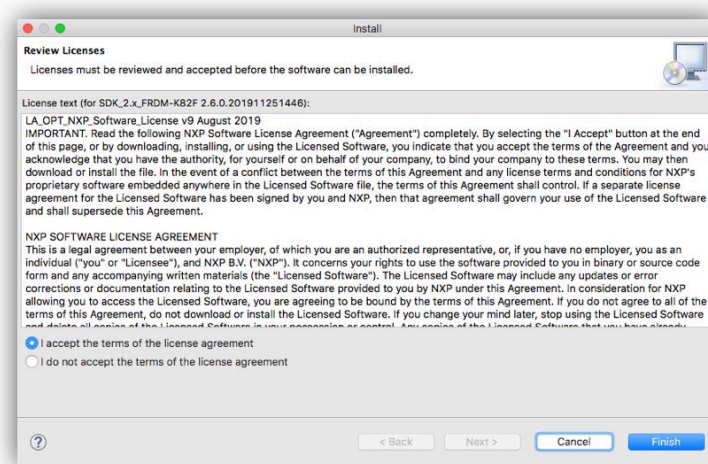
Figure 23. Plugin SDK installation license

Monitor the download and install progress via the Installation dialog:
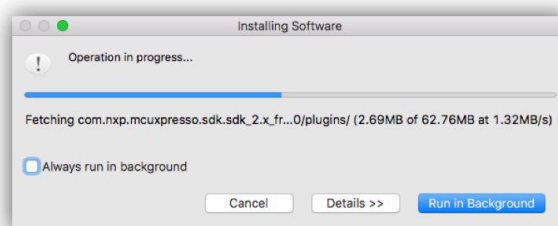
Figure 24. Plugin SDK installation progress

If you click *Run in Background*, control is returned to the IDE. The SDK does not become available until the download and installation is complete - at that time, it is possible to launch a Wizard when choosing a *Create* or *Import* option. While it is possible to restart the Plugin SDK Installer, any existing SDK installation must be complete before starting another Install.

Note: When starting the IDE for the first time, data for this display is automatically loaded in the background. If starting the Plugin SDK Installer promptly after the IDE starts, there may be a short pause while the data populates.

4.2.2 SDK part support via SDK Builder

NXP also provides SDKs for toolchains (including MCUXpresso IDE) via their SDK Builder site. Through this site (login required), NXP MCU users may request builds for NXP MCUs that they can configure to include a range of software features. Once built, the user can download and install the SDK into MCUXpresso IDE - this is the *Classic* method for installing SDKs as used in all previous versions of MCUXpresso IDE. SDKs installed in this way are now referred to as **FileSystem SDKs** since they become a shared resource for any IDE installation rather than part of a particular IDE installation.

You can install these SDKs via a simple 'drag and drop' mechanism or from the dedicated dropdown menu in the Installed SDKs view, which then automatically enhances the IDE with new part and board knowledge (and usually a large range of examples).

Generate and download SDKs for MCUXpresso IDE as required using the SDK Builder on the MCUXpresso Tools website at:

https://mcuxpresso.nxp.com/

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

4.2.3 Obtaining and installing an SDK via SDK Builder

Users of earlier versions of the IDE may be more familiar with this model of SDK build and installation.

SDKs are installed and managed via the *Installed SDKs* view, which is located by default as the first tab within the Console view. See Major components item 3 for more information.

SDKs are free to download (login is required); MCUXpresso IDE offers a link to the SDK portal (shown below) from the Installed SDK Console view, which opens in an external browser. From this portal, the required SDKs can be downloaded onto the host machine. Alternatively, you can open the portal by going to *Help -> Additional Resources -> MCUXpresso SDK Builder*.
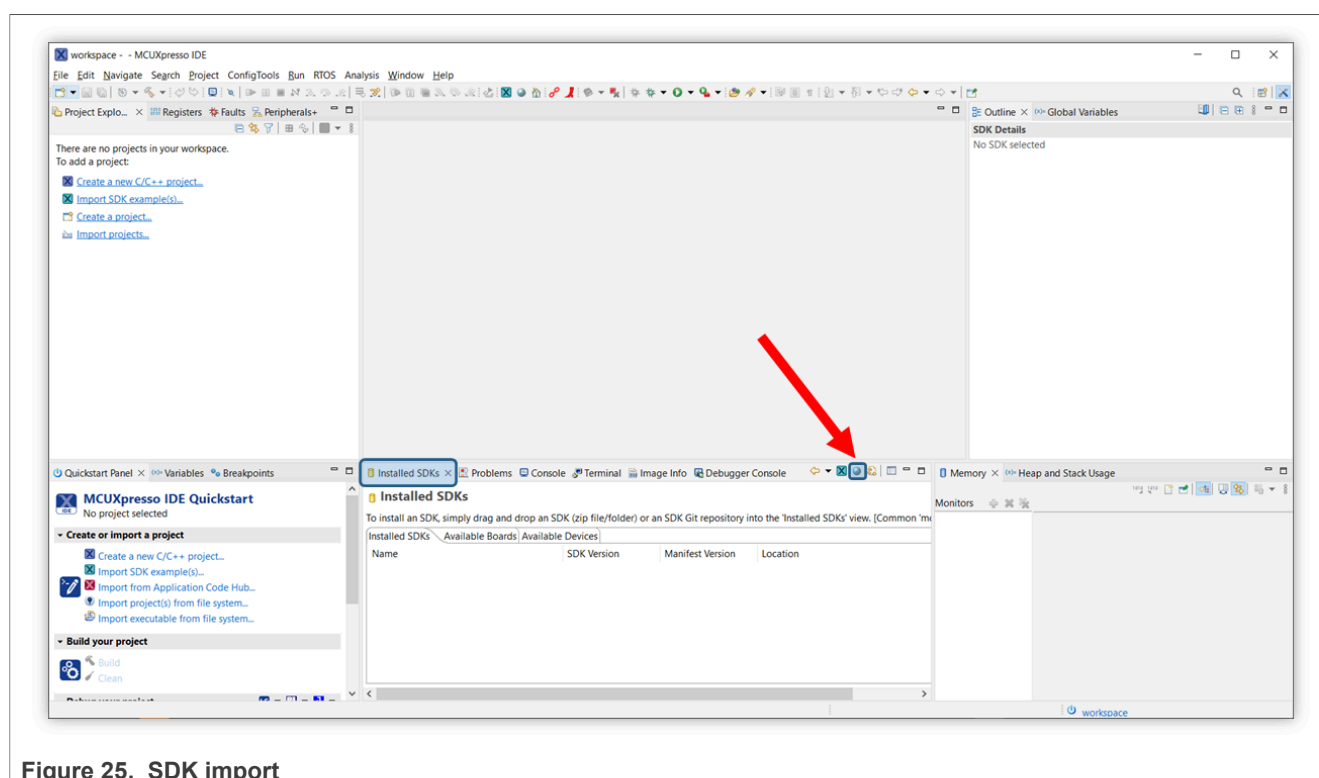
Figure 25. SDK import

Once downloaded, you can install an SDK package(s) by simply *dragging* from the downloaded location into the *Installed SDKs* view or by using the dedicated dropdown menu in the view. In case of using the *dragging* method, once *dropped*, a dialog prompts you to confirm the import – click OK. The SDK package(s) are then automatically installed into the MCUXpresso IDE part support repository.

Once complete the "Installed SDKs" view updates to show you the package(s) that you have installed.
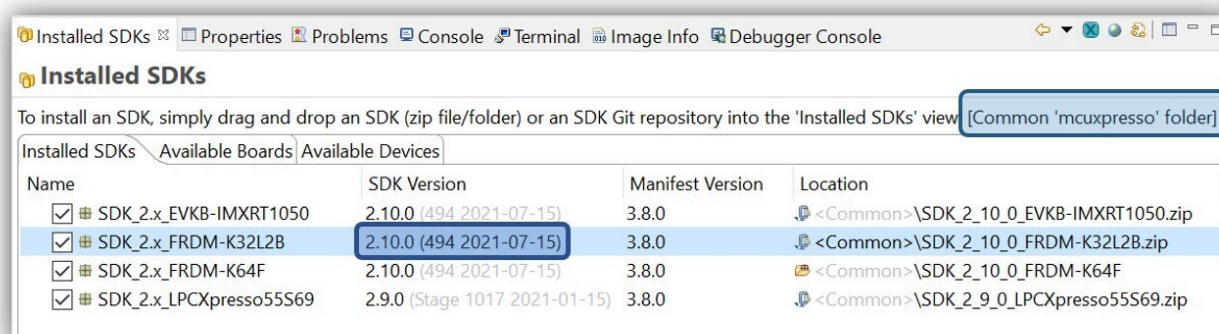
Figure 26. SDK import view

By default, SDKs are installed into a *Common* folder and are therefore available to any MCUXpresso IDE instance. Alternatively, it is also possible to install SDKs into the current workspace making their installation local to that workspace. The selected install location is shown in the SDK Window text as highlighted above. Also highlighted is the new version information string (displayed in gray), this feature allows different SDK builds to be distinguishable. Also see SDK advanced importing for further information on SDK installation options.

SDK Notes:

- Released in parallel with MCUXpresso IDE version 11.9.0 are updated SDKs (MCUXpresso SDK v2.15). These are indicated by their version 2.15.abc and a manifest version 3.14.0 in the Installed SDKs view. While older SDKs are still compatible with the newest MCUXpresso IDE version, it is recommended that users check and update to the latest available SDK package.
  - Installed SDK view tooltips display comprehensive version information.
- MCUXpresso IDE can import an SDK as a zipped package or unzipped folder (or zipped Plugin). Typically importing as a zipped package is expected.
  - The main consequence of leaving SDKs zipped is that you are not able to create (or import projects) into a workspace with linked references back to the SDK source files.
- Importing an SDK via drag and drop copies the required files and the original file/folder remains unaffected. The copied files are installed into a default location allowing imported SDKs to be shared among different IDE instances/installations and workspaces. Data from imported SDKs populate the wizards with available MCU and board information. In addition, they are parsed to generate part support and make example projects and drivers available, and so on.
  - By default, SDKs (like workspaces) are located in the user's local storage, this means they are only available to the user who performed the installation. Also see SDK advanced importing for details on how to use a shared location if needed.
- Once installed, the part support provided by the SDKs is regenerated. This regeneration is required because an MCUs part support may be specified (with different versions) within more than one SDK. On rare occasions, it may be necessary to force regeneration of SDK part support. You can do this by clicking the *Recreate and Reload* button within the top right block inside the *Installed SDK* view, or by right-clicking within the view and selecting *Recreate*.

4.2.4 Installing SDKs by importing a remote SDK Git repository

NXP also provides SDKs via its MCU SDK Git repository. You can install these SDKs automatically by using the wizard from *Installed SDKs* view.
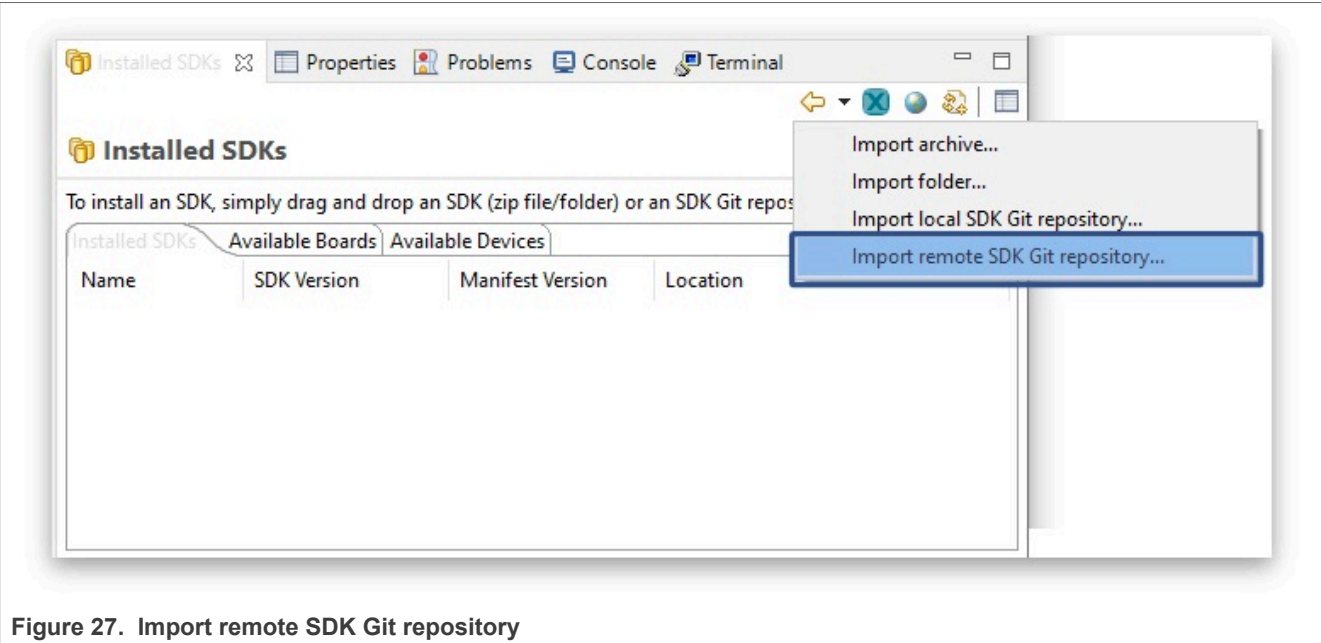
Figure 27. Import remote SDK Git repository

After selecting *Import Remote SDK Git Repository...* from the menu, the following window opens:
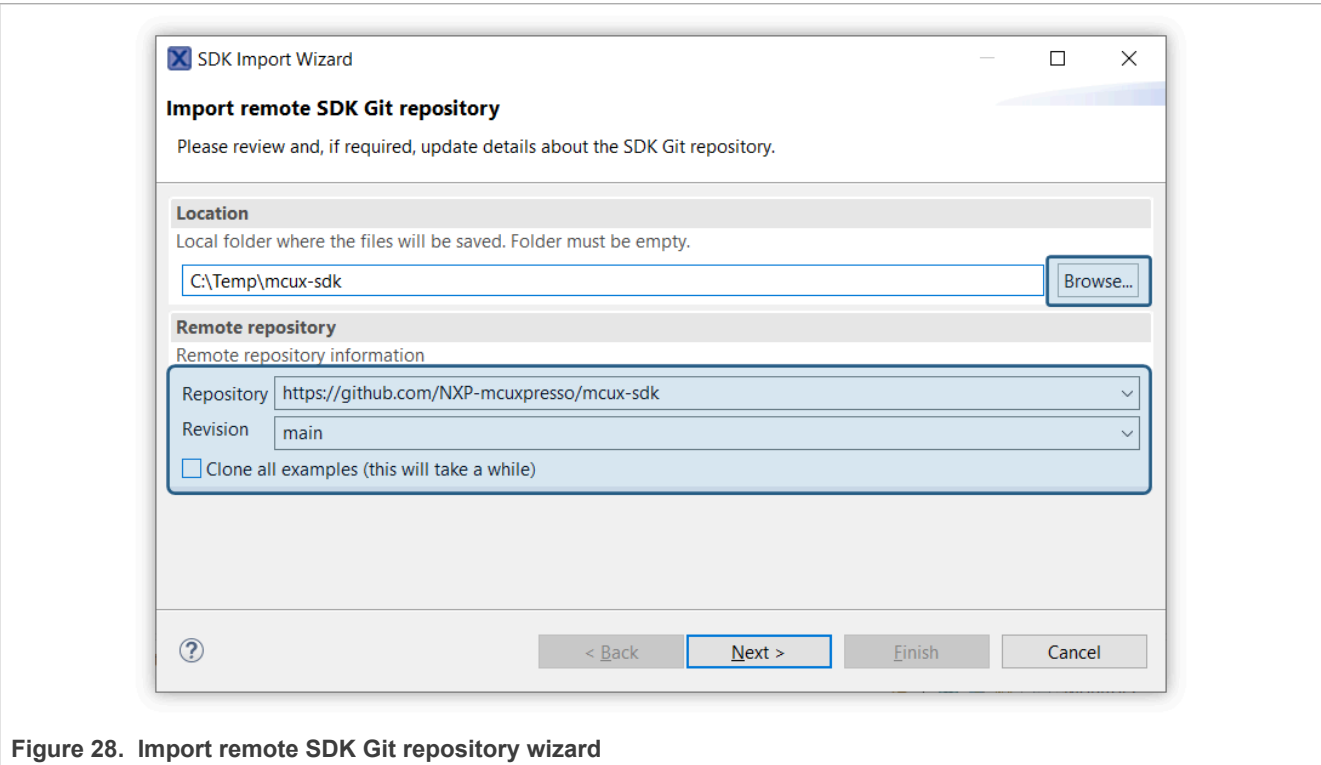
Figure 28. Import remote SDK Git repository wizard

You have to select an empty folder where the SDK Git repository is to be cloned. Revision can be "main" if you want the latest state, another branch from the Revision dropdown list, or any commit SHA.

Note: To speed up the remote import process, the default operation does not clone the example sources for every available board. Instead, they are downloaded on-demand whenever a specific example is selected for import. However, if you prefer a complete download from the start, you can select the *Clone all examples*

checkbox. This option allows you to have all examples readily available, but keep in mind that this leads to an increased download duration.

After clicking *Next*, the wizard continues to clone and configure the repository using the 'west' utility.
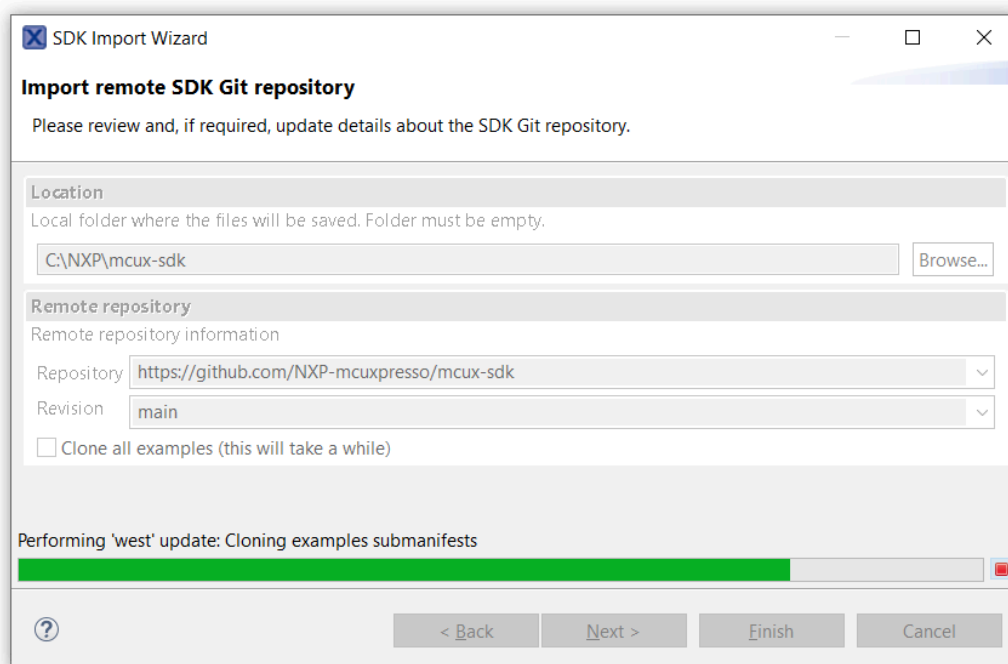
Figure 29. Import progress

Note that, depending on the speed of your Internet connection, the operation may take a few minutes to complete. After cloning has finished, the wizard advances to the next page, providing you with the option to import the repository. If you do not wish to change the manifest location, pressing _Finish_ imports the repository with the default settings. The "Import SDK example(s)" wizard can also open automatically, once the wizard is closed, if the associated checkbox is ticked.
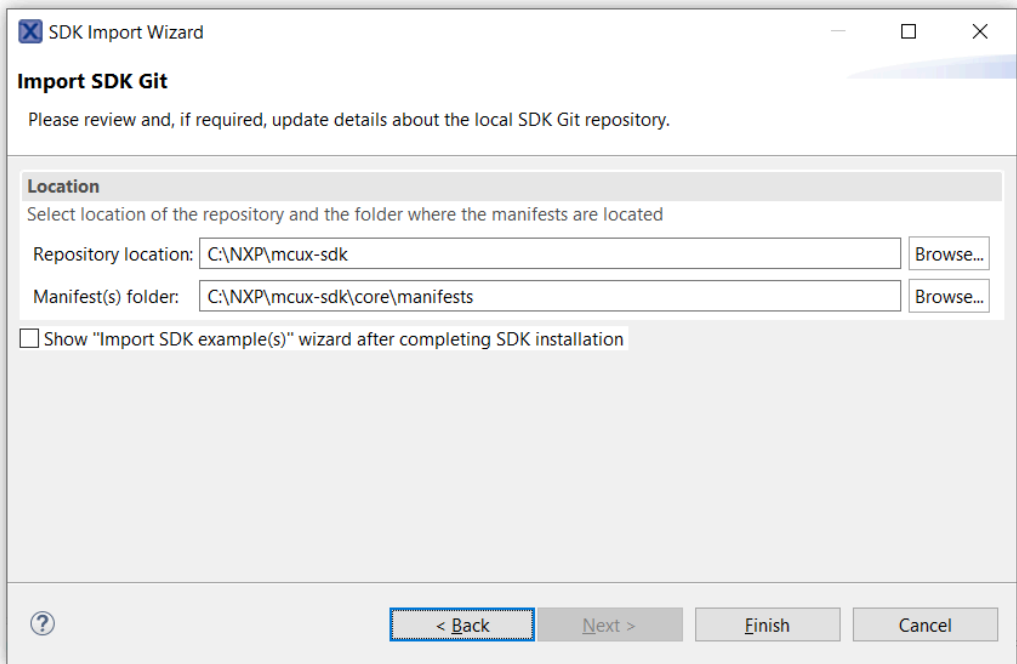
Figure 30. Importing cloned SDK Git repository

Afterward, the SDKs are imported and shown in the *Installed SDKs* view.
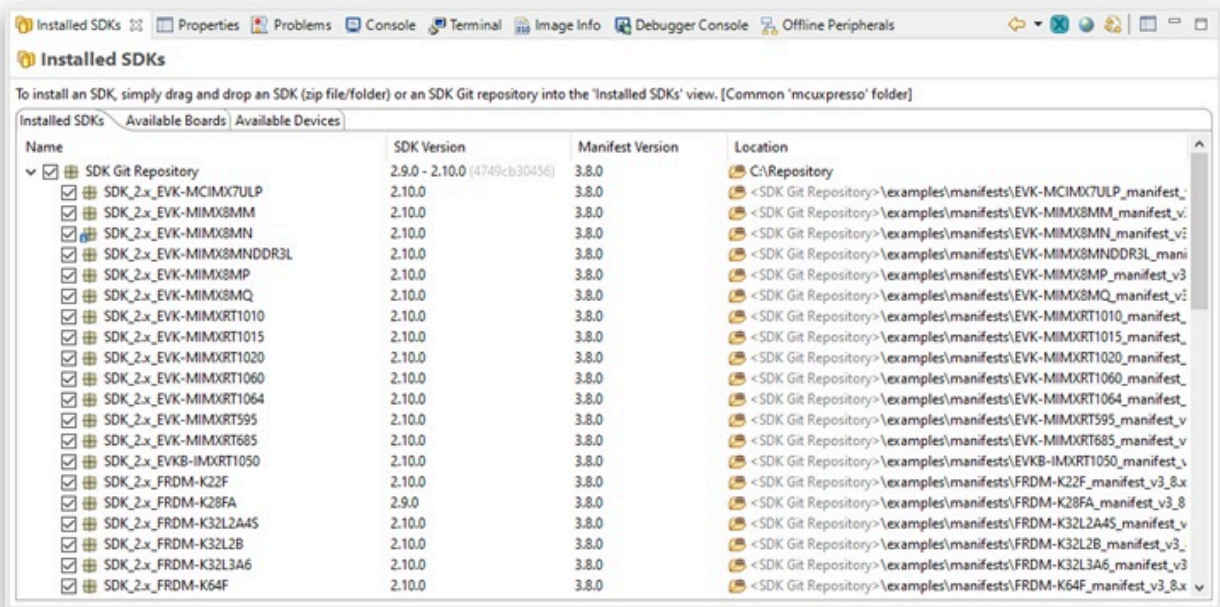
Figure 31. Installed SDK Git repository

Important Note: You must have both Git and West installed to use this wizard. If West is not installed or not found in the PATH environment variable, the IDE displays the following warning:
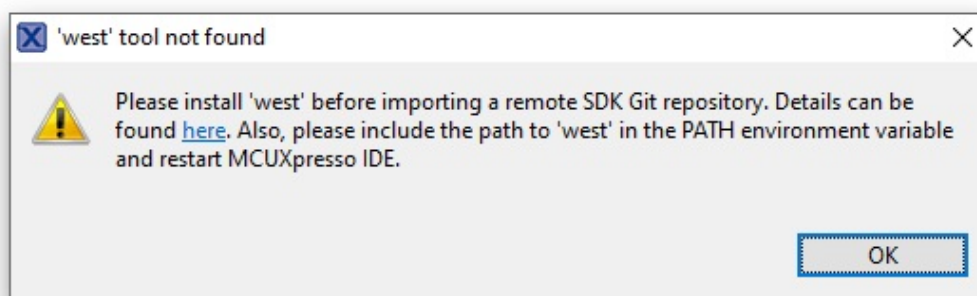
Figure 32. Import remote warning

On MacOS one way to add West to the PATH environment variable is to use the "launchctl" command. Usage:

```
sudo launchctl config user path $PATH:{West absolute path}
```

4.2.5 Installing SDKs by importing a local clone of an SDK Git repository

If you use command line to obtain a local copy of the remote SDK Git repository, you can import it using *Import Local SDK Git Repository...* menu form *Installed SDKs* view.
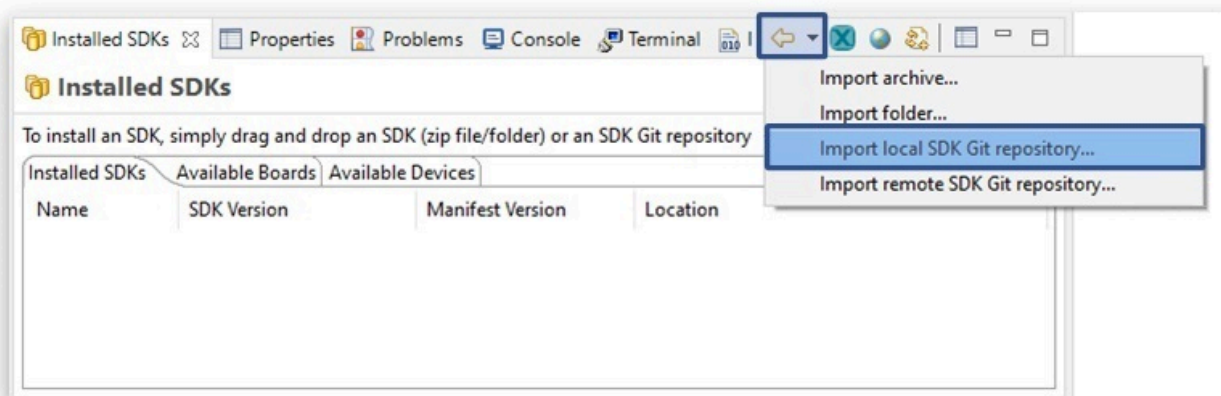
Figure 33. Import local SDK Git repository

4.2.6 Installed SDKs operations

The installed SDKs view now incorporates 3 tabs. In addition to the *Installed SDKs* tab, new *Available Boards* and *Available Devices* tabs are provided. These tabs expose the supported boards and devices provided by the installed SDKs and allow the direct invocation of New Project and Example Import Wizards:
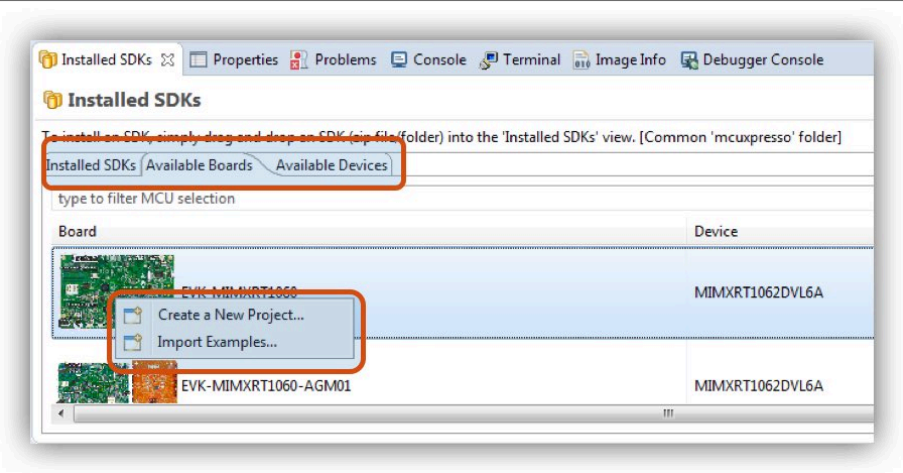
Figure 34. SDK tabbed views

Various other operations are available from the Installed SDKs view some from a right-click menu options:
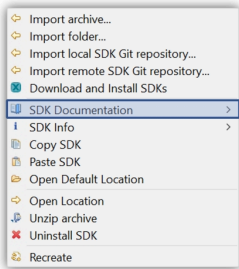
Figure 35. Installed SDKs options

Important Note: It is not possible to unzip Plugin SDKs from this view. However, you can convert them to FileSystem SDKs Do not attempt to manually modify a Plugin SDK in any way, doing so could lead to a loss of SDK part support from the IDE. You can delete Plugin SDKs either by using the *Uninstall SDK* button from the Installed SDKs view, or using the *Uninstall* button from the Install MCUXpresso SDKs view.

From here you can perform many actions such as view associated embedded SDK documentation that would otherwise require the unzipping and exploration of the SDK structure.

The Installed SDKs view shows whether the SDKs are stored as zipped archives or regular folders. MCUXpresso IDE offers the option to unzip a filesystem SDK archives in place via a right-click option onto the selected SDK (as below).
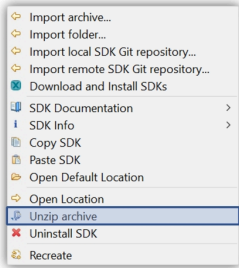
Figure 36. Unzip archive option

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

After unzipping an SDK, its icon updates to reflect that it is now stored internally as a folder.
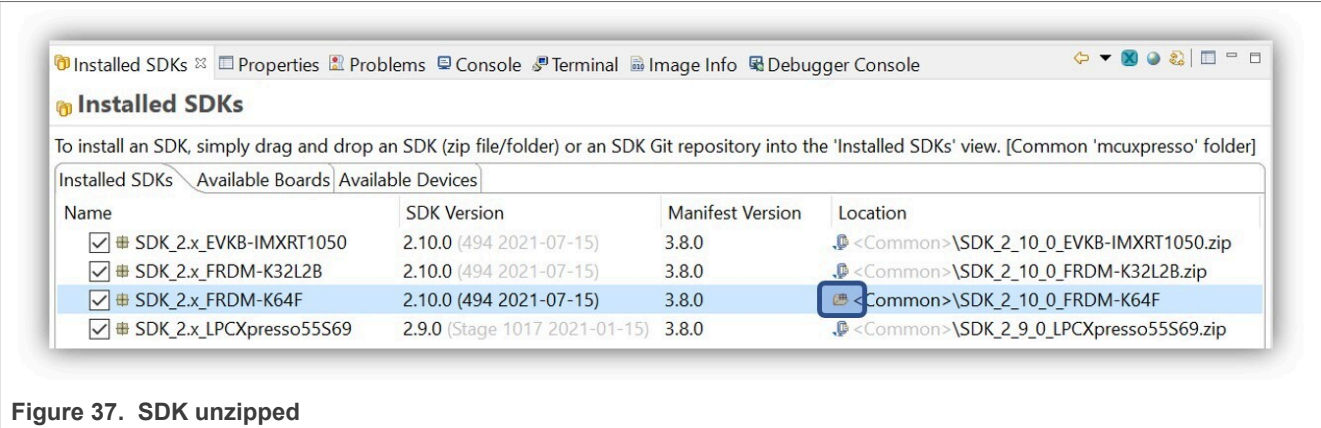
Figure 37. SDK unzipped

Many other options are available such as examining SDK XML description files, Copying and Pasting SDKs, and managing the library of installed SDKs.

Tip: *To edit (and save) SDK XML files, you must first unzip the SDK and change the following preference: Preferences -> MCUXpresso IDE -> SDK Handling -> Misc, uncheck the read-only mode option. Once saved, changes become permanent for that SDK installation.*

Tip: *In addition to the other SDK options, you can paste an SDK into the Installed SDK view from the file system or another IDE instance.*

Finally, SDK part support automatically regenerates when a new SDK is installed. If a project is imported and the expected part support is not available, then select *Recreate* from the right-click menu option to force a recreation of the SDK part support.

4.2.6.1 Converting a Plugin SDK into a file system SDK

On occasion, it may be useful to migrate a Plugin SDK to become a file system SDK - for example, if you require the SDK to be unzipped or to be shared with other IDE installations. To do this simply select the Plugin SDK within the Installed SDK view then from the right-click menu select *Copy* followed by *Paste*. This launches an Import operation and copies the SDK contents from the Plugin into the default SDK file system location. This SDK is the preferred choice over the Plugin version.

Note: A Plugin SDK is part of an IDE installation and can only be deleted using the dedicated "Uninstall" buttons from Installed SDKs and Install MCUXpresso SDKs views.

4.2.6.2 Uninstalling (deleting) an installed SDK

Plugin SDKs become part of the IDE and so you cannot simply delete them from the file system. Always use the *Uninstall* button from the Install MCUXpresso SDKs view or the *Uninstall SDK* button from the Installed SDKs view.
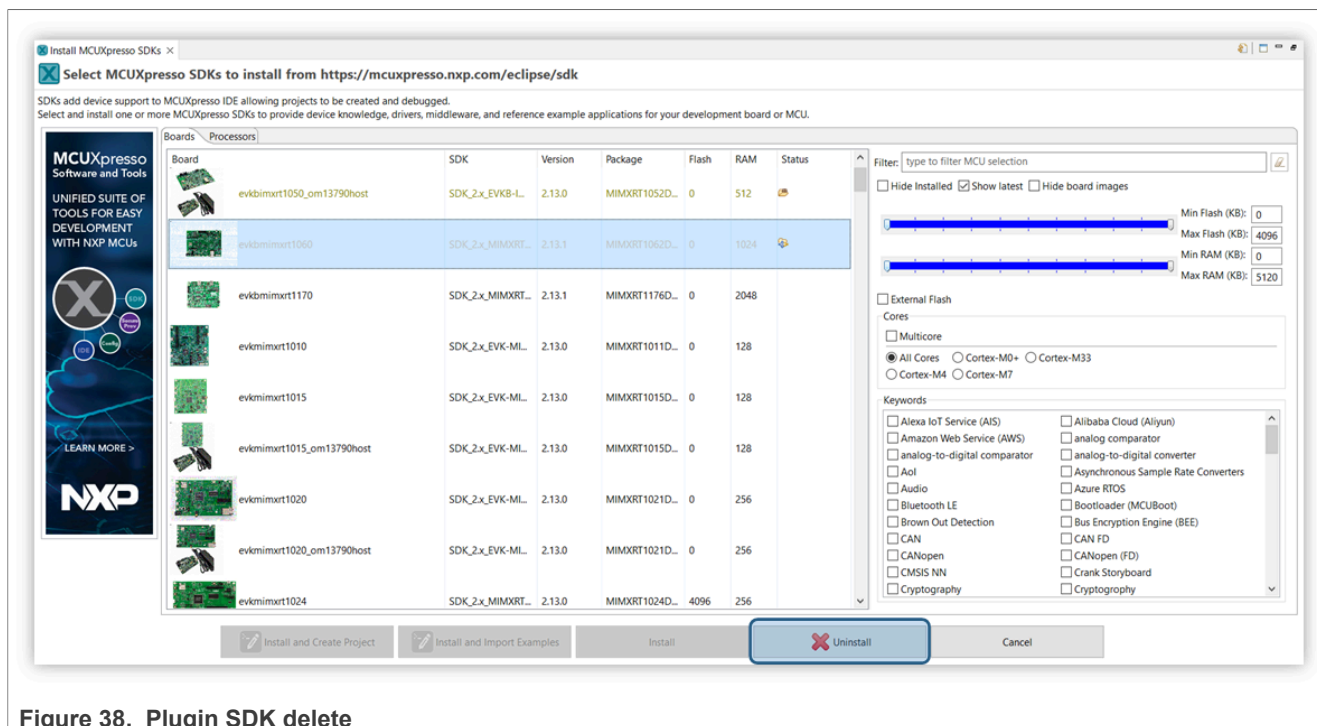
Figure 38. Plugin SDK delete

Note: A file system SDK is always the preferred choice over a Plugin SDK, allowing the effective replacement of a Plugin SDK by the installation of a file system SDK offering equivalent features.

If an SDK has been installed by the 'Drag and Drop' method, then a copy of the SDK will have been installed into the *Default Location*. You can uninstall and delete SDKs installed in this location via a right-click option. After uninstalling an SDK, part support is automatically recreated for the remaining SDKs. See Uninstallation considerations for more information.

Alongside each installed SDK, there is a checkbox. If unchecked, the SDK is hidden from MCUXpresso IDE until rechecked. If multiple SDKs are installed that contain shared part support, then this feature may be useful to force the selection of part support from a particular SDK. See Shared part support handling for more information.

Manually delete or hide SDKs installed into nondefault file system locations if they are no longer required. **Note**: you may have to quit MCUXpresso IDE to delete these SDKs. See SDK importing and configuration for more information.

SDKs installed from a Git repository can only be uninstalled by deleting the entire repository from the Installed SDKs view.

4.2.7 Installed SDKs features

You can explore each of the SDKs within the Installed SDKs View to examine content such as Components, Memory Settings, included Examples, and so on.
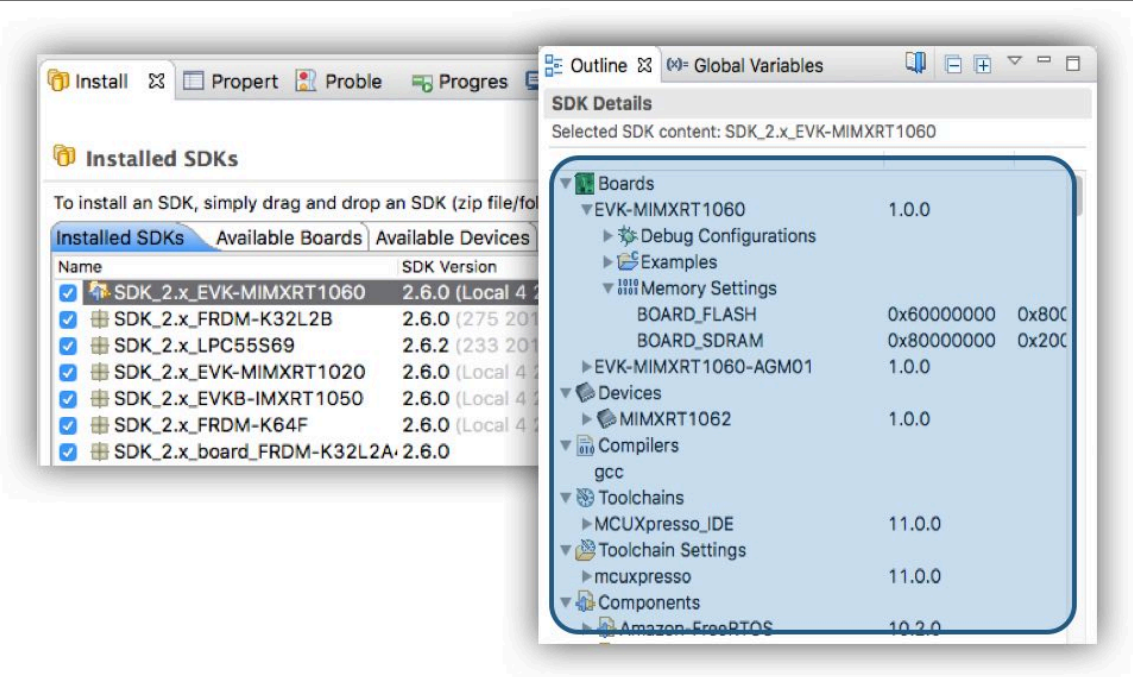
Figure 39. SDK explore

4.2.8 Advanced use

SDK importing via drag and drop incorporates two features. First, the location where the SDK is copied, and second, the automatic scanning of this location to create the required *Part Support*. You can explore and change the behavior via a preference *Preferences -> MCUXpresso IDE -> SDK Handling -> Installation* leading to the window below:
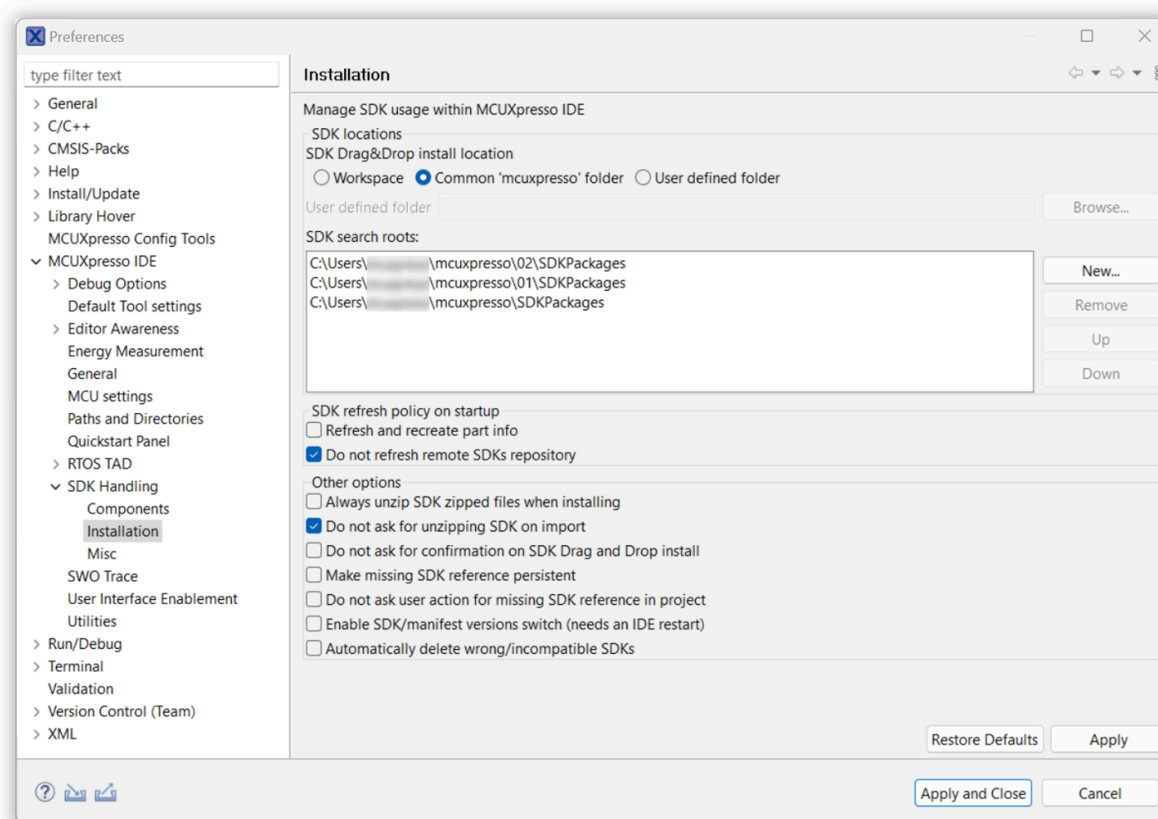
Figure 40. SDK installation preferences

You can see in the above graphic that two search locations are present. The *02* path is the default search path for MCUXpresso IDE version 11.0, earlier versions of MCUXpresso IDE used the *01* path. This older path only appears if the location actually contains installed SDKs (typically installed via an earlier version of MCUXpresso IDE). The reason for these separate paths is to allow users to have both the latest and older versions of MCUXpresso IDE installed without presenting incompatible versions of SDK to older versions of the tools. See SDK compatibility with earlier versions of MCUXpresso IDE for more information.

- Workspace
- Common (the default)
- User Defined

You can change the default *Common* install location to either the currently selected workspace or a User-Defined location. Once doing this, a new SDK Search Root path is automatically added to the search roots list.

Note: while you have the choice to remove other search roots if so desired, it is not possible to remove the currently selected drag-and-drop location.

In addition, from this dialog, you can add new search paths to folders where you have stored or plan to store SDK folders/zips. Those SDKs appear in the Installed SDKs View along with those from the default location when the Installed SDK view refreshes.

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

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

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

Note: Only the default SDK location(s) is persistent between workspaces. You have to create any other locations for each workspace as required.

Also displayed in the dialog (above) several 'checkbox' options that are discussed below:

- Refresh and recreate ... if checked, during IDE start-up, the IDE re-examines each SDK and recreates inside the current workspace the SDK part support.
- Do not refresh remote SDKs ... if checked, during start-up the IDE does not connect to the remote SDKs repository to identify the SDKs available for installation. This operation helps pre-populating the Install MCUXpresso SDKs view with installable SDKs.
- Always Unzip SDK ... if checked, unzip a zipped SDK on import.
- Do not ask for unzipping ... if checked (default), the IDE does not prompt the user to consider unzipping the SDK.
- Do not ask for confirmation ... if checked, the IDE imports an SDK via drag and drop without requesting user confirmation.
- Make missing SDK reference persistent ... this setting controls the persistence setting when the option below is checked.
- Do not ask for User action ... see [shared part support](#) - if checked, make this SDK association setting without prompting the user.
- Enable SDK/manifest version... if multiple SDKs for the same part are installed, this option, if checked, also allows the selection of an older SDK from within the Installed SDK view via a dropdown menu on the SDK Version.
  - Also, some SDKs include older versions of the manifest (XML description) ... if checked, this option allows an older manifest version to be selected from within the Installed SDK view via a dropdown menu on the Manifest Version.
- Automatically uninstall ... if checked, delete an SDK found in drag and drop install location that is incompatible with MCUXpresso IDE.

4.2.9 Advanced use

Additional miscellaneous SDK preferences are also available. These checkbox options are shown below:
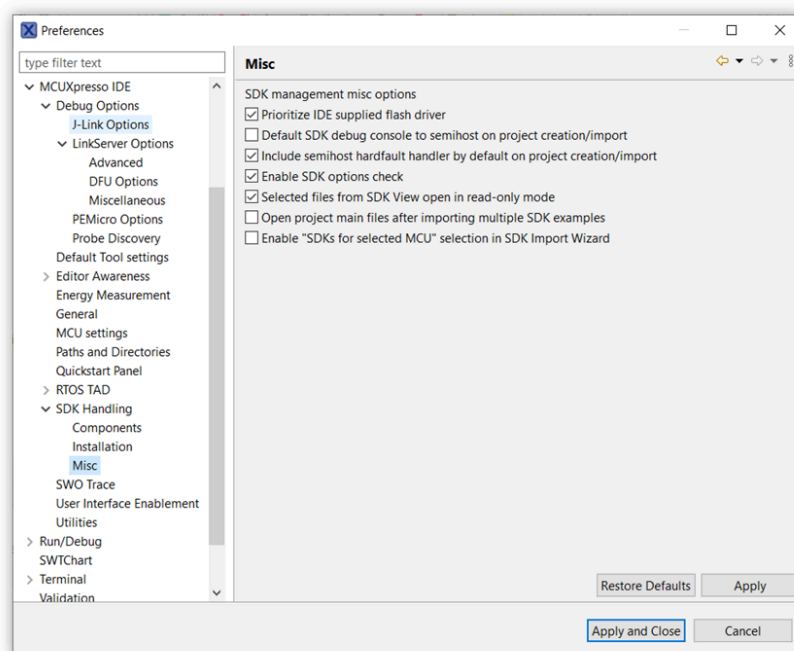
Figure 41. SDK preferences misc

Where:

- Prioritize IDE-supplied flash drivers ... typically, LinkServer flash drivers are supplied as part of the SDK part support for a particular MCU. However, these LinkServer flash drivers are usually duplicated within the IDE installation where newer versions might be found. This option, checked by default, causes the IDE-supplied drivers to be used in preference to SDK-supplied flash drivers. Searching the flash driver directory of the IDE in preference to SDK dynamically part support files also simplifies flash driver development

- Default SDK debug console to semihost ... this option, checked by default, sets project defines to select semihosting as the output format

- Include semihost hard fault handler ... this option, checked by default, causes a minimal hard fault handler to be included within new and imported projects. The purpose of this handler is to send semihost operations to null when no debug tools are connected. Without such a handler, any semihosted operation halts the MCU when no debug tools are connected. This is probably the most useful option for early project development, however, this may clash with any *real* hard fault handler.

- Enable SDK options check ... this option, checked by default, allows the IDE to check the options of an SDK example on import and attempt to resolve any incompatible options found.

- Selected files from SDK view ... this option, checked by default, forces any file opened from the Installed SDKs view to be opened in Read-Only mode. This is to protect SDK files from accidental corruption. Note: this option only applies to SDKs that are imported unzipped.

- Open Project main files ... an imported example project is opened within the project explorer view and the source file containing the main function is opened. This option, unchecked by default, allows this to occur if importing multiple files at the same time.

4.2.10 Important notes for SDK users

Installing an SDK into MCUXpresso IDE adds to its default capabilities, but SDKs come in many different configurations and versions. The section below discusses some of the issues that users may experience when working with SDKs.

4.2.10.1 Only SDKs created for MCUXpresso IDE can be used

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

4.2.10.2 SDK compatibility with earlier versions of MCUXpresso IDE

A new SDK version 2.15 has been released in parallel with MCUXpresso IDE version 11.9.0. However, this SDK format includes features that are not compatible with earlier versions of MCUXpresso IDE. As a result, these new SDKs may fail to install or offer reduced features when used in older versions of MCUXpresso IDE.

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

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

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

4.2.10.3 Shared part support handling

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

If two SDKs with matching part support are installed, and the SDK providing part support is later deleted, then the IDE automatically uses part support from the remaining SDK.

Finally, if a project is created with one SDK part support - for example Freedom K64, and then:

• That SDK is changed to another SDK with compatible part support - for example TWR K64
• The project is shared with another user who has a different SDK that includes compatible part support (perhaps an SDK that has only device support).

A dialog similar to the one below appears for each project where this occurs:
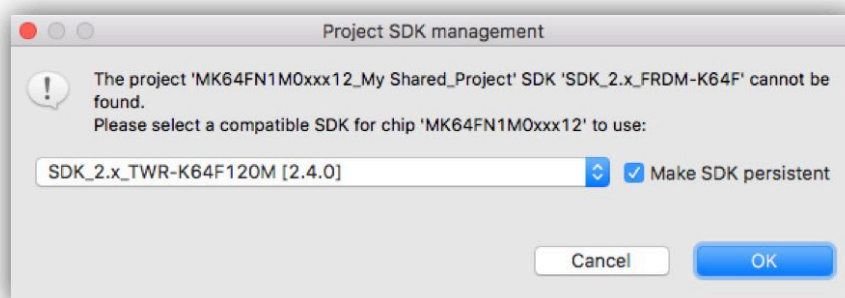
Figure 42. Project SDK management dialog

Where the option to *Make persistent* permanently changes the project to be associated with the selected SDK. If unticked, the IDE accepts the change as temporary and writes no data back to the project.

Note: When making this new association, the project contains files from one SDK but is associated with another. Refreshing project or using the component management feature, may copy incompatible code into the project.

4.2.10.4 Building a Fat SDK

You can generate an SDK for a selected part (processor type/MCU) or for a board. If you only select a part, then the generated SDK contains both part support and 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 to add the board support automatically.

If you choose a part that has no directly matching board, say the Kinetis MK63… then the generated SDK contains:

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

4.2.10.5 Uninstallation considerations

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

4.2.10.6 Sharing projects

Note: Also see Enhanced project sharing features below:

If you build a project using part support from an SDK and then export it - 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 MCU of the project.

4.2.10.7 Viewing SDK documentation

Beginning with version 2.15.000, SDK packages ship with documentation written in markdown format. To facilitate reading this format, MCUXpresso IDE comes pre-installed with the **Mylyn WikiText Markdown UI** plugin. This allow you to view formatted text and markdown source directly in the Eclipse editor.
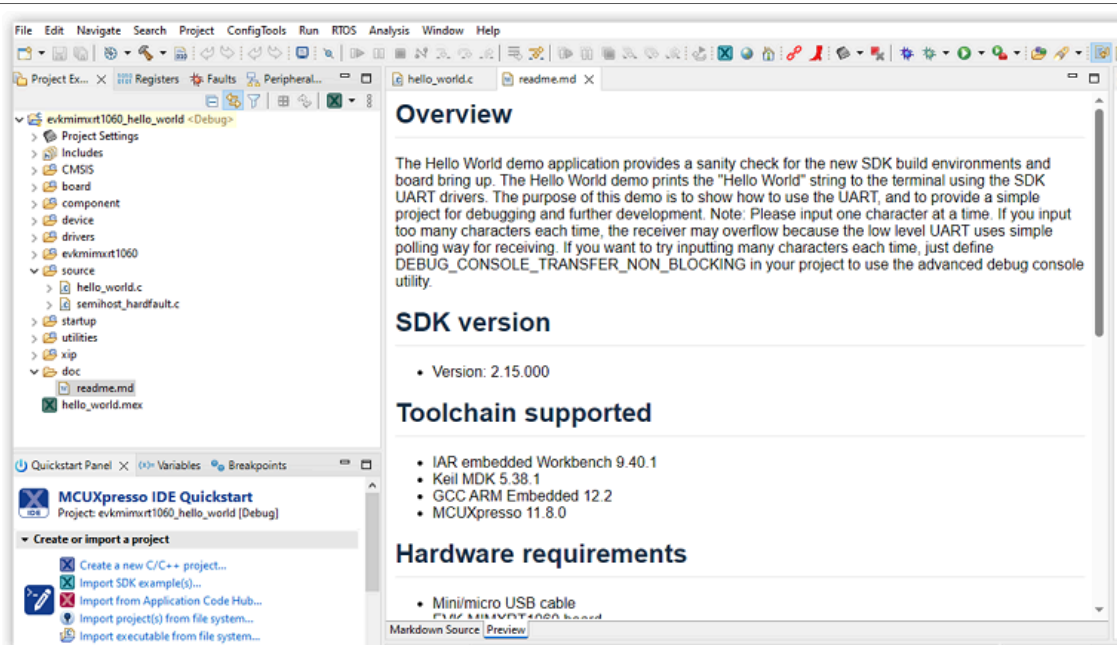
Figure 43. Markdown editor

4.3 Enhanced project sharing features

MCUXpresso IDE has a range of features designed to improve the ease of project sharing. These features combine to streamline the sharing and collaboration process.

4.3.1 Project drag and drop

In addition to the existing project import and export capabilities available from the **Quickstart** panel, a new set of features has been introduced to ease the transfer of projects.

Previously, the import of a project required browsing to a project location followed by an import. Now ...

- You can import projects into a workspace by simply dragging and dropping a folder (or zip) containing one or more projects into the Project Explorer view
- You can copy projects from one IDE instance to another by simply dragging and dropping from one Project Explorer view to another

Eclipse also offers the following functionality:

- You can also export projects by dragging from the Project Explorer view onto a host filer
  - **Warning**: Take care here since the default Eclipse behavior when dragging is to **move** files from the workspace rather than performing a **copy**. You can modify this behavior to **copy** on Mac via holding the Option Key, and on Windows via holding Ctrl. Note that **if** the underlying files of a project are moved, the project remains visible within the project explorer view but is longer usable. Perform a project explorer refresh (F5) in this case.

Tip: If you move a project accidentally (as described above), you can reimport it by dragging it back from the filer location into the project explorer view (the original project must be removed first otherwise a clash of names prevents import).

4.3.2 Project-local SDK part support

One weakness of the SDK model of extending the capabilities of the IDE comes when sharing projects with colleagues - since they must also have the same SDK installed to use this shared project.

To avoid this problem, SDK projects (and examples) can be modified to contain a local copy of the required SDK part support.

SDK project may be enhanced to contain local SDK part support

• SDK-based projects can now import a cache of part knowledge from an installed SDK
  – Simply right-click a project and select *add SDK Part Support*
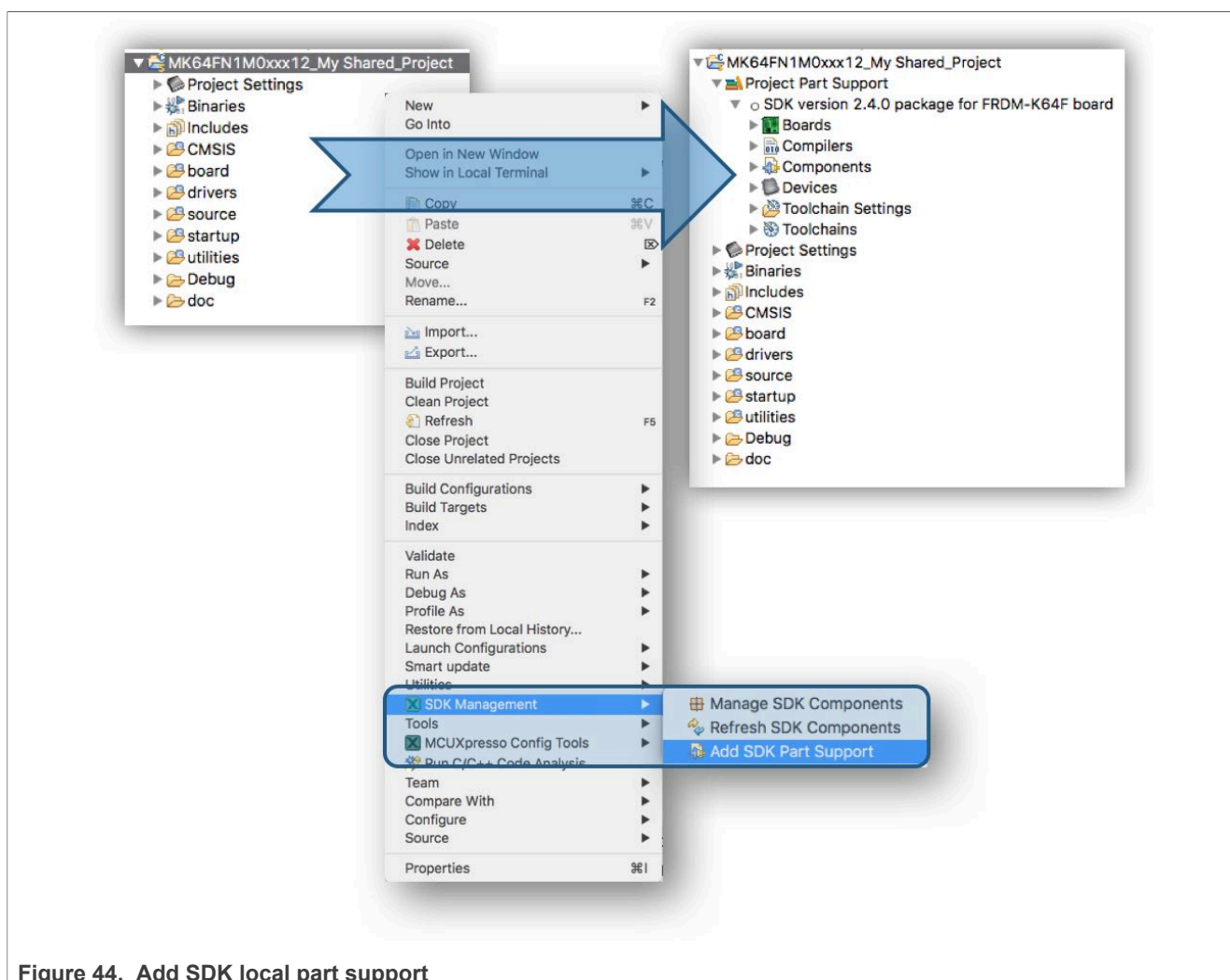
Figure 44. Add SDK local part support

• Another user can then use such projects (if using MCUXpresso IDEs version 10.2.0 or later) without first downloading and installing the appropriate SDK
  – In such cases, the local part support of the project is visible as an installed SDK
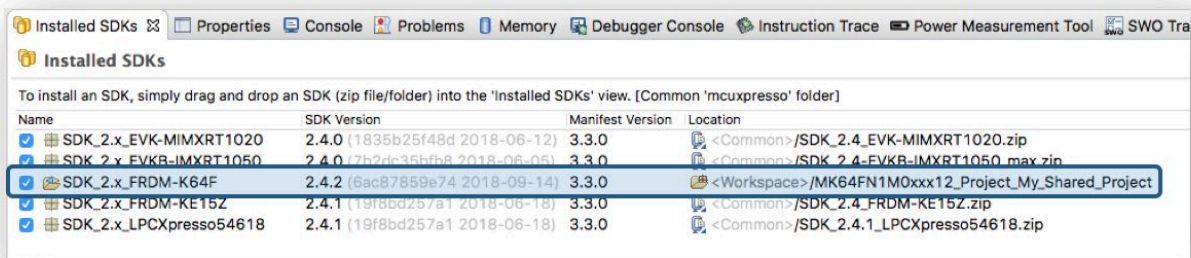
Figure 45. View SDK local part support

Note: this feature is not designed to replace the need for ultimately installing an SDK, since there are implications in project size, and so on. rather it is intended as a short-term solution to decouple projects from the requirement for an SDK.

Finally, you can remove local part support in the same way as you have added it. Simply right-click a project and select *SDK Management -> Remove SDK Part Support*. After doing this, you must install an appropriate SDK in order to use the project.

4.3.3 Project-local support files

Supporting files required for debugging such as flash drivers, LinkServer Connect and Reset scripts are usually found (automatically) either within an SDK or installed by default within the LinkServer installation folder.

However, on occasion, bespoke flashdrivers and/or scripts may be required. While you could store and reference these files from various locations within the file system, to enhance project sharing, you can now include such files directly within a project and locally reference them.

To use script and flash driver files in this way, first, simply drag them into the local Project structure:
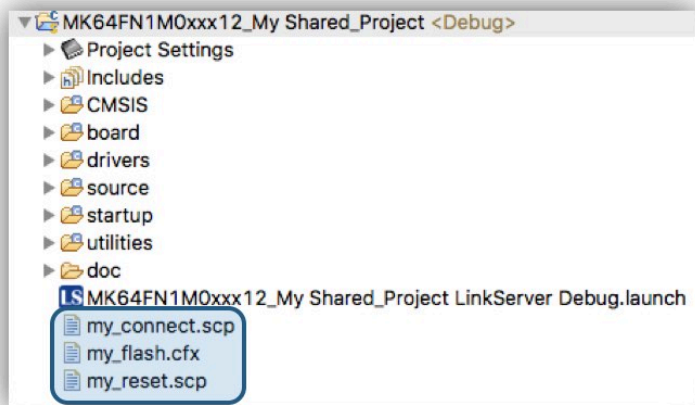
Figure 46. Project structure

You can now use LinkServer launch configurations to directly browse to local scripts (connect or reset) as shown below:
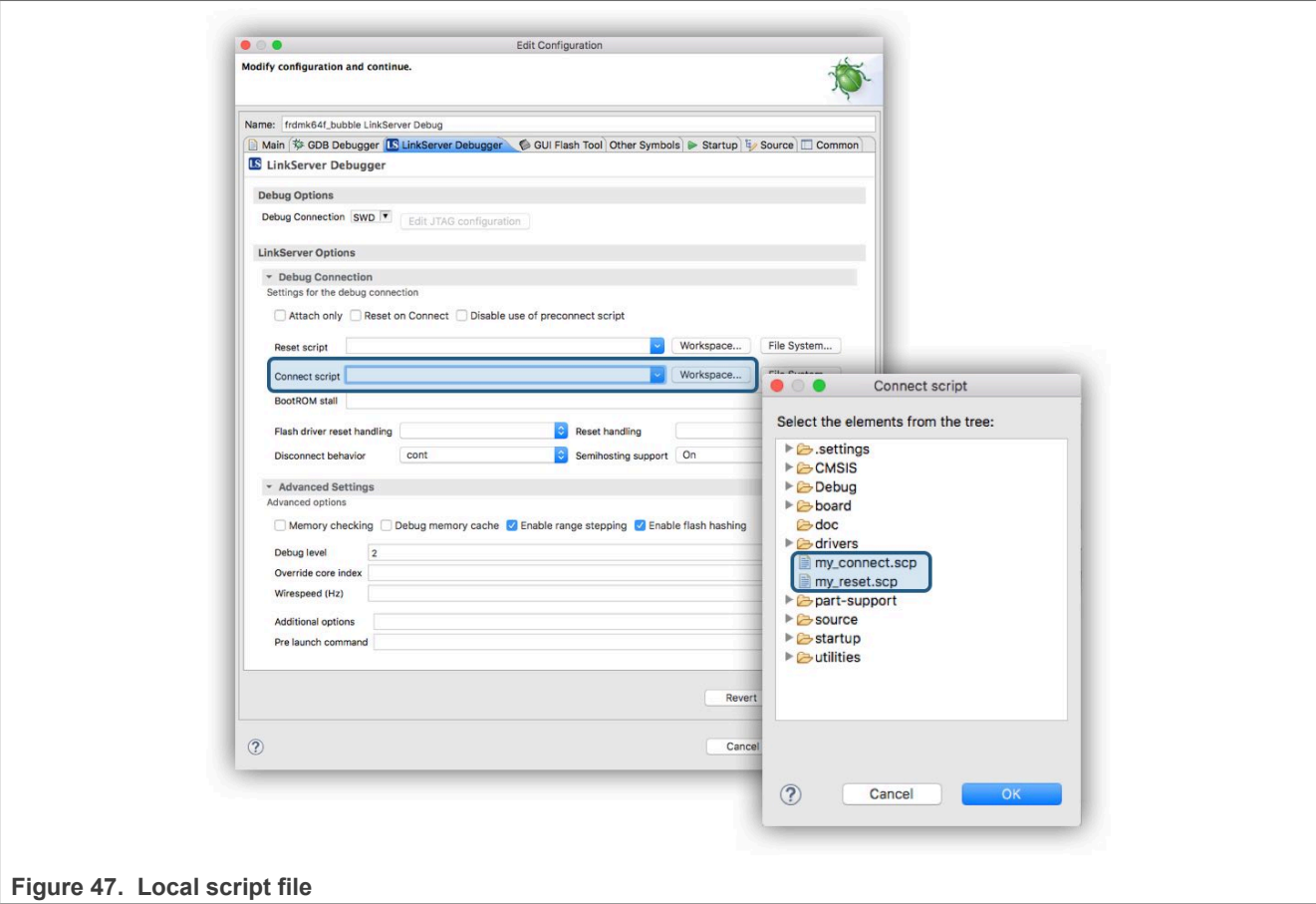
Figure 47. Local script file

Similarly, you can reference a project-local flash driver by editing the memory configuration of the project and again browsing for the required flash driver within the project as below:
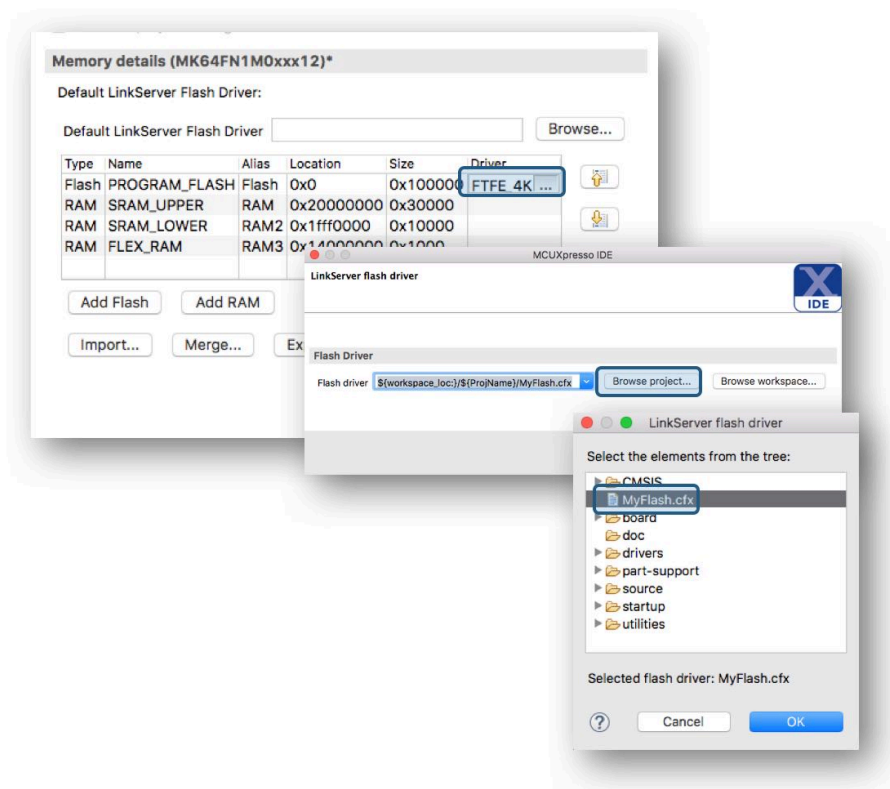
Figure 48. Local flash driver

See also Memory configuration and linker scripts.

The features described above are rarely required, but on the occasions where shared projects have bespoke debug files, the above scheme should simplify the sharing and use of MCUXpresso IDE projects.

4.3.4 Export project to local SDK Git repository

It is now possible to export a project into a local SDK Git Repository. The board and device for which the project was created have to be supported by the local SDK otherwise it can't be exported.

To export a certain project, right-click it inside the *Project Explorer* and select *SDK Management -> Export to SDK Git repository...*.

Figure 49. Export to local SDK Git repository

The name of the new example is the name of the project, but it can be modified. For SDKs with split manifests, the user can also select the manifest where the project will be referred. A combo box presents all the categories available in the SDK for the current board. For the new example, you can select one of the available categories or select a new one. You can also choose the project location, which has to be in the same folder or in a subfolder of the manifest where you have chosen to export the project. After clicking *OK*, the SDK Git repository will contain the desired example.

The new example will be available in the *SDK Import Wizard*.
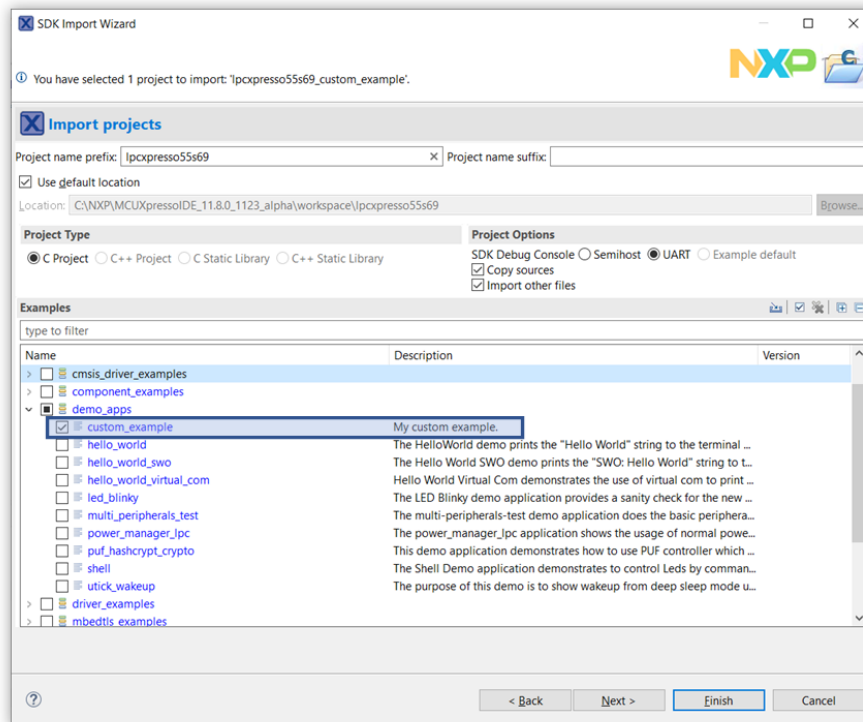
Figure 50. New example exported to local SDK Git repository

Limitations - Projects using linked references to the SDK source code are not supported. - Once exported to the local SDK Git repository, you can import the new example only with the "Copy Sources" option set (to copy source code files inside the project).

5 Creating new projects using installed SDK part support

For creating a project using *Preinstalled part support* see: Creating projects using preinstalled part support

Locate the Quickstart panel at the bottom left of the MCUXpresso IDE perspective and see the first entry, *Create a new C/C++ project*.
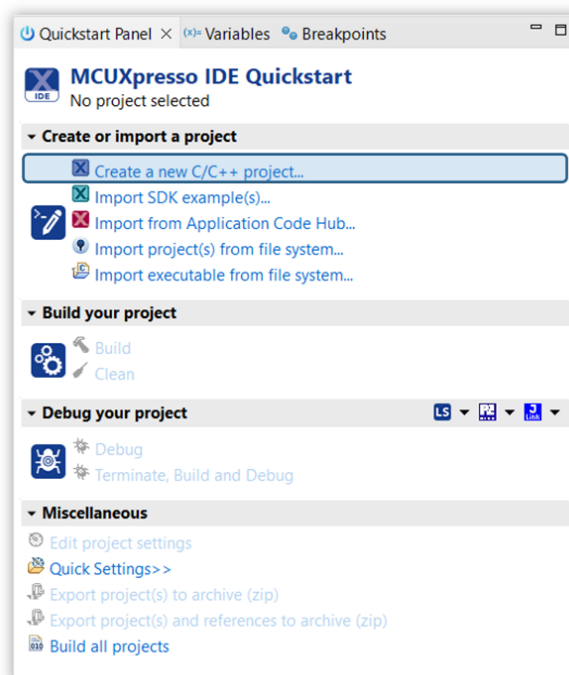
Figure 51. SDK projects

The **New Project Wizard** guides the user in creating new projects from the installed SDKs (and also from preinstalled part support - which are discussed in a later chapter).

Click *Create a new C/C++ project* to launch the New Project Wizard as detailed below:

5.1 New Project Wizard

The New Project Wizard begins by opening the "Board and/or device selection" page, which contains a range of features described below:

Figure 52. New Project Wizard first page

1. A display of all parts (MCUs) installed via SDKs. Click to select the MCU and filter the available matching boards. You can hide SDK part support by clicking the triangle (highlighted in the blue oval)

2. A display of all preinstalled parts (these are all LPC or Generic M parts). Click to select the MCU and filter the available matching boards (if any). You can hide preinstalled part support by clicking the triangle (highlighted in blue)

3. A display of all boards from both SDKs or matching LPCOpen packages. Click to select the board and its associated MCU.
   • Boards from SDK packages have **SDK** superimposed onto their image.

4. Some description relating to the selection of the user

5. A display to show the matching SDK for a chosen MCU or Board. If more than one matching SDK is installed, the user can select the SDK to use from this list

6. Any Warning, Error, or Information related to the current selection

7. An input field to filter the available boards, for example, 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.

Tip: *Upon project creation, the wizard remembers the selected board and/or MCU and selects them the next time it is opened. To remove this selection, click the clear filter button (or any background white space)*

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

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

First, to reduce the number of boards displayed, we can simply type '64' into the filter (7). Now, the wizard only displays boards with MCUs matching '64'.
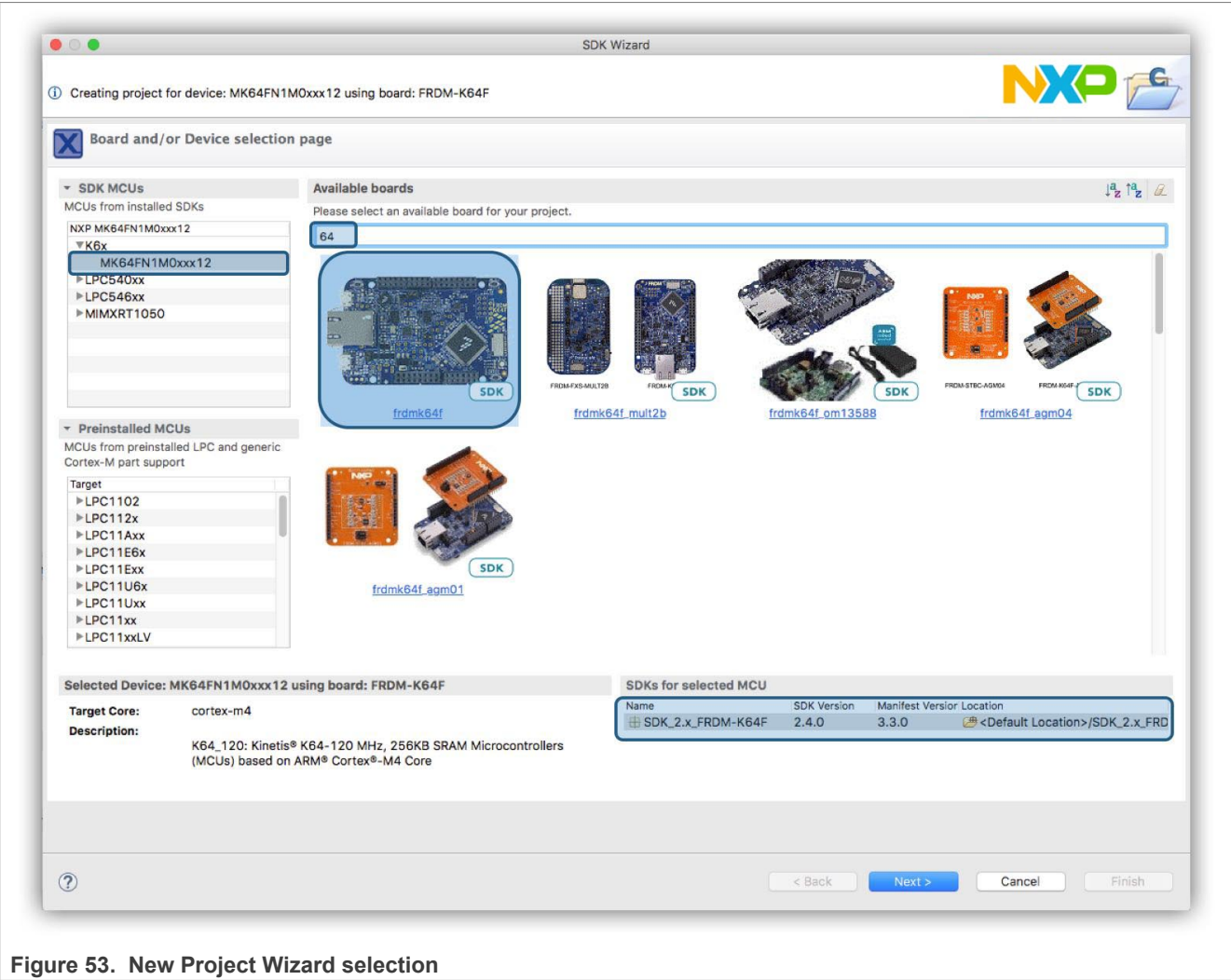
Figure 53. New Project Wizard selection

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

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

Note: The project has a default name based on the MCU name. If this name matches a project within the workspace, for example, the wizard has previously been used to generate a project with the default name, then

the error field shows a name clash and the 'next' and 'finish' buttons are 'grayed out'. To change the name of the new project; the blank 'Project Name Suffix' field can be used to quickly create a unique name but retain the original prefix.

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

However, the wizard offers the flexibility to select/change many build, library, and source code options. We describe these options and the components of this first Wizard page below.
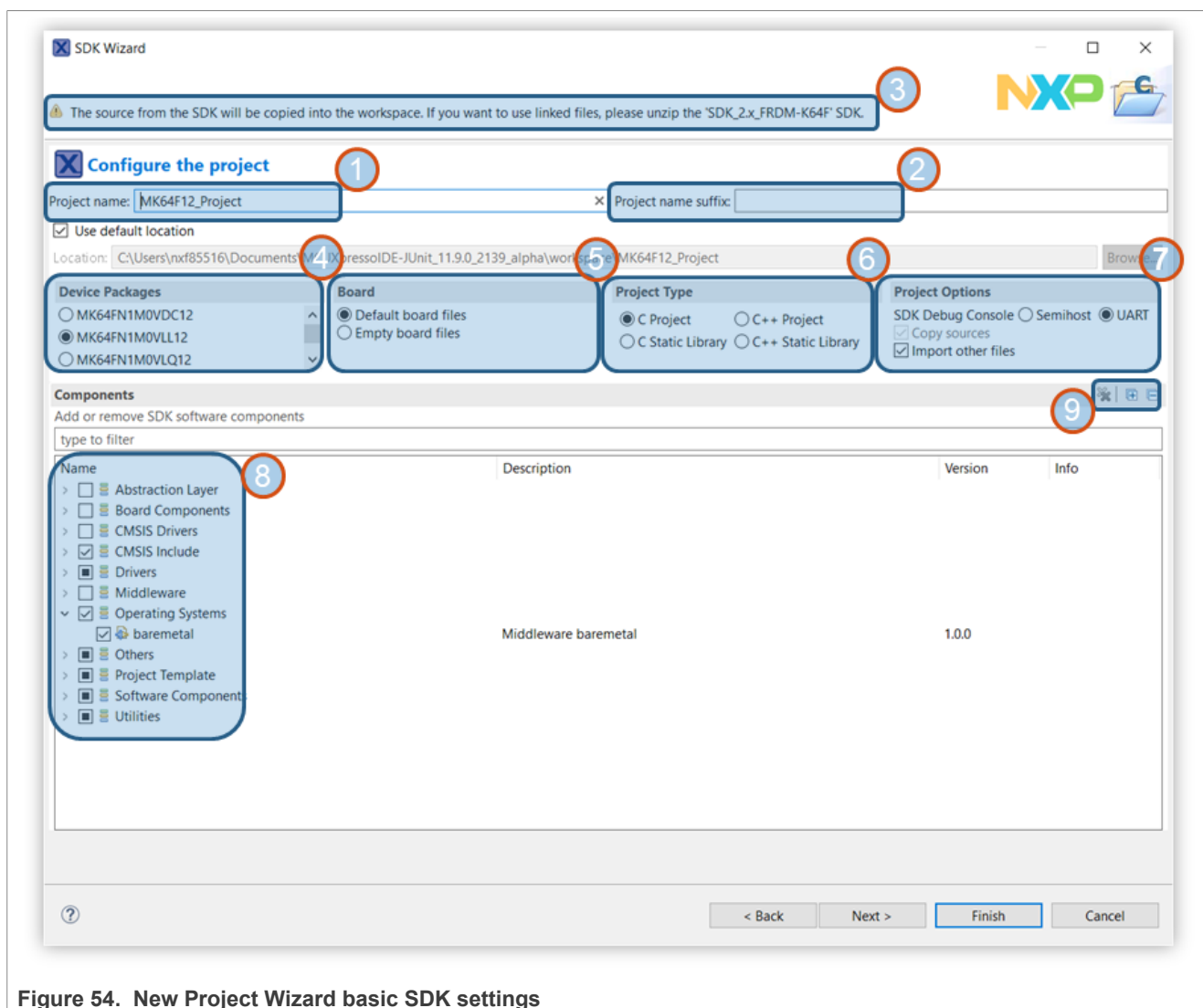
Figure 54. New Project Wizard basic SDK settings

1. Project Name: the wizard automatically selects the default project name prefix based on the part selected on the previous screen
   - **Note**: due to restrictions in the length of filenames accepted by the Windows version of the underlying GCC toolchain, it is recommended to keep the length of project names to 56 characters or less. Otherwise, you may see project build error messages regarding files not being found, particularly during the link step.
2. Project Suffix: the user can enter an optional suffix to append to a project name here.
3. Errors and Warnings: the wizard displays any error or warning here. The 'Next' option is not available until after handling every error. Errors may include such things as dependency problems or, for example, the

selecting of a project name that matches an existing project name in your workspace. The suffix field (2) allows a convenient way to create a unique project name.

4. MCU Package: the user can select the device package 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, otherwise, empty files are created. These options do not appear if the user selected a part rather than a board on the previous screen.
   - If you intend to use board-specific features such as output over UART, you should ensure that you have selected Default board files.

6. Project Type: you can select C or C++ projects or libraries. Selecting 'C' automatically selects RedLib libraries, while selecting C++ selects NewlibNano libraries. See [C/C++ library support](#)

7. Project Options:
   - Semihost: causes the Semihosted variant of the chosen library to be selected. For C projects this defaults to Redlib Semihost-nf. Semihosting allows IO operations such as printf and scanf to be emulated by the debug environment.
   - UART: causes the nohost variant of the chosen library to be selected. For C projects this defaults to Redlib Nohost. IO operations such as printf and scanf occur via UART (or emulated UART provided by the debug probe over USB).
   - Copy Sources: for zipped SDKs, this option is ticked and grayed out. For unzipped SDKs, the wizard allows the creation of projects using linked references to the SDK sources.

8. Components:
   - OS: this provides the option to pull in and link against Operating System resources such as FreeRTOS.
   - Driver: enables the selection of supporting driver software components to support the MCU peripheral set.
   - CMSIS Drivers: code and headers for standard ARM hardware.
   - CMSIS Include: causes a CMSIS folder containing various support code such as Clock Setup, header files to be created. It is recommended to leave the associated options ticked.
   - Utilities: a range of optional supporting utilities.
     – For example, select the debug_console to use SDK Debug Console handling of IO.
     – Selecting this option causes the wizard to substitute the (SDK) PRINTF() macro for C Library printf() within the generated code.
       – The debug console option relies on the debug probe communicating to the host via VCOM over USB (LPC-Link2 and OpenSDA debug probes support this feature).
   - Middleware: enables the selection of various middleware components.
   - Project Template: adds support files for the selected device/board.
   - Depending on the SDK selected, additional options may also appear.

9. Utility buttons: you can use these to clear all selections, expand component sets, or collapse them. These buttons affect only the currently filtered results.

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

Important Note: Any components (OS, driver, utilities, middleware, and so on) selected by default within this wizard are available for use within the project. However, the linker may remove the components supporting functions from the generated image if they are not referenced from within the user's project code. Also, selecting a component automatically selects any dependencies. Finally also note that this is an additive process, **removing components may leave unresolved dependencies resulting in a project that does not build**.

Note: Some middleware components are not currently compatible with the New project wizard functionality and so are hidden. The recommended approach if such components are required is to import an example including the component and then modify this as required. See [SDK project component management](#) for details of how this might be done.

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

5.1.2 SDK New Project Wizard

The advanced configuration page takes certain default options based on settings from the first wizard project page, for example, a C project pre-selects Redlib libraries, whereas a C++ project pre-selects NewlibNano.

Figure 55. New Project Wizard advanced SDK settings
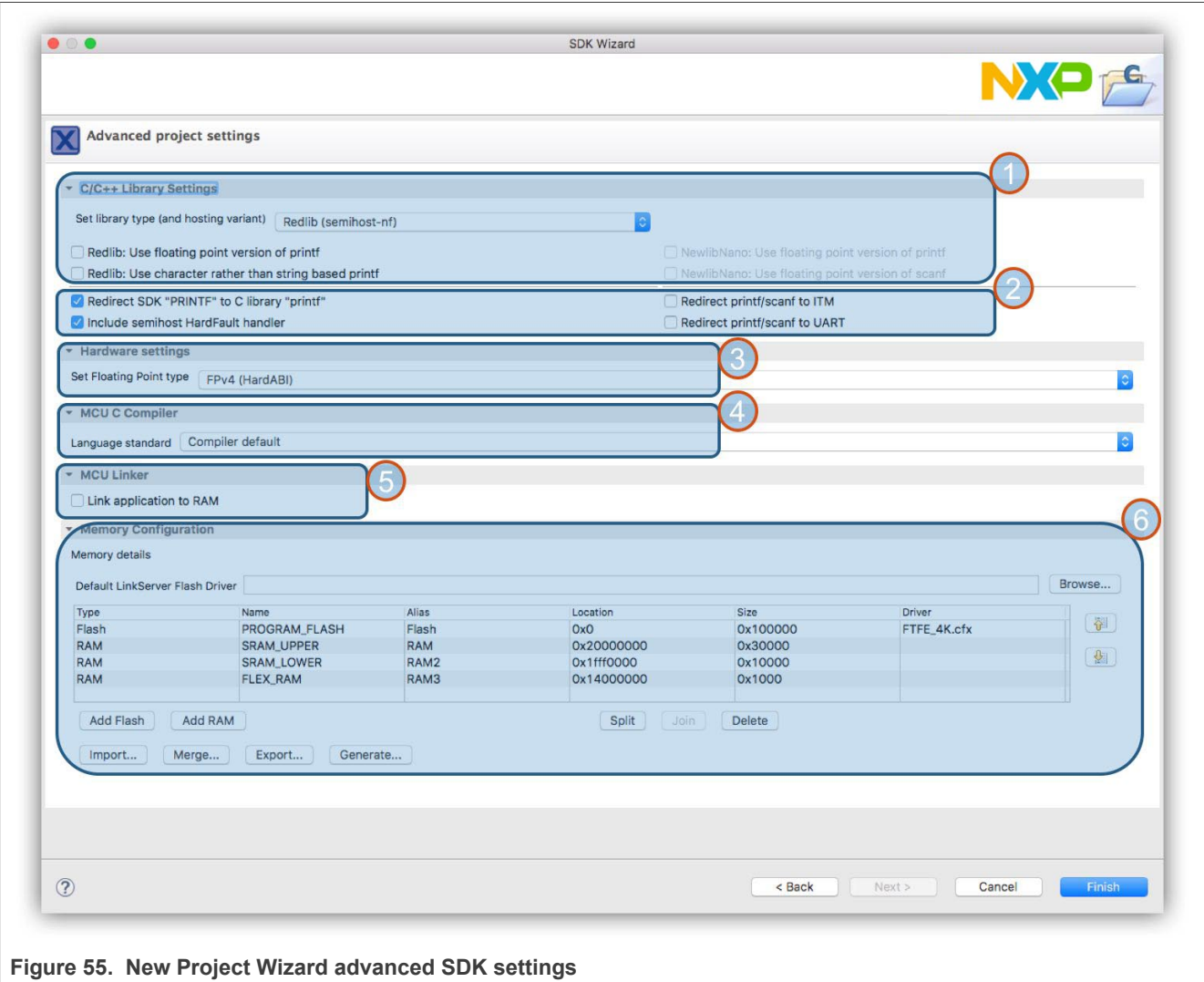
1. This panel allows the selection of library variants. See C/C++ library supportNote: if you selected a C++ project on the previous page, then the Redlib options are grayed out.

   -

   Redlib (none)
   Redlib (nohost)
   Redlib (semihost)
   Redlib (nohost-nf)
   ✓ Redlib (semihost-nf)
   Redlib (semihost-mb)
   Redlib (semihost-mb-nf)
   NewlibNano (none)
   NewlibNano (nohost)
   NewlibNano (semihost)
   Newlib (none)
   Newlib (nohost)
   Newlib (semihost)
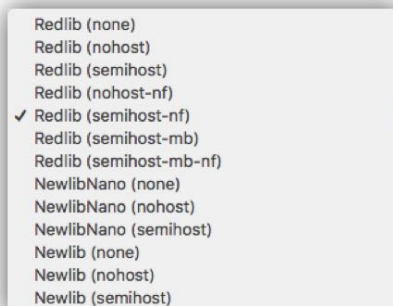
   Figure 56. Library variants

   - Also, based on the selection, you can choose several options to modify the capability (and size) of printf support
   - Redlib Floating Point printf: If this option is ticked, floating point support for printf is automatically linked in. This allows printf to support the print of floating point variables at the expense of larger library support code. Similarly, for Newlib.
   - Redlib use Character printf: selecting this option avoids heap usage and reduce code size but make printf operations slower.

2. This panel allows you to set options related to Input/Output. See C/C++ library support
   - Redirect SDK "PRINTF": many SDK examples use a PRINTF macro, selecting this option causes redirection to C library IO rather than options provided by the SDK debug console.
   - Include Semihost Hardfault Handler: selected by default, this option when checked adds a hard fault handler to the project sources. This handler is specifically written to deal with the situation that occurs if a semihosted function such as printf is executed when there are no debug tools attached to support the operation. If this occurs, this handler catches the operation and safely returns to the executing application. Uncheck this option if you do not wish to use semihosted libraries or you intend to use your own hard fault handler. See semihosted printf for more information.
   - Redirect printf/scanf to ITM: causes a C file `retarget_itm.c` to be pulled into your project. This then enables printf/scanf I/O to be sent over the SWO channel. The benefit of this is that I/O operations can be performed with little performance penalty. Furthermore, these routines do not require debugger support and, for example, could be used to generate logging that would effectively go to Null unless debug tools were attached. **Note**: This feature is not available on Cortex M0 and M0+ parts.
     – Find more information in the MCUXpresso IDE SWO Trace Guide.

- Redirect printf/scanf to UART: Sets the define SDK_DEBUGCONSOLE_UART causing the C libraries printf functions to re-direct to the SDKs debug console UART code.

3. Hardware Settings: from this dropdown, you can set options such as the type of floating point support available/required. This defaults to an appropriate value for your MCU.

-

Figure 57. Hardware settings

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

-
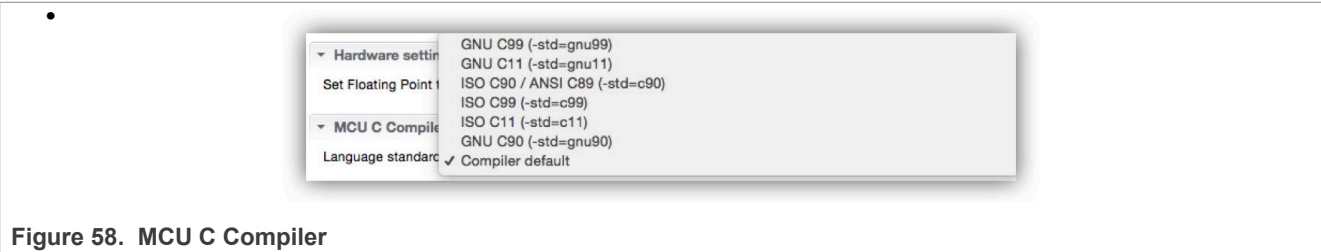
Figure 58. MCU C Compiler

5. Link Application to RAM checkbox reflects or sets the option to force the linker to ignore any defined flash regions and link the application to the first RAM region defined. This option is a copy of the flag at *Properties -> C/C++ Build -> Settings -> Managed Linker Script -> Link application to RAM***Note**: This setting is only sensible for projects under development, since debug control or a bootloader is required to load the code/ data into RAM and simulate a processor reset.

6. 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 only applies to LinkServer (CMSIS-DAP) debug connections.

- From this dialog, you may edit the default memory setting of the project in place if required and therefore also the automatically generated linker scripts. See Memory configuration and linker scripts

5.2 Project build

To build a project (created by the New Project Wizard), simply select the project in the 'Project Explorer' view, then go to the '**Quickstart**' Panel and click the **build button** to build the project. This builds the active configuration of the selected project, where newly created projects default to the *Debug* configuration.

Note: MCUXpresso IDE creates projects with two build configurations, Debug and Release (but you can also add more if required). These differ in the default level of compiler optimization. Debug projects default to None (- O0), and Release projects default to (-Os). For more information on switching between build configurations, see How do I switch between Debug and Release builds?

The console view displays the build log, as shown below.
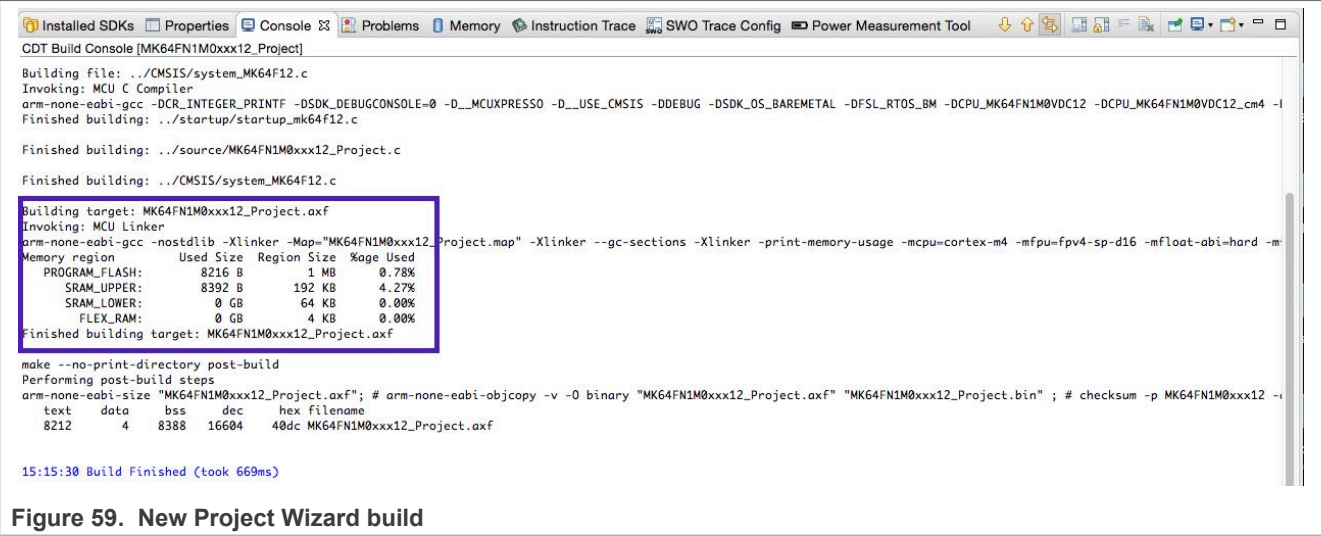
Figure 59. New Project Wizard build

We also show below the memory usage of the project as highlighted in the above screenshot:

```
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 builds and links against the first Flash memory found within the memory configuration of the device. For most MCUs there is only one Flash device available. In this case our project requires 8216 bytes of Flash memory storage, 0.78% of the available Flash storage.

RAM is used for global variables, the heap, and the stack. MCUXpresso IDE provides a flexible scheme to reserve memory for Stack and Heap. The above example build has reserved 4 KB each for the stack and the heap. See Memory configuration and linker scripts for detailed information.

Also see Image information for details on how to explore the composition of an image in detail.

5.2.1 Build configurations

By default, MCUXpresso IDE creates each project with two different "build configurations": **Debug** and **Release**. Each build configuration contains a distinct set of build options. Therefore, a **Debug** build typically compiles its code with optimizations disabled (@-O0@) and **Release** compiles its code optimizing for minimum code size (@-Os@). You can see the currently selected build configuration for a project after its name in the Build/Clean/Debug options of the **Quickstart** Panel.

6   Importing example projects (from installed SDKs)

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

To import examples from an installed SDK, go to the **Quickstart** panel and select **Import SDK example(s)**.
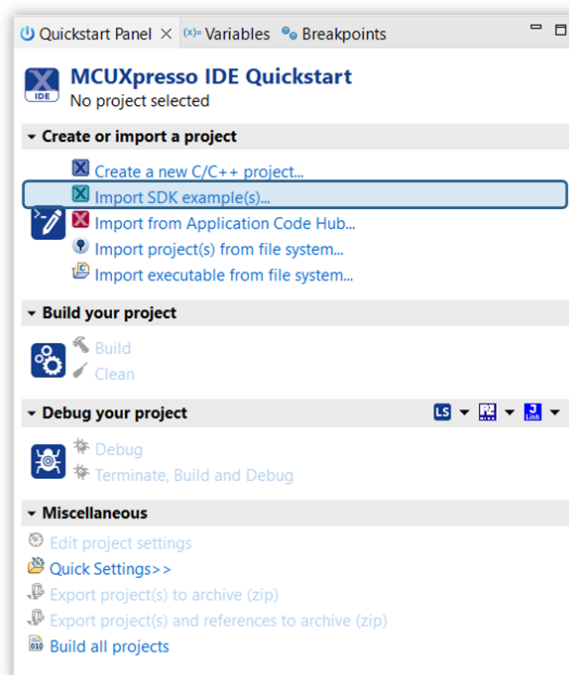
Figure 60. SDK example

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

Like the New Project wizard, this initially launches a page allowing MCU/board selection. However, now, this displays only SDK-supported parts and boards.

Figure 61. SDK example board
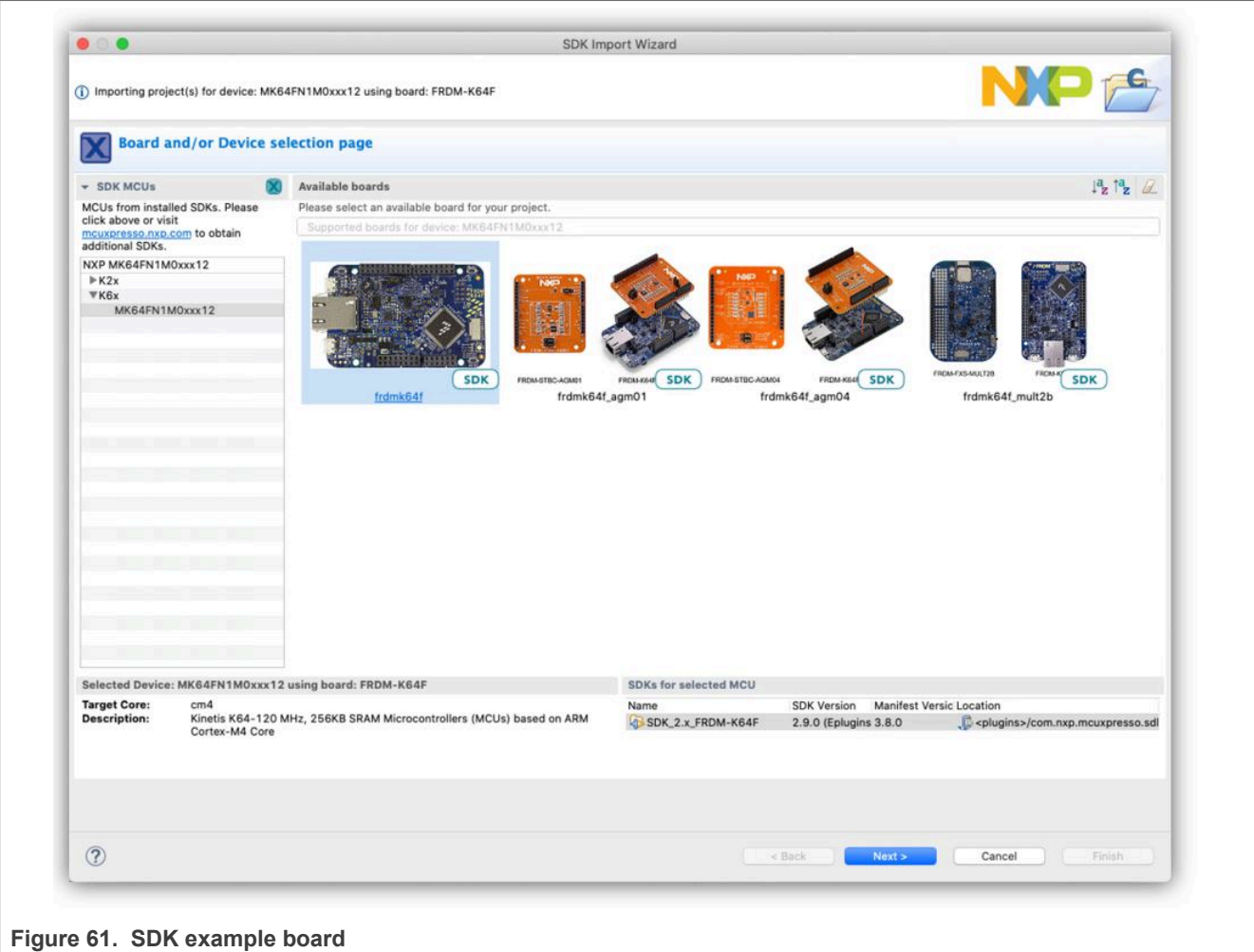
6.1 SDK example import wizard

Selection and filtering work in the same way as for the New Project Wizard, but be aware that examples are created for particular development boards, therefore you must select a board to move to the 'Next' page of the wizard.

In the case of *SDK Example Import Wizard*, the *SDKs for selected MCU* control is disabled and the IDE automatically performs the selection of the proper SDK source for a specific chosen board/device. This prevents the unwanted selection of an SDK, which can lead to getting files from a wrong source. This situation can occur when the same device appears in multiple SDKs.
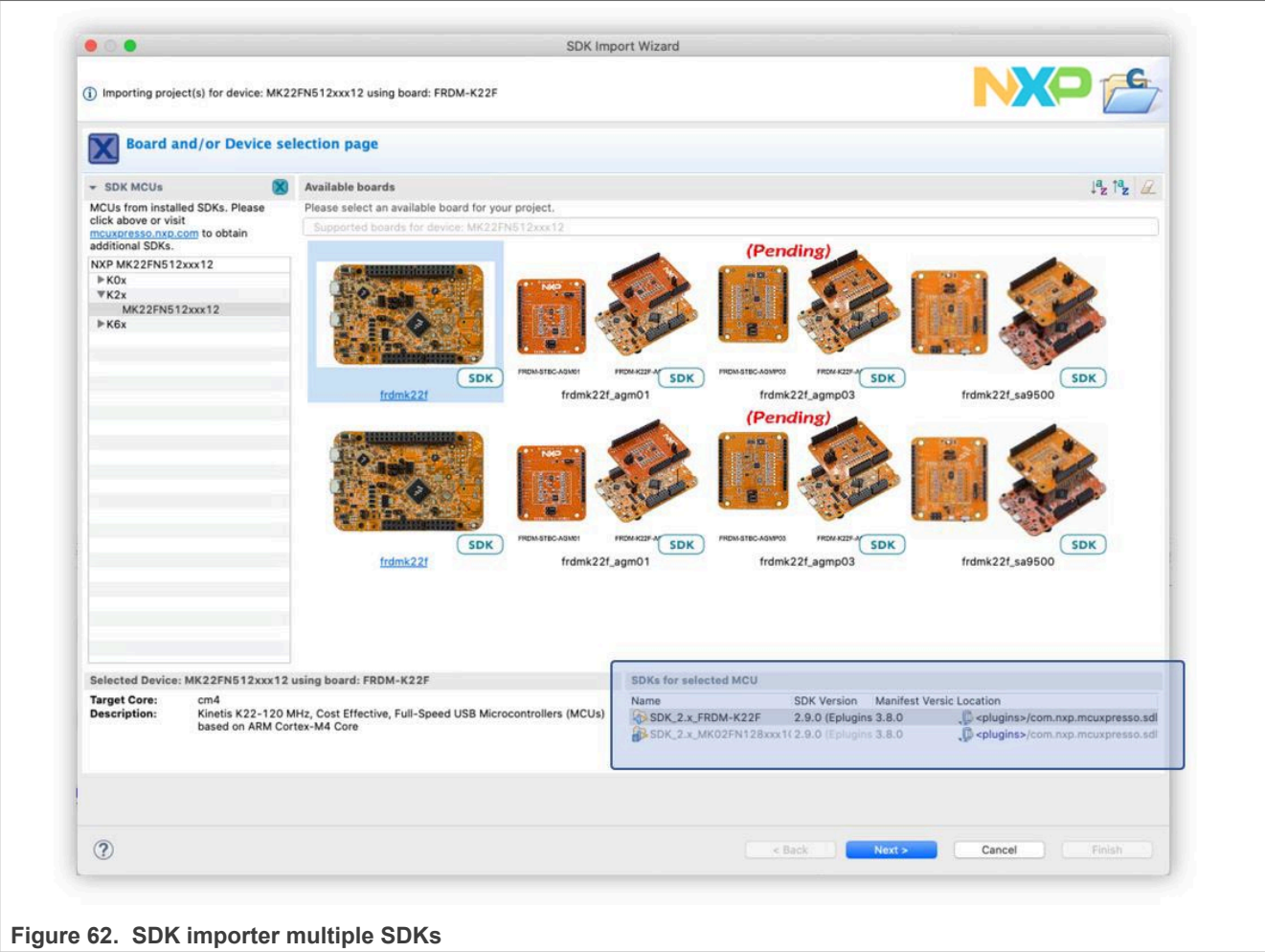
Figure 62. SDK importer multiple SDKs

Note: Even if not recommended, if it is necessary, the user can still force the option. You can remove this hard-coded selection by selecting the option from *Preference -> MCUXpresso IDE -> SDK Handling -> Misc -> Enable "SDKs for selected MCU" selection in SDK Import Wizard*. Then, you can select a different SDK.
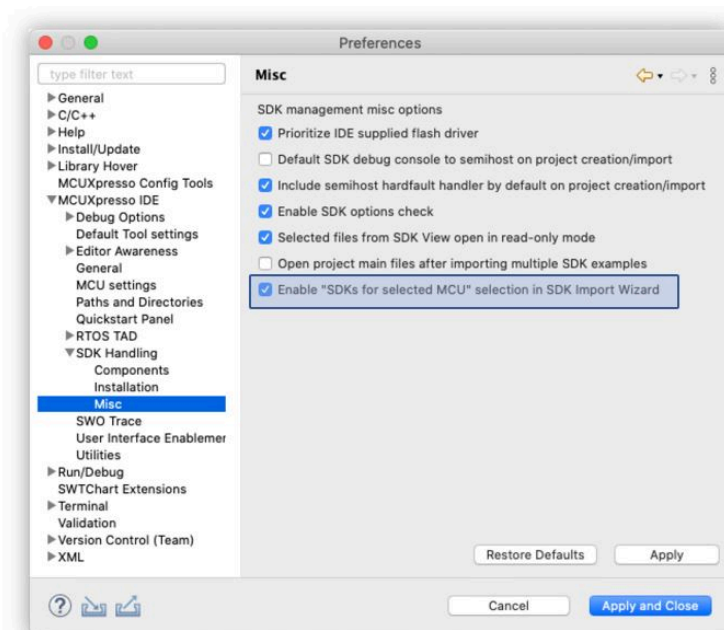
Figure 63. SDK importer force manual SDK selection

6.1.1 SDK example import wizard

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 you have selected a single example for import. This is because examples may set specific options, and therefore changing settings globally is not sensible.

The first page offers all the available examples in various categories. You can expand these to view the underlying hierarchical structure. We explain the various settings and options below:

Note: The project has a default name based on the MCU name, Board name, and Example name. If this name matches a project within the workspace, for example, if the wizard has previously been used to generate an example with the default name, then the error field shows a name clash and the 'next' and 'finish' buttons are grayed 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, for example, by adding '1'.

MCUXpresso IDE creates 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. We describe these options and the components of this first wizard page below.
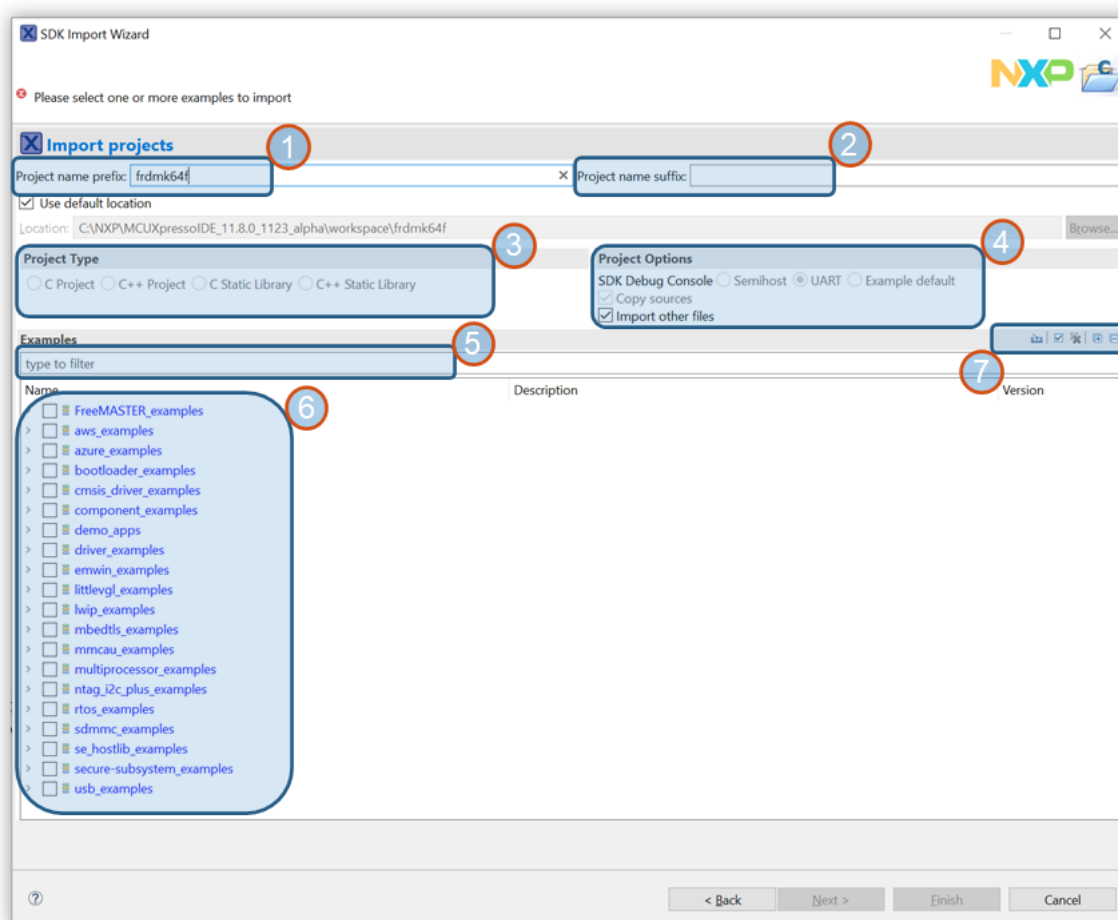
Figure 64. SDK example selection

1. Project Name: An automatically created project name follows the form: *boardname_examplename*
2. Project Suffix: You can enter an optional suffix to append to a project name 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: The pre-set type of the example being imported controls this option. If you wish to import more than one example, then these options appear grayed out.
4. Project Options:
   - 'SDK Debug Console': After selecting an example(s), you can use this option to control IO between the semihost console, UART, or the examples' default setting.
   - 'Copy sources': For unzipped SDKs, you can untick this option to create a project containing source links to the original SDK files. This option should only be unticked with care, since editing the linked example source overwrites the original files!
   - 'Import other files': By default, non-source files such as graphics are filtered out during import, check this box to import all files.
5. Examples Filter: Enter text into this field to find possible matches, for example, enter 'LED' or 'bubble' to find examples present in many SDKs. This filter is case-insensitive.

6. Examples: The example list broken into categories. **Note**: for some parts, there are many potential examples to import

7. Various options (from left to right):

   • Opens a filer window to allow the use to import an example 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, it is possible to select 'Finish' to finish the wizard. Alternatively, if the user has selected only one example, 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, 263 is indicated for the FRDM MK64 SDK example shown below. Importing many examples takes time... Consider that each example may consist of many files and associated description XML. A single example import may only take a few seconds, but this time adds up for each additional example. Furthermore, the operation of the IDE may be impacted by a large number of projects in a single workspace, therefore it is suggested that example imports be limited to sensible numbers.

Note: Due to restrictions in the length of filenames accepted by the Windows version of the underlying GCC toolchain, it is recommended to keep the length of project names to 56 characters or less. Otherwise, you may see project build error messages regarding files not being found, particularly during the link step.
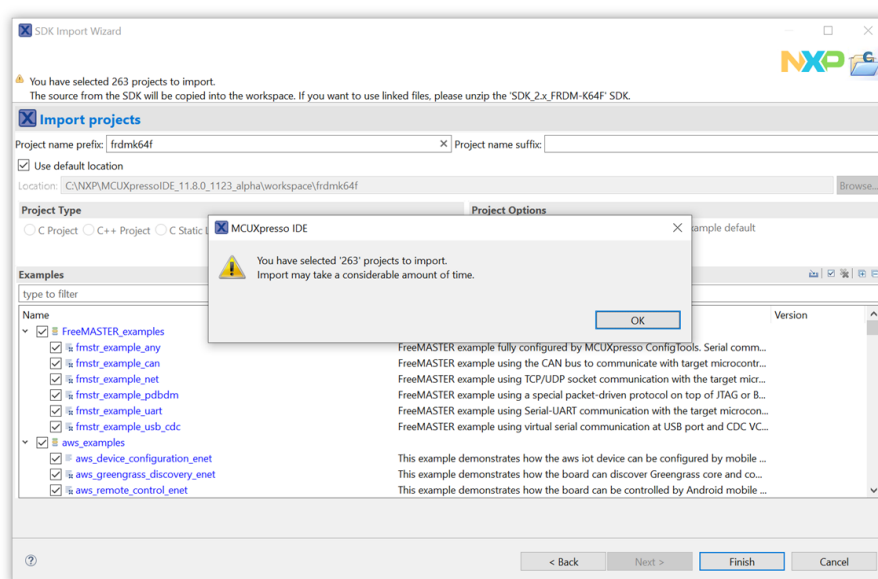
Figure 65. SDK example selection many

6.1.2 SDK example import wizard

The advanced configuration page (shown below) takes certain default options based on the examples selected; for example, a C project pre-selects Redlib libraries, whereas a *C++* project pre-selects NewlibNano.
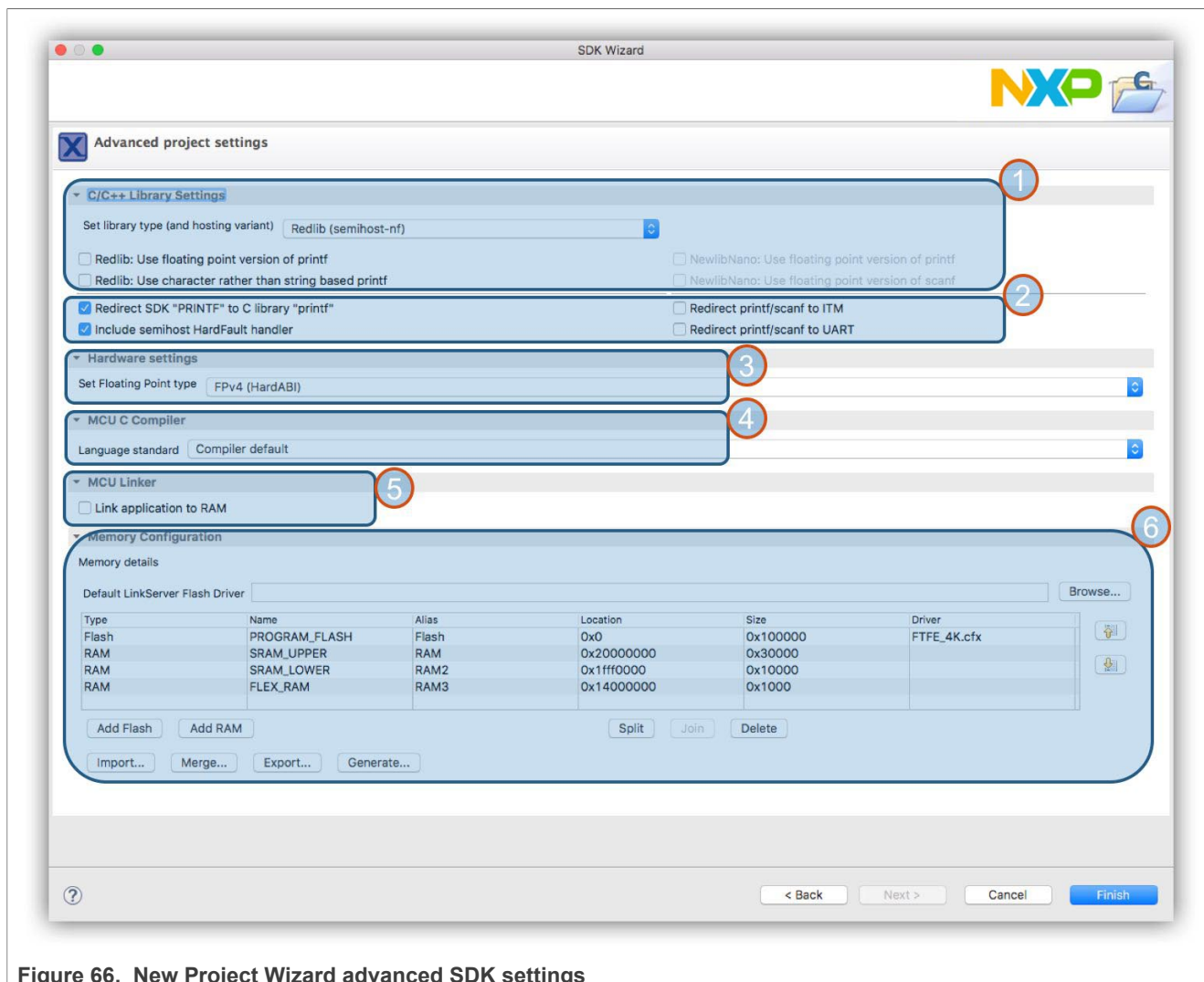
Figure 66. New Project Wizard advanced SDK settings

These settings closely match those in the SDK New Project Wizard description. Therefore, see SDK New Project Wizard: Advanced Options 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

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

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

To create an XML "fragment" for an existing project in your workspace, right-click 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 there are no selected projects) are converted into a fragment within a new folder created in the workspace itself:

Figure 67. Project Explorer Example XML

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

Then select a board and then click the button "Import from XML…" (highlighted below and described in the previous section). You will see the examples definitions from the external fragment in the list of examples as shown and selected below.
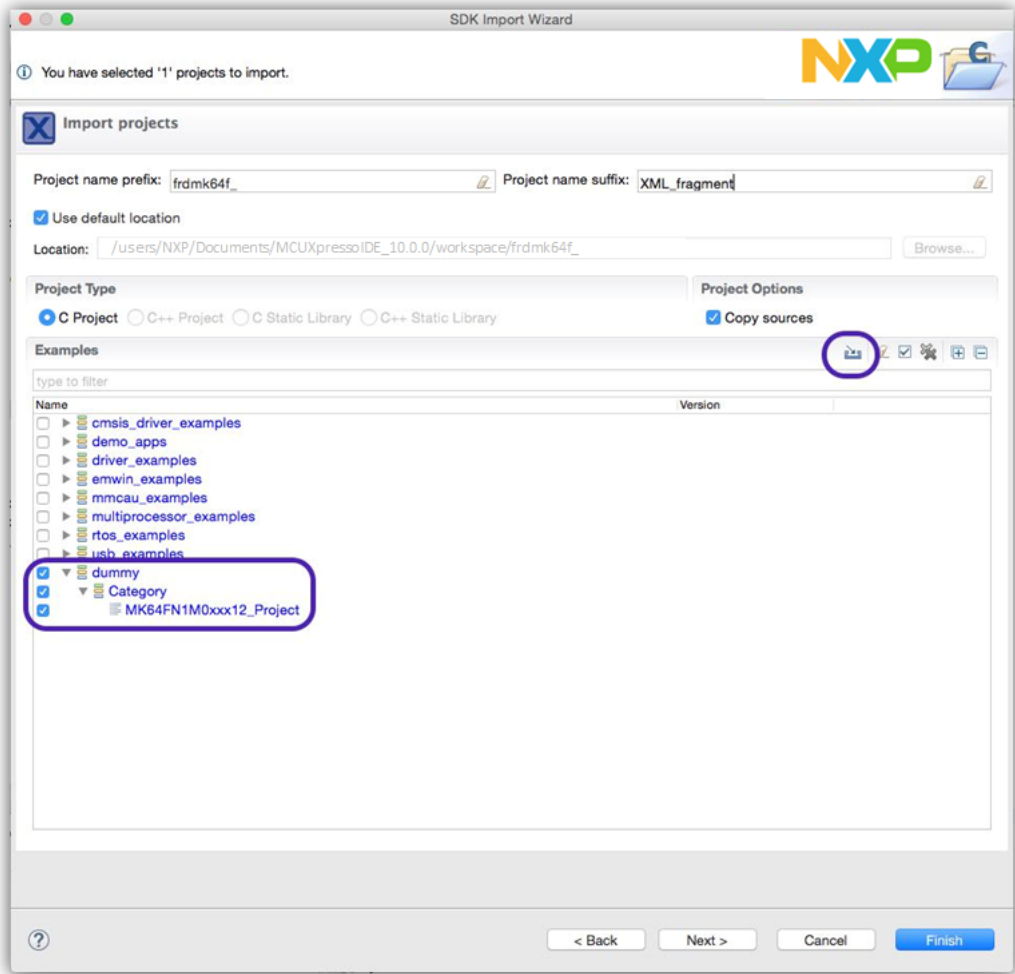
Figure 68. SDK Wizard Import from XML

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

6.1.4 Importing examples to nondefault locations

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

7   Importing projects from Application Code Hub

The Application Code Hub (ACH) repository enables engineers to easily find microcontroller software examples, code snippets, application software packs, and demos developed by NXP's experts. This space provides a quick, easy, and consistent way to find microcontroller applications. The official website provides filtering and searching options to quickly find specific applications.

MCUXpresso IDE integrates the Application Code Hub with all its designed features and allows cloning and importing repositories, SDKs, and projects through dedicated views and wizards.

7.1 MCUXpresso IDE offering

MCUXpresso IDE allows interaction with Application Code Hub through dedicated views and wizards. There are several links to Application Code Hub inside the IDE but the high-level user experience and interaction with the feature are similar.

There are two ways to interact with Application Code Hub. On the one hand, there is the guided wizard, with the important sections highlighted in the screenshot below.
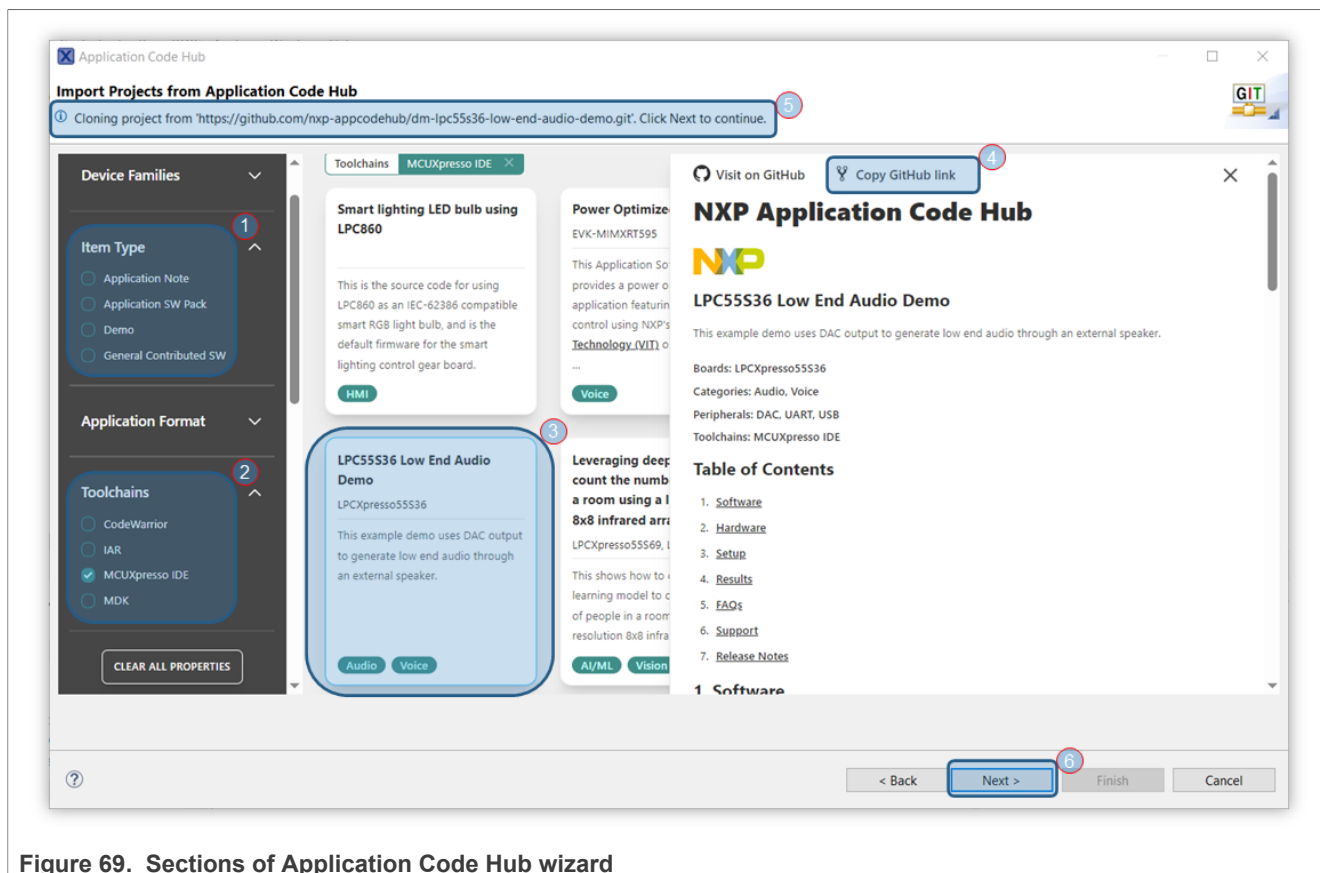
Figure 69. Sections of Application Code Hub wizard

The following sections are most relevant:

1. Filter for types of Application Code Hub projects.
2. Filter for toolchains associated with Application Code Hub projects. The IDE makes sure to filter only MCUXpresso IDE-specific projects while browsing the list of projects.
3. Project selection
4. "Copy GitHub Link" button that is used to instruct the wizard on what project must be handled
5. Guidance and confirmation about the currently selected project
6. "Next" button that activates only the user has selected a project and has copied its GitHub link to Clipboard

On the other hand, it is also possible to render Application Code Hub inside an Eclipse-specific view.
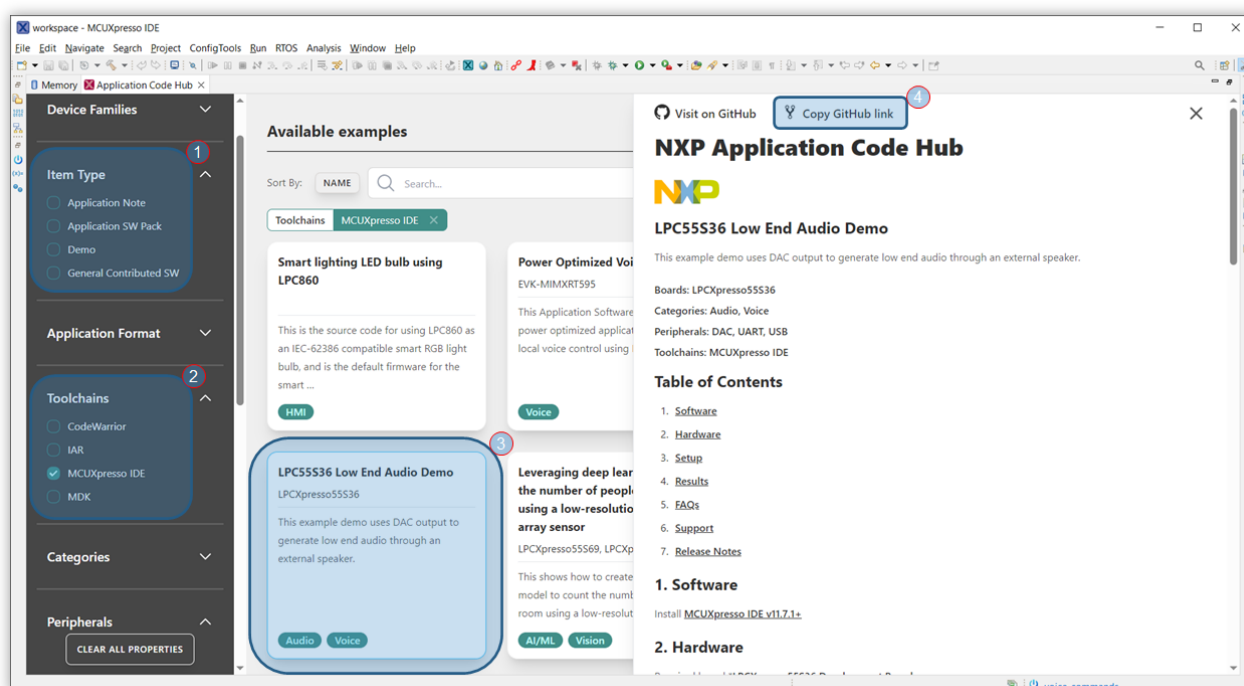
Figure 70. Sections of Application Code Hub view

The highlighted sections inside the view represent:

1. Filter for types of Application Code Hub projects.
2. Filter for toolchains associated with Application Code Hub projects. The IDE makes sure to filter only MCUXpresso IDE-specific projects while browsing the list of projects.
3. Project selection.
4. "Copy GitHub Link" button that is used to instruct the wizard on what project must be handled.

The view does not expose any Eclipse-specific UI controls but the "Copy GitHub Link" offered by the Application Code Hub website provides the linkage between the IDE and the website. In other words, once the user selects a project and clicks "Copy GitHub Link", the IDE checks compatibility with MCUXpresso IDE and identifies the project type to be able to further process the request. In the case of dealing with a noncompatible project, a warning message appears, as illustrated in the picture below.
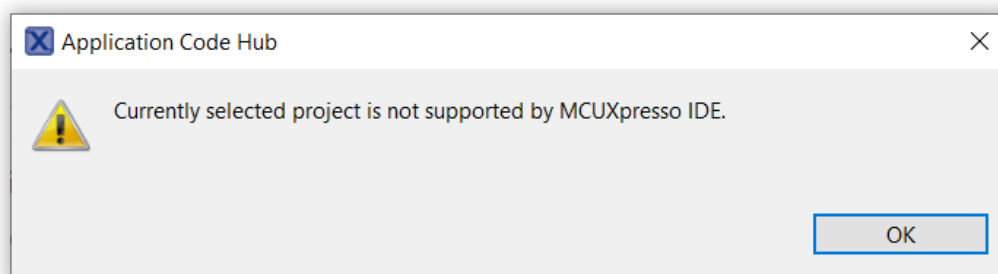
Figure 71. Unsupported project selected in Application Code Hub view

The links that allow access to Application Code Hub features are highlighted in the sections described below.

7.1.1 The import wizard

Access the wizard by going to *File -> Import -> Application Code Hub -> MCUXpresso Projects from Application Code Hub*.
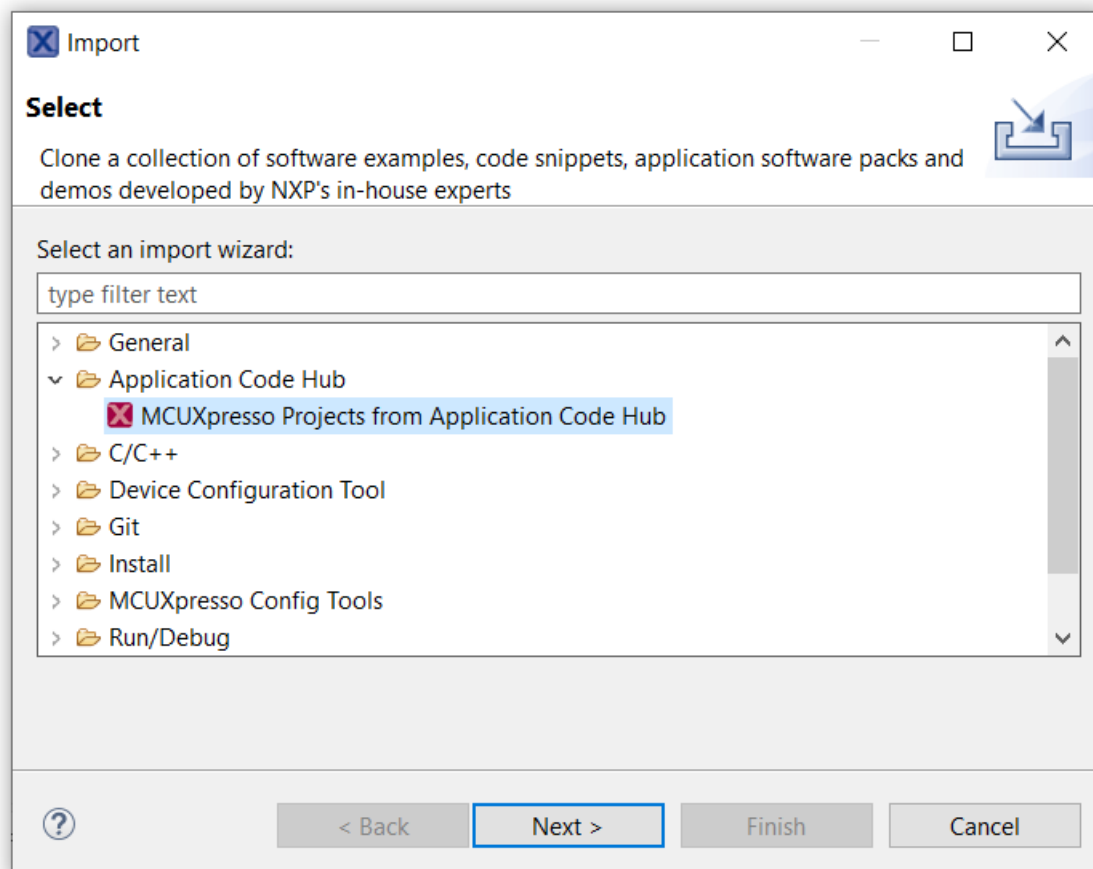
Figure 72. Import wizard from Application Code Hub

7.1.2 The MCUXpresso IDE Quickstart panel link to Application Code Hub import wizard

Open the MCUXpresso IDE Quickstart panel and click the "Import from Application Code Hub..." link.
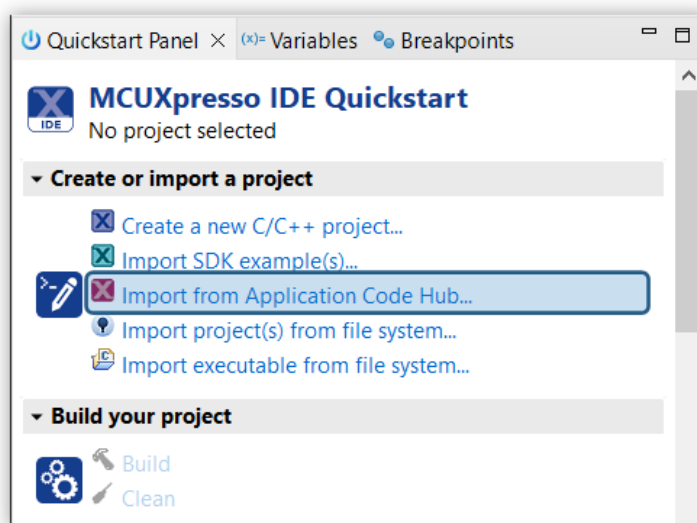
Figure 73. Quickstart panel link to Application Code Hub import wizard

7.1.3 The Additional Resources link to Application Code Hub import wizard

Open the wizard by going to *Help -> Additional resources -> Application Code Hub*.
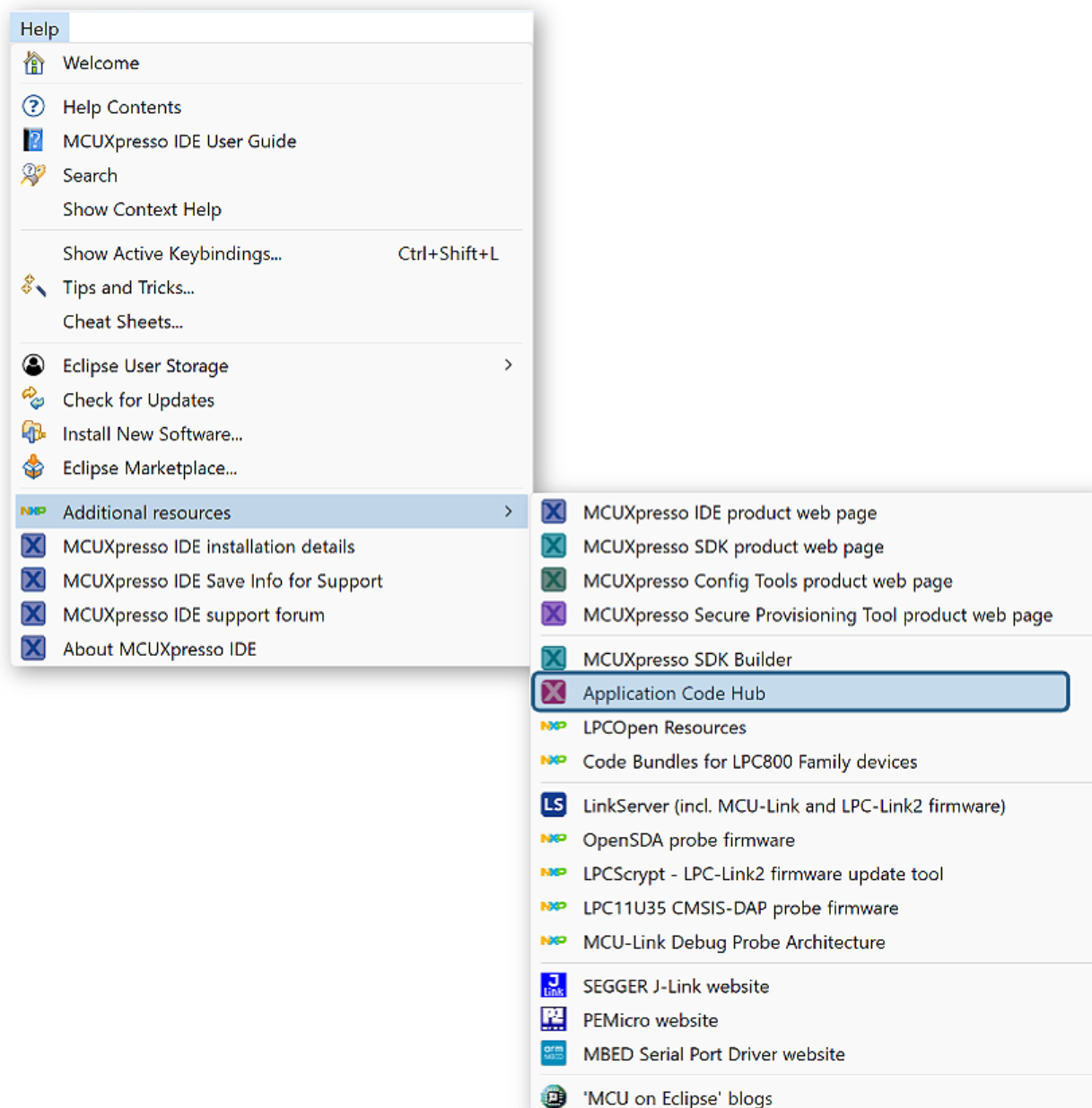
Figure 74. Additional resources link to Application Code Hub import wizard

7.1.4 The dedicated view that renders the Application Code Hub website

To open the view, go to *Window -> Show View -> Other -> MCUXpresso IDE -> Application Code Hub*.
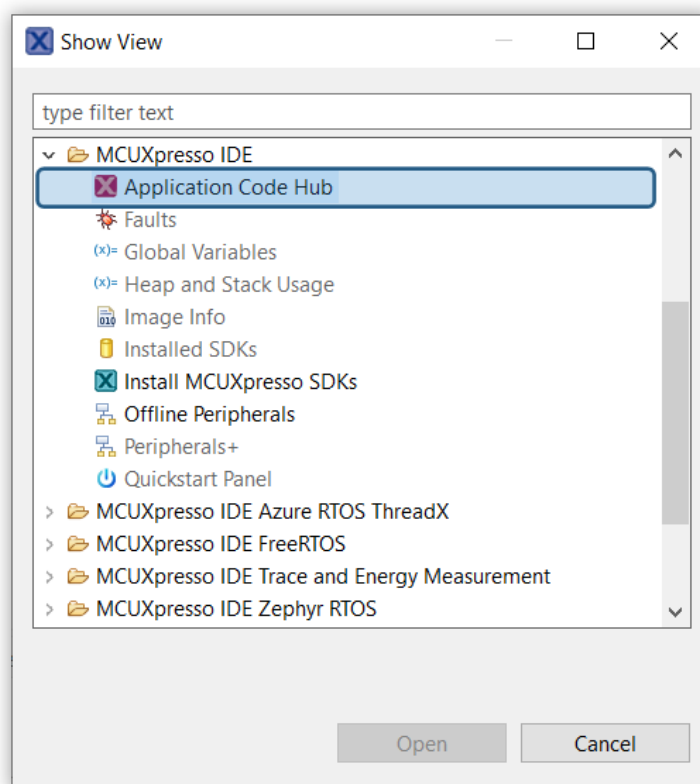
Figure 75. Open Application Code Hub view

7.2 Import of Application SW Packs

The import of Application SW Packs follows a similar flow as Import remote SDK Git repository As a result, the installation of the "west" tool is a dependency for this use case. The IDE checks for availability and shows a relevant error message in case it has not been able to find it.

To import an Application SW Pack from Application Code Hub, you must select a project in one of the previously described wizards, or view. Depending on the project type, the IDE chooses the appropriate pages that are required to complete the cloning and importing steps. In the following sections, we explain the Application SW Packs-specific wizard pages.

7.2.1 Cloning and initialization of Application SW Pack

The first page of the Application SW Pack-specific wizard collects information about the local folder that is used for cloning and the branch to clone. The URL of the repository is prefilled based on project selection inside the Application Code Hub website.
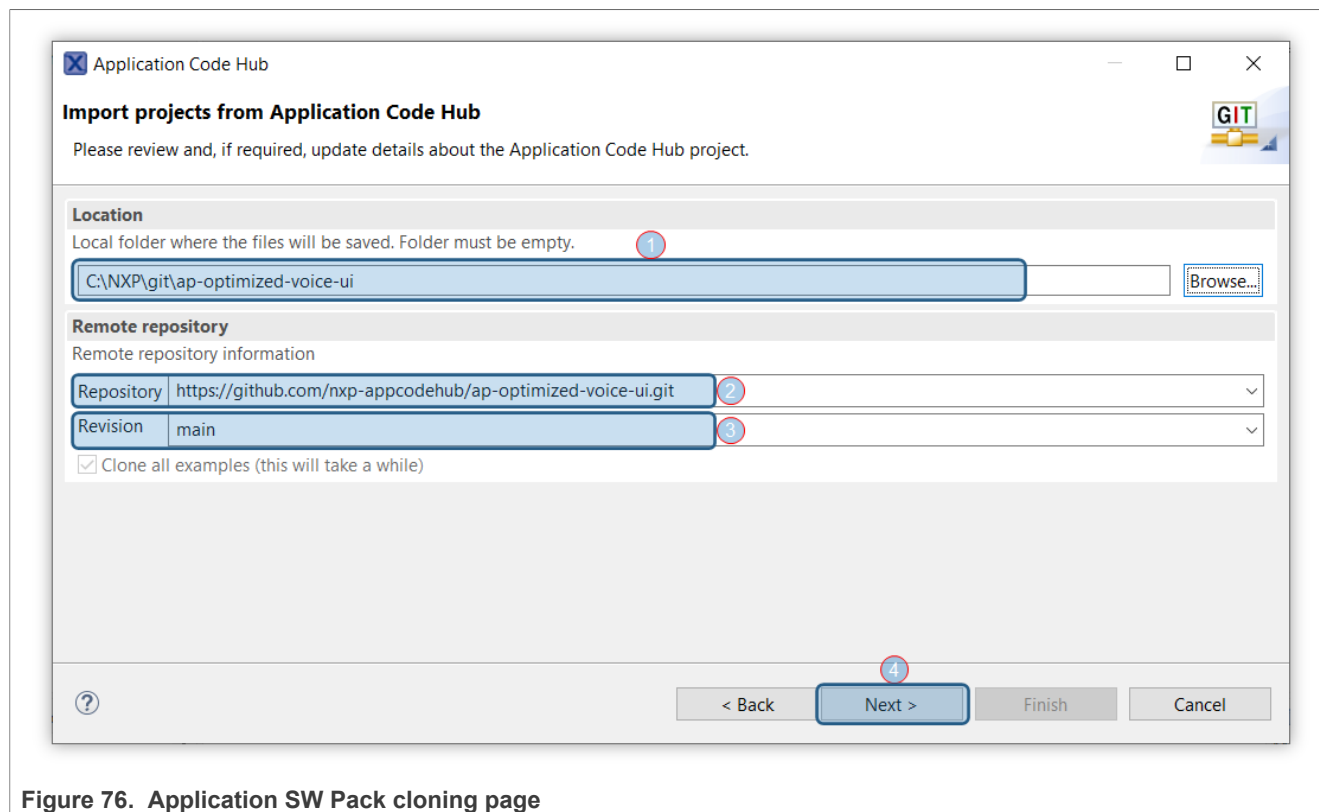
Figure 76. Application SW Pack cloning page

In the above picture, we identify the following relevant sections:

1. Local folder used for cloning the repository. The IDE pre-fills it with a default path but you can change it by browsing to a user-preferred path.
2. URL of the remote repository. The IDE pre-fills it based on the project selection in the previous wizard page.
3. The branch (or the revision) to clone. The IDE retrieves the remote branches and populates the drop-down box with available branches.
4. Button that advances to cloning, initialization, and configuration of the repository.

If all the inputs are successfully validated, the "Next" button allows the cloning, initialization, and configuration of the repository. Once pressed, the wizard advances to the next page, depending on the repository size and on the network connection.
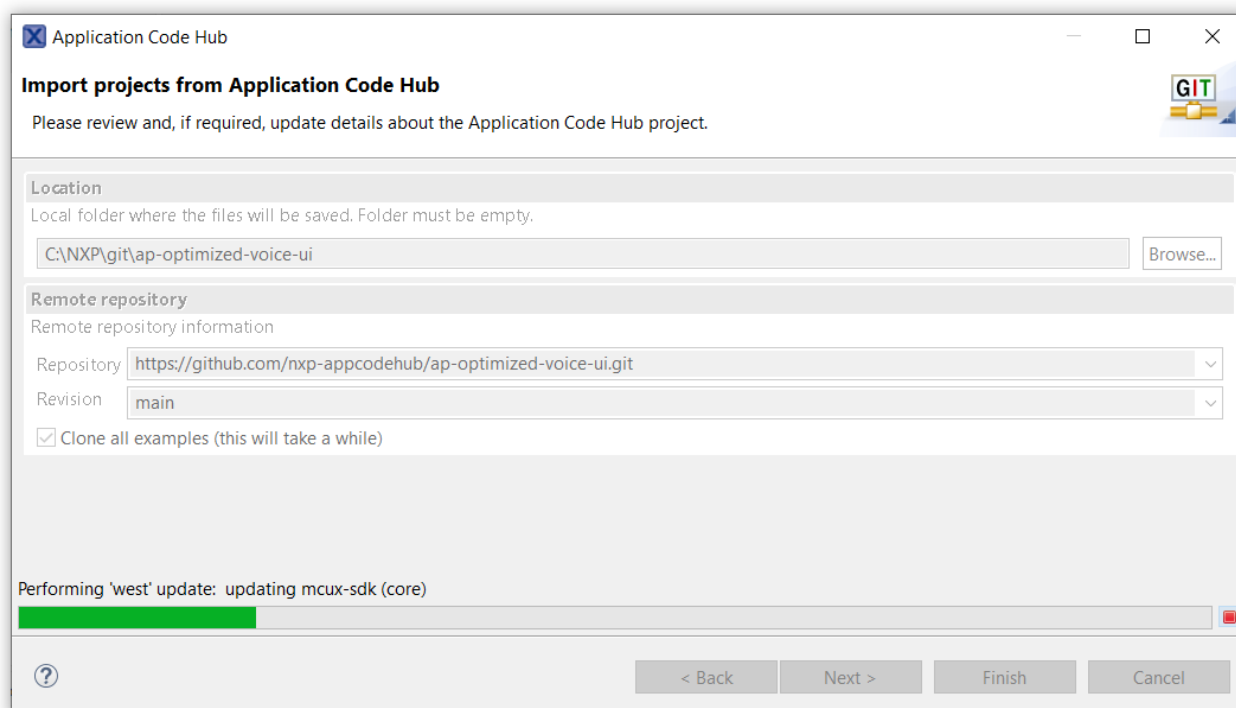
Figure 77. Application SW Pack cloning page with progress

7.2.2 Importing the Application SW Pack in Installed SDKs

Once cloning, initialization, and configuration steps are finished, the wizard allows the import of the actual Application SW Pack. This final step makes the IDE aware of the pack and allows importing any example projects available inside the pack.

In the picture below, we show the last page of the wizard. The following items are relevant for the import action:

1. Local folder of the cloned repository. This is pre-filled with the path specified on the previous page of the wizard.
2. Location of the manifest files, describing the content of the pack. The IDE usually pre-fills it with a path pointing inside the "examples" subfolder.
3. Checkbox that instructs the IDE to start automatically the "Import SDK Example(s)" wizard, once the pack is successfully imported.
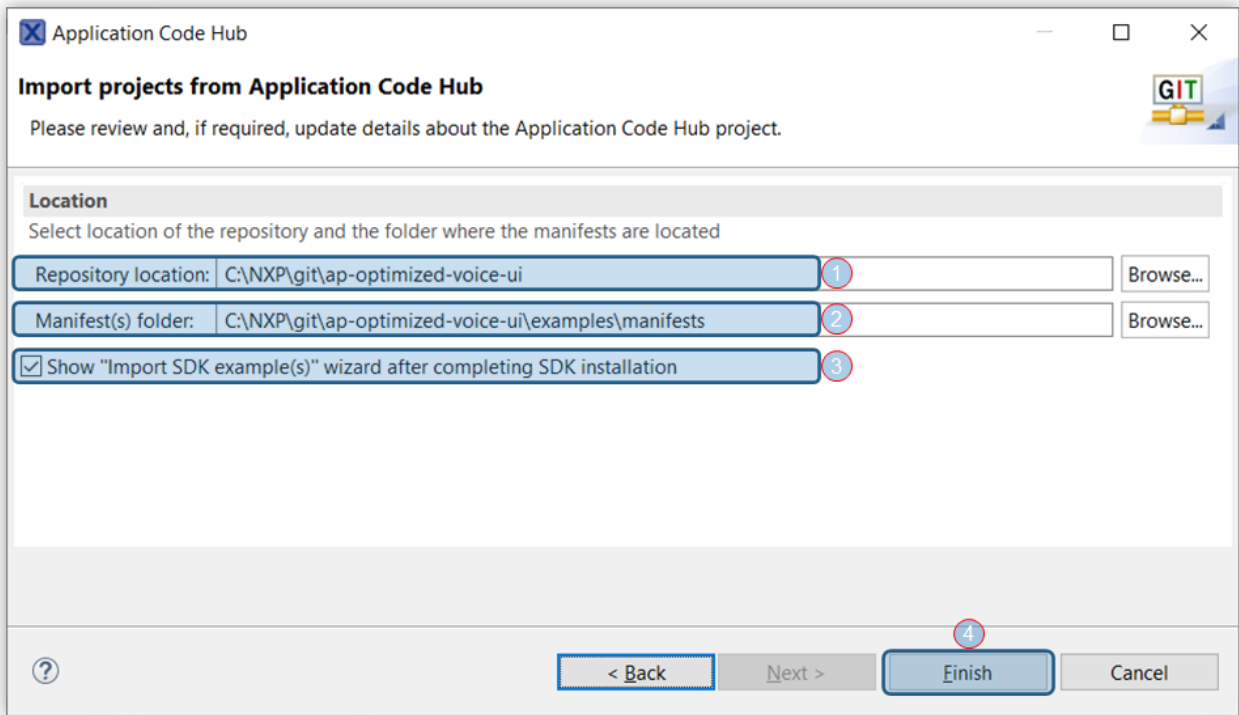
Figure 78. Application SW Pack importing page

The pack is also visible in the Installed SDKs view, once the wizard is closed. The view updates, as shown in the screenshot below.
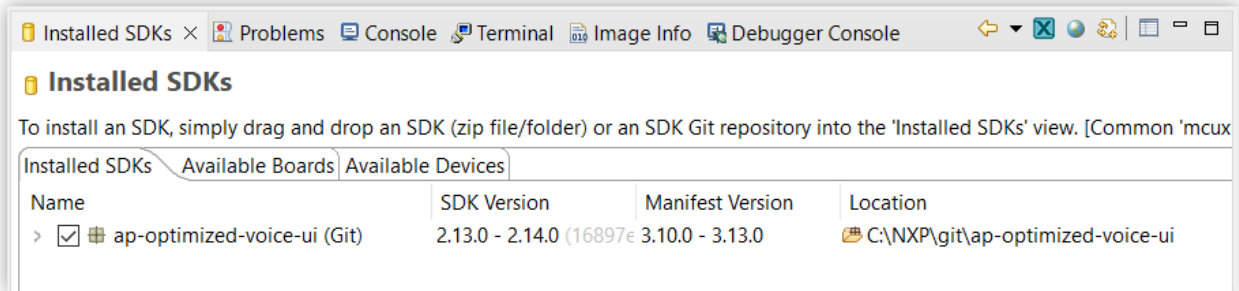
Figure 79. Application SW Pack imported in Installed SDKs view

7.3 Import MCUXpresso IDE-specific projects

This use case relies on the EGit plugins that are preinstalled inside MCUXpresso IDE. You can open the import wizard using one of the methods described above and, similar to the Import of Application SW Packs, the first page renders the Application Code Hub website, allowing selection of the project.

Once the IDE identifies the type of project that must be cloned, several wizard pages guide users to allow the import of MCUXpresso IDE-specific projects.

The first page of the wizard, shown in the picture below, displays the following information about the repository that is about to be cloned:

1. URL of the repository

2. Available remote branches
3. "Next" button, advancing the wizard to the next page
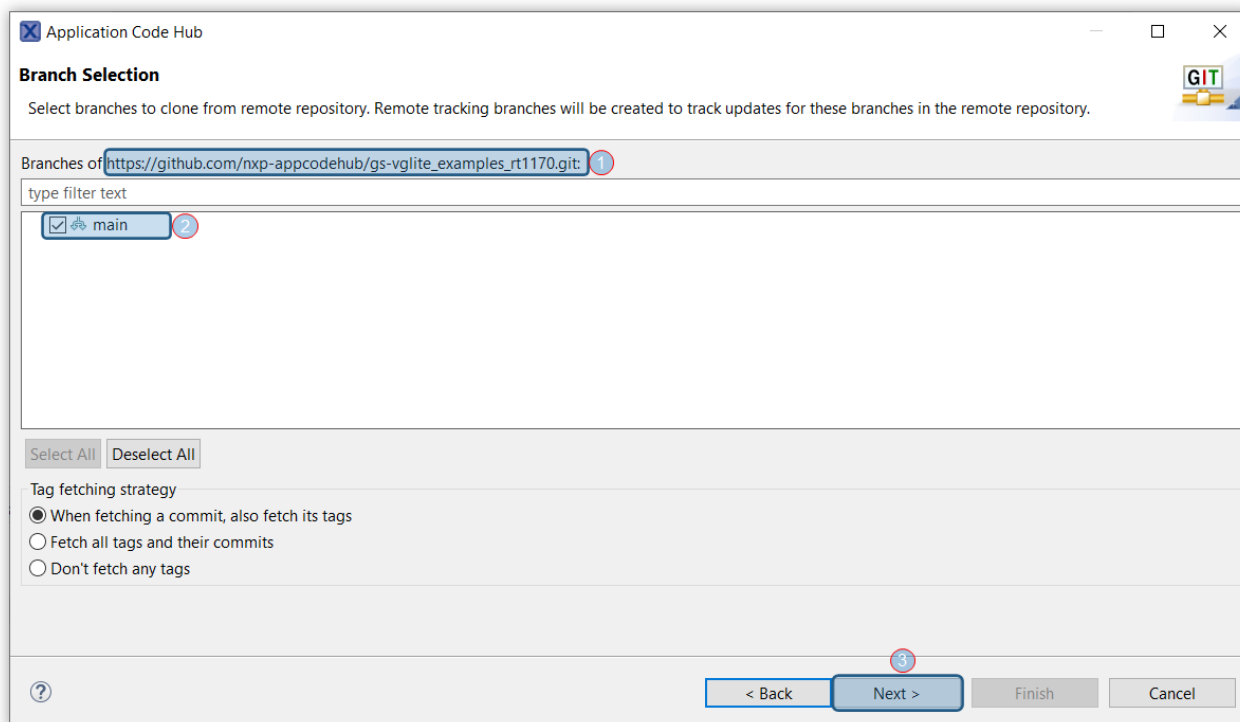
Figure 80. Branch selection of Application Code Hub project

Once pressing "Next", the following page is shown, with the following controls:

1. Local folder where the repository will be cloned.
2. Initial branch to be set in the local clone.
3. "Next" button.
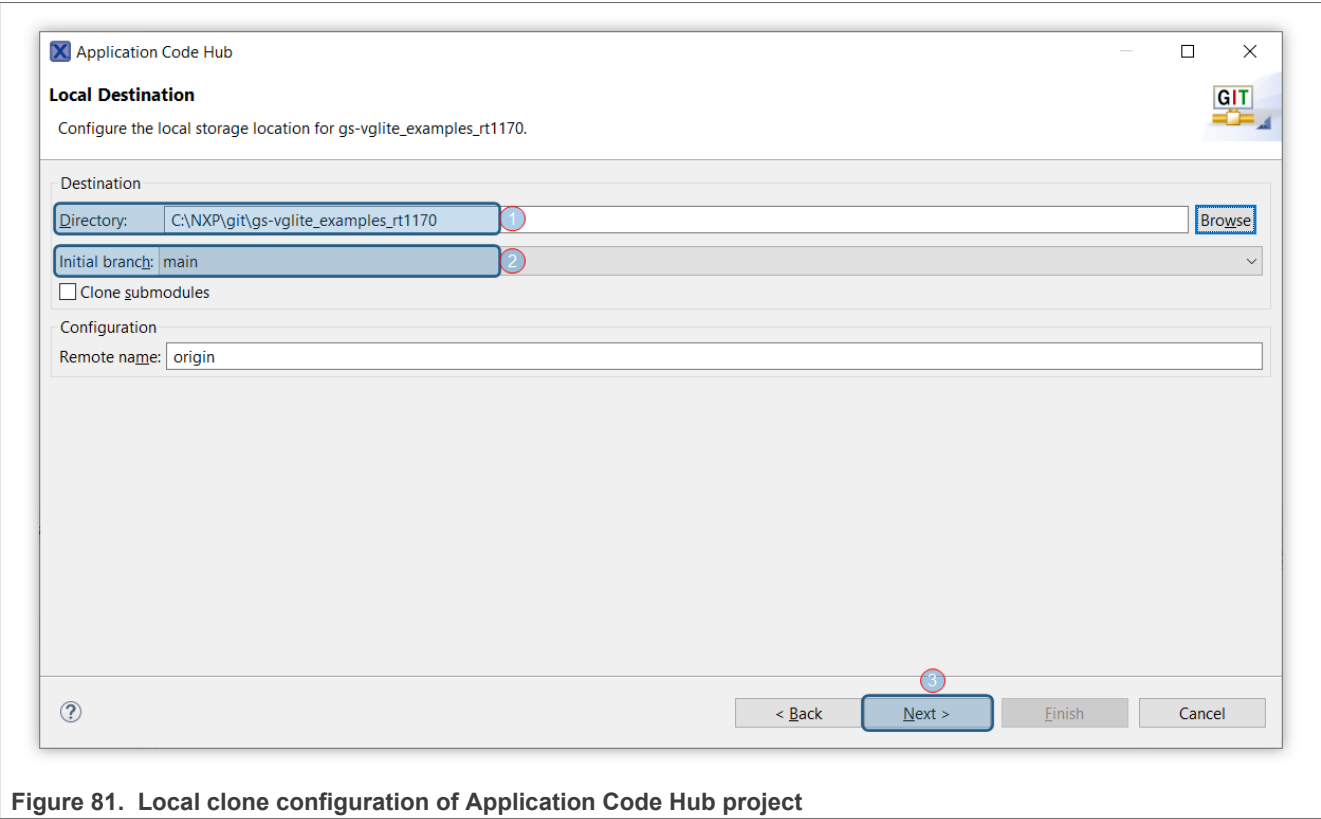
Figure 81. Local clone configuration of Application Code Hub project

The actual cloning and configuration happen after clicking the "Next" button. Make sure that you selected "Import Existing Eclipse projects" in the wizard page illustrated below, and then click "Next".
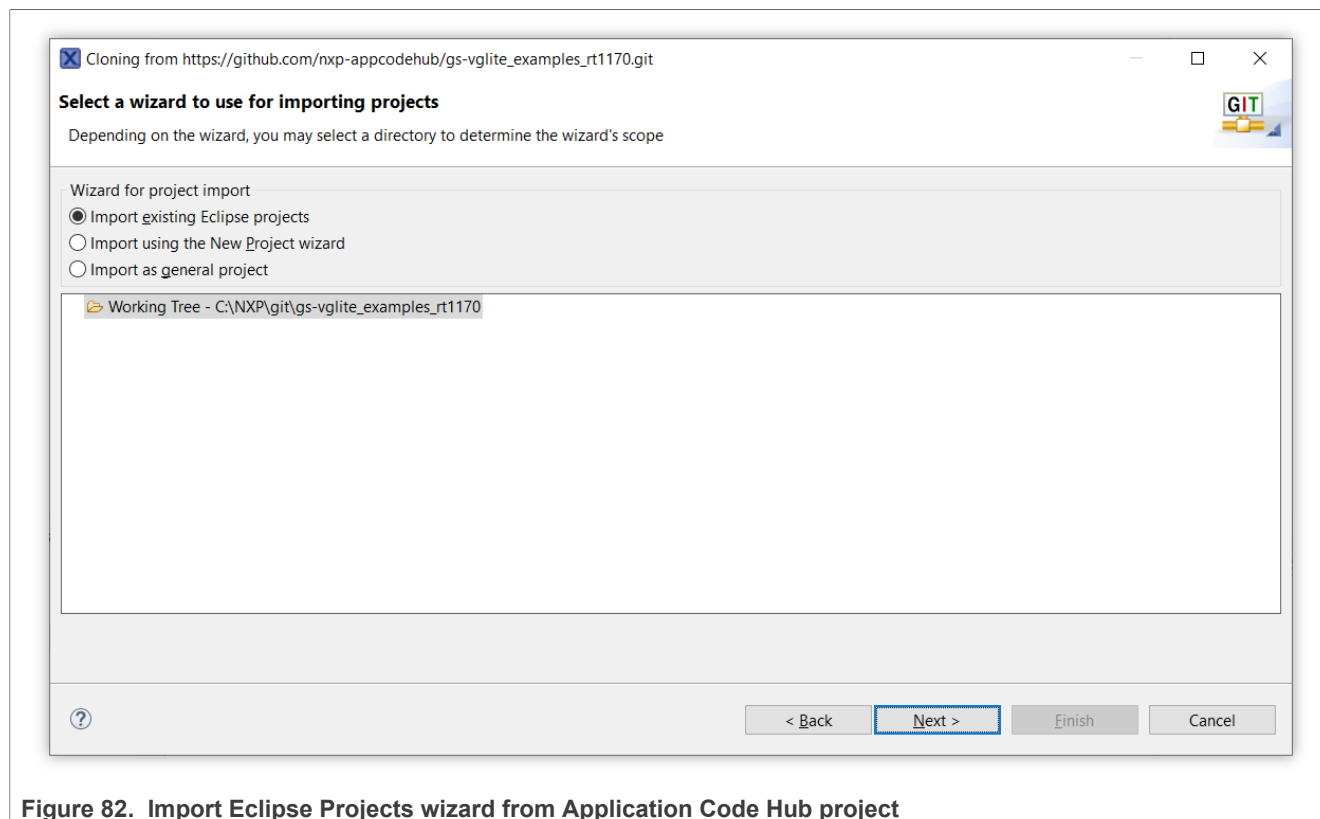
Figure 82. Import Eclipse Projects wizard from Application Code Hub project

The IDE searches for valid Eclipse-specific projects, listing all of them as shown in the wizard page depicted in the screenshot below. Select the desired projects and click "Finish".
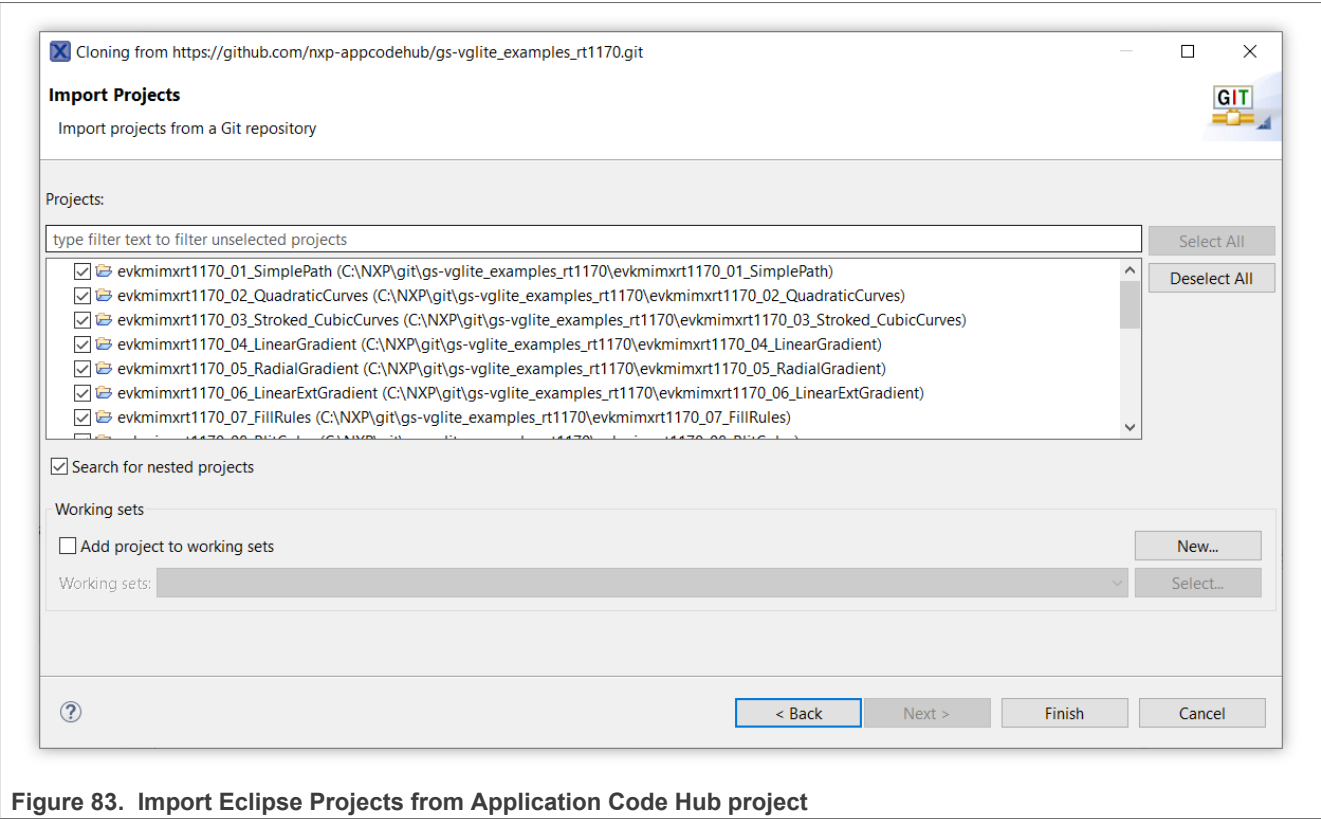
Figure 83. Import Eclipse Projects from Application Code Hub project

All the projects should now be listed inside the Project Explorer view.

8   SDK project component management

Projects and examples created from SDKs contain several software components such as peripheral drivers and/or middleware. In previous versions of MCUXpresso IDE, the option to add components was only available when creating a new project and was not possible for imported examples. MCUXpresso IDE version is able to easily add (or remove) SDK components to a previously created or imported example project via a new *Manage SDK components* wizard. To launch the Manage SDK Components wizard, simply select the chosen project in the Project Explorer view and then click the *package* icon as indicated below:
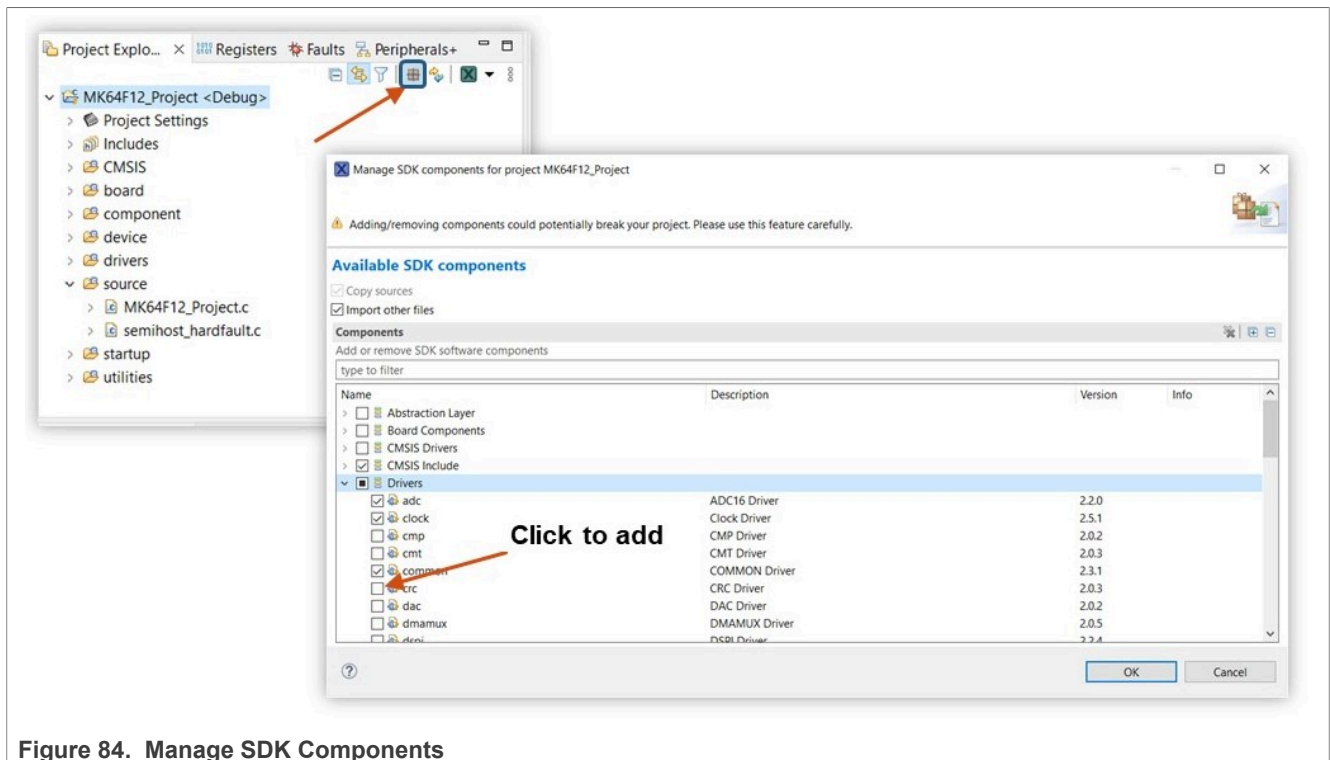
Figure 84. Manage SDK Components

Note: This powerful feature can add (or remove) SDK components **and their dependencies** at a source file level, relying on metadata contained within the SDK. However, also note the following points:

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

Various SDK Component Management options are available from *Preferences -> MCUXpresso IDE -> SDK Handling -> Components*.

8.1 SDK project component management example

To demonstrate the use of this feature, we add the *dac* driver to a project. To do this, launch the Manage SDK components wizard, click the *dac* driver component then click 'OK'.

Next, a dialog appears, listing all of the source files required by this component - as below.
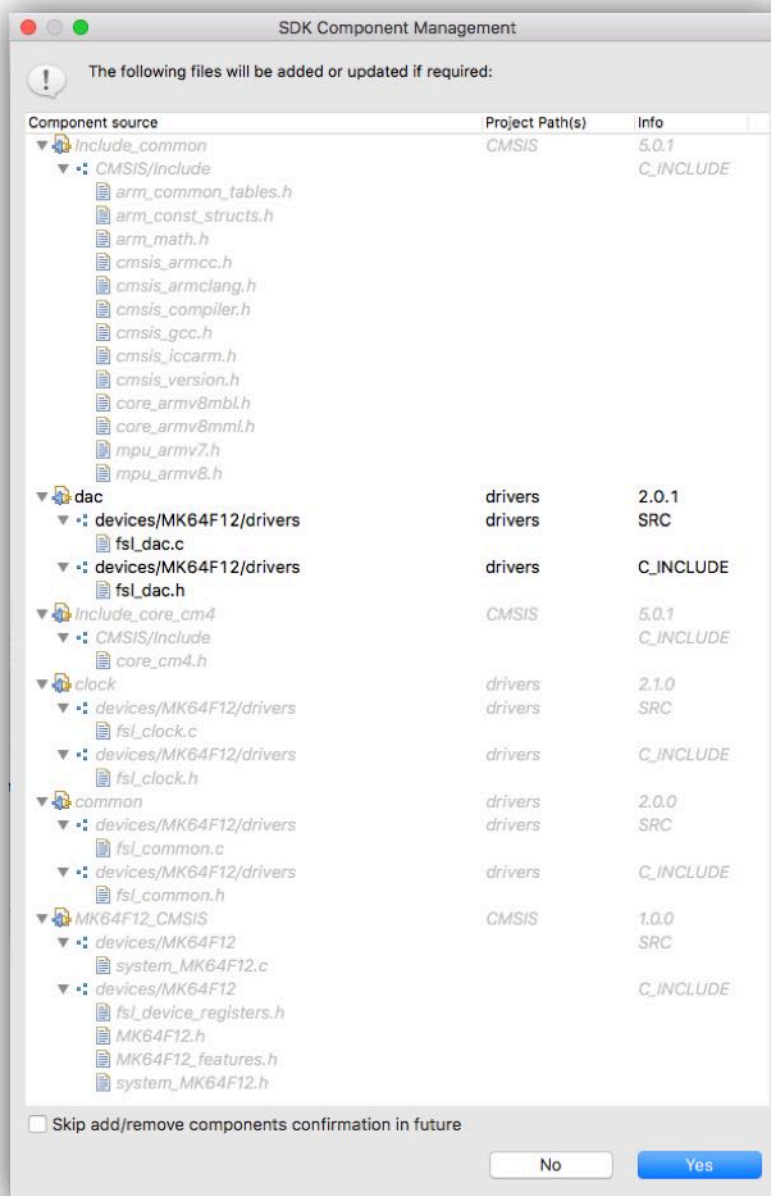
Figure 85. SDK Component Management

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

Click 'Yes' to add these source files to your project.

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

Should a source file difference be found, then a dialog as below appears:
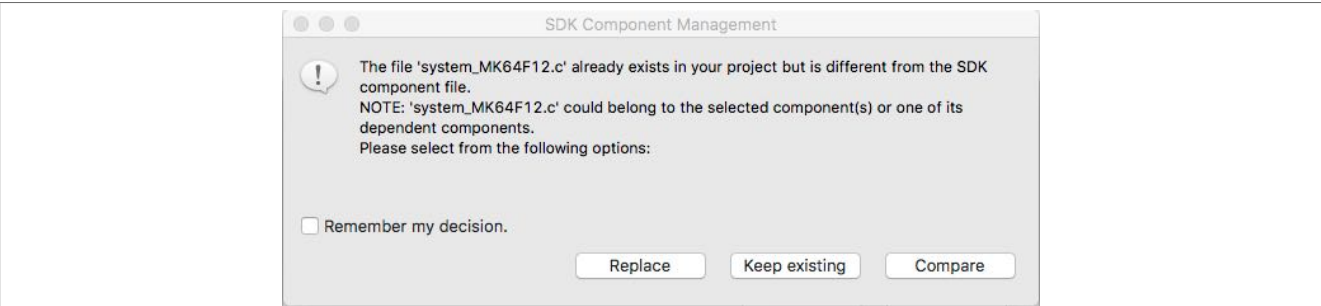
Figure 86. SDK Component Management file difference

From here, you can choose from the following options:

- **Replace** click to overwrite the project's file from the SDK version
- **Keep Existing** click to keep the existing project file unchanged
- **Compare** click to compare the two files - this launches the Eclipse file compare utility, which allows the user to compare the new SDK file with the project copy

In this example, we click 'Compare' ...

Below, you can note the discovery of a modification in the user project source file:
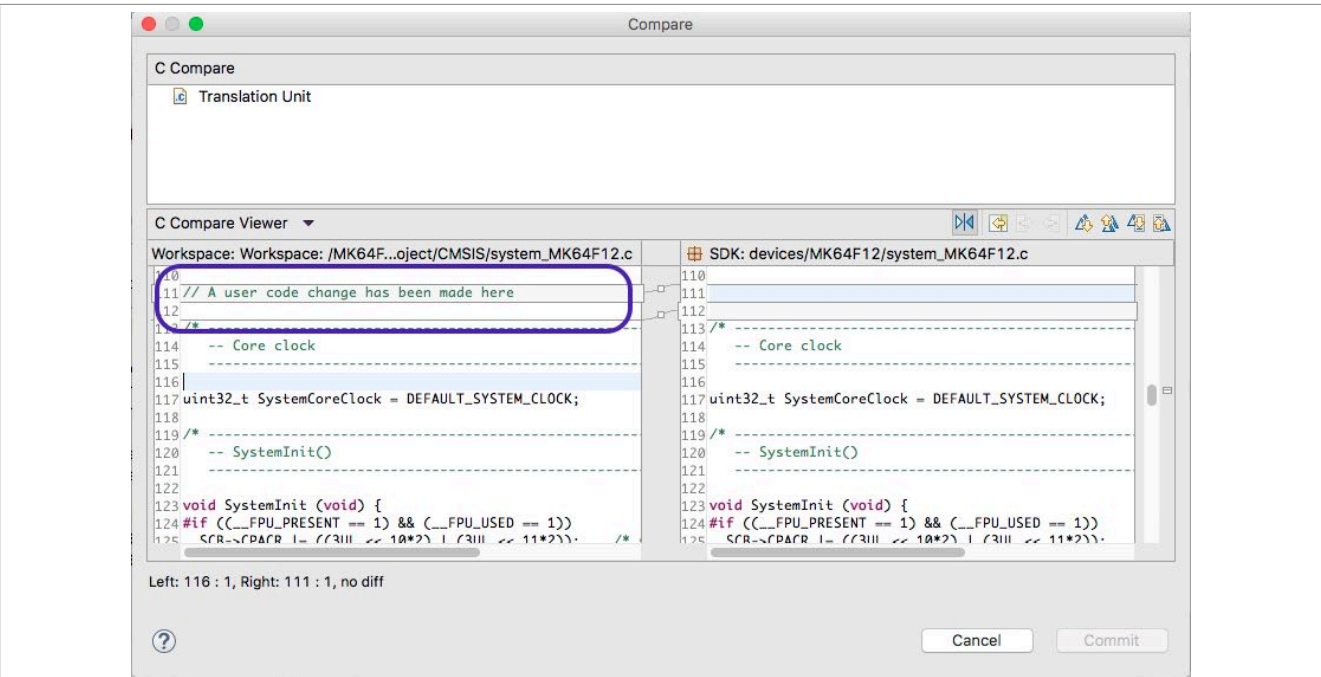
Figure 87. SDK Component Management file compare

The Compare utility allows you to examine any change and to decide regarding which code lines to choose or ignore. When you have decided, click 'Commit' to use these changes or 'Cancel' to return to the dialog and decide the action to take for the file.

Finally note the application build sizes before the addition:

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

```
        FLEX_RAM:          0 GB        4 KB      0.00%
Finished building target: MK64FN1M0xxx12_Project.axf
```

Followed by the application sizes after the addition.

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

These are the same!

This is because although new source files have been added to the project, there is (probably) no code in the project that references them, and therefore the IDE does not include any new functions or data in the final image. To use any new component, some of its new functionality must be referenced.

Note: Some middleware components, such as USB, are not compatible with the Add/Remove component functionality and so do not appear in the Add/Remove dialog. The recommended approach if such components are required is to import an example including the component and modify it as required. We will address this restriction in a future release.

Also see Image information for details on how to explore the composition of an image in detail.

8.2 SDK project refresh

Using the above technology, you can refresh MCUXpresso IDE projects with updated SDK components.

When new SDKs are released for a particular MCU/Board, many source files are updated, bugs fixed, features added, and so on. If such a new SDK replaces an existing SDK within MCUXpresso IDE, you can optionally add any updated (or changed) source files, configuration files or source file sections to an existing project using an identical mechanism as described above.

To use this feature, simply select a project in the project explorer view and click Refresh SDK Components as indicated below.
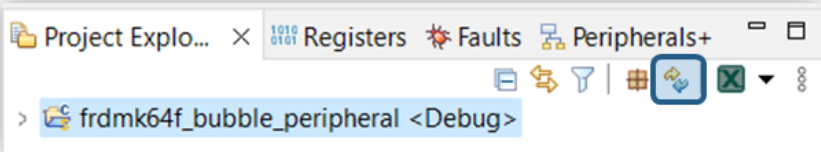
Figure 88. Refresh SDK Components toolbar action

The SDK Component Management wizard guides you through the update process. The SDK Component Management window, lists all the files that will be added or updated.
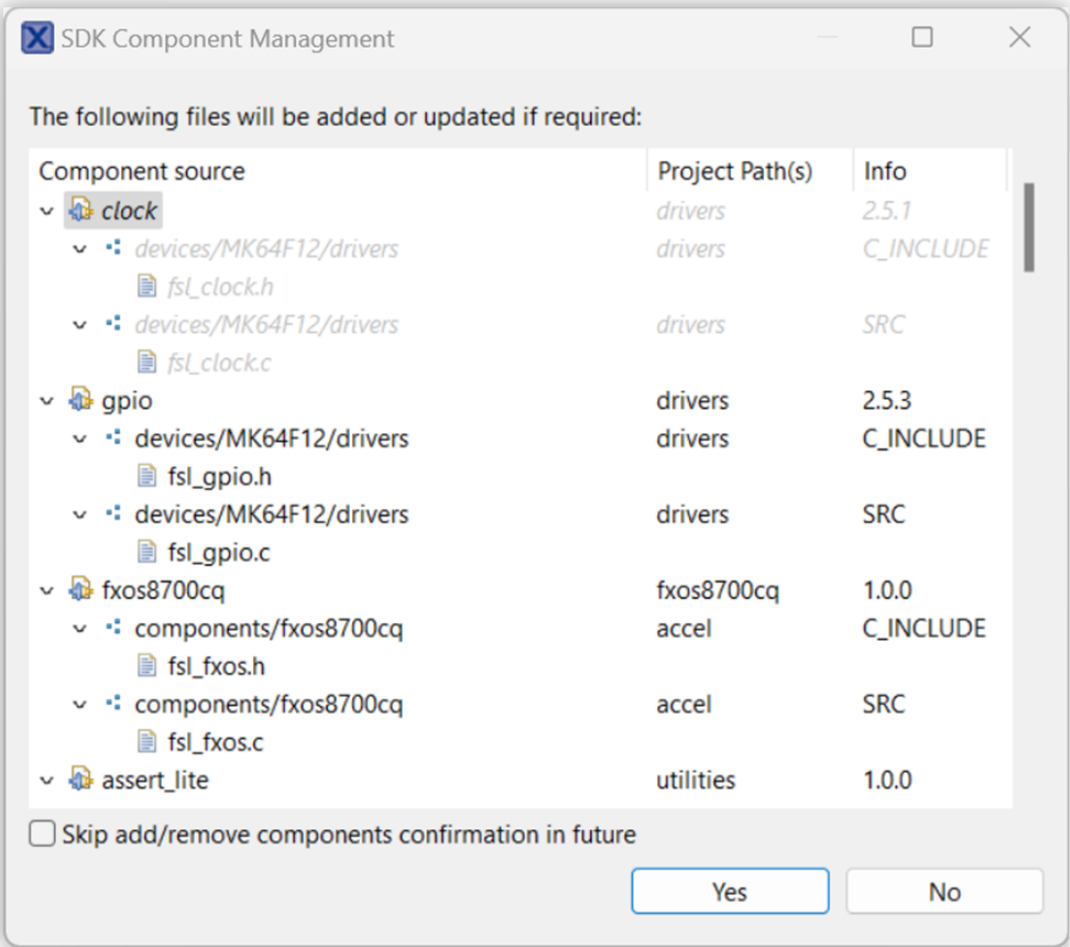
Figure 89. SDK Component Management project refresh

During the project update process, you might be asked to take actions for files that exist in the project but that are different compared to the ones available in the SDK. In this case, MCUXpresso IDE differentiates between regular project files and dedicated configuration files that are used by individual components. Components are usually delivered with template configuration files but the example projects available in the SDK might have a customized configuration file for particular components. If this is the case, you should keep the file already available in the project instead of adding the new template configuration file.
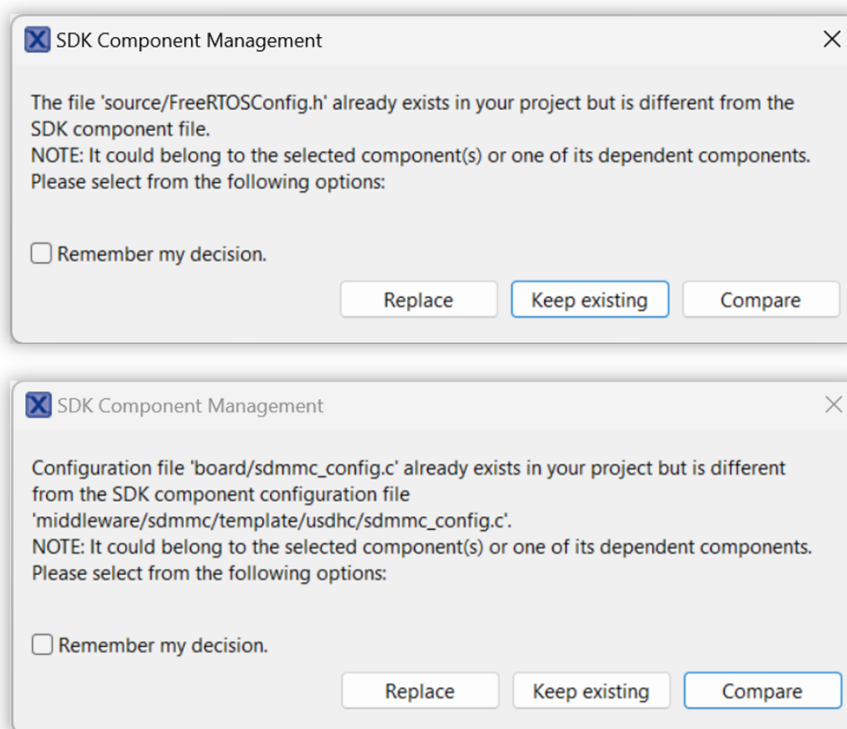
Figure 90. SDK Component Management for existing files

9 Open-CMSIS component management

MCUXpresso IDE integrated ARM CMSIS Plugins to explore Open-CMSIS packs and import (middleware) components into an Eclipse project. With this feature, you can install the desired CMSIS-Pack via CMSIS-Pack Management (*Perspective -> Open Perspective -> Other -> CMSIS-Pack Manager*) and you can add middleware components to the project using RTE Configuration view.

9.1 Install a pack

To start adding components, first open CMSIS-Pack Manager and install the needed packs.
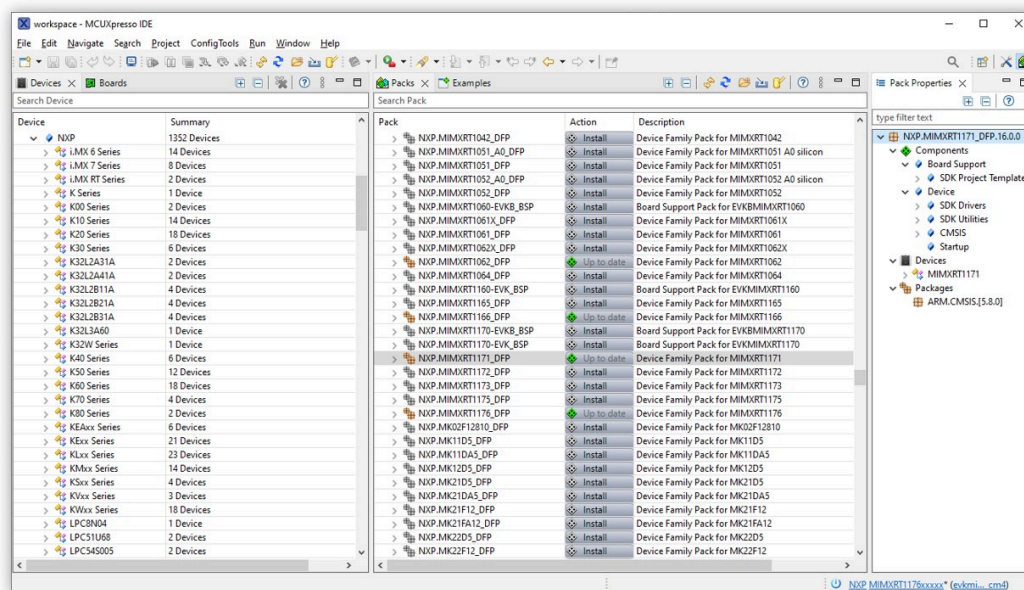
Figure 91. CMSIS-Pack Manager

From the Packs view (toolbar) you can: Reload, Check for updates on Web, Import Packs from disk, and so on. To install a pack, select one, click **Install**, and accept the license. After installation, its status will be **Up to date** and a green icon will appear.

Note: When using the CMSIS-Pack Manager for the first time, an index update occurs to populate the list of available packs on the web.

Once the packs are installed, go back to **Develop** Perspective.

9.2 Add an Open-CMSIS-Pack component to a project

Create an NPW/SDK example for your device and add the support for Open-CMSIS by right-clicking the project entry in *Project Explorer -> SDK Management -> Add Open-CMSIS Components*.
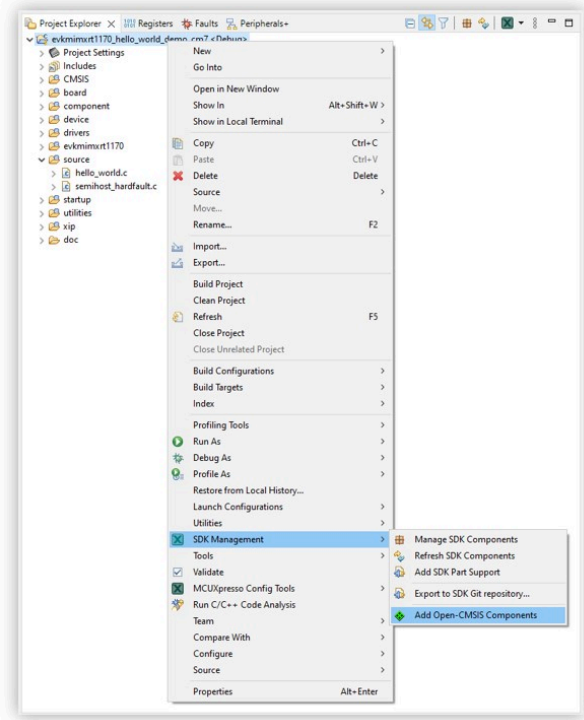
Figure 92. Add Open-CMSIS Components

The RTE Configuration view opens and displays all the available components from installed packs.
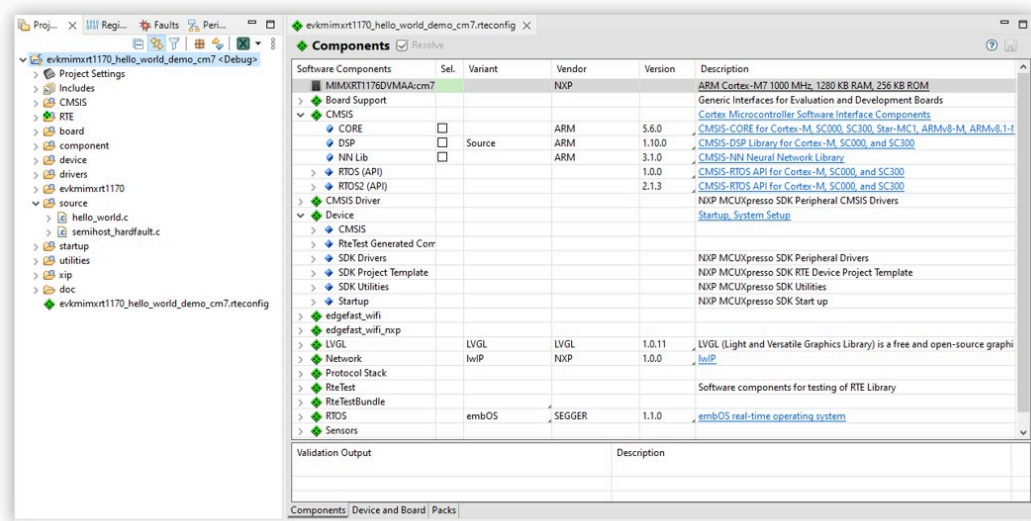
Figure 93. RTE Configuration view

9.3 Manage components inside the project

This feature:

- Allows installation of multiple components, automatically checking and resolving dependencies.
- Automatically generates the required configuration, template, and header files.

• Updates project configuration (for example, build settings, compiler flags, include paths, linked libs, and so on).

10 Creating new projects using preinstalled part support

For Creating projects using SDKs, see Creating new projects using installed SDK part support.

To explore the range of preinstalled parts/MCUs simply click 'Create a new C/C++ project' in the **Quickstart** panel. This opens a page similar to the figure below:
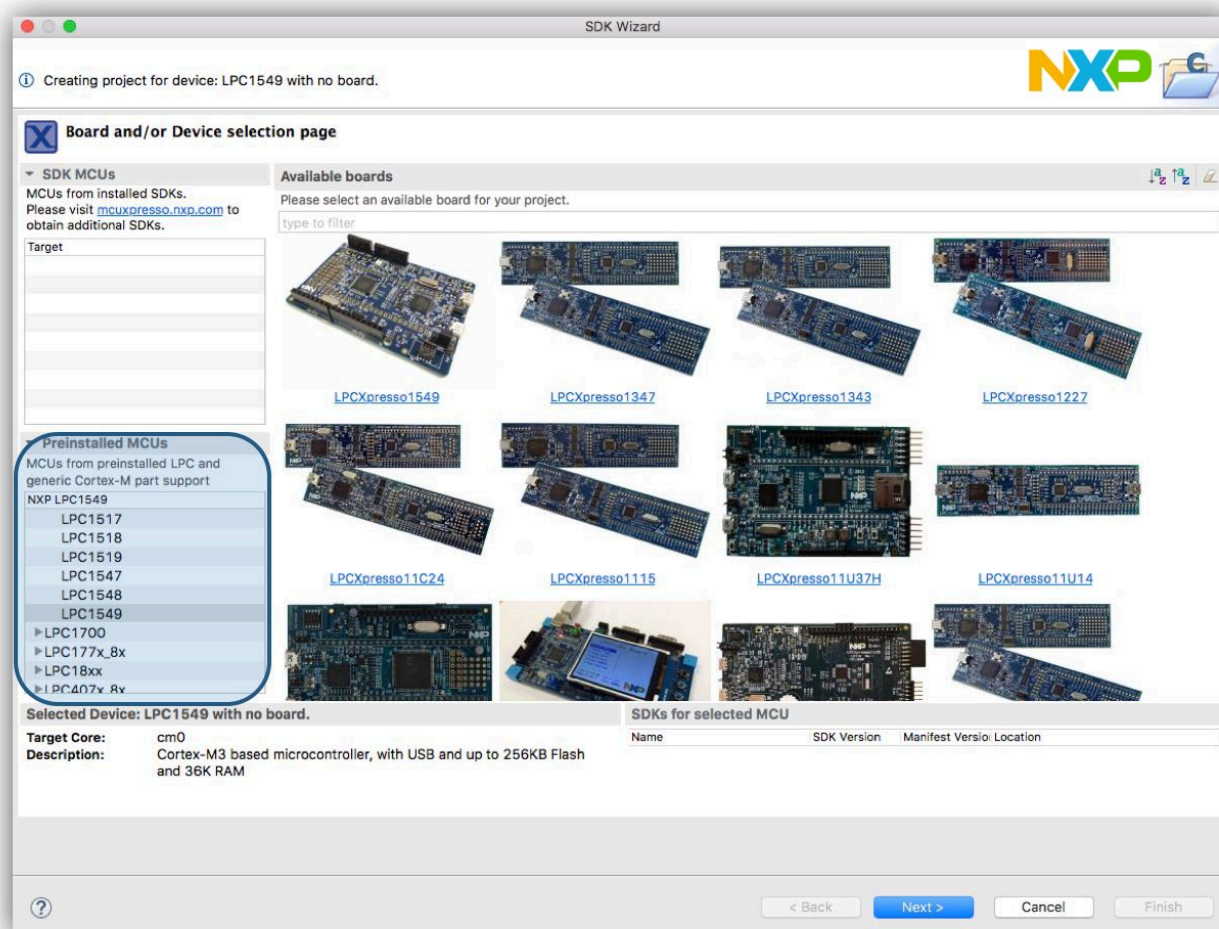
Figure 94. New Project Wizard preinstalled

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

You can also see a range of related development boards indicating whether a matching board support library (LPCOpen or CodeBundles) is available.

For details of this page see: New Project Wizard details

10.1 New Project Wizard

This wizard page provides several 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 library is available, so we can locate this MCU using the board filter. **Note**: Boards are displayed where either LPCOpen or CodeBundle projects exist.

Note: A description of LPCOpen can be found in the section LPCOpen software drivers and examples

To reduce the number of boards displayed, we can simply type '4337' into the filter so only boards with MCUs containing '4337' are displayed.
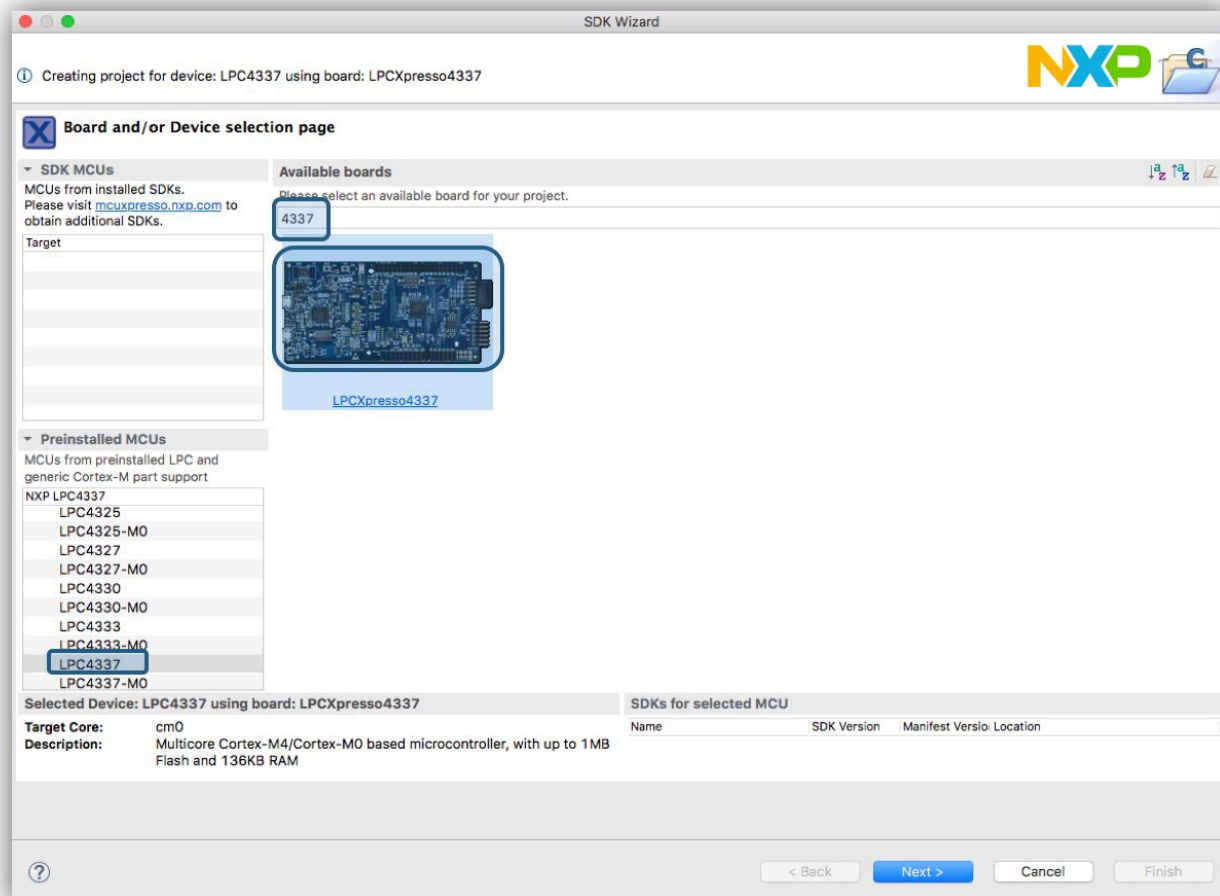
Figure 95. New Project Wizard selection for Preinstalled MCUs

When you select a board as highlighted in the above figure, the wizard also selects automatically the matching MCU (part).

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

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

With a chosen board selected, now click 'Next' to launch the next level of wizards. These wizards for Preinstalled MCUs are similar to those featured in LPCXpresso IDE and are described in the next section.

10.2 Creating a project

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 cores internally. For these MCUs, multicore options are also available (as below):
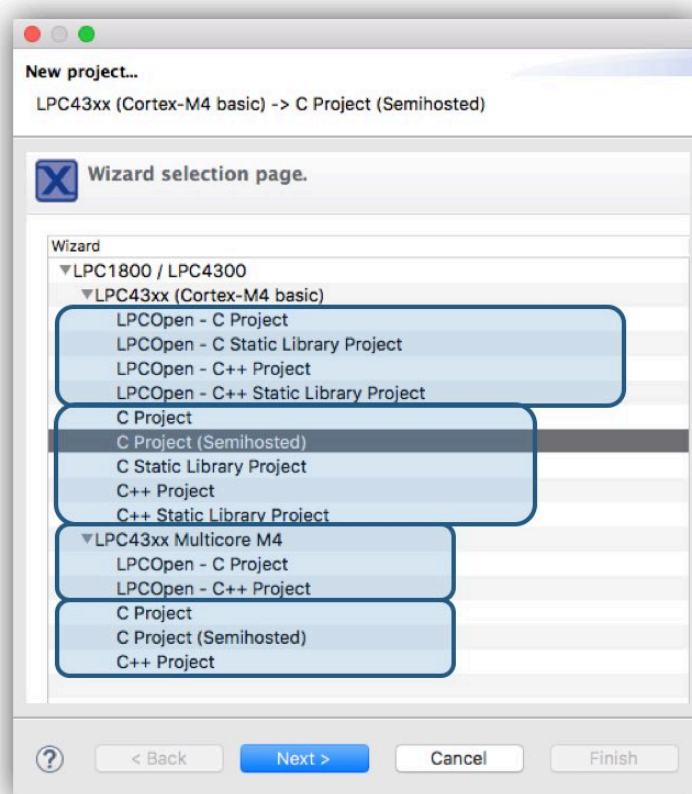
Figure 96. New project: wizard selection

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

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

10.2.1 Selecting the wizard type

For most MCU families MCUXpresso IDE provides wizards for two forms of project: LPCOpen and non-LPCOpen. For more details on LPCOpen, see Software drivers and examples 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 is also included to initialize the board and enable an 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 is also included to initialize the board and enable an 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 also contains a "liblinks.xml" file, which the smart update wizard can use on the context-sensitive menu to create links from application projects to this library project. For more details, 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 causes the text to display within the Console View of MCUXpresso IDE. This is implemented using "semihosting" functionality. See the section on Semihosting for more information.

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

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

10.2.3 Wizard options

The wizard presents a set of pages (that vary based on the chosen MCU), many of these pages typically require no user change since the common default values are already preset. The pages may include:

10.2.3.1 LPCOpen library project selection

When creating an LPCOpen-based project, the first option page that you see is the LPCOpen library selection page.
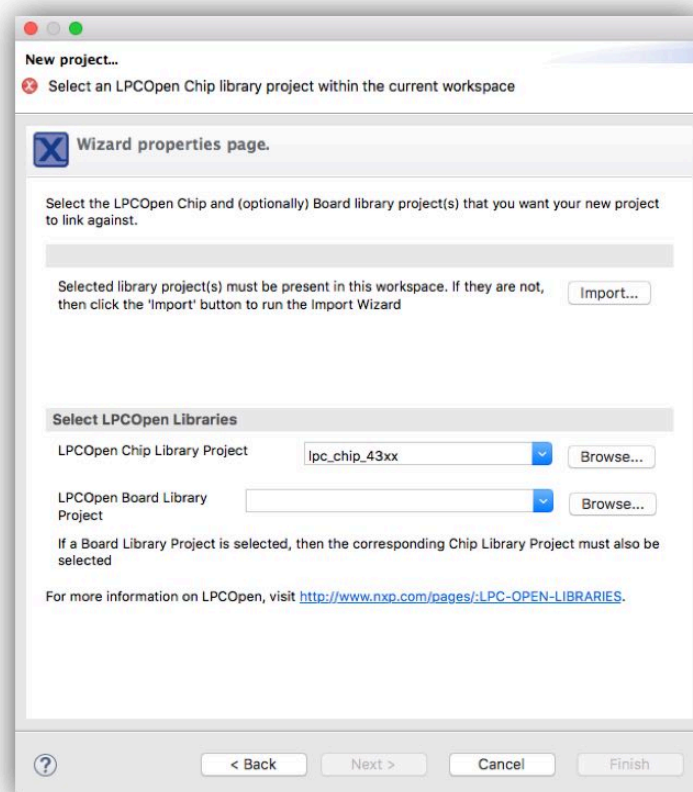
Figure 97. LPCOpen library selection

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

You then must select the LPCOpen Chip library for your MCU using the workspace browser (and for some MCUs an appropriate value is also available from the dropdown next to the Browse button). **Note**: the wizard does 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, sometimes an appropriate value may also be available from the dropdown next to the Browse button). Although the selection of a board library is optional, it is recommended that you do this usually.

10.2.3.2 CMSIS-CORE selection

For backward 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, and code for initializing an MCU and accessing various aspects of the functionality of the Cortex CPU itself. MCUXpresso IDE typically provides support for CMSIS through the provision of CMSIS library projects. You can find CMSIS-CORE library projects 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). MCUXpresso IDE does sometimes 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 MCUXpresso IDE allows you to select which (if any) CMSIS-CORE library you want to link to from the project you are creating. **Note**: you have to import the appropriate CMSIS-CORE library project into the workspace before the wizard allows you to continue.

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

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

Note: The use of LPCOpen instead of CMSIS-CORE library projects is usually recommended for new projects. (LPCOpen actually builds on top of many aspects of CMSIS-CORE.) For more details, see Software drivers and examples.

10.2.3.3 CMSIS DSP library selection

ARM's **Cortex Microcontroller Software Interface Standard** (or **CMSIS**) specification also provides a definition and implementation of a DSP library. 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: You can use the CMSIS DSP library with both LPCOpen and non-LPCOpen projects.

10.2.3.4 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 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 for new projects.

10.2.3.5 Enable the 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 sets appropriate build options so that code is built to use the hardware floating-point unit and also causes startup code to enable the unit to be included.

10.2.3.6 Code Read Protect

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

10.2.3.7 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

10.2.3.8 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, see the FAQ at:

https://community.nxp.com/message/630654

10.2.3.9 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, see NXP's application note, AN11015, "Adding ISP to LPC1102 systems".

10.2.3.10 Memory configuration editor

For certain MCUs such as the LPC18xx and LPC43xx, the wizard presents the option to edit the target memory configuration. This is because these parts may use external SPIFI Flash memory and therefore this can be described here if required. For more information, see: LinkServer Flash support and also Memory configuration and linker scripts

Note: You can also edit the memory configuration post-project creation.

10.2.3.11 Redlib printf options

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

• Use the 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 helps 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()` use `malloc()` to provide a temporary buffer on the heap to generate the string to be displayed. Enable this option to switch to using "character-by-character" versions of these functions (which do not require heap space). This can be useful, for example, if you are retargeting printf() to write out over a UART - since in this case, it is pointless creating a temporary buffer to store the whole string, only to print it out over the UART one character at a time.
  – For MCUs where the wizard does not provide this option, you can cause the same effect by adding the symbol `CR_PRINTF_CHAR` to the project properties.

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

For more information, see C/C++ library support.

10.2.4 Project created

Having selected the appropriate options, you can then click the Finish button, and the wizard creates your project for you, together with the appropriate startup code and a simple `main.c` file. Build options for the project are 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 11.5 and Section 14.

11 Importing example projects (from the file system)

MCUXpresso IDE supports two schemes for importing examples:

• From SDKs - using the **Quickstart** Panel -> Import SDK example(s). See Importing examples projects (from SDK)
• From the filing system - using the **Quickstart** Panel -> Import project(s) from file system
  – We discuss this option below:

Tip: *You can import MCUXpresso IDE project(s) directly into a workspace by simply dragging a folder (or zip) containing MCUXpresso IDE projects onto the Project Explorer view.* **Note**: *this imports all projects within a folder (or zip). You can also export projects by dragging directly from the Project Explorer view onto a filer, or directly into another instance of the IDE. See* Enhanced project sharing features *for more information. Due to underlying Eclipse changes in Version 11.1.0, you can only use Drag and Drop to import projects, when one or more projects exists within a workspace.*

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

MCUXpresso IDE installs with many example projects for preinstalled parts, that you can import directly into a workspace. These are located at:

```
<install_dir>/ide/Examples
```

and consist of:

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

11.1 Code bundles for LPC800 family devices

The LPC800 Family of MCUs is ideal for customers who want to make the transition from 8-bit 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 available in both README and source file comments. Code Bundles for LPC800 family devices are made available at the time of the product launch of the series, ready for use with a range of tools including MCUXpresso IDE.

Find more information on code bundles together with the latest downloads at:

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

11.2 LPCOpen software drivers and examples

Note: LPCOpen is no longer under active development. SDKs now provide support for new MCUs from NXP. Certain parts such as some members of the LPC54xxx families are available with both LPCOpen and SDK support.

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 MCUXpresso IDE

CMSIS/Peripheral Driver Library/code bundle software packages are still available, from within your *install_dir/ ide/Examples/Legacy* folder. However, you should only use these for existing development work. When starting a new evaluation or product development, we would recommend the use of LPCOpen if available.

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

https://www.nxp.com/lpcopen

11.3 Importing an example project

To import an example project from the file system, locate the **Quickstart** panel and select 'Import projects from Filesystem'
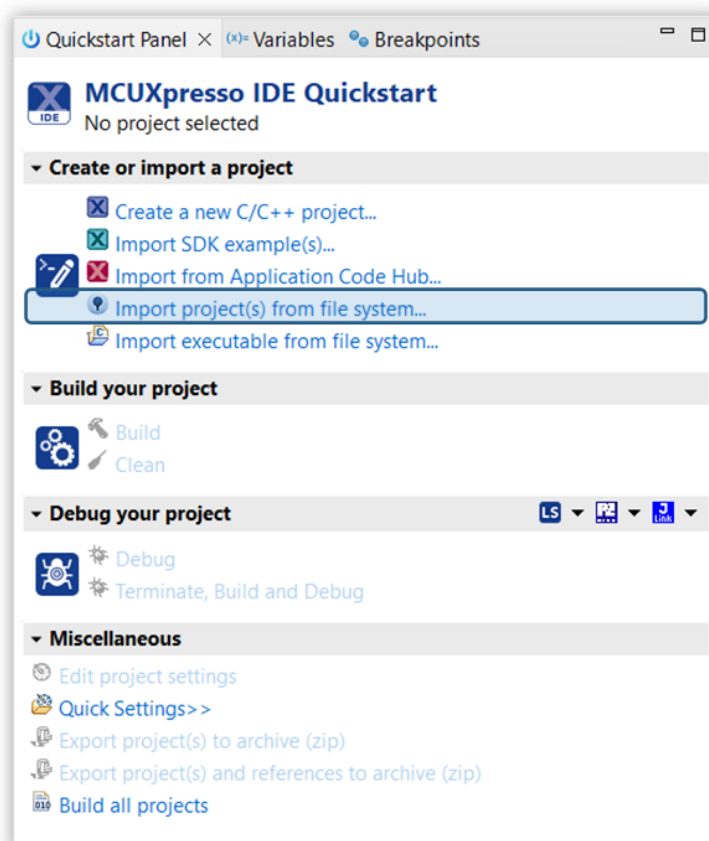
Figure 98. Importing project(s)

From here, you can browse the file system.
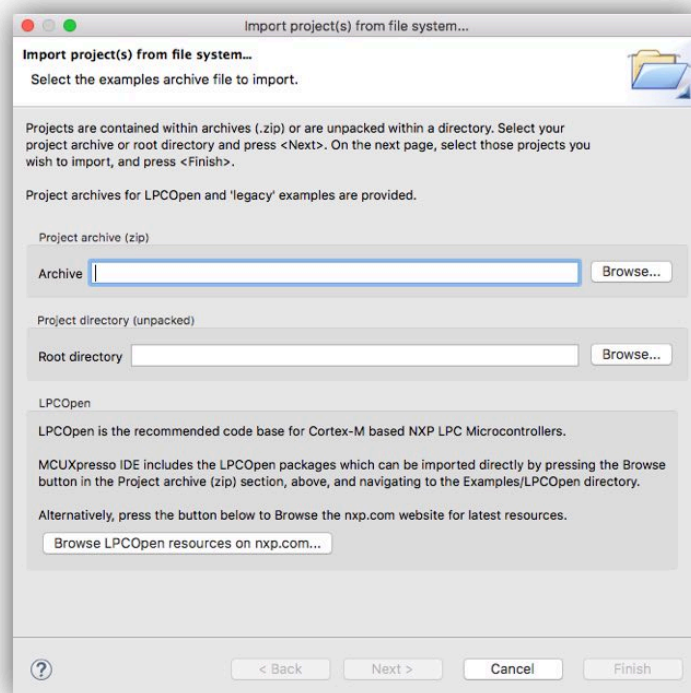
Figure 99. Importing examples

- **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 https://www.nxp.com/lpcopen or the supplied, but deprecated, sample code located within the Examples/Legacy subdirectory of your MCUXpresso IDE installation).
- **Browse** to locate projects stored in directory form on your local system (for example, you can use this to import projects from a different workspace into the current workspace).
- **Browse LPCOpen resources** to visit https://www.nxp.com/lpcopen and download an appropriate LPCOpen package for your target MCU. This option automatically opens a web browser onto a suitable links page.

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

11.3.1 Importing examples for the LPCXpresso4337 development board

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

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

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

Once the package has finished downloading, return to the Import Project(s) dialog and click 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**. Then you see a list of projects within the archive, as shown in Figure 100.
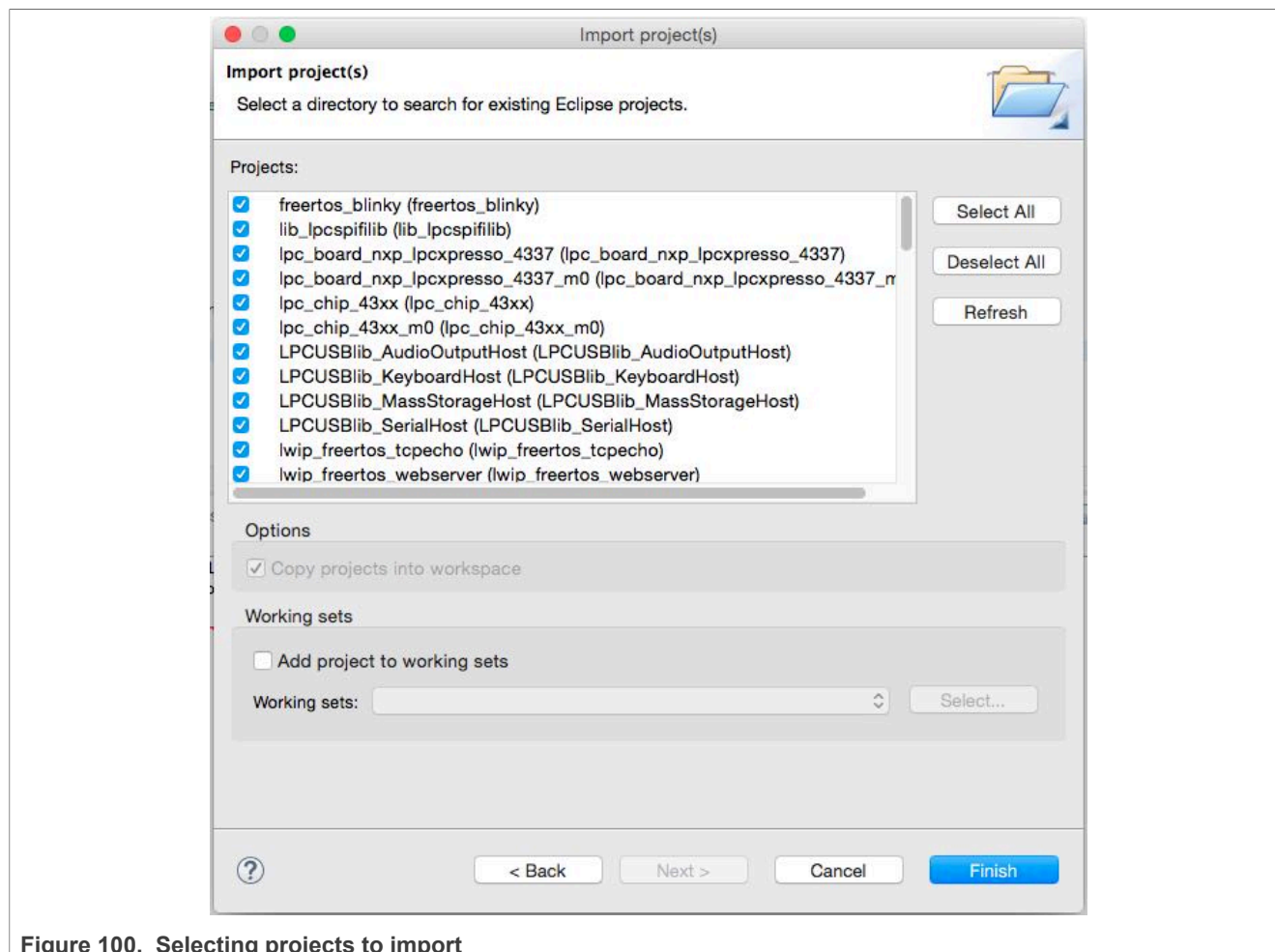
Figure 100. Selecting projects to import

Select the projects that you want to import and then click **Finish**. The examples are 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 true the first time that you import an example set, when you are not necessarily aware of any dependencies between projects. Usually, an archive of projects contains 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 fail to build.

11.4 Exporting projects

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

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

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

Also see Enhanced project sharing features for information about dragging and dropping projects.

11.5 Building projects

Building the projects in a workspace is a simple case of using the **Quickstart** Panel to "Build all projects". Alternatively, you can select a single project in the 'Project Explorer' View and build it. **Note**: building a single project may also trigger a build of any associated or referenced project.

11.5.1 Build configurations

By default, MCUXpresso IDE creates each project with two different "build configurations": **Debug** and **Release**. Each build configuration contains a distinct set of build options. Therefore, a **Debug** build typically compiles its code with optimizations disabled (@-O0@) and **Release** compiles its code optimizing for minimum code size (@-Os@). You can see the currently selected build configuration for a project after its name in the Build/Clean/Debug options of the **Quickstart** Panel.

For more information on switching between build configurations, see How do I switch between Debug and Release builds.

12 Importing existing executables

You can also import existing executables and further use them for debugging with MCUXpresso IDE. Importing an existing executable generates a new project that you can use to attach or download the executable to the target; when using debug, the code is first downloaded to Flash and then debugging starts, while attach is used to debug an application that was already flashed (see Connecting to a running target (attach) for more details). It is important to note that you cannot use the generated project for rebuilding the executable. The newly created project contains a symbolic link to the imported executable (the executable is not copied or moved from its original location).

To import an existing executable, go to the **Quickstart** panel and select **Import executable from file system**.
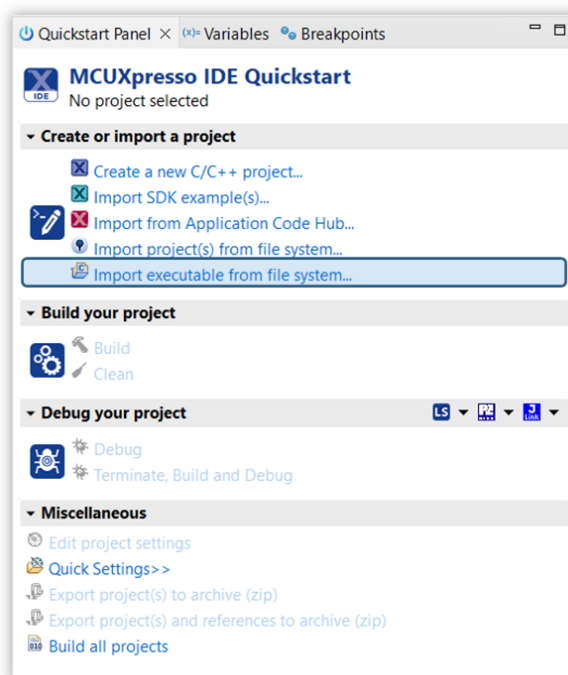
Figure 101. Import executable

You can also access the "MCUXpresso Executable Importer" by going to "File" -> "Import" and then expanding the "C/C++" category.

The wizard allows the selection of files having "elf"/"axf" extension.
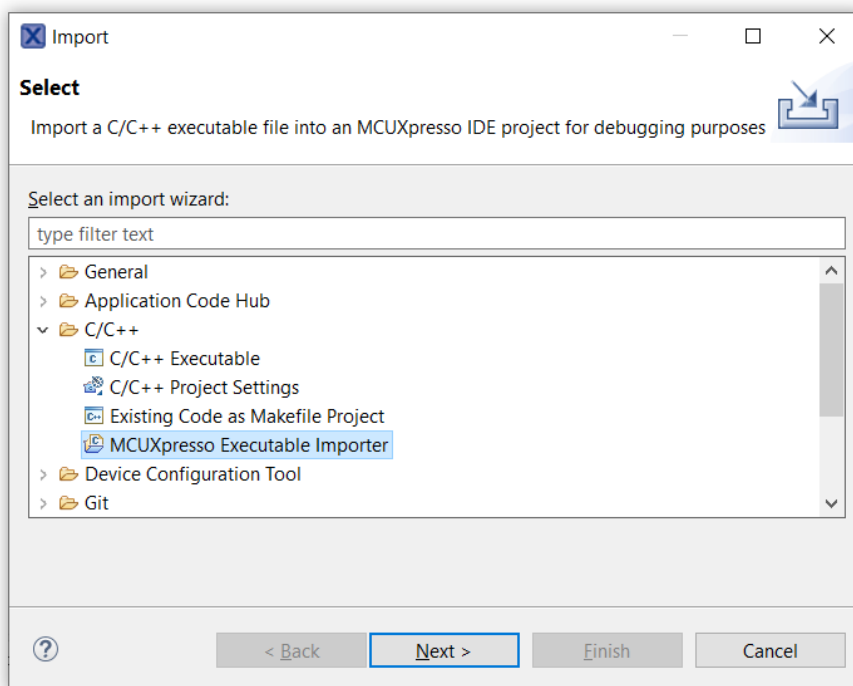
Figure 102. Import wizard

Once opened, several pieces of information have to be provided to finish the wizard. The fields are described below.
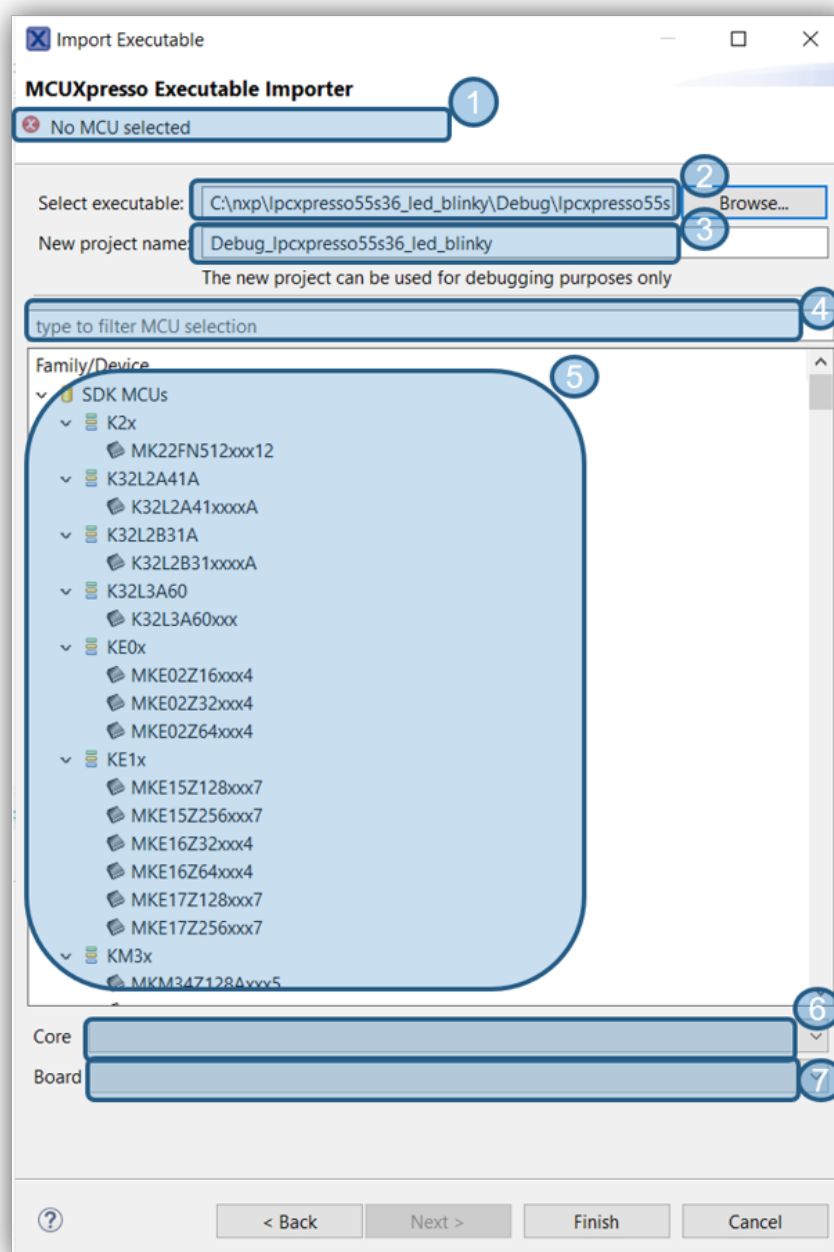
Figure 103. MCUXpresso Executable Importer

1. Wizard validation status. Validation errors appear here.
2. Path to the C/C++ executable file.
3. Name to be assigned to the newly created project.
4. Text filter used to filter the available MCUs.
5. The IDE needs to associate an MCU with the newly created project. Select one from the list. The IDE identifies MCUs from the list based on the installed SDKs and on the available preinstalled parts.
6. In the case of a multicore device, you must also select the core to use for debugging. You can do this using the "Core" drop-down.
7. The "Board" drop-down allows the selection of the actual board.

Once the wizard is finished, a new project appears in Project Explorer, as illustrated in the picture below. See Debugging a project for details about how to start using the project for debugging.
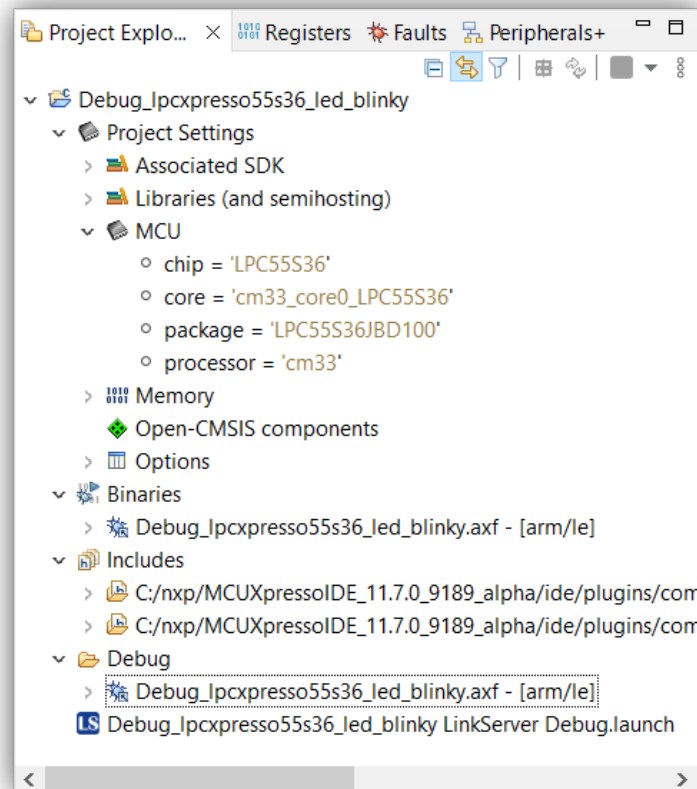
Figure 104. MCUXpresso Executable Importer project

If the sources that were used to build the executable are still available in the original build folder, then they are accessible for source-level debugging. However, since this is not usually the case, you need to specify the location of the sources to access them from your project.
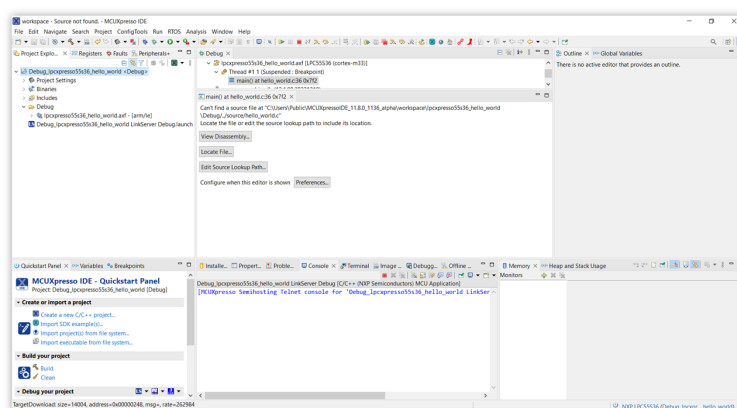
Figure 105. Project Source Look Up

At this point, you can press "Locate File..." and add the source file location. Another option for source mapping is to press "Edit Source Lookup Path..." and then press the "Add" button, finally you should add a container to the source lookup path. You can later edit or remove this entry as needed.
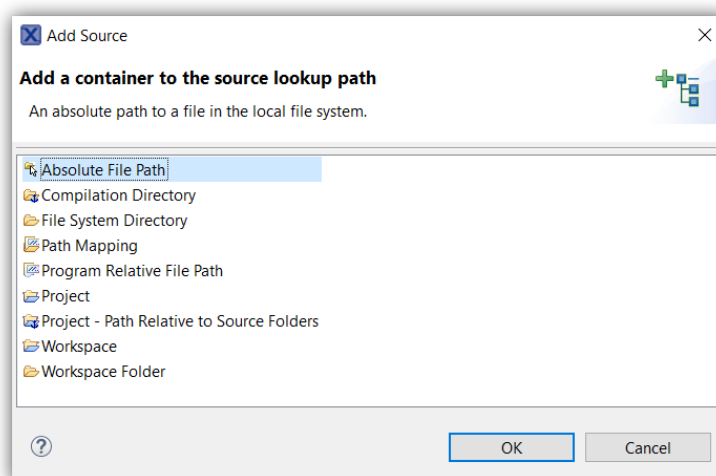
Figure 106. Add Source wizard

13   Debug solutions overview

MCUXpresso IDE installs with built-in support for 3 debug (hardware) solutions; comprising the Native LinkServer (including CMSIS-DAP) as used in LPCXpresso IDE. Plus support for both PEmicro and SEGGER J-Link. 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, see the chapter Debugging a project.

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.

13.1 Starting a debug session

With a suitable board and debug probe connected (usually via USB), to start a debug session:

1. Select a project to debug within the MCUXpresso IDE Project View
2. Click **Debug** from within the MCUXpresso IDE **Quickstart** View

•
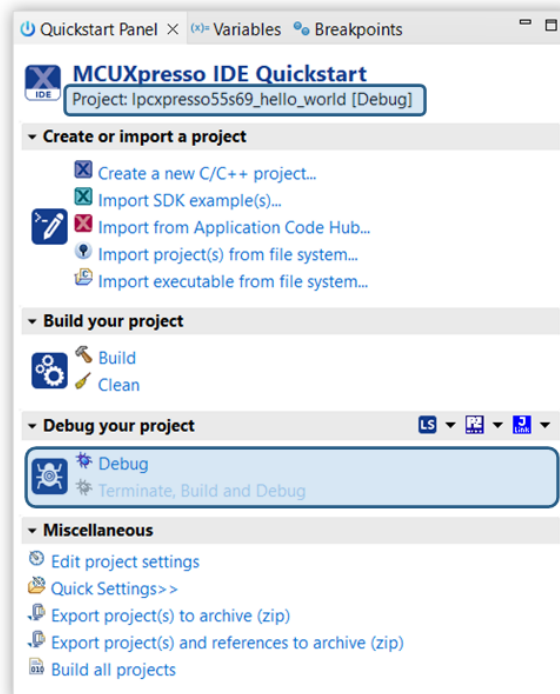
Figure 107. Quickstart Panel - Debug

- A *debug probe discovery* operation is automatically performed to display the available debug connections (that is, the detected debug probes), including LinkServer, PEmicro, and J-Link compatible probes

3. Select the required debug probe and click **OK**
   - At this stage, an automatic creation of a launch configuration takes place within the project, complete with debug-specific configurations
   - If the debug connection is successful, a Debug view appears typically showing the project has stopped on main()

•
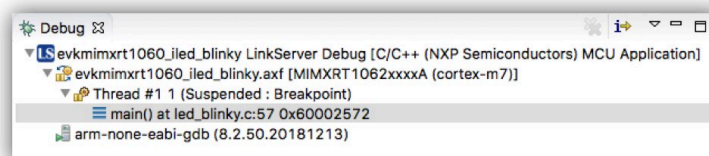
Figure 108. Debug View

Tip: *After debugging a project, the launch configuration contains details of the debug probe used. Subsequent debug sessions automatically select this probe if it is available.*

From this point onwards, one of the debug solutions mentioned above controls the low-level debug operations.

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

- Automatic inheritance of part knowledge
- Automatic downloading (programming) of generated image to target Flash memory
  – LinkServer/CMSIS-DAP Flash programming - see the chapter Introduction to LinkServer Flash drivers

- Automatic halt on main()
- Setting breakpoints and watchpoints
- Stepping (single, step in, step out, and so on)
- Viewing and editing local variablesregistersperipheralsmemory
- Viewing and editing global variables
- Live global variables
- Viewing disassembly
- Semihosted IO
- All debug solutions support Instruction Trace see the Instruction Trace Guide for more information
- GUI Flash Tool
- All debug solutions support SWO Trace, including profiling, interrupt trace, and so on see the SWO Trace Guide for more information
- Viewing details of execution faults via the Faults view (automatically displayed for faults generated during LinkServer debug, a pause is required for other debug solutions)

Additional documentation is also available covering:

- Power/Energy Measurement - see Energy Measurement Guide
- FreeRTOS Debug - see FreeRTOS Debug Guide
- Azure RTOS ThreadX Debug - see Azure RTOS ThreadX Debug Guide
- Zephyr RTOS Debug - see Zephyr RTOS Debug Guide
- MQX RTOS Debug - see MQX RTOS Debug Guide

Note: In addition, MCUXpresso IDE dynamically manages each debug solutions connection requirements allowing multiple sessions to be started without conflict. For debug of multicore MCUs, refer to the section Debugging multicore projects.

It is important to note that certain operations such as the handling of features via Launch configurations may be different for each debug solution. Furthermore, advanced debug features and capabilities may vary between solutions and even similar features may appear different within the IDE.

PEmicro and SEGGER debug solutions also provide several advanced features. Find details at their respective web sites.

13.2 An introduction to launch configuration files

Each project in MCUXpresso IDE stores its debug properties locally in **.launch** files (known as Launch Configuration files).

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

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

When the user first tries to debug a project, MCUXpresso IDE performs a **Debug Probe Discovery** operation and present the user with a list of debug probes found. **Note**: You can filter the debug solutions searched from this dialog as highlighted, removing options that are not required speeds up this process.
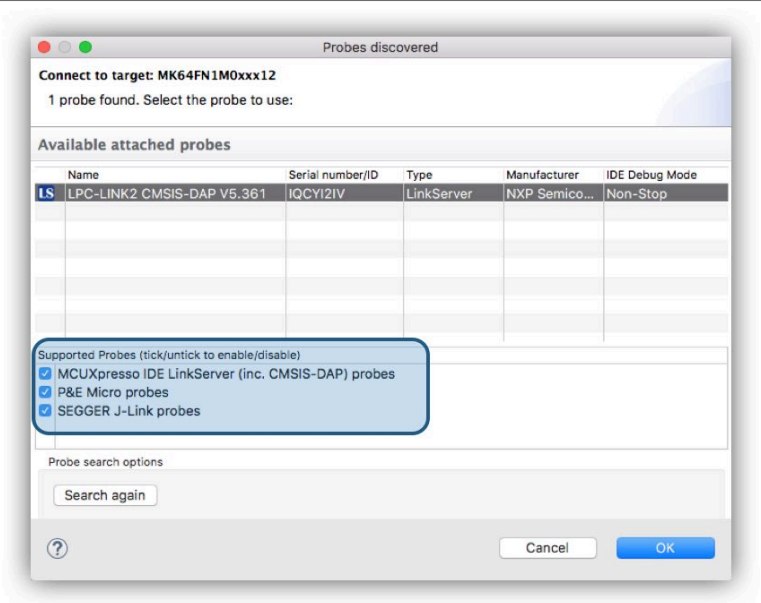
Figure 109. Debug probe discovery

Once the user has selected the debug probe and has clicked 'OK', the IDE automatically creates a default launch configuration file for that debug probe (LinkServer launch configuration shown below).
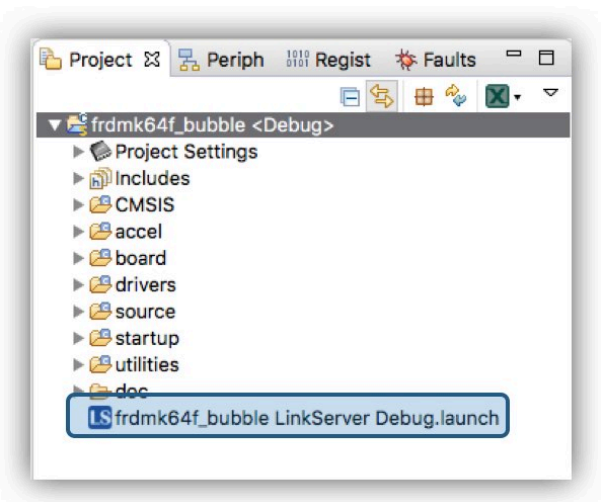
Figure 110. Launch configuration files

Note: The IDE creates a launch configuration only for the currently selected build configuration.

For many debug operations, these files do not require any attention and can essentially be ignored. However, if changes are required, you should not edit these files manually. You should rather explore their properties within the IDE.

The simplest way to do this is to click to expand the Project within the 'Project Explorer' pane, then simply double-click a launch configuration file to automatically open the launch configuration *Edit Configuration* dialog.

Note: This dialog has a number of internal tabs, the *Debugger* tab (as shown below) contains the Debug main settings. See also the Project GUI Flash Tool.
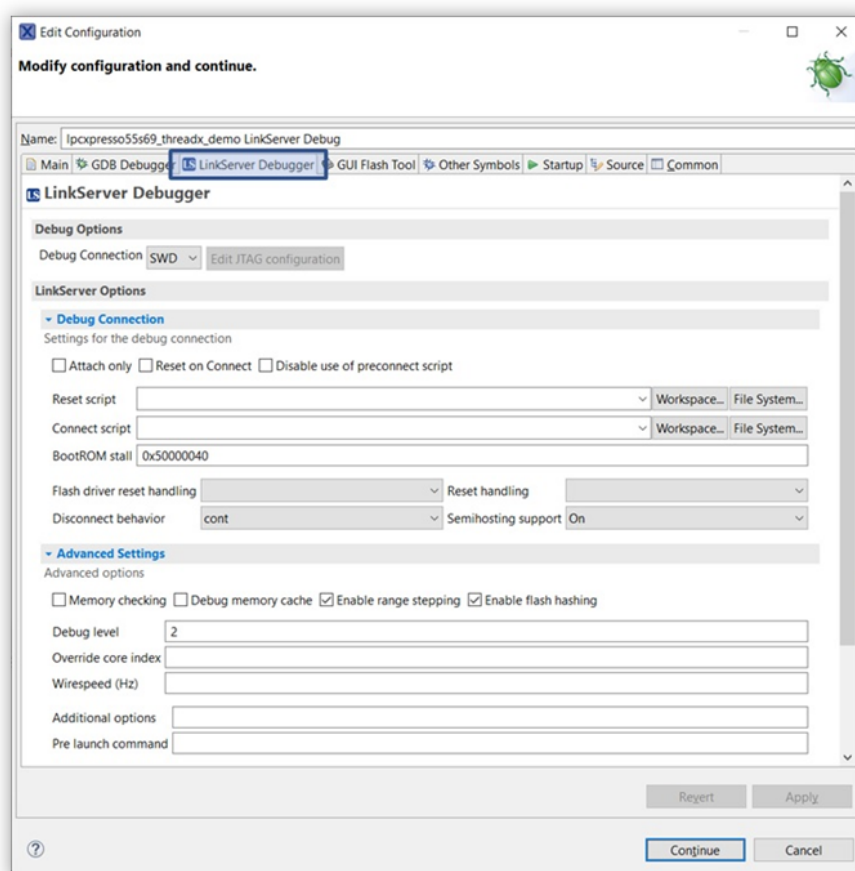
Figure 111. Launch Configuration

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

Note: Once a project has an associated launch configuration, it will always use it for its 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.

Tip: To simplify this operation, you can force a probe discovery by holding the Shift key while launching a debug session from the Quickstart panel. If the new debug connection completes, the IDE creates a new project launch configuration, replacing any existing launch configurations. Alternatively, the Debug shortcuts are available to force the use of a particular debug solution.

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

For further information, see the section Launch configurations.

13.3 LinkServer debug connections

The native debug connection of MCUXpresso IDE (known as LinkServer) is supported via a standalone tool that the MCUXpresso IDE installer installs and configures. You can find more information about NXP's LinkServer solution on the official LinkServer website and inside the documentation page, available after installation - see *mcuxpresso_install_dir/ide/LinkServer/Readme.md*. You can also configure the path to the LinkServer used for

debug (and other) operations by using the LinkServer preferences page LinkServer supports debug operations through the following debug probes:

- MCU-Link and MCU-Link Pro with CMSIS-DAP firmware
- Evaluation boards incorporating MCU-Link with CMSIS-DAP firmware
- LPC-Link2 with CMSIS-DAP firmware
- LPCXpresso V2/V3 Boards incorporating LPC-Link2 with CMSIS-DAP firmware
- CMSIS-DAP firmware installed onto on-board debug probe hardware (as shipped by default on LPCXpresso MAX and CD boards)
  - For more information on LPCXpresso boards, see: https://www.nxp.com/lpcxpresso-boards
  - Additional driver may be required: https://developer.mbed.org/handbook/Windows-serial-configuration
- CMSIS-DAP firmware installed onto on-board OpenSDA debug probe hardware (as shipped by default on certain Kinetis FRDM and TWR boards)
  - Known as DAP-Link and mBed CMSIS-DAP: https://www.nxp.com/opensda
  - Additional driver may be required: https://developer.mbed.org/handbook/Windows-serial-configuration
- Other third-party CMSIS-DAP probes such as Keil uLINK with CMSIS-DAP firmware (https://www2.keil.com/mdk5/ulink) should work with LinkServer but we cannot guarantee it
- Legacy RedProbe+ and LPC-Link
- RDB1768 development board built-in debug connector (RDB-Link)
- RDB4078 development board built-in debug connector

Note: MCUXpresso IDE automatically tries to softload the latest CMSIS-DAP firmware onto LPC-Link2 or LPCXpresso V2/V3 boards. For this to occur, it is necessary to set the DFU link on these boards. Refer to the documentation of the board for details.

13.4 LinkServer debug operation

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

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

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

Non-Stop uses GDB's "non-stop mode" and allows data to be read from the target while an application is running. Currently, this mechanism is used to support the Live global variable and Live heap features.
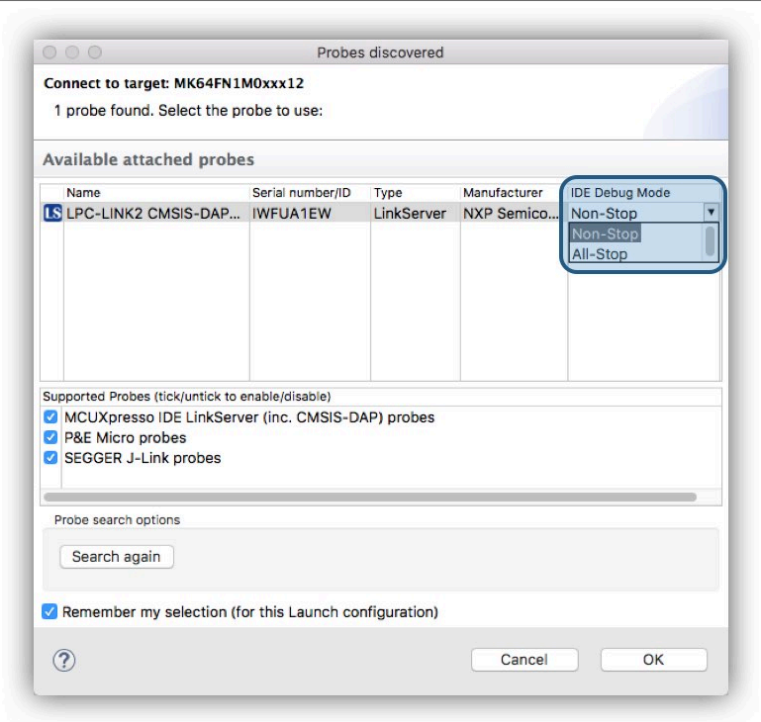
Figure 112. Debug probe discovery non-stop

Click 'OK' to start the debug session. At this point, the launch configuration files for the project are created. LinkServer Launch configuration files contain the string 'LinkServer' and have an *LS* icon.

Note: If you leave "Remember my selection" option ticked, then the launch configuration file stores the probe details, and the IDE selects automatically this probe on subsequent debug operations for this project.

For a description of some common debugging operations using supported debug probes, see Common debugging operations.

MCUXpresso IDE defaults to the selection of "Non-Stop" mode when performing a LinkServer probe discovery operation. You can change this default from an MCUXpresso IDE Preference via:

Preferences -> MCUXpresso IDE -> Debug Options -> LinkServer Options -> Miscellaneous
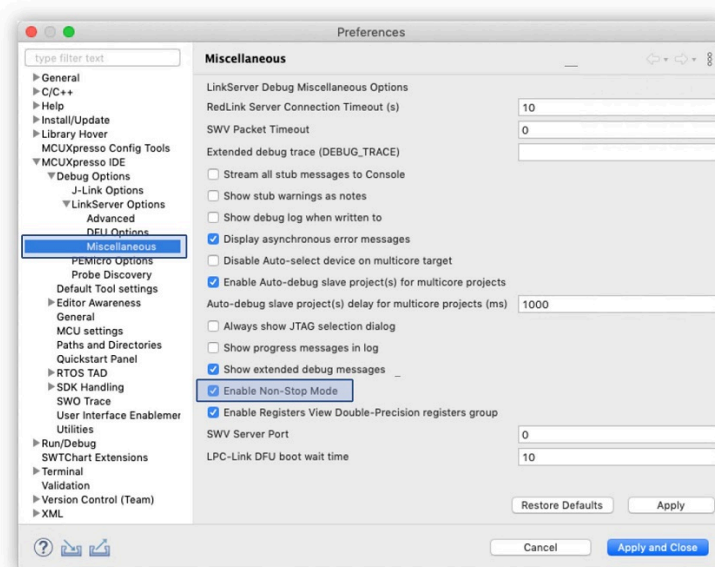
Figure 113. LinkServer non-stop preference

For a given project, its launch configuration stores the Non-Stop mode option. For projects that already have launch configurations, you can change this option from the GDB Debugger tab as shown below.
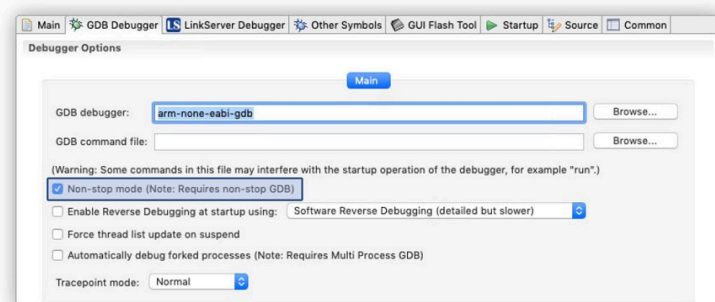
Figure 114. LinkServer non-stop control

13.4.1 LinkServer debug scripts

LinkServer debugging supports a scripting language, which is discussed in the section scripts.

A LinkServer debug connection has 3 potential *callouts* where scripts can be referenced, typically to perform some non-standard behavior.

Connect Script a *Connect Script* overrides the default debug connection behavior. Typically such scripts are used to prepare the debug target (MCU) for a debug operation that may otherwise fail due to some target setting that cannot be guaranteed post reset. A common requirement could be to ensure that RAM is available for Flash Programming operations. If required, a *Connect Script* is referenced within a LinkServer debug *Launch Configuration*.

Reset Script a *Reset Script* overrides the default debug reset behavior. *Reset Scripts* are less commonly required than *Connect Scripts* but can be used to work around issues where a standard Reset may not allow debug operations to survive. If required, a *Reset Script* is referenced within a LinkServer debug *Launch Configuration*.

On rare occasions, it may be useful to add a *Connect or Reset Script* to a project, see Project sharing for more information on how this can be done.

Preconnect Script a *Preconnect Script* is a little different. Such a script (if present) prepares the target MCU for an initial debug connection that may/would otherwise fail. *Preconnect Scripts* are not specified within a launch configuration, rather the IDE automatically invokes them for a given target based on built-in intelligence. However, you can disable their use by a checkbox within the *Launch Configuration* of the project. On rare occasions, it may be useful to add a *preconnect script* to a project - you can do this by placing a file called *LS_preconnect.scp* within the directory of the project.

Note: In most circumstances, such scripts are supplied and referenced (via SDKs) automatically so no user intervention or action is required.

13.5 LinkServer path configuration

MCUXpresso IDE v11.9.0 is the first IDE version that integrates the standalone NXP LinkServer product. The MCUXpresso IDE installer automatically installs it and you can find it in the folder located at the same level as the MCUXpresso IDE product installation. A symbolic link is also created inside *mcuxpresso_install_dir/ide* that points to the actual LinkServer installation linked to IDE, more specifically *mcuxpresso_install_dir/ide/Link Server*.

Note: LinkServer-specific support files from folders like *mcuxpresso_install_dir/ide/binaries* are now part of the LinkServer package and have been moved accordingly. The IDE refers all these files from the LinkServer installation folder.

You can configure the default LinkServer used by the IDE by going to **Window -> Preferences -> MCUXpresso IDE -> Debug Options -> LinkServer Options**. Find the specific section highlighted in the picture below. The default path, pointing to the LinkServer that was installed along with the IDE, is not editable but is listed for awareness. You can also configure a custom path but you must take care to ensure that the IDE is compatible with the configured LinkServer.
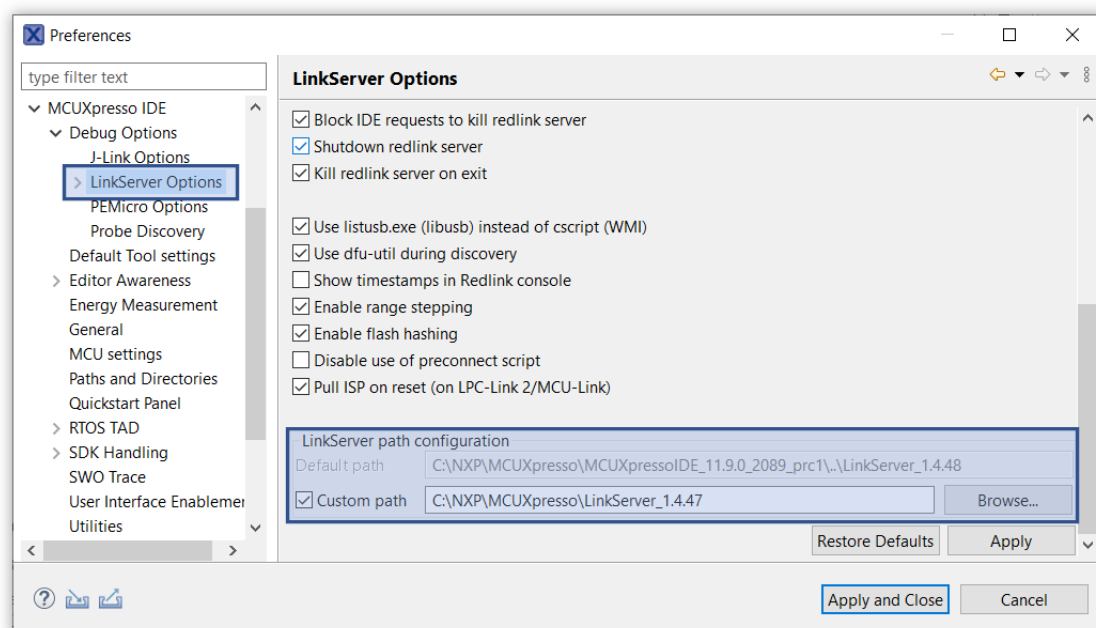
Figure 115. LinkServer path configuration option

Note: If you configure a custom LinkServer, the symbolic link *mcuxpresso_install_dir/ide/LinkServer* still points to the original LinkServer that was installed by the MCUXpresso IDE installer.

Document feedback

13.6 LinkServer troubleshooting

13.6.1 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 console known as the Debug log. This log is displayed when a Debug operation begins, but, by default, is replaced by another view when execution starts. The debug log is a standard log within the Console view of the IDE. To display this log, select the Console and then click to view the various options (as below):
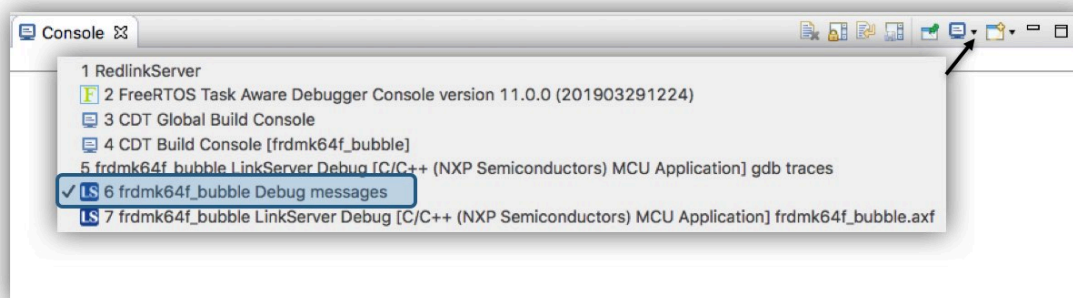
Figure 116. Console view

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

Further down in this log, you can see the selection of a Flash driver (FTFE_4K), the identification of the part being debugged (in this case a K64), the programming progress, and the speed of the Flash programming operation (in this case over 95 kB/s).

Tip: *A line similar to flash variant 'K 64 FTFE Generic 4K' detected (1 MB = 256*4K at 0x0) is displayed for LinkServer Flash programming operations. The size of the detected flash (in this example it is 1 MB) and sector size (4 kB) is displayed here. The sector size may be important since multiples of this size represent valid base addresses for flash programming operations. For example, if the programming of more than one image is required, the second image must begin on a 4 kB boundary beyond the end of any previously programmed image.*

```
MCUXpresso IDE RedlinkMulti Driver v11.1 (Nov 21 2019 14:13:54 -
 crt_emu_cm_redlink build 204)
Found part description in XML file MK64F12_internal.xml
Reconnected to existing LinkServer process.
============= SCRIPT: kinetisconnect.scp =============
Kinetis Connect Script
Connecting to Probe Index = 1
This probe = 1
This TAP = 0
This core = 0
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 V5.361 (NXP Semiconductors)
Serial Number:  IQCYI2IV
VID:PID:  1FC9:0090
USB Path: USB_1fc9_0090_314000_ff00
Using memory from core 0 after searching for a good core
debug interface type     = Cortex-M3/4 (DAP DP ID 2BA01477) over SWD TAP 0
processor type           = Cortex-M4 (CPU ID 00000C24) on DAP AP 0
number of h/w breakpoints = 6
number of flash patches  = 2
number of h/w watchpoints = 4
Probe(0): Connected&Reset. DpID: 2BA01477. CpuID: 00000C24. Info: <None>
Debug protocol: SWD. RTCK: Disabled. Vector catch: Disabled.
Content of CoreSight Debug ROM(s):
RBASE E00FF000: CID B105100D PID 04000BB4C4 ROM (type 0x1)
ROM 1 E000E000: CID B105E00D PID 04000BB00C Gen SCS (type 0x0)
ROM 1 E0001000: CID B105E00D PID 04003BB002 Gen DWT (type 0x0)
ROM 1 E0002000: CID B105E00D PID 04002BB003 Gen FPB (type 0x0)
ROM 1 E0000000: CID B105E00D PID 04003BB001 Gen ITM (type 0x0)
ROM 1 E0040000: CID B105900D PID 04000BB9A1 CSt TPIU type 0x11 Trace Sink - TPIU
ROM 1 E0041000: CID B105900D PID 04000BB925 CSt ETM type 0x13 Trace Source -
 Core
ROM 1 E0042000: CID B105900D PID 04003BB907 CSt ETB type 0x21 Trace Sink - ETB
ROM 1 E0043000: CID B105900D PID 04001BB908 CSt CSTF type 0x12 Trace Link -
 Trace funnel/router
NXP: MK64FN1M0xxx12
DAP stride is 4096 bytes (1024 words)
Inspected v.2 On chip Kinetis Flash memory module FTFE_4K.cfx
Image 'Kinetis SemiGeneric Nov  7 2019 19:12:49'
Opening flash driver FTFE_4K.cfx
Sending VECTRESET to run flash driver
Flash variant 'K 64 FTFE Generic 4K' detected (1MB = 256*4K at 0x0)
Closing flash driver FTFE_4K.cfx
Connected: was_reset=true. was_stopped=true
Awaiting telnet connection to port 3330 ...
GDB nonstop mode enabled
Opening flash driver FTFE_4K.cfx (already resident)
Sending VECTRESET to run flash driver
Flash variant 'K 64 FTFE Generic 4K' detected (1MB = 256*4K at 0x0)
Writing 26880 bytes to address 0x00000000 in Flash
00001000 done  15% (4096 out of 26880)
00002000 done  30% (8192 out of 26880)
00003000 done  45% (12288 out of 26880)
00004000 done  60% (16384 out of 26880)
00005000 done  76% (20480 out of 26880)
00006000 done  91% (24576 out of 26880)
00007000 done 100% (28672 out of 26880)
Sectors written: 7, unchanged: 0, total: 7
Erased/Wrote sector  0-6 with 26880 bytes in 276msec
Closing flash driver FTFE_4K.cfx
Flash Write Done
```

```
Flash Program Summary: 26880 bytes in 0.28 seconds (95.11 KB/sec)
Starting execution using system reset and halt target
Stopped (Was Reset)  [Reset from Unknown]
Stopped: Breakpoint #1
```

13.6.2 Flash programming

Most debug sessions begin with the programming of Flash, followed by a reset of the MCU. **Note**: If flash programming should fail then the debug operation is aborted.

Starting with MCUXpresso IDE version 11.1.0 - most LinkServer flash drivers now implement a *Verify Same* operation (via a flash hashing mechanism) for any flash sector that is unchanged from previous debug operations.

Starting with MCUXpresso IDE version 11.9.0 - LinkServer flash drivers now reside inside the separate LinkServer package.

Below is a fragment of a debug log repeating the previous debug operation. The log reports the Sectors that were unchanged from the previous operation and the resultant overall speed of the flash operation - in this case, the equivalent of a programming speed of 937 kB/s.

```
...
Opening flash driver FTFE_4K.cfx (already resident)
Sending VECTRESET to run flash driver
Flash variant 'K 64 FTFE Generic 4K' detected (1MB = 256*4K at 0x0)
Writing 26880 bytes to address 0x00000000 in Flash
Sectors written: 0, unchanged: 7, total: 7
Erased/Wrote sector  0-6 with 26880 bytes in 28msec
Closing flash driver FTFE_4K.cfx
Flash Write Done
Flash Program Summary: 26880 bytes in 0.03 seconds (937.50 KB/sec)
Starting execution using system reset and halt target
Stopped (Was Reset)  [Reset from Unknown]
Stopped: Breakpoint #1
```

Note in the unlikely event of this feature causing problems, you can disable it from a project LinkServer Launch Configuration by unchecking the *Enable Flash hashing* option. Alternatively, you can disable the feature as a workspace preference via *MCUXpresso IDE -> Debug Options -> LinkServer Options -> Enable flash hashing*.

Below is a brief discussion of the most common-low level flash operations:

1. Sector Erase: internally, Flash devices are divided into a number of sectors (or blocks), where a sector is the smallest size of Flash that can be erased in a single operation. A sector is larger than a page (see below). Sectors are usually the same size for the whole Flash device, however, this is not always the case. The base address of a sector is aligned on a boundary that is a multiple of its size. A sector erase is usually the first step in a flash programming sequence.
2. Page Program: internally, Flash devices are divided into a number of pages, where a page is the smallest size that can be programmed in a single operation. A page is smaller than a sector. A page base address is aligned on a boundary that is a multiple of its size.
3. Mass Erase: a mass erase resets 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).

The programming of an image (or data) comprises repeated operations of sector erase followed by a set of program page operations; until the sector is fully programmed or there is no more data to program.

One of the common problems when programming Kinetis parts relates to their use of the Flash configuration block at offset 0x400. For more information, see: Kinetis MCUs Flash Configuration Block. Flash sector sizes on Kinetis MCUs range from less than 1 kB to 8 kB, therefore the first Sector Erase performed may clear the value

of this block to all 0xFFs, if this is not followed by a successful program operation and the part is reset, then it likely reports as 'Secured' and subsequent debugging is not possible until recovering the part.

Such an event can occur if accidentally performing a debug operation on the 'wrong board', so a wrong Flash programmer is invoked.

Note: LinkServer mass erase operations restore this Flash configuration block automatically for Kinetis parts. However, if a Kinetis device is mass erased by sector, this mechanism is bypassed, therefore you should not perform this operation on Kinetis parts!

Should you need to recover a 'locked' part see the section LinkServer GUI Flash Tool.

13.6.3 LinkServer executables

LinkServer debug operations rely on 3 main debug executables.

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

If a debug operation fails, or a crash occurs, it is possible that one or more of these processes may fail to shut down correctly. Therefore, if the IDE has no active debug connection but is experiencing problems making a new debug connection, ensure that none of these executables is running. To simplify this process, an IDE button allows you to kill all low-level debug executables (for all debug solutions). Therefore, if a debug operation fails or a crash occurs, simply click this button before starting a new debug operation.

13.7 PEmicro debug connections

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

Currently, we have tested using:

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

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

13.8 PEmicro debug operation

The process to debug via a PEmicro compatible debug probe is 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 PEmicro 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 launch configuration files for the project are created. **Note**: PEmicro Launch configuration files contain the string 'PE'.

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

To simplify debug operations, MCUXpresso IDE automatically starts PEmicro's GDB Server and selects and dynamically assigns the various ports needed as required. This means that you can start, terminate, restart, and so on, multiple PEmicro debug connections, all without the need for any user connection configuration. You can control these options if required by editing the PEmicro launch configuration file.

For more information, see Common debugging operations.

Note: If the project already had a PEmicro launch configuration, this is 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 PEmicro debug probes are supported by PEmicro software. This includes Part Support handling, Flash Programming, and many other features. If encountering problems, PEmicro maintains a range of support forums at https://www.pemicro.com/forums/.

Note: If a debug operation fails, or a crash occurs, it is possible that one or more debug processes may fail to shut down correctly. Therefore, if the IDE has no active debug connection but is experiencing problems making a new debug connection, ensure that none of these executables is running. To simplify this process, an IDE button allows you to kill all low-level debug executables (for all debug solutions). Therefore, if a debug operation fails or a crash occurs, simply click this button before starting a new debug operation.

13.8.1 PEmicro differences from LinkServer debug

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

When used with PEmicro debug probes, the debug environment is provided by the PEmicro debug server. This debug server does not 100% match the features provided by native LinkServer connections. However, basic debug operations are similar to LinkServer debug.

For a description of some common debugging operations using supported debug probes, see Common debugging operations.

Note: LinkServer advanced features such as Power Measurement are not available via a PEmicro debug connection. However, additional functionality may be available using PEmicro-supplied plugins.

13.8.2 PEmicro software updates

PEmicro support within MCUXpresso IDE is via an Eclipse plugin. The PEmicro update site is automatically added to the list of Available Software Update sites.

To check whether an update is available, select:

Help -> Check for Updates

Any available updates from PEmicro are then listed for selection and installation.

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

13.9 SEGGER debug connections

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

Currently, we have tested using:

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

13.9.1 SEGGER software installation

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

The installation location is similar to:

```
On Windows: C:/Program Files/SEGGER/JLink
On Mac: /Applications/SEGGER/JLink
On Linux: /opt/SEGGER/JLink
```

Note: The SEGGER J-Link package is available in two flavors. MCUXpresso IDE currently installs and uses the 64-bit version on all operating systems. Older IDE versions used the legacy 32-bit Windows package but starting with MCUXpresso IDE v11.5.0, the 64-bit package is shipped. The installation folder for the 32-bit Windows version is usually:

```
On Windows: C:/Program Files (x86)/SEGGER/JLink
```

MCUXpresso IDE automatically locates the required executable and it is remembered as a workspace preference. This can be viewed or edited within the MCUXpresso IDE preferences as below.
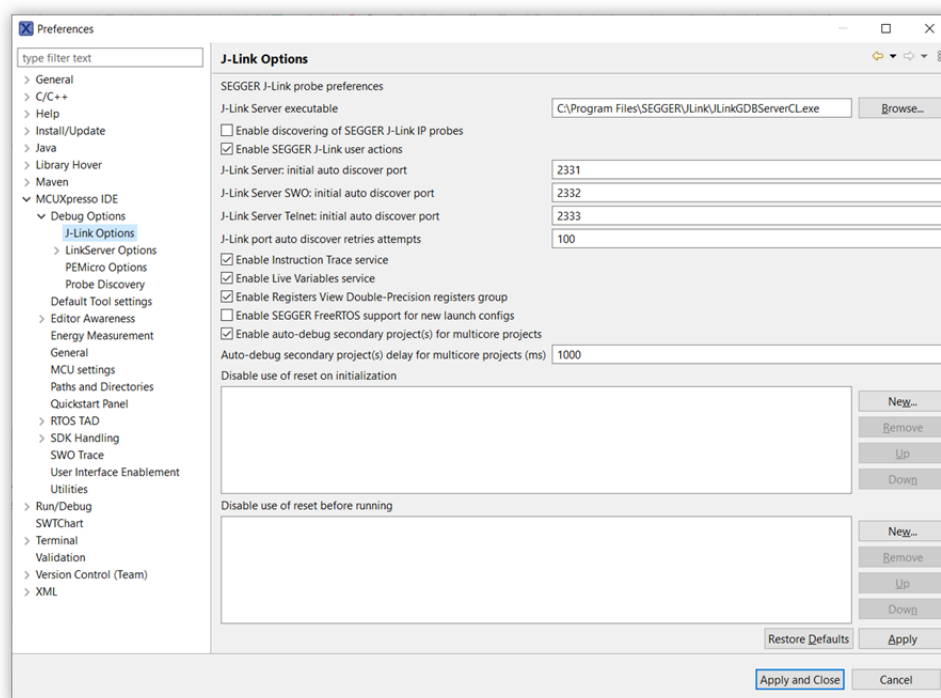
Figure 117. Segger preferences

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

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

MCUXpresso IDE continues to use the SEGGER installation path as referenced in the workspace of a project unless it cannot find the required executable (for example, the referenced installation has been deleted). If this occurs:

1. The IDE automatically searches for the latest installation it can find. If this is successful, the workspace preference is automatically updated
2. If the IDE cannot find a SEGGER installation, the user is prompted to locate an installation

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

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

For example, under Windows you could set:

```
MCUX_SEGGER_SERVER="C:/Program Files (x86)/SEGGER/JLink_V630k/
jLinkGDBServerCL.exe"
```

This location is then used, overriding any workspace preference that may be set.

13.9.1.1 SEGGER software uninstallation

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

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

13.10 SEGGER debug operation

The process to debug via a J-Link compatible debug probe is the same as for a native LinkServer (CMSIS-DAP) compatible debug probe. Simply select the project via the 'Project Explorer' view then click Debug from the **Quickstart** Panel and select the SEGGER Probe from the Probe Discovery Dialogue.

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

To simplify debug operations, MCUXpresso IDE automatically starts SEGGER's GDB Server and selects and dynamically assigns the various ports needed as required. This means that you can start, terminate, restart, and so on, multiple SEGGER debug connections, all without the need for any user connection configuration. You can control these options if required by editing the SEGGER launch configuration file.

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

For more information, see Common debugging operations.

User guide Rev. 4 — 14 January 2025 Document feedback

122 / 316

Note: If the project already had a SEGGER launch configuration, this is 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.

Tip: If Reset before running is set in the Launch configuration, then a default intelligent reset is used. This reset automatically supports running from Flash or RAM. A specific reset type can optionally be set from the free-form text field if required, consult SEGGER's documentation for available reset types.

Important Note: SEGGER software supports low-level debug operations via SEGGER debug probes. This includes Part Support handling, Flash Programming, and many other features. If encountering problems, SEGGER provides a range of support forums at https://forum.segger.com/.

13.10.1 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 SEGGER debug server provides the debug environment. This debug server does not 100% match the features provided by native LinkServer connections. However, basic debug operations are similar to LinkServer debug.

For a description of some common debugging operations using supported debug probes, see Common debugging operations.

Note: LinkServer features such as Power Measurement are not available via a SEGGER debug connection. However, additional functionality may be available using external SEGGER-supplied applications.

13.11 SEGGER troubleshooting

When performing a debug operation to a SEGGER debug probe, the launch configuration file provides a set of arguments that are used to call the SEGGER GDB server. The command and resulting output are logged within the IDE SEGGER Debug Console. You can view the console below:

Figure 118. Segger Server

You can copy and call the command independently of the IDE to start a debug session and explore connection issues.

Below is the shortened output of a successful debug session to a Kinetis K64 Board.

```
[05-1-2023 11:26:24] Executing Server: "C:\Program Files\SEGGER\JLink
\JLinkGDBServerCL.exe" /
-SettingsFile "C:\Users\MCUXpresso\Documents\MCUXpressoIDE_11.7.0\workspace /
frdmk64f_bubble_peripheral\Debug\frdmk64f_bubble_peripheral JLink Debug
 SettingsFile.jlink" /
-nosilent -swoport 2332 -select USB=174505240 -telnetport 2333 -singlerun -
endian little /
```

```
-noir -speed 4000 -port 2331 -vd -device MK64FN1M0xxx12 -if SWD -halt -
reportuseraction /
SEGGER J-Link GDB Server V7.84a Command Line Version

JLinkARM.dll V7.84a (DLL compiled Dec 22 2022 16:11:39)

Command line: -SettingsFile C:\Users\MCUXpresso\Documents
\MCUXpressoIDE_11.7.0\workspace /
frdmk64f_bubble_peripheral\Debug\frdmk64f_bubble_peripheral JLink Debug
SettingsFile.jlink / -nosilent -swoport 2332 -select USB=174505240 -telnetport
 2333
-singlerun -endian little / -noir -speed 4000 -port 2331 -vd -device
 MK64FN1M0xxx12 -if
SWD -halt -reportuseraction
-----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:      localhost only
Generate logfile:              off
Verify download:               on
Init regs on start:            off
Silent mode:                   off
Single run mode:               on
Target connection timeout:     0 ms
------J-Link related settings------
J-Link Host interface:         USB
J-Link script:                 none
J-Link settings file:          C:\Users\MCUXpresso\Documents
\MCUXpressoIDE_11.7.0\workspace /
frdmk64f_bubble_peripheral\Debug\frdmk64f_bubble_peripheral JLink Debug
 SettingsFile.jlink
 ------Target related settings------
 Target device:                MK64FN1M0xxx12
Target device parameters:      none
Target interface:              SWD
Target interface speed:        4000kHz
Target endian:                 little

Connecting to J-Link...
J-Link is connected.
Device "MK64FN1M0XXX12" selected.
Firmware: J-Link Pro V4 compiled Sep 22 2022 15:00:37
Hardware: V4.00
S/N: 174505240
Feature(s): RDI, FlashBP, FlashDL, JFlash, GDB
Checking target voltage...
Target voltage: 3.26 V
Listening on TCP/IP port 2331
Connecting to target...
InitTarget()
Found SW-DP with ID 0x2BA01477
DPIDR: 0x2BA01477
CoreSight SoC-400 or earlier
Scanning AP map to find all available APs
AP[2]: Stopped AP scan as end of AP map has been reached
AP[0]: AHB-AP (IDR: 0x24770011)
AP[1]: JTAG-AP (IDR: 0x001C0000)
Iterating through AP map to find AHB-AP to use
AP[0]: Core found
```

```
AP[0]: AHB-AP ROM base: 0xE00FF000
CPUID register: 0x410FC241. Implementer code: 0x41 (ARM)
Found Cortex-M4 r0p1, Little endian.
FPUnit: 6 code (BP) slots and 2 literal slots
CoreSight components:
ROMTbl[0] @ E00FF000
[0][0]: E000E000 CID B105E00D PID 000BB00C SCS-M7
[0][1]: E0001000 CID B105E00D PID 003BB002 DWT
[0][2]: E0002000 CID B105E00D PID 002BB003 FPB
[0][3]: E0000000 CID B105E00D PID 003BB001 ITM
[0][4]: E0040000 CID B105900D PID 000BB9A1 TPIU
[0][5]: E0041000 CID B105900D PID 000BB925 ETM
[0][6]: E0042000 CID B105900D PID 003BB907 ETB
[0][7]: E0043000 CID B105900D PID 001BB908 CSTF
Connected to target
Waiting for GDB connection...Connected to 127.0.0.1
Reading common registers: R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12,
 SP, / LR, PC, XPSR
Connected to 127.0.0.1
Reading common registers: R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12,
 SP, / LR, PC, XPSR
Read 4 bytes @ address 0x000047E6 (Data = 0x46BD3714)
Read 4 bytes @ address 0x000047E6 (Data = 0x46BD3714)
Read 4 bytes @ address 0x00000E14 (Data = 0x687B6078)
Reading 64 bytes @ address 0x2002FE40
Read 4 bytes @ address 0x00000E14 (Data = 0x687B6078)
Reading 64 bytes @ address 0x2002FE40
Received monitor command: reset
Reset: Halt core after reset via DEMCR.VC_CORERESET.
Reset: Reset device via AIRCR.SYSRESETREQ.
AfterResetTarget()
Resetting target
Downloading 16016 bytes @ address 0x00000000 - Verified OK
Downloading 8496 bytes @ address 0x00003E90 - Verified OK
Downloading 16 bytes @ address 0x00005FC0 - Verified OK
J-Link: Flash download: Bank 0 @ 0x00000000: Skipped. Contents already match
Writing register (PC = 0x      1d4)
Read 4 bytes @ address 0x000001D4 (Data = 0xF002B672)
Read 4 bytes @ address 0x000001D4 (Data = 0xF002B672)
Reading common registers: R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12,
 SP, / LR, PC, XPSR
Read 4 bytes @ address 0x000001D4 (Data = 0xF002B672)
Reading 64 bytes @ address 0x00000F00
Read 2 bytes @ address 0x00000F22 (Data = 0xF107)
Received monitor command: semihosting enable
Semi-hosting enabled (Handle on breakpoint instruction hit)
Received monitor command: exec SetRestartOnClose=1
Executed SetRestartOnClose=1
Received monitor command: reset
Reset: Halt core after reset via DEMCR.VC_CORERESET.
Reset: Reset device via AIRCR.SYSRESETREQ.
AfterResetTarget()
Resetting target
Setting breakpoint @ address 0x00000F22, Kind = 2, Type = THUMB, BPHandle =
 0x0001
Starting target CPU...
...Breakpoint reached @ address 0x00000F22
Reading common registers: R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12,
 SP, / LR, PC, XPSR
Removing breakpoint @ address 0x00000F22, Size = 2
```

```
Read 4 bytes @ address 0x00000F22 (Data = 0x031CF107)
Reading 64 bytes @ address 0x2002FFC0
Read 4 bytes @ address 0xE00FFFF4 (Data = 0x00000010)
...
```

Note: If a SEGGER debug operation is not successful, the IDE generates an error dialog, and the user can click the *'Details'* button to display a copy of the SEGGER server log. One possible reason for a SEGGER debug operation failing is due to a device name mismatch. MCUXpresso IDE tries to supply the expected device name to the SEGGER server, however, on rare occasions, this may not be the name expected. The SEGGER launch configuration Device entry can be populated via a dropdown list or via a user-supplied device name.

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

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

Where *$(CWD)* represents the current working directory of the debug connection, that is, the dynamically created project build configuration folder.

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

IDE button ▓ allows you to kill all low-level debug executables (for all debug solutions). Therefore, if a debug operation fails or a crash occurs, simply click this button before starting a new debug operation.

14 Debugging a project

This chapter describes many of the common debug features supported by the debug solutions within MCUXpresso IDE. Also refer to the chapter [Debug solutions overview](#) for more details of the supported debug solutions and management of debug operations.

14.1 Debugging overview

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

This connection to the debug probe is usually made via USB to the host computer (although IP probes from PEmicro 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, Tower, and EVK boards also incorporate a built-in debug probe accessed by one of the development boards USB connections.

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

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

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

Note: Some LPCXpresso development boards have two USB connectors fitted. Make sure that you have connected the lower connector marked DFU-Link. 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 usually a Kinetis K20 MCU, which is approximately 6 mm square. The USB nearest to this MCU is the OpenSDA connection.

14.1.1 Debug launch

To debug a project on your target MCU, simply highlight the appropriate project in the 'Project Explorer', and then in the **Quickstart** Panel click the large **Debug**, as in Figure 119, alternatively click the blue bug icon ✳ to perform the same action.
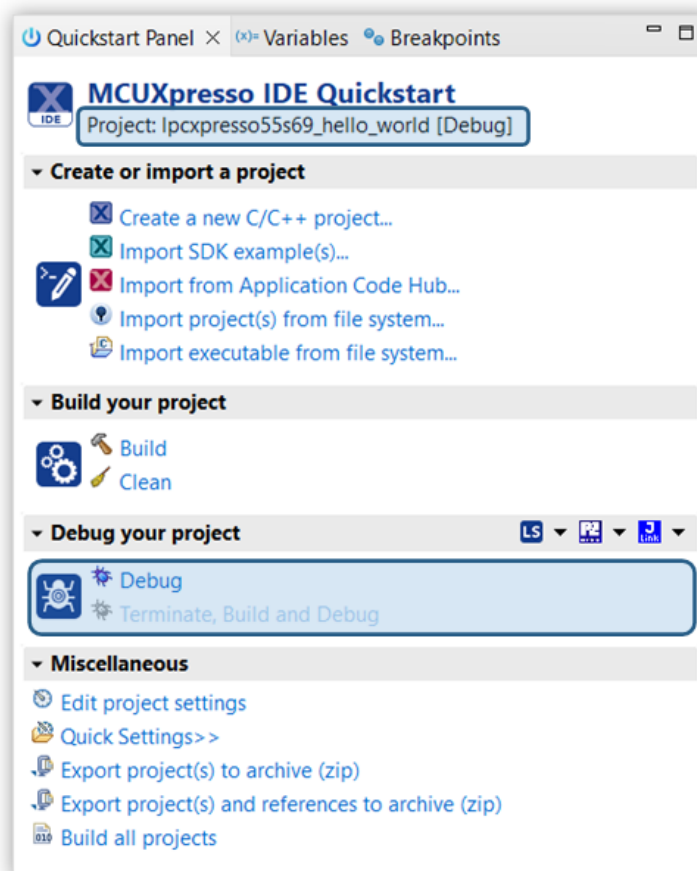
Figure 119. Launching a debug session

Note: Do not use the green bug icon.

Note: This default behavior can be changed by editing the Workspace preference located at *Preferences -> Run/Debug* because this invokes the standard Eclipse debug operation and so skips certain essential MCUXpresso IDE debug steps.

For a newly created project, a debug operation performs various steps. By default, it first builds the selected project and (assuming there are no build errors) launch a debug probe discovery operation (see next section) to allow the user to select the required debug probe. A launch configuration file is automatically created with default options (per build configuration) and is associated with the project. Like the build configuration of a project, launch configuration files control what occurs each time a debug operation is performed. See the section An introduction to launch configuration files for more information.

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

By default, once you have selected a debug probe (and clicked 'OK'), the binary contents of the .axf file are automatically 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 is automatically selected to perform the Flash programming operation. Next, a default breakpoint is set on the first instruction in `main()`, the application starts (by performing or simulating a processor reset), and code executes until hitting the default breakpoint. See the section on Breakpoints for additional information.

14.1.2 Debug probe selection dialog (probes discovered)

The first time you debug a project, the IDE performs a probe discovery operation and displays the discovered Debug Probes for selection. This shows a dialog listing all supported probes that are attached to the host computer. In the example shown in Figure 120, a LinkServer (LPC-Link2), a PEmicro Multilink, and also a J-Link (OpenSDA) probe have been found.
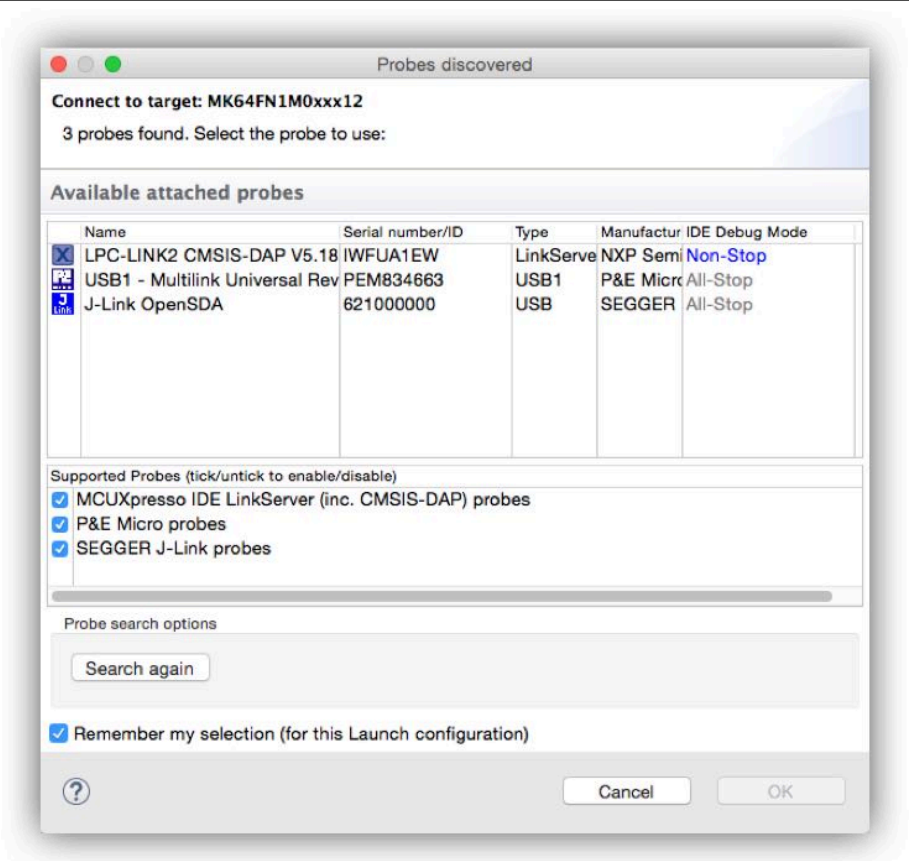
Figure 120. Attached probes: debug emulator selection

Note: if it finds only one probe, the IDE selects it automatically, so simply click OK or hit return to use the probe displayed.

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 always return the same ID, however, debug probes such as the LPC-Link2 return a unique ID for each probe - in our example **IWFUA1EW**.

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

However, if you perform a debug operation and the IDE cannot find the previously remembered debug probe, then it performs a debug probe discovery operation from within the same family, for example, LinkServer, PEmicro, or SEGGER. If you are using a different debug probe from the same family of debug probes, simply select the new probe to replace the previously selected probe.

See also debug shortcuts.

Sometimes a probe discovery finds no debug probes and returns a dialog as below:
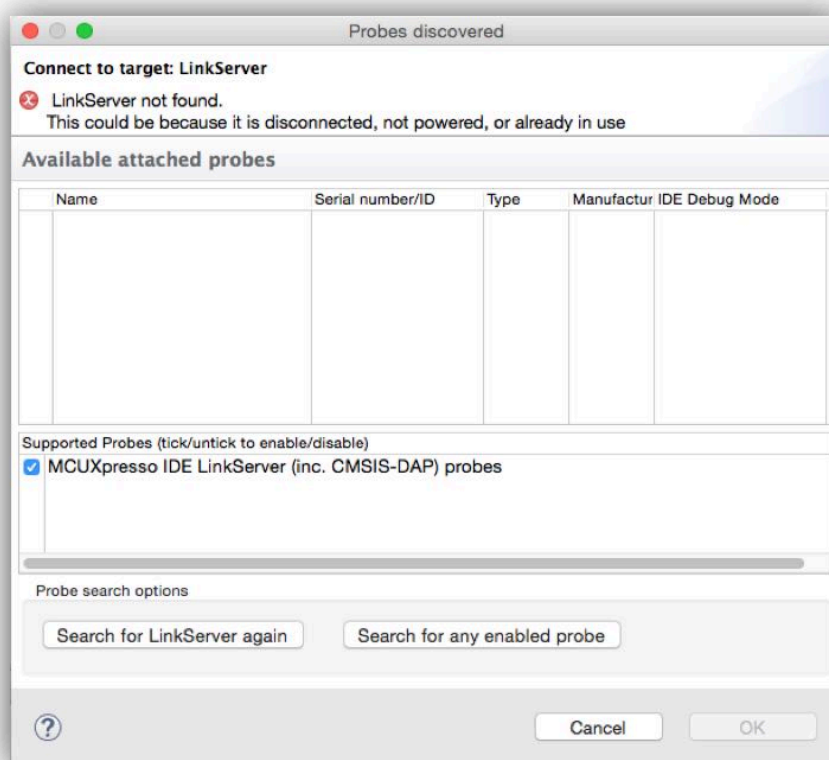
Figure 121. LPC-Link2 no longer connected

This might have been because you forgot to connect the probe, in which case simply connect it to your computer and select **Search again**. Some CMSIS-DAP probes also support disabling the debug capabilities via a jumper pin, in which case the IDE filters them out. Consult your probe's manual and make sure that the jumper configuration enables debugging operations.

Notes:

• The "Remember my selection" option is enabled by default in the Debug Emulator Selection Dialog, and causes the selected probe to be stored in the launch configuration for the current configuration (typically Debug or Release) of the current project. You can therefore remove the probe selection at any time by simply deleting the launch configuration.
• Select a probe for each project that you debug within a workspace (and for each configuration within a project).

- If you wish to debug a project using a different family of debug probe(s), then the simplest option is to delete the launch configuration files associated with the project and start a debug operation. See the section "An Introduction to Launch Configuration files for more information. Also see Debug shortcuts.

14.1.2.1 Automatic probe selection.

When using CMSIS-DAP probes with target information available, launching a debugging session causes the probe selection dialog to autoselect the entry that matches the board and device description of your project. Discovered probes that have target information available are decorated with an info icon. Hovering over those probe entries reveals a tooltip containing the detected target information.
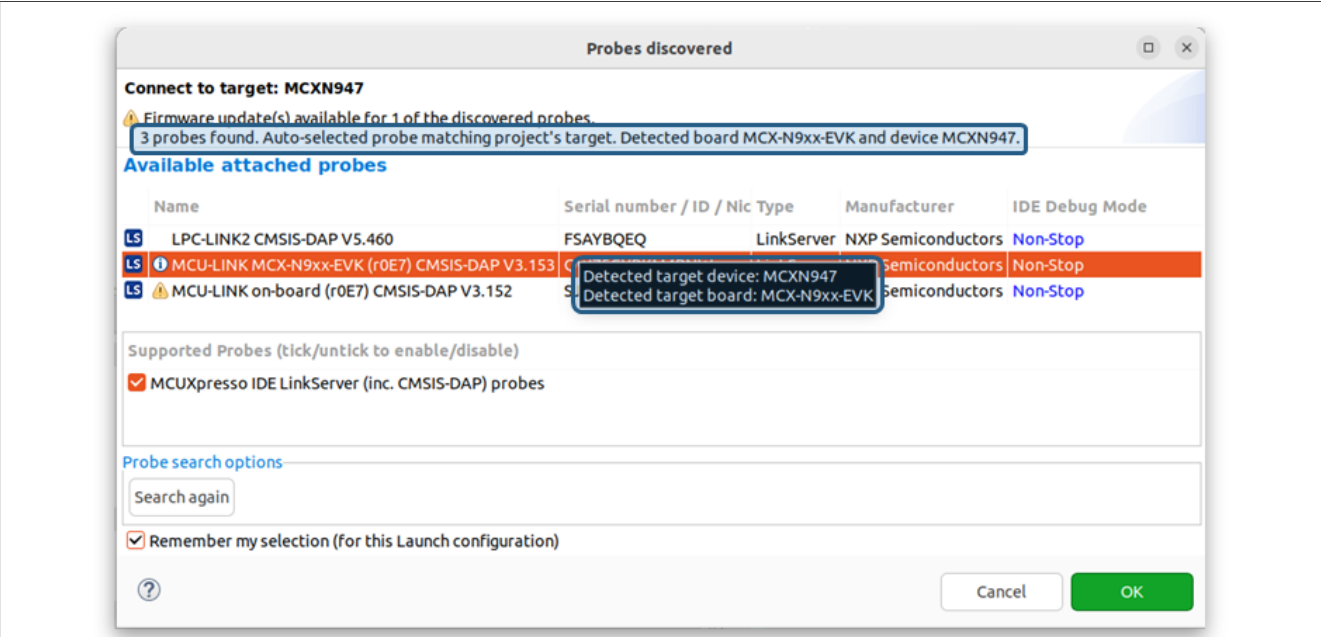
Figure 122. Probe automatically selected based on the detected target

As a safety mechanism, if you select a probe that is configured for a different board or device than the ones your project is configured for, the IDE warns you of the mismatch before proceeding with the debug operation.
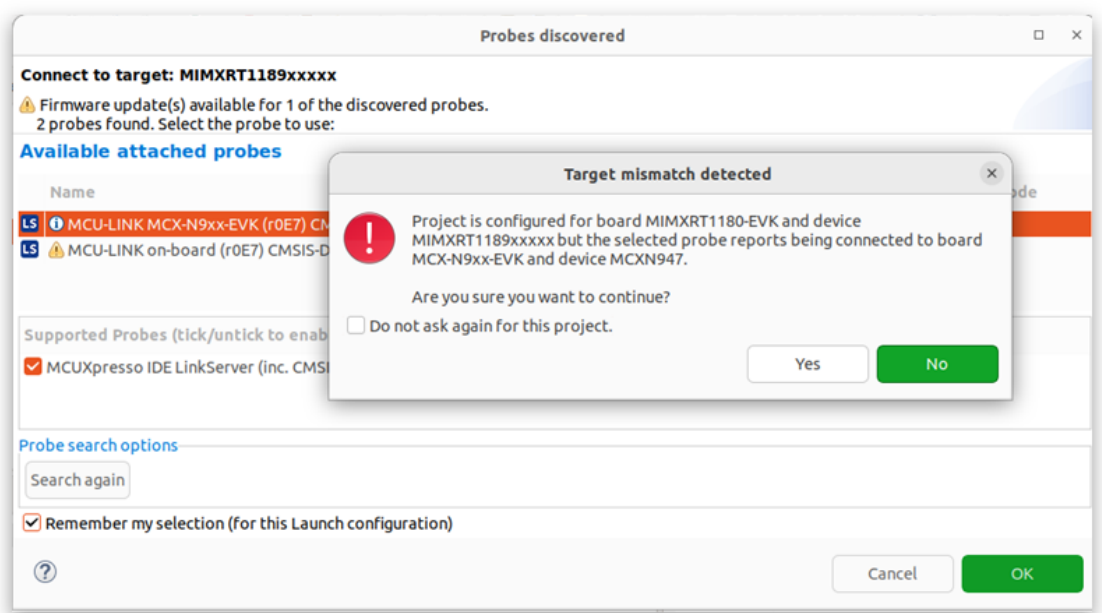
Figure 123. Dialog warning user of target mismatch

14.1.2.2 Firmware version check on MCU-Link / MCU-Link Pro probes

For MCU-Link and MCU-Link Pro probes, the **Probes Discovered** indicates if a newer firmware version is available.
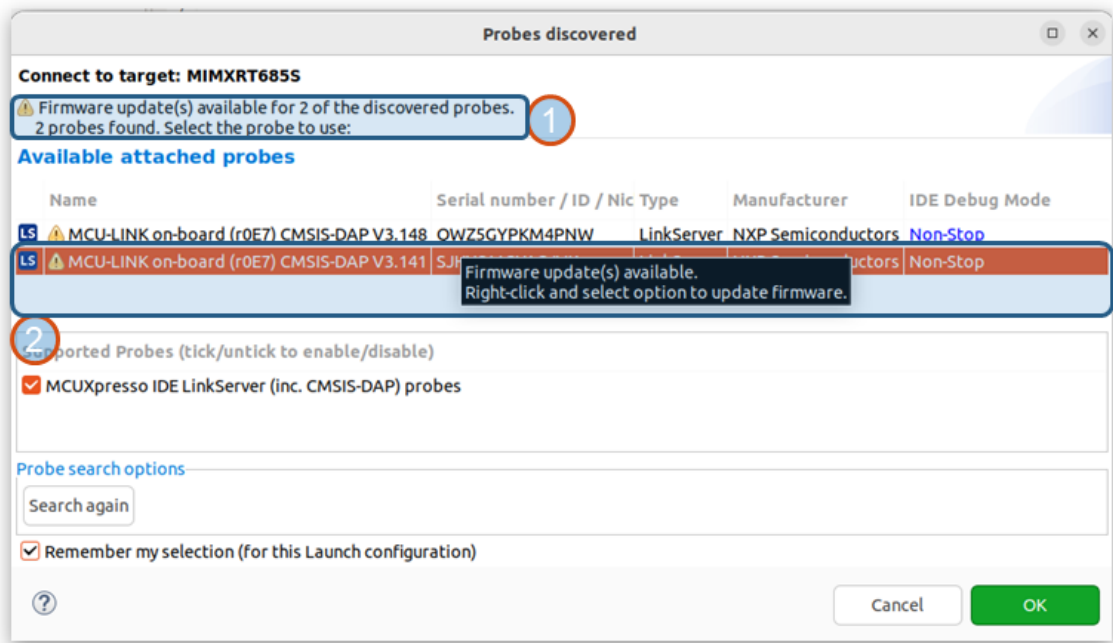
Figure 124. MCU-Link available firmware update indication

1. Once the Probes Discovered opens, a warning indicates if there are debugger probes that require a firmware update.
2. A warning icon decorates each discovered probe for which a firmware update is available. Additionally, hovering over a table entry displays a tooltip informing of the possibility of updating the probe. Right-Click on a particular probe entry and two options may appear:
   - Bundled update option that uses the firmware package locally bundled with the currently configured LinkServer installation.
   - Remote update option that downloads the latest released firmware package and installs it on your system.
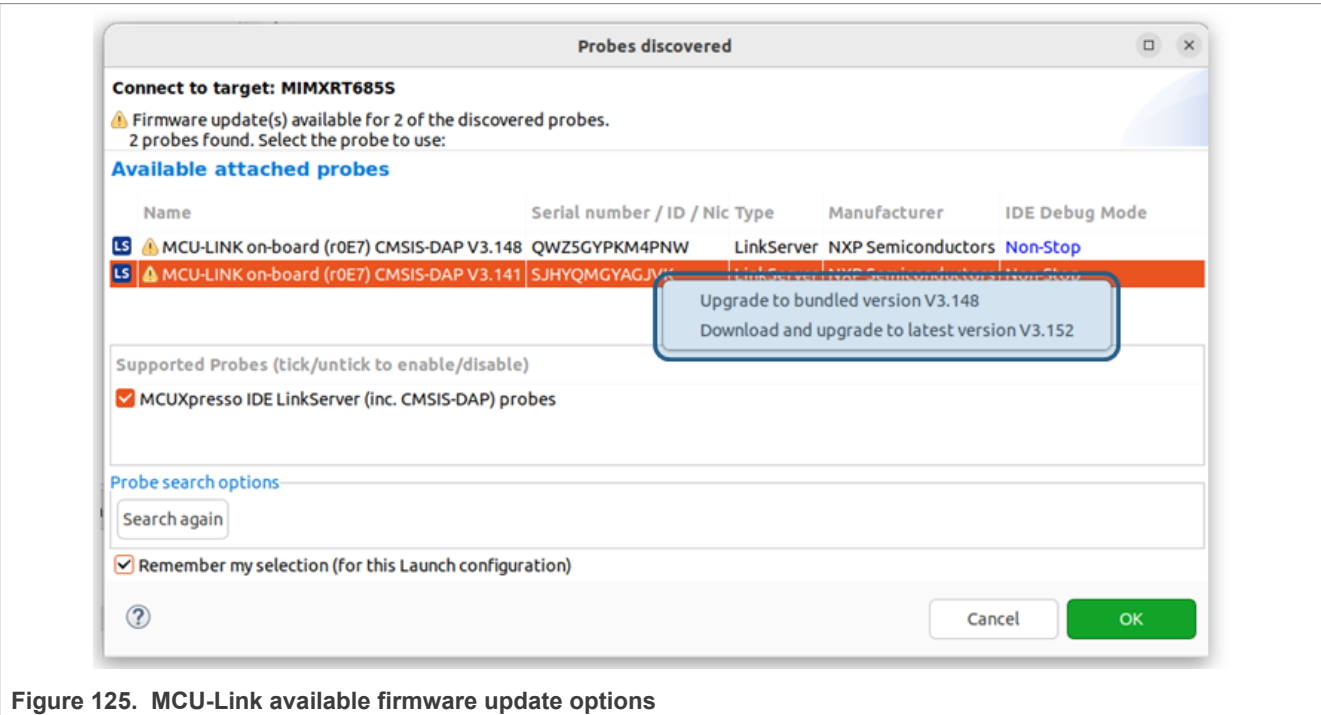
Figure 125. MCU-Link available firmware update options

Select an update option and a dialog will appear and inform you of the progress and status of the update process. Once the update finishes, clicking OK refreshes the probe selection table showing the newly updated MCU-Link probe with the latest firmware version.
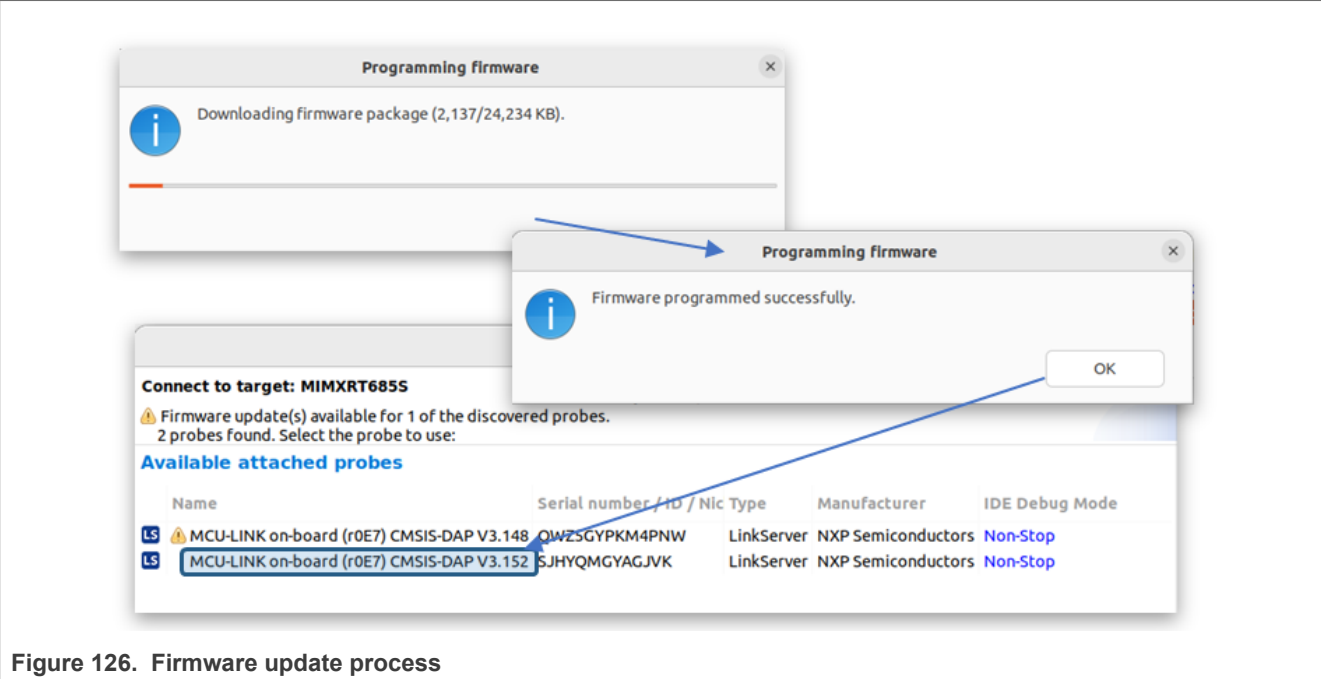
Figure 126. Firmware update process

14.1.3 Controlling execution

When you have started a debug session a default [breakpoint](#) is set on the first instruction in `main()`, the application starts (by simulating or performing a processor reset), and code executes until hitting the default "breakpoint.

You can now control program execution by using the common debug control buttons, as listed in [Table 1](#), which you can see on the global toolbar. The call stack is shown in the Debug View, as in [Figure 127](#).
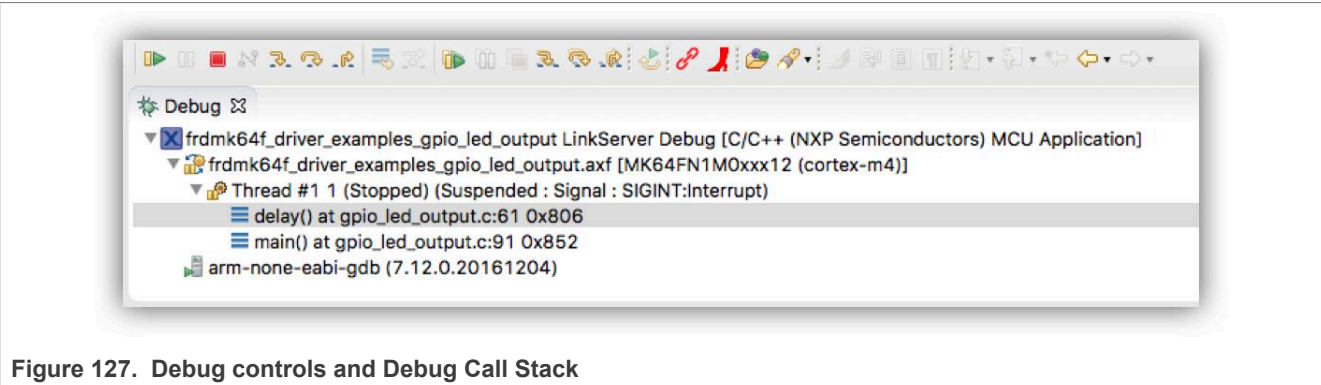
Figure 127. Debug controls and Debug Call Stack

Table 1. Program execution controls

Button	Description	Keyboard shortcut
	Restart program execution (from reset)	
	Run/Resume the program	F8
	Pause Execution of the running program	
	Terminate the debug Session	Ctrl + F2

Table 1. Program execution controls...*continued*

Button	Description	Keyboard shortcut
	Clean up debug	
	Run, Pause, Terminate all debug sessions	
	Step over a C/C++ line	F6
	Step into a function	F5
	Return from a function	F7
	Step in, over, out all debug sessions	
	Show disassembled instructions	

*Tip: Clean up debug kills all debug processes associated with LinkServer, PEmicro, and SEGGER debug connections. This may be necessary if the IDE restarts with a connected debug session or if a crash occurs - and removes any failed or orphaned debug processes. **Note**: a warning appears with the option to cancel before performing any action since this kills all connected debug sessions.*

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

Note: Typically a user only has a single active debug session. However, if there is more than one debug session, you can choose the active session by clicking within the debug call stack within the Debug view. All debug views reflect the selected session.

Setting a breakpoint

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

Restarting the application

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

Stopping debugging

To stop debugging press the **Terminate/Stop** button. This action disconnects MCUXpresso IDE from the target (board). The subsequent behavior is controllable by the disconnect behavior.

Pause debugging

Typically, debugging is paused due to the action of a breakpoint or watchpoint since these are set to observe the target when an event of interest has occurred. However, the pause button can be used to pause the target at an instant of time.

To pause debugging

If you are debugging using the **Debug Perspective**, then to switch back to the **C/C++ Perspective** when you stop your debug session, click the **C/C++** tab in the upper right area of MCUXpresso IDE (as shown in Figure 10).

14.2 Launch configurations

Launch Configuration files are automatically created within the root directory of a project the first time a debug operation occurs. They are typically named:

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

A file is created for the build variant being debugged and is used to store the settings for the debug connection for that build configuration.

Normally, there is no need to edit launch configurations, as the default settings created by the IDE are suitable. However, in some circumstances, you may need to manage them - typically under direction from an FAQ. In such cases, you can do this via the "Launch Configurations" entry on the context-sensitive menu available from the Project Explorer view...
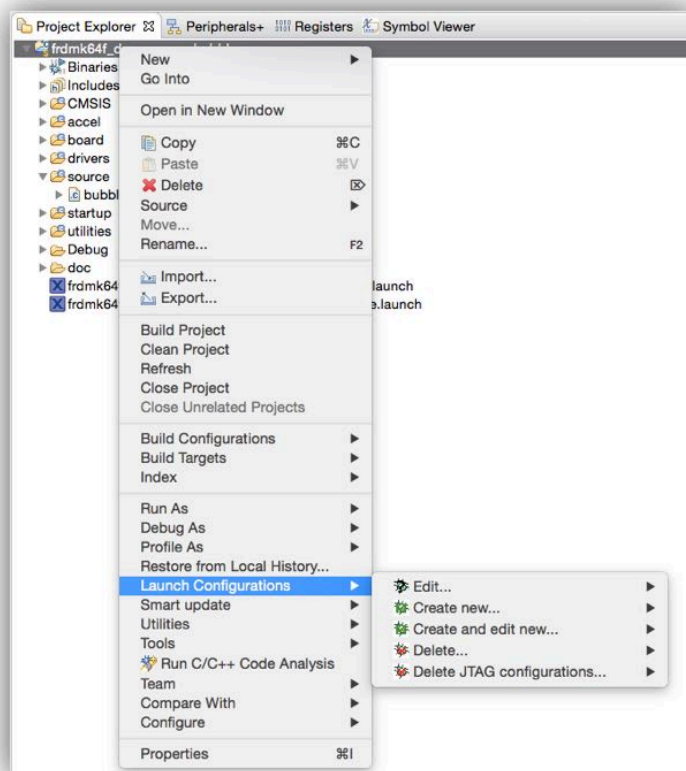
Figure 128. Create a launch configuration

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

A number of options are available here:

Edit...

• Allows various debug settings to be modified.
  – Typically not required since the default options are correct for most debug operations.

Create new...

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

Create and edit new...

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

Delete...

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

Delete JTAG Configuration...

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

14.2.1 Editing a launch configuration (LinkServer)

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

After selecting the "Edit..." or "Create and edit New" launch the configuration menu entry, then you see a new dialog box pop up, which looks similar to the following...
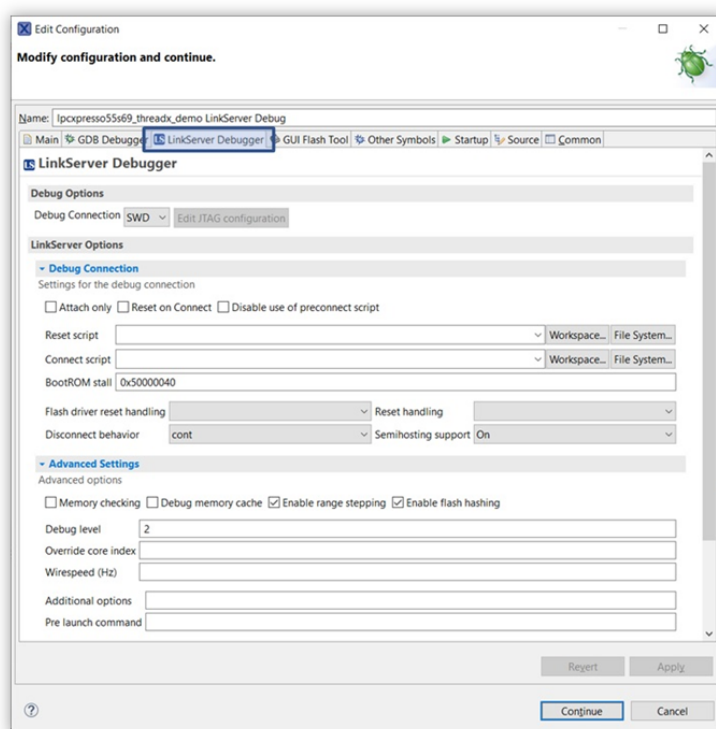
Figure 129. Edit a launch configuration

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

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

- Changing the initial [breakpoint](#) on debug startup
  - When the debugger starts, it automatically sets an initial (temporary) breakpoint on the first statement in main(). If desired, you can change where this initial breakpoint is set, or even remove it completely.
- Modifying the Debugger connect behavior
  - via a Connect Script, for example, kinetisconnect.scp
- Modifying the Debugger reset behavior.
  - *Flash driver reset handling* is used to run the RAM-loaded flash driver, while *Reset handling* is used to start the image loaded into flash/RAM. Possible choices:
    - *SOFT*: Maintain the current software environment but change the SP and PC. The values for SP and PC are read from the first two words of the binary image, so *SOFT* is not applicable in case of flash images, which begin with a bootheader / configuration block.
    - *VECTRESET*: Execute a hardware reset of the core and catch the vectored "reset" event. ARMv8-M cores lack this reset mechanism, so *SOFT* reset is used instead.
    - *SYSRESETREQ*: Execute a hardware restart of the system and catch the vectored "reset" event.
    - *Default*: For *Flash driver reset handling*, it usually means *VECTRESET*, except for specific parts. For *Reset handling*, it means *SYSRESETREQ* for flash and *VECTRESET* for RAM.
  - If the *Reset script* input field is not empty, the specified reset script executes instead of the selected *Reset handling*.
  - *BootROM stall*: Temporarily stall the boot loader post initialization on a read watchpoint to reestablish debug control. Useful for ARMv8-M cores, which lack vector catch.
- Connecting to a target via JTAG rather than SWD
  - If supported by the target, you can edit the Debug type
- Connecting to a running target
  - Set Attach only to True (see also [debug shortcuts)](#)
- Changing the debug stub connection parameters
  - By default, the IDE attempts to start the GDB stub when initiating a debug connection. As a result, the launch configuration is automatically created with the "Automatically start debug server" checkbox enabled. If the GDB stub is already running, and the GDB client must connect to that instance, it must be unchecked.
  - Controls associated with the host name and with the network port number that are used for listening to GDB client connections are grayed out by default. You can change these parameters by disabling "Automatically start debug server".
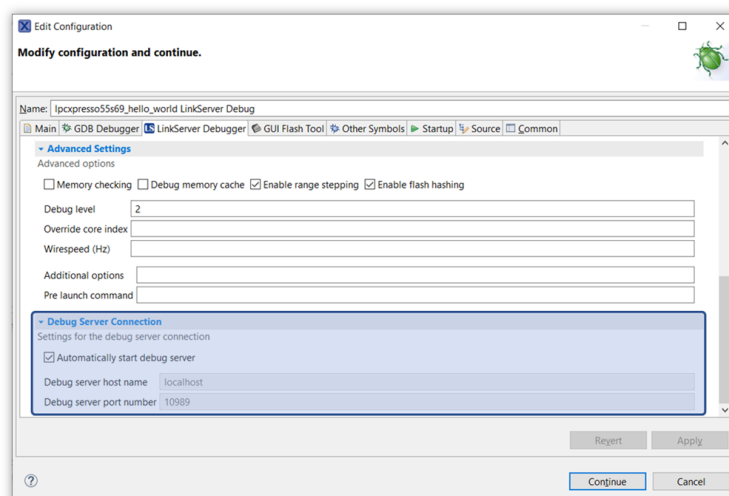
Figure 130. Debug Server connection

You can also change the Debug Server Connection parameters by using the Preferences page. All launch configurations are created using the values specified there. Changes in the Preferences page do not affect any existing launch configurations. Go to **Window -> Preferences -> MCUXpresso IDE -> Debug Options -> LinkServer Options** to change any of the debug server-related parameters.
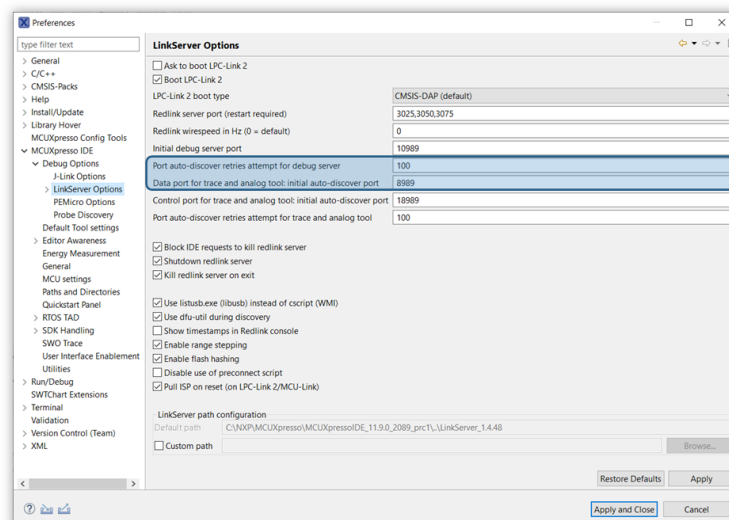
Figure 131. Debug Server connection

14.2.1.1 Target boot configuration

- Changing target boot configuration
  - LinkServer debug configuration has a section for Target Boot Control (highlighted below):
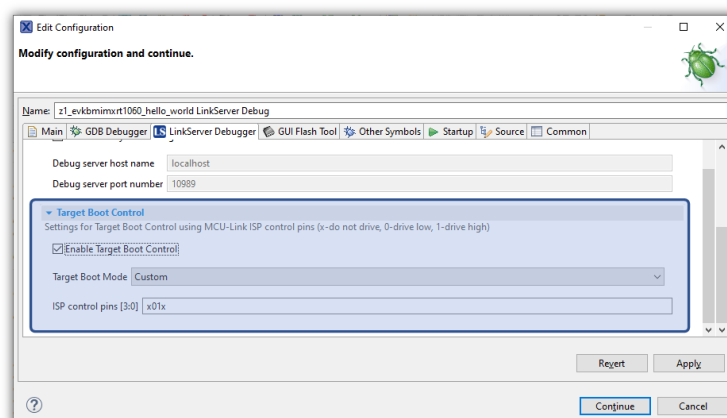
Figure 132. Target Boot Control

- The main purpose of this feature is to configure how the device boots on the reset requests issued during the debug session. This feature requires an **MCU-Link debug probe** with the ISPx boot control features implemented.
- By default, the target boot control is disabled when the launch configuration is created. If a specific target boot configuration is required, you must enable the target boot control, change the value present in the edit box, and/or select the boot mode from the list. At this moment, there are no predefined target boot mode entries in the list. The combo box has two generic entries:
  - **Custom** entry: at least one ISP control pin is configured to be driven (custom configuration)
  - **Empty** entry: no ISP control pin is driven (default).
- When the target boot control is enabled, the selected configuration is stored on the probe and a wire reset is issued to apply it at the very beginning of the debug session. Also, it overrides the **Pull ISP on reset** option from **Window -> Preferences -> MCUXpresso IDE -> Debug Options -> LinkServer Options**.
- **Note**: The on-board MCU-Link probes have support for up to 4 ISP control pins, being able to drive up to four target device pins, but this depends on how the target board is designed. MCU-Link Pro and the MCU-Link base probes have one ISP control pin (ISP0).
- **WARNING**: Depending on the selected target boot mode (ISP control pins state), debug session failures might occur in cases where code exists in the source of the boot media (for example, in flash) and that code prevents the debugger from gaining control.

14.3 Common debug operations and launch configurations

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

14.3.1 Debug Quickstart shortcuts

MCUXpresso IDE Quickstart panel incorporates Debug shortcut buttons. These buttons invoke actions **only** from their respective debug solutions.
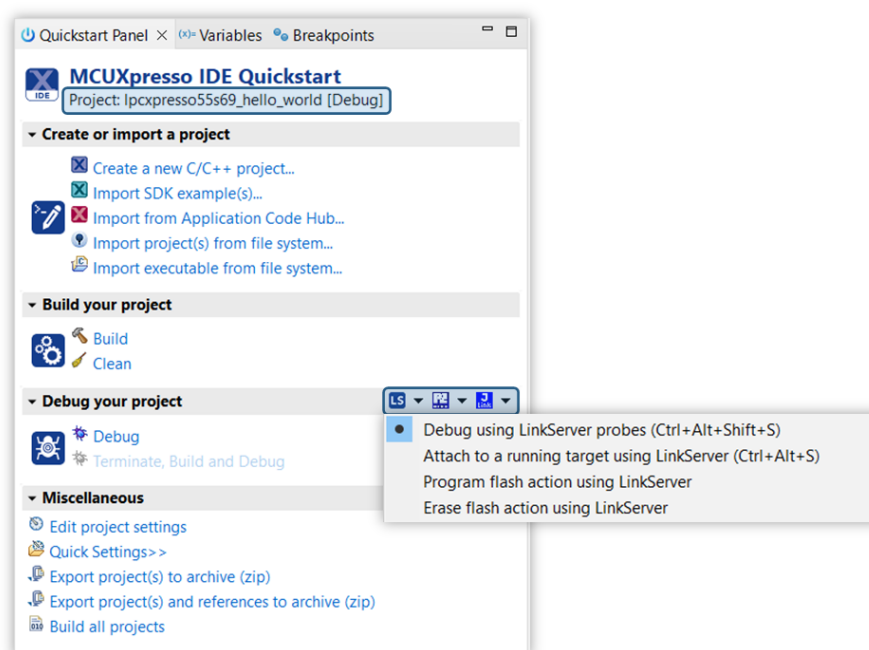
Figure 133. Debug shortcuts (LinkServer shown)

Each button provides the same 4 options for each debug solution:

Debug (default): make a **Debug** connection to the chosen debug probe. Create a launch configuration if not present. Set the attach mode **False**. **Note**: a normal debug operation inherits a launch configuration attach setting, whereas this operation forces attach mode to False. If a launch configuration exists, set its attach setting to **False**, and make no other changes.

Attach: make an **Attach** connection to a LinkServer compatible debug probe. Create a launch configuration if not present. Set the attach mode to **True**. Gives the launch configuration a *A* decorator to show that Attach is the set configuration. button. If a launch configuration exists, set its attach setting to **True**, and make no other changes.

Program Flash: perform the launch configuration *Program action*. By default, this programs the 'project' into flash. Build the selected project if required and create a default launch configuration if one is not present.

Erase Flash: perform the launch configuration *Erase action*. By default, this erases the flash memory via a mass erase. Creates a default launch configuration if one is not present.

Note: the selected action is remembered for subsequent shortcut uses, and the tooltip shows the action to perform.

Tip: If an attach operation is performed, the created launch configuration has Attach set to True. Therefore, any subsequent debug operations are in Attach Mode, until either you edit the launch configuration to set Attach to false, or you use the Debug shortcut again to force the attach mode to false.

14.3.2 Connecting to a running target (attach)

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

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

Important Note: Be sure to read and understand the section on semihosted printf and debugging and also the implications in the related section on library selection.

14.3.2.1 LinkServer

Edit the project launch configuration by double-clicking the launch config file, select the Debugger tab and Target configuration view, and then set the 'Attach only' setting to True as below:
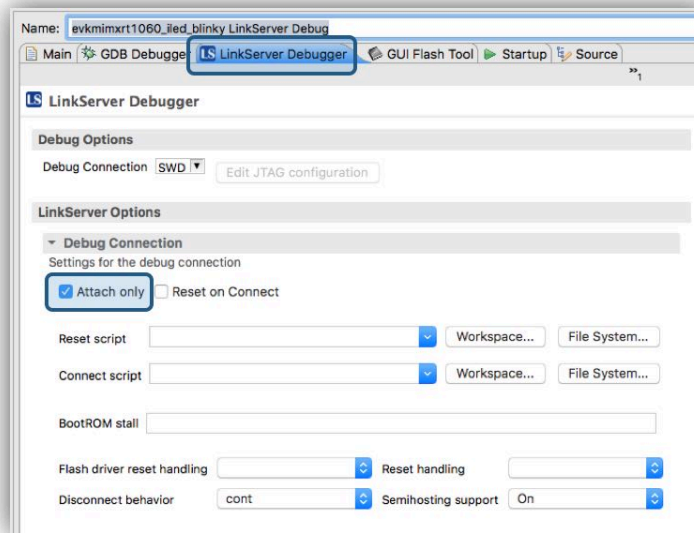
Figure 134. Debug Launch Attach mode

When making a debug connection, the target continues running until a pause occurs. However, if the IDE Debug Mode is set to Non-Stop (the default) then Global variables values can be explored and displayed.

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

14.3.2.2 PEmicro

Edit the project launch configuration by double-clicking the launch config file, select the Startup tab, and then set the 'Attach to a running target' checkbox as below:
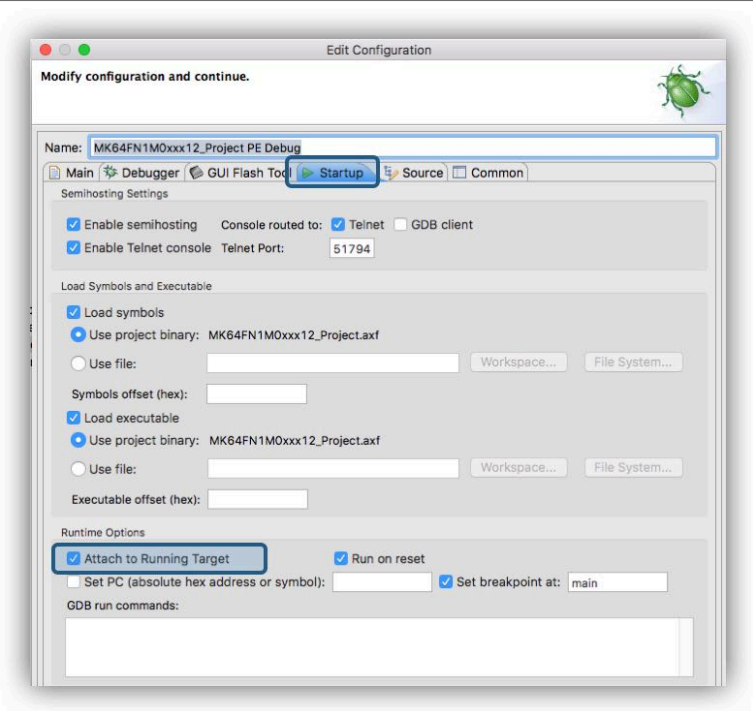
Figure 135. Debug Launch Attach mode PEmicro

When making a debug connection, the target continues running until a pause occurs.

14.3.2.3 SEGGER JLink

Edit the project launch configuration by double-clicking the launch config file, select the Debugger tab, and then set the 'Attach to a running target' checkbox as below:
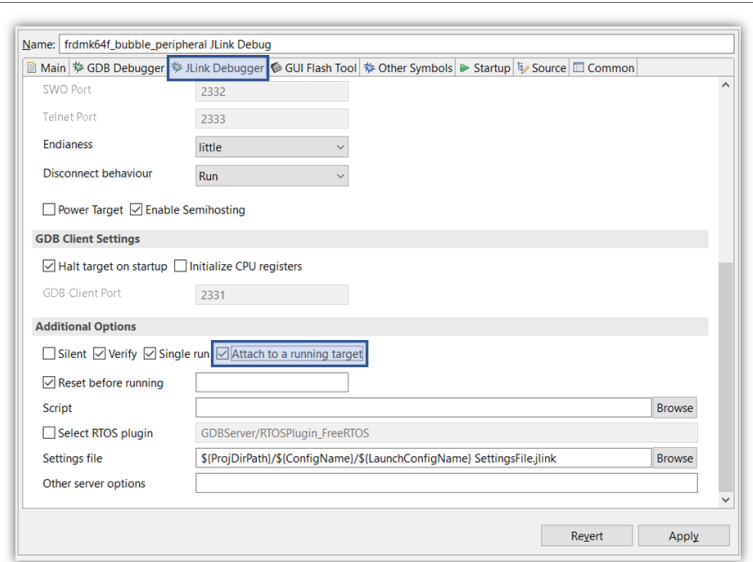
Figure 136. Debug Launch Attach Segger

When making a debug connection, the target continues running until a pause occurs.

14.3.3 Controlling the initial breakpoint (on main)

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

14.3.3.1 LinkServer

To debug from the start of the image, edit the project launch configuration by double-clicking the launch config file, select the Debugger tab, replace main with ResetISR.
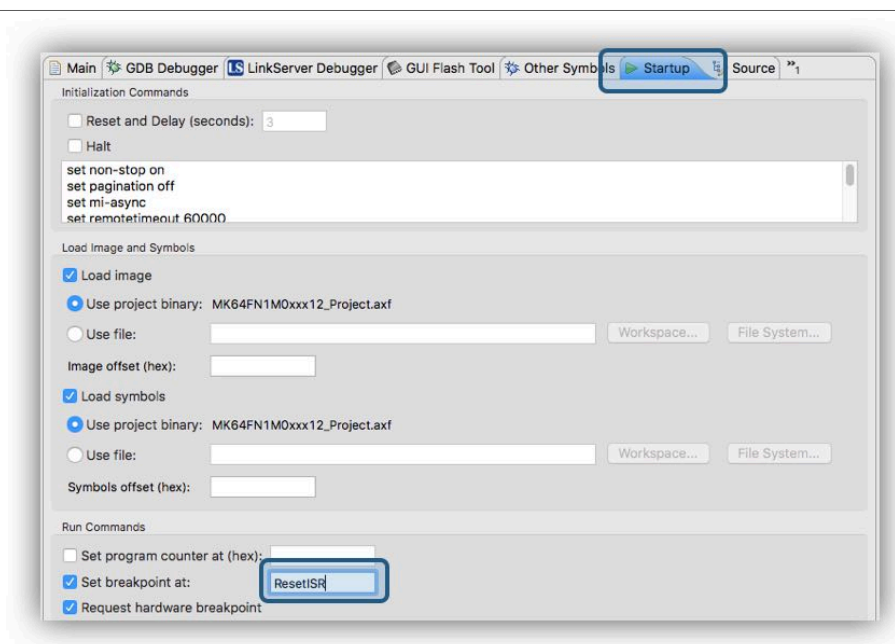
Figure 137. Debug Launch ResetISR

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

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

14.3.3.2 PEmicro

Edit the project launch configuration by double-clicking the launch config file, select the Startup tab, replace main with ResetISR.
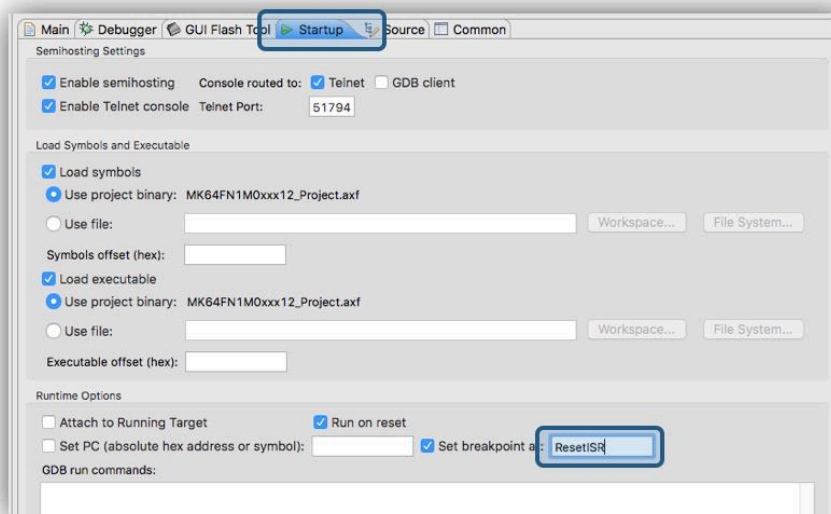
Figure 138. Debug Launch ResetISR PEmicro

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

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

14.3.3.3 SEGGER JLink

Edit the project launch configuration by double-clicking the launch config file, select the Startup tab, replace main with ResetISR.
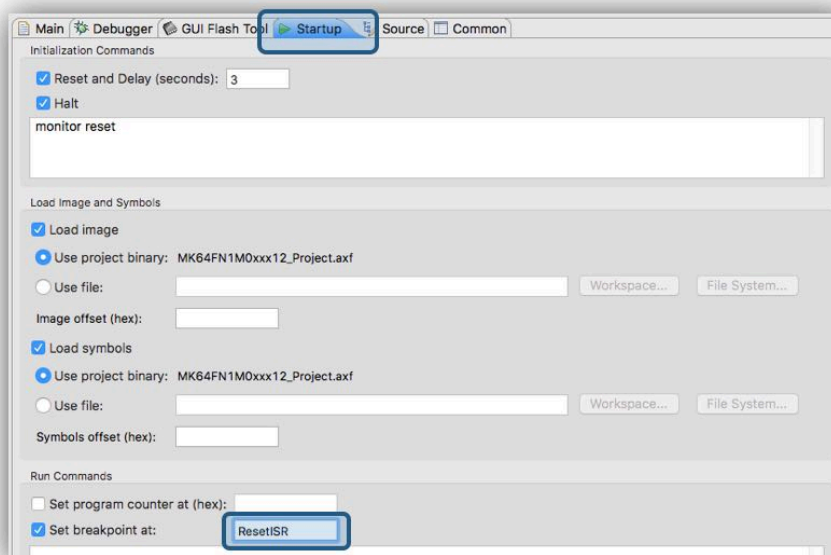
Figure 139. Debug Launch ResetISR Segger

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

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

14.3.4 Debugging pre-loaded binaries (add symbols) and additional images

In a typical debug scenario, a project is built, programmed into flash, and debugged. However, a common requirement may be to debug via a bootloader or debug additional code preloaded (into flash) by another project. For a good debug experience, symbolic information (and source) for additional project code must be made available to the debug environment.

Another requirement may be to load additional images alongside the main application.

You can now easily add symbolic information from additional projects or specify other images for the debugger to load. These files can be added via the *Other Symbols and Images* tab on the launch configuration of a project as shown below.
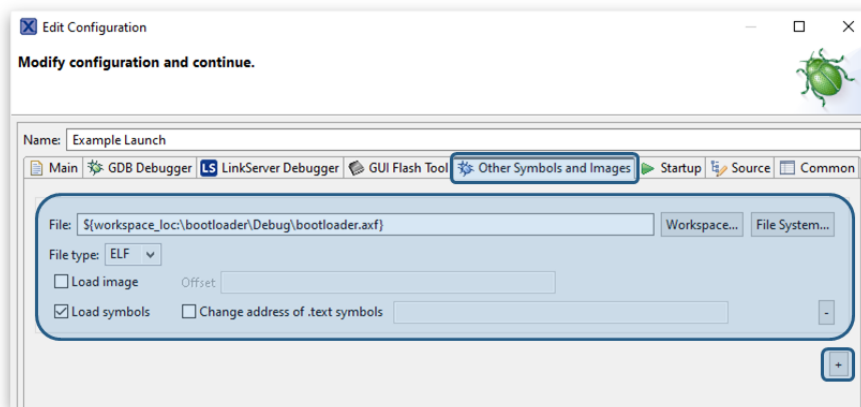
Figure 140. Debug Launch additional debugging data

To add symbolic information from other projects, simply browse to their axf files and select the *Load symbols* option. You can use the default address or set a new address for the text section of the image.

Furthermore, if you require the debugger to program the image to target memory when starting the debug session, select the *Load image* option. This option is also available for images in binary or Intel HEX formats. You can control the address at which the debugger loads the image by specifying an *Offset* hexadecimal value. For ELF and HEX type images, the offset value is added to the load address specified in the image (loadable sections for ELF, records for HEX). For binary files, it is the absolute address at which the image is loaded.
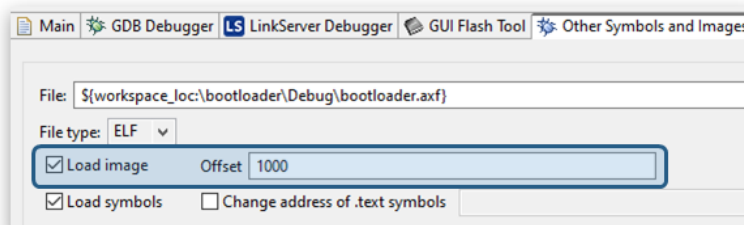
Figure 141. Load image option

Use the + button to specify additional symbolic information or loadable images.

Document feedback

14.3.5 Disconnect behavior

Once the user has completed a debug session, the debugger connection can be terminated via the Terminate button! The exact behavior of the target depends on the particular debug solution.

14.3.5.1 LinkServer

For LinkServer, the launch configuration contains a set of options to control what the target should do when terminated. The default option is for the target to continue running from the current PC value. However, you can change this by selecting a new setting within the launch configuration.
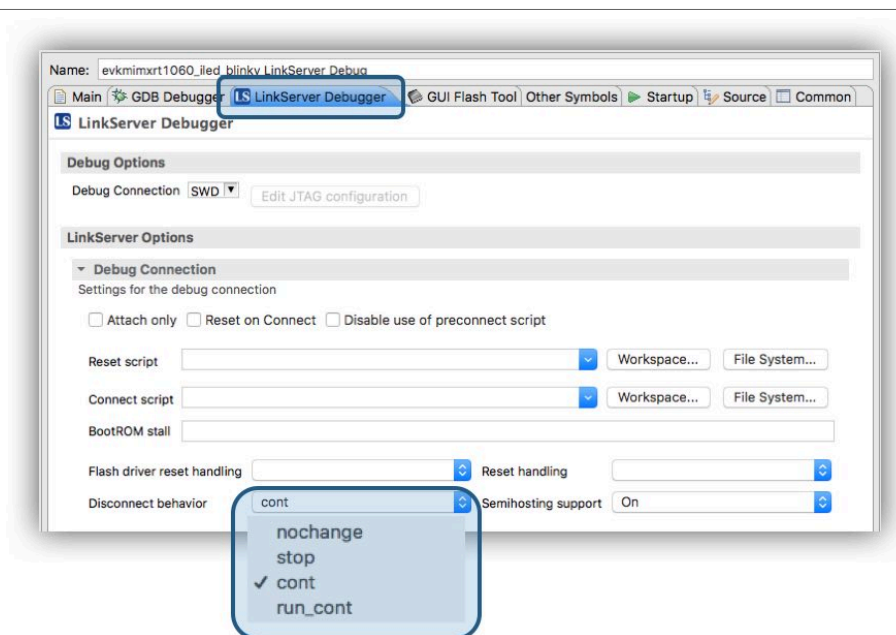
Figure 142. Debug Launch disconnect mode

Where:

- **nochange** - leaves the target in its current state
- **stop** - leaves the target in debug state, that is, halted
- **cont** - the default, either starts the image from its current PC value or leaves it running
- **run cont** - resets the target and lets it run

14.3.5.2 PEmicro

The Terminate button forces the target to halt. Alternatively, for PEmicro debug the IDE supports another option - to disconnect and force the target to run. You can achieve this via the disconnect ⚡ button.

14.3.5.3 SEGGER JLink

The target will *Run* on disconnect by default. You can change the launch configuration option, *Disconnect behavior* to *Halt*, causing the target to halt on disconnect.

14.3.6 Project Flash programming

Launch configuration dialogs now contain a GUI Flash Tool tab. This along with the Advanced GUI Flash Tool and Debug shortcuts provide access to the flash programming capabilities of each of the supported debug solutions.

For each debug solution, the options vary slightly but the presentation is broadly the same as shown below. These options are self-describing.
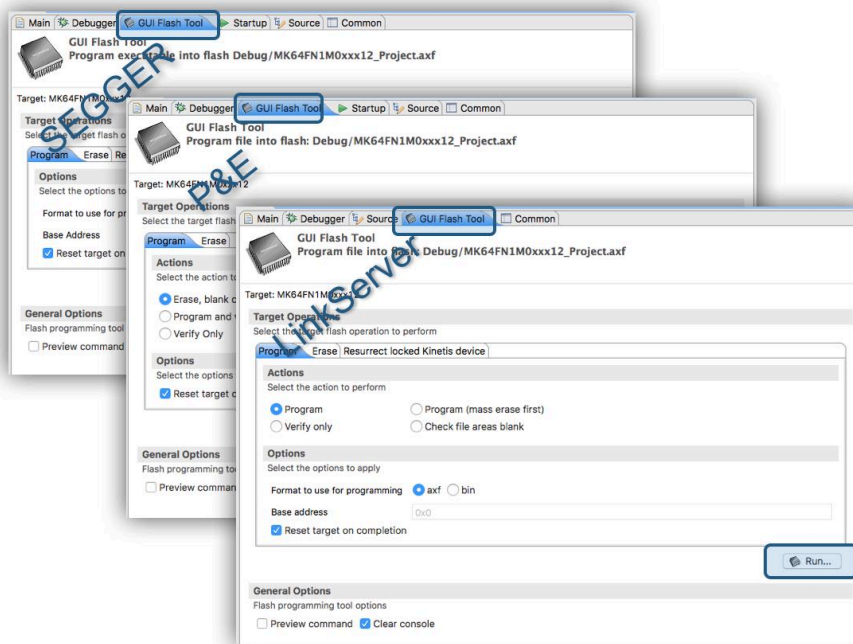
Figure 143. Debug Launch Flash programming

To perform the selected operation, simply click the *Run* button.

Important Note: By default, a launch configuration has *Program* as the default Program action, and *Mass Erase* as the default Erase action. When the user changes the settings, they are stored within the launch configuration of that project and remain until a manual change occurs (or until the deletion of the launch configuration). When using Debug shortcuts they action the current settings within the selected projects launch configuration (or if none exists, create a new default launch configuration) - therefore if the *Program* action is set to *Verify*, a Verify is performed as the Program action.

14.4 Breakpoints

When viewing the source (or disassembly) during a debug session, you can toggle breakpoints by simply clicking/double-clicking in the leftmost side of the source view, typically shown as a light blue column. This is also where the breakpoint symbol appears when you set it. You can do this when the target is paused or running.

Breakpoints (and Watchpoints) also appear in the Breakpoints view. You can also use this view to delete or disable them. If you are using the "Develop" perspective, then by default it is in the bottom left of the MCUXpresso IDE window tabbed with the Quickstart and other views.

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

14.4.1 Breakpoint types

At a basic level, there are 2 types of breakpoints:

- Hardware: these are limited in quantity but can be set on ROM (Flash) or RAM. The debug hardware built into the CPU provides these breakpoints.
- Software: these are implemented by a software instruction *BKPT* and can in normal circumstances only be placed on addresses within RAM (since the underlying code must be changed). These breakpoints can be applied in any quantity. The debugger invisibly places (and removes) them.

Usually, the debugger automatically decides the best breakpoint to use for a particular memory type or circumstance and this is invisible to the user.

Simplistically, software breakpoints are placed in RAM and hardware breakpoints are placed in ROM (Flash).

Tip: On some systems, a bootloader may copy code from ROM into RAM for execution - if a symbol within this code is breakpointed - such as main(), then the debugger may select a software breakpoint since it knows that main() resides in RAM. A problem can arise if the debugger sets the software breakpoint before the bootloader has relocated the code. If this occurs, any software breakpoint is overridden by the relocated code. MCUXpresso IDE includes support for plain load images, - to ensure that this problem does not arise in this case, MCUXpresso IDE forces a hardware breakpoint onto main(). This is not overridden since this breakpoint type makes no changes to memory.

14.4.2 Breakpoints resources

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

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

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

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

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

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

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

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

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

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

14.4.3 Skip all breakpoints

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

14.5 Watchpoints

Watchpoints are Breakpoints for Data and are often referred to as Data Breakpoints. Watchpoints are a powerful aid to debugging and work by allowing the monitoring of global variables, peripheral accesses, stack depth, and so on. The number of watchpoints that you can set varies with the MCU family and implementation.

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

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

From this view, you can locate global and static variables then simply select **Toggle Watchpoints**.
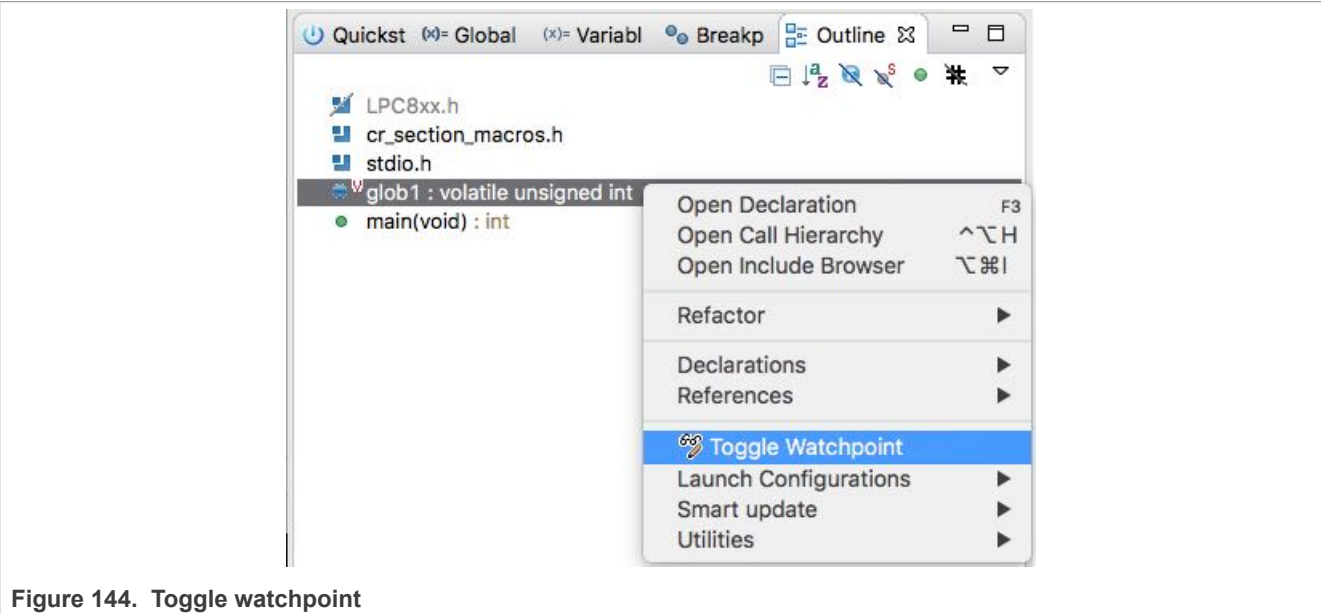
Figure 144. Toggle watchpoint

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

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

Note: To distinguish easily between Breakpoints and Watchpoints within the Breakpoint view, you can choose to group entries by Breakpoint type. From within the Breakpoints view, click the Eclipse Down Arrow Icon Menu, then you can select Group By Breakpoint Types as shown below:
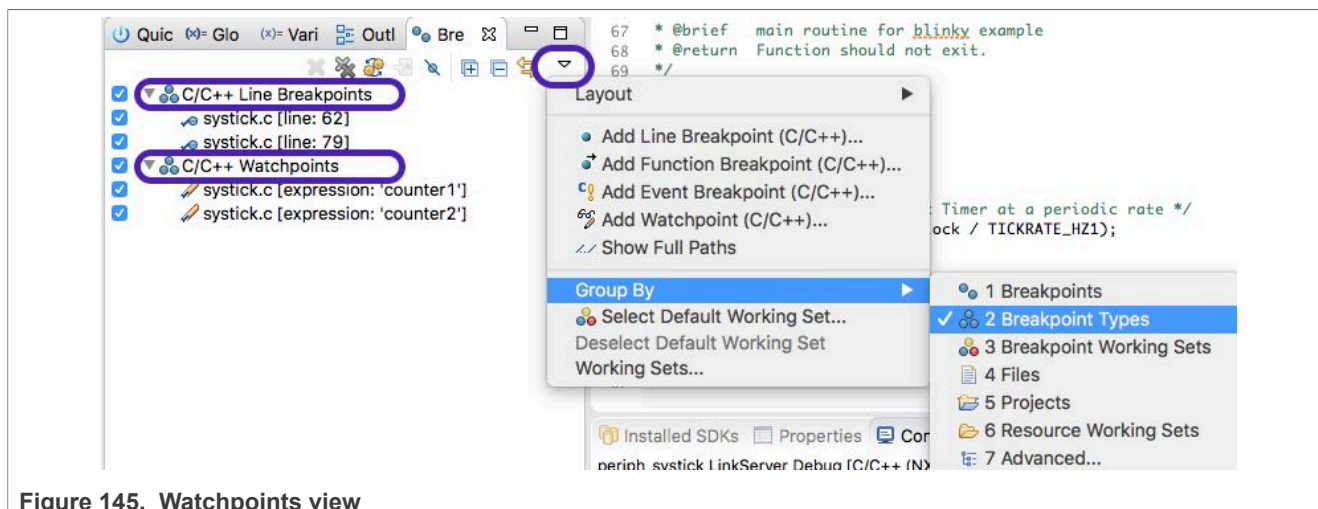
Figure 145. Watchpoints view

As you can see from the above graphic, the option to set a watchpoint is also available directly from the Breakpoint view. When set from here, you are offered an unpopulated dialog - simply entering an address causes a watchpoint to be created, monitoring accesses to that location.

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

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

Note: The implementation of watchpoints results in the CPU performing any monitored access before a halt occurs (unlike instruction breakpoints - which halt the CPU before the underlying instruction executes). When a watchpoint is hit, you can see some 'skid' beyond the instruction that performed the watched data access. If the instruction after the data access changes program flow (for example, a branch or function return), then the IDE may not show the instruction or statement that caused the CPU to halt.

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

14.5.1 Using Watchpoints to monitor stack depth

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

Stacks on ARM-based processors use a Full Descending scheme and so have the potential to descend into areas of memory used for other purposes (typically holding global data or the heap). Establishing the maximum depth of an applications stack can be a challenge, especially since any memory corruption due to excessive stack use may not be immediately apparent. Watchpoints may be used to monitor and trap the stack exceeding a particular depth during execution, enabling positive reassurance that the true stack depth is understood.

The graphic below shows the use of the breakpoint view feature *Add Watchpoint (C/C++) ...* where an address has been selected to watch for the Stack reaching 0x10007D00.
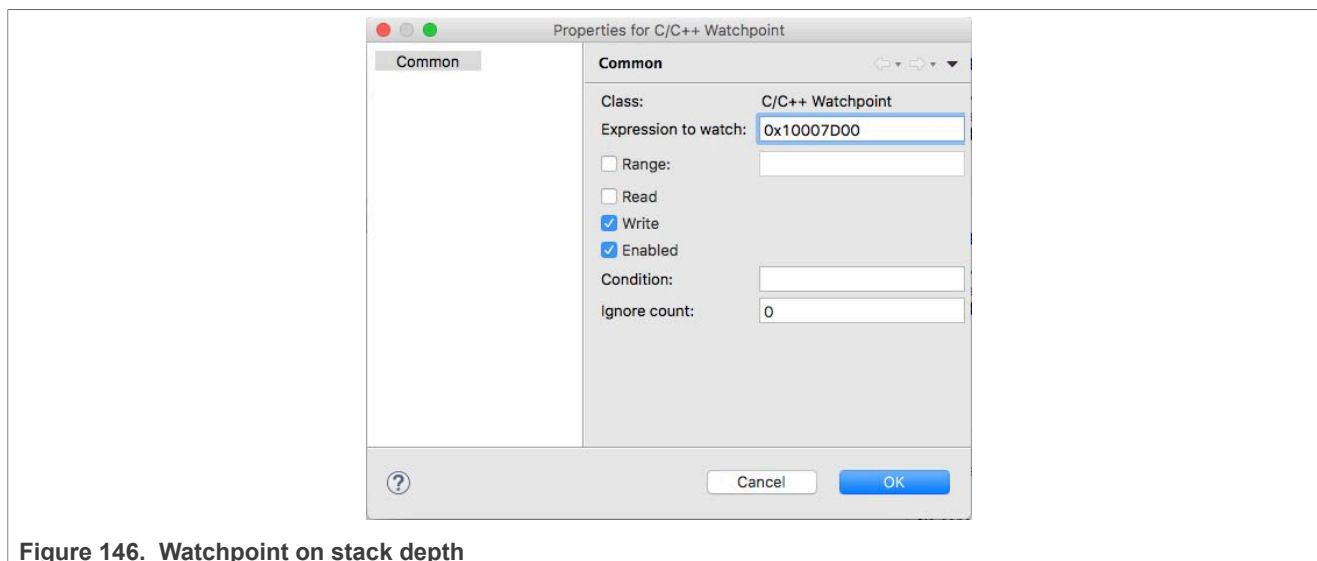
Figure 146. Watchpoint on stack depth

14.6 Registers

The Register view, by default located next to the Project Explorer view, displays the internal ARM CPU registers when the core is halted, that is, when there is an active debug connection but the target is paused. The contents of the registers view vary depending on the nature of the ARM CPU inside the MCU being debugged, however, the base register is available for all MCUs.

The Register list as displayed is made up of the Basic Register set (Core Registers), Fault and Status Registers, Pseudo Registers, and finally Floating point Registers (for Cortex M4/M7, and so on). Since the register set for many MCUs is large, individual register groups can now be hidden if required to reduce screen usage.

Note: For many debug tasks, the values of the CPU registers are of little concern, however when debugging at the disassembly level (and single stepping), these values can be a powerful debugging aid. For an in-depth understanding of the ARM register set for the CPU within your NXP MCU, consult the documentation available from ARM.

Tip: Even when operating in LinkServer None Stop mode, registers cannot be read or written when the target is executing and the register display may appear blank.

14.6.1 Basic register set (core registers)

The basic register set comprises the 16 32-bit core registers of the CPU (r0 - r15), plus the program status register, certain registers have a special function:

- r13 - SP Stack Pointer, this holds the address of the last entry on the stack
- r14 - LR Link Register, this holds the return address for a BL (branch with link) instruction
- r15 - PC Program Counter, this holds the address of the instruction (to be) executed
- xpsr - program status register, this combines the Application (APSR), Interrupt (IPSR), and Execution (EPSR) program status registers, reflecting the state of the CPU
- flags - set by certain instructions performing arithmetic operations (contained within the APSR)

The register set (for a Cortex M4 CPU) is displayed below:
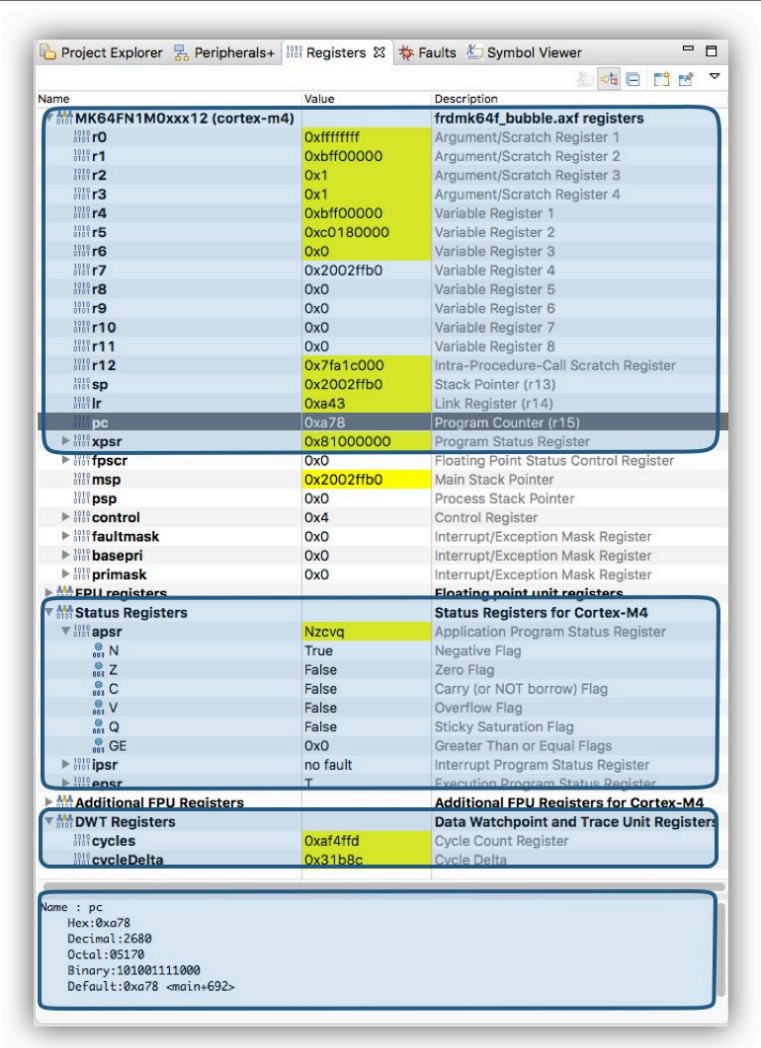
Figure 147. Registers view

Note: in this graphic, the floating point registers have been hidden

Four blocks of registers are highlighted within the graphic.

• Registers r0 - r15 and the xpsr (the components of this are shown below in the status registers)
• Status registers apsr ipsr and epsr, these registers combine together to form the xpsr
  – Certain bit fields such as the CPU flags are expressed alpha-numerically in this view
• Cycles is a memory-mapped register that increments for each core clock tick. CycleDelta is a pseudo register that records the cycles since the last pause (see more below).
• Details view displays the selected register in various formats

When paused, all of these registers can be read (or written). The ability to write values to the registers set is a powerful debug feature but should be used with care.

14.6.1.1 CycleDelta

CycleDelta holds the number of core clock ticks that have occurred since the last time the CPU was paused. For example, if you run from the default breakpoint on main to a breakpoint, *cycledelta* contains the number of clock ticks that occurred while executing this section of code. If a *step* is performed, the *cycledelta* is the

number of clock ticks for the code being stepped. If *stepping* at the instruction level, this value is often 1 because many instructions execute within a single clock cycle.

14.6.1.2 Vectpc

In previous versions of MCUXpresso IDE, the pseudo register VectPC was used to display a value when the CPU has experienced a Hard Fault. This functionality has been replaced by the Faults view.

14.7 Faults

During application development, errors within a program or algorithm may lead to a CPU fault (Hard Fault). These faults include:

- usage fault - such as a divide by zero
- bus fault - such as abort triggered by a memory controller
- mem manage - such as a fault triggered by a memory protection unit

Such errors can be difficult to locate, so to aid the debugging of such problems MCUXpresso IDE incorporates a *Faults view*.

If a fault occurs, the new Faults view automatically appears and the CPU halts (LinkServer). The view offers a set of features including identifying the nature of the fault, the location (link) of the code that caused the fault, and the location (link) of the function that called the 'fault' function.

Note: for non-LinkServer debug probes, a fault may leave the application running within the default fault handler (usually implemented as a *while(1)*), therefore a pause might be necessary to see that a fault has occurred.
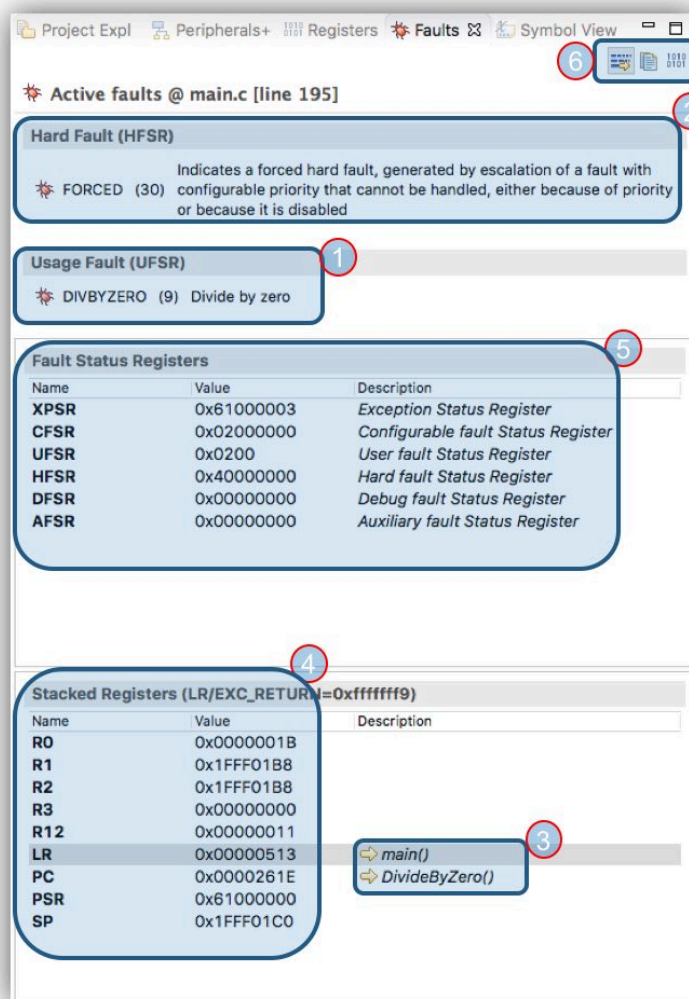
Figure 148. Faults View major features

This view is titled with the source file and line number that caused the error. The view contains the following features:

1. The Fault that occurred - in this example, a Usage Fault of type Divide by Zero
   - certain faults may need to be enabled within the CPU, for example Divide by Zero is enabled in the Cortex M4 Configuration and Control register
2. The Action that was taken - in this example a Hard Fault was generated
3. Links to the source of the fault function and its caller function, located from stacked registers
4. Values of the registers automatically stacked on entry to the fault handler
5. Fault status registers that may offer further information
6. Additional options including:
   - Button to cause disassembly to be opened in parallel with sources (3)
   - Button to copy the fault details to the clipboard
   - Button to display all fault registers and descriptions rather than the

In some circumstances, a hard fault might be caused early on during the initialization of the system before the breakpoint on main() is hit. This may mean that the fault is triggered before the debugger can take action to display the faults view. If this happens, try setting a breakpoint in the startup code - this might then allow your

code to load without the hard fault being triggered. You should then be able to single step/run until the cause of the hard fault is hit. You will then see the Faults View displayed.

Tip: *If a repeated fault occurs that is difficult to debug, instruction trace could be enabled (when supported by the MCU) and the captured trace dumped when the fault is trapped. Looking back at the captured instructions should help find the reason for the fault condition. See the MCUXpresso IDE Instruction Trace guide for more information.*

Note: Typically a Fault on an embedded system is fatal, however, this view also assists users in developing and testing fault handlers for recoverable fault situations.

14.8 Peripherals

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

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

Figure 149. Peripherals view

In this view, each peripheral is listed along with its value, base address, access, and a brief description. The view also exposes the inner peripheral registers and offers bit field enumerations to greatly simplify both reading existing configurations and setting new values.
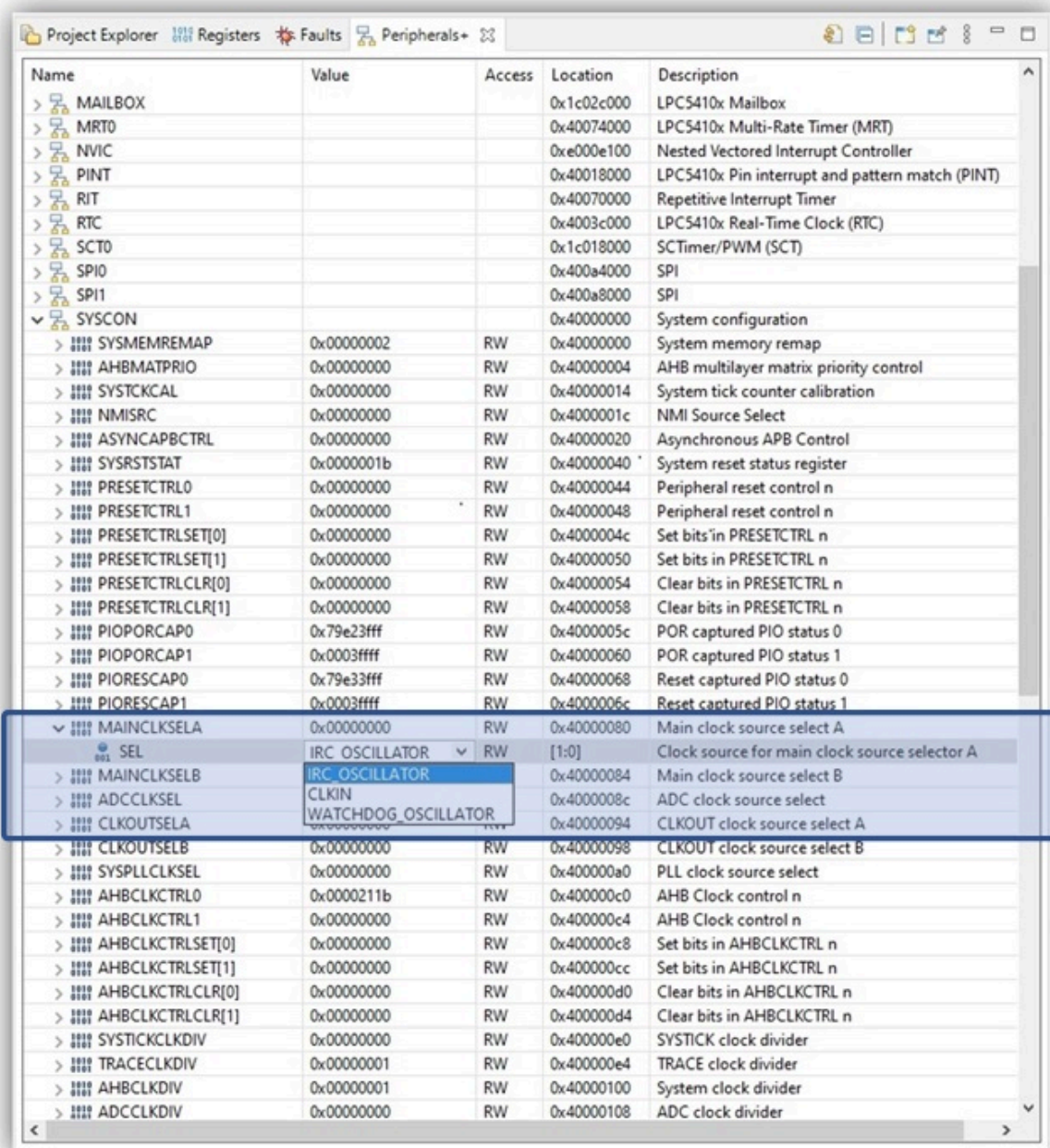
Figure 150. Peripheral register view modifying bit field value

Important Note: When an MCU powers up, many peripherals are unavailable because they are unpowered/ not clocked. Attempting to access a peripheral in this state fails, and the peripheral simply displays them in red. Certain peripherals may be partially available, while unavailable sections are again displayed in red. Entries that have changed are displayed in yellow.

Tip: Even when operating in LinkServer None Stop mode, peripherals cannot be read or written when the target is executing. The main peripheral display may appear blank when the target is executing regardless of LinkServer mode.

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

The view also lists in the main menu the device memory regions. If these memory regions are selected, a standard hex memory display is created. Memory regions are not peripherals in the normal sense but are included here so their memory space can be easily displayed.
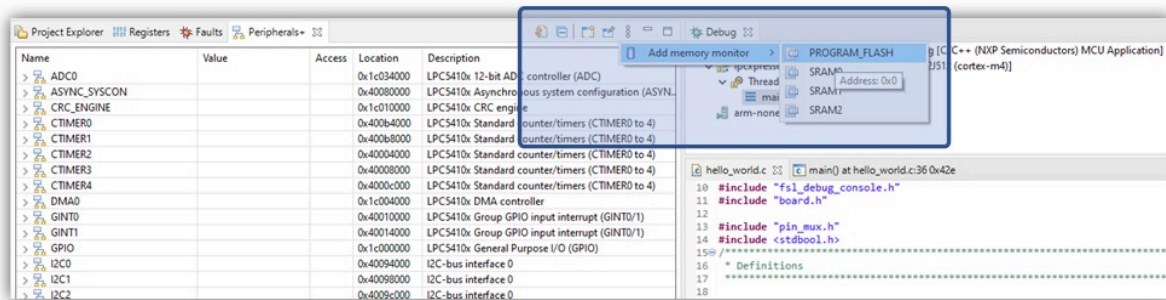
Figure 151. Peripherals view memory regions

14.8.1 Custom SVD file

Users can specify a custom SVD file location inside a project. You can achieve this from *Project Properties -> Run/Debug Settings -> MCU Settings -> SVD Selection*. The file is used by the Peripherals+ view when debugging that project.
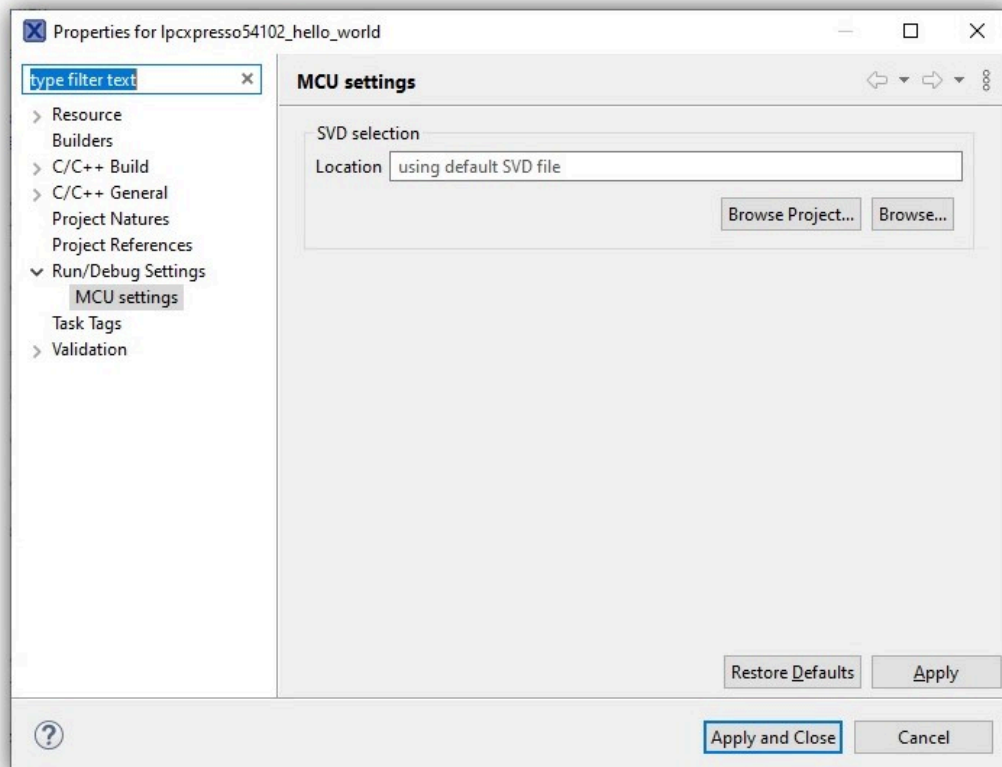
Figure 152. SVD custom selection

The SVD file can be imported from:

- the selected project
- a custom file location

In the default case, where this file is not specified, the SVD is loaded from the associated SDK.

14.9 Offline Peripherals

MCUXpresso IDE provides a way to inspect the peripheral registers without the need for an active debug session. Registers are shown with the fields, but without the actual values. Instead, the view shows the reset value for the registers.
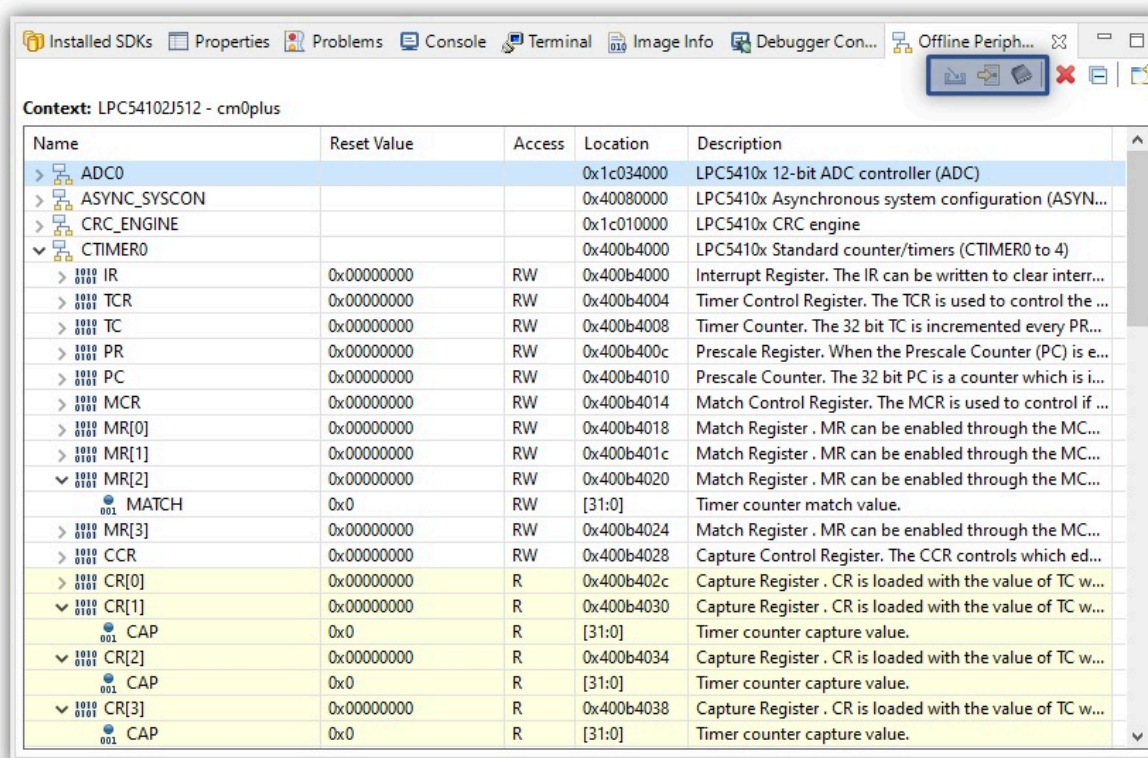
Figure 153. Offline Peripherals view

The view provides three ways of importing peripherals:

- from a local file
- for the device/core used by the selected project
- from a list of available devices

Tip: *Holding the mouse over the bit field shows a tooltip that includes the detailed description for the current value and also the descriptions for the other possible values.*
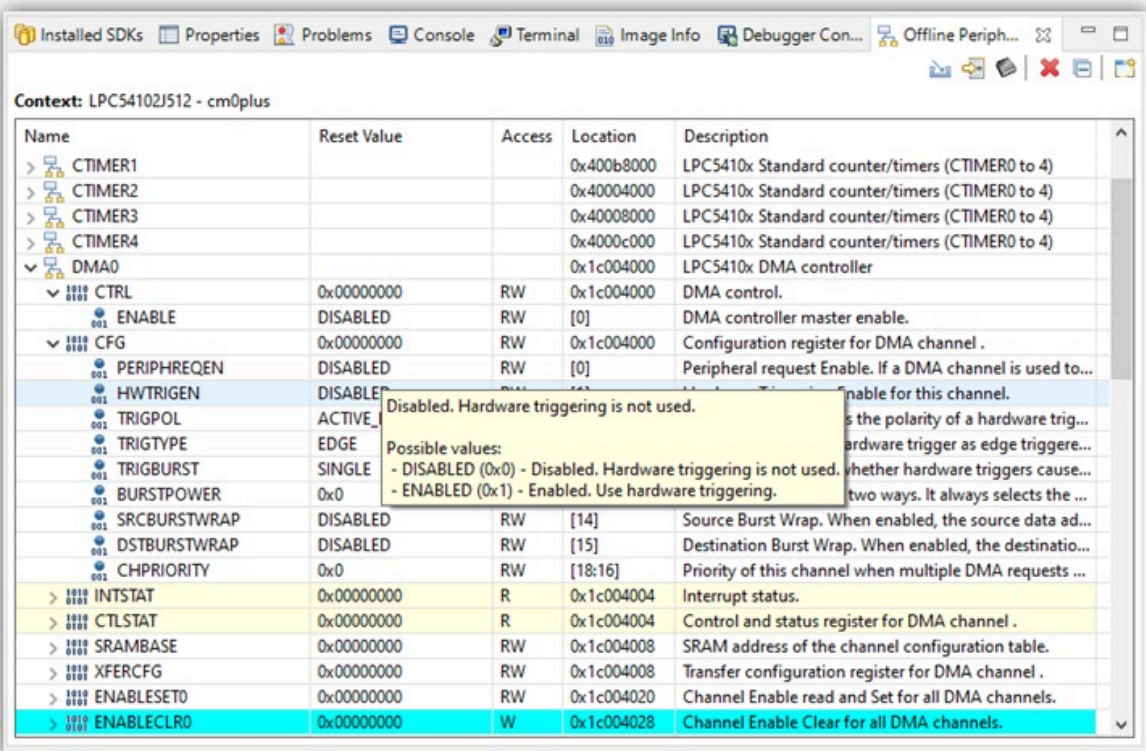
Figure 154. Offline Peripherals view bit field information

14.9.1 Loading custom SVD file in Offline Peripherals view

To load the custom SVD file, which was set into Project Properties, push the "Load peripherals for device used by selected project" button. The location from where it was provided can be found in Context.
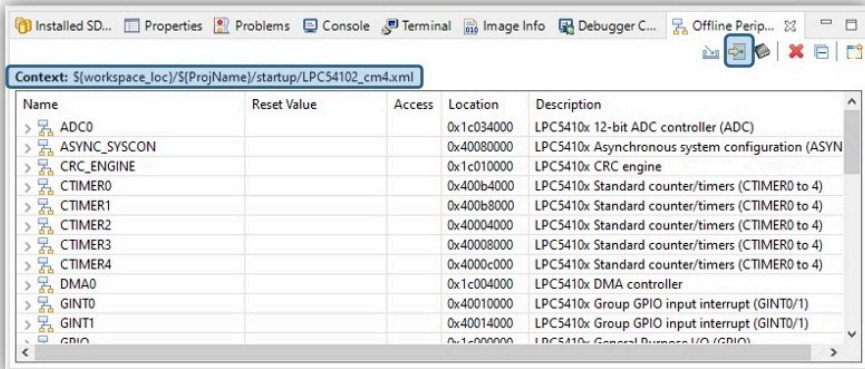
Figure 155. Load custom SVD file

14.10 Global and live global variables

Global and Static variables are stored within system RAM memory and can therefore be accessed by the debug chain (read and potentially written) while an application is both paused and running.

Note: The ARM processor inside the NXP MCU uses a load-store architecture, this means that a global variable must be read (loaded) from memory and then written back by the processor (if changed). The value of the variable displayed corresponds to the value in memory and this may potentially be different from the value held by the processor. Modern MCUs execute millions of instructions every second, so any variable observed while an application is running may have been changed many times from the value displayed in the view, therefore take care that this is understood before attempting to change a variable value within the Global variable view.

This view can be populated from a selection of the global variables of a project. Simply click the "Add global" button to launch a dialog:
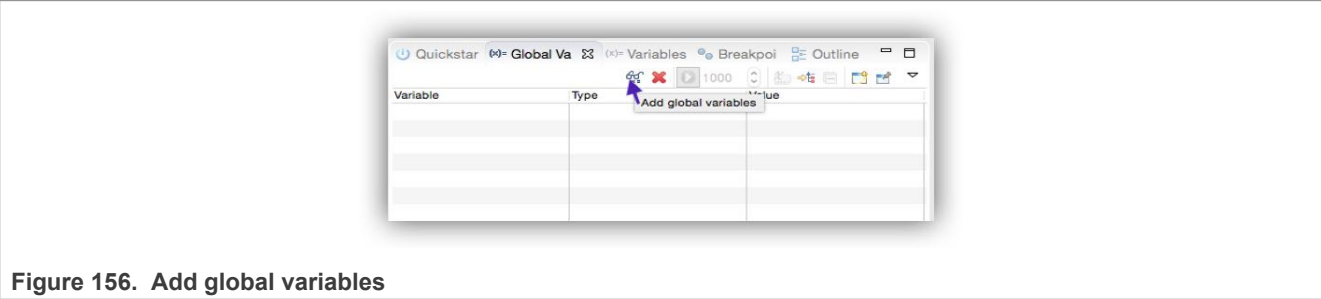
Figure 156. Add global variables

This then displays a list of the global variables available in the image being debugged. Select the ones of interest via their checkboxes and click OK:
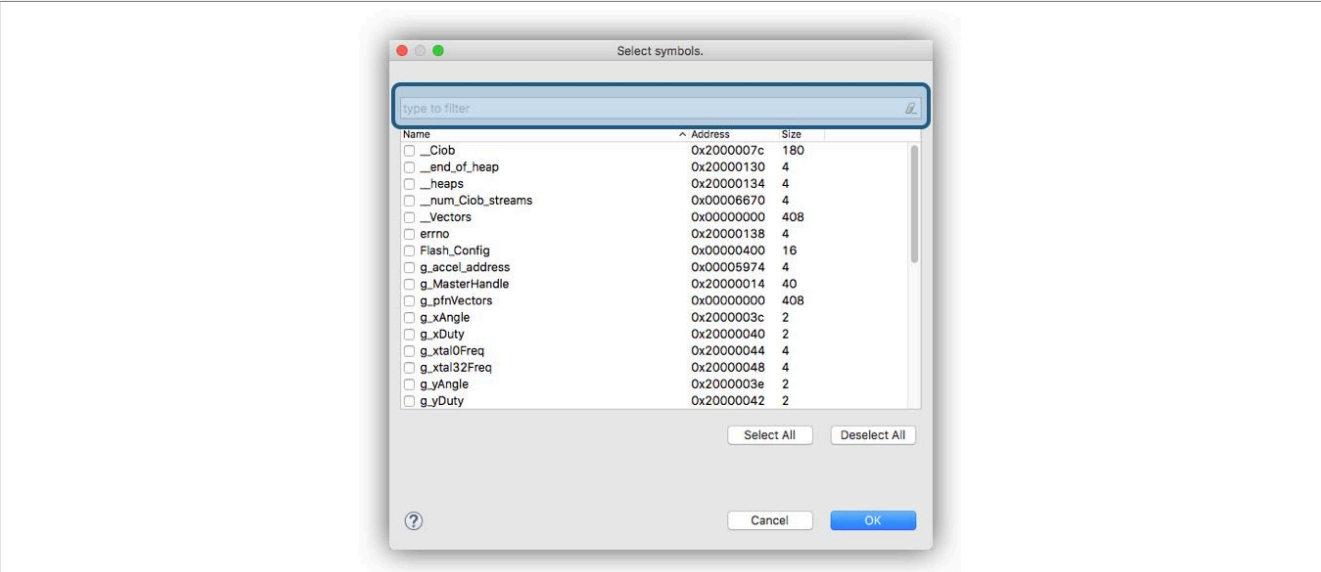
Figure 157. Global variable selector

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

Once selected, the chosen variables are remembered for that occurrence of the dialog.

For all supported debug chains, there is now the capability to view global variable values when the debug target (MCU) is running. When this feature is used, these are known as "**Live Variables**".

For variables to be "Live":

- The target must be running
- The enable/disable (run) button clicked (as shown highlighted below)

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

Figure 158. Global variable display

Also available is the ability to enter an expression (using standard C notation) or symbol. The expression is evaluated and the address displayed in the Address column.

Figure 159. Global variable display expression

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

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

The usefulness of **Live Variables** reduces as the number of Globals monitored increases, and ultimately there is a limit as to how many variables can be updated at the selected frequency. However, a complex list of variables can be monitored if required. For example:
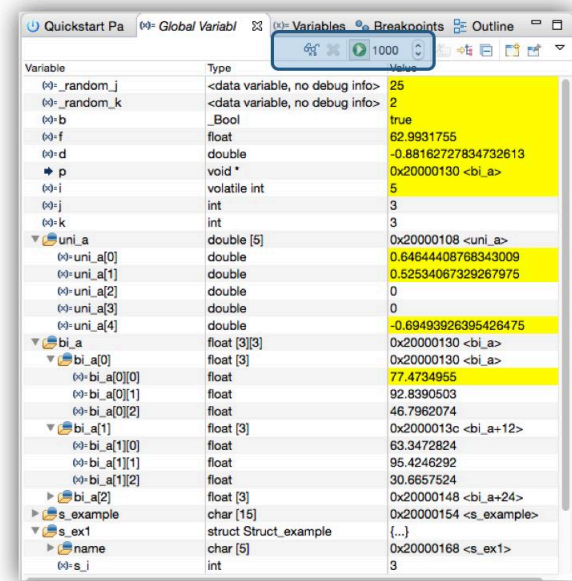
Figure 160. Global variable display complex

14.11 Live global variable plotting

In addition to displaying *Live Variables*, the IDE can also trace (sample) their values for plotting as graphs, logging, or calculating statistics.

By default, it is assumed that variable values may be traced but alternatively, their values can be displayed in a details view via a right-click menu selection.
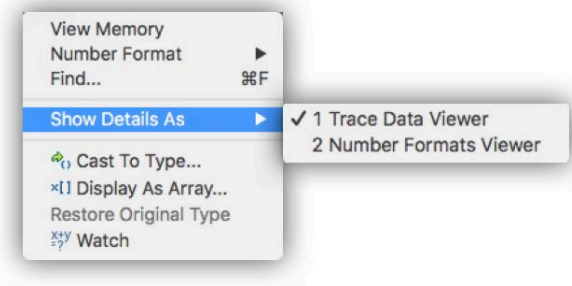
Figure 161. Live Variables details view

Variables can only be traced if they have first been added to the Global Variable panel as discussed in the previous section. The selection of variables to plot is made by clicking to highlight the variable of interest.

Note: Once a variable has been selected, the timebase (uptime) begins and variable values are sampled and displayed. If additional variables are selected, their values join the display at the current uptime. If a variable is unselected, its values are no longer sampled and displayed. If however, it is selected again within the same debug session, it is displayed along with any previously captured values. During any period it was not selected, its values show as zero.

Tip: *If the display is paused, data is still captured, but the new values are not displayed, this can help detailed viewing of the data. Once un-paused, the captured data is added to the display.*

Note: If the target is paused, time (x-axis) continues to advance although the display is not updated until the target is resumed.

14.11.1 Live Global Variable graphing details

In the example below, two variables have been added to the Global variable view and both have been selected for tracing.
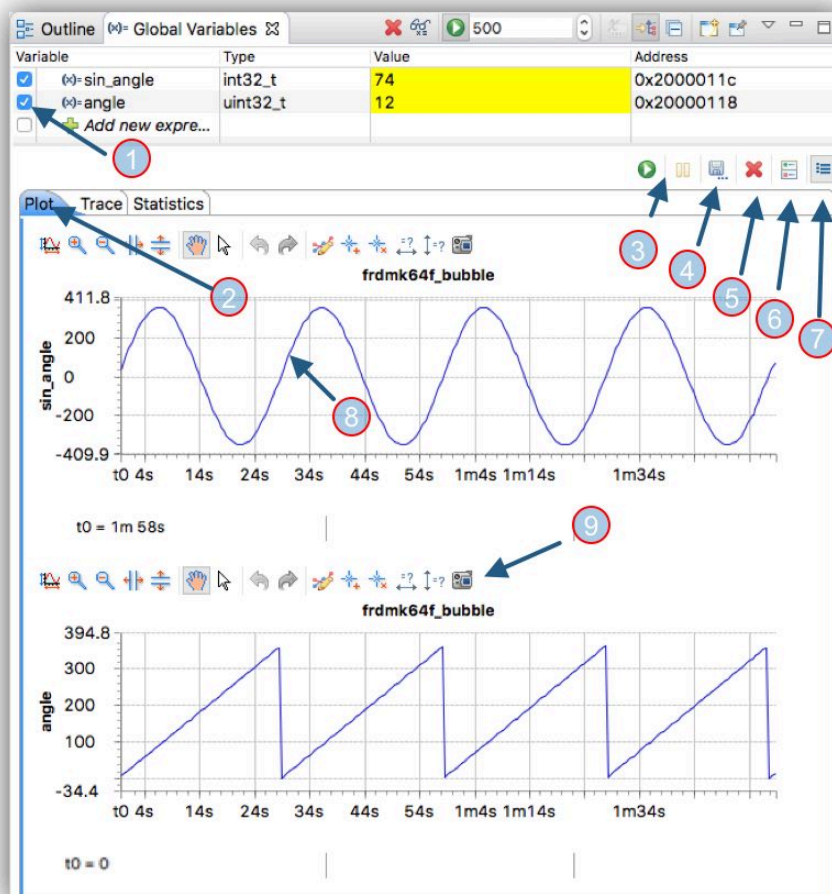
Figure 162. Global variable graphing major features

The highlighted features are explained below:

1. Selected variables. Click the checkbox to select a variable for plotting
   - Once selected, the variable exists in the internal database of values and remains until the debug session is terminated (even if it is later unselected)
2. Plot types: the traced data may be viewed in three ways:
   - Plot - display as a graph over time
   - trace - log the values
   - Statistics - calculate statistics for the traced values (max, min, average)
3. Resume and Pause: Click Resume to start plotting variables. Click to pause the graph display updated. Variables values are still captured but the screen does not update
4. Save: Click to save the captured data.
   - The size of the PNG is proportional to the size of the global view. Therefore, for more detail, increase the size of the global view before saving
   - This button offers the option to save each of the Plot types: Plot (PNG), Trace (TSV), Statistics (TSV)
5. Clear Data: Display: Click to discard any traced date

6. Show or Hide the Graph Toolbar
7. Multiple/Single Graphs: Click to toggle the display between separate graphs for each variable and all variables plotted on a single graph
8. Click the graph to see the X, Y coordinates for the selected point
9. Graph Toolbar - explained below

Clicking the button marked as (7) combines individual graphs into a single graph view.
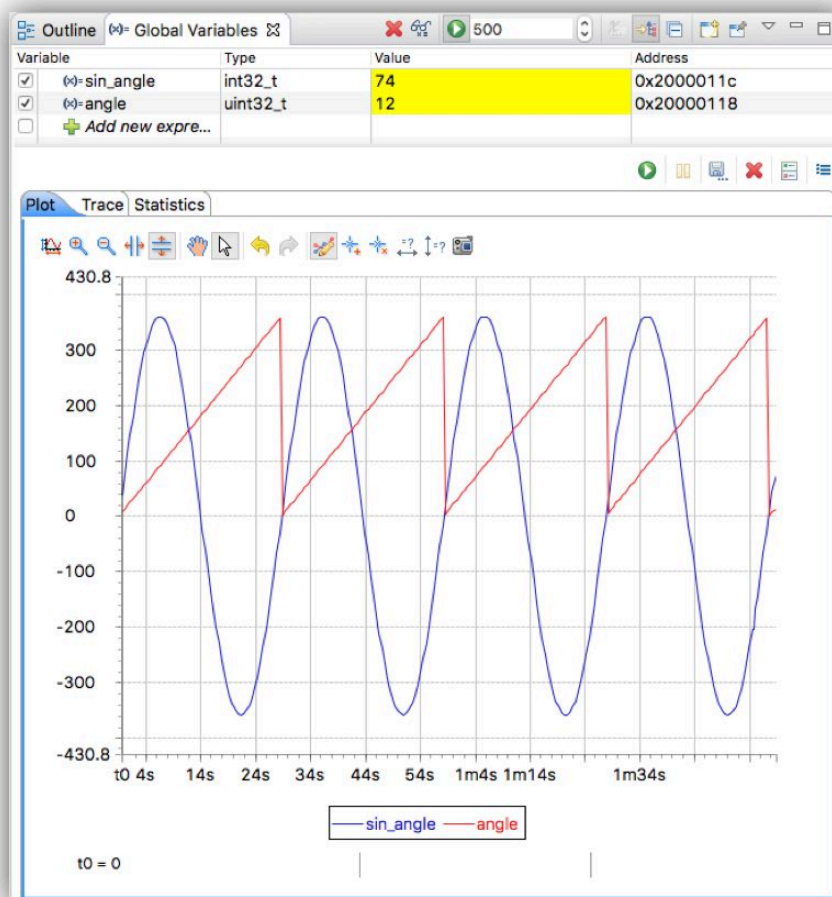
Figure 163. Multiple global variables on a single graph

Each graph view has an optionally visible Toolbar (6). The annotated image below shows a magnified version of the Toolbar.
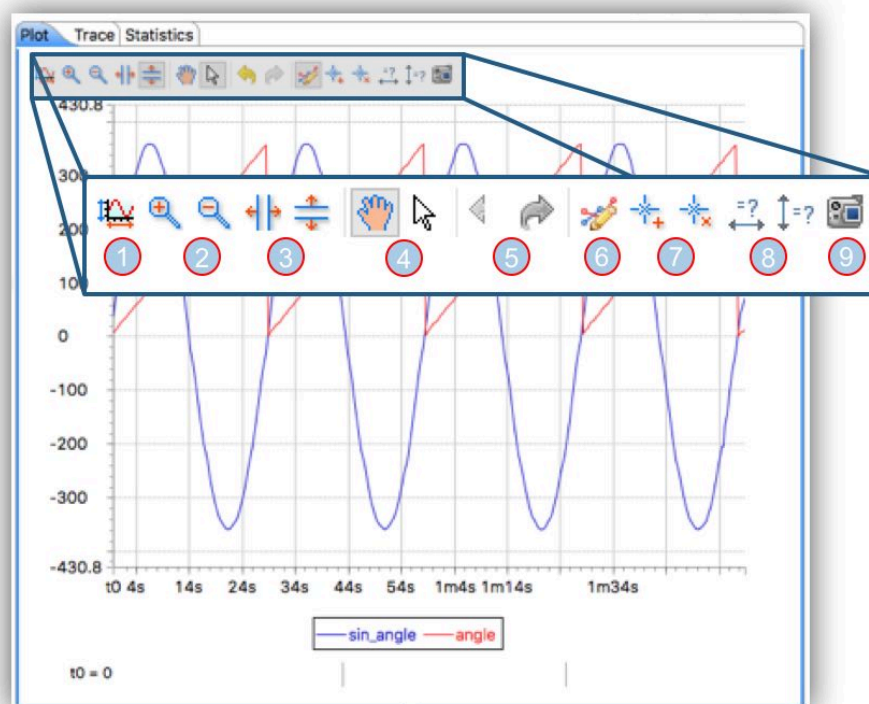
Figure 164. Global variable graph toolbar

Where:

1. Autoscale the display to show all of the data
2. Zoom In and Out: Select the desired button and then click into the area of the graph where zoom is required
3. Zoom Horizontally and Vertically: Select the desired button and then select and drag within the graph to perform the desired zoom
4. Panning and None: Select Panning to click and drag a zoomed display. Select None to prevent interaction with the graph
5. Undo and Redo: Click these buttons to cycle through previous actions
6. Add a Legend (shown)
7. Add and remove Annotations. Annotations can be named and snap to a plotted point and display its value
8. Measure Horizontally or Vertically: Click and drag to snap between plotted points to measure the value of their separation
9. Save the graph as a PNG file

14.12 Heap and Stack view

Located by default in the MCUXpresso IDE Develop perspective, along with the Memory view at the bottom right of the perspective.

One of the common issues within embedded system development is allocating the appropriate memory for heap and stack usage. The Heap and Stack View offers the ability to monitor heap and stack usage within their allocated regions of memory. The View allows the monitoring of heap usage in real time (while an application is running). However. since the value of the Stack is held within a processor register, Stack usage can only be updated when the application is paused.

The Heap and Stack view displays usage with respect to the configured heap and stack sizes as set within the Projects Properties at: *C/C++Build -> Settings -> Manager Linker Script -> Heap and Stack placement*

Figure 165. Heap and Stack view

This view automatically updates when the target is paused. To enable updating of the heap usage when the debug target is running, click the *Run* icon at the top of the view to enable or disable updates to the view. The frequency of the updates can be set between 500 ms and 10 seconds.

Tip: Although real-time monitoring of the stack is not possible, a watchpoint could be used to force a target halt when an access to a particular stack depth is performed. See further details in the section on Advanced heap and stack placement

The symbols used to generate this view are created by the Managed Linker Script mechanism. However, other symbols can be substituted if required via the workspace preferences as shown below:
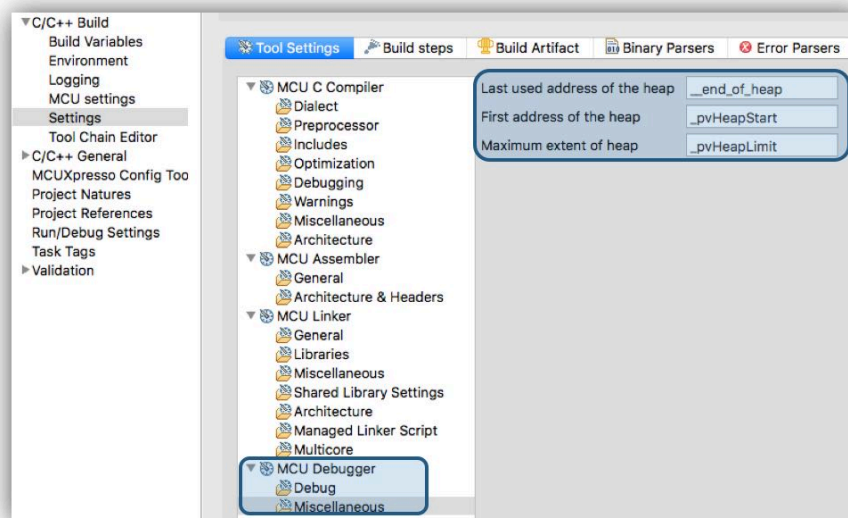
Figure 166. Heap and Stack view symbols

Tip: As a guide the memory usage % display is colored green when more than half of the available memory is free, then changing through yellow to red if more memory is used

14.13 Additional debug features

14.13.1 Local variables

Situated alongside the **Quickstart** panel, the local variable view displays the local variables in scope when the target is paused. Typically, local variables are held within processor registers and so they cannot be accessed when the processor is running. From this view registers can be viewed and their values edited if required.
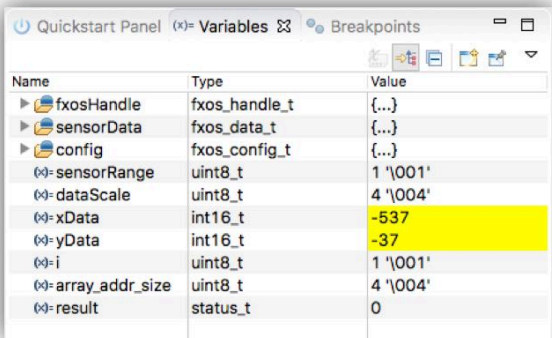
Figure 167. Local variables view

14.13.2 Disassembly view

The Disassembly view allows the code of an application to be viewed at the assembler level (as generated by the compiler).

The view can be enabled (if required) via the *Instruction Stepping button* within a debug stack view. This button has two functions, in that it both spawns the view and also switches stepping mode from source level to assembler level. Assembler level stepping is typically used in conjunction with the Registers view to examine the detailed behavior of short pieces of code.

Stepping mode can be returned to source level by re-clicking this button.
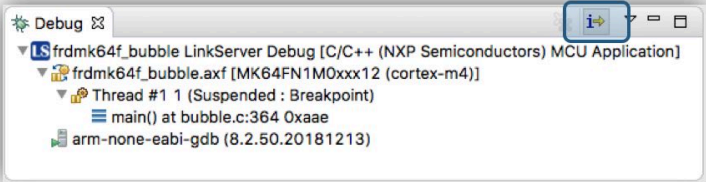
Figure 168. Disassembly enable

Once enabled, the disassembly view displays the low-level assembler instructions usually from the current PC.
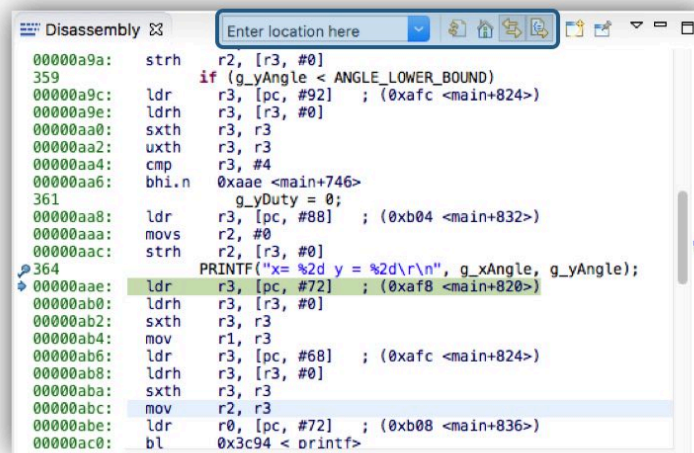
Figure 169. Disassembly view

The view has a number of features including:

- Setting a new address to view
- Refreshing the view contents (this might be useful if the underlying code may have changed)
- The linking and unlinking from the current debug session (PC)
- The intermixing of source code lines with their related assembler instructions
  - The usefulness of this feature decreases as compiler optimization increases

14.13.3 Memory view

Stacked by default in the MCUXpresso IDE Develop perspective, along with the Heap and Stack view. The memory view allows debug target memory to be explored in a traditional manner. The view can be populated with target memory regions via the Peripherals or by entering required address values.
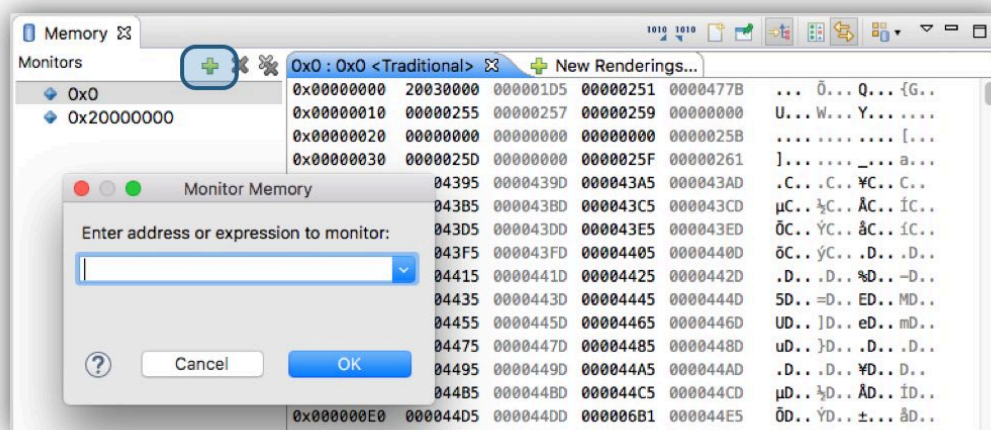
Figure 170. Memory view

Note: Although it is technically possible to populate this view while the target is running, this mode of operation is not currently supported. A particular memory of interest can be monitored *live* via Global variable expressions if required.

15 Configuring a project

When a project is imported or generated using a wizard, there are many configuration options available at creation time. However, once a project has been created or if a project is shared by other means, then there still may be a requirement to make changes.

The range of possible project changes is almost infinite but below we discuss several common changes that may be required and the potential ramifications that may be encountered. Note that many of these changes can be started from the Virtual nodes of a project.

Note: This section only discusses a few of the common changes that may be made. Also see the sections on Memory configuration and linker scripts Flash drivers Library support and the additional Config Tool documentation for a more comprehensive description of the options available.

15.1 Changes available via Quickstart Quick Settings

MCUXpresso IDE provides quick access to a range of project settings via the **Quickstart** Panel as shown below:
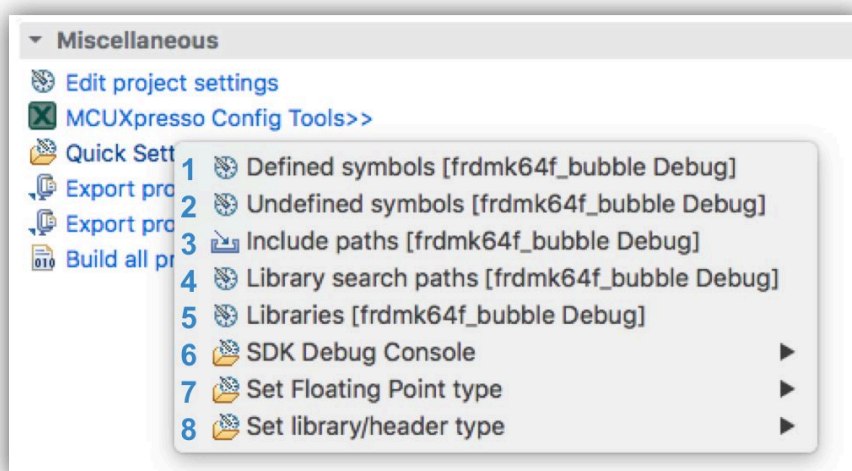
Figure 171. Quick Settings

Note: These settings apply to the active build configuration of the selected project only and simplify access to commonly used settings normally accessed from *Properties -> C/C++ Build -> Settings* Also note that Quick Settings changes may be made to multiple projects if more than one project is selected (where their settings are compatible).

Tip: The current setting for Debug Console, Floating Point, and Library type is shown.

1. Defined symbols - select to edit the (-D) symbols
2. Undefined symbols - select to edit the (-U) symbols
3. Include paths - select to edit the (-I) the include paths
4. Library search paths - select to edit the (-L) the library
5. Libraries - select to edit the (-I) the linker libraries search
6. SDK Debug Console - select the SDK Debug Console's PRINTF output to be via UART or to redirect via the C libraries printf function
   • Selecting printf increases the size of the project binary compared to the UART output
   • For semihosted printf output to be generated, the project must be linked against a suitable library

- For more information, see the section on <u>Semihosting and the use of printf</u>
7. Set Floating Point type - select to switch between the available Floating Point options
    - For more information, see the section on <u>Hardware floating-point support</u>
8. Set Library/Header type - select to switch the current C/C++ Library
    - For more information, see the section on <u>C/C++ library support</u>

15.2 Project settings

Many features of a Project can be viewed (and edited) via Virtual Nodes. Project Virtual Nodes are contained within a Project structure and provide virtual folders to display and allow the easy editing of project settings.
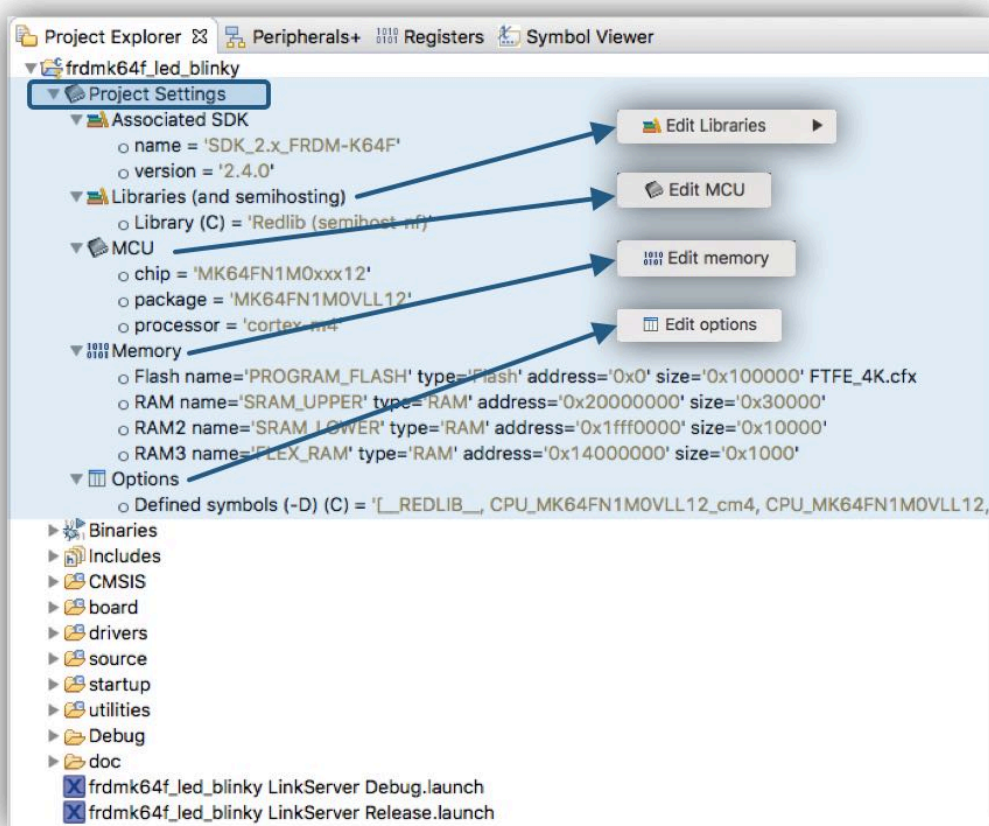
Figure 172. Project settings

These are automatically generated for any project and provide a quick way to view many key project settings. In addition, a right-click these nodes provides direct options to edit the associated settings that otherwise require many more mouse clicks to reach.

15.3 Changing the MCU (and associated SDK)

All projects are associated with a particular MCU at creation time. The target MCU determines the project memory layout, startup code, LinkServer flash driver, libraries, supporting sources, launch configuration options, and so on, so changing the associated MCU of a project should not be undertaken unless you have a total grasp of the consequence of this change.

Therefore rather than changing the associated MCU of a project, it is strongly recommended that instead a new project is generated for the desired MCU and this new project is edited as required.

However, on occasion, it may be expedient to reset the MCU (and associated SDK) of a project and this can be achieved as follows. From the project virtual nodes, select *Edit MCU*.
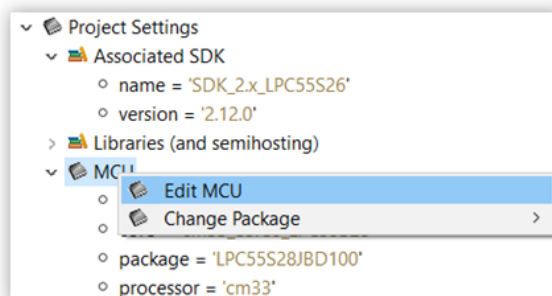
Figure 173. Edit MCU

You are then presented with the MCU Setting dialog (as below)
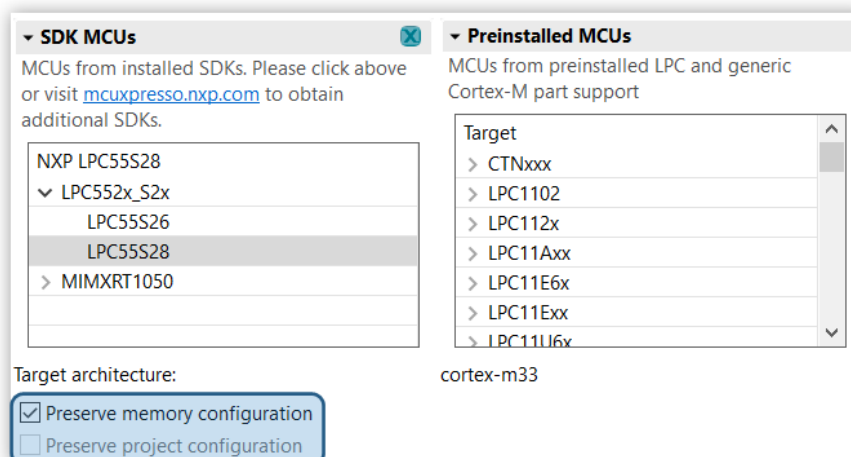
Figure 174. Select MCU

From here, an alternative MCU can be selected but note that two checkboxes must be set as required **before** this is done:

- Preserve Memory Configuration - if set (the default), the original project memory settings is preserved, otherwisem, the MCU setting for the chosen MCU replaces the original settings
- Preserve Project Configuration - if not set (the default), the new MCUs configurations (such as Cortex Architecture) replaces the original settings

When the new MCU is selected, a warning dialog as below is generated:
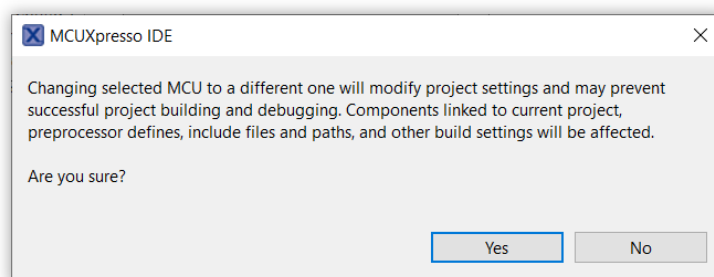
Figure 175. Select MCU warning

Project changes are only made if *Yes* is selected and *Apply* and/or *Apply and Close* are then further clicked to close the *Properties* dialog.

The actual changes that are made inside the project depend on a few more user inputs asked before completing the entire process:

1. Confirm selection of the new board, new device package, and new core associated with the project.
2. Allow removal of the SDK components associated with the old MCU. Only SDK components that have an associated component to the new MCU are actually removed at this step.
3. Allow addition of SDK components associated with the new MCU. Only SDK components that had an associated component to the old MCU are actually added at this step.

Note: Back up the original project before initiating the change of device process.

15.3.1 Confirm device information

This step allows the selection of the new board, new device package, and new core. These selections depend on the SDK (or no SDK if dealing with preinstalled part support) that is associated with the new MCU. An MCU can be fitted on multiple boards, comes in different packages and might be a multicore device.
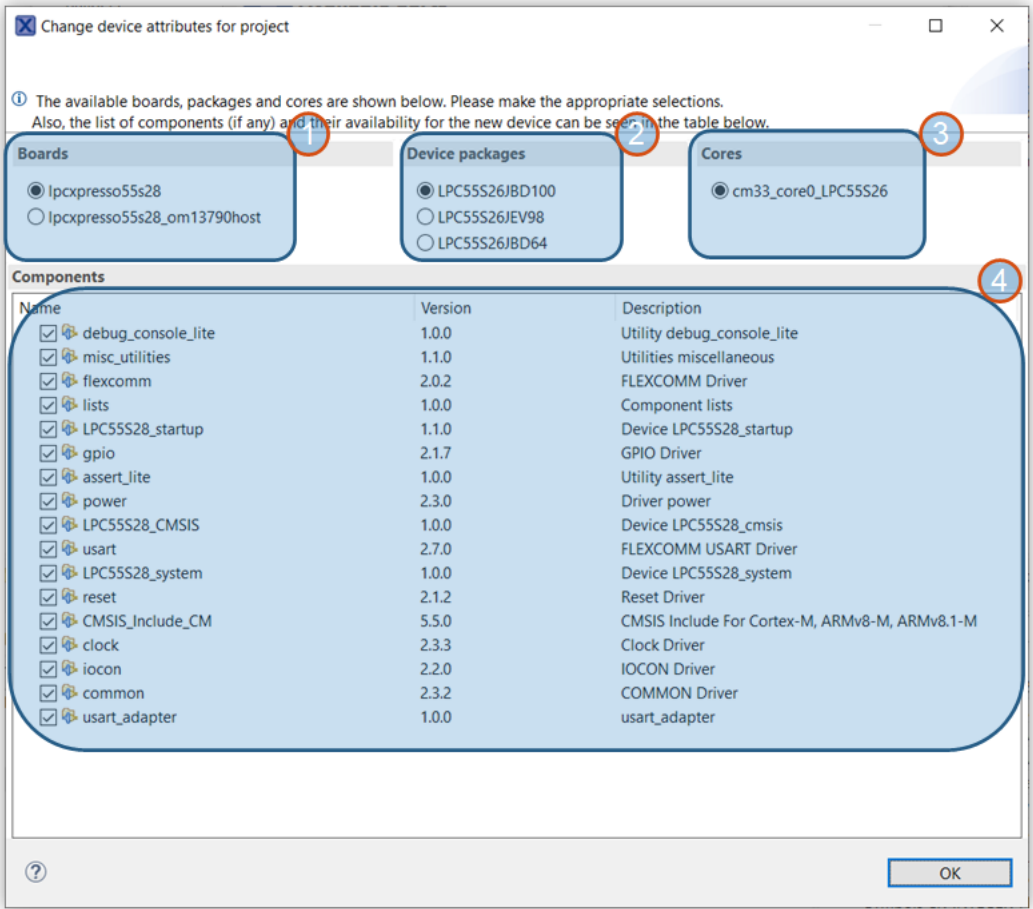
Figure 176. Change device attributes (same SDK)

In the above screenshot, the following sections can be highlighted:

1. Available boards
2. Available device packages
3. Available cores
4. The list of SDK components found inside the project. The table contains a read-only list of checkbox items. If an entry is ticked, it means that the SDK component associated with the old MCU has an associated SDK component for the new MCU. Tooltips offer more details about the old-to-new mapping. All SDK components that are selected inside the table are "migrated" (that is, old components removed, new components added) during the process of changing the device.
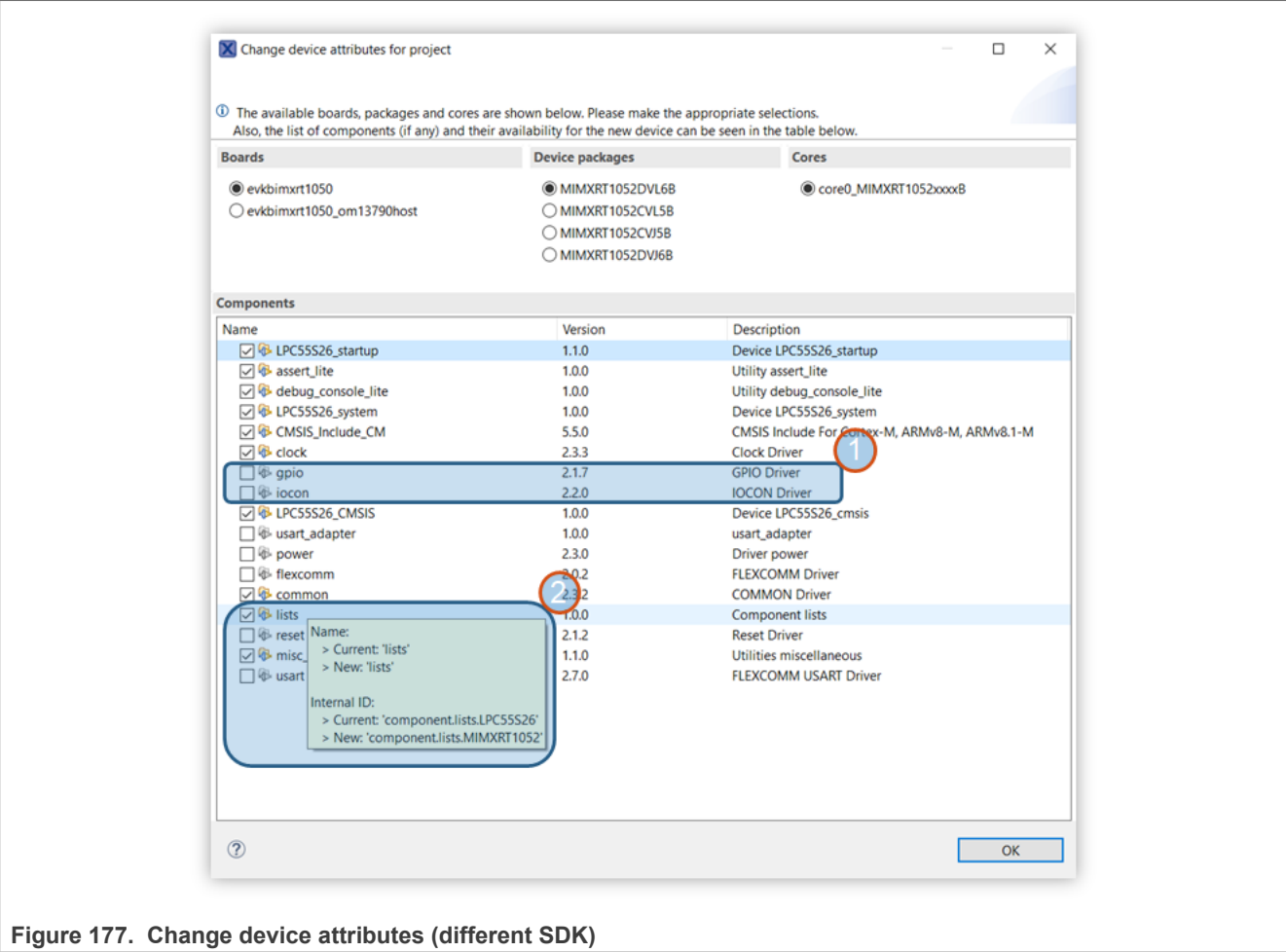
Figure 177. Change device attributes (different SDK)

We can see in the above screenshot some components that are not going to be migrated (1). This is because the IDE was unable to match the old internal SDK component ID to an ID associated with a new SDK component for the new MCU. The tooltips (2) offer some insights about the actual migration.

When switching from an SDK-supported part to a preinstalled part, all SDK components information is lost from the project description. However, no source files are removed/changed/added along the process.
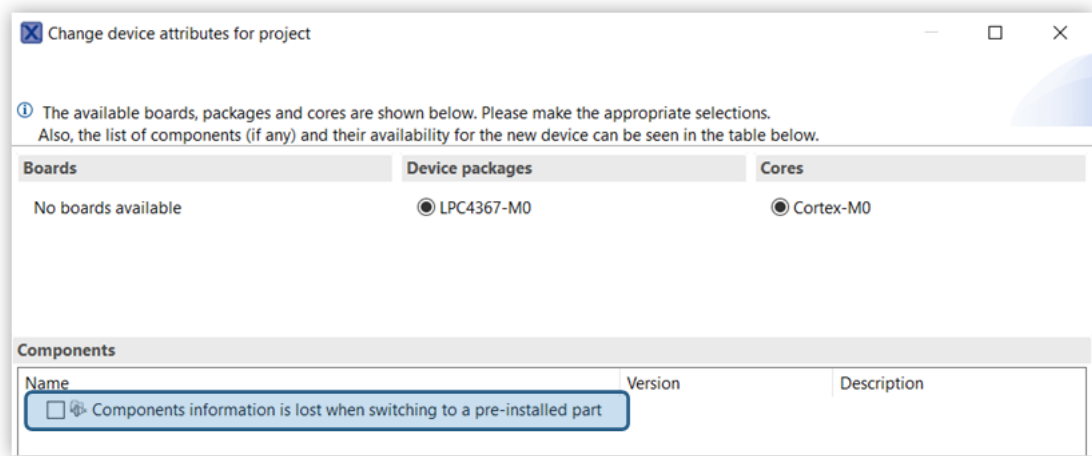
Figure 178. Change device attributes (preinstalled part)

15.3.2 Removal of SDK components associated with the old MCU

The following step requires confirmation on the removal of outdated SDK components (that is, associated with the old MCU). Files listed in the dialog are removed from the project and also replaced by their counterpart associated with the new device. These are SDK-specific source files and no user changes are expected to be found inside. In this context, any change is lost once the removal of components is confirmed.
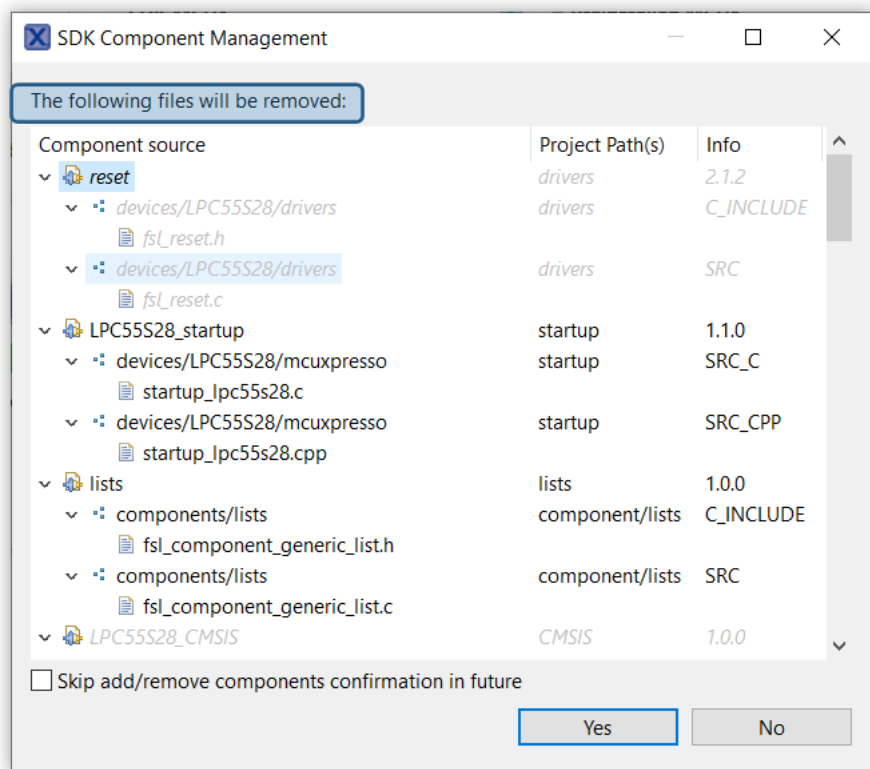
Figure 179. Confirm removal of outdated SDK components

15.3.3 Addition of SDK components associated with the new MCU

This is the last step of the process. At this point, components associated with the new MCU/SDK are added to the project. Depending on the imported SDK type (zipped or unzipped), files can be copied inside the project or linked. Only unzipped SDKs allow the linking of source files.
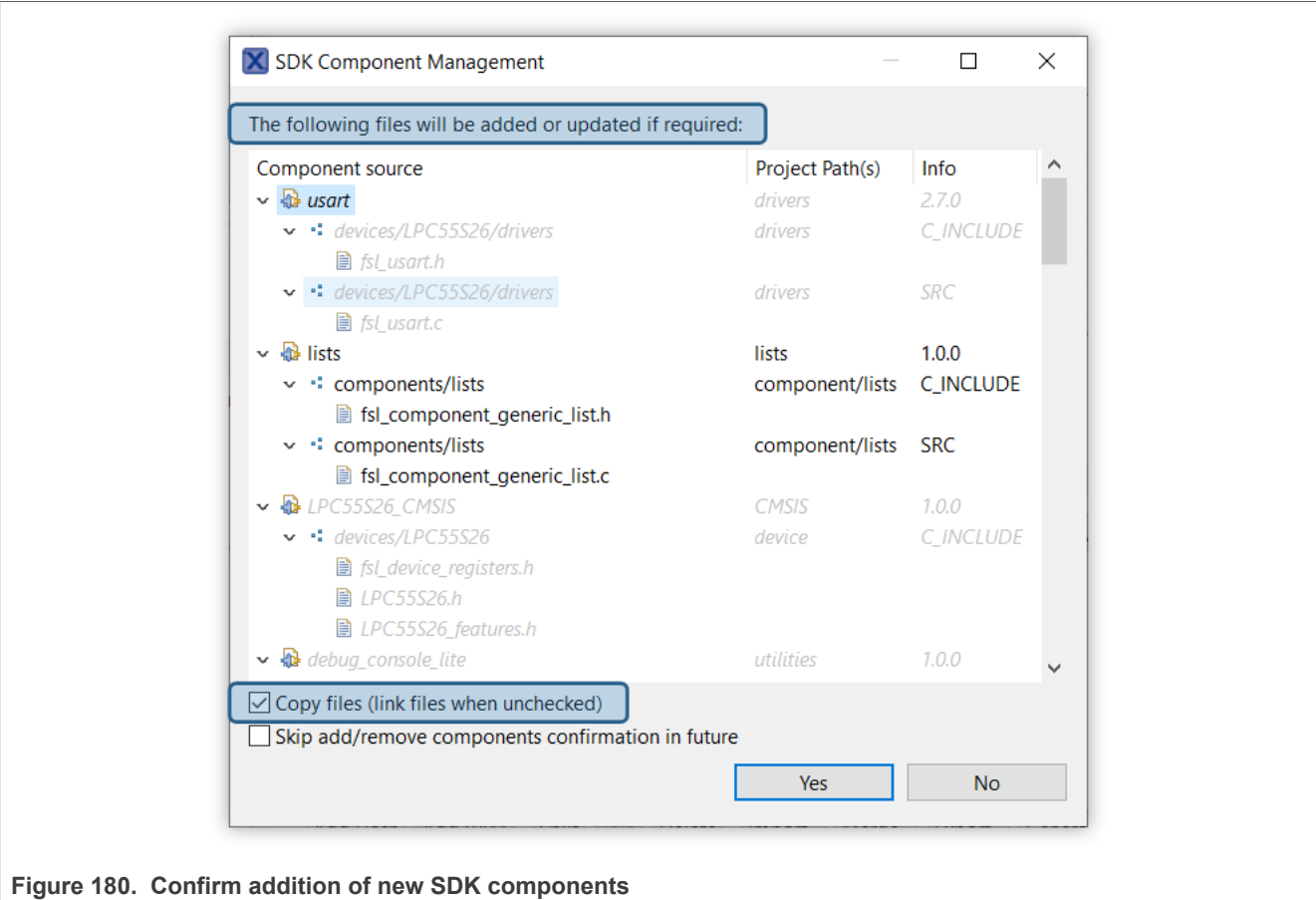
Figure 180. Confirm addition of new SDK components

15.4 Changing the MCU (SDK) package type

MCUs are commonly available in a range of package types. Different packages may impact the options available on the MCU external pins, for example, the number of GPIO lines. MCUXpresso IDE makes no use of this package type, however it is significant to the included MCUXpresso Config Tools.

As shown in the previous section, from the project virtual nodes, select *Edit MCU*.
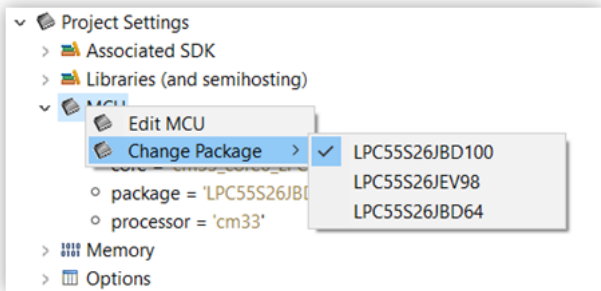
Figure 181. Edit package

Then select *Change Package* and choose the package required.

16 MCUXpresso Config Tools

This chapter provides an introduction to the features of the MCUXpresso Config Tools installed by default with MCUXpresso IDE. The Config Tools present new perspectives in addition to the Develop and Debug perspectives of the IDE.
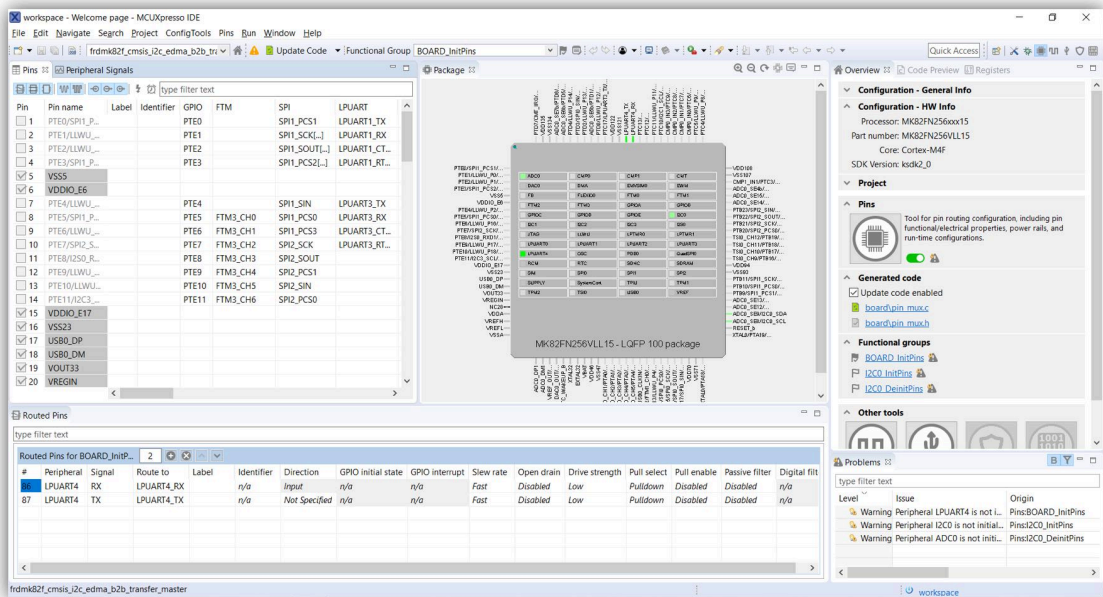
Figure 182. Config Tools showing Pins perspective

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

16.1 Using the Config Tools

MCUXpresso IDE includes the following Config Tools:

- Pins Tool
  - allows you to configure pin routing and generates 'pin_mux.c and .h' source files
- Clocks Tool
  - allows you to configure system clocks and generates 'clock_config.c and .h' source files
- Peripherals Tool
  - allows you to configure other peripherals and generates 'peripherals.c and .h' source files
- Device Configuration Tool
  - allows you to configure the initialization of memory interfaces of your device and generate dcd.d and dcd.h source files in C array or binary format
- TEE Tool
  - allows you to configure security policies of memory areas, bus masters, and peripherals, to isolate and safeguard sensitive areas of your application and generate tzm_config.c and .h source files.

MCUXpresso Config Tools can be used to review or modify the configuration of SDK example projects or new projects based on SDK 2.x. To open the tool, simply right-click the project in Project Explorer and select the appropriate Open command:
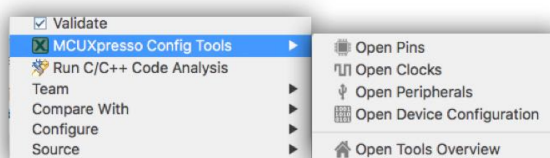
Figure 183. Config Tools launch

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

16.1.1 Tool perspectives

Each tool is displayed in a separate perspective. Once the configuration is opened, you can switch between perspectives to review/modify the configuration of each tool - using the toolbar on the upper right part of the IDE window:
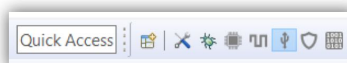
Figure 184. Switch Config perspective

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

16.1.2 Pins tool

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

16.1.3 Clocks tool ⓤ

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

16.1.4 Peripherals tool ⓥ

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

16.1.5 Device Configuration tool ▦

The Device Configuration tool allows you to configure the initialization of memory interfaces of your device. Use the Device Configuration Data (DCD) view to create different types of commands and specify their sequence, define their address, values, sizes, and polls.

16.1.6 TEE tool ♡

In the Trusted Execution Environment, or TEE tool, you can configure the security policies of memory areas, bus masters, and peripherals, to isolate and safeguard sensitive areas of your application. You can set the security policies of different parts of your application in the Security access configuration and its sub-views, and review these policies in the Memory map and Access overview views. Use the User Memory Regions view to create a convenient overview of memory regions and their security levels.

16.1.7 Generate code

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

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

16.1.8 SDK components

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

17 The GUI Flash tool

The GUI Flash tool provides flash programming capabilities for **all supported debug solutions**.

As well as implementing seamless programming of Flash when starting a debug session, MCUXpresso IDE enables the Flash programming capabilities of the supported debug solutions to be accessed directly, both via the GUI and from the command line (which might be useful for performing small production runs).

These flash programming capabilities can be accessed from three distinct places with the IDE.

First, the most feature-capable (advanced) variant is launched via the IDE button (and is described in this section):

Figure 185. GUI Flash Tool button

Clicking this launches a dialog similar to:
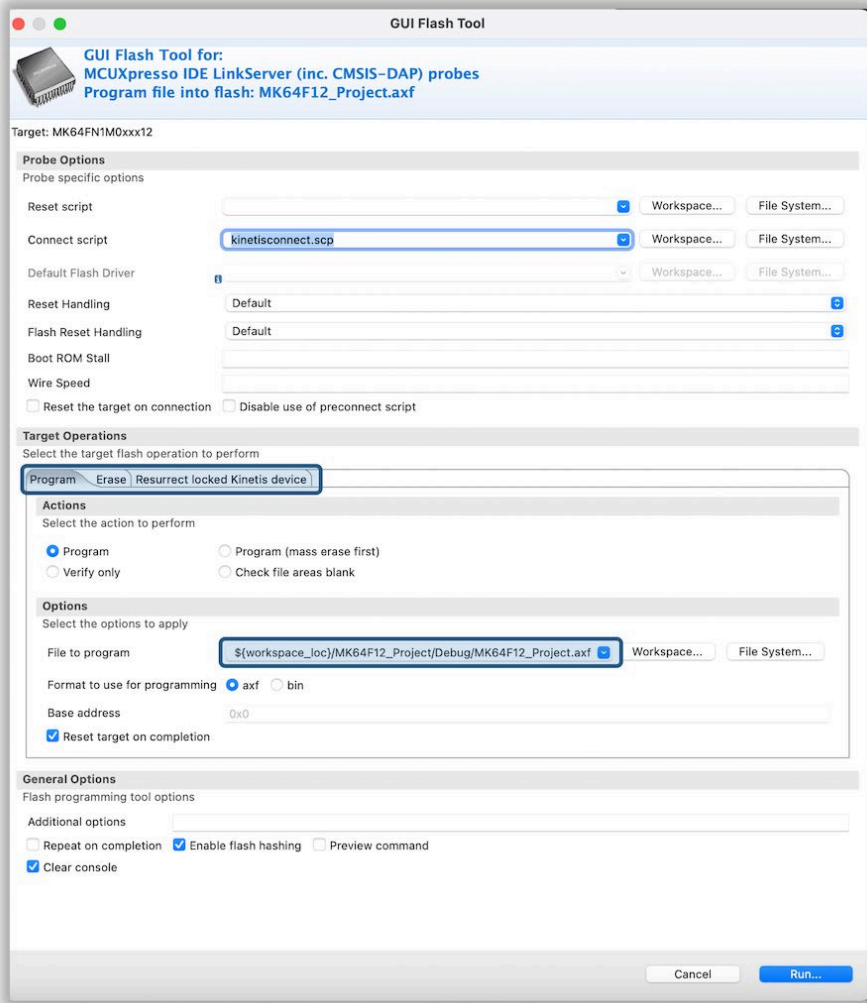
Figure 186. GUI Flash Tool

Note: This dialog varies subtly for each debug solution.

Second, project launch configurations now contain a GUI Flash Tool Tab providing project-specific flash operations. See Debug solutions overview for more information.

Finally, the **Quickstart** panel Debug Shortcuts provide easy access for simple project flash programming. See Debug Quickstart shortcuts for more information.

Tip: *For multicore MCUs, the core selection is usually made automatically, but for GUI flash operations, it may be necessary to take direct control of core selection, so this option is made available to the user.*

17.1 The advanced GUI Flash Tool

The operations below are supported for each debug solution.

1. Programming a .axf or .bin file into flash
2. Flash Mass Erase
3. Various debug solution-specific features

When launched, each debug solution presents a dialog similar to the LinkServer variant - described below:
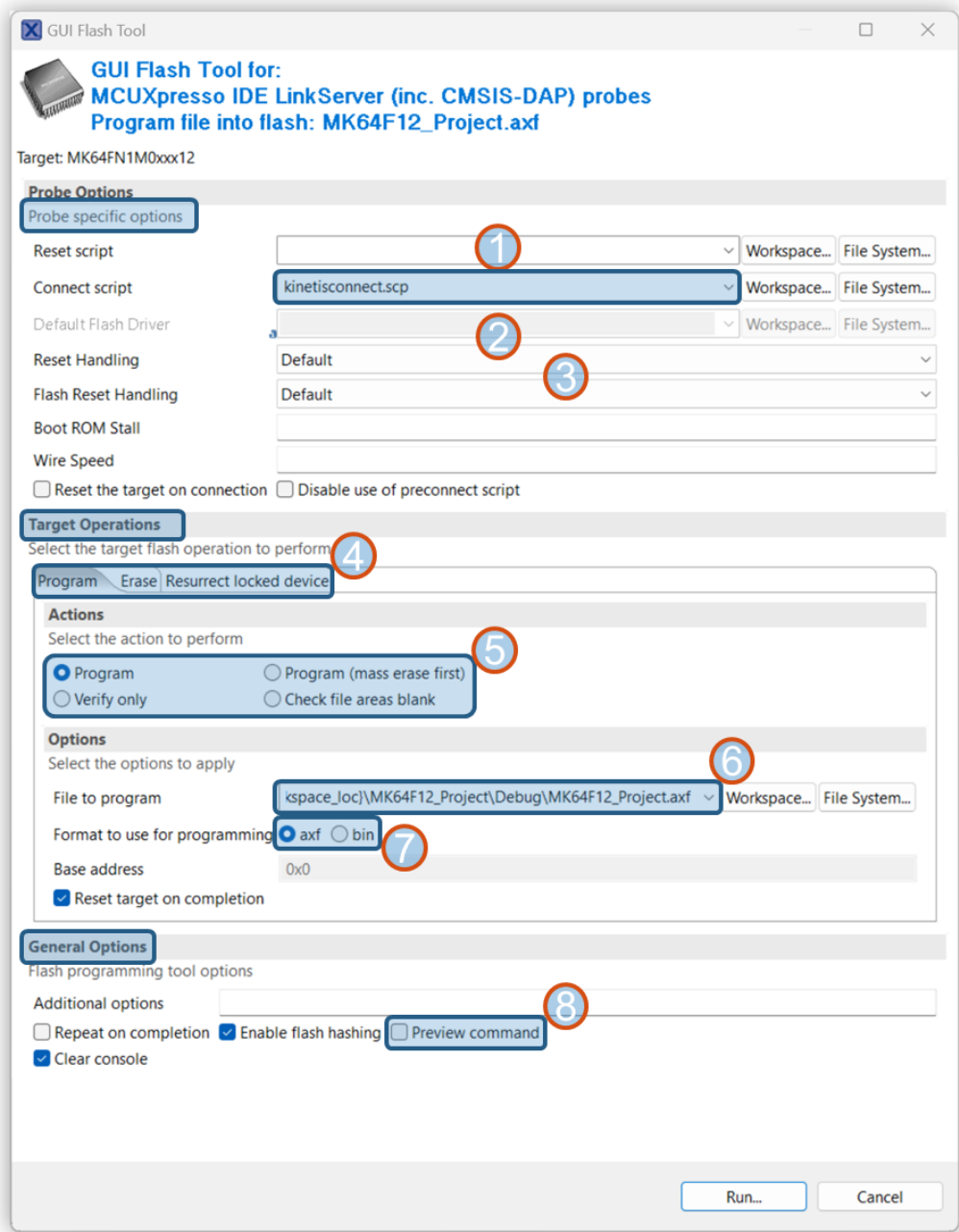
Figure 187. GUI Flash Tool major features

Note: Probe options (highlighted above) are different for each debug solution, whereas Target and General Options (also highlighted) are broadly similar.

Tip: A project must first be selected before the Advanced GUI Flash Tool can be launched. The device and other project configurations (such as flash drivers) are inherited from this selected project. The advanced GUI Flash tool does not create or use the information within project-associated launch configurations.

1. Reset and Connect scripts: Any SDK-specified Reset or Connect scripts are automatically selected. A different script can be selected if required using the workspace or file system shortcut buttons. If specified, a Reset script overrides the Reset Handling.

2. Reset Handling: The device default reset handling can be overridden from the selection: Default, SYSRESETREQ, VECTRESET, SOFT

3. Flash Reset Handling: The flash drivers default reset handling can be overridden from the selection: Default, SYSRESETREQ, VECTRESET, SOFT

4. Program/Erase/Resurrect locked device
   - **Program view** (displayed) should be selected to program an application of binary into flash. Only the Program options are described below.
   - **Erase view** should be selected for options to erase a flash device to its blank state.
     - Offers options to Mass erase, Erase by sector, and Check blank (to verify a blank flash).
     - Fenerally flashes do not need to be erased, since program operations automatically erase sections of the flash as required. However, on occasion, it can be useful to erase a flash, most often because the image in the flash is causing problems.
     - Erase by sector is not recommended for Kinetis parts since this leaves the device fully erased and therefore in a locked state - should this occur, use the option below ...
   - **Resurrect locked device** view should be selected to recover a *locked* device. This operation is applicable to MCXC and Kinetis series MCUs only.

5. Programming actions:
   - **Program:** the default action programs the selected application or binary erasing only the required sections of the flash device.
   - **Program (mass erase first):** erases the whole device before programming the selected application or binary. This ensures that any previous flash contents are erased.
   - **Verify only:** this option compares the contents of flash with the selected application or binary. **Note**: most flash programming operations are verified at the programming stage. Flash contents are not changed.
   - **Check file area blank:** this can be used to verify that a program operation does not overwrite any data already programmed into flash. Flash contents are not changed.

6. File selection: if the selected project contains a built .axf file, then this is automatically selected. Alternatively, a different file can be selected using the workspace and file system shortcut buttons.

7. Format: these radio buttons are preset by the *File to Program* type. However, if a .axf file is selected, clicking bin automatically generates a .bin from the selected .axf.
   - for file types containing no base address information, such as .bin, a base address must be specified.

8. Preview command: select this option to be presented with a preview programming command to be issued and a script that can perform this action independently of the IDE (see below)
   - The previewed command can be edited if required, and changes are reflected within the script. Various shell script *flavors* can be selected, and finally, the script can be copied to the clipboard with a single click

Finally, click *Run* to execute the flash programming operation, a dialog displaying the success of the operation is displayed once the program operation has been completed.

17.1.1 Advanced GUI Flash Tool command preview

As discussed in point 8 above, the GUI Flash Tool can optionally display the command to be issued - allowing the opportunity to edit the command before execution.
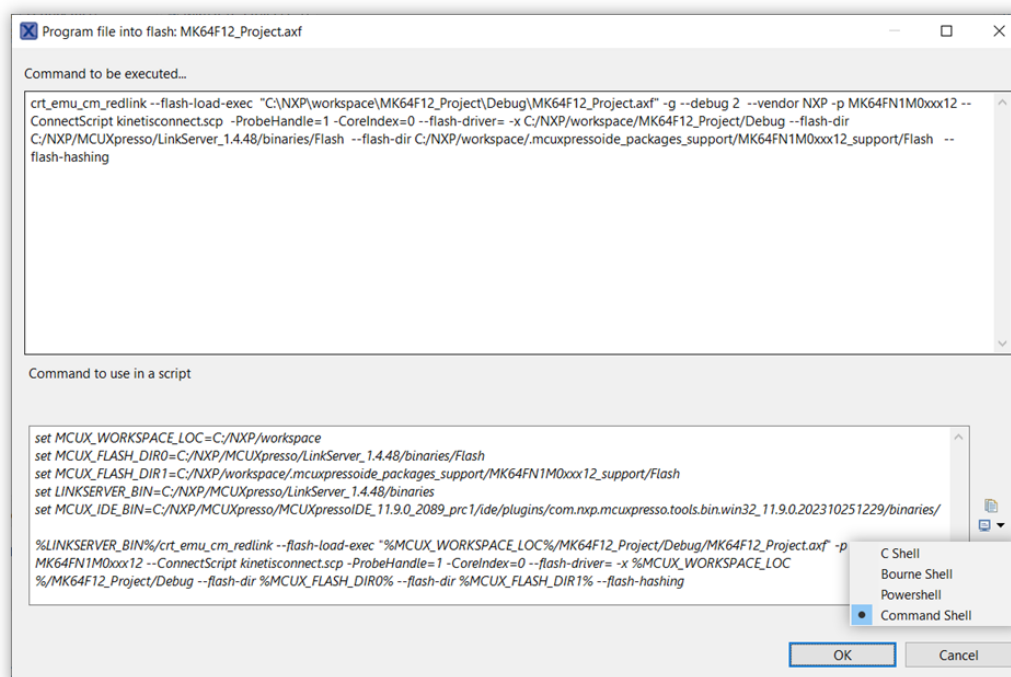
Figure 188. GUI Flash Tool command preview

In addition to displaying the command to be issued, the dialog also contains a script that can be issued independently of the IDE to perform the flash programming operation. Changes to the command to be executed are also reflected within the script.

Notes

- The script setups the local environment to be independent of your local shell configuration. However, components of MCUXpresso IDE are referenced so the script can only be used if MCUXpresso IDE is installed and any referenced workspace files are present.
- Debug probes may install drivers when **first** seen by a host, this driver installation may take some time to complete.
- MCUXpresso IDE is able to maintain connection to multiple debug probes, while the IDE can dynamically maintain knowledge of connected probes, any generated command line is a snapshot of a given instance. Therefore, it is essential that only a single debug probe is connected if the command script is to be captured for re-use.
- Typically, LPC-Link2 or LPCXpresso V2 and V3 boards have debug probe firmware soft loaded automatically by the IDE when a debug operation is first performed. Therefore, to use these debug probes from the command line they must either have their firmware softloaded or have probe firmware programmed into the Flash. Probe firmware can be soft-loaded from the command line by use of scripts *boot_link1* for LPC-Link and *boot_link2* for LPC-Link2, these are located at *mcuxpresso_install_dir/ide/binaries*. To program debug probe firmware into the Flash memory of an LPC-Link2 debug probe, see: https://www.nxp.com/LPCSCRYPT

17.1.2 Advanced GUI Flash Tool logged output

When a GUI Flash Tool operation is performed, the low-level output is logged into the debug log. A snippet of a LinkServer successful program operation is shown below:

```
...
Loading 'MK64FN1M0xxx12_Project.axf' ELF 0x00000000 len 0x3CF8
```

```
Opening flash driver FTFE_4K.cfx (already resident)
Sending VECTRESET to run flash driver
Writing 15608 bytes to address 0x00000000 in Flash
1 of 1 (  0) Writing pages 0-3 at 0x00000000 with 15608 bytes
(  0) at 00000000: 0 bytes - 0/15608
( 26) at 00000000: 4096 bytes - 4096/15608
( 52) at 00001000: 4096 bytes - 8192/15608
( 78) at 00002000: 4096 bytes - 12288/15608
(100) at 00003000: 4096 bytes - 16384/15608
Erased/Wrote page  0-3 with 15608 bytes in 693msec
Closing flash driver FTFE_4K.cfx
(100) Finished writing Flash successfully.
Flash Write Done
Loaded 0x3CF8 bytes in 1081ms (about 14kB/s)
Reset target (system)
Starting execution using system reset
```

17.1.3 Advanced GUI Flash Tool programming an arbitrary binary

The GUI Flash tool is usually used to program a binary generated from the .axf file of a Project. However, on occasion, it might be required to program a binary (or .axf) file generated elsewhere. This can be achieved by generating a project with the required memory/chip combination and simply dropping the .bin file into this project. When the GUI Flash tool is invoked, the user can browse for the required binary file and program this in the usual way.

18   LinkServer Flash support

LinkServer (CMSIS-DAP) Flash drivers are used by LinkServer debug connections only. Refer to the section on LinkServer debug for details of the LinkServer debug solution.

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

At the time a debug connection is started by MCUXpresso IDE, a LinkServer debug session running on the host typically downloads a Flash driver into RAM on the target MCU. It then communicates with the downloaded Flash driver via the debug probe to program the required code and data into the Flash memory.

In addition, the loadable Flash driver mechanism also provides the ability to support Flash drivers, which can be used to program external Flash memory (for instance via the SPIFI Flash memory interface on LPC18x, LPC40xx, LPC43xx, LPC5460x, and i.MX RT families). The sources for some of these drivers are provided in the */LinkServer/Examples/Flashdrivers* subdirectory accessible within the MCUXpresso IDE installation directory. These are part of the actual LinkServer package that is installed in a separate folder from the IDE.

Note: Quad SPI (QSPI) and SPIFI are used interchangeably within this section. The term SPIFI (SPI Flash Interface) is commonly used to reference LPC use of QSPI.

LinkServer Flash drivers have a .cfx file extension. For Preinstalled MCUs, the Flash driver used for each part/ family is located in the *LinkServer/binaries/Flash* subdirectory of the MCUXpresso IDE installation - note that these are part of the LinkServer package that is installed in a separate folder than the IDE. For SDK-installed MCUs, the Flash driver is generally supplied within the SDK, although copies may also be provided in the */ LinkServer/binaries/Flash* subdirectory.

Important Note: LinkServer flash drivers are fully integrated into the MCUXpresso IDE Managed Linkerscript build mechanism and specified within SDK metadata. Other debug solutions invoke MCU-specific flash programming strategies based on their debug implementation's knowledge of the MCU being debugged.

UG10055

© 2025 NXP B.V. All rights reserved.

User guide

Rev. 4 — 14 January 2025

Document feedback

186 / 316

18.1 Default vs per-region Flash drivers

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

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

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

For example, this allows a project based on an LPC43xx device with internal Flash to also make use of an external SPIFI Flash device. This is achieved by removing the default Flash driver from the memory configuration and instead explicitly specifying the Flash driver to use for each Flash memory block (per-region Flash drivers). A typical use case could be to create an application to run from the internal Flash of the MCU that makes use of static constant data (for example, for graphics) stored in an external SPIFI device. An example memory configuration is shown below:
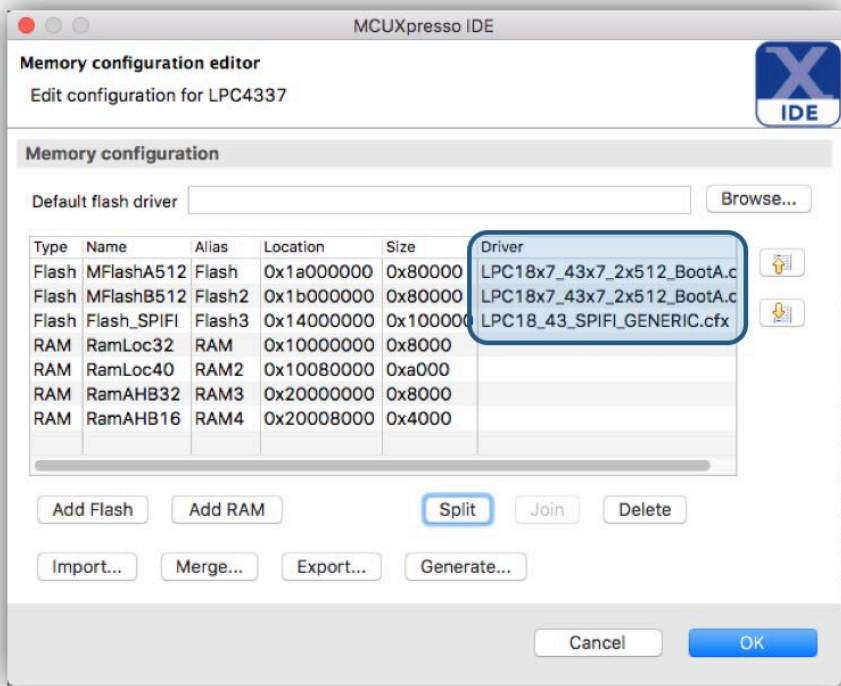
Figure 189. Per region drivers

Note: SDK-installed MCU support always uses Per-Region Flash drivers.

18.2 Advanced Flash drivers

Most wizard-generated projects or projects imported from SDKs (or LPCOpen) are preconfigured with an appropriate LinkServer flash driver for the target flash device. As a result, in many cases, users need to pay little attention to the actual flash driver being used. However, for MCUs supporting complex flash strategies or external flash devices, the situation is more complex. This section discusses these situations but note that, even in these cases, the flash driver may be automatically selected and so require no user attention.

18.2.1 LPC18xx / LPC43xx internal Flash drivers

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

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

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

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

18.2.2 LPC SPIFI QSPI Flash drivers

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

At the time of writing, these LPC devices consist of:

Table 2. SPIFI details

LPC part	SPIFI address	Bootable	Flash driver
LPC18xx/LPC43xx	0x14000000	Yes	LPC18_43_SPIFI_GENERIC.cfx
LPC40xx	0x28000000	No	LPC40xx_SPIFI_GENERIC.cfx
LPC5460x	0x10000000	No	LPC5460x_SPIFI_GENERIC.cfx
LPC540xx	0x10000000	Yes	LPC540xx_SPIFI_GENERIC.cfx

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

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

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

18.2.2.1 Flash devices supported by our LPC SPIFI Flash drivers

The paragraph below contains information that is largely deprecated - see the section Flash drivers using SFDP.

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

```
GD25Q32C
MT25QL128AB
MT25Q512A
MT25Q256A
N25Q256
N25Q128
N25Q64
N25Q32
PM25LQ032C
MX25L1606E
MX25L1635E
MX25L3235E
MX25R6435F
MX25L6435E
MX25L12835E
MX25V8035F
MX25L8035E
S25FL016K
S25FL032P
S25FL064P
S25FL129P 64kSec
S25FL129P 256kSec
S25FL164K
S25FL256S 64kSec
S25FL256S 256kSec
S25FL512S
W25Q40CV
W25Q32FV
W25Q64FV
W25Q128FV
W25Q256FV_Untested
W25Q80BV
```

18.2.3 i.MX RT QSPI and Hyper Flash frivers

I.MX.RT MCUs support external flash via a QSPI/Hyperbus interface, and a range of LinkServer flash drivers supporting devices fitted to EVK development boards are included with MCUXpresso IDE (as described below).

Note: these drivers are also supplied in source project form so they may be used as a base for the development of drivers for other external flash parts. These driver projects can be found at *Examples/Flashdrivers/NXP/ iMXRT*.

Table 3. Flash details

i.MX RT part	Base address	Bootable	Flash driver
i.MX RT 1050	0x60000000	Yes	MIMXRT1050-EVK_S26KS512S.cfx
i.MX RT 1050	0x60000000	Yes	MIMXRT1050-EVK_IS25WP064A.cfx

Table 3. Flash details...*continued*

i.MX RT part	Base address	Bootable	Flash driver
i.MX RT 1050	0x60000000	Yes	MIMXRT1050-EcoXiP_ATXP032.cfx
i.MX RT 1020	0x60000000	Yes	MIMXRT1020-EVK_IS25LP064.cfx

When used with the appropriate SDK for your development board, the correct driver is automatically selected.

Important Note: For an application to Boot and execute in place (XIP) from these flash devices (post reset), a correct header for the specific device **MUST be programmed into the flash (as part of the Project)**. SDK examples are built to include an appropriate header automatically, however, MCUXpresso IDE does not prevent users from programming projects without headers into these devices. If this occurs, the application does not boot and subsequent flash programming operations may fail.

Should this occur, the recommended recovery procedure is to change the boot strategy of the board (via DIP switches) to prevent booting from QSPI or hyperflash. Power cycle the board and then perform a Mass Erase of the flash. Next, reprogram with an image that has an appropriate header, restore the boot strategy, and power cycle again.

Tip: *In addition, these drivers are complemented by a range of self-configuring drivers supporting all current i.MX RT EVK boards, see* Flash drivers using SFDP protocol *for more information on the drivers and this methodology.*

18.2.4 Flash drivers using SFDP (LPC and i.MX RT)

As discussed above, *programming these Flash memories provides several challenges because the size of memory (if present) is unknown, and the actual memory device is also unknown.*

LinkServer **Generic** flash drivers attempted to solve this problem by recognizing specific devices (via their JEDEC ID) and then setting their sizes and programming parameters accordingly. However, this mechanism only works if the device is recognized by the flash driver, and in consequence fails if any device is not recognized.

This issue, combined with the sheer volume of devices available, has forced a different approach to be taken. Fortunately, modern flash devices typically contain a data block describing their properties including device size, low-level structure and programming details, and so on. These data blocks and their use are collectively known as *Serial Flash Discovery Protocol* or SFDP. The standard for these blocks is described by JEDEC JESD216 standard(s).

Introduced in MCUXpresso IDE version 10.2.0 are a range of Generic flash drivers built to self-configure via SFDP data and these have been extended for later MCUXpresso IDE versions. The current list of supported SFDP drivers is shown below:

Table 4. SFDP Flash details

Part	Base address	Bootable	Flash driver
LPC18xx/LPC43xx	0x14000000	Yes	LPC18_43_SPIFI_SFDP.cfx
LPC546xx	0x10000000	No	LPC546xx_SPIFI_SFDP.cfx
LPC540xx	0x10000000	Yes	LPC540xx_SPIFI_SFDP.cfx
LPC55S36	0x10000000	Yes	LPC553x_FlexSPI_A_MXIC_OPI.cfx
i.MX RT 1170	0x30000000	Yes	MIMXRT1170_SFDP_MXIC_OPI.cfx
i.MX RT 1170	0x30000000	Yes	MIMXRT1170_SFDP_QSPI.cfx
i.MX RT 1160	0x30000000	Yes	MIMXRT1160_SFDP_MXIC_OPI.cfx

Table 4. SFDP Flash details...*continued*

Part	Base address	Bootable	Flash driver
i.MX RT 1160	0x30000000	Yes	MIMXRT1160_SFDP_QSPI.cfx
i.MX RT 1064	0x70000000	Yes	MIMXRT1064.cfx
i.MX RT 1060	0x60000000	Yes	MIMXRT1060_SFDP_HYPERFLASH.cfx
i.MX RT 1060	0x60000000	Yes	MIMXRT1060_SFDP_QSPI.cfx
i.MX RT 1050	0x60000000	Yes	MIMXRT1050_SFDP_HYPERFLASH.cfx
i.MX RT 1050	0x60000000	Yes	MIMXRT1050_SFDP_QSPI.cfx
i.MX RT 1024	0x60000000	Yes	MIMXRT1024.cfx
i.MX RT 1020	0x60000000	Yes	MIMXRT1020_SFDP_QSPI.cfx
i.MX RT 1015	0x60000000	Yes	MIMXRT1015_SFDP_QSPI.cfx
i.MX RT 1010	0x60000000	Yes	MIMXRT1010_SFDP_QSPI.cfx
i.MX RT 600	0x8000000	Yes	MIMXRT600_FlexSPI_A_MXIC_OPI.cfx
i.MX RT 600	0x8000000	Yes	MIMXRT600_FlexSPI_A_SFDP_QSPI.cfx
i.MX RT 600	0x8000000	Yes	MIMXRT600_FlexSPI_B_MXIC_OPI.cfx
i.MX RT 600	0x8000000	Yes	MIMXRT600_FlexSPI_B_SFDP_QSPI.cfx
i.MX RT 500	0x8000000	Yes	MIMXRT500_SFDP_MXIC_OSPI.cfx
i.MX RT 500	0x8000000	Yes	MIMXRT500_SFDP_QSPI.cfx
PN7640	0x218000	Yes	PN76xx.cfx

Important Note: for some i.MX RT parts, the current SDKs reference the device-specific flash driver rather than the SFDP version. However, you can modify your project to use the SFDP version if required. Flashdrivers cannot detect whether QSPI or Hyperflash is fitted on a board, therefore it is the responsibility of the user to ensure that the correct driver is used.

Note: The i.MX RT 1024 and 1064 MCUs incorporate a flash device within the MCU package itself however, the flash driver still uses the SFDP mechanism to detect the device and therefore is listed in the table above.

18.2.4.1 QSPI SFDP issues and limitations

Some (usually older) QSPI parts do not support the SFDP mechanism and are therefore not programmable via this protocol. However since some of these QSPI devices are fitted to NXP (LPC) manufactured development boards, some basic assumptions are made by these drivers if SFDP data is not found. In such a case, the device and its size are assumed to be 1 MB and some standard programming mechanisms are used. This scheme should ensure that NXP LPC development boards with QSPI can be used with this driver type.

Note: this information is correct at the time of writing and only applies to LPC Drivers - future development of these drivers may change their capabilities.

18.2.4.2 Flash programming log

When programming code or data into flash, a portion of the debug log displays the flash programming operations (as below):

```
Inspected v.2 External Flash Device on SPI using SFDP JEDEC ID
 LPC18_43_SPIFI_SFDP.cfx -(1)
Image 'LPC1843_JEDEC_SFDP May  1 2018 15:32:05'
```

```
Opening flash driver LPC18_43_SPIFI_SFDP.cfx
 ----------------------------------------(2)
Sending VECTRESET to run flash driver
flash variant 'JEDEC_SFDP_EF4014' detected (1MB = 16*64K at 0x14000000)
 ---------------(3)
Closing flash driver LPC18_43_SPIFI_SFDP.cfx
NXP: LPC43S37
Connected: was_reset=true. was_stopped=false
Awaiting telnet connection to port 3330 ...
GDB nonstop mode enabled
Opening flash driver LPC18_43_SPIFI_SFDP.cfx (already resident)
 ----------------------(4)
Sending VECTRESET to run flash driver
Writing 1046900 bytes to address 0x14000000 in Flash
 -------------------------------(5)
Erased/Wrote page  0-15 with 1046900 bytes in 7548msec
 -----------------------------(6)
Closing flash driver LPC18_43_SPIFI_SFDP.cfx
Flash Write Done
Flash Program Summary: 1046900 bytes in 7.55 seconds (135.45 KB/sec)
 ------------------(7)
Stopped: Breakpoint #1
```

Note: When accessing unknown flash devices, the driver is called twice. First to identify the device and second to perform the required programming. In a situation where multiple devices are being programmed, the flash driver(s) may be (re)loaded for each use.

Where:

1. *SFDP JEDEC ID* is the method used to access the flash and *LPC18_43_SPIFI_SFDP.cfx* is the flash driver used
2. The driver named above is loaded and initialized (this step setups clocks, pin muxing, and performs some investigation of the connected device)
3. The driver returns a string *JEDEC_SFDP* indicating that SFDP data was found and successfully read
   - The JEDEC ID of the device was read as *EF4014*, in this case corresponding to a Winbond 25Q80DVSIG (as fitted to the LPC-Link2 board used in Target mode)
   - The size of the device was read as 1 MB divided up into 16 64KB Sectors/Blocks - these blocks are the erase size that is used for programming and so any operation to program this flash must start on an address aligned to this 64 kB size
4. The driver is opened a second time (without reloading since it remains from the previous call)
5. The project that referenced this driver requested that 1,046,900 bytes of data be written to the address starting 0x14000000, as set within the memory configuration of the project
6. The write operation is performed via 16 page writes
   - **Note**: this flash driver (like many LinkServer drivers) uses a virtual page size that is larger than the actual flash device page size to optimize driver operation
7. Finally, a summary of the operation is printed showing the flash programming performance

Note: If the driver fails to find SFDP data, it attempts to program the device with standard routines. If this occurs, the size is assumed to be 1 MB and the flash variant is reported as *ID* rather than *SFDP* as shown below:

```
flash variant 'JEDEC_ID_EF4014' detected (1MB = 16*64K at 0x14000000)
```

On occasion, some devices that report the same JEDEC ID are different, in this particular case, the device is a similar Winbond 25Q80BVSIG, that is, **..BV** rather than **..DV**

18.2.4.3 QSPI programming and booting

When dealing with an external flash, it is important to understand the difference between the flash programming operation performed by the flash driver and the subsequent use of the flash for executing code and/or providing data. Essentially, the responsibility of the flash driver ends with a successful program operation, after this point, the correct operation of the MCU/SPI flash combination lies elsewhere.

Therefore, once the MCU is reset (or power cycled), the responsibility for the configuration of the device and operation lies entirely outside MCUXpresso IDE and instead lies with one or all of the following:

• Development board/MCU boot settings
  – These may be DIP switches or Jumpers providing inputs to the MCU boot flow, alternatively, these could be OTP bits programmed within the MCU
• MCU's BootROM's ability to understand and setup the device
  – BootROMs on devices such as the LPC1800 and LPC4300 have an inbuilt understanding of certain QSPI devices allowing them to be configured for boot. However, this boot process may fail with some QSPI flash although it has been correctly programmed
  – BootROMs on devices such as the LPC540xx and RT10xx rely on the correct header (XIP) information being programmed (as part of the Application) into the QSPI flash itself. If this data is incorrect (or not present), the boot/reset fails.
• Devices that incorporate both internal boot flash and external SPIFI/QSPI flash such as the LPC546xx typically place the responsibilities for QSPI configuration on the user application, where this might include
  – Setup of pinmuxing
  – QSPI/SPIFI clock setup
  – Flash interface initialization
  – QSPI initialization (this may be QSPI device-specific)
    – Including setup of appropriate waitstates for QSPI operation at the selected QSPI clock frequency

18.2.4.4 FlexSPI Flash reset

A number of i.MX RT MCUs that support external flash via the FlexSPI interface implement a flash device reset sequence.

During FlexSPI boot the boot process requires the FlexSPI Flash device to be in a certain mode, for example, 1-bit SPI compatible mode. The Flash device is naturally in this mode after a POR reset because the power-up sequence resets it with the RT MCU device together. However, the Flash device is not in 1-bit SPI compatible mode if the flash device is configured to DPI mode, QPI mode, or Octal mode when any non-POR resets happen. In such cases, special processing is required by the boot process to restore the Flash device to 1-bit SPI-compatible mode before continuing access to the Flash device. In general, this can be achieved by using a GPIO to assert a reset pin on the Flash device. The bootloader can perform the reset process and reset the Flash device to 1-bit SPI-compatible mode based on fuse configuration, using the GPIO specified by the combination of FLEXSPI_RESET_PIN_PORT and FLEXSPI_RESET_PIN_GPIO.

When starting a flash-resident debug session in MCUXpresso IDE this reset sequence may need to be performed by the flash driver as well. Flash drivers for i.MX RT500 and RT600 MCUs implement this functionality.

Note: Custom boards may not be wired identically to EVK development boards in regards to the actual pin dedicated to flash device reset. In such cases, the pre-connect script must be modified to pass to the flash drivers the relevant information about the GPIO pin used for flash reset.

18.3 Kinetis Flash drivers

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

Kinetis Flash drivers generally follow a simple naming convention, that is, **FTFx_nK_xx** where:

- FTFx is the Flash module name of the MCU, where x can take the value E, A, or L
- nK represents the Flash sector size the Flash device supports, where n can take the value 1, 2, 4, 8
  - A sector size is the smallest amount of Flash that can be erased on that device
- xx represents optional additional characters for special case drivers, for example, __Tiny for use on parts with a small quantity of RAM
  - A further optional _D suffix is used to show the driver is written to target Data Flash rather than the more common Program Flash

So for example, the Flash driver of a K64F MCU is called *FTFE_4K*, because the K64F MCU uses the FTFE Flash module type and supports a 4 kB Flash sector size.

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

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

```
Inspected v.2 On chip Kinetis Flash memory module FTFE_4K.cfx
Image 'Kinetis SemiGeneric Feb 17 2017 17:24:02'
Opening flash driver FTFE_4K.cfx
Sending VECTRESET to run flash driver
Flash variant 'K 64 FTFE Generic 4K' detected (1MB = 256*4K at 0x0)
Closing flash driver FTFE_4K.cfx
Connected: was_reset=true. was_stopped=true
Awaiting telnet connection to port 3330 ...
GDB nonstop mode enabled
Opening flash driver FTFE_4K.cfx (already resident)
Sending VECTRESET to run flash driver
Flash variant 'K 64 FTFE Generic 4K' detected (1MB = 256*4K at 0x0)
Writing 25856 bytes to address 0x00000000 in Flash
00001000 done  15% (4096 out of 25856)
00002000 done  31% (8192 out of 25856)
00003000 done  47% (12288 out of 25856)
00004000 done  63% (16384 out of 25856)
00005000 done  79% (20480 out of 25856)
00006000 done  95% (24576 out of 25856)
00007000 done 100% (28672 out of 25856)
Erased/Wrote sector  0-6 with 25856 bytes in 301msec
Closing flash driver FTFE_4K.cfx
Flash Write Done
Flash Program Summary: 25856 bytes in 0.30 seconds (83.89 KB/sec)
```

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

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

```
K40    FTFL_2K (256KB)
```

18.4 Configuring projects to span multiple Flash devices

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

18.5 The LinkServer GUI Flash Programmer

The LinkServer GUI Flash Programmer has been replaced by the debug solution independent GUI Flash Tool.

18.6 The LinkServer command-line Flash Programmer

While the information below is still current, for most users this functionality has been replaced by features within the the GUI Flash Tool.

18.6.1 Command-line programming

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

The MCUXpresso IDE Flash programming utility is part of the external LinkServer package but can also be accessed from:

```
<install_dir>/ide/LinkServer/binaries/
```

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

For example:

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

loads the AXF file LPC11U68_App.axf into Flash on an LPC11U68.

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

18.6.1.1 Programming an image into Flash

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

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

It is also possible to flash binary files using:

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

Where:

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

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

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

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

18.6.1.2 Programming Flash with SDK Part Support

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

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

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

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

On to the command line already described. For example:

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

```
  --flash-dir ~/mcuxpresso/01/.mcuxpressoide_packages_support/LPC54018_support/
Flash
```

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

Windows:

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

MacOS or Linux:

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

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

18.6.1.3 Programming Flash taking MCUXpresso IDE project memory edits into account

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

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

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

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

```
-x "sdk_parts_directory"
```

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

18.6.1.4 Programming Flash for complex debug connections

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

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

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

--connectscript "connectscript"

In addition to connect scripts, some chips also require a preconnect script that prepares the target MCU for the initial debug connection. A preconnect script can be supplied by adding the following option to the command line already described:

--preconnectscript "preconnectscript"

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

--resetscript "resetscript"

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

18.6.1.5 Finding the correct parameters from MCUXpresso IDE

Note: A simple way of finding the correct commands and options is to use the GUI Flash Programmer described above, the completion dialog shows the exact command line invoked by the GUI. On this line, the IDE will have chosen the correct:

- target name
- a default Flash driver, *flashdriver*
- a connect script to be run, if needed
- a preconnect script to be run, if needed
- a reset script to be run, if needed with --flash-load-exec
- an sdk_parts_directory where XML information about the part being used (if it is provided via an SDK) can be found
- an sdk_flash_directory where flash drivers supporting the part being used (if it is provided via an SDK) can be found

Note: The details appear and are relevant only if a project supporting the relevant chip or board is selected in the project explorer view.

For example, the command line produced might be:

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

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

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

18.6.1.6 Dealing with errors during Flash operations

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

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

There are a number of ways to address this situation:

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

18.6.1.7 Validating the content of Flash

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

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

it is also possible to verify binary files using:

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

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

For example:

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

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

18.6.1.8 Erasing the Flash

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

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

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

For example:

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

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

18.6.1.9 Validating that Flash has been erased

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

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

For example:

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

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

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

It is also possible to verify binary files using:

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

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

For example:

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

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

18.6.1.10 Examples

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

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

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

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

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

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

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

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

```
    --flash-dir ~/mcuxpresso/01/.mcuxpressoide_packages_support/
MK64FN1M0xxx12_support/Flash
```

To delete the Flash on an LPC1343:

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

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

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

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

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

19 C/C++ library support

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

19.1 Overview of Redlib, Newlib, and NewlibNano

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

By default, MCUXpresso IDE uses 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 to provide a version of Newlib focused on code size. Using NewlibNano can help dramatically reduce the size of your application compared to using the standard version of Newlib - for both C and C++ projects.

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

19.1.1 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.
- inttypes.h
  – An implementation of the C99 inttypes.h header is provided.
- itoa
  – itoa() is a 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.

19.1.2 Newlib vs NewlibNano

Differences between Newlib and NewlibNano include:

- NewlibNano is optimized for size.
- The printf and scanf family of routines have been re-implemented in NewlibNano to remove a direct dependency on the floating-point input/output handling code. Projects that need to handle floating-point values using these functions must now explicitly request the feature during linking.
- The printf and scanf family of routines in NewlibNano support only conversion specifiers defined in the C89 standard. This provides a good balance between a small memory footprint and a full-feature formatted input/ output.
- NewlibNano removes the now redundant integer-only implementations of the printf/scanf family of routines (iprintf/iscanf, and so on). These functions are now aliases to 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

19.2 Library variants

Each C library family is provided in a number of different variants: None, Nohost and Nohost-nf, Semihost and Semihost-nf (Redlib only). These variants each provide a different set of 'stubs' that form the 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 therefore provides differing levels of functionality:

- **Semihost(-mb)**
  – This library variant provides an implementation of all functions, including file I/O. The file I/O is directed through the debugger and is performed on the host system (semihosting). For example, printf/scanf uses the debugger console window and fread/fwrite operates on files on the host system. **Note**: emulated I/O is relatively slow and can only be used when debugging.
- **Semihost(-mb)-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 and Nohost-nf**
  – This library variant provides the string and memory handling functions and some file-based I/O functions. However, it assumes that you have no debugging host system, therefore any file I/O does 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.

Note: -mb library variants are not selected by default during any wizard project creation, however they may optionally be selected for enhanced semihost performance with the penalty of slightly larger RAM usage. See Semihosted printf for additional information.

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 see build errors in the form:

- Linker error "Undefined reference to 'xxx'"

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

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

Or if linking against NewlibNano(None):

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

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

19.3 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.

19.3.1 Manually switching

Alternatively, you can make the required changes to your project properties manually as follows...

When switching between Newlib(Nano) and Redlib libraries, you must also switch the headers (since the 2 libraries use different header files). To do this:

1. Select the project in Project Explorer
2. Right-click and select Properties
3. Expand C/C++ Build and select Settings
4. In the Tools settings tab, select Miscellaneous under MCU C Compiler. **Note**: Redlib is not available for C++ projects
5. In Library headers, select Newlib or Redlib
6. In the Tools setting tab, select Architecture & Headers under MCU Assembler
7. In Library headers, select Newlib or Redlib

Repeat the above sequence for all Build Configurations (typically Debug and Release).

To then change the libraries actually being linked with (assuming you are using Managed linker scripts):

1. Select the project in Project Explorer
2. Right-click and select Properties
3. Expand C/C++ Build and select Settings
4. In the Tools settings tab, select Managed Linker Script under MCU Linker
5. In the Library drop-down, select the Newlib, NewlibNano, or Redlib library variant that you require (None, Nohost, Semihost, Semihost-nf).

Again, repeat the above sequence for all Build Configurations (typically Debug and Release). **Note**: Redlib is not available for C++ projects.

19.4 What is Semihosting?

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

19.4.1 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. 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 console view of the IDE.

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.

19.4.2 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 is effectively a "write character to stdout". The debug tools 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 the hard drive of your PC, and fscanf can then be used to read from that file.

19.4.3 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 stopped while 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.

In MCUXpresso IDE version 10.2.0, semihosting performance has been enhanced to deliver roughly double the speed when compared with the previous IDE release. Furthermore, a new **MB** library variant is been supplied that delivers a significant further improvement in performance when combined with LinkServer debug connections. This library along with new LinkServer debug support provides the added benefit of no impact on code execution performance.

19.4.4 Important notes about using Semihosting

When you have linked with the semihosting library, your application no longer works standalone - it only works when connected to the debugger.

Semihosting operations cause the CPU to drop into a "debug state", which means that during the data transfer between the target and the host PC, no code (including interrupts) gets executed on the target. Therefore, if your application uses interrupts, then it is normally advisable to avoid the use of semihosting while interrupts are active - and certainly within interrupt handlers themselves. If you still have 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 ITM channel of Cortex-M CPUs.

19.4.5 Semihosted printf and debugging

Semihosting is common to all supported debug solutions so the implications of this mechanism should be understood:

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

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

Introduced in MCUXpresso IDE version 10.1.0: New projects and newly imported SDK example projects automatically include a semihost hard fault handler (as can be seen in the image below). The purpose of this handler is to prevent the problem described above. Now, if a semihosted operation is performed without debug tools attached, the new semihost hard fault handler will be entered. The handler checks to see if a semihosted operation caused it to be entered and if so, simply return.
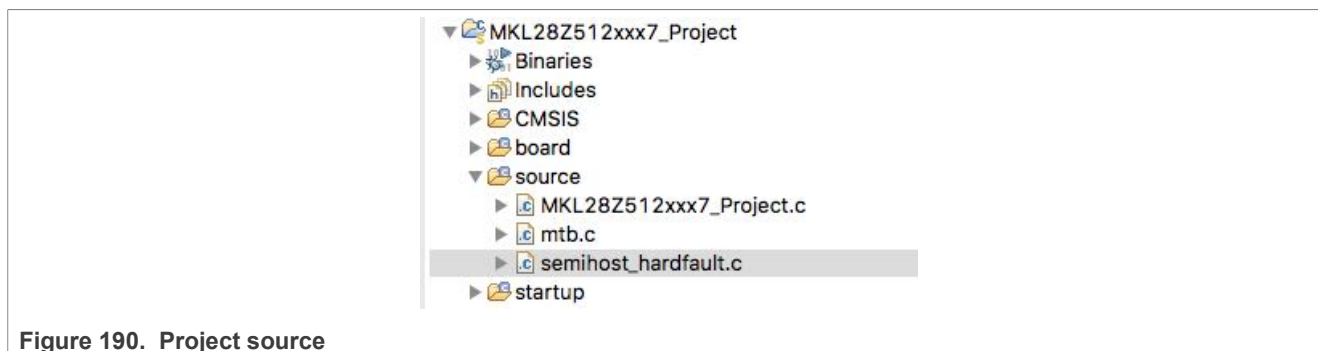
Figure 190. Project source

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

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

```
__SEMIHOST_HARDFAULT_DISABLE
```

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

Introduced in MCUXpresso IDE version 10.2.0: The inclusion of the hard fault handler can be controlled via a preference *preferences -> MCUXpresso IDE -> SDK Options -> Include semihost hardfault handler ...*, where the default is to include.

*Tip: Introduced in MCUXpresso IDE version 10.2.0: is the optional Redlib Semihost **MB** library variant. This library provides enhanced semihosting performance from LinkServer debug connections (other debug solutions perform as before) with the added benefit of no impact on code execution performance. There is a small penalty of slightly larger code and data sizes compared to other Redlib Semihost libraries. This optional library is recommended for users needing high semihosting performance and/or having slow debug probe performance.*

19.4.6 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

19.5 Use of printf

By default, the output from printf() (and puts()) is 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: If you only require the display of fixed strings, then using puts() rather than printf() noticeably reduces the code size of your application.

19.5.1 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.

19.5.1.1 Character vs string output

By default printf() and puts() functions output the generated string at once, so that a single semihosted operation can output the string to the console of the debugger. **Note**: these versions of printf() /puts() use malloc() to provide a temporary buffer on the heap to generate the string to be displayed.

It is possible to switch to using "character-by-character" versions of these functions (which do not require heap space) by specifying the build define "CR_PRINTF_CHAR" (which should be set at the project level). This can be useful, for example, if you are retargeting printf() to write out over a UART (as detailed below)- as in this case, it is pointless creating a temporary buffer to store the whole string, only to then print it out over the UART one character at a time.

19.5.1.2 Integer-only vs full printf (including floating point)

The printf() routine incorporated into Redlib is significantly smaller than that in Newlib. Therefore, 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 an "integer only" (non-floating point compatible) variant of printf. This further reduces code size.

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.

19.5.2 NewlibNano printf variants

By default, NewlibNano uses non-floating point variants of the printf and scanf family of functions, which can help to reduce dramatically 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.

19.5.3 Newlib printf variants

Newlib provides an "iprintf" function, which implements integer-only printf.

19.5.4 Printf when using LPCOpen

If you are building your application against LPCOpen, you may find that printf output does not get displayed in the debug console of MCUXpresso IDE 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 have to modify your projects so that:

1. Your main application project is linked to the "semihost" variant of the C library, and
2. You can 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

19.5.5 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 typically causes the output to be sent out via a UART (which may be connected to an on-board debug probe, which sends it back to the host over a USB VCOM channel). This is controlled by the macro **SDK_DEBUGCONSOLE**:

- If SDK_DEBUGCONSOLE == 0
  - PRINTF is directed to the 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 the 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, printf output is still 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.

19.5.6 Retargeting printf/scanf

By default, the printf function outputs text to the debug console using the "semihosting" mechanism.

In some circumstances, this output mechanism may not be suitable for your application. Instead, you may want printf to output via an alternative communication channel such as a UART or - on Cortex-M3/M4 - the ITM channel of SWO Trace. In such cases, you can retarget the appropriate portion of the bottom level of the library.

The section "How to use ITM Printf" below provides an example of how this can be done.

Note: when retargeting these functions, you can typically link against the "nohost" variant of the C Library, rather than the "semihost" one.

19.5.6.1 Redlib

To retarget Redlib's printf(), you have to provide your own implementation of the function __sys_write():

```
int __sys_write(int iFileHandle, char *pcBuffer, int iLength)
```

The function returns the number of unwritten bytes if error, otherwise 0 for success.

Similarly, if you want to retarget scanf(), you have to provide your own implementation of the function __sys_readc():

```
int __sys_readc(void)
```

The function returns the character read.

Note: these two functions effectively map directly onto the underlying "semihosting" operations.

19.5.6.2 Newlib / NewlibNano

To retarget printf(), you need to provide your own implementation of the Newlib system function _write():

```
int _write(int iFileHandle, char *pcBuffer, int iLength)
```

The function returns the number of unwritten bytes if error, otherwise 0 for success.

To retarget scanf, you need to provide your own implementation of the Newlib system function _read():

```
int _read(int iFileHandle, char *pcBuffer, int iLength)
```

The function returns the number of characters read, stored in pcBuffer.

More information on the Newlib system calls can be found at: https://sourceware.org/newlib/libc.html#Syscalls

19.5.7 How to use ITM printf

ITM Printf is a scheme to achieve application IO via a debug probe without the usual semihosting penalties.

19.5.7.1 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 use 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 the 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, see the MCUXpresso IDE LinkServer SWO Trace Guide.

19.5.7.2 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 causes the file retarget_itm.c to be generated in your project to carry out the redirection.

19.5.7.3 ITM printf with LPCOpen

To use this functionality with an LPCOpen project, you have to: Include the file `retarget_itm.c` in your project - available from the `Examples/Misc` subdirectory of your LinkServer 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 want just linking against the LPCOpen Chip library, then this is all you have to do. However, if you are also linking against an LPCOpen board library then you will likely see build errors in 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 enabling the line: #define DEBUG_SEMIHOSTING, or locating the source file board.c within the LPCOpen board library and commenting out the line: #include "retarget.h"

19.6 itoa() and uitoa()

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

19.6.1 Redlib

To ease porting, 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 the 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 the 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.

19.6.1.1 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
```

19.6.1.2 Standards compliance

As noted above, itoa() / uitoa() are not standard C library functions. A standard-compliant alternative for some cases may be to use sprintf() - though this is likely to cause an increase in the size of your application image:

```
sprintf(vstring,"%d",value); // convert to decimal
sprintf(vstring,"%x",value); // convert to hexadecimal
sprintf(vstring,"%o",value); // convert to octal
```

19.6.2 Newlib/NewlibNano

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

19.7 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 are handled automatically.

However, if you are not using the managed linker script mechanism, then you have to define which library files to use in your linker script. To do this, add one of the following entries before the SECTION line in your linker script:

- Redlib (None), add
  - [C project only]: GROUP (libcr_c.a libcr_eabihelpers.a)
- Redlib (Nohost), add
  - [C projects only]: GROUP (libcr_nohost.a libcr_c.a libcr_eabihelpers.a)
- Redlib (Semihost-nf), add
  - [C projects only]: GROUP (libcr_semihost_nf.a libcr_c.a libcr_eabihelpers.a)
- Redlib (Semihost), add
  - [C projects only]: GROUP (libcr_semihost.a libcr_c.a libcr_eabihelpers.a)
- NewlibNano (None), add
  - [C projects]: GROUP (libgcc.a libc_nano.a libm.a libcr_newlib_none.a)
  - [C++ projects]: GROUP (libgcc.a libc_nano.a libstdc++_nano.a libm.a libcr_newlib_none.a)
- NewlibNano (Nohost), add
  - [C projects]: GROUP (libgcc.a libc_nano.a libm.a libcr_newlib_nohost.a)
  - [C++ projects]: GROUP (libgcc.a libc_nano.a libstdc++_nano.a libm.a libcr_newlib_nohost.a)
- NewlibNano (Semihost), add
  - [C projects]: GROUP (libgcc.a libc_nano.a libm.a libcr_newlib_semihost.a)
  - [C++ projects]: GROUP (libgcc.a libc_nano.a libstdc++_nano.a libm.a libcr_newlib_semihost.a)
- Newlib (None), add
  - [C projects]: GROUP (libgcc.a libc.a libm.a libcr_newlib_none.a)
  - [C++ projects]: GROUP (libgcc.a libc.a libstdc++.a libm.a libcr_newlib_none.a)
- Newlib (Nohost), add
  - [C projects]: GROUP (libgcc.a libc.a libm.a libcr_newlib_nohost.a)
  - [C++ projects]: GROUP (libgcc.a libc.a libstdc++.a libm.a libcr_newlib_nohost.a)
- Newlib (Semihost), add
  - [C projects]: GROUP (libgcc.a libc.a libm.a libcr_newlib_semihost.a)
  - [C++ projects]: GROUP (libgcc.a libc.a libstdc++.a libm.a libcr_newlib_semihost.a)

In addition, if using NewlibNano, then the tick box method of enabling printf/scanf floating point support in the Linker pages of Project Properties is also not available. In such cases, you can enable 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.

20 Memory configuration and linker scripts

20.1 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 also specifies a Flash driver to be used to program that Flash memory (for use with LinkServer "native" debug probes).

For preinstalled MCUs, the definition of the memory map is contained within the MCU part knowledge that is built into the product. For MCUs installed into MCUXpresso IDE from an SDK, the definition of the memory map is loaded from the manifest file within the SDK structure.

But in both cases, the defined memory map is used by 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.

The memory map of a project can be viewed and modified by the user to add, remove (split/join), or reorder blocks using the in-place Memory Configuration Editor. For example, if a project targets an MCU that supports external Flash (for example, SPIFI), then its memory map can be easily extended to define the SPIFI memory region (base and size). In addition, an appropriate Flash driver can be associated with the newly defined region.
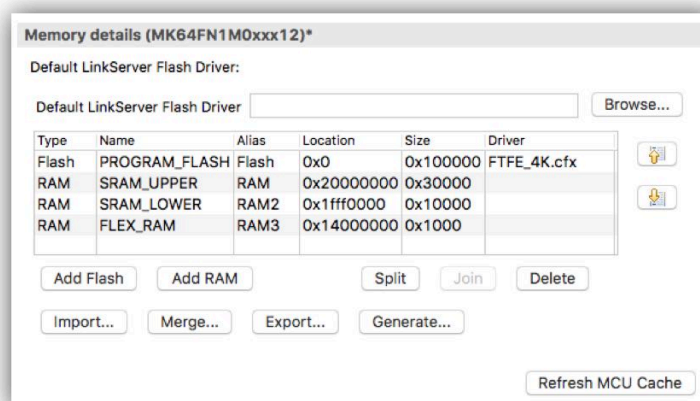
Figure 191. Memory configuration

Introduced in MCUXpresso IDE version 10.3.0 Memory configurations can be edited directly in place rather than requiring a separate *Edit* to launch a separate dialog. In place editing of memory configurations is incorporated within all project wizards and project properties views.

20.2 Managed linker script overview

By default, the use of "managed linker scripts" is enabled for projects. This mechanism allows MCUXpresso IDE to create automatically a script for each build configuration that is suitable for the MCU selected for the project and the C libraries being used. It creates (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
```

User guide **Rev. 4 — 14 January 2025** Document feedback

212 / 316

This set of hierarchical files is used to define the C libraries being used, the memory map of the system, and the way your code and data are placed into the memory map. These files are located in the build configuration subdirectories of your project (typically - Debug and Release).
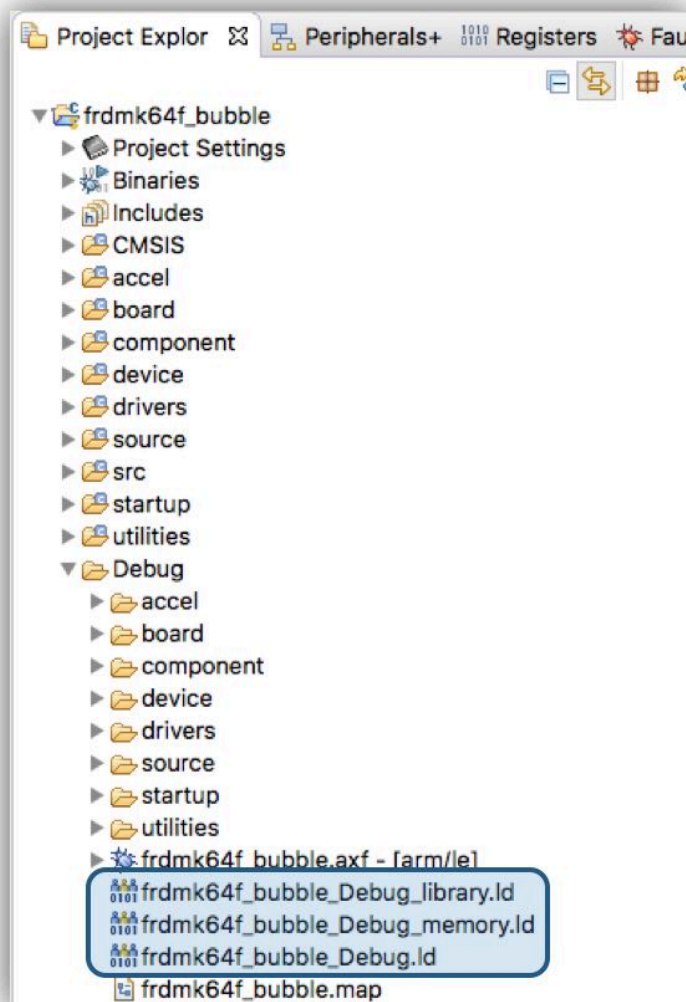
Figure 192. Project Explorer Debug folder linker scripts

The managed linker script mechanism also automatically considers memory map changes made in the Memory Configuration Editor and other configuration changes, such as C/C++ library settings.

See also the section on Heap and Stack view.

20.3 How are managed linker scripts generated?

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 controls 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* folder of the project directory structure. Any such templates that you provide locally then override the default ones built into MCUXpresso IDE. A full set of the default linker templates (.ldt)

files are provided inside */LinkServer/Wizards/linker* subdirectory of the IDE install. *LinkServer* is a symbolic link to the LinkServer installation folder.

20.4 Default image layout

Code and initial values of initialized data items are placed into the first bank of Flash (as shown in the memory configuration editor). During startup, MCUXpresso IDE startup code copies the data into the first bank of RAM (as shown in the memory configuration editor), and zero initializes the BSS data directly after this in memory. This process uses a global section table generated into the image from the linker script.

Other RAM blocks can also have data items placed into them under user control and the startup code also initializes these automatically. See later in this chapter for more details.
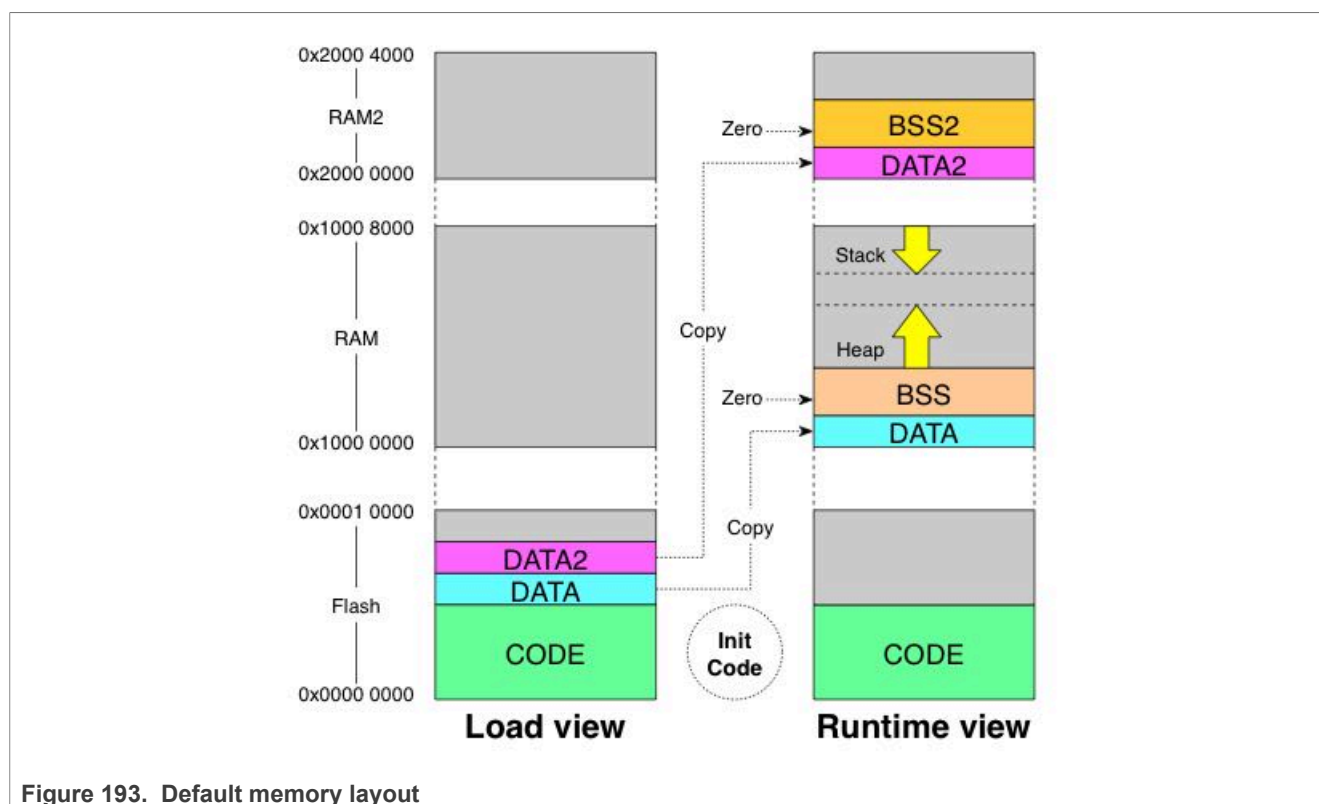
Figure 193. Default memory layout

Note: The above memory layout is simply the default used by the managed linker script mechanism of the IDE. 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 also locates the heap and stack in the first RAM bank, such that:

• the heap is located directly after the BSS data, growing upward through memory
• the stack located at the end of the first RAM bank, growing down toward the heap

Again, this heap and stack placement is a default and it is easy to modify the locations for a particular project, as 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.

20.5 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 occupies 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.

20.5.1 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:       25960 B         1 MB      2.48%
      SRAM_UPPER:        8472 B       192 KB      4.31%
      SRAM_LOWER:         0 GB        64 KB      0.00%
       FLEX_RAM:          0 GB         4 KB      0.00%
Finished building target: frdmk64f_bubble.axf
```

The memory regions displayed here 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 builds and links against the first Flash memory found within the memory configuration of the MCU. For most MCUs there is only one Flash device available. In this case, our project requires 25,960 bytes of Flash memory storage, 2.48% of the available Flash storage.

RAM is used for global variables, the heap, and the stack. MCUXpresso IDE provides a flexible scheme to reserve memory for Stack and Heap. This build has reserved 4 kB each for the stack and the heap contributing 8 kB to the overall 8472 bytes reported.

If using the *'LPCXpresso style'* of heap and stack placement (described later in this chapter), the RAM consumption provided by this feature is only that of your global data. It does not include any memory consumed by your stack and heap when your application is actually executing.

Note: A project imported into MCUXpresso IDE may not have been created with this option. To add this, right-click the project and select *C/C++ Build ->Settings -> MCU Linker -> Miscellaneous then click '+' and add --print-memory-usage*.

20.5.1.1 Comparing code size

This summary provides a quick method to *see* the usage of the memory regions and also changes in efficiency. Below are examples of Memory Usage for the same project compiled on an older version of MCUXpresso IDE vs the current version.

Code size with MCUXpresso IDE version 11.0.x:

```
Memory region         Used Size  Region Size  %age Used
    BOARD_FLASH:        40244 B        64 MB      0.06%
       SRAM_DTC:         8580 B       128 KB      6.55%
       SRAM_ITC:          0 GB       128 KB      0.00%
        SRAM_OC:          0 GB       256 KB      0.00%
    BOARD_SDRAM:          0 GB        32 MB      0.00%
Finished building target: evkbimxrt1050_bubble_peripheral.axf
```

Code size with MCUXpresso IDE version 11.1.x:

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

```
       BOARD_FLASH:        36192 B          64 MB       0.05%
         SRAM_DTC:          8580 B         128 KB       6.55%
         SRAM_ITC:             0 GB        128 KB       0.00%
          SRAM_OC:             0 GB        256 KB       0.00%
       BOARD_SDRAM:            0 GB         32 MB       0.00%
Finished building target: evkbimxrt1050_bubble_peripheral.axf
```

See the section on the Image information view for details on further image exploration.

20.5.2 arm-none-eabi-size

In addition, a post-build step normally invokes 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 does not include any memory allocated for 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 the file in Project Explorer and selecting the *Binary Utilities -> Size* option.

20.5.3 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 analyze in more detail the contents of your application image. When you do a build, this causes a file called *projectname.map* to be created in the Debug (or Release) subdirectory, which can be loaded into the editor view. This contains a large amount of information, including:

- A list of archive members (library objects) included with details
- A list of discarded input sections (because they are unused and the linker option --gc-sections is enabled)
- The location, size, and type of all code, data, and bss items that have been placed in the image

20.6 Image information (info)

The *Image Info* view provides tools for detailed analysis of an image structure and memory footprint.

The Image Info view is stacked by default in the MCUXpresso IDE Develop perspective, along with Problems and/or Console views.

The toolbar icons for this view are shown and detailed below:
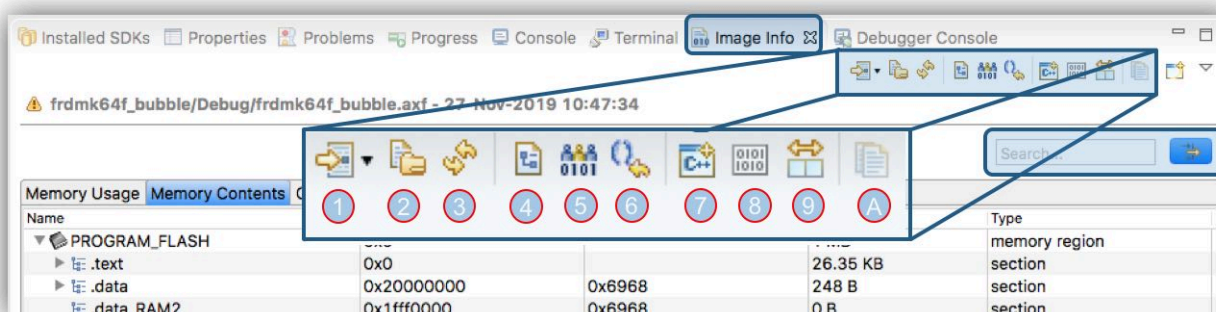
Figure 194. Image Info toolbar

Where:

1. Loads the build artifact (.axf) associated with the active build configuration of the currently selected project for analysis. This is the simplest option to follow to populate this view.
   • Alternatively, an image, object, or static library can be dragged onto this view
   • Once loaded, the selected artifact name and build information (plus warnings if any) are displayed as a title to the view
   • If more than one project (or file) is selected and more than one Image Info view is open within the IDE, then the additional views are also populated from the selection
   • Also, if more than one build configuration is available, the dropdown option allows *All build configurations* to be opened
   • This icon is grayed out if the active build configuration of the selected project has not been built
2. Browse to a build artifact containing symbolic information
3. Reload information from the currently loaded build artifact
   • This may be required when a project is rebuilt from changed sources
4. Open the Map file associated with the currently selected build artifact
   • This file opens up within the editor view where enhanced syntax highlighting helps navigation
5. Open the Linker Script (.ld file) associated with the currently selected build artifact
   • This file opens up with the editor view where enhanced syntax highlighting helps navigation and understanding
6. Locate the main symbol if present in the current tab
7. Enable/Disable C++ name mangling
   • This uses the c++filt binutils application to demangle C++ symbols from the view
     – All (mangled) items from the view are affected - not only the current selection
8. Toggle between sizes in bytes and larger units (kB, MB, and so on.)
9. Click to compare with contents from another (new) Image Info view using the standard Eclipse compare utility
   • To use this feature, create a second Image Info view and load with another image, object, and so on, click compare in both views
10. (A) Copy highlighted information to the clipboard
    • Copied information is held in .tsv format with the table headers added to the selection

Tip: These options are also available from a right-click menu within the Image Info view.
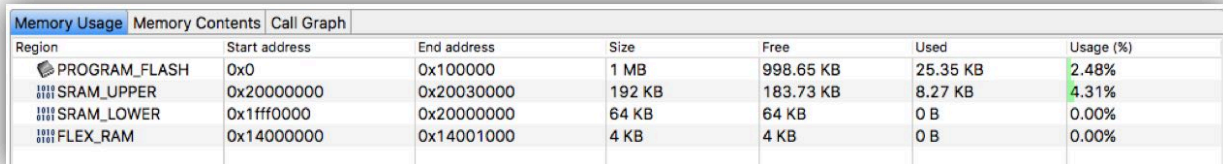
Also highlighted is the search/filter button, this can be used to switch between the highlighting of lines containing an entered search item and only displaying matching lines. This feature can be useful to remove clutter from large groups of items.

Note: Information from the highlighted lines is shown in the Properties view.

The Image Information view (usually) consists of three subviews offering - Memory Usage, Memory Contents, and (static) Call Graph information.

20.6.1 Memory usage

The Memory Usage view shows how much memory (Flash and RAM) is used by the associated build artifact.

Region	Start address	End address	Size	Free	Used	Usage (%)
PROGRAM_FLASH	0x0	0x100000	1 MB	998.65 KB	25.35 KB	2.48%
SRAM_UPPER	0x20000000	0x20030000	192 KB	183.73 KB	8.27 KB	4.31%
SRAM_LOWER	0x1fff0000	0x20000000	64 KB	64 KB	0 B	0.00%
FLEX_RAM	0x14000000	0x14001000	4 KB	4 KB	0 B	0.00%

Figure 195. Image Info memory usage

The memory regions displayed are the same as the build artifact of the selected project (typically the generated elf (.axf) file of a project). The detailed information is broadly the same as that provided by the Linker --print-memory-usage switch however, this view can be used to compare easily memory usage from one build to another following code changes, improvements, different build configurations, and so on.

Tip: *As a guide the memory usage % display is colored green when more than half of the available memory is free, then changing from yellow to red if more memory is used.*
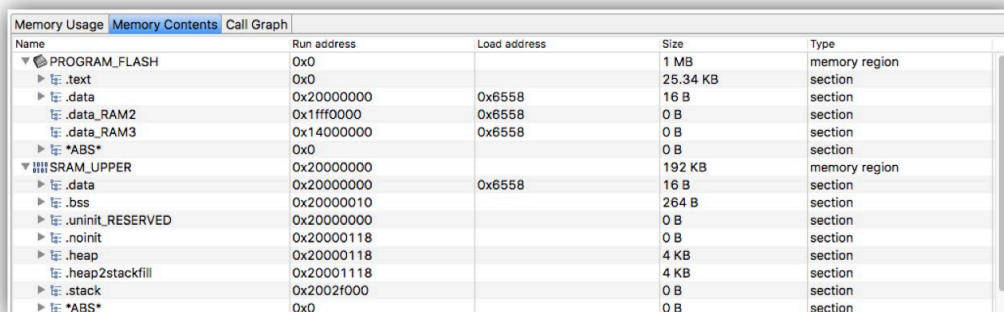
Note: The Memory Usage tab is not displayed in the following situations:

• A not-yet-linked file (*.o) was processed
• A static library (*.a) was processed
• A build artifact from outside the current workspace was processed - memory regions cannot be obtained in this case

Double-click a Memory region to jump its *Contents*.

20.6.2 Memory contents

The Memory Contents view provides a detailed view of the contents of each memory region. The image below shows various linker sections distributed within the memory regions.

Name	Run address	Load address	Size	Type
PROGRAM_FLASH	0x0		1 MB	memory region
.text	0x0		25.34 KB	section
.data	0x20000000	0x6558	16 B	section
.data_RAM2	0x1fff0000	0x6558	0 B	section
.data_RAM3	0x14000000	0x6558	0 B	section
ABS	0x0		0 B	section
SRAM_UPPER	0x20000000		192 KB	memory region
.data	0x20000000	0x6558	16 B	section
.bss	0x20000010		264 B	section
.uninit_RESERVED	0x20000000		0 B	section
.noinit	0x20000118		0 B	section
.heap	0x20000118		4 KB	section
.heap2stackfill	0x20001118		4 KB	section
.stack	0x2002f000		0 B	section
ABS	0x0		0 B	section

Figure 196. Image Info memory contents

Double-clicking or pressing the *Enter* key on any selected symbol opens its definition.
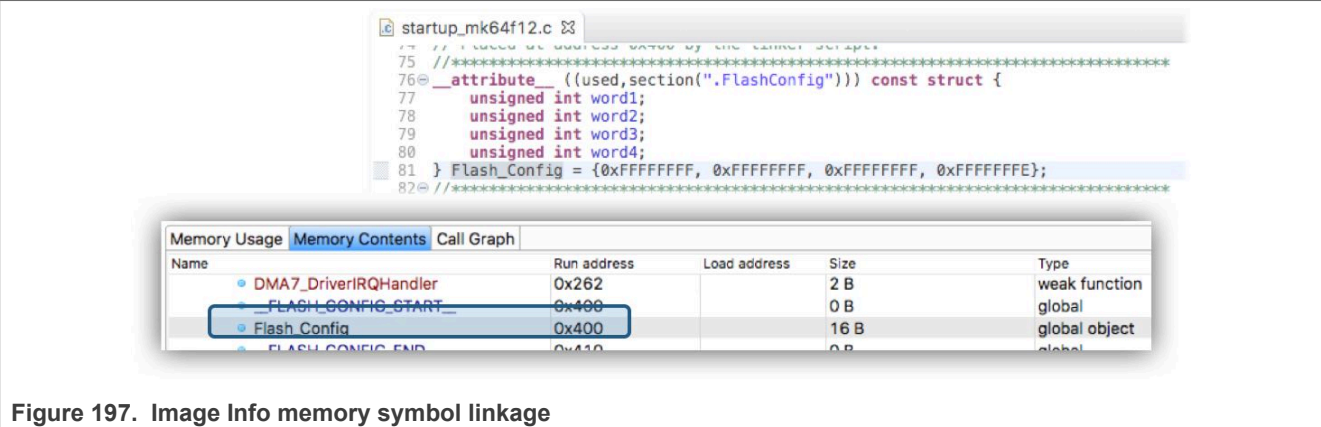
Figure 197. Image Info memory symbol linkage

Note: If a symbol cannot be found within the sources, for example, the symbol is within a C library function, a message is displayed in the Eclipse status bar.

Selecting multiple lines within this view totals their memory usage.
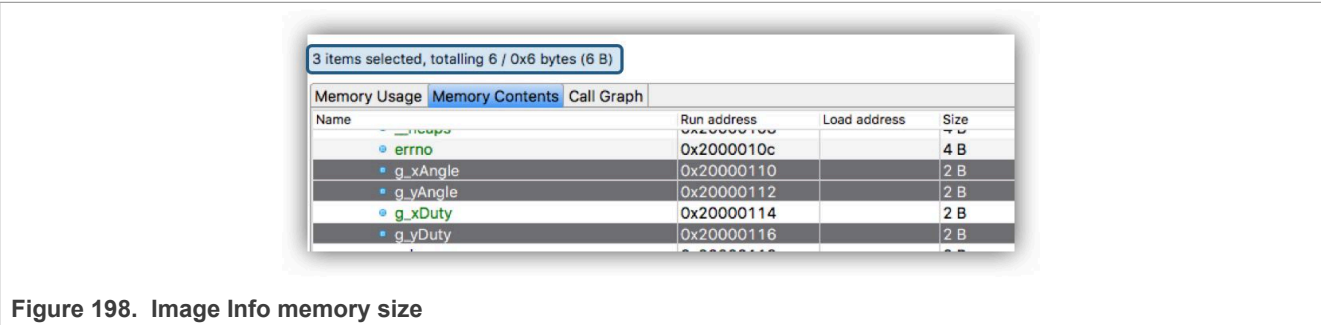
Figure 198. Image Info memory size

Note: When selecting multiple symbols, the sum of their individual sizes is computed without considering any space that was used for padding or alignment within the section. As a result, the actual section size might differ compared to the size of a multiple symbols selection. The real section size within the application is the size displayed next to the section name, in the appropriate table column.

20.6.3 Call graph

The Call Graph tab shows the static stack cost for the selected build artifact as generated via the -fstack-usage compiler option. The generation of Stack Usage information is now a default option within MCUXpresso IDE version 11.0.0 but can be controlled via the Workspace project property shown below:
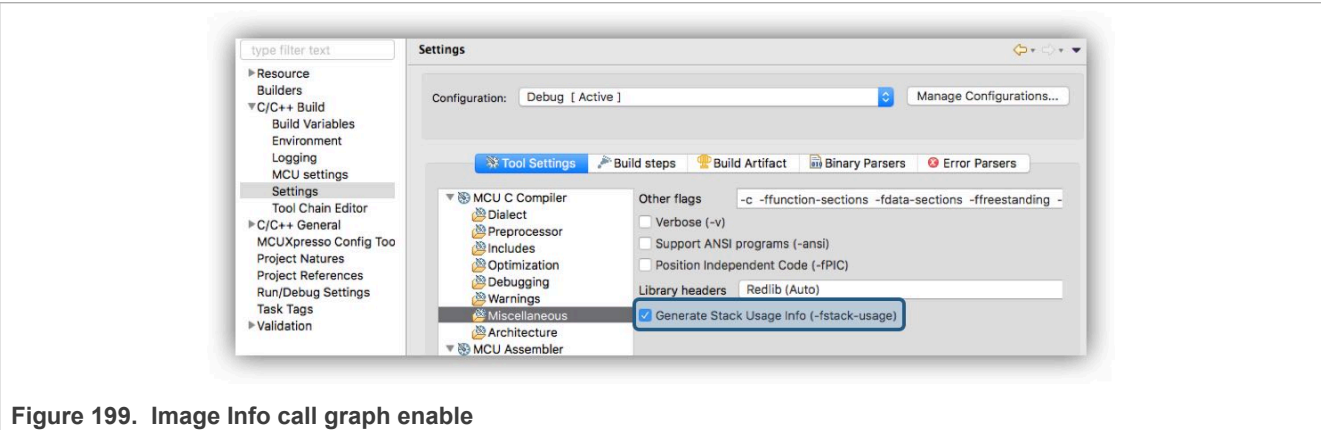
Figure 199. Image Info call graph enable

This option enables the generation of .su (stack usage) files by the compiler and these are consumed (along with other information) to populate the Call Graph view. **Note**: the generation of these additional files has minimal impact on project build times.

If a project has been built and loaded, the call graph information for the selected build configuration is available. Below is a truncated view of a call graph display, expanded and highlighted to display the main() function.

Figure 200. Image Info call graph

In this view, the columns have the following meaning:

- Function: displays the function name
- Depth: displays the maximum call depth
  - Where N means that the function has at least 1 child with a depth of N-1
  - And 0 means there are no child functions
- Location: function location within the source (file:line)
  - This is empty if no source is found
- Type: show static or dynamic allocation type
- Local Cost: shows the number of bytes allocated by the function itself
- Full Cost: shows the number of bytes allocated by the function itself plus that of the deepest child function
- Comment: shows additional information such as recursive calls

Within the view, symbols are colored to convey meaning, as follows:

Figure 201. Image Info call graph display types

1. A symbol in black can be double-clicked to open the associated source code
2. A symbol in gray has no associated source information
   - This might indicate an assembly or library symbol
3. A symbol with circular arrows indicates it has a recursive call so its stack costs cannot be added to the full cost
4. Exception handlers in gray (not shown) group any root symbol with a *Handler* suffix

Finally, if for any reason Call Graph information is limited or stale, clear self-describing warnings are displayed.

20.6.4 Use of filters

The search filter now supports both simple and regular expression search.

Below a filter for the symbol *main* locates '__main' and 'main'.
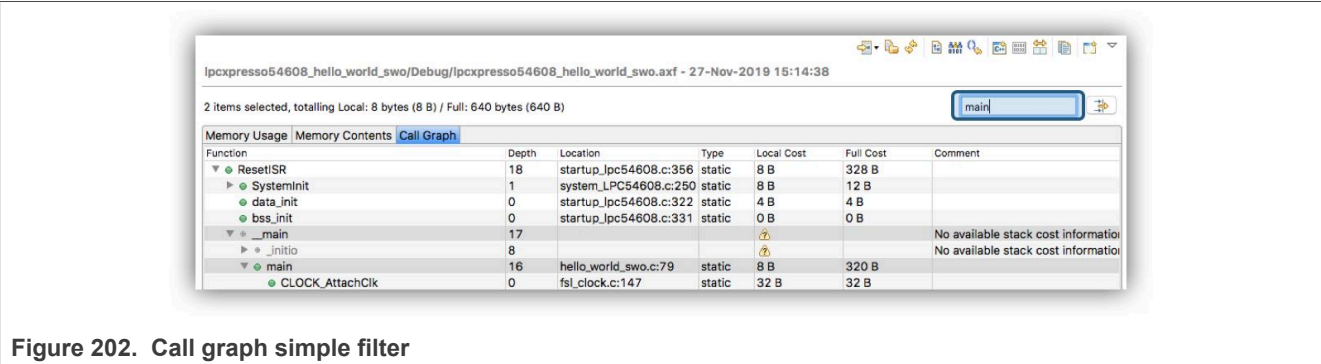
Figure 202. Call graph simple filter

Regular expression filter supports standard regex searching...
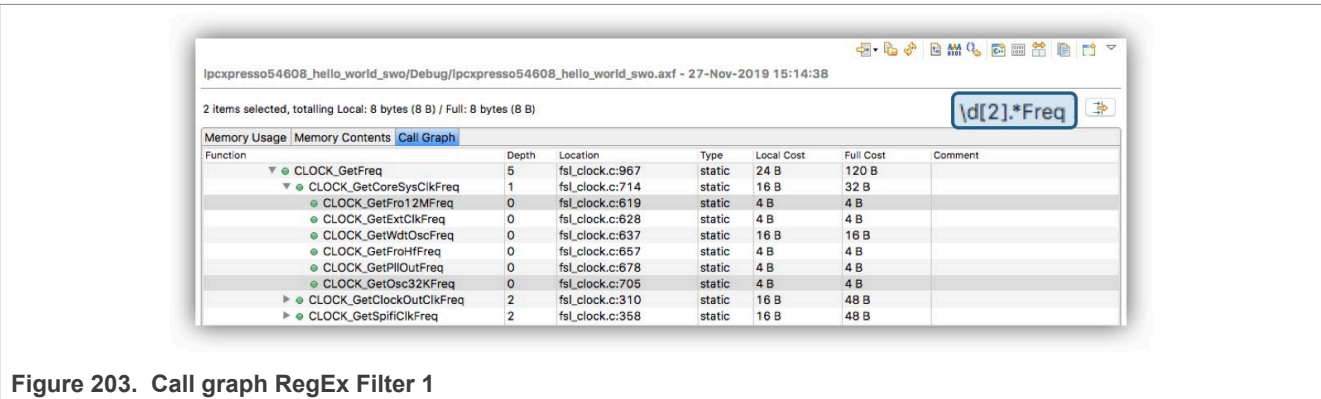
Figure 203. Call graph RegEx Filter 1

Use of NOT searching - search for CLOCK but not containing Xtal ...
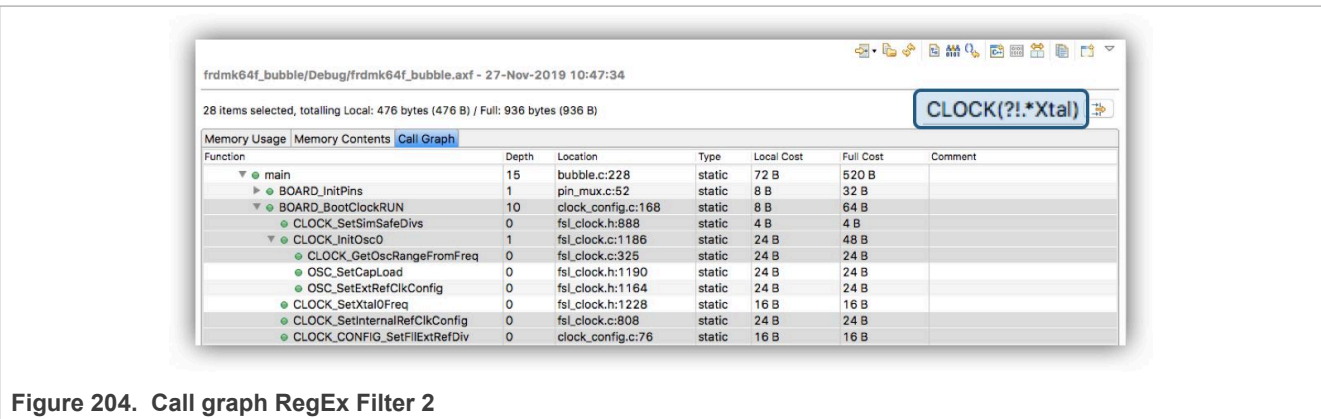
Figure 204. Call graph RegEx Filter 2

Note: If an error occurs when entering a regular expression, the message becomes red as you type and the tooltip indicates the expression error.

Tip: Typically, to search for a string within a regular expression, you would write (.*)string(.*) ... To remove this requirement from users, strings are guarded by default at the beginning and end resulting in a search

for anything containing the string. A side effect of this guard is that you can't search for something starting or ending with '.'.

20.7 Enhanced syntax highlighting

Introduced in MCUXpresso IDE version 11.0.0, additional editor capability delivering Enhanced Syntax Highlighting for GNU Linker Script .ld files (also Linker Script template and .map files). The primary goal of these enhancements is to simplify the exploration of these files and also ease the manual creation of Linker Script files for situations where the autogenerated linker script mechanism of MCUXpresso IDE cannot support the required configuration.

The new editor is invoked automatically by double-clicking the .ld, .ldt, or .map file within the project explorer view. If needed, this functionality can be disabled via *Preferences -> MCUXpresso IDE -> Editor Awareness*, the changes taking effect after restarting MCUXpresso IDE.

Note: To ensure that enabling and disabling the editors work as expected, MCUXpresso IDE should be launched in clean mode. This can be done either by calling the IDE executable from the command line with the *-clean* argument or by adding *-clean* on the first line of the *.ini* configuration file (which can be found in the same folder as the MCUXpresso IDE executable).
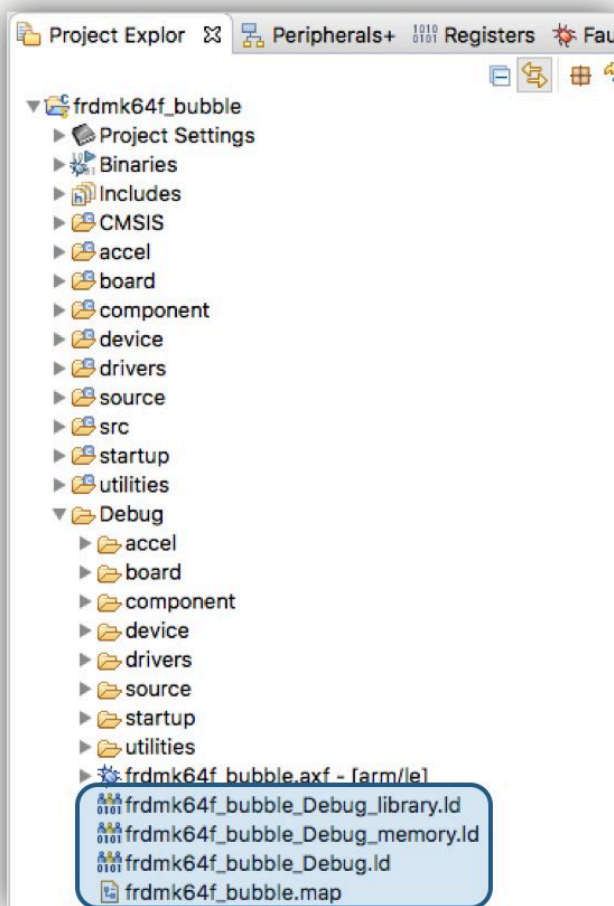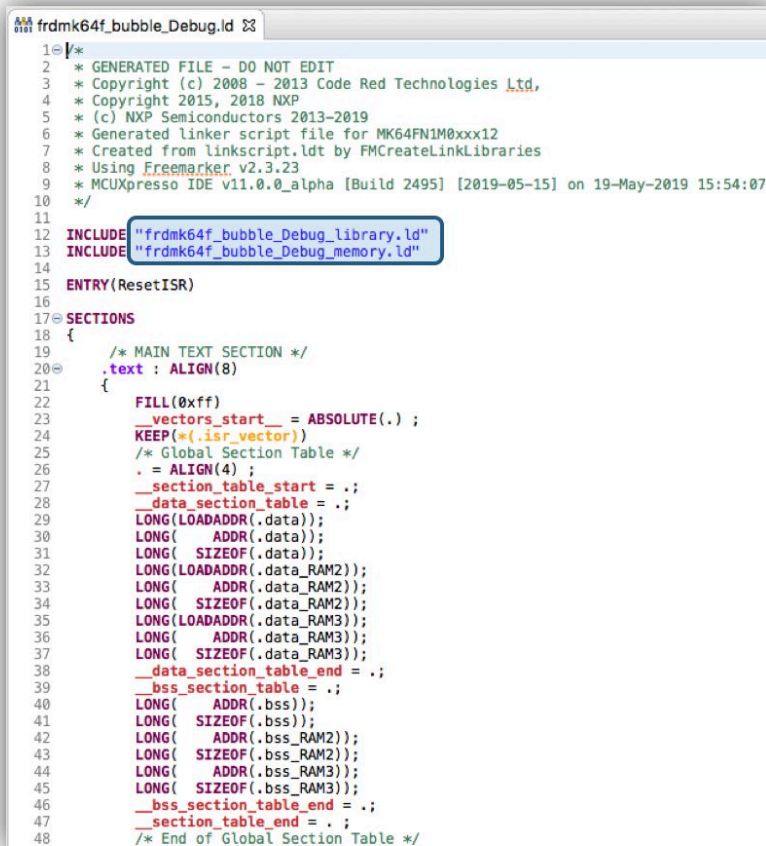
Figure 205. Project build configuration files

Note: these files are automatically generated by the Managed linker script mechanism for the selected build configuration when a project is built

Once a file is opened as below, a number of features are available.

Figure 206. Linker description file

Include files and Symbols source (as highlighted) can be opened in a new editor view via Ctrl + Click (Cmd + Click for Mac) on their filename.

The Editor also provides context-aware code completion accessible by pressing Ctrl + Space.
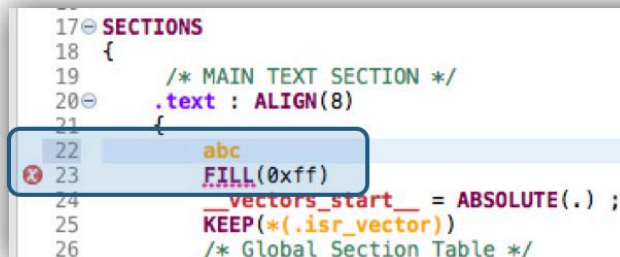
Figure 207. Auto completion
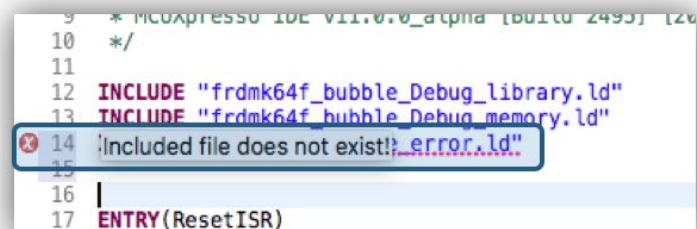
The editor also provides error checking - validating that any changes are in accordance with the linker script syntax.

Figure 208. Error checking syntax

Furthermore, INCLUDE paths are verified and any error is shown as below.

Figure 209. Error checking files

Error markers are shown on the navigation bar and in the title of the editor window.

The Outline view displays an outline of the file that is currently open in the editor area.
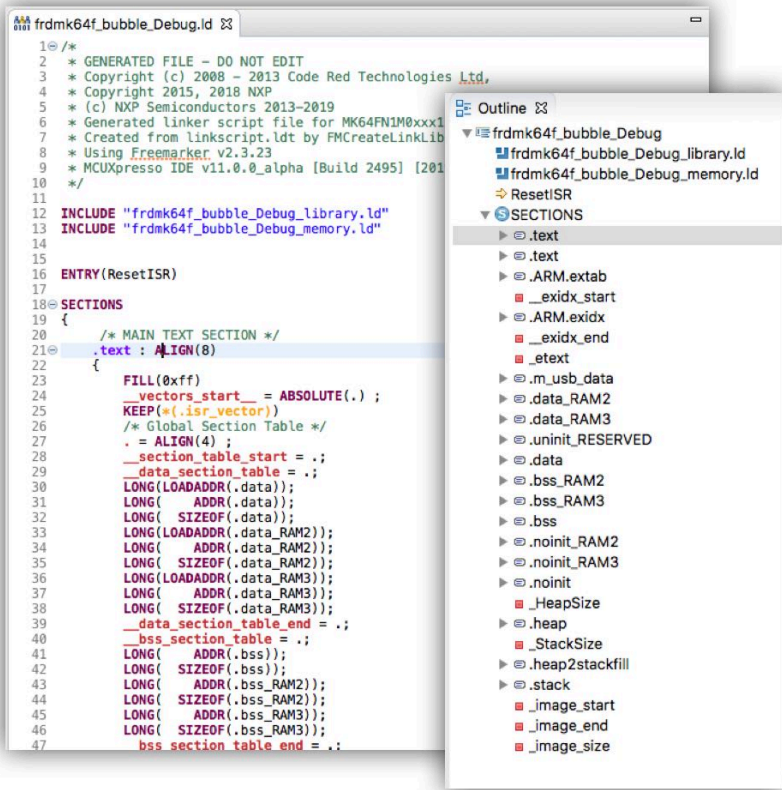
Figure 210. Linker description Outline association ld

This is useful for navigation through complex autogenerated .map files
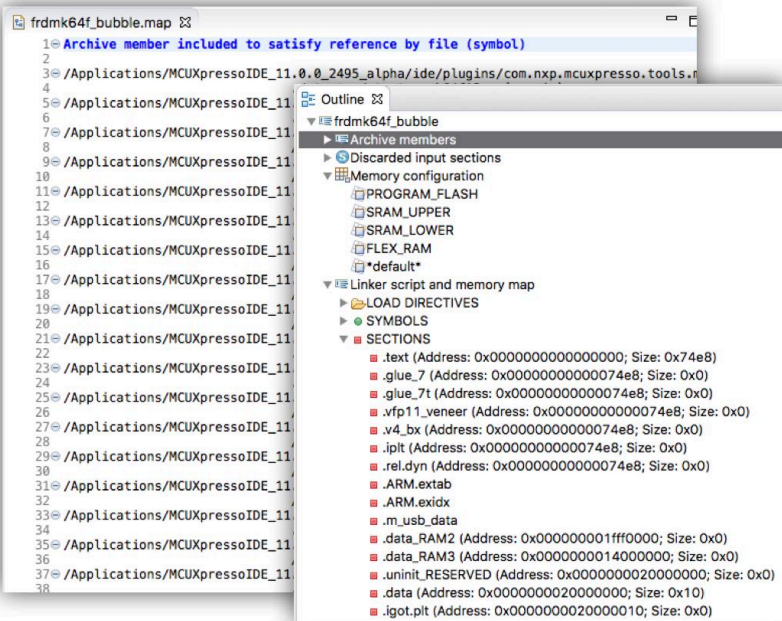
Figure 211. Map file Outline association map

Tip: *Right-clicking within the outline view allows the opening of related source files.*

Finally, if required, the colors used for syntax highlighting can be configured via *Preferences -> MCUXpresso IDE -> Editor Awareness* as below.
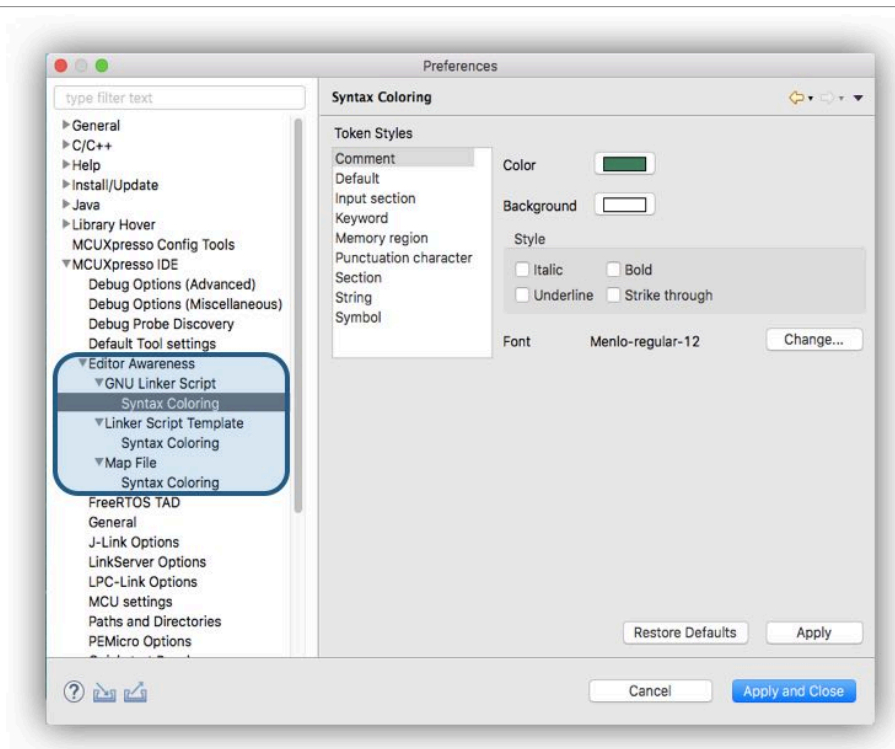
Figure 212. Editor awareness preferences for syntax coloring

20.8 Other options affecting the generated image

20.8.1 LPC MCUs - Code Read Protection

Most of NXP's LPC Cortex-M-based MCUs, which have internal Flash memory, contain "Code Read Protection" (CRP) support. This mechanism uses one of a number of known values being placed in a specific location in Flash memory to provide several 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.

20.8.1.1 CRP

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 requirements of the user.

The New Project wizard contains an option to allow linker support for placing a CRP word to be enabled when you create a new project. This is typically enabled by default. This wizard option actually then controls the "Enable CRP" checkbox of the Project Properties linker Target tab.

In addition, the wizard creates a file, 'crp.c' which defines the 'CRP_WORD' variable, which contains the required CRP value. A set of possible values is provided by the NXP/crp.h header file that this then includes. Therefore, for example, 'crp.c' typically contains:

```
#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 have to modify the CRP word to contain an appropriate value.

Important Note: 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 using CRP, you are advised to refer to the User Manual for the LPC MCU that you are using.

20.8.1.2 CRP

The support for CRP in LPC parts imported into MCUXpresso IDE from an SDK is similar to the Preinstalled MCUs. However, rather than having a separate crp.c file, the CRP_WORD variable definition is found within the startup code.

20.8.2 Kinetis MCUs - Flash Config Blocks

Kinetis MCUs provide an alternative means of protecting the user's image in Flash using the Flash Configuration Block. The Flash Configuration Field is located at addresses 0x400-0x40F and unlike the LPC CRP mechanism, only specific values give access, whereas any other values are likely to lock the part.

The value of the Flash Configuration block for a project is provided by the following structure, which will be found in the startup code:

```
__attribute__ ((used,section(".FlashConfig"))) const struct {
    unsigned int word1;
    unsigned int word2;
    unsigned int word3;
    unsigned int word4;
} Flash_Config = {0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFE};
```

which is then placed appropriately by the linker script generated by the managed linker script mechanism.

```
/* Kinetis Flash Configuration data */
. = 0x400 ;
PROVIDE(__FLASH_CONFIG_START__ = .) ;
KEEP(*(.FlashConfig))
PROVIDE(__FLASH_CONFIG_END__ = .) ;
ASSERT(!(__FLASH_CONFIG_START__ == __FLASH_CONFIG_END__),
 "Linker Flash Config Support Enabled, but no .FlashConfig
 section provided within application");
/* End of Kinetis Flash Configuration data */
```

Important Note: The support for placing the Flash Configuration Block can be disabled by unticking a checkbox of the Project Properties linker Target tab. However, this is not advisable as it is likely to result in a locked MCU.
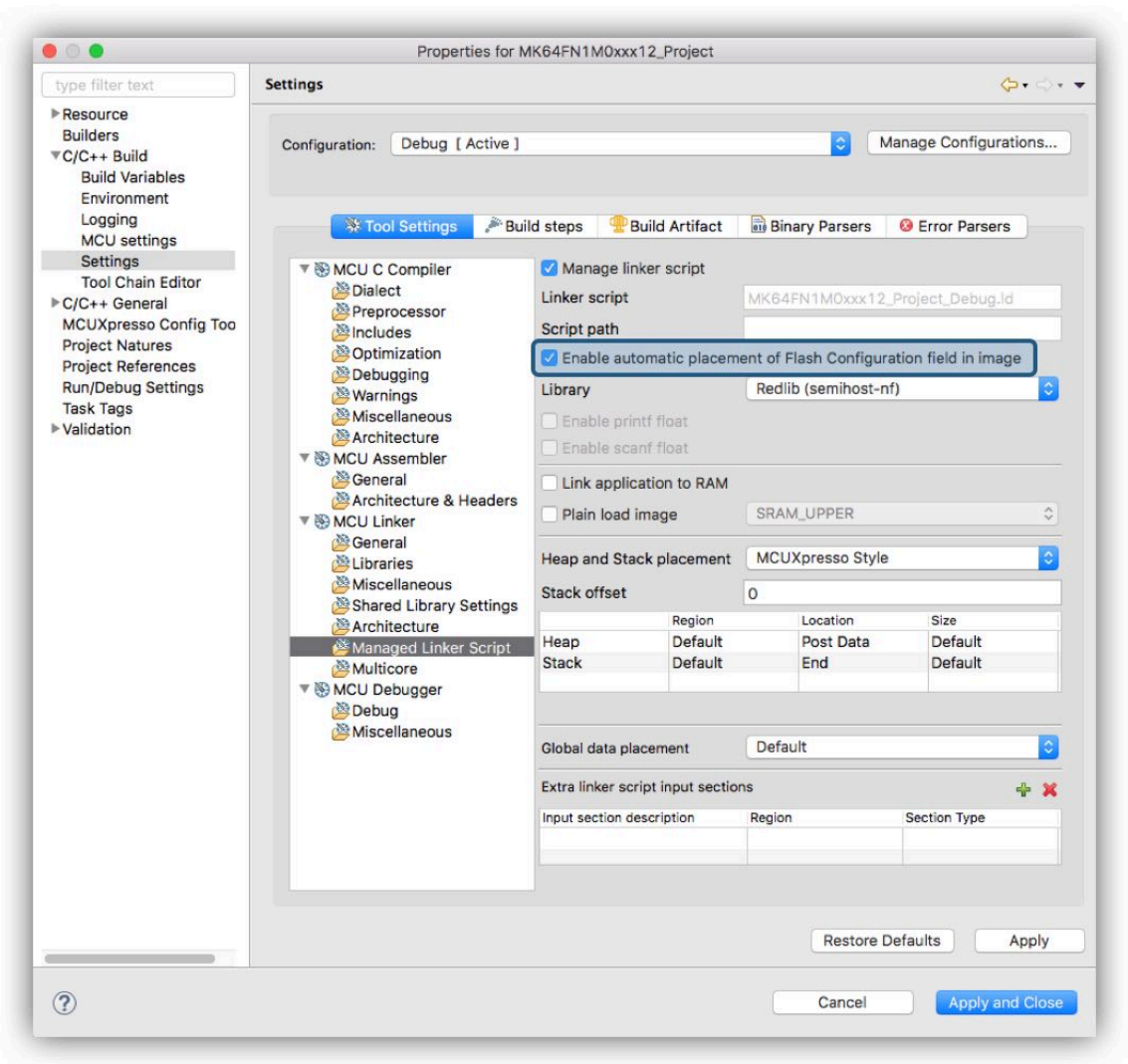
Figure 213. Linker settings

20.8.3 Placement of USB data

For MCUs where part support is imported from an SDK, the managed linker script mechanism supports the automatic placement of USB global data (as used by the SDK USB Drivers), including for parts with dedicated USB_RAM (small or large variants).

20.8.4 Plain load image

The LPC540xx family provides no built-in flash, but rather offers a quad SPI Flash Interface (SPIFI) so that external flash can be used. The most straightforward way of using the external flash is that the image is built to be programmed into the external flash and executed directly from the same location (XIP - eXecute In Place).

However, the LPC540xx boot ROM also offers an alternative way of using the external flash - such that the application is programmed into the flash, but the boot ROM relocates it into a bank of the onboard SRAM for

execution. Generally, it is expected that the SRAMX bank (at address 0x0) is used for this. An application that runs in this manner is known as a "plain load image".

The managed linker script mechanism of MCUXpresso IDE offers a simple way of configuring an application project so that it builds as a plain load image. This can be controlled for a particular build configuration via:

Project -> Properties -> C/C++ Build -> Settings -> Tools Settings -> MCU Linker -> Managed Linker Script.
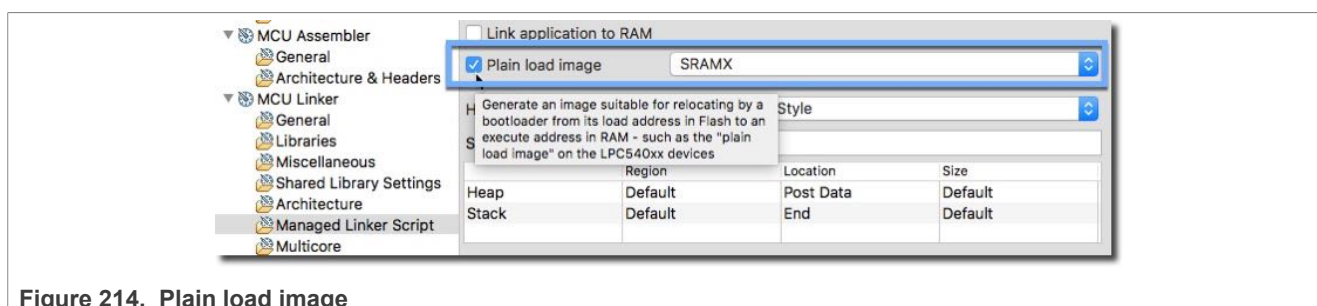
Figure 214. Plain load image

See also the Project settings shortcuts.

Enabling the "Plain load image" option:

1. Modifies the generated linker script so that the main code section is located so that it is programmed into flash, but expect to be copied into the specified RAM bank by the boot ROM before being executed
2. Modifies the startup code, using symbols provided from the generated linker script, so that the appropriate data is placed into the image so that the boot ROM knows that it has to relocate the image from flash into RAM.

Note 1: This functionality requires the application project to be based on the LPC540xx part support from SDK v2.4.0 (or later).

Note 2: The size of the application image (including the initialized global data) must be less than the size of the RAM bank that the code executes from.

Note 3: LPC540xx supports plain load images being executed from either address 0x0 or address 0x20000000. However, if the RAM at 0x20000000 is used then the debugger is not able to stop on the default breakpoint on main(). This is because a hardware breakpoint has to be used (as the copying of the code from flash into RAM by the boot ROM would overwrite a software breakpoint), but the Cortex-M4 cannot set a hardware breakpoint this high in the memory map.

20.8.5 Link application to RAM

The MCUXpresso IDE *managed linker mechanism* defaults to placing the code and initialized data values to the first Flash region listed within the memory configuration of a project, as discussed in the Default image layout section.

On occasion, it can be useful to debug a project directly from RAM since this offers some benefits such as avoiding the flash programming element of the debug session, and so on. Linking to RAM could be achieved by deleting the Flash memory regions from the memory configuration of the project and rebuilding the application - however, this is not the most convenient approach!

Therefore, MCUXpresso IDE offers the option to *tell* the managed linker script mechanism to simply ignore any flash regions listed in the memory configuration of the project via a simple checkbox at:

Project -> Properties -> C/C++ Build -> Settings -> Tools Settings -> MCU Linker -> Managed Linker Script
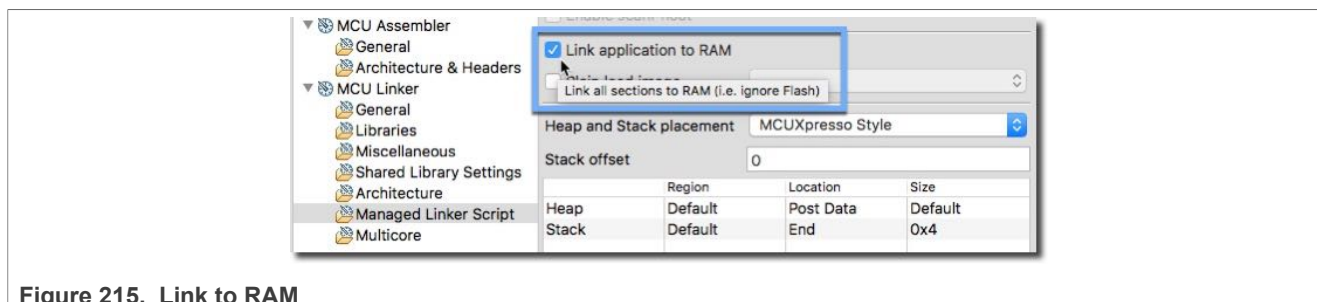
Figure 215. Link to RAM

See also the Project settings shortcuts.

With this option set, the application instead links to the first RAM region listed within the memory configuration of the project.

There are two important considerations when developing with RAM based projects:

1. They require support from the debug environment to be run and so may not execute in the same manner as a true application running from an MCU reset. See the section RAM projects with LinkServer for more information. **Note**: if you are using debug solutions other than LinkServer, additional user setup may be required.
2. Unlike projects running from Flash, global variable load and execute addresses are by default the same. The consequence of this is that global variables values persist at their current value if an application is restarted. Therefore, this is not recommended, and instead, a restart should be achieved by terminating and restarting the whole debug session. See also: Placement of specific code/data items

Note: Some MCU/development boards use SDRAM. These memories are typically initialized by the MCU BootROM during reset and this initialization may require user-supplied configuration data to be programmed into flash. Therefore, you **must** ensure that any SDRAM regions are correctly initialized before they are used for RAM-based debug operations.

20.9 Modifying the generated linker script / memory layout

The linker script generated by the managed linker script mechanism is 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.

20.10 Using the Memory Configuration Editor

The Memory Configuration 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 displays, by default, the memory regions that have been defined by MCUXpresso IDE itself (from installed or SDK part support).
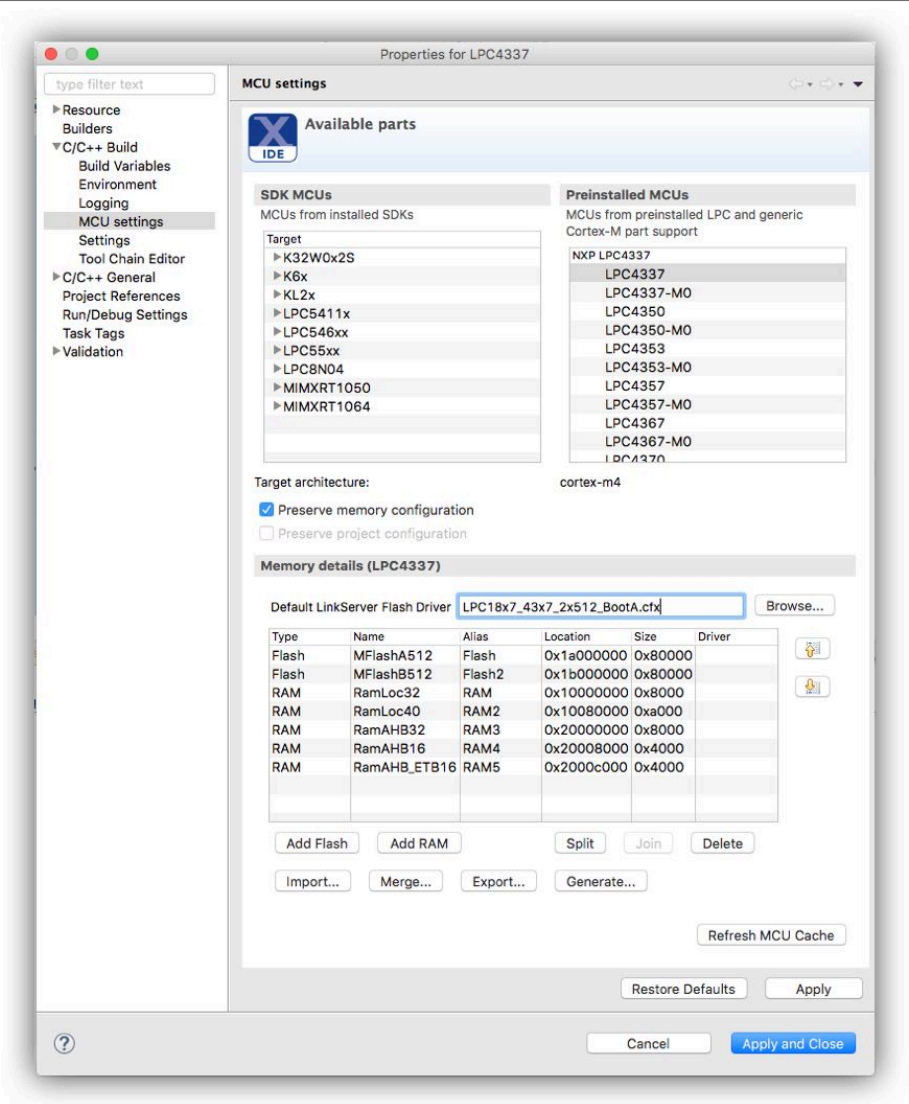
Figure 216. LPC4337... default memory regions

20.10.1 Editing a memory configuration

In the example below, we show how the default memory configuration for an LPC4337... can be changed.

Introduced in MCUXpresso IDE version 10.3.0, the memory configuration can simply be edited in place to create the desired memory map.
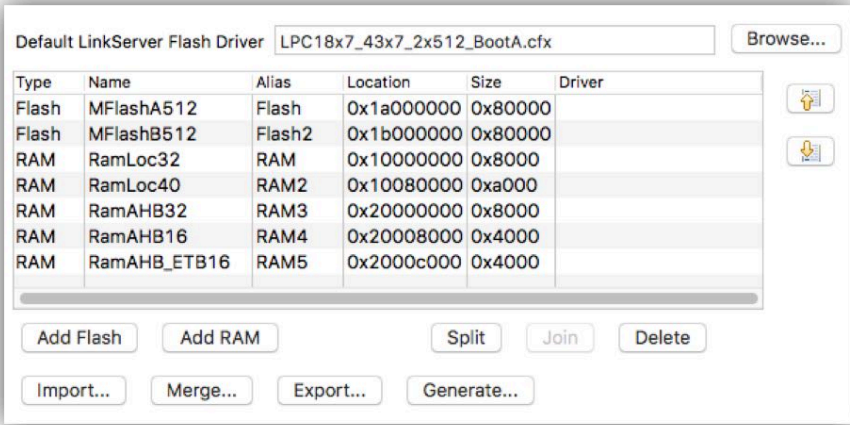
Figure 217. Memory configuration editor

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

For simplicity, the **additional** memory regions are given sequential aliases, starting from 2, so RAM2, RAM3, and so on (as well as using their "formal" region name - for example, RamAHB32).

Table 5. 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 the code is placed in the first RAM block, and data is placed in the block following the one used for the code (regardless of whether the code block was RAM or Flash).
Generate	Generates local part support for the selected MCU.
Driver	Highlighted in blue, shows the selection of a per-Flash region Flash driver. Click this field to see a dropdown of all available drivers. See: LinkServer Flash support
Browse(Flash driver)	Select the appropriate driver for programming the Flash memory specified in the memory configuration. For more information, see the section on flash drivers

The name, location, and size of this new region can be edited in place. **Note**: 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 32 kB, enter `32768`, `0x8000` or `32k`

- To enter a region size of 1 MB, enter `0x100000` or `1m`

Note: Memory regions must be located on four-byte boundaries, and be a multiple of four bytes in size.

The screenshot below shows that the dialog after the "Add Flash" button has been clicked. Use the highlighted up/down buttons to move this region to be top of the list. This action forces the managed linker script mechanism of MCUXpresso IDE to link against this new flash region.
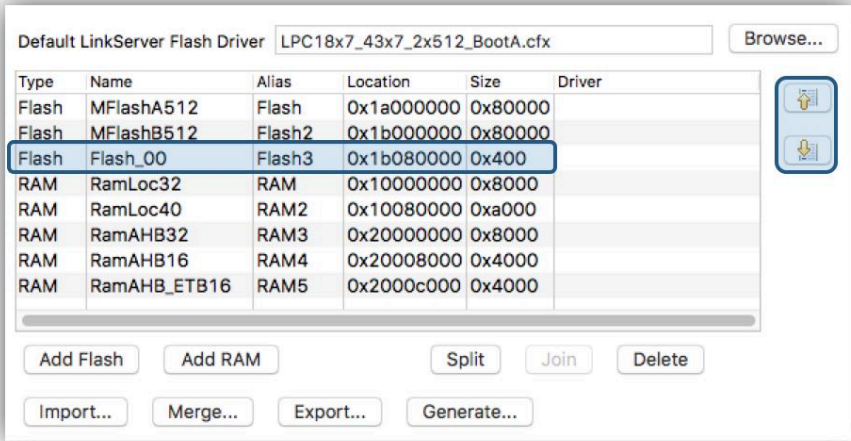
Figure 218. Effect of Add Flash

Tip: Once a change has been made, ensure that a mouse click is made outside any changed cell, this action forces the change to be recognized by Eclipse.
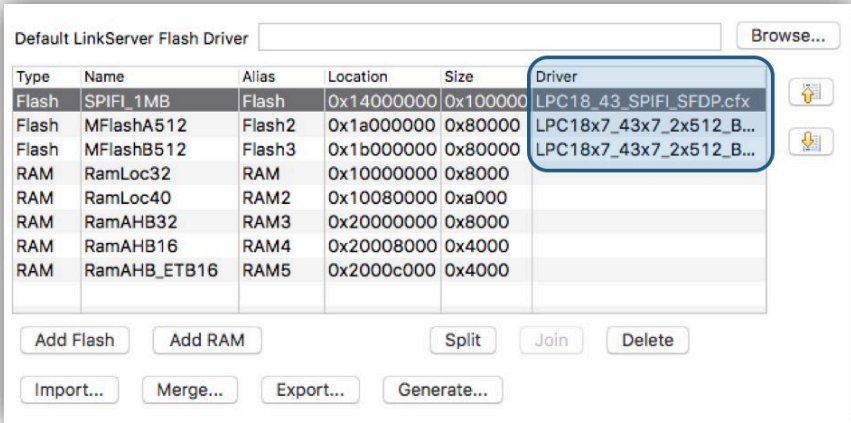
Figure 219. Updated MCU settings

Here you can see that the new region has been named SPIFI_1MB, its base address set to 0x14000000, its size to 1 MB and the default Flash driver has been deleted and an SFDP 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: Once the memory details have been modified, the selected MCU as displayed on the "Status Bar" (at the bottom of the IDE window) is displayed with an asterisk (*) next to it. This provides an indication that the MCU memory configuration settings for the selected project have been modified.

20.10.2 Device-specific vs default Flash drivers

When a project is configured to use additional Flash devices via the Memory Configuration Editor, the Flash driver to be used for programming that Flash device has to be specified in the Driver column. Typically for a SPIFI device, this should be:

- *LPC18_43_SPIFI_GENERIC.cfx* (for *LPC18/LPC43* series MCUs)
- *LPC40xx_SPIFI_GENERIC.cfx* (for *LPC407x/8x* MCUs)
- *LPC5460x_SPIFI_GENERIC.cfx* (for *LPC5460x* MCUs)
- *LPC540xx_SPIFI_GENERIC.cfx* (for *LPC540xx* MCUs)

For further information, also see the section on [flash drivers](#).

20.10.3 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.

20.10.4 Copying Memory Configurations

Memory configurations can be exported for import into another project. Use the Export and Import buttons for this purpose.

20.11 Global data placement

By default, global data items are located at run time in the 'default' memory region (that is, the first RAM block displayed in the memory configuration area).

However, MCUXpresso IDE version 10.2 introduced a mechanism to the Managed Linker Script mechanism to allow the user to specify a specific memory region to be used for the global data, without the need to change the order of the RAM blocks in the memory configuration editor.

This can be done via the Managed Linker Script page of Project Properties:
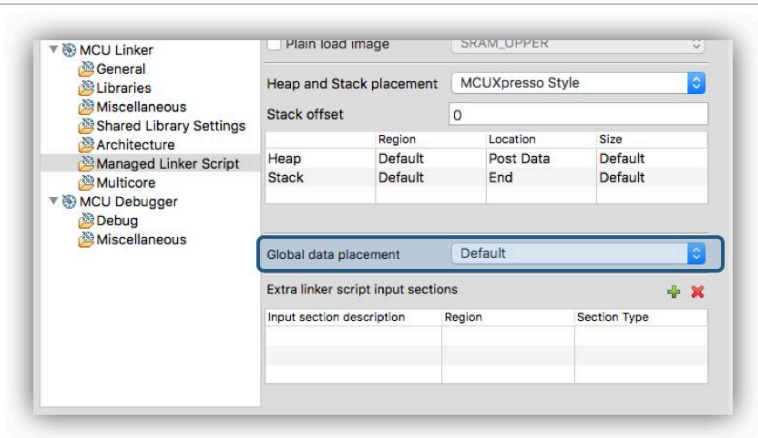
Figure 220. MCUXpresso IDE global data placement

To change the memory region to be used, simply use the dropdown box to select the memory region you wish to locate the global data.

Note: the above placement of global data applies to global data items that are not explicitly placed elsewhere in the memory map see: Placement of specific code/data items.

20.12 Modifying heap/stack placement

MCUXpresso IDE provides two models of heap/stack placement. The first of these is the "LPCXpresso Style", which is the mechanism provided by the previous generation LPCXpresso IDE. This is the default model used for projects created for Preinstalled MCUs. The second model is the "MCUXpresso style". This is the default model used for projects created for MCUs imported from SDKs.

The heap/stack placement model being used for a particular project/build configuration can be modified by right-clicking the project and selecting:

Project Properties -> C/C++ Build -> Settings -> MCU Linker -> Managed Linker Scripts
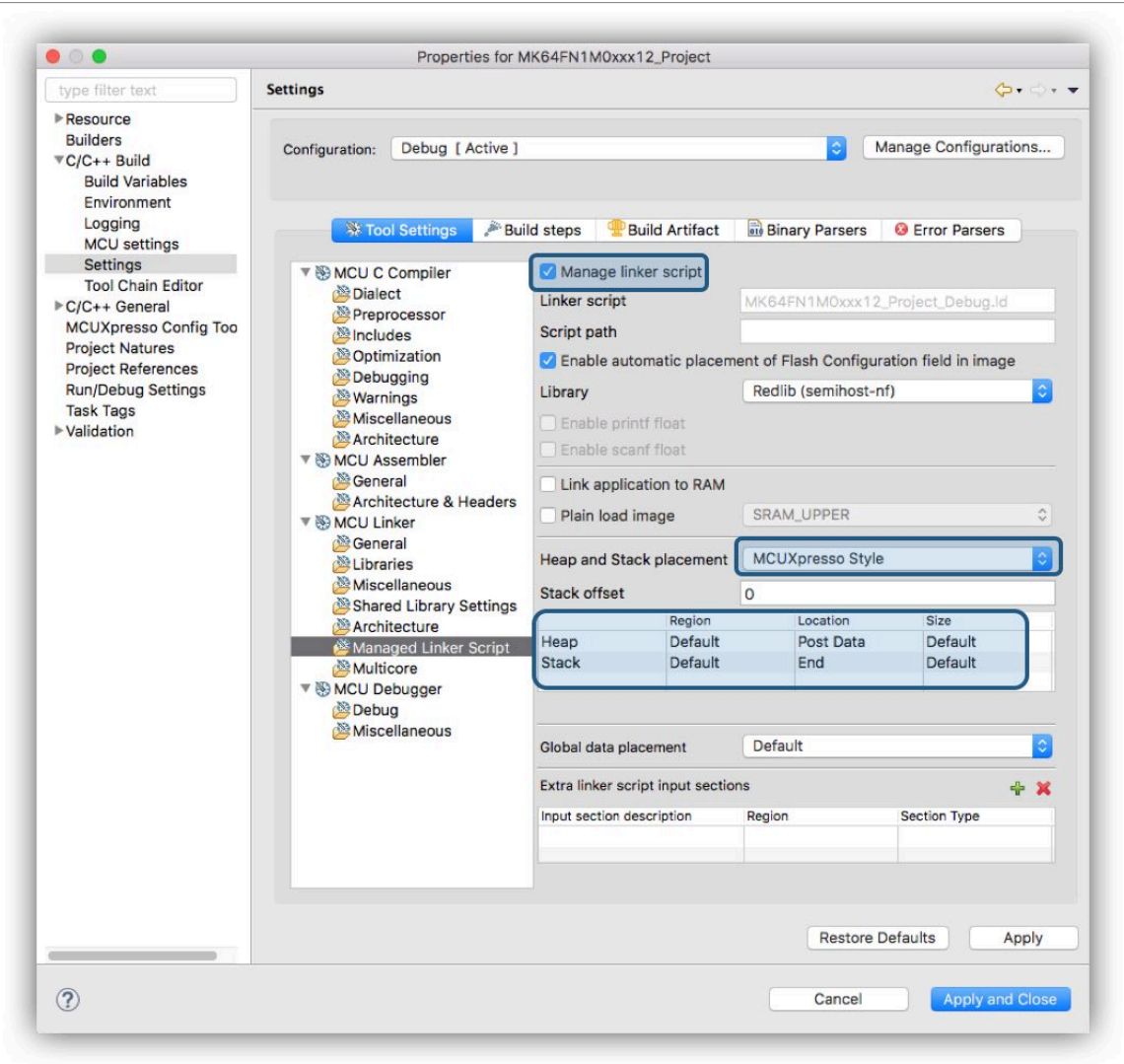
Figure 221. MCUXpresso IDE Linker Settings

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

20.12.1 MCUXpresso style heap and stack

By default, the heap and stack are placed in the "default" memory region (that is, 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 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 shown in Memory Configuration Editor

Location

- Start: Place at the start of the specified RAM bank.
- Post Data: Place after any data in the specified RAM bank. Default for heap.
- End: Place at the end of the specified RAM bank. Default for stack.

Size

- Default: 1/16th of the memory region size, up to a maximum of 4 kB (and a minimum of 128 bytes). Hovering the cursor over the field shows the current value that is used.
- Value: Specify the exact required size. Must be a multiple of 4. **Note**: 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 32 kB, enter 32768, 0x8000, or 32k.
  – A value of 0 can be entered to prevent any heap use by an application.
    – **Note**: For semihosted printf to operate without any heap space, you must enable the "character only" version. For Redlib, define the symbol "CR_PRINTF_CHAR" (at the project level) and remove other semihosting defines such as CR_INTEGER_PRINTF. Character-only semihosted printf is slower than the default version and may display differently depending on your debug solution.

Note: The MCUXpresso style of setting heap and stack has the advantage over the LPCXpresso style described below in that the memory allocated for heap/stack usage is also considered in the image size information displayed in the Build console when your project is built.

20.12.2 LPCXpresso style heap and stack

By default, the heap and stack are still placed in the "default" memory region (that is, 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, 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 32 kB 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

20.12.3 Reserving RAM for IAP Flash programming

The IAP Flash programming routines available in NXP's LPC MCUs generally use some of the on-chip RAM when executed. For example, on the LPC1343 the top 32 bytes of on-chip RAM are used. Therefore, if you are calling the IAP routines from your own application, you have to ensure that this memory is not used by your main application - which typically means by the stack.

However, with the managed linker script mechanism, it is easy to modify the start position of the stack (remember that stacks grow down) to avoid this clash with the IAP routines. To do this go to:

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

and modify the value in the "Stack Offset" field from 0 to 32. This works whether you are using the LPCXpresso style or MCUXpresso style of heap/stack placement.
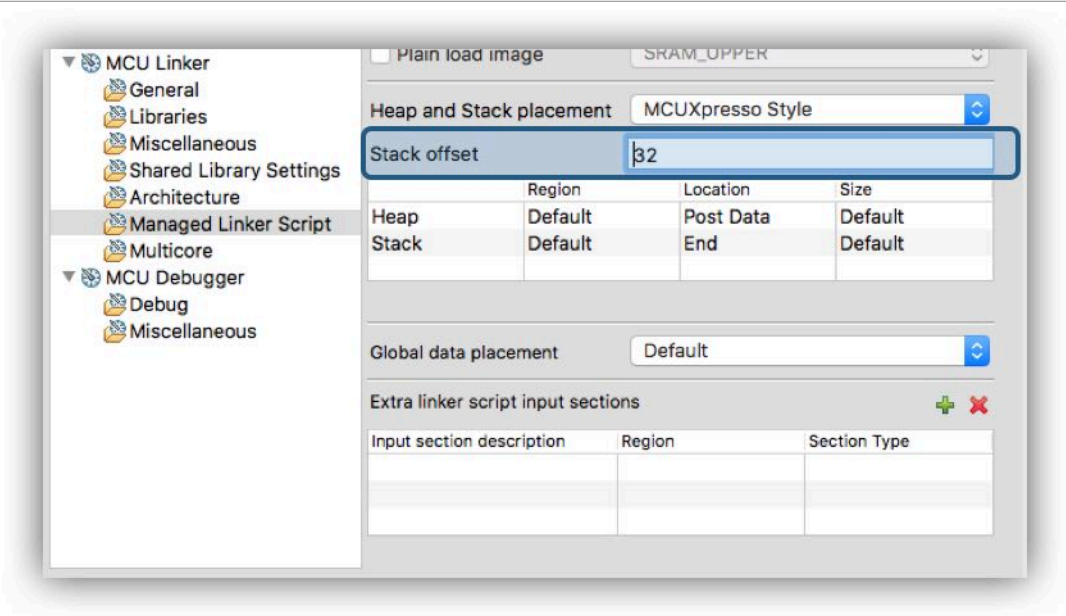
Figure 222. MCUXpresso IDE linker reserve stack space

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.

20.12.4 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. Therefore, if you want to detect automatically if the stack exceeds the memory set aside for it - other mechanisms must be used. For example:

- Identify a suitable memory region (or portion of one) that will fault for accesses below the region's base address, then locate the stack as desired within this region and watch for a possible 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

20.12.5 Heap checking

By default, the heap used by the malloc() family of routines grows upward from the end of the user data in RAM up toward the stack - a "one region memory model".

When a new block of memory is requested, the memory allocation function _sbrk() calls to the following function to check for heap overflow:

```
unsigned __check_heap_overflow (void * new_end_of_heap)
```

This should return:

- **1** - If the heap overflows
- **0** - If the heap is still OK

If 1 is returned, Redlib's malloc() sets errno to ENOMEM and return a null pointer to the caller.

The default version of **__check_heap_overflow()** built into MCUXpresso IDE-supplied C libraries carries 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. 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 assumes that the heap and stack are in the same region. This check is not enabled by the default implementation within the C library as it can break in some circumstances - for example when the heap is being managed by an RTOS.

20.12.6 Checking the heap from your application

The symbol **__end_of_heap** indicates the current end of the heap and can be used by user code to track heap usage. For instance:

```
extern unsigned int __end_of_heap;
 :
```

```
end_of_heap = __end_of_heap;
myBuffptr=(uint32_t*)malloc(20*sizeof(uint32_t));
new_end_of_heap = __end_of_heap;
```

However, it should be noted that the location this points to includes any last block that has been free'd. In other words, it effectively provides the maximal extent of the heap so far, not the end of the currently "active" last block.

Therefore, in some cases, if you check __end_of_heap before calling malloc(), then again afterward, it is possible that the value does not change if the heap request can be fulfilled using the free'd last block, that is, there is no need to extend the heap further. In certain cases, __end_of_heap can reduce, for example, if a block at the end of the heap is freed and a smaller block is subsequently allocated.

20.13 Placement of specific code/data items

It is possible to change the placement of specific code/data items within the final image without modifying the FreeMarker linker script templates. Such placement can be controlled via macros provided in an MCUXpresso IDE-supplied header file, which can be pulled into your project using:

```
#include <cr_section_macros.h>
```

Alternatively Introduced in MCUXpresso IDE version 10.2, the managed linker script mechanism now also provides a means of placing arbitrarily named code or data sections into a specified memory region of the generated image and is described in the next section. (See also Global data placement).

20.13.1 Placing code and data into different memory regions

Unlike the macros provided by cr_section_macros.h (described later), this method does not require any change to the source code declaring the affected code/data (which basically renames the generated code/data sections to match the memory region name). And often, it can avoid the need to provide project local FreeMarker linker script templates (described later in this chapter).

To place the code or data, you simply have to add the details of the section name, the memory region to place it in, and the type of the code/data, as per the below screenshot(s):
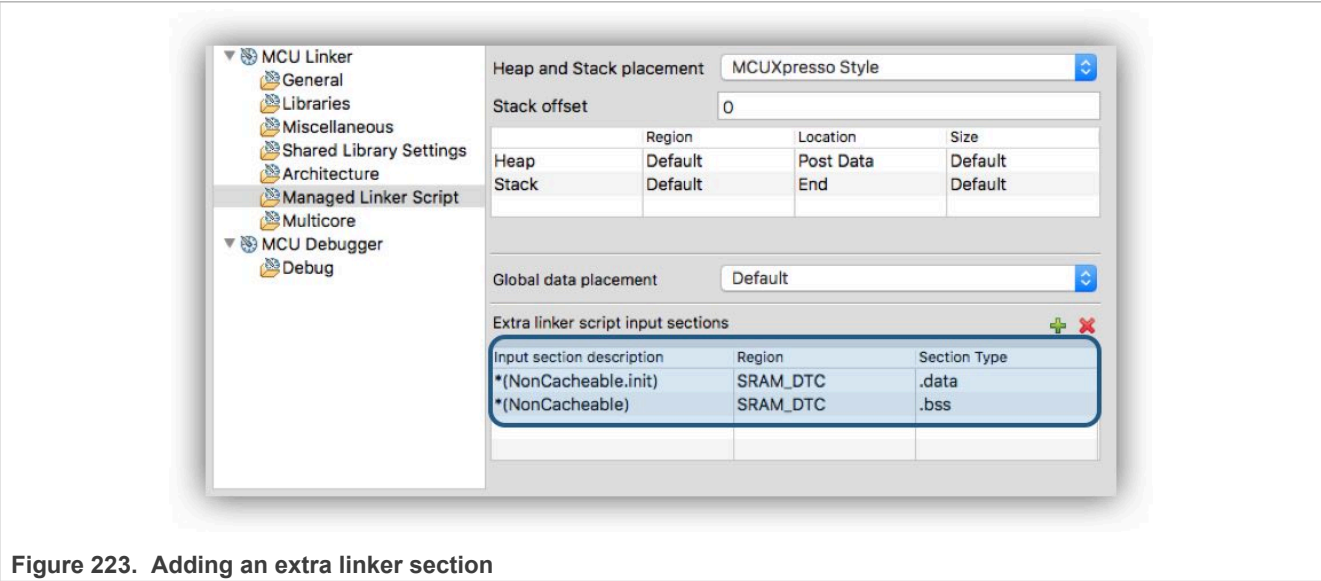
Figure 223. Adding an extra linker section

which modifies the generated linker script to contain the sections specified in the appropriate region:

```
/* Main DATA section (SRAM_DTC) */
.data : ALIGN(4)
{
    FILL(0xff)
    _data = . ;
    *(vtable)
    *(NonCacheable.init)
    . = ALIGN(4) ;
    _edata = . ;
} > SRAM_DTC AT>BOARD_FLASH
```

```
/* MAIN BSS SECTION */
.bss : ALIGN(4)
{
    *(NonCacheable)
    *(COMMON)
    . = ALIGN(4) ;
    _ebss = .;
    PROVIDE(end = .);
} > SRAM_DTC
```
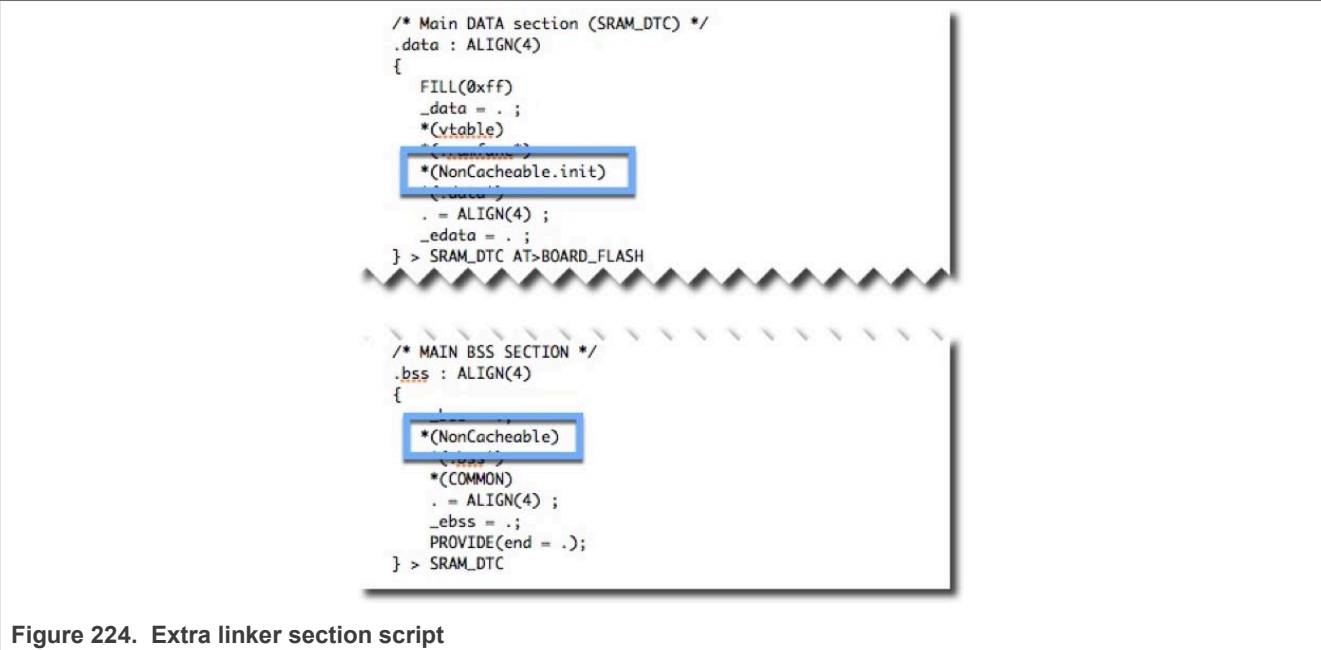
Figure 224. Extra linker section script

The second example graphic shows both the placement of a constant data table and also the powerful technique of specifying a project source folder and placing the entire contents of that folder (.text sections of flash2) into a chosen flash device. Using this scheme the user can drag and drop source files within the project structure to choose which location to use for their linkage and so their flash storage.

Extra linker script input sections		
Input section description	Region	Section Type
*(.big_const_data_table)	FLASH2	.rodata
flash2/(.text*)	FLASH2	.text
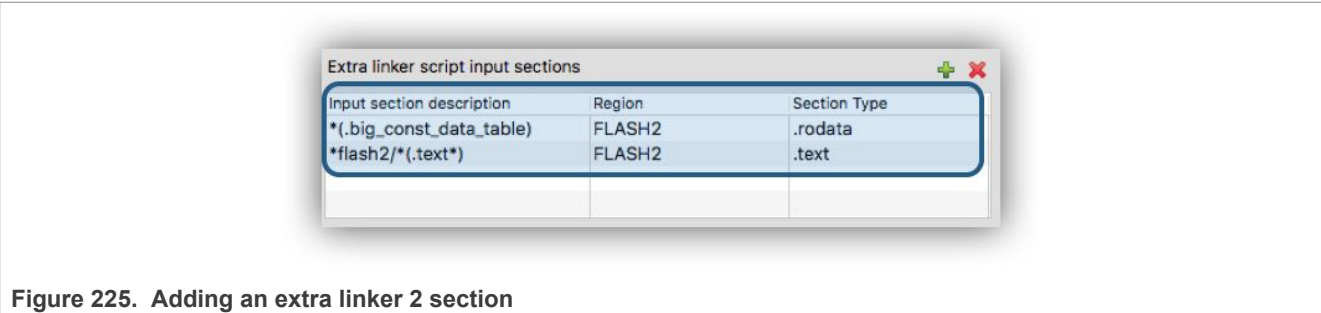
Figure 225. Adding an extra linker 2 section

Note: that the format of the "input section description" is as detailed in the GNU Linker documentation, which can be found within the built-in help system of the IDE:

Help -> Help Contents -> Tools (Compilers, Debugger, Utilities) -> GNU Linker -> Linker Scripts -> SECTIONS Command -> Input Section Description

Or directly in the online GNU documentation at:

https://sourceware.org/binutils/docs/ld/Input-Section-Basics.html

Also, this functionality only allows you to add sections to the linker script, not to remove something that the managed linker script already puts in. Therefore, if you have to remove part of the contents of the generated linker script - then you still have to modify the underlying FreeMarker linker script templates.

Finally, remember that the GNU linker script mechanism functions such that the first match encountered for a section wins (not the best match found). Therefore, this mechanism is just a request, not a guarantee. Always check the generated linker script and the map file output by the link step to confirm the expected placement of sections. In some problem cases, you may be able to force the required placement by use of an EXCLUDE in one memory region, and the section in the required region.

20.13.2 Placing data into different RAM blocks using macros

Many MCUs provide more than one bank of RAM. By default, the managed linker script mechanism places 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, and so on (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:

```
.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 like this:

```
// create an unitialised 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];
```

Sometimes, you might need a finer level of granularity than just placing a variable into a specific memory bank, and rather have 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.

20.13.3 Noinit memory sections

Normally global variables in an application 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 automatically copies the initial values of the ".data" sections from Flash to RAM, and zero-initialize ".bss" data sections directly in RAM.

The managed linker script mechanism of MCUXpresso IDE also supports the use of ".noinit" data within your application. Such data is similar to ".bss" except that it does 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 has to explicitly set up such data before using it - otherwise, the initial value of such a global variable is random (that is, it depends 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 on-chip LCD controller (for example NXP LPC178x and LPC408x parts).

 Document feedback

20.13.3.1 Making global variables Noinit

The linker script generated by the managed linker script mechanism of the IDE contains a section for each RAM memory block to contain ".noinit" items, and the ".data" and ".bss" items. **Note**: For a particular RAM memory block, all ".data" items are placed first, followed by ".bss" items, and then ".noinit" items.

However, normally for a particular RAM memory block where you are going to 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 like this:

```
// 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 like this:

```
// create a noinit integer variable in the main block of RAM
__NOINIT_DEF int noinit_var ;
```

20.13.4 Placing code/rodata into different FLASH blocks

Most MCUs only have one bank of Flash memory. But with some parts more than one bank may be available - and in such cases, by default, the managed linker script mechanism still places 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 can access external Flash (typically SPIFI) and their built-in internal Flash (for example, 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 varies depending on the 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 like this:

```
__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, you can use the __RODATA_EXT and __TEXT_EXT macros to place functions/rodata into a more specifically named section, for example:

```
__TEXT_EXT(Flash2,systick_delay) void systick_delay(uint32_t delayTicks) {
 :
}
```

is placed into the section ".text.$Flash2.systick_delay" rather than ".text.$Flash2".

20.13.5 Placing specific functions into RAM blocks

In most modern MCUs with built-in Flash memory, code is normally executed directly from Flash memory. Various techniques, such as prefetch buffering are used to ensure that code executes with minimal or zero wait states, even a higher clock frequency. See the documentation for the MCU that you are using for more details.

However, it is also possible to place specific functions into any of the defined banks of RAM for the target MCU, as displayed in:

Project -> Properties -> C/C++ Build -> MCU settings

and sometimes there can be advantages in relocating small, time-critical functions so that they run out of RAM instead of Flash.

For simplicity, the **additional** memory regions are named sequentially, starting from 2, (as well as using their "formal" region name - for example, RamAHB32). So, for a device with 3 RAM regions, alias names RAM, RAM2, and RAM3 will be available.

This placement is controlled via macros provided in a header file, which can be pulled into your project using:

```
#include <cr_section_macros.h>
```

The macro __RAMFUNC can be used to locate a function in 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) {...
```

To initialize RAM-based code (and data) into specified RAM banks, the managed linker script mechanism creates 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).

20.13.5.1 Long branch veneers and debugging

Due to the distance in the memory map between Flash memory and RAM, you 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 does not have any source-level debug information associated with it, so if you try to step in to a call to your code in RAM, typically the debugger steps 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.

20.13.6 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 vector table of the CPU 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, you can easily reclaim this unused space by choosing one or more functions to be placed into this unused memory. To do this, you simply have to decorate their definitions with the macro __AFTER_VECTORS, which is supplied in the "cr_section_macros.h" header file.

Obviously, to do this effectively, you have to identify functions, which 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, therefore 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 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 have to remove the __AFTER_VECTORS macro from the definition of one or more of your functions.

20.14 FreeMarker linker script templates

By default, MCUXpresso IDE projects use a managed linker script mechanism, which automatically generates a linker script file without user intervention - allowing the project code and data to be laid out in memory based on the IDE's knowledge of the memory layout of the target MCU.

However, sometimes the linker script generated in this way may not provide exactly the memory layout required. MCUXpresso IDE therefore provides a highly flexible and powerful linker script template mechanism to allow the user to change the content of the linker script generated by the managed linker script mechanism.

20.14.1 Basics

FreeMarker is a template engine: a generic tool to generate text output (HTML web pages, e-mails, configuration files, source code, and so on) based on templates and changing data. Built into MCUXpresso IDE is a set of templates that are processed by the FreeMarker template engine to create the linker script. Templates are written in the FreeMarker Template Language (FTL), which is a simple, specialized language,

not a full-blown programming language like PHP. Full documentation for FreeMarker can be found at https://freemarker.org/docs/index.html.

MCUXpresso IDE automatically invokes FreeMarker, passing it a data model that describes the memory layout of the target together with a 'root' template that is processed to create the linker script. This root template, #includes further 'component' templates. This structure allows a linker script to be broken down into various components and allows a user to provide their own templates for a component, instead of having to (re-)write the whole template. For example, component templates are provided for text, data, and bss sections, allowing the user to provide different implementations as necessary, but leaving the other parts of the linker script untouched.
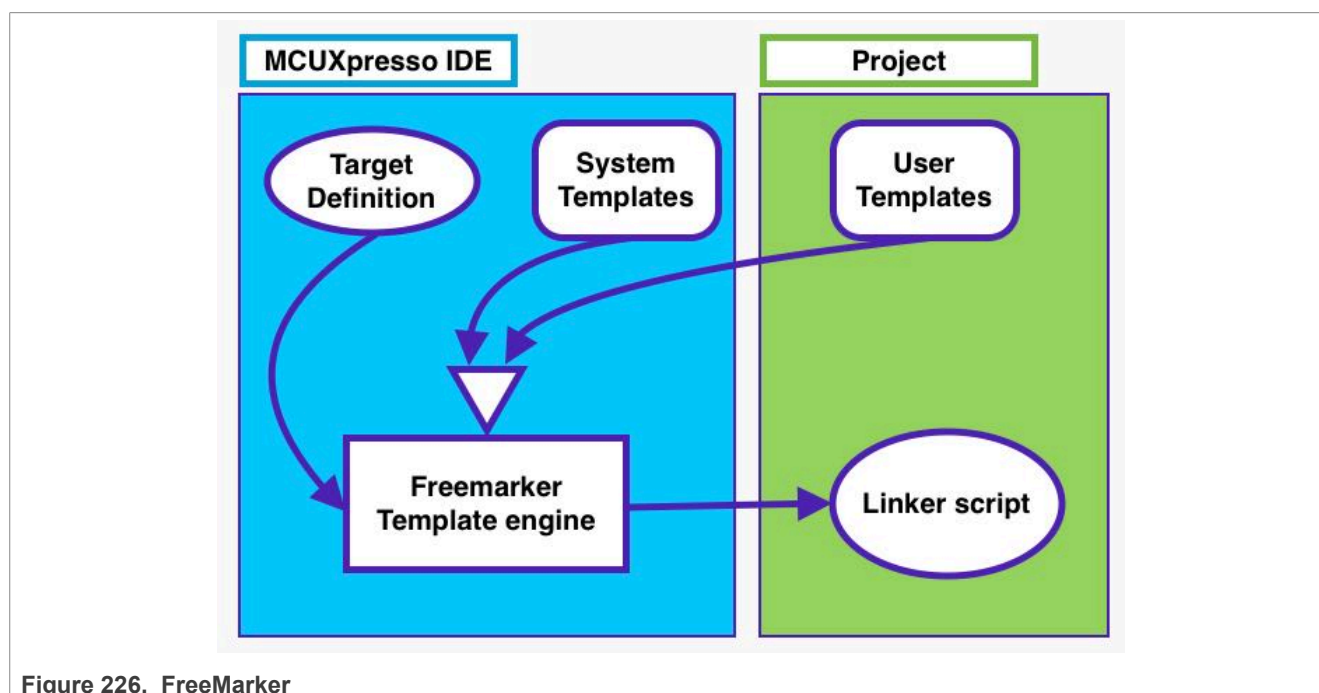
Figure 226. FreeMarker

20.14.2 Reference

FreeMarker reads input files, copies text, and processes FreeMarker directives and 'variables', and writes an output file. As used by the managed linker script mechanism of MCUXpresso IDE, the input files describe the various components of a linker script which, together with variables defined by the IDE, are used to generate a complete linker script. Any of the component template input files may be overridden by providing a local version in the project.

The component template input files are provided as a hierarchy, shown below, where each file #includes those files nested below. This allows for individual components of the linker script to be overridden without having to supply the entire linker script, increasing flexibility, while maintaining the benefits of Managed Linker Scripts.

20.14.2.1 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)
- sgstubs_fixed.ldt (allow absolute veneer table for TrustZone application)
- text_section.ldt (text sections for each secondary Flash)
  - text_section_multicore.ldt (text sections for multicore targets)
    - text_section_multicore_checks_partfamily.ldt (part-specific sanity checks)
  - extrasections_text.ldt (additional linker sections)
  - text.ldt (for inserting *text)
  - extrasections_rodata.ldt (additional linker sections)
  - rodata.ldt (for inserting rodata)
- boot_hdr.ldt (allows placement of optional header before main code section)
  - boot_hdr_partfamily.ldt
- main_text_section.ldt (the primary text section)
  - global_section_table.ldt (the global section table)
  - crp.ldt (the CRP information)
  - flashconfig.ldt (flash security prototype)
  - extrasections_text.ldt (additional linker sections)
  - main_text.ldt (for inserting *text)
  - extrasections_rodata.ldt (additional linker sections)
  - freertos_debugconfig.ldt (to force placement of FreeRTOSDebugConfig rodata section)
  - main_rodata.ldt (read-only data)
  - cpp_info.ldt (additional C++ requirements)
- exdata.ldt (placement of LPCXpresso style exdata sections)
- sgstubs.ldt (allow absolute veneer table for TrustZone application)
- 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_section_multicore_checks_partfamily.ldt (part-specific sanity checks)
  - extrasections_data.ldt (additional linker sections)
  - data.ldt (for inserting *data)
- mtb_default_section.ldt (special section for MTB (cortex-m0+ targets)
- uninit_reserved_section.ldt (uninitialized data)
- main_data_section.ldt (primary data section)
  - extrasections_data.ldt (additional linker sections)
  - main_data.ldt (for inserting *data)
- ecrp.ldt (Enhanced Code Read Protection support)
- bss_section.ldt (secondary bss sections)
  - extrasections_bss.ldt (additional linker sections)
  - bss.ldt (for inserting *bss)
- main_bss_section.ldt primary bss section)
  - extrasections_bss.ldt (additional linker sections)

  – main_bss.ldt (for inserting *bss)
- noinit_section.ldt (no-init data)
  – extrasections_noinit.ldt ([additional linker sections](#))
- noinit_noload_section.ldt (no-load data)
- exdata_sdk.ldt (placement of MCUXpresso style exdata sections)
  – data_section_exceptions_multicore_sdk.ldt (additional multicore exdata sections information)
  – text_section_exceptions_multicore_sdk.ldt (additional multicore exdata sections information)
- stack_heap_sdk_postdata.ldt (placement of MCUXpresso style heap/stack)
- stack_heap_sdk_end.ldt (placement of MCUXpresso style heap/stack)
- stack_heap.ldt (define the stack and heap)
- checksum.ldt (create the LPC checksum)
- image_size.ldt (provide basic symbols giving location and size of image)
- symbols.ldt (provide additional symbols needed to built image)
  – symbols_partfamily.ldt (part specific "symbol")
- 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)

20.14.2.2 Linker script search paths

Whenever a linker script template is used, MCUXpresso IDE searches 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)

Therefore, 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.

For example, *<#include "super@user.ldt">*

20.14.2.3 Linker script templates

Copies of the default linker script templates used within MCUXpresso IDE can be accessed through the */LinkServer* symbolic link found inside the IDE installation, more specifically */LinkServer/Wizards/linker*. The templates are part of the external LinkServer package. These can be used as the basis of any project local scripts you wish to write.

20.14.2.4 Predefined variables (macros)

List (sequence) variables (used in _#list_)

libraries[]

- list of the libraries to be included in the "lib" script
  - **–** for example (Redlib nohost)

```
libraries[0]=libcr_c.a
libraries[1]=libcr_eabihelpers.a
```

configMemory[] list of each memory region defined in the memory map for the project. Each entry has the following fields defined:

- name - the name of the memory region
- alias - the alias of the memory region
- location - the base address of the memory
- size - the size of the memory region
- sizek - the printable size of the memory region in k or M
- mcuPattern
- defaultRAM - boolean indicating if this is the default RAM region
- defaultFlash - boolean indication if this is the default Flash region
- RAM - boolean indicating if this is RAM
- Flash - boolean indicating if this is Flash

For example:

```
configMemory[0]= name=MFlashA512 alias=Flash location=0x1a000000
size=0x80000 sizek=512K bytes mcuPattern=Flash flash=true RAM=false
defaultFlash=true defaultRAM=false
```

```
configMemory[2]= name=RamLoc32 alias=RAM location=0x10000000
size=0x8000 sizek=32K bytes mcuPattern=RAM flash=false RAM=true
defaultFlash=false defaultRAM=true
```

Slaves[] list of the secondaries in a Multicore project. This variable is only defined in multicore projects. Each entry has the following fields defined:

- name - name of the secondary reference
- enabled - boolean indicating if this secondary reference is enabled
- objPath - path to the object file for the Secondary image
- linkSection - name of the section this secondary entity is to be linked in
- runtimeSection
- textSection - name of the text section
- textSectionNormalized - normalized name of the text section
- dataStartSymbol - name of the Symbol defining the start of the data
- dataEndSymbol - name of the Symbol defining the end of the data

for example:

```
slaves[0] = name=M0APP objectPath=${workspace_loc:/MCB4357_Blinky_DualM0/Debug
/MCB4357_Blinky_DualM0.axf.o}linkSection=Flash2 runtimeSection= textSection=
.core_m0app textSectionNormalized=_core_m0appdata StartSymbol=__start_data
```

```
dataEndSymbol=__end_data enabled=true;</notextile>
```

Simple variables include:

- 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 secondary - true if is a secondary 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 secondary (only present for secondaries)
- 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)

20.14.2.5 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: 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**: System properties are case-sensitive.

20.14.2.6 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, therefore the slightly longer-winded method of specifying the option).

20.15 FreeMarker linker script template examples

The use of FreeMarker linker script templates allows more wide-ranging changes to be made to the generated link script than is possible using the *cr_section_macros.h* macros. The following examples provide some examples of this.

20.15.1 Relocating code from FLASH to RAM

If you have specific functions in your code base that you wish to place into a particular block of RAM, then the simplest way to do this is to decorate the function definition using the macro __RAMFUNC described earlier in this chapter.

However, once you want to relocate more than a few functions, or when you don't have direct access to the source code, this becomes impractical. In such cases, the use of FreeMarker linker script templates is a better approach. The following sections provide a number of such examples.

20.15.1.1 Relocating particular objects into RAM

Sometimes, it may be required to relocate all of the functions (and rodata) from a given object file in your project into RAM. This can be achieved by providing three linker script template files into a *linkscripts* folder within your project. For example, if it was required that all code/rodata from the files foo.c and bar.c were relocated into RAM, then this could be achieved using the following linker script templates:

main_text.ldt

```
*(EXCLUDE_FILE(*foo.o *bar.o) .text*)
```

main_rodata.ldt

```
*(EXCLUDE_FILE(*foo.o *bar.o) .rodata)
*(EXCLUDE_FILE(*foo.o *bar.o) .rodata.*)
*(EXCLUDE_FILE(*foo.o *bar.o) .constdata)
*(EXCLUDE_FILE(*foo.o *bar.o) .constdata.*)
. = ALIGN(${text_align});
```

main_data.ldt

```
*foo.o(.text*)
*foo.o(.rodata .rodata.* .constdata .constdata.*)
*bar.o(.text*)
*bar.o(.rodata .rodata.* .constdata .constdata.*)
```

```
  . = ALIGN(${text_align});
 *(.data*)
```

What each of these EXCLUDE_FILE lines (in main_text.ldt and main_rodata.ldt) is doing in pulling in all of the sections of a particular type (for example .text), except for the ones from the named object files. Then in main_data.ldt, we specify explicitly that the text and rodata sections should be pulled in from the named object files. **Note**: that with the GNU linker, LD, the first match found in the final generated linker script is always used, which is why the EXCLUDE_FILE keyword is used in the first two template files.

Note: EXCLUDE_FILE only acts on the closest input section specified, which is why we have four 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.

20.15.1.2 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*)
```

Note: When extending this example to more than one library, mind the semantics of the EXCLUDE_FILE directive which is pulling in all of the sections of a particular type, except for the ones from the named object files. Therefore, all of the library objects need to be specified in EXCLUDE_FILE for a particular section type (in main_text.ldt and main_rodata.ldt templates). For example:

```
 *(EXCLUDE_FILE(*libMYLIBRARYPROJ1.a *libMYLIBRARYPROJ2.a
  *libMYLIBRARYPROJ3.a:) .text*)
```

On the other hand, the library objects in main_data.ldt template, since they are not using EXCLUDE_FILE, need to be listed on separate lines:

```
*libMYLIBRARYPROJ1.a:(.text*)
*libMYLIBRARYPROJ2.a:(.text*)
*libMYLIBRARYPROJ3.a:(.text*)
```

20.15.1.3 Relocating the majority of an application into RAM

In some situations, you may wish to run the bulk of your application code from RAM - typically just leaving the 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 causes the main body of the code to be relocated into the main (first) RAM bank of the target MCU, which by default also contains data/bss, as well as the stack and heap.

Important Note: The code that performs this relocation is executed early within the reset handler (within *startup_xx* file). However, there is the potential for other critical functions to be called **before** this relocation is performed, for example, *SystemInit()* may be called first to perform essential operations such as enabling RAM!

Any function that is called before the relocation is performed **must not** itself be relocated! For the specific case above, the following changes to main_text.ldt and main_rodata.ldt are required:

main_text.ldt

```
*startup_*.o (.text.*)
*system_*.o (.text.*)
*(.text.main)
*(.text.__main)
```

main_rodata.ldt

```
*startup_*.o (.rodata .rodata.* .constdata .constdata.*)
*system_*.o (.rodata .rodata.* .constdata .constdata.*)
. = ALIGN(${text_align});
```

Finally, 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.

20.15.2 Configuring projects to span multiple Flash devices

Most MCUs only have one bank of Flash memory. But with some parts more than one bank may be available - and in such cases, by default, the managed linker script mechanism still places 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 can access external Flash (typically SPIFI) alongside their built-in internal Flash (for example, LPC18xx, LPC40xx, LPC43xx, LPC546xx).

The macros provided in the "cr_section_macros.h" header file provide some ability to control the placement of specific functions or rodata items into the second (or even third) bank of Flash. However, the use of FreeMarker linkers script templates allows 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 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).

20.16 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 FreeMarker template mechanism should allow you to avoid the need for writing your own linker scripts. But if you do wish to do this, then untick the appropriate option at:

Properties -> C/C++ Build -> Settings -> MCU Linker -> Managed Linker Script

And then in the field *Script Path* provide the name and path (relative to the current build directory) of your own, manually maintained linker script.

In such cases you can either create your own linker script from scratch, or you can use the managed linker scripts as a starting point. One 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 prevents 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 also have to include the relative path information in the include inside the top-level script file.

For more details on writing your own linker scripts, see the GNU Linker (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:

https://www.embedded.com/design/mcus-processors-and-socs/4026080/Building-Bare-Metal-ARM-Systems-with-GNU-Part-3

See also the section on Enhanced syntax highlighting to review editor assistance when manually creating Linker Scripts.

21 Multicore projects

Additional information can be found on the MCUXpresso IDE Community pages Specifically, see the blog articles:

LPC55xx Multicore Applications with MCUXpresso IDE and also the article: *Using LPC55S69 SDK Trustzone examples with MCUXpresso IDE v11.0.0*

21.1 Introduction

Multicore MCUs can be designed in many ways, however, within MCUXpresso IDE there is an underlying expectation that one core (the primary) controls the execution (or at least the startup) of code running on other (secondary) core(s).

Multicore application projects as described below consist of two (or more) linked projects - one project containing the code of the primary core and the other project(s) containing the code of the secondary core. The 'Primary' project contains a link to the 'Secondary' project, which causes the output image from the 'Secondary'

to be included into the 'Primary' image when the primary project is built. Building the primary project triggers the secondary project to be built first.

After a power-on or Reset, the primary core boots and is then responsible for booting the secondary core. However, this relationship only applies to the booting process; after boot, an application may treat either of the cores as primary or secondary.

For this concept to work, the memory configurations of these related projects must be carefully managed to avoid unintended overlap or contention. One way to achieve this is by linking the secondary application to execute entirely from a RAM location unused by the primary. Our automatic linkerscript generation then locates the code of the secondary within the generated image of the primary, this code is relocated to the correct RAM location by the initialization code of the primary project at runtime.

In practice, the memory configuration of the primary project is the same as for a single core project, whereas the memory configuration of the secondary project is set to use a 'spare' or dedicated secondary RAM region. In addition, a shared region may be used for communication between the CPUs.

Note: MCUs supporting dedicated Flash regions for each core can also be supported by this scheme, in such cases the secondary project would simply be linked to the secondary core's Flash location.

To complete the story ... the primary project is debugged first, which leads to the combined image being programmed into Flash and the primary code executed. The initialization code of the primary core copies (in addition to other things) the code of the secondary core into RAM (if appropriate) and then stops on Main. When the secondary project is debugged, the launch configuration is automatically set to 'Attach' by the IDE since there is no need for this code to be programmed/downloaded by the debugger. When the primary application is resumed, it releases the secondary core and both projects can be debugged as required.

Important Note: Multicore MCUs may offer significant flexibility in how they can be used. The mechanism described above (as used in example projects) is not necessarily the only way (or even the best way) for a user's multicore projects to be configured. However, it has been chosen as the simplest and safest way to demonstrate the concepts and issues involved.

MCUXpresso IDE allows for the easy creation of "linked" projects that support the targeting of multicore MCUs.

The rest of this chapter describes the use of the LPC5411x multicore MCU, however, the concepts discussed are the same (or similar) for other multicore MCUs such as the LPC43xx and LPC5410x.

21.2 Creating a primary/secondary project pair (using an SDK)

The example described below is based on the LPC5411x multicore MCU using the LPCXpresso54114 SDK.

Note: Be sure to have installed the LPCXpresso54114 SDK into MCUXpresso IDE if you wish to follow this example.

21.2.1 Creating the M0 secondary project

As discussed above, the configuration of the primary project has to reference the secondary project, therefore the secondary project should be created first.

Launch the New Project Wizard and select the LPCXpresso54114 SDK board. Entering 54114 into the boards filter reduces the number of boards to help selection, then click Next.
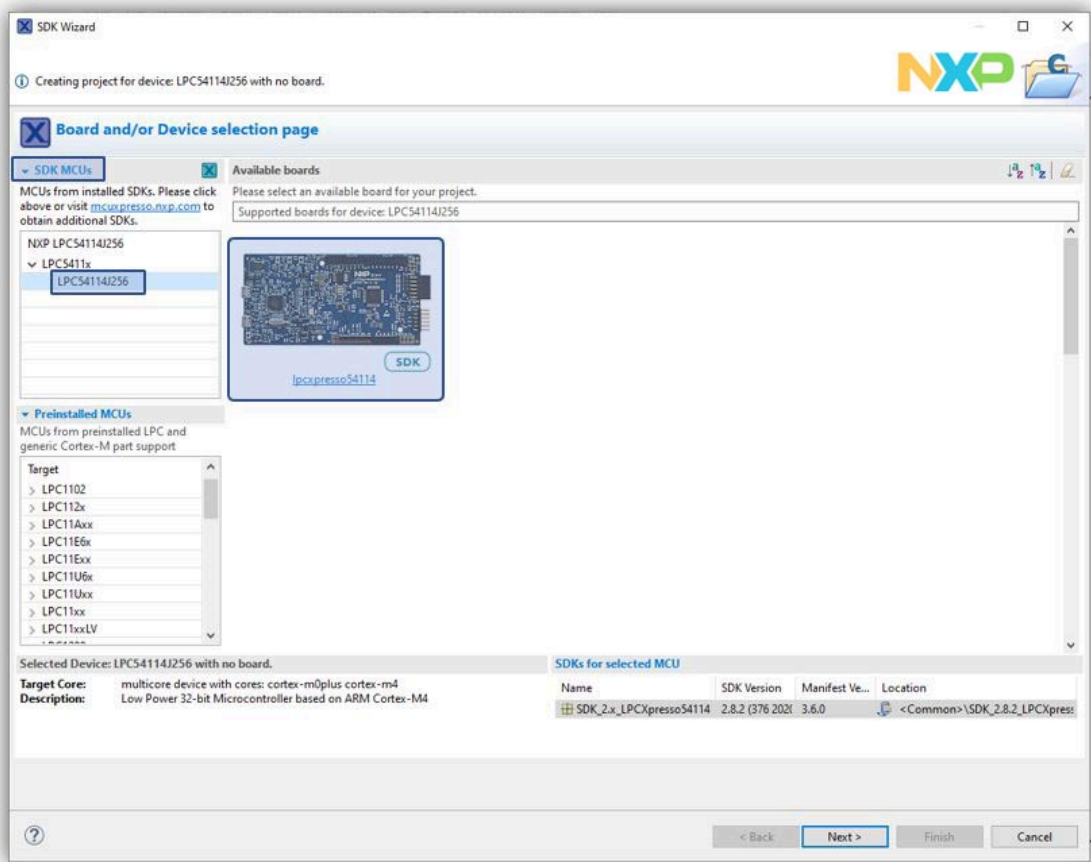
Figure 227. New Project Wizard SDK multicore M0

From the next wizard page, select the cm0plus Core, and see that the M0SLAVE is selected in the core options. Also the project is automatically given the suffix M0SLAVE. Drivers, utilities, and so on can be selected at this stage for the secondary project if required.
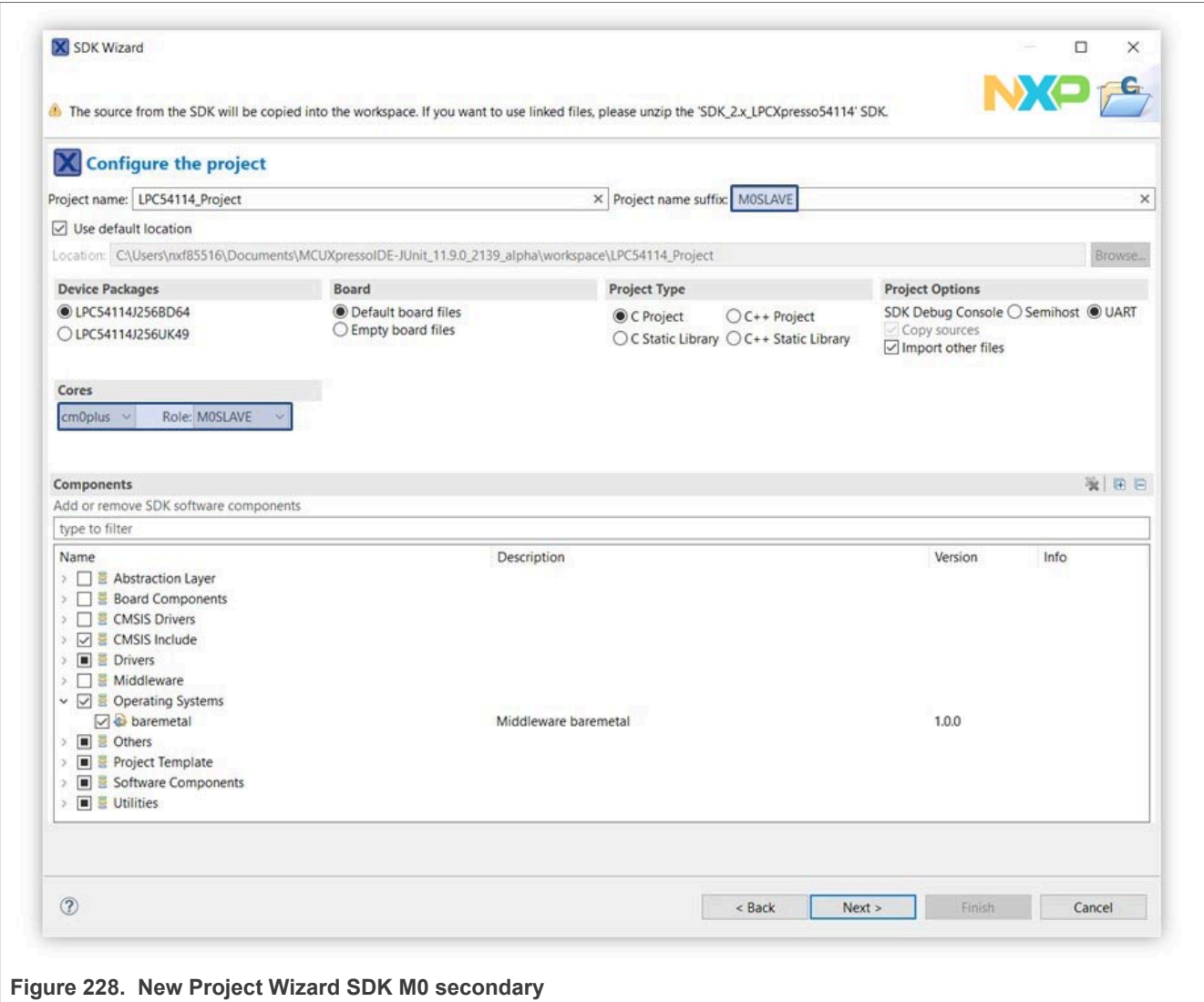
Figure 228. New Project Wizard SDK M0 secondary

Next, the M0 secondary memory configuration has to be set.

Note: The managed linker script mechanism of MCUXpresso IDE defaults to link code to the first Flash region in this view (if one exists) and use the first RAM region for data, heap, and stack.

To force our project to link to a private area of RAM, we must ensure that the Flash region is removed and the chosen RAM bank is at the top of the list of memory regions. Note here that the SDK we are using has presented the RAM regions in a non-sequential order. In this example, we configure the memory so that the M0 secondary project links to the RAM region starting at address 0x20010000 (the first region).
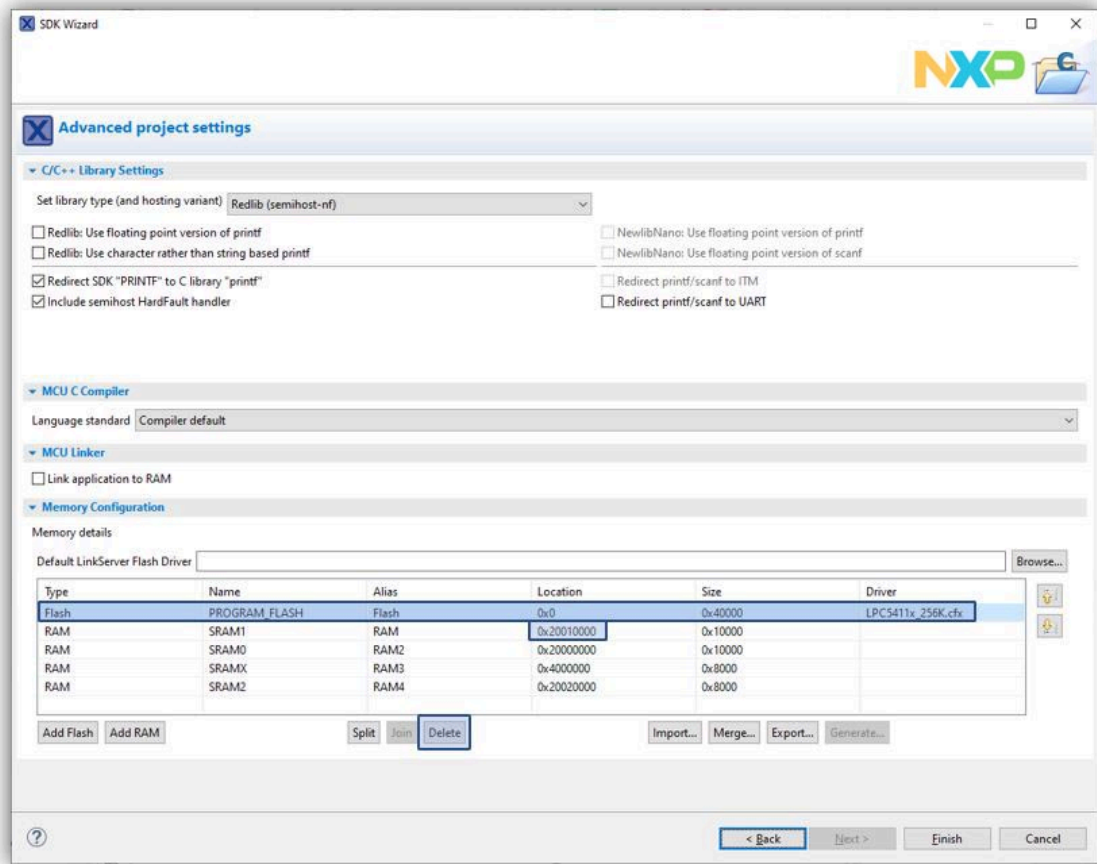
Figure 229. New Project Wizard SDK M0 secondary memory

From this wizard, select the PROGRAM_FLASH and click Delete to remove the region. Ensure that the top RAM region has the base address (location) 0x20010000, then click Finish to complete the creation of the secondary project.

Tip: Memory regions can be reordered by selecting a region and using the up/down arrows to move the selected region.

21.2.2 Creating the M4 primary project

To create the primary project, launch the New Project Wizard and again select the LPCXpresso54114 SDK board, and click Next. This time, select the cm4 Core, and **click the MASTER checkbox**, this configures the wizard to create a multicore project. The Project is automatically given the suffix MASTER.

Drivers, utilities, and so on, can be selected at this stage for the primary project if required.
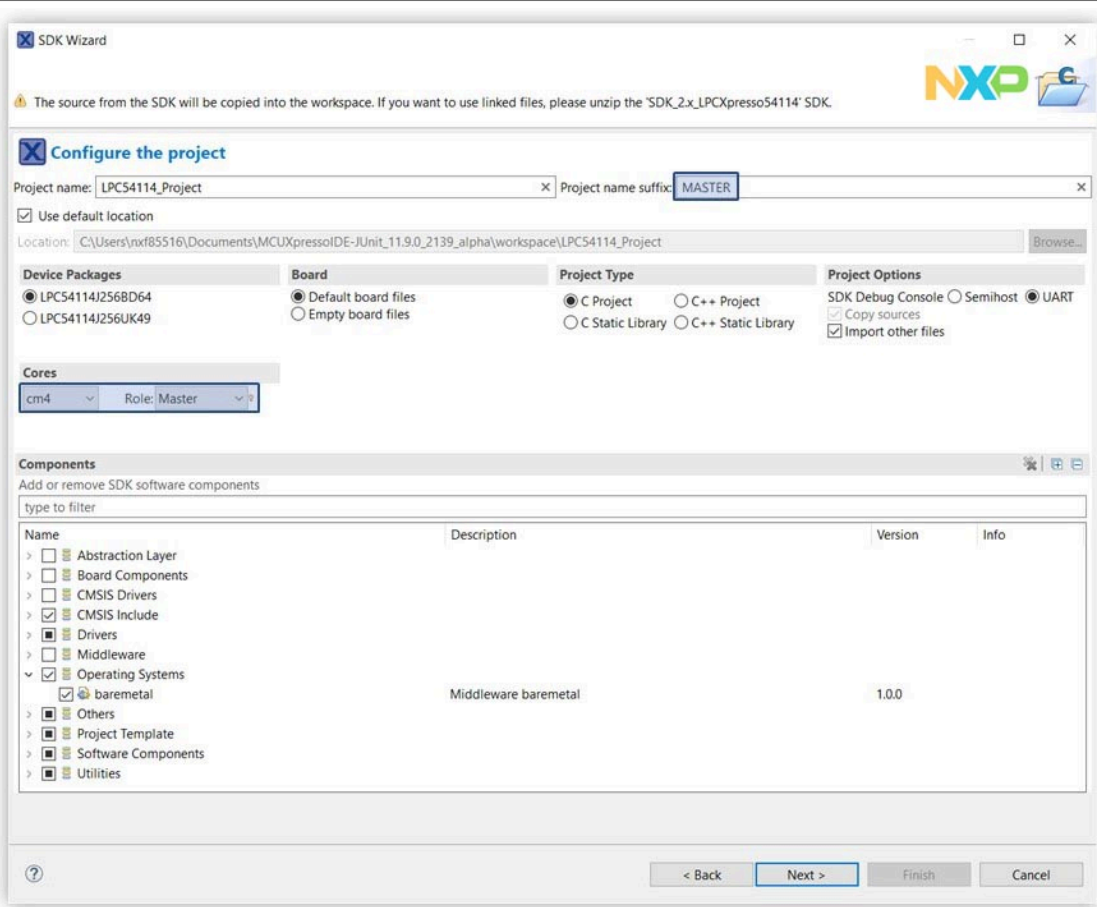
Figure 230. New Project Wizard SDK M4 primary

Next, the memory configuration of the M4 primary project has to be set. Typically we might leave the memory setting unaltered, however, the SDK we are using presents the RAM regions in a non-sequential way. In this example, we wish to select the RAM region at 0x20000000 for the primary projects data and the Flash at 0x0 for the primary projects code (and also a copy of the secondary projects code)

Note: MCUXpresso IDEs managed linker script mechanism defaults to link code to the first Flash region in this view (if one exists) and use the first RAM region for data, heap, and stack.
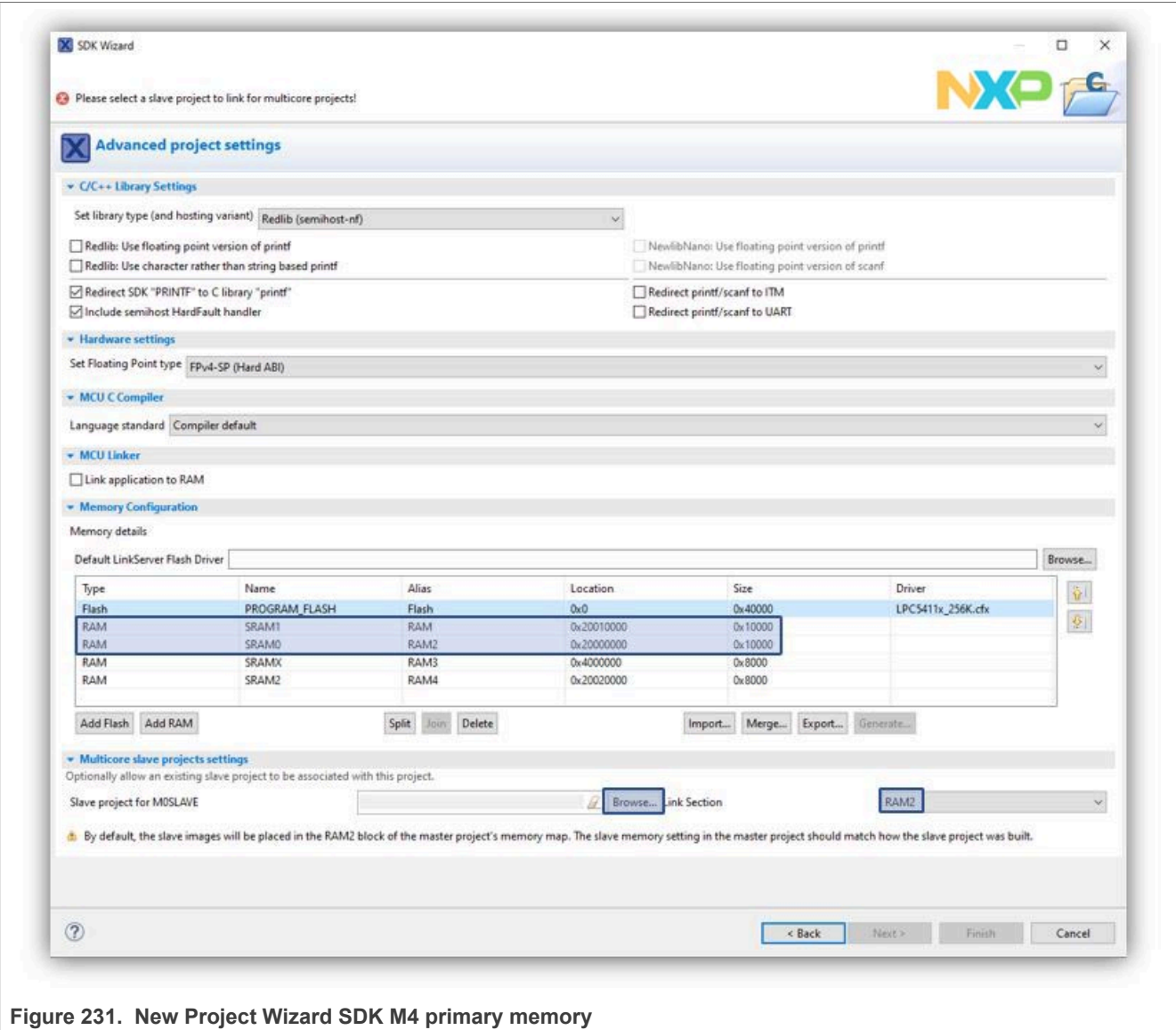
Figure 231. New Project Wizard SDK M4 primary memory

To adjust the memory layout, select the second RAM region (at location 0x20000000) and click the 'Up' arrow to move this to the top of the RAM regions. The highlighted regions as shown above will be effectively swapped.

Once this has been done, click 'OK'.

Next, click Browse to locate a secondary project within the workspace and select the previously created secondary project, then click 'OK'.
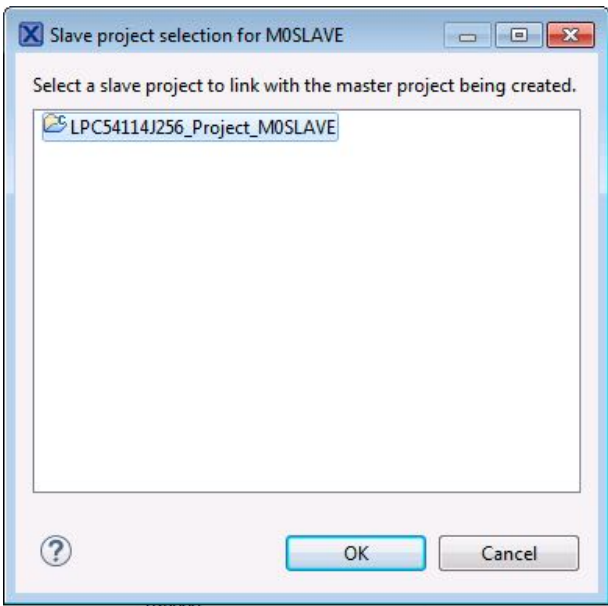
Figure 232. New Project Wizard SDK M4 primary/secondary selection

Note: ensure the Link Section name (default of RAM2 highlighted) selects a primary memory region that matches the linked address of the secondary project. In this case, RAM2 should correspond to the address 0x2001000. If required, other memory regions can be selected here but note: the first Flash Region and the first RAM Region are not included in the dropdown list because it is assumed that these will be used for the primary Project. If required, this setting can be changed later from:

Project Properties -> C/C++ Build -> Settings -> Multicore

Where all of the memory regions are available for selection.

Below we can see the edited project settings for the primary project.
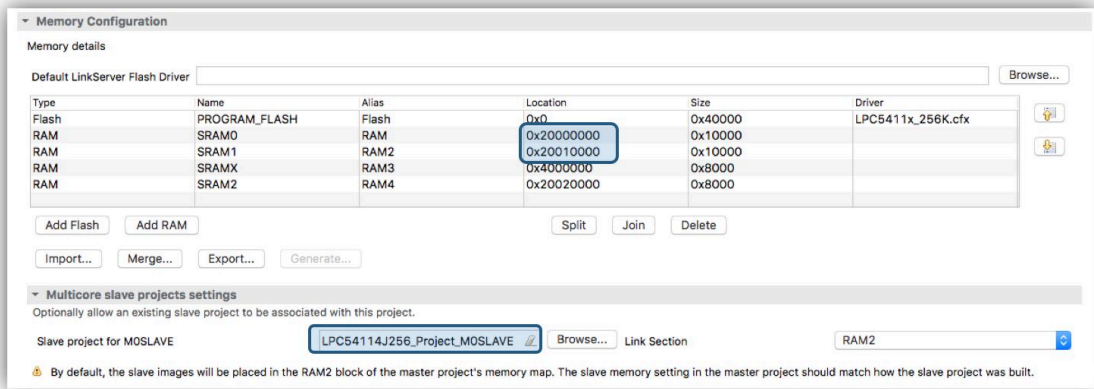
Figure 233. New Project Wizard SDK M4 primary project

Finally, click Finish to generate the primary project.

Note: if the memory regions of these projects overlap, the linker generates an error similar to:

```
M0SLAVE execute address differs from address provided in source image
```

To fix this issue, review (and edit) the memory settings of the related projects so that their addresses do not overlap via *Project Properties -> C/C++ Build -> MCU settings*.

21.3 Creating a primary/secondary project pair (using preinstalled part support)

The example described below is based on the LPC5411x multicore MCU.

Note: It is recommended to create and build LPC541xx multicore projects, which are linked to LPCOpen. Therefore, before you follow the below sequence, ensure that you have imported the chip and (optionally) the board library projects (for both the M4 and M0+) from an LPCOpen package for the LPC5410x family or LPC5411x family (depending upon your target part).

21.3.1 Creating the M0 secondary project

As discussed above, the configuration of the primary project has to reference the secondary project, therefore the secondary project should be created first.

Launch the New Project Wizard and select the LPC54114-M0 from the Preinstalled MCUs.
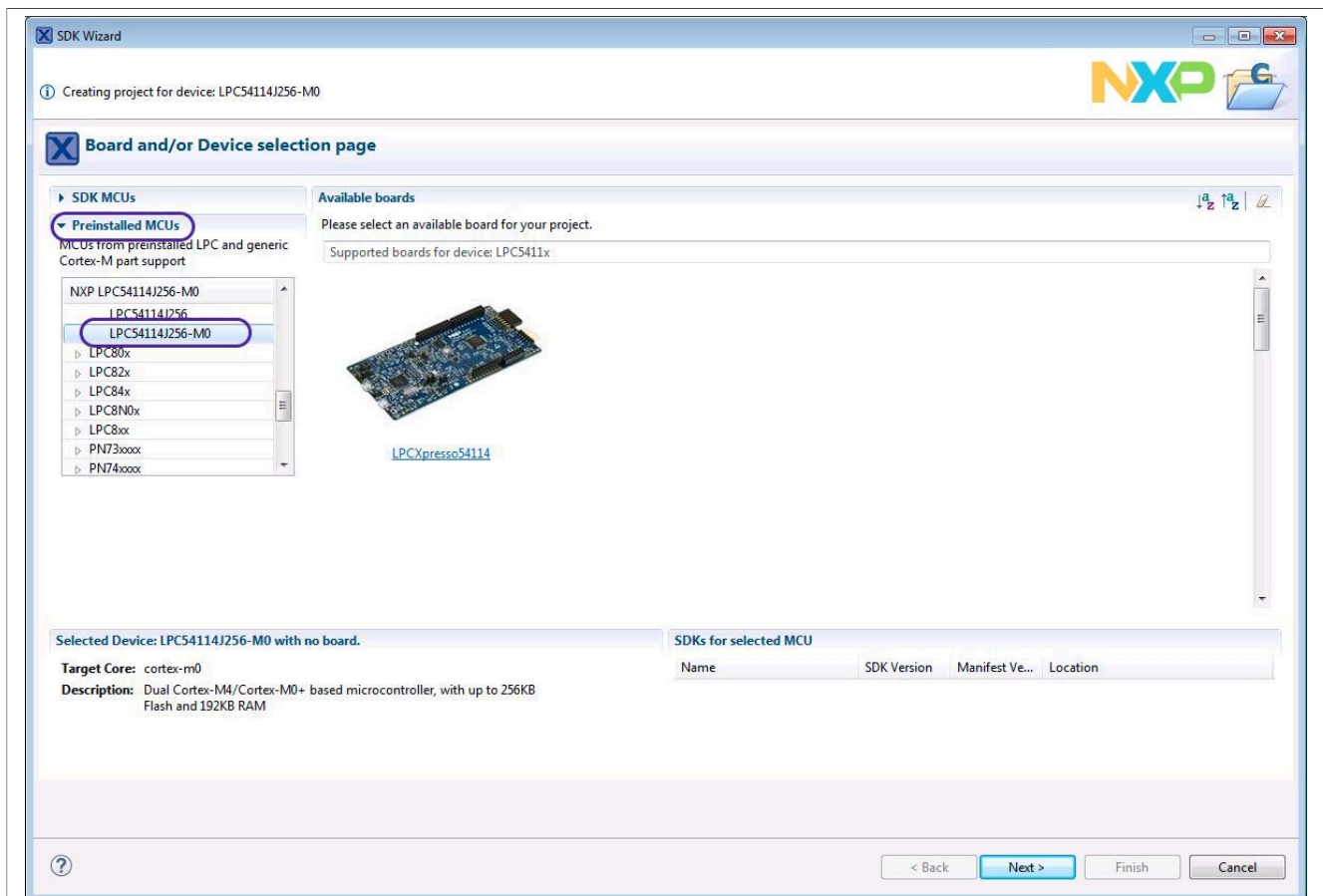
Figure 234. New Project Wizard preinstalled M0

Next, select a multicore M0 secondary project type. Below we have selected an LPCOpen - C Project.
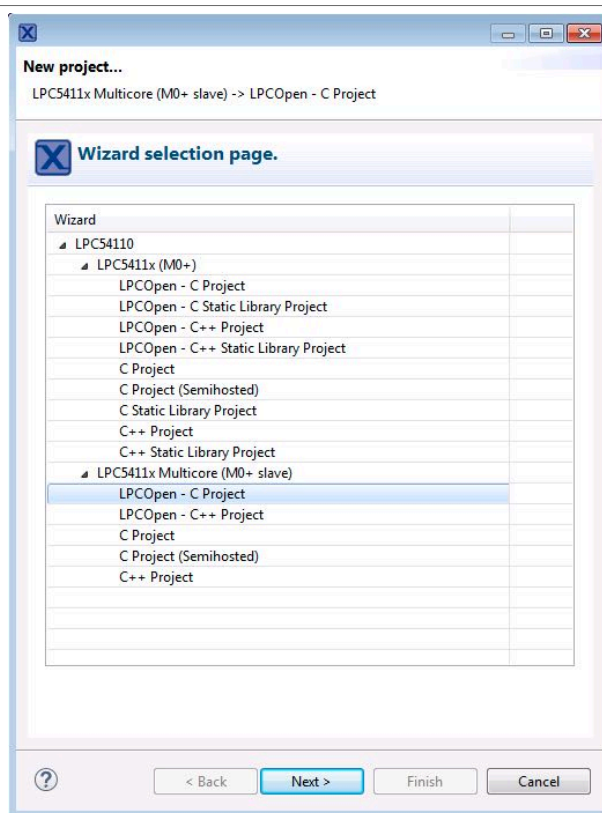
Figure 235. New Project Wizard preinstalled M0 C project

Next, name the project, for example LPC54114_M0_Slave, then click next until the Memory Configuration page is reached. From here, we can see the MCU memory regions.

Note: The managed linker script mechanism of MCUXpresso IDE defaults to link code to the first Flash region in this view (if one exists) and use the first RAM region for data.

To force our project to link to a private area of RAM, simply delete the Flash and first RAM region (RAM0_64) from this view (since these are used for the M4 primary project). To do this, select the regions and click Delete. Since there is no longer any Flash region, the default Flash driver can also be removed.
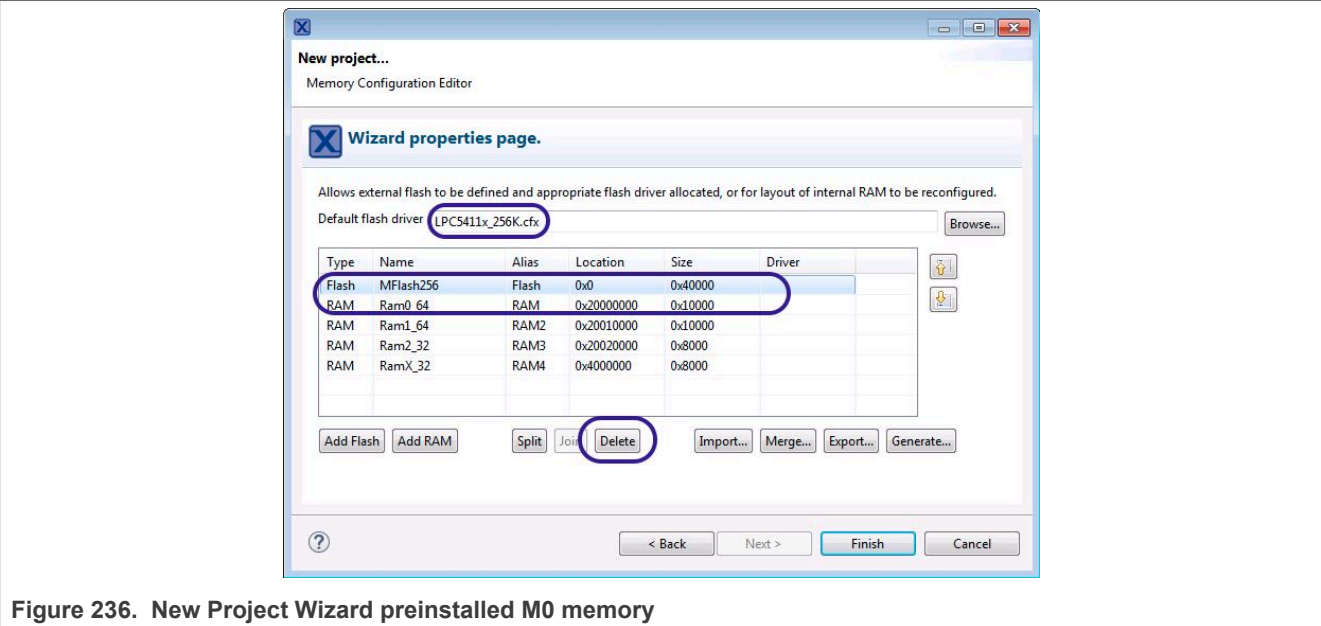
Figure 236. New Project Wizard preinstalled M0 memory

The memory setting should then look as below. In this case, the code and data of our secondary project are linked to address 0x20010000 with the stack set to the top of this region.
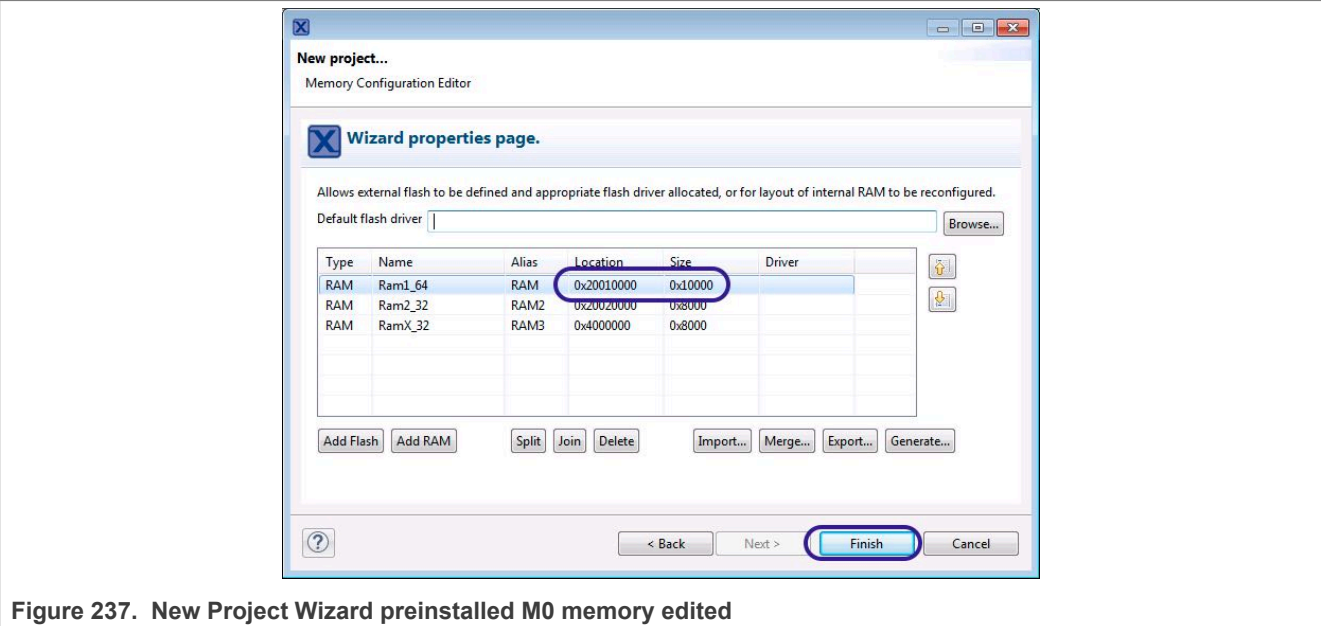
Figure 237. New Project Wizard preinstalled M0 memory edited

Now, click Next -> Finish to complete the M0 secondary projects creation.

21.3.2 Creating the M4 primary project

To create the primary, Launch the New Project Wizard again and this time select the LPC54114 (M4) part and click Next. Select the matching 'MultiCore M4 Master -> LPCOpen -C Project' and click Next again. Now, name the new project, for example, LPC54114_M4_Master and click next until the Multicore Project Setup page is reached.

Note: The wizard presents an identical memory configuration page, but on this occasion, no editing is required since the default Flash and RAM settings are used.

From here, click browse to select the previously created secondary project from the existing workspace.
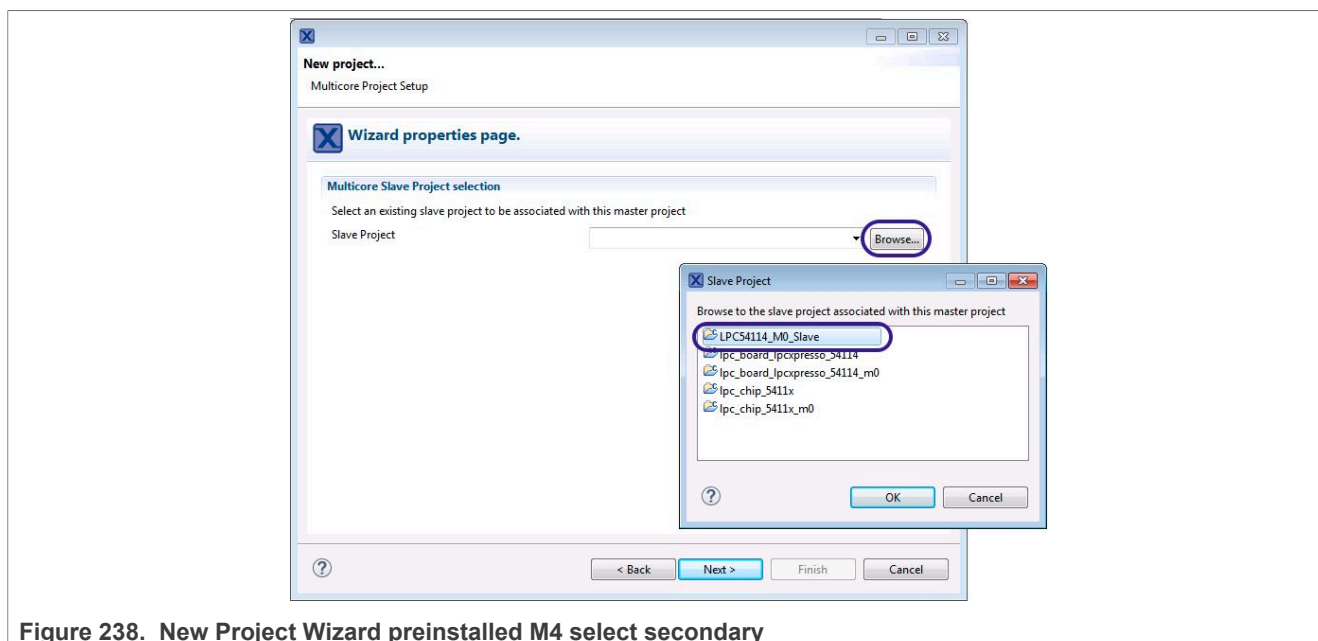
Figure 238. New Project Wizard preinstalled M4 select secondary

Now, click Next -> Finish to complete the M4 primary projects creation.

21.4 Debugging multicore projects

The debug story for multicore MCUs can vary with their implementation and also the chosen debug solution.

Our multicore model, as described above, assumes that the primary project both copies the secondary MCUs code and data (into RAM) but also releases the secondary core from reset. Therefore, the primary project should be run (debugged) first and typically run to main(). Once here, the instantiation of the code of the secondary core is complete, but the secondary core was not released. On some MCUs, a debug connection can be made to the secondary core before it has been released, but on others, this is only possible after they are released.

Note: Secondary projects debug launch configurations may require user modification before a debug connection can be made. See the section Secondary project debug.

In our example LPC54114, the debug connection of the secondary core can be made when the primary core reaches main(). The debug window then looks similar to that below.
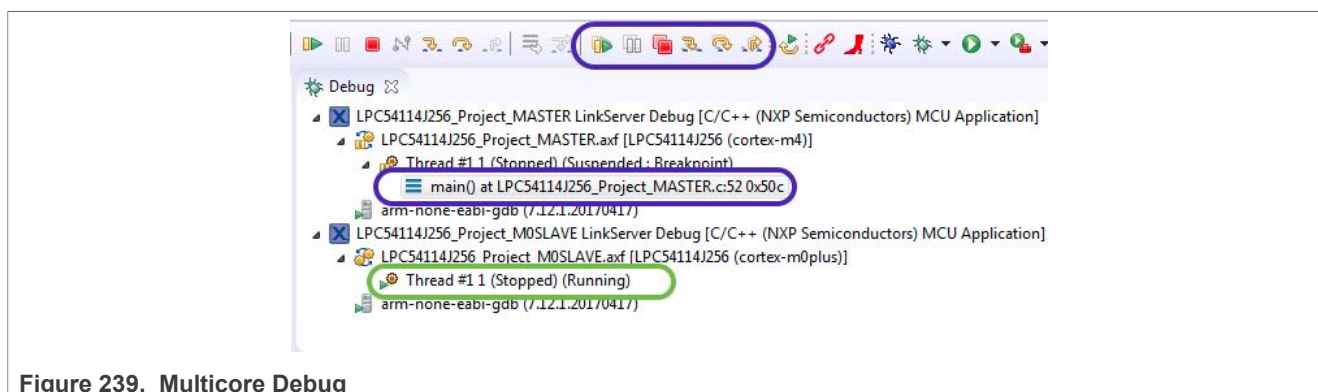
Figure 239. Multicore Debug

Note above: The multicore debug controls have been highlighted, these controls differ from the standard controls in that they operate on all cores being debugged. Via these, the system to be stepped, run, paused, terminated, and so on.

In addition, the debug stack of the M4 primary core (blue) is shown stopped at main, while the stack of the secondary core (green) is waiting to be released by the primary core; clicking between these stacks changes the debug scope of the IDE from one core to the other. The currently selected core is the one used for displaying many of the debug-related views, such as Registers and Locals.

21.4.1 Controlling debug views

It is also possible to create copies of many of the debug-related views, and then lock each copy to a particular core (as described below).

For example, to create two register views, one for the M4 and one for the M0+ ...First of all, use the 'Open New view' button in the Registers view to create a second Registers view:
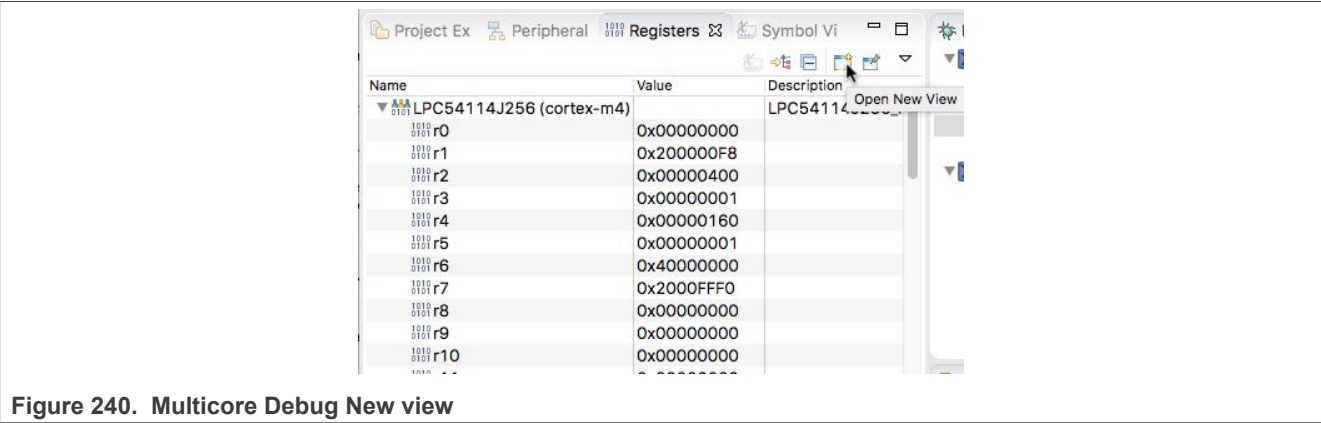
Figure 240. Multicore Debug New view

Now, pin the original view to the core currently selected in the Debug, using the 'Pin to Debug context' button:
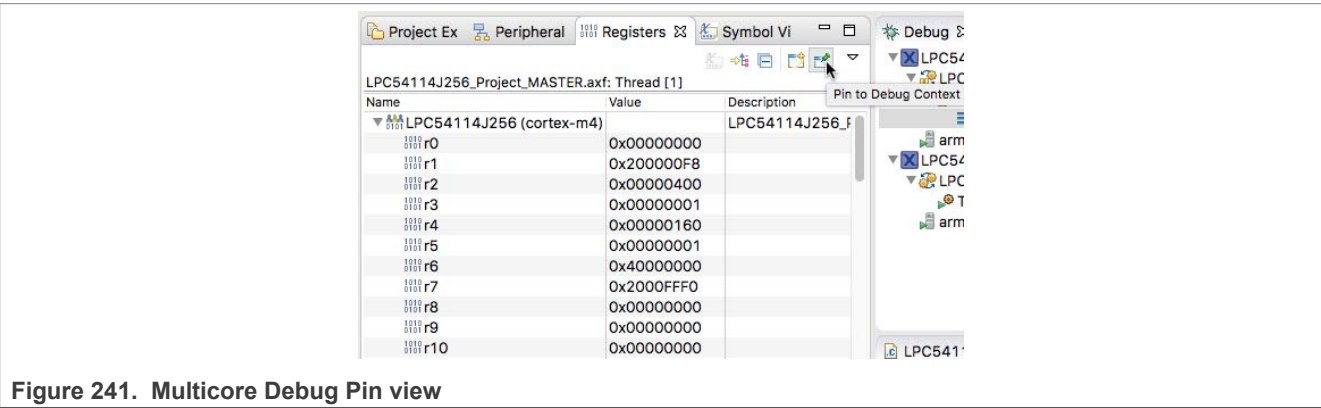
Figure 241. Multicore Debug Pin view

Now, select the other core in the Debug view, and go to the second Register view. Use the 'Pin to Debug Context' button from this view to lock this second Registers view to the selected core:
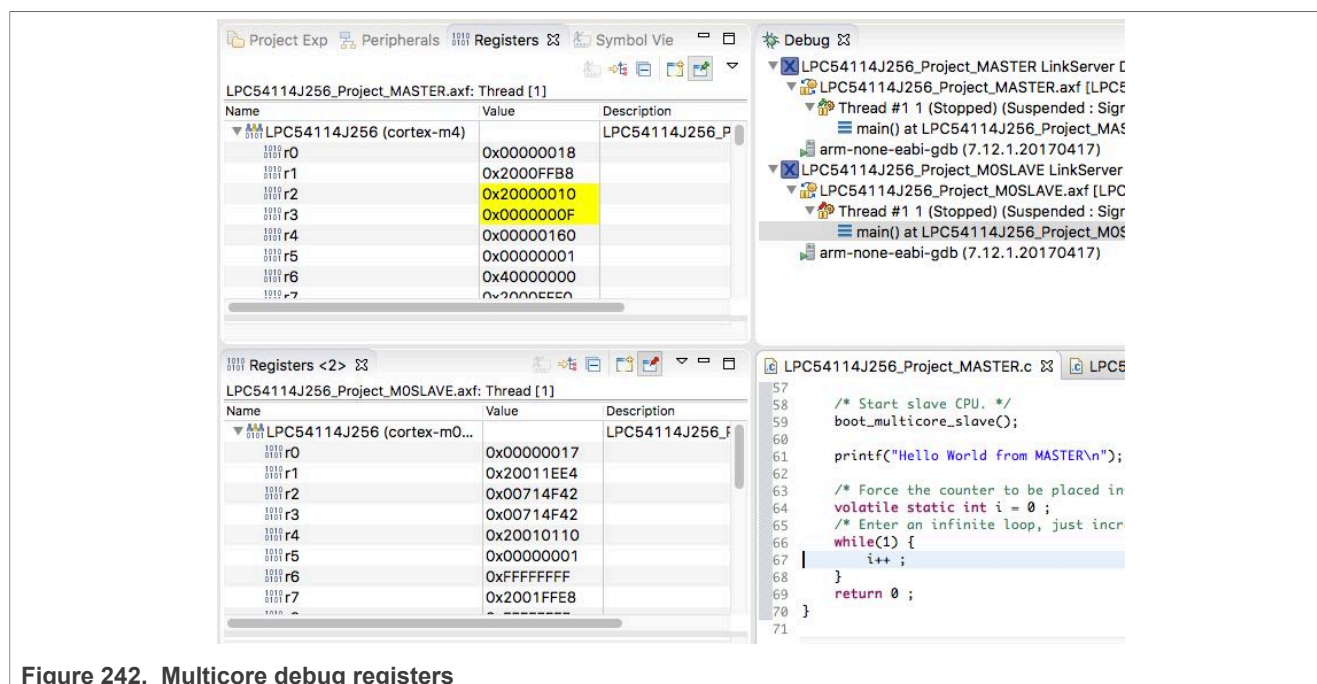
Figure 242. Multicore debug registers

21.4.2 Secondary project debug

Typically, the primary project is debugged first in the same way as a single CPU project. However, the debug launch configuration of the secondary project may require special settings to establish a debug connection to the secondary CPU.

MCUXpresso IDE automatically configures the correct settings for secondary launch configurations on all debug solutions. However, there might be situations when the debug settings of the secondary project may require modifications.

- **Core Selection** - within a multicore MCU there is more than one CPU (sometimes referred to as a device). The debug connection has to be made to the appropriate internal CPU for both the primary and secondary projects.
  - **LinkServer CMSIS-DAP Debug**: this process is **automatic** and hidden from the user. The selection details are stored within the build configuration folder(s) of the project and take the suffix .jtag or .swd
  - **PEmicro Debug**: appropriate cores inside primary and secondary launch configurations are **automatically** selected by the IDE
  - **SEGGER Debug**: appropriate cores inside primary and secondary launch configurations are **automatically** selected by the IDE. However, there might be certain situations when the secondary core should be manually selected. In this case, the IDE displays a warning about the incapability of finding a matching core (based on the project description).
- **Attach mode** for the secondary CPU - as described above, the debug connection to the secondary core(s) should be via an attach. This option is set **automatically** when the debug launch configuration is created.
- **Managing the Debug Server** - this is the low-level interface between the debugger and the target. The launch configuration is **automatically** correctly configured when the debug connection is made.

21.4.3 Auto-debug secondary project(s) for multicore projects

When using LinkServer as a debug connection, secondary projects can be automatically debugged once initiating debug on the primary project. This behavior is controlled by a LinkServer-specific preference that can be accessed via Eclipse menu -> **Window -> Preferences -> MCUXpresso IDE -> Debug Options -**

> LinkServer Options -> Miscellaneous -> "Enable auto-debug secondary project(s) for multicore projects". By default, the preference is enabled in a fresh workspace, meaning that debug sessions for secondary projects are automatically started.
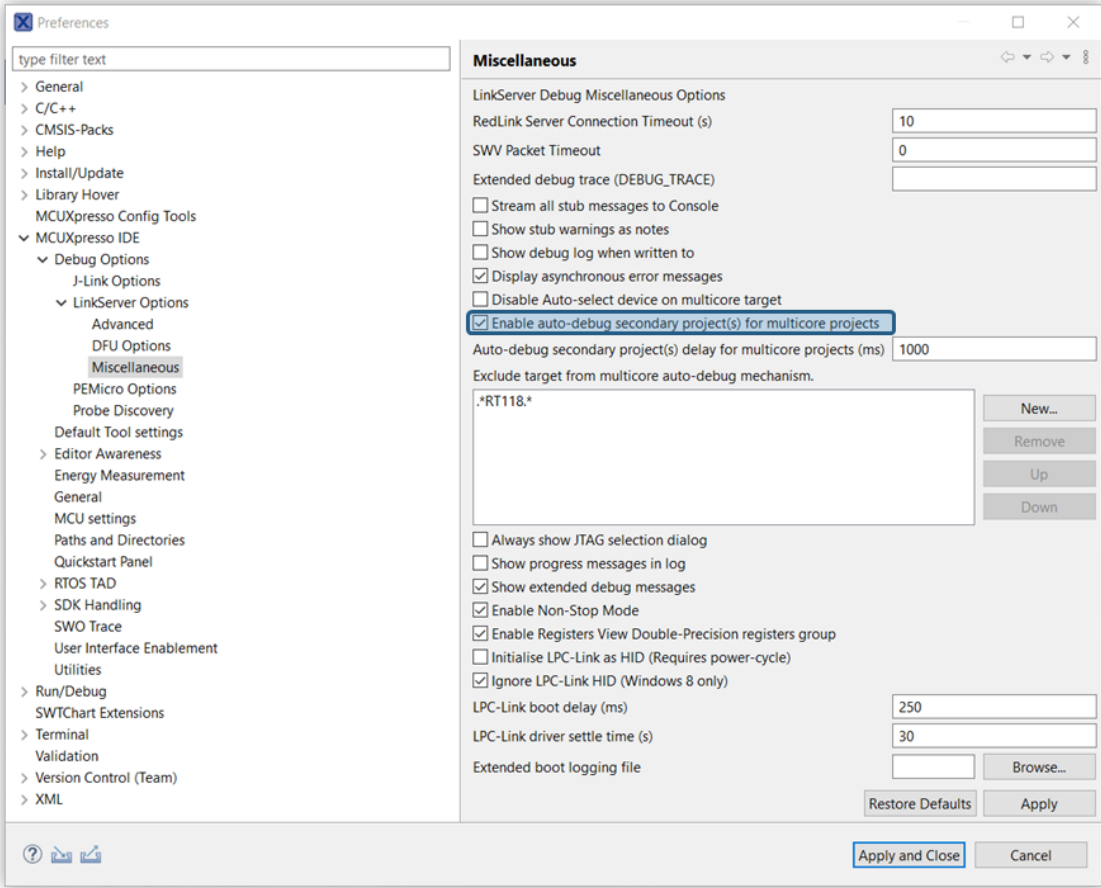
Figure 243. Auto-debug secondary project enable option

Subsequent debug sessions started for secondary projects are delayed and the actual delay is specified in another preference, as depicted below. If required, users can change the default 1000 ms value.
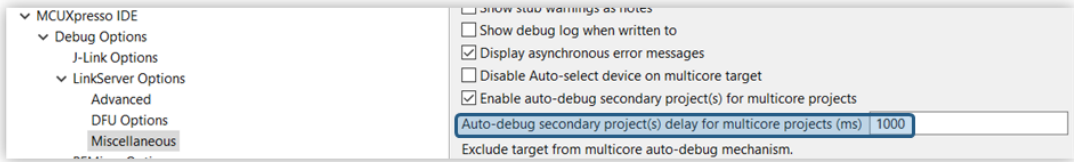
Figure 244. Auto-debug secondary project delay option

To refine which projects are affected by auto-debug, the preference page allows you to specify, via regular expressions, a list of excluded devices. Projects targeting devices whose names match any of the entries in the list do not trigger the auto-debug mechanism for secondary project(s).
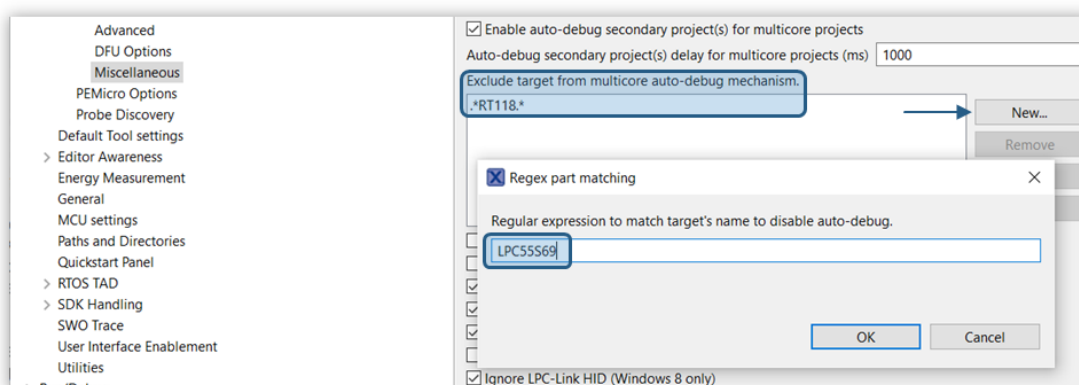
Figure 245. Auto-debug secondary project filter list

In the case, you don't want to have this feature enabled (so if you want to start debug sessions for each core independently), uncheck this option.

Similar to LinkServer, the option for auto-debug of secondary project(s) for multicore projects becomes enabled by default for multicore debug purposes when using PEmicro and J-Link. To configure the enablement, go to the appropriate Preferences page:

- **PEmicro**: **Window -> Preferences -> MCUXpresso IDE -> Debug Options -> PEMicro Options -> "Enable auto-debug secondary project(s)"**
- **J-Link**: **Window -> Preferences -> MCUXpresso IDE -> Debug Options -> J-Link Options -> "Enable auto-debug secondary project(s)"**

21.5 Multicore projects additional information

21.5.1 Defines

A number of compiler defines are automatically created for LPC5410x projects to allow conditional compilation of certain blocks of code depending upon whether a specific project is configured to be a secondary, a primary, or neither.

__MULTICORE_MASTER

- Defined automatically for a project that has been configured to be a primary project

__MULTICORE_MASTER_SLAVE_M0SLAVE

- Defined automatically for a project that has been configured to be a primary project and has had a secondary project associated with it (therefore indicating to the primary project which CPU type the secondary project is for).

__MULTICORE_M0SLAVE

- Appropriate one defined automatically for a project that has been configured to be a secondary project

__MULTICORE_NONE

- Defined automatically for a project, which has not been configured as either a secondary or primary project

Note: The multicore support within MCUXpresso IDE is highly flexible and provides functionality beyond that required for the LPC5411x family. Therefore, the symbols __MULTICORE_MASTER_SLAVE_M4SLAVE and __MULTICORE_M4SLAVE are also provided for completeness.

21.5.2 Secondary boot code

boot_multicore_slave() is called by the primary project code created directly by the New project wizard to release the secondary core from sleep.

Note: The source files containing this function are included in all LPC541xx projects, but are conditionally compiled so that it is included only when required. This has been done to allow projects originally created, for example, as a secondary project, to be reconfigured (via the project properties – linker multicore tab) as a primary project.

21.5.3 Reset handler code

When configured as a primary project, the LPC541xx startup file is built with additional (assembler) code at the beginning of the reset handler, ResetISR(), with the 'standard ' reset handler code moved to ResetISR2().

This additional code is required to allow correct booting of both the primary and secondary cores. It is written in assembler to force it to be "Thumb1" code, and therefore runnable by both cores.

22 Appendix - Additional hints and tips

These additional hints and tips extend the information provided in the main body of this guide.

22.1 Part support handling from SDKs

MCUXpresso IDE needs specific device information provided by the SDK to properly:

- Create/import projects
  - With part-specific startup code
- Define memory layout
- Create debugging launch configuration
- Perform flash programming

This detailed part knowledge is known as **Part Support**.

22.1.1 SDK version control

MCUXpresso IDE obtains new Part Support from *installed* SDKs. The internal database of the IDE only uses SDKs with the highest version number (latest version is v2.9). For example, a user may have installed two SDKs for a single part:

- SDK_2.3.0_FRDM-K64F
- SDK_2.0.0_FRDM-K64F

The IDE loads only the 2.3.0 version of that SDK, and also provides a warning in the SDK View header:

Figure 246. Installed SDKs version warning

In this situation, it is likely that the user no longer needs the older version of the SDK. Therefore the IDE provides an option to delete this older SDK by clicking the warning message, and clicking the 'X'.

Figure 247. Installed SDKs delete older version

Note: Installation of a new SDK for a part always replaces any previously installed older SDK for that part, even if the new SDK is deactivated (by unchecking the associated tick box). Deactivating an SDK results in the removal of part support and wizard from internal views. These are restored after activating the SDK again.

22.1.2 SDK manifest versioning

Along with SDK versioning, also the internal manifest in an SDK can have multiple versions. MCUXpresso IDE loads the manifest associated with its internal version head info. Therefore, assuming an IDE with the internal head version set to 3.3, we could have an SDK with the following manifests:

- Manifest version 3.3
- Manifest version 3.2
- Manifest version 3.1
- Manifest version 3.0

In such a case, the IDE loads the manifest version 3.3.

After loading, the IDE validates the manifest against the schema version head, and if for any reason this is not valid, it tries with the other schema versions. If it cannot validate the manifest 3.3, then it tries with manifest 3.2, validating it, and so on. Manifest version details appear in the SDK View, while the Error log shows any validation errors that have appeared in the process.

In the case that the IDE loads an older manifest, or in the case the SDK contains a manifest 3.4 and the IDE manifest head is 3.3, the SDK View decorates the SDK image with a warning and, by clicking the SDK, a message appears in the SDK view header:
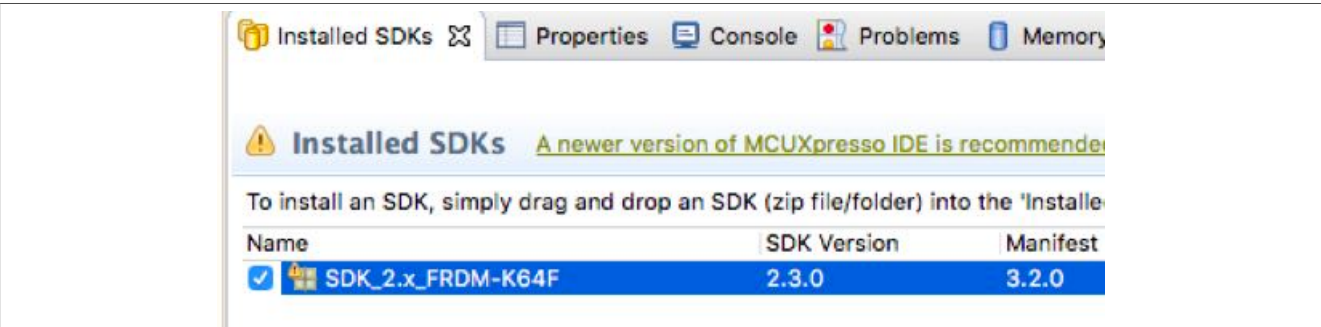
Figure 248. Installed SDKs new version message

The full error looks like: *"A newer version of the MCUXpresso IDE is recommended for use with the selected SDK. Please update your MCUXpresso IDE in order to get full SDK features"*

Note: Even if not intended, newer SDKs may support features not understood by the current version of the IDE. A message appears to warn users that there is a mismatch between the SDK and IDE capabilities.

22.1.3 Device versions

If the user installs more than one SDK containing the same device (that is, a device with the same identifier), the IDE loads the part support from the device with the highest version number, regardless of which SDK it is located within. If two or more SDKs have the same device with the same version number, then the order the host OS presents them to the IDE determines which SDK to use.

If an SDK in the Installed SDK view contains a device that is not installed (because another SDK supplies it), its image (and the device in the SDK tree) is decorated with an icon:
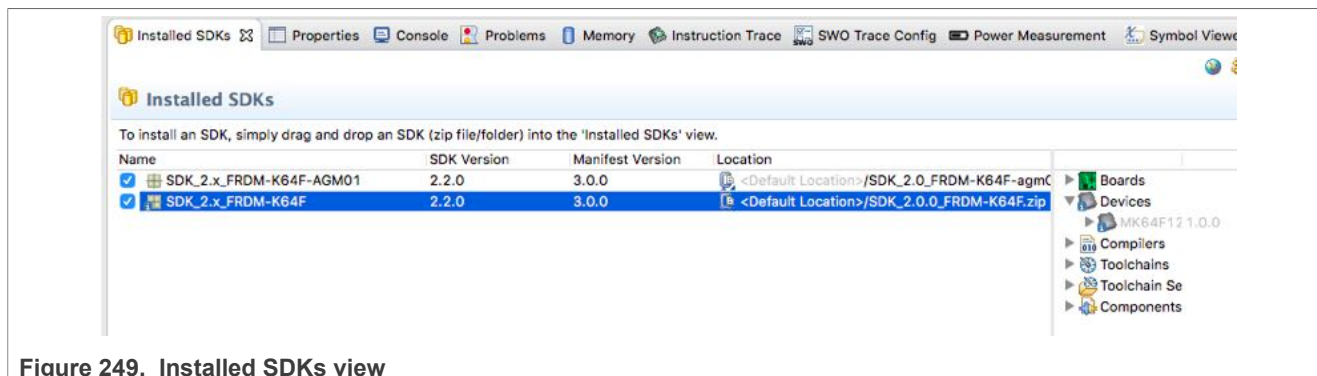
Figure 249. Installed SDKs view

22.2 How do I switch between Debug and Release builds?

By default, MCUXpresso IDE projects automatically have two build configurations, *Debug* and *Release*. Typically a project is developed using the Debug build variant, but switched to Release late in the development cycle to benefit from more compilation optimisations.

22.2.1 Changing the build configuration of a single project

You can switch between Debug and Release build configurations by selecting the project you want to change the build configuration for in the Project Explorer view, then using one of the below methods:

- Select the menu item **Project->Build Configuration->Set Active** and select **Release** or **Debug** as necessary
- Use the dropdown arrow next to the 'sundial' (Manage configurations for the current project) icon on the main toolbar (next to the 'hammer' icon) and select **Release** or **Debug** as necessary. Alternatively, you can use the dropdown next to the 'hammer' icon to change the current configuration and then immediately trigger a build.
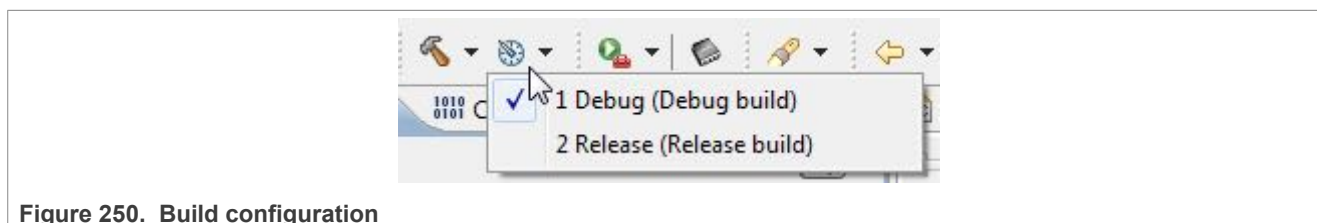
Figure 250. Build configuration

- Right-click in the Project Explorer view to display the context-sensitive menu and select the **Build Configurations->Set Active** entry.

22.2.2 Changing the build configuration of multiple projects

It is also possible to set the build configuration of multiple projects at once. This may be necessary if you have a main application project linked with a library project, or you have linked projects for a multicore MCU such as an LPC43xx or LPC541xx (one project for the primary Cortex-M4 CPU and another for a secondary Cortex-M0/M0+ CPU).

To do this, first of all, you need to select the projects that you wish to change the build configuration for in the Project Explorer view - by clicking to select the first project, then use shift-click or control-click to select additional projects as appropriate. If you want to change all projects, then you can simply use Ctrl-A to select all of them.

Note: it is important that when you select multiple projects, you should ensure that none of the selected projects are opened out - in other words, when you selected the projects, you must not have been able to see any of the files or the directory structure within them. If you do not do this, then some methods for changing the build configuration will not be available.

After selecting the required projects, you then need to simply change the build configuration as you would do for a single project.

22.3 Editing hints and tips

The editor view within Eclipse, which sits under the MCUXpresso IDE, provides a large number of powerful features for editing your source files.

22.3.1 Link Project Explorer view to the active editor

Eclipse offers the possibility to highlight the file opened in the active editor, inside Project Explorer view. When multiple files are opened, the switch to a new editor also updates the Project Explorer selection. You can control the enablement of this feature by using the "Link with Editor" toggle button inside the Project Explorer view, as illustrated in the picture below.
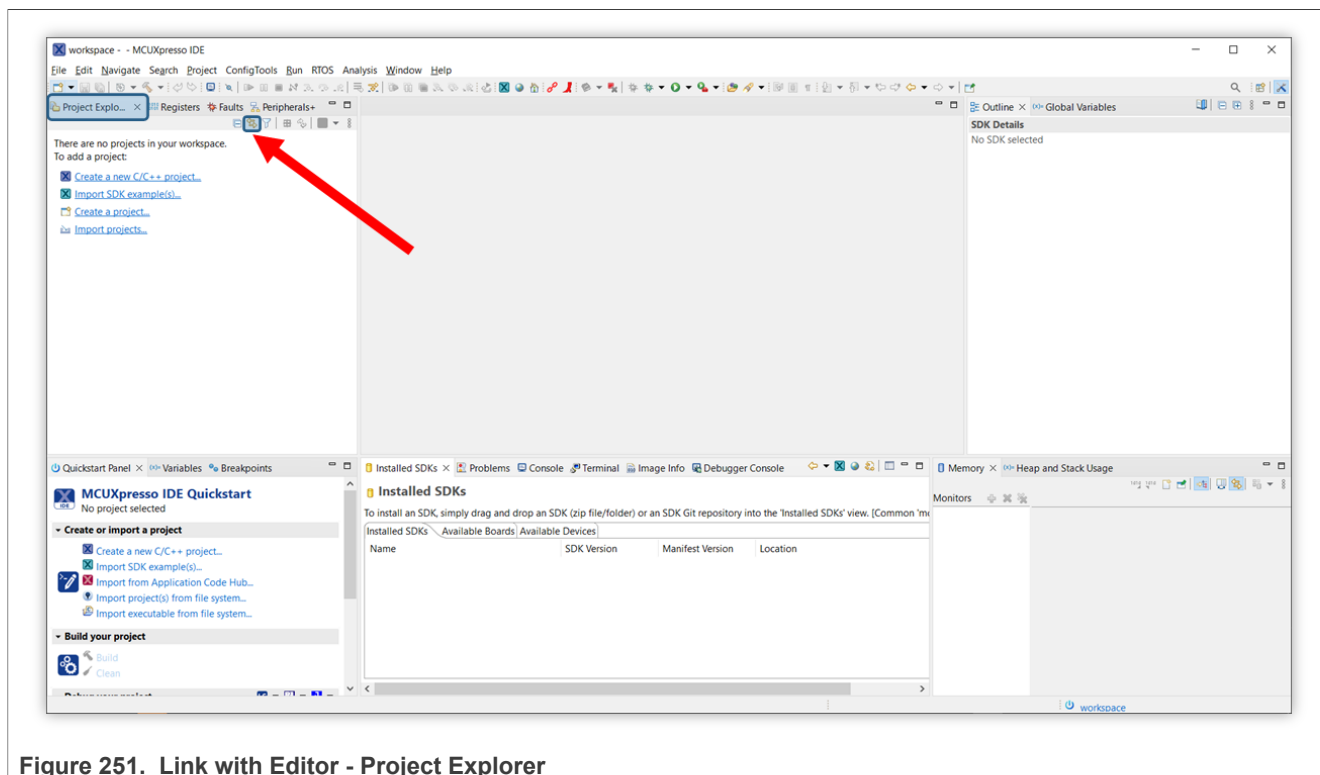
Figure 251. Link with Editor - Project Explorer

MCUXpresso IDE adds extra flexibility for the above-mentioned feature by allowing users to set a certain configuration to use at IDE startup. The user can control the enablement of "Link with Editor" by using the Preferences page accessed via **Window -> Preferences -> MCUXpresso IDE -> General**. The screenshot below highlights the two relevant checkboxes:

1. It controls MCUXpresso IDE-specific feature. In other words, if ticked, the IDE enables or disables the "Link with Editor" functionality according to the next checkbox.
2. It controls whether to enable or disable "Link with Editor" at IDE startup. This is only taken into consideration when the previous checkbox is ticked.
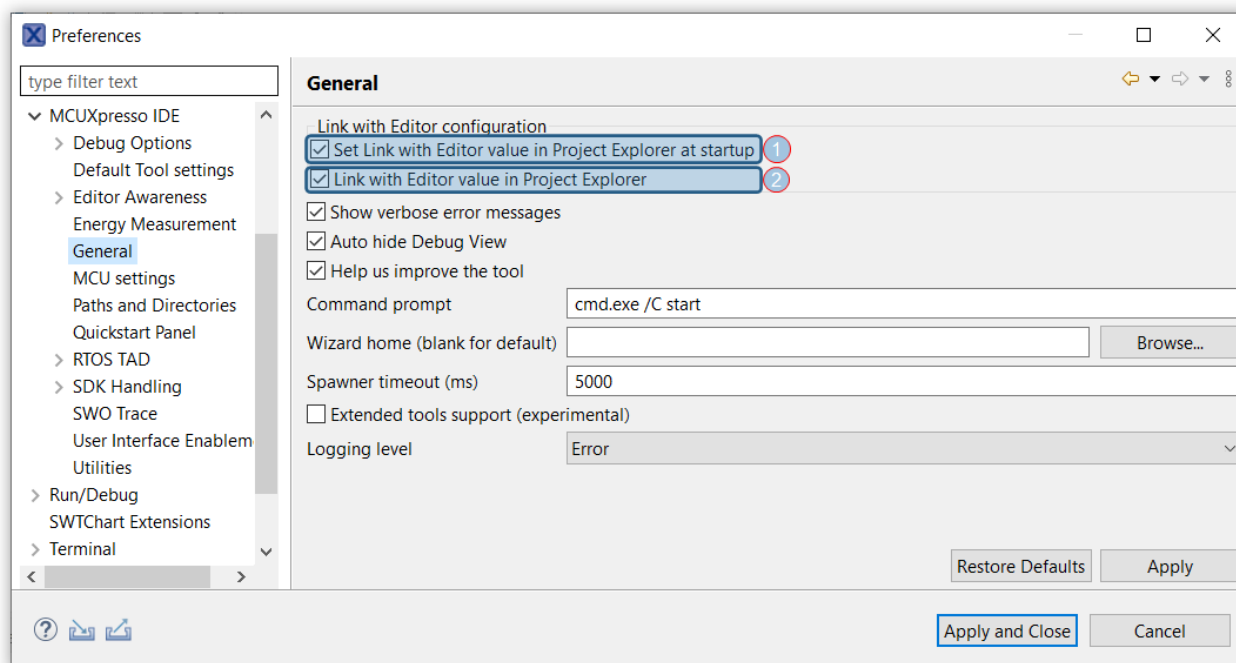
Figure 252. Link with Editor - Preferences page

Note that "Link with Editor" also works while having an active debug session and navigating through the code actively debugged.

22.3.2 Multiple views onto the same file

The **Window -> Editor** menu provides several ways of looking at the same file in parallel.

• **Clone**: two editor views onto the same file
• **Toggle Split Editor**: splits the view onto the current file into two (either horizontally or vertically)

22.3.3 Viewing two edited files at once

To see more than one file at the same time, simply click the file tabs that you have open in the editor view, and then keep the mouse button held down and drag that file tab across to the right. After you've moved to the side, or below, an outline should appear, showing you the future placement of that tab after releasing the mouse button.

22.3.4 Source folding

Within the editor view, functions, structures, and so on, may be folded to show the structure and hide the details.

The user can control folding via right-clicking in the margin of the editor view to bring up the context-sensitive menu, then selecting **Folding -> <option required>**.

When folding is enabled, you can then click the **+** or **-** icon that now appears in the margin next to each function, structure, and so on, to expand or collapse it, or use the **Folding -> Expand all** and **Folding -> Collapse all** options from the context-sensitive menu.

It is also possible to control various settings for Folding through:

Preferences -> C/C++ -> Editor -> Folding

22.3.5 Editor templates and Code completion

Within the editor, a number of related pieces of functionality allow you to enter code quickly and easily.

First of all, templates are fragments of code that can be inserted in a semi-automatic manner to ease the entering of repetitive code - such as blocks of code for C code structures such as for loops, if-then-else statements, and so on.

Second, the indexing of your source code that is done by default by the tools, allows for auto-completion of function and variable names. This is known as "content assist".

- Ctrl-Space at any point lists available editor templates, function names, and so on
- Ctrl-Shift-Space displays function parameters
- Alt-/ for word completion (press multiple times to cycle through multiple options)

In addition, the predefined templates are user-extensible via:

Preferences -> C/C++ -> Editor -> Templates

22.3.6 Brace matching

The editor can highlight corresponding open and closing braces in a couple of ways.

First of all, if you place the cursor immediately to the right of a brace (either an opening or closing brace), then the editor displays a rectangle around the corresponding brace.

Second, if you double-click immediately to the right of a brace, then the editor automatically highlights all of the text between this brace and the corresponding one.

22.3.7 Syntax coloring

Syntax Coloring specifies how to render your source code in the editor view, with different colors used for different elements of the code. The settings used can be modified in:

Preferences -> C/C++ -> Editor -> Syntax Coloring

Note that you can configure general text editor settings such as the background color in:

Preferences -> General -> Text Editors

You can also configure fonts in:

Preferences -> General -> Appearance -> Colors and Fonts

22.3.8 Comment/uncomment block

The editor offers various ways of commenting in or out one or more lines of text. The user can access these by using the Source entry of the editor context-sensitive menu, or using the following keyboard shortcuts...

- Select the line(s) to comment, then hit Ctrl-/ to comment out using // at the start of the line, or uncomment if the line is commented out.

- Select the line(s) to comment, then hit Ctrl-Shift-/ to block comment out (placing /* at the start and */ at the end).
- To remove a block comment, hit Ctrl-Shift-.

22.3.9 Format code

The editor can format your code to match the coding standards in use (**Preferences -> C/C++ -> Code Style**). This can automatically deal with layout elements such as indentation and where to place braces. You can perform this action on the currently selected text by using the Source->Format entry of the editor context-sensitive menu, or using the keyboard shortcuts Ctrl-Shift-F. If no text is selected, then the formatting takes place on the whole of the current file.

22.3.10 Correct indentation

As you enter code in the editor, it attempts to automatically indent your code appropriately, based on the code standards in use, and also the layout of the preceding text. However, in some circumstances, for example after manually laying out text, you may end up with incorrect indentation.

This can usually be corrected using the Source->Correct Indentation entry of the editor context-sensitive menu, or using the keyboard shortcut Ctrl-I.

Alternatively, use the "Format code" option, which fixes other layout issues in addition to indentation.

22.3.11 Insert spaces for tabs in editor

You can configure the IDE so that when editing a file, pressing the TAB key inserts spaces instead of tab characters. To do this go to:

Preferences -> General -> Editors -> Text Editors

and tick the "Insert spaces for tabs" box. If you tick "Show white-space characters" you can see whether a tab character, or space characters are being inserted when you press the TAB key.

22.3.12 Replacing tabs with spaces

To replace existing tabs with spaces throughout the file, open the Code Style preferences:

Preferences -> C/C++ -> Code Style

- Select a Code Style profile and then select Edit...
- Choose the Indentation tab
- For the Tab policy, select Spaces only
- Apply the changes
  - **Note**: If the Code Style has not been edited before, you must rename the Profile before applying the change.
- The new style is applied when the source is next formatted using Source -> Format

22.4 Hardware floating-point support

Most ARM-based systems - including those based on Cortex-M0, M0+, and M3, have historically not implemented any form of floating point in hardware. This means that any floating point operations contained in your code are converted into calls to library functions that then implement the required operations in the software.

However, many Cortex-M4 based MCUs do incorporate a single-precision floating point hardware unit. **Note**: the optional Cortex-M4 floating-point unit implements single-precision operations (C/C++ float) only. Therefore,

if your code uses double-precision floating point (C/C++ double), then any such floating-point operations contained in your code are still converted into calls to library functions that then implement the required operations in the software.

Similarly, Cortex-M7-based MCUs may incorporate a single-precision or double-precision floating-point hardware unit.

22.4.1 Floating-point variants

When implementing a hardware floating-point unit, ARM defines that it may be used in one of two modes.

SoftABI

- Single-precision floating-point operations are implemented in hardware and therefore provide a large performance increase over code that uses traditional floating-point library calls, but when making calls between functions, any floating-point parameters are passed in ARM (integer) registers or on the stack.
- SoftABI is the 'most compatible' as it allows code that is not built with hardware floating-point usage enabled to be linked with code that is built using software floating point library calls.

HardABI

- Single-precision floating-point operations are implemented in hardware, and floating-point registers are used when passing floating-point parameters to functions.

22.4.2 Floating point use - preinstalled MCUs

When targeting preinstalled MCUs, MCUXpresso IDE generally assumes that when using the Cortex-M4 hardware floating point, then the SoftABI is used. Therefore generally, this is the mode that example code (including, for example, LPCOpen chip and board libraries) is compiled for. This is done as it ensures that components tend to work out of the box with each other.

When you use a project wizard for a Cortex-M4 where a hardware floating-point unit may be implemented, there is an option to enable the use of the hardware within the options of the wizard. This defaults to SoftABI - for compatibility reasons.

Selecting this option makes the appropriate changes to the compiler, assembler, and linker settings to cause SoftABI code to be generated. It also typically enables code within the startup code generated by the wizard that turns on the floating-point unit.

You can also select the use of HardABI in the wizards. Again, this causes the appropriate tool settings to be used. But if you use this, you must ensure that any library projects used by your application project are also configured to use HardABI. If such projects exist, then you can manually modify the compiler/assembler/linker settings in Project Properties to select HardABI.

Warning: Creating a project that uses HardABI when linked library projects have not been configured and built with this option results in link time errors.

22.4.3 Floating point use - SDK-installed MCUs

When targeting SDK installed MCUs, MCUXpresso IDE generally assumes that when hardware floating point is available, then the HardABI is used. This generally works without a problem as generally projects for such MCUs contain all required code (with no use of library projects).

However, it is still possible to switch to using SoftABI using the "Advanced Properties settings" page of the |New project" and "Import SDK examples" wizards.

22.4.4 Modifying floating-point configuration for an existing project

If you wish to change the floating point ABI for an existing project (for example to change it from using SoftABI to HardABI), then go to:

Quickstart -> Quick Settings -> Set Floating Point type

and choose the required option.

Alternatively, you can configure the settings manually by going to:

Project -> Properties -> C/C++ Build -> Settings -> Tool Settings

and changing the setting in **ALL** of the following entries:

- MCU C Compiler -> Architecture -> Floating point
- MCU Assembler -> Architecture & Headers -> Floating point
- MCU Linker -> Architecture -> Floating point

Note: For C++ projects, you also have to modify the setting for the MCU C++ Compiler. **Warning**: Remember to change the setting for all associated projects, otherwise linker errors may result.

22.4.5 Do all Cortex-M4 MCUs provide floating point in hardware?

Not all Cortex-M4-based MCUs implement floating point in hardware, so check the documentation provided for your specific MCU to confirm.

In particular, with some MCU families, some specific MCUs may not provide hardware floating point, even though most of the members of the family do (for example the LPC407x_8x). Therefore, it is a good idea to double-check the documentation, even if the project wizard in the MCUXpresso IDE for the family that you are targeting suggests that hardware floating point is available.

22.4.6 Why do I get a hard fault when my code executes a floating-point operation?

If you are getting a hard fault when your application tries to execute a floating point operation, then you are almost certainly not enabling the floating-point unit. This is normally done in the LPCOpen or SDK initialization code, or else in the startup file that MCUXpresso IDE generates. But if there are configuration issues with your project, then you can run into problems.

For more information, see the Cortex-M4 Technical Reference Manual, available on the ARM website.

22.5 LinkServer scripts

The LinkServer debug server supports a Basic-like programming language that can be used to script low-level target operations. Within a LinkServer debug connection, we provide two callouts where scripts can be referenced (if required). The first callout is intended to assist with the initial debug connection, via a Connect Script, and the second is to assist with the targets reset via a Reset Script.

These scripts are specified within a LinkServer launch configuration file and are preselected if needed for projects performing standard connections to known debug targets.

Note: Starting with MCUXpresso IDE v11.9.0, LinkServer-specific scripts are part of the standalone LinkServer package. Therefore, they can be inspected directly inside the LinkServer installation folder or via the */LinkServer* symbolic link, more specifically */LinkServer/binaries/Scripts*.

22.5.1 Supplied scripts

A set of scripts are supplied within the MCUXpresso IDE installation at:

```
<install dir>/ide/LinkServer/binaries/Scripts
```

These scripts are used to prepopulate LinkServer launch configuration files when needed.

The purpose of certain scripts is described below:

- **kinetismasserase.scp** - invoked by the GUI Flash Programmer to resurrect a locked Kinetis device
- **kinetisunlock.scp** - if for any reason the GUI Flash Programmer fails to resurrect a locked Kinetis part (as above), this script can be specified in place of the above and the recovery attempt repeated
- **MCXC_masserase.scp** - invoked by the GUI Flash Programmer to resurrect a locked MCXC device
- **MCXC_unlock.scp** - if for any reason the GUI Flash Programmer fails to resurrect a locked MCXC part (as above), this script can be specified in place of the above and the recovery attempt repeated
- **delayexample.scp** - an example script showing how a delay can be performed

Note: Some chips also require a preconnect script that prepares the target MCU for the initial debug connection. A set of preconnect scripts can be found within the MCUXpresso IDE installation at:

```
<install dir>/ide/LinkServer/binaries/ToolScripts
```

22.5.2 User scripts

Additional user-generated scripts can be added directly to the product installation but more typically they should be located within a project. The LinkServer launch configuration allows the location of scripts to be either project-relative, absolute, or product-local.

22.5.3 Debugging code from RAM

[This section is deprecated - see Converting projects to tun from RAM with LinkServer for details of the improved scheme]

MCUs have well-defined boot strategies from reset, typically they first run some internal manufacturer boot ROM code that performs some hardware setup and then control passes to code in flash (that is, the user's Application).

On occasion, it can be useful to run and debug code directly from RAM. Since an MCU does not boot from RAM, a scheme is needed to take control of the reset mechanism of the debugger. This can be achieved with the use of a LinkServer reset script.

Within MCUXpresso IDE, certain pre-created scripts are located at:

```
{install dir}/ide/LinkServer/binaries/Scripts
```

Contained in this directory is a script called *kinetisRamReset.scp* (see below).

```
10  REM Kinetis K64F Internal RAM (@ 0x20000000) reset script
20  REM Connect script is passed PC/SP from the vector table in the image by the
 debugger
30  REM For the simple use case we pass them back to the debugger with the
 location of \
    the reset context.
40  REM
50  REM Syntax here is that '~' commands a hex output, all integer variables are
 a% to z%
```

```
70  REM Find the probe index
80  p% = probefirstfound
90  REM Set the 'this' probe and core
100 selectprobecore p% 0
110 REM NOTE!! Vector table presumed RAM location is address 0x20000000
120 REM The script passes the SP (%b) and PC (%a) back to the debugger as the
 reset context.
130 b% = peek32 this 0x20000000
140 a% = peek32 this 0x20000004
145 d% = 0x20000000
150 print "Vector table SP/PC is the reset context."
160 print "PC = "; ~a%
170 print "SP = "; ~b%
180 print "XPSR = "; ~c%
185 print "VTOR = "; ~d%
190 end
```

This reset script assumes that the user intends to run code from RAM at 0x20000000 - this is the value of the SRAM_Upper RAM block on Kinetis parts.

Note: To build a project to link against RAM, you can simply delete any flash entries within the memory configuration of the project. If the MCUXpresso IDE default linker settings are used, then the project links to the first RAM block in the list. For many Kinetis parts, this address matches the expected address within the script. For some parts (for example KLxx) however, the first RAM block may take a different value. This problem can be resolved by editing the script or modifying the RAM addresses of the project.

For users of LPC parts, the RAM addresses are different but the principal remains the same. Within the *Scripts* directory, you can find a RAM reset script for the LPC18LPC43 parts. This script is identical to the one above apart from the assumed RAM address.

Finally, to use the script, simply edit the launch configuration of the project for the 'Reset Script' entry, and browse to the appropriate 'RAMReset.scp' script.

Note: When executing code from RAM, the Vector table of the project can also be located at the start of the RAM block. Cortex-M MCUs can locate their vector table using an internal register called *VTOR* (the vector table offset register). Typically, this register is set automatically by the startup or init code of a project. However, if execution fails when an interrupt occurs, check that this register is set to the correct value.

22.5.4 LinkServer scripting features

LinkServer scripts are written in a simple version of the BASIC programming language. In this variant of BASIC, 26 variables are available (%a through %z). On entry to the script, some variables have assigned values:

```
a% is the PC
b% is the SP
c% is the XPSR
d% is the VTOR
```

On exit from the script, a% is loaded into the PC, b% is loaded into the SP, and d% is loaded into the VTOR, therefore providing a way for the script to change the startup behavior of the application.

They offer functionality as shown below:

Generic BASIC-like functions that only work inside scripts

```
GOTO 'LineNumber'
IF 'relation' THEN 'statement'
REPEAT: Start of a repeat block
UNTIL 'relation': End with condition of repeat block
```

```
BREAKREPEATTO 'LineNumber': Premature end of a repeat loop
GOSUB 'LineNumber'
RETURN
TIME: Returns a 10ms incrementing count from the host
```

Generic BASIC-like functions

```
PEEK8  {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address>
PEEK16 {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address>
PEEK32 {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address>
POKE8  {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address> <Data>
POKE16 {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address> <Data>
POKE32 {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address> <Data>
QPOKE8  {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address> <Data>
QPOKE16 {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address> <Data>
QPOKE32 {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address> <Data>
QSTARTTRANSFERS {[THIS] | [<ProbeIndex> <CoreIndex>]} <NumReads>
MEMDUMP {[THIS] | [<ProbeIndex> <CoreIndex>]} <Byte Address> <Length>
MEMLOAD {[THIS] | [<ProbeIndex> <CoreIndex>]} <FileName> <Byte Address> <Length
 Limit>  Loads binary file data to memory
MEMSAVE {[THIS] | [<ProbeIndex> <CoreIndex>]} <FileName> <Byte Address> <Length>
 Saves memory to binary file
PRINT "TEXT"[;[~]Variable | Constant]: Print statement. Prints quoted text
 and/or value of an internal variable (a%% - z%%), or constant integer
 expression in decimal, or hexadecimal[~] format
TIME: Returns an incrementing centisecond count from the host
TIMEMS: Returns an incrementing millisecond count from the host
WAIT <msec>: Wait for the number of milliseconds before proceeding
LIST: Lists a loaded script
NEW: Erases a loaded script from memory
RENUMBER <Delta>: Renumber script lines with Delta increment (default is 10)
LOAD <"FILENAME">: Loads a script from the current, absolute, or relative
 directory
SAVE <"FILENAME">: Saves a script to the current, absolute, or relative
 directory
```

Probe related functions

```
PROBELIST: Enumerates and returns an indexed list of known probe types
PROBENUM: Returns the number of probes attached
PROBEOPENBYINDEX <ProbeIndex> [<"FILENAME">]: Opens the probe associated with
 ProbeIndex
FILENAME is text of <key = value> pairs used for internal configuration
PROBEOPENBYSERIAL <"SerialNumber">: Opens the probe associated with SerialNumber
PROBECLOSEBYINDEX <ProbeIndex>: Closes the probe associated with ProbeIndex
PROBECLOSEBYSERIAL <"SerialNumber">: Closes the probe associated with
 SerialNumber
PROBEFIRSTFOUND: Returns the THIS ProbeIndex or index of the first probe in the
 enumerated list
PROBETIME <ProbeIndex>: Returns elapsed time from firmware boot, if supported
PROBESTATUS [<ProbeIndex>]: Returns an indexed list summary of the status of the
 probes connected to the system
PROBEVERSION <ProbeIndex>: Returns version information about probe firmware
PROBEDAPINFO <ProbeIndex>: Returns CMSIS-DAP probe information
PROBEISOPEN <ProbeIndex>: Returns TRUE or FALSE
PROBEHASJTAG <ProbeIndex>: Returns TRUE or FALSE
PROBEHASSWD <ProbeIndex>: Returns TRUE or FALSE
PROBEHASSWV <ProbeIndex>: Returns TRUE or FALSE
PROBEHASETM <ProbeIndex>: Returns TRUE or FALSE
```

```
PROBERESET <ProbeIndex> <ResetType>: Resets the probe (use 1 for ISP reset)
```

Core/TAP related functions

```
CORECONFIG {[THIS] | [<ProbeIndex>]}: Queries the scan chain configuration
CORESCONFIGURED <ProbeIndex>: Returns TRUE or FALSE
APLIMIT {[THIS] | [<ProbeIndex>]}:  <APIndex>:  Limit the AP Query (set once)
APLIST  {[THIS] | [<ProbeIndex>]}: [<APlimit>]: Detailed list of APs
 connected to the specified probe. APLimit restricts queries to the AP index.
CORELIST {[THIS] | [<ProbeIndex>]}: [<APLimit>]: Detailed list of APs/Cores
 connected to the specified probe. APLimit restricts queries to the AP index.
COREREADID {[THIS] | [<ProbeIndex> <CoreIndex>]}: Returns the DpID
DEBUGMAILBOXREQ {[THIS] | [<ProbeIndex> <APIndex>]} <Request>: Debug Mailbox
 Request
```

Wire related functions

```
WIRESWDCONNECT {[THIS] | [<ProbeIndex>]}: Configures the wire for SWD and
 returns the DpID
WIREJTAGCONNECT {[THIS] | [<ProbeIndex>]}: Configures the wire for JTAG
WIREDISCONNECT {[THIS] | [<ProbeIndex>]}: Closes the wire connection (SWD/JTAG)
WIREISPRESET {[THIS] | [<ProbeIndex>]}: Resets an LPC part into the ISP
 bootloader
WIREBOOTCONFIGSET {[THIS] | [<ProbeIndex>]} <"DATA">: Stores boot configuration
 data
 that will be automatically applied during subsequent reset commands.
 DATA is a string with up to 4 characters describing how each ISP_CTRL[3..0] pin
 should be handled: '0' (= drive low), '1' (= drive high), 'x' (= do not drive)
WIREBOOTCONFIGGET {[THIS] | [<ProbeIndex>]}: Returns previously stored
 configuration data
WIREBOOTCONFIGREAD {[THIS] | [<ProbeIndex>]}: Returns the current state of
 ISP_CTRL[3:0] pins
WIREBOOTCONFIGAPPLY {[THIS] | [<ProbeIndex>]} <1/0>: Immediately starts/stops
 driving
 the ISP_CTRL pins based on previously stored boot configuration data
WIRETIMEDRESET <ProbeIndex> <ms>: Asserts (Low) reset for ms milliseconds and
 returns the end state of the wire
WIREHOLDRESET <ProbeIndex> <State>: Asserts/Releases (Low/High) reset and
 returns the end state of the wire
WIRESETSPEED <ProbeIndex> <Hz>: Requests a particular wire speed in Hz
WIREGETSPEED <ProbeIndex>: Returns the current wire speed
WIRESETIDLECYCLES <ProbeIndex> <Cycles>: Sets the number of idle cycles between
 debug transactions
WIREGETIDLECYCLES <ProbeIndex>: Returns the current number of debug idle cycles
WIREISCONNECTED <ProbeIndex>: Returns TRUE or FALSE if WIRESWDCONNECT or
WIREJTAGCONNECT is complete
WIREGETPROTOCOL <ProbeIndex>: Returns SWD or JTAG
SELECTPROBECORE <ProbeIndex> <CoreIndex> : Sets the THIS parameter Probe/Core
 pair
THIS: Displays the current Probe, Core pair
```

Cortex-M related functions

```
CMINITAPDP {[THIS] | [<ProbeIndex> <CoreIndex>]}: Initialize a CMx core ready
 for debug connections
CMUNINITAPDP {[THIS] | [<ProbeIndex> <CoreIndex>]}: UnInitialize a CMx core
 (de-assert debug and system power-up)
CMWRITEDP {[THIS] | [<ProbeIndex> <CoreIndex>]} <REG> <DATA>: Returns zero on
 success
```

```
CMWRITEAP {[THIS] | [<ProbeIndex> <CoreIndex>]} <REG> <DATA>: Returns zero on
  success
CMREADDP {[THIS] | [<ProbeIndex> <CoreIndex>]} <REG>: Returns data
CMREADAP {[THIS] | [<ProbeIndex> <CoreIndex>]} <REG>: Returns data (handles
RDBUF on AP reads)
CMCLEARERRORS {[THIS] | [<ProbeIndex> <CoreIndex>]}
CMHALT {[THIS] | [<ProbeIndex> <CoreIndex>]}
CMRUN {[THIS] | [<ProbeIndex> <CoreIndex>]}
CMSTEP {[THIS] | [<ProbeIndex> <CoreIndex>]}
CMREGS {[THIS] | [<ProbeIndex> <CoreIndex>]}
CMDEBUGSTATUS {[THIS] | [<ProbeIndex> <CoreIndex>]}
CMWRITEREG {[THIS] | [<ProbeIndex> <CoreIndex>]} <RegNumber> <Value>
CMREADREG {[THIS] | [<ProbeIndex> <CoreIndex>]} <RegNumber>
CMWATCHLIST {[THIS] | [<ProbeIndex> <CoreIndex>]}
CMWATCHSET {[THIS] | [<ProbeIndex> <CoreIndex>]} <DWTIndex> <Address> [<RW|R|W>]
CMWATCHCLEAR {[THIS] | [<ProbeIndex> <CoreIndex>]} <DWTIndex>
CMBREAKLIST {[THIS] | [<ProbeIndex> <CoreIndex>]}: List the FPB breakpoints
CMBREAKSET {[THIS] | [<ProbeIndex> <CoreIndex>]} <Address>: Set an FPB
CMBREAKCLEAR {[THIS] | [<ProbeIndex> <CoreIndex>]} [<Address>]: Clear an FPB
CMSYSRESETREQ {[THIS] | [<ProbeIndex> <CoreIndex>]}: System reset request
CMVECTRESETREQ {[THIS] | [<ProbeIndex> <CoreIndex>]}: Core reset request
CMRESETVECTORCATCHSET {[THIS] | [<ProbeIndex> <CoreIndex>]}: Enable reset
  vector catch
CMRESETVECTORCATCHCLEAR {[THIS] | [<ProbeIndex> <CoreIndex>]}: Disable reset
  vector catch
```

Miscellaneous

```
HELP: display help on LinkServer commands
VERSION: returns the LinkServer version
CONNECTIONS: display active connections
```

Scripts can be specified within a LinkServer launch configuration to be run before a connection and/or before a reset.

22.6 RAM projects with LinkServer

MCUs have well-defined boot strategies from reset, typically they first run the internal manufacturer boot ROM code to perform some hardware setup and then pass control to code in flash (that is, the user's Application).

Most examples and wizards create projects to run from MCU flash memory but on occasion, it can be useful to debug code directly from RAM. There are two stages to such a task:

1. Modify a project so that it links to run from RAM
2. Modify the default reset mechanism to ensure that the RAM image is executed

To build a project to link against RAM, simply delete any flash entries within the memory configuration of the project. If the MCUXpresso IDE default linker settings are used, then the project links against the first RAM block in the list (provided no Flash entry is present). Alternatively, from:

Project Properties -> C/C++ Build -> Settings -> MCU Linker -> Manager Linker Script, you can check the entry *Link application to RAM*.

Note: if the project has already been built to link to flash, then it should be cleaned before being rebuilt.

Since an MCU does not automatically boot from RAM, a scheme is needed to take control of the reset mechanism of the debugger. This can be achieved via the use of a **SOFT** reset type. LinkServer launch configurations can take an additional option, add the line *--reset soft* to override the default reset type. Or preferably, set the reset type to *'SOFT'* as shown below.
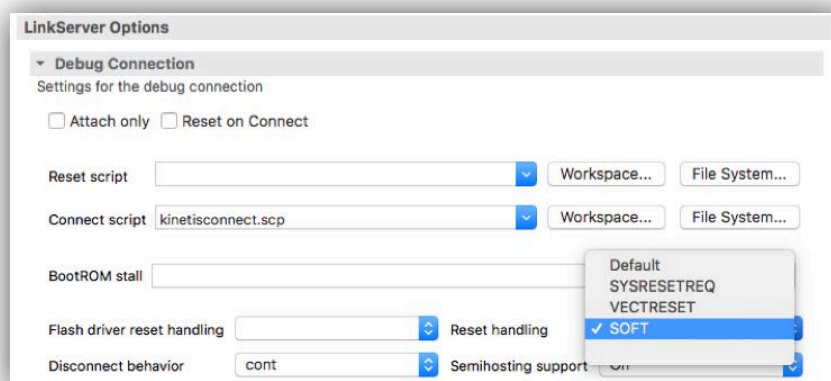
Figure 253. LinkServer SOFT Reset option

A soft reset is performed by setting the PC to the images resetISR() address, the stack pointer to the top of the first RAM region, and VTOR (Vector Table Offset Register) to the base address of the first RAM region.

Note: Typically, MCU RAM sizes are smaller than Flash sizes, therefore such a scheme may not be suitable for larger images.

22.6.1 Advantages of developing with RAM projects

There are a number of advantages when debugging from RAM:

- Breakpoints in RAM do not require dedicated HW resources, essentially there is no limit to the number of breakpoints that can be set.
- Flash programming step is not required, so the build and debug cycle are faster.
- Development of secondary bootloaders is free from BootROM considerations.
- No risk of accidentally triggering Flash security features.
- No requirement to understand or have flash programming capability allowing code (including flash drivers) can be developed.
- Any flash contents are preserved while debugging.
- Unit development of large applications.

Note: It should be remembered that since the MCU does not undergo a true hardware reset, peripheral configurations are inherited from one debug session to the next.

22.7 The Console view

The Console view contains a number of different consoles providing textual information about the operation of various parts of MCUXpresso IDE. It is located by default in the bottom right of the Debug Perspective, in parallel with a number of other views – including the 'Installed SDKs' view.

The actual consoles available within the Console view depend upon what operations are currently taking place – in particular a number of consoles only becomes available once a debug session is started.

The currently displayed console provides a local toolbar, with icons to do things like copying the contents of the console or clearing its contents.

To see the list of currently available consoles, and, if required, change to a different one:

1. Switch to the Console View

2. Using the toolbar within the Console View click the drop-down arrow next to the **Display Selected Console** icon (which looks like a small monitor)
3. Select the required console from the dropdown list
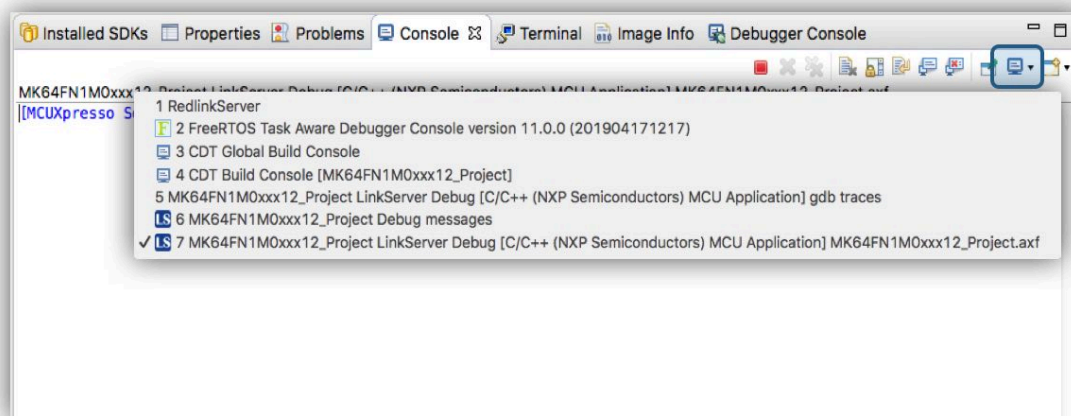
Figure 254. Select Console

22.7.1 Console types

Consoles you can typically see include the following...

22.7.1.1 Build Console and Global Build Console

The Build Console (sometimes referred to as the *Build Log*) is used by the MCUXpresso IDE build tools (compiler, linker, and so on) to display output generated when building your project. In fact, MCUXpresso IDE has two build consoles - one of which records the output from building the current project, and the second a global build console, which records the output from building all projects.

By default, the number of lines stored in the Build Console is limited to 500 lines. You can increase this to any reasonable number as follows:

1. Select the **Windows->Preferences** menu option
2. Now, choose **C/C++ -> Build -> Console**
3. Increase the **"Limit Console out (number of lines)"** to a larger number, for instance 5000.

Note: This setting, like most within the MCUXpresso IDE is saved as part of your workspace. Therefore, you have to make this change each time you create a new workspace.

Other options that can be set in Preferences include whether the console is cleared before a build, whether it should be opened when a build starts, and whether to bring the console to the top when building.

Once your build has been completed, if you have any build errors displayed in the console, clicking them causes, by default, the appropriate source file to be opened at the appropriate place for you to fix the error.

22.7.1.2 FreeRTOS task-aware debugger console

This console displays status about the FreeRTOS TAD views. Enablement and persistence of the logs can be controlled via the Preferences page. For more details, see the *MCUXpresso IDE FreeRTOS Debug Guide*.

22.7.1.3 Azure RTOS ThreadX task-aware debugger console

This console displays status about the Azure RTOS ThreadX TAD views. Enablement and persistence of the logs can be controlled via the Preferences page. For more details, see the *MCUXpresso IDE Azure RTOS ThreadX Debug Guide*.

22.7.1.4 Zephyr RTOS task-aware debugger console

This console displays status about the Zephyr RTOS TAD views. Enablement and persistence of the logs can be controlled via the Preferences page. For more details, see the *MCUXpresso IDE Zephyr RTOS Debug Guide*.

22.7.1.5 gdb traces and arm-none-eabi-gdb consoles

These consoles give access to the GDB command line debugger that sits underneath the graphical debugging front end of MCUXpresso IDE.

22.7.1.6 RedlinkServer/LinkServer console

This console gives access to the server application that sits at the bottom of the debug stack when using a debug probe connected via the MCUXpresso IDEs native "LinkServer" debugging mechanism. LinkServer commands can be entered from this console.

22.7.1.7 Debug messages console

The Debug Messages console (sometimes referred to as the *Debug Log*) is used by the debug driver to display additional information that may help understand connection issues when debugging your target MCU.

22.7.1.8 Semihosting console

This console, generally displayed with *.axf*, allows semihosted output from the application running on the MCU target to be displayed, and potentially for input to be sent down to the target.

22.7.1.9 SWO and Trace console

This console displays all information related to CoreSight components creation and configuration, starting with the base address of the ROM table that is being scanned for the purpose of CoreSight identification. For more information, refer to *MCUXpresso_IDE_SWO_Trace_Guide.pdf* documentation.

22.7.2 Copying the contents of a console

Occasionally, you may wish to copy out the contents of a console. For instance, the MCUXpresso IDE support team may ask you to provide the details of your Build Console in a forum thread. To do this:

1. Clean, then build your project.
2. Select the appropriate Build Console as above:
3. Select the contents (for example, Ctrl-A)
4. Copy to the clipboard (for example, Ctrl-C).
5. Paste from clipboard into forum thread (for example, Ctrl-V). If there is a large amount of text in the build console, it is advisable to paste it into a text file, which can be ZIPed if appropriate.

Note: some consoles provide a button in their local toolbar to copy or save their contents.

22.7.3 Relocating and duplicating the Console view

By default, the Console view is positioned in parallel with some other views. This can mean that if a console is being regularly updated with new output (for instance the view displaying semihosted output from the application running on the target MCU), then by default this may cause the console to keep jumping to the foreground – therefore hiding other views that you are using (for instance one of the SWO Trace views).

To avoid this, you may wish to relocate the Console. To do this:

1. Click and hold down on the Console view
2. Continue to hold down, and drag the cursor to the location where you want the Console view to be displayed
3. Then release the mouse click, and the Console view is placed at the required position
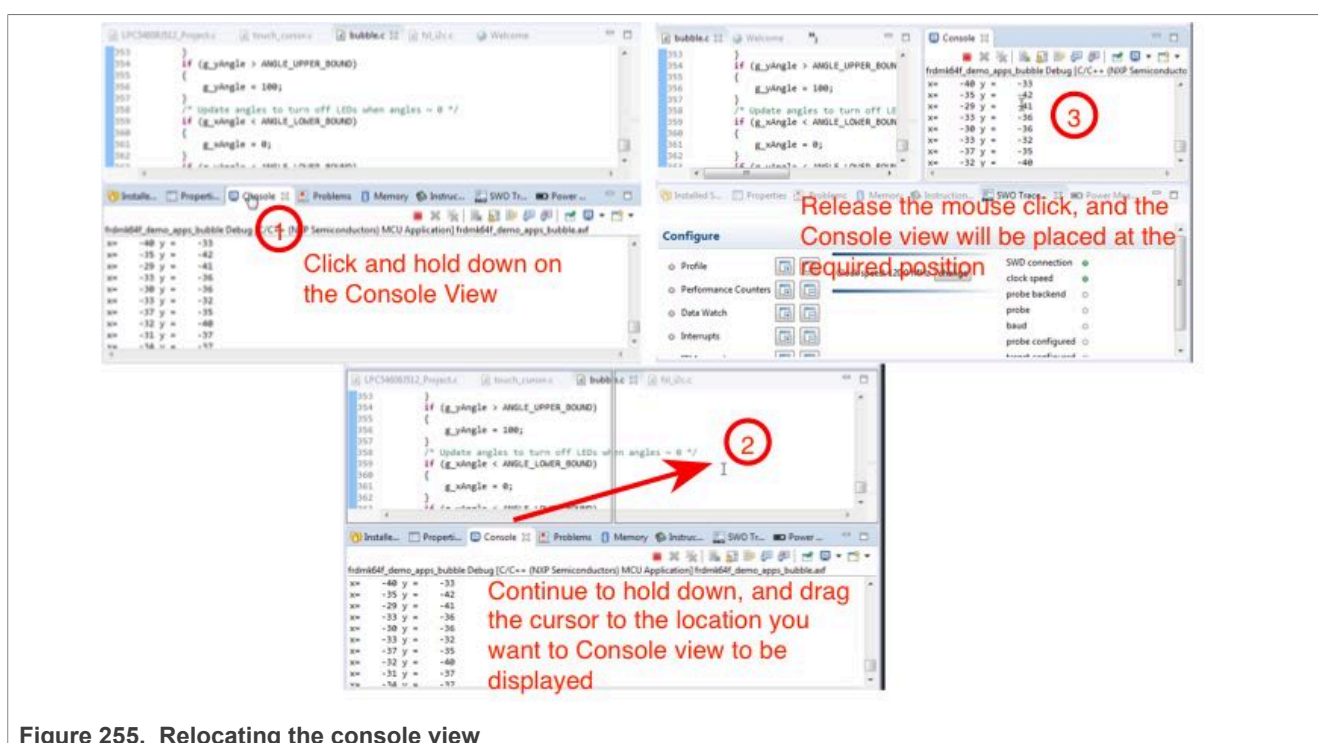
Figure 255. Relocating the console view

Another alternative is to spawn a duplicate instance of the Console view. This allows multiple consoles to be visible at the same time. To do this use the Open Console button on the Console view local toolbar.
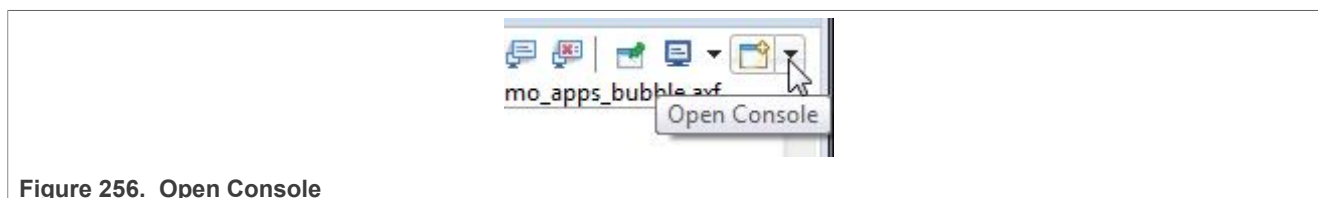
Figure 256. Open Console

and then select "New Console View"
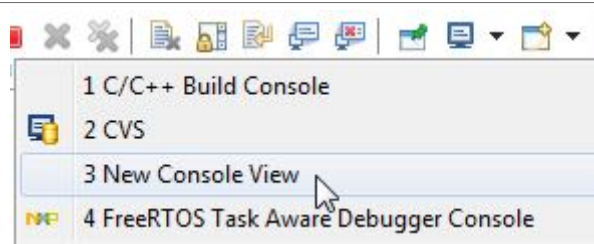
Figure 257. New Console View

This then displays a second console view, which can be dragged and dropped to a new location within the Perspective, as shown for the single Console view case described above.
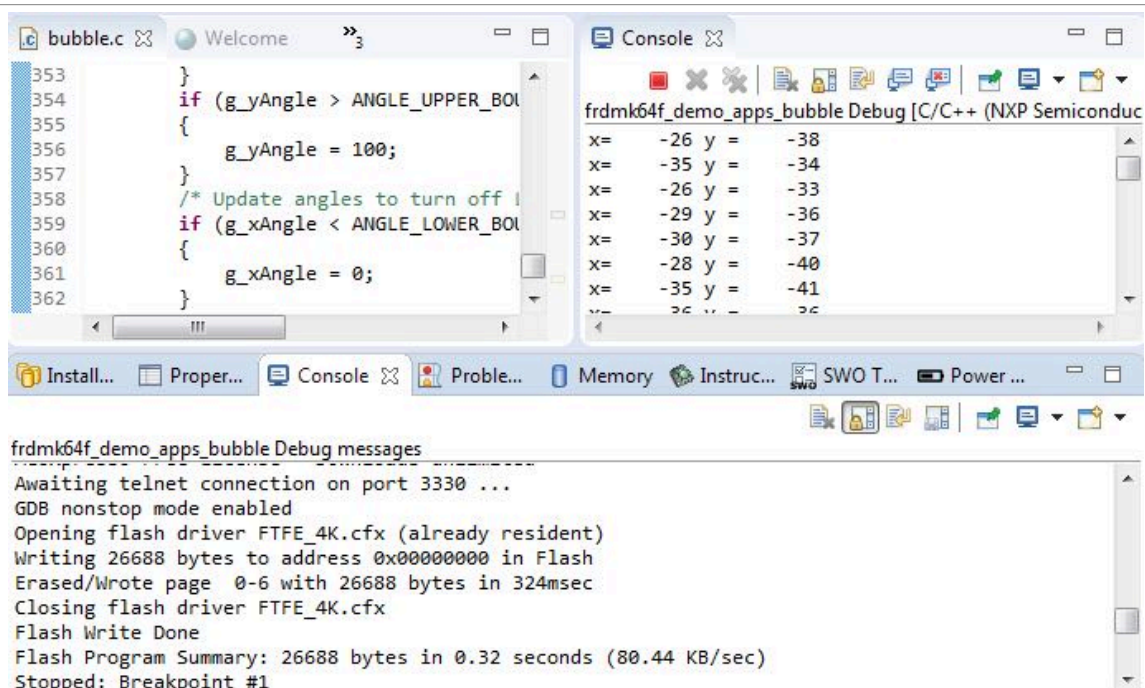
Figure 258. Duplicating the Console view

Having opened a second console view, select which console you want displayed in it, and then use the "Pin Console" button to ensure that it does not switch to one of the other consoles when output is displayed.

Figure 259. Pin Console

22.8 Using Terminal view for UART communication with a target

MCUXpresso IDE provides a Terminal View, which can be used to display UART (serial) input/output between a host PC and the target MCU. In situations where a debug probe is built into the target board, UART comms are often possible via a VCOM connection over the same USB cable as the debug connection. However, where this is not the case a serial_to_USB cable can be used, alternatively, if the target MCU has a built-in USB then a VCOM port can implemented in the application code running on the target MCU.

Using a Terminal View offers an alternative way of interacting with the target when compared to semihosting output via the debug channel (which is displayed in the Console View). There are pros and cons to both approaches, but one distinct advantage to using the Terminal View for serial output is that you can interact with the target MCU without a debug session being active!

To use the Terminal View within MCUXpresso IDE, the first thing you have to do is open it (as it is not visible by default). To do this go to: *Window -> Show View -> Other* and select Terminal.
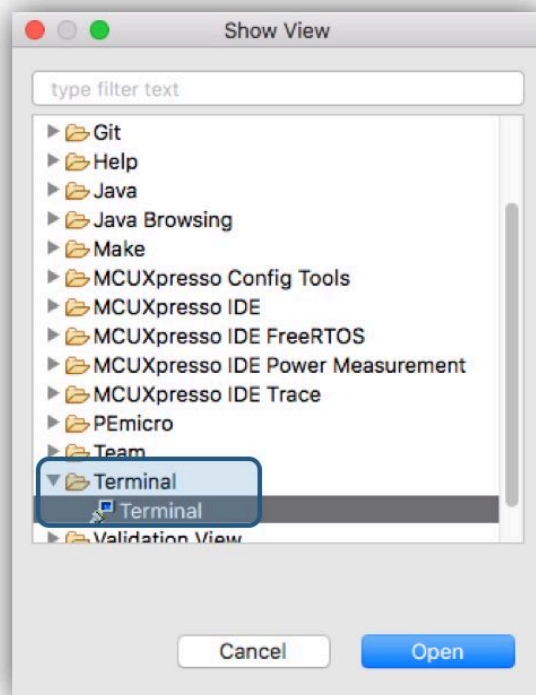
Figure 260. Selecting Terminal view

Alternatively, just type "Terminal" into the "Quick Access" button in the top right of the window of the IDE.

Next, ensuring that the serial connection between your PC and the target MCU is active first, click the "Open a Terminal" button in the toolbar of the Terminal View:
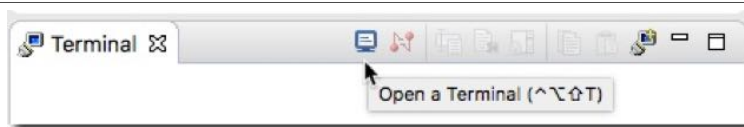
Figure 261. Open a Terminal

Note: If using the LPC-Link2 built into many LPCXpresso boards, then you have to make sure the probe has been booted before the serial connection can be available. You can do this manually by using the "Boot Debug probe" button in the toolbar toward the top of the IDE window. Or else you can pre-program the probe firmware into flash using LPCScrypt.

Now, select the type of terminal required - a serial one:
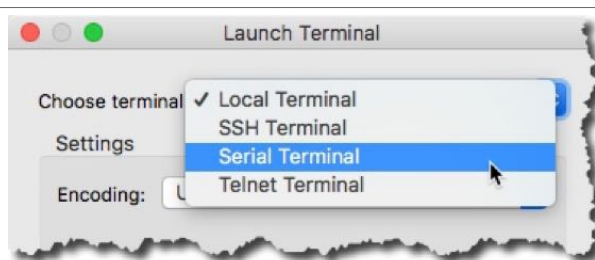
Figure 262. Terminal types

And then select the appropriate settings:
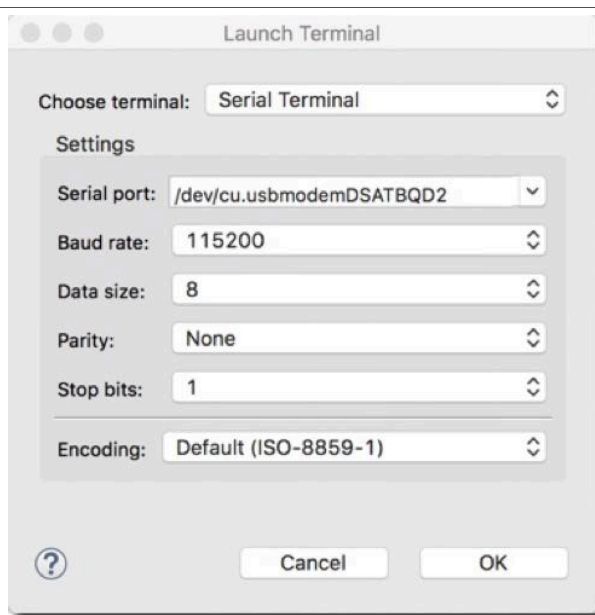
Figure 263. Terminal settings

Note: that if you are receiving serial output via USB (for instance over a VCOM port from the debug probe), then the default settings should normally be fine. The one setting you do have to get correct is the Serial port to use. This varies depending on what devices are connected to your PC, what OS you are running, and what the source for your serial port is.

For instance, if you are running on Windows, then the simplest way to identify the required serial port is to open "Device Manager" (typically via the "Start Menu"), and then expand the "Ports" tab. This should allow you to identify the appropriate COM port needed.

After configuring the settings as required, click the "OK" button. You should now see serial output from the application running on the target MCU within the Terminal View:
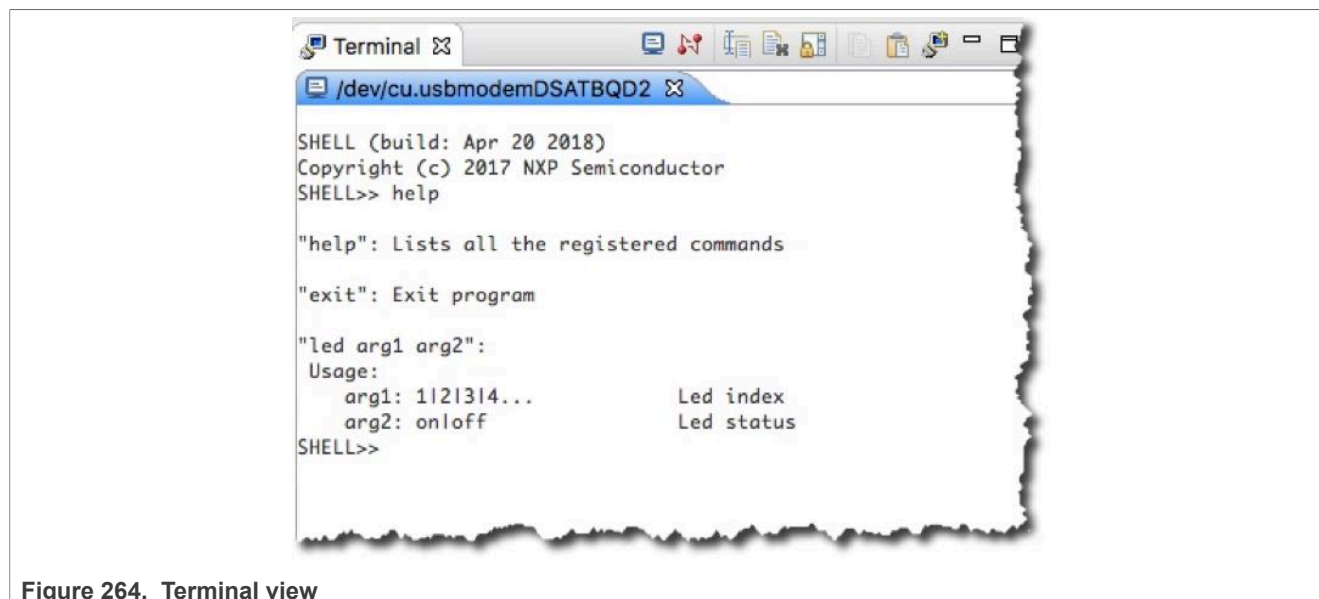
Figure 264. Terminal view

Note: The Terminal view only offers a simple terminal mechanism with a few configuration options. If you require more control over the way the terminal behaves, you may still have to use a standalone terminal application, such as PuTTY, CoolTerm, or Tera Term.

22.9 Using and troubleshooting LPC-Link2

22.9.1 LPC-Link2 hardware

LPC-Link2 is a powerful, low-cost debug probe design from NXP Semiconductors based on the LPC43xx MCU. It has been implemented into a number of different systems, including:

• The standalone LPC-Link2 debug probe
• The debug probe built into the range of LPCXpresso V2/V3 boards

For more details, see https://www.nxp.com/lpcxpresso-boards.

22.9.2 Softloaded vs pre-programmed probe firmware

One thing that most LPC-Link2 implementations offer is the ability to either softload the debug probe firmware (using USB DFU functionality) or to have the debug probe firmware pre-programmed into flash.

Programming the firmware into flash has some advantages, including:

• Allows the use of the LPC-Link2 with toolchains that, unlike MCUXpresso IDE, do not support softloading of the probe firmware
• Better supports the use of LPC-Link2 as a small production run programmer
• Allows the LPC-Link2 to be used with SEGGER J-Link firmware as an alternative to the normal CMSIS-DAP firmware. For more details, visit https://www.segger.com
• Avoids issues that the re-enumeration of the LPC-Link2 can sometimes trigger as the firmware softloads (particularly where virtual machines are in use)

The recommended way to program the firmware into the flash of LPC-Link2 is NXP's LPCScrypt flash programming tool. For more details, see https://www.nxp.com/LPCSCRYPT.

However, when used with MCUXpresso IDE, softloading the probe firmware is the recommended method of using LPC-Link2 in most circumstances.

This ensures that the firmware version matching the MCUXpresso IDE version can automatically be loaded when the first debug session is started (so normally the latest version). It also allows different probe firmware variants to be softloaded, depending on current user requirements.

For this to work, you have to make sure that the probe hardware is configured to allow DFU booting. To do this:

• For standalone LPC-Link2: remove the link from header JP1 (nearest USB)
• For LPCXpresso V2/V3: add a link to the header "DFU link"

22.9.3 LPC-Link2 firmware variants

As well as providing debug probe functionality, NXP's CMSIS-DAP firmware for LPC-Link2 by default also includes bridge channels to provide:

• Support for SWO Trace capture from the MCUXpresso IDE
• Support for Power Measurement from the MCUXpresso IDE (certain LPCXpresso V3 boards only)
• Support for a UART VCOM port connected to the target processor (LPCXpresso V2/V3 boards only)
• Support for an LPCSIO bridge that provides communication to I2C and SPI slave devices (LPCXpresso V3 boards only)

However, two other variants of the CMSIS-DAP firmware are provided that remove some of these bridge channels.

• **"Non Bridged":** This version of firmware provides debug features only – removing the bridged channels such as trace, power measurement, and VCOM. By removing the requirement for these channels, USB bandwidth is reduced, therefore this firmware may be preferable if multiple debug probes are to be used concurrently. The non-bridged build also provides an increase in download and general debug performance.
• **"VCOM Only":** This version of firmware provides only debug and VCOM features. The removal of the other bridges allows better VCOM performance (though generally, the bridged firmware provides more than good enough VCOM performance).

A particular workspace can be switched to softload a different firmware variant via: **Preferences -> MCUXpresso IDE -> Debug Options -> LinkServer Options -> LPC-Link2 boot type**.
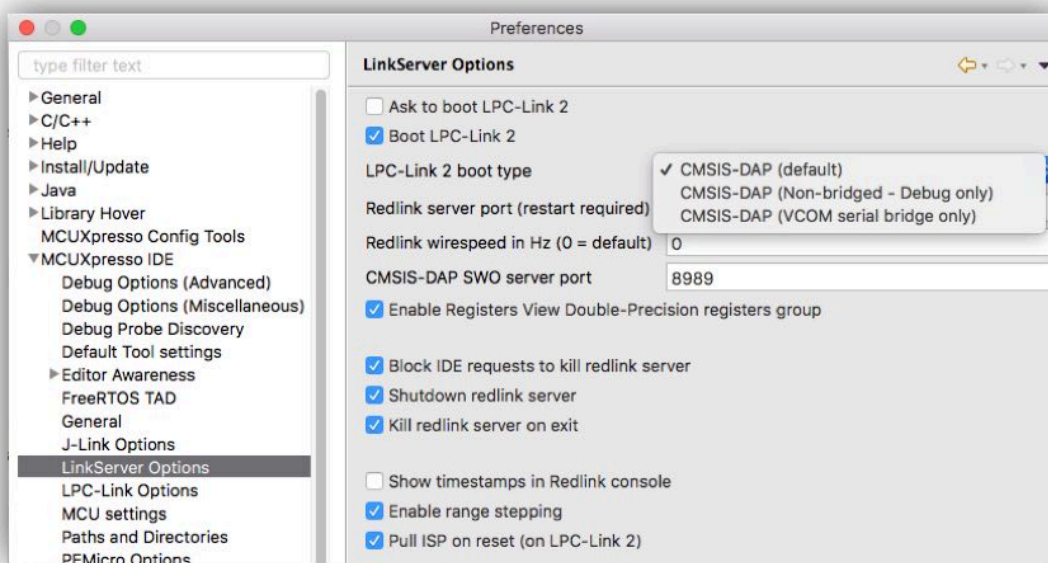
Figure 265. LPC-Link2 boot type selection

Note: If a mix of bridged and unbridged debug probes is required, then it is recommended that these probes are pre-programmed with the required debug firmware. This can easily be done via LPCScrypt.

22.9.4 Manually booting LPC-Link2

The recommended way to use LPC-Link2 with the MCUXpresso IDE is to allow the GUI to boot and softload a debug firmware image at the start of a debug session.

Normally, LPC-Link2 is booted automatically (when configured to operate in DFU mode) however, under certain circumstances - such as when troubleshooting issues, or using the LinkServer command line flash utility, you may need to boot it manually.

22.9.4.1 LPC-Link2 USB details

The standard utilities to explore USB devices on MCUXpresso IDE-supported host platforms are:

- Windows - Device Manager
  - **–** MCUXpressoIDE also provides a listusb utility in:
    - – *install_dir/ide/binaries/Scripts*
- Linux - terminal command: lsusb
- Mac OS X - terminal command: system_profiler SPUSBDataType

Before boot, LPC-Link2 appears as a USB device with details:

```
Device VendorID/ProductID:    0x1FC9/0x000C    (NXP Semiconductors)
```

and appears in Windows -> Devices and Printers, as below:
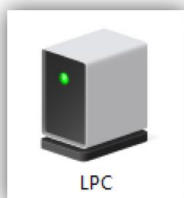
After boot, LPC-Link2 appears by default as a USB device with details:

```
Device VendorID/ProductID:    0x1FC9/0x0090
```

and appears in Windows -> Devices and Printers similar to below:
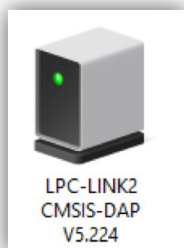
Note: Text details vary depending on version number and which probe firmware variant is booted.

22.9.4.2 Booting from the command line

MCUXpresso IDE provides a boot script for all supported platforms. To use this script first of all connect the LPC-Link2 to your PC then enter the commands into a DOS command prompt (or equivalent):

```
cd <install_dir>\ide\LinkServer\binaries
boot_link2
```

This invokes the dfu-util utility to download the probe firmware into the RAM of the LPC43xx MCU of LPC-Link2 and then re-enumerate the probe.

22.9.4.3 Booting from the GUI

It is also possible to boot LPC-Link2 manually from the MCUXpresso IDE GUI, which may be a more convenient solution than using the command line. To do this, first of all, connect the LPC-Link2 to your PC, then locate the red Boot icon on the toolbar:
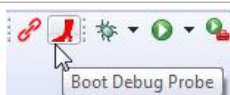
Figure 266. Boot Debug Probe

and then click OK in the dialog displayed:
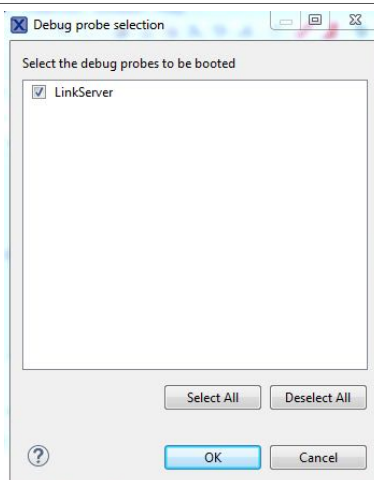
Figure 267. Debug probe selection

22.9.5 LPC-Link2 Windows drivers

The drivers for LPC-Link2 are installed as part of the main MCUXpresso IDE installation process.

Note: One thing to be aware of is that the first time you debug using a particular LPC-Link2 on a particular PC, the drivers have to be loaded. This first time can take a variable period of time depending upon your PC and operating system version. This may mean that the first debug attempt fails, as the IDE may time out waiting for the booted LPC-Link2 to appear. In such a case, a second debug attempt should complete successfully. Otherwise, try booting the LPC-Link2 manually and checking the drivers load correctly.

If you have to reinstall the drivers, then the installer can be found at:

```
C:\nxp\<linkserver_install_dir>\drivers\lpc_driver_installer.exe
```

22.9.6 LPC-Link2 failing to enumerate

On some systems, after booting LPC-Link2 with CMSIS-DAP firmware, the booted debug probe does not enumerate correctly and the MCUXpresso IDE (or other toolchain) is unable to see the debug probe. This problem is normally caused by an old, obsolete, version of the VCOM driver being found by Windows instead of the correct driver. To see if this is the cause of a problem on your computer, find the version number of the LPC-Link2 VCOM driver. The obsolete driver version is 1.0.0.0.

22.9.6.1 To find the version number of the LPC-Link2 VCOM driver

If you are using a soft-booted LPC-Link2 debug probe, start by booting your LPC-Link2, as described in Manually booting LPC-Link2 If your LPC-Link2 debug probe is booting from an image preprogrammed into the flash, you can skip this step.

Once your LPC-Link2 has booted, find the device in Device Manager and look at the driver version number.

- Open the Windows Device Manager
- Expand the "Ports (COM and LPT)" section
- Right-click "LPC-LinkII UCom Port", and select Properties
- Click the Driver tab of the Properties dialog
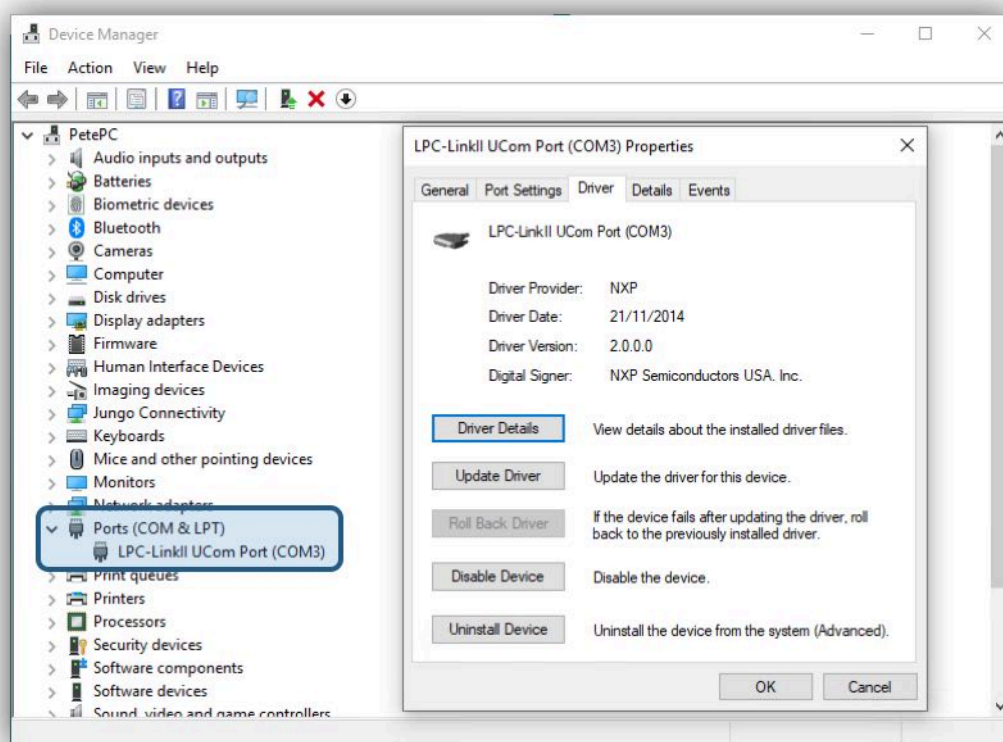
Figure 268. Device manager

Note: This image shows the current correct version of the driver (2.0.0.0).

22.9.6.2 Removing the obsolete 1.0.0.0 LPC-LinkII UCOM driver

To remove the obsolete driver, perform the following actions:

1. In Device Manager, right-click the LPC-LinkII UCOM device and select Uninstall
2. If there is an option to delete the driver software, make sure it is checked, and press OK
3. Select the menu item Action->Scan for hardware changes
4. In the Windows Control Panel, select Add/Remove program or Uninstall a program option
5. Find the LPC Driver Installer and right-click choose Uninstall
6. Let the uninstaller complete
7. Switch back to the Device Manager and Scan for hardware changes again
8. If the LPC-LinkII UCOM driver version is still present, Uninstall it again (steps 1 to 3) and repeat until the LPC-LinkII UCOM driver no longer appears
9. Now, run the lpc_driver_installer.exe found in the MCUXpresso IDE "Drivers" directory

Note: A reboot is recommended after running the lpc_driver_installer.exe installer.

Now, manually reboot the probe again (if softloading) and check **Windows - Devices and Printers** to see if the device now appears correctly as an LPC-Link2 CMSIS-DAP Vx.xxx.

If this fails to correct the problem, there is one final thing to try:

• Open a Command Prompt as the Administrative user and run the following commands

```
cd %temp%
pnputil -e >devices.txt
notepad devices.txt
```

• Search devices.txt for an entry similar to this, and note down the Published name (oemXX.inf)

```
Published name :          oem38.inf
Driver package provider :  NXP
Class :                    Ports (COM & LPT)
Driver date and version :  09/12/2013 1.0.0.0
Signer name :              NXP Semiconductors USA. Inc.
```

• Using the name notes above, run the following command (replacing XX with the number found above)

```
pnputil -f -d oemXX.inf
```

22.9.7 Troubleshooting LPC-Link2

If you have been able to use LPC-Link2 in a debug session, but now see issues such as "No compatible emulator available" or "Priority 0 connection to this core already taken" when trying to perform a debug operation ...

• Ensure you have shut down any previous debug session
  – Close a debug session (press the Red 'terminate' button) before starting another debug session
• It is possible that the debug driver is still running in the background. Use the task manager or equivalent to kill any tasks called:
  – redlinkserv
  – arm-none-eabi_gdb*
  – crt_emu_*

MCUXpresso IDE provides an IDE button 🔲 to kill all low-level debug executables.

A failure occurring while initiating a debug connection might also be caused by the GDB client being unable to communicate with the GDB stub. In this case, the error usually indicates a networking-related error, such as "Connection timed out". The firewall and the launch configuration should be checked. The stub has to be

listening on a networking port to communicate with the GDB client. For more details about the Debug Server Connection parameters, see also Editing a launch configuration (LinkServer) section.

If your host has never worked with LPC-Link2, then the following may help to identify the problem:

- Try manually booting your LPC-Link2 as per Manually booting LPC-Link2 and ensure that the drivers have been installed correctly.
- Try a different USB cable!
- Try a different USB port. If your host has USB3 and USB2, then try a USB2 port.
  - There are known issues with motherboard USB3 firmware. Ensure that your host is using the latest driver from the manufacturer. **Note**: this is not referencing the host OS driver but the motherboard firmware of the USB port
- If using a USB hub, first try a direct connection to the host computer.
- If using a USB hub, try using one with a separate power supply - rather than relying on the supply over USB from your PC.
- Try completely removing and reinstalling the host device driver. See also LPC-Link2 fails to enumerate above.
- If using Windows 8.1 or later, then sometimes the Windows USB power settings can cause problems. For more details, use your favorite search engine to search for "windows usb power settings" or similar.

22.10 Using and troubleshooting MCU-Link

22.10.1 MCU-Link hardware

MCU-Link is a new powerful and cost-effective debug probe architecture that can be used seamlessly with MCUXpresso IDE and is also compatible with third-party IDEs that support CMSIS-DAP protocol.

There is a range of debug solutions based on the MCU-Link architecture, which include the standalone low-cost base model (MCU-Link probe), a fully featured MCU-Link Pro probe, and various implementations built into NXP evaluation boards. MCU-Link Pro includes many additional features to facilitate embedded software development, like energy consumption analysis and support of peripheral and host emulation via USB bridging functions. On-board implementations support all the base model features and can optionally support additional features available on MCU-Link Pro.

MCU-Link solutions are based on the powerful, low-power LPC55S69 microcontroller, and all versions run the same firmware from NXP.

MCU-Link common features:

- CMSIS-DAP firmware to support all NXP Arm Cortex®-M based MCUs with SWD or JTAG debug interfaces
- High-speed USB host interface
- USB to target UART bridge (VCOM)
- SWO profiling and I/O features

MCU-Link Pro additional features:

- SEGGER J-Link firmware option
- Circuitry to measure the target's supply voltage and current drawn
- Trigger-based measurement
- Analog signal trace input
- A second USB to target UART bridge (VCOM)
- USB SPI and I2C bridges for programming/provisioning and host-based application development
- Option to power target system at up to 350 mA (at 1.8 V or 3.3 V)
- On-board, user-programmable LPC804 for peripheral emulation
- Multiple status LEDs for diagnosis of issues

• Target reset button

For more details, refer to the Getting Started guides available on the product web pages on nxp.com:

• MCU-Link debug probe: https://www.nxp.com/pages/:MCU-LINK
• MCU-Link Pro debug probe: https://www.nxp.com/pages/:MCU-LINK-PRO

22.10.2 MCU-Link CMSIS-DAP firmware

MCU-Link debug probes are factory-programmed with NXP's CMSIS-DAP protocol-based firmware, which also supports all other features supported in hardware.

Besides SWD/JTAG debug probe functionality, NXP's CMSIS-DAP firmware for MCU-Link by default also includes bridge channels to provide:

• Support for SWO Trace capture from the MCUXpresso IDE
• Support for a UART VCOM port (UART interface to target)
• Support for a second UART VCOM port [MCU-Link Pro]
• Support for USB serial I/O (SPI, I2C, GPIO) bridge compatible with LIBUSBSIO [MCU-Link Pro]
• Support for energy measurement from the MCUXpresso IDE [MCU-Link Pro]

MCU-Link probe type and firmware details are displayed in the Probes Discovered dialog of MCUXpresso IDE when the probe is attached.

Available attached probes

	Name	Serial number / ID / Nickname	Type	Manufacturer	IDE Debug Mode
LS	MCU-LINK Pro (r0CF) CMSIS-DAP V2.249	I2PDSK2HWMUQD	LinkServer	NXP Semiconductors	Non-Stop
LS	MCU-LINK on-board (r0C7) CMSIS-DAP V2.249	AAM01QI2CPGRH	LinkServer	NXP Semiconductors	Non-Stop

Figure 269. Available attached probes

Probes Discovered indicates if a newer firmware version is available. It is recommended to update the MCU-Link firmware to the latest version using the provided firmware update utility. Go to the LinkServer for Microcontrollers web page and navigate to "Software Development Tools". LinkServer installation packages for each host OS are shown. Download and run the installer for your host OS. Installing LinkServer also installs MCU-LINK_installer - the firmware update utility. A step-by-step installation guide is provided on the probe web page on nxp.com

22.10.2.1 CMSIS-DAP versions

Firmware versions V2.xxx implement an older version of CMSIS-DAP 1.1.0 and use USB HID as an interface to the host PC.

Firmware versions V3.xxx are based on the latest CMSIS-DAP version 2.1.0 and use WinUSB as an interface to the host PC and are therefore faster. Since the firmware implements Microsoft descriptors to declare WCID (Windows Compatible ID), no additional WinUSB driver is required on Windows.

Note: *Firmware versions 3.xxx are supported in MCUXpresso IDE 11.7.0 or newer. If using an older MCUXpresso IDE product, install MCU-Link Firmware version V2.263.*

22.10.2.2 MCU-Link USB details

The standard utilities to explore USB devices on MCUXpresso IDE-supported host platforms are:

- Windows – Device Manager
  - **–** MCUXpressoIDE also provides a listusb utility in:
    - – *install_dir/ide/LinkServer/binaries/Scripts*
- Linux – terminal command: lsusb
- macOS – terminal command: system_profiler SPUSBDataType

In ISP mode (firmware update enabled), MCU-Link appears as a USB device with details:

```
Device VendorID/ProductID:   0x1FC9/0x0021
```

In normal use, MCU-Link appears as a USB device with details:

```
Device VendorID/ProductID:   0x1FC9/0x0143
```

MCU-Link appears in Windows Control Panel -> Hardware and Sound -> Devices and Printers similar to below:
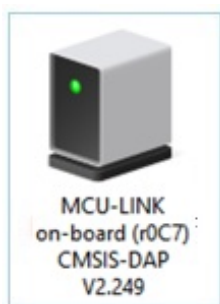
Note: Text details vary depending on firmware version and probe configuration.

22.10.3 MCU-Link host drivers

MCU-Link debug probes are supported on Windows 10, macOS, and Ubuntu Linux platforms. MCU-Link probes use standard OS drivers, however, on Windows, an inf driver is provided to allow displaying friendly names in Device Manager for the MCU-Link VCom Port(s). The driver for MCU-Link is installed as part of the MCUXpresso IDE installation process as well as during the installation of the firmware update utility. If you have to reinstall the driver, it can be found at:

```
MCUXpresso IDE: <install_dir>\LinkServer\drivers\MCU-Link
MCU-LINK_installer: <install_dir>\LinkServer\drivers
```

To install the driver, navigate to MCU-Link drivers and install the file by right-clicking –> Install:

- mcu-link-vcom.inf

22.10.4 MCU-Link JLink-compatible firmware

A custom version of J-Link firmware to make MCU-Link Pro compatible with SEGGER's popular J-Link LITE is also available, but note that this firmware is limited to supporting debug (including SWO) and VCOM features only.

22.10.5 Troubleshooting MCU-Link

If you have been able to use MCU-Link in a debug session, but now see issues such as "No compatible emulator available" or "Priority 0 connection to this core already taken" when trying to perform a debug operation ...

- Ensure you have shut down any previous debug session
  – Close a debug session (press the Red 'terminate' button) before starting another debug session
- It is possible that the debug driver is still running in the background. Use the task manager or equivalent to kill any tasks called:
  – redlinkserv
  – arm-none-eabi_gdb*
  – crt_emu_*

Use MCUXpresso IDE button to kill all low-level debug executables.

A failure occurring while initiating a debug connection might also be caused by the GDB client being unable to communicate with the GDB stub. In this case, the error usually indicates a networking-related error, such as "Connection timed out". The firewall and the launch configuration should be checked. The stub has to be listening on a networking port to communicate with the GDB client. For more details about the Debug Server Connection parameters, see also Editing a launch configuration (LinkServer) section.

If your host has never worked with MCU-Link, then the following may help to identify the problem:

- Try updating the MCU-Link firmware and ensure that the drivers have been installed correctly. See MCU-Link firmware above.
- Make sure it is plugged into a high-speed USB 2.0 port!
- Try a different USB cable!
- Try a different USB port. If your host has USB3 and USB2, then try a USB2 port.
  – There are known issues with motherboard USB3 firmware. Ensure that your host is using the latest driver from the manufacturer. **Note**: this is not referencing the host OS driver but the motherboard firmware of the USB port
- If using a USB hub, first try a direct connection to the host computer.
- If using a USB hub, try using one with a separate power supply – rather than relying on the supply over USB from your PC.
- Try completely removing and reinstalling the host device driver. See MCU-Link Windows drivers above.

22.11 Creating bin, hex, or S-Record files

When building a project, the MCUXpresso IDE tools create an ARM executable format (AXF) file - which is actually a standard ELF/DWARF file. This file can be programmed directly down to your target using the MCUXpresso IDE debug functionality, but it may also be converted into various formats suitable for use in other external tools.

22.11.1 Simple conversion within the IDE

The simplest way to create a one-off binary or hex file is to open up the Debug (or Release) folder in Project Explorer right-click the .axf file, and "**Binary Utilities -> Create binary**" (or Create hex, S-Record).
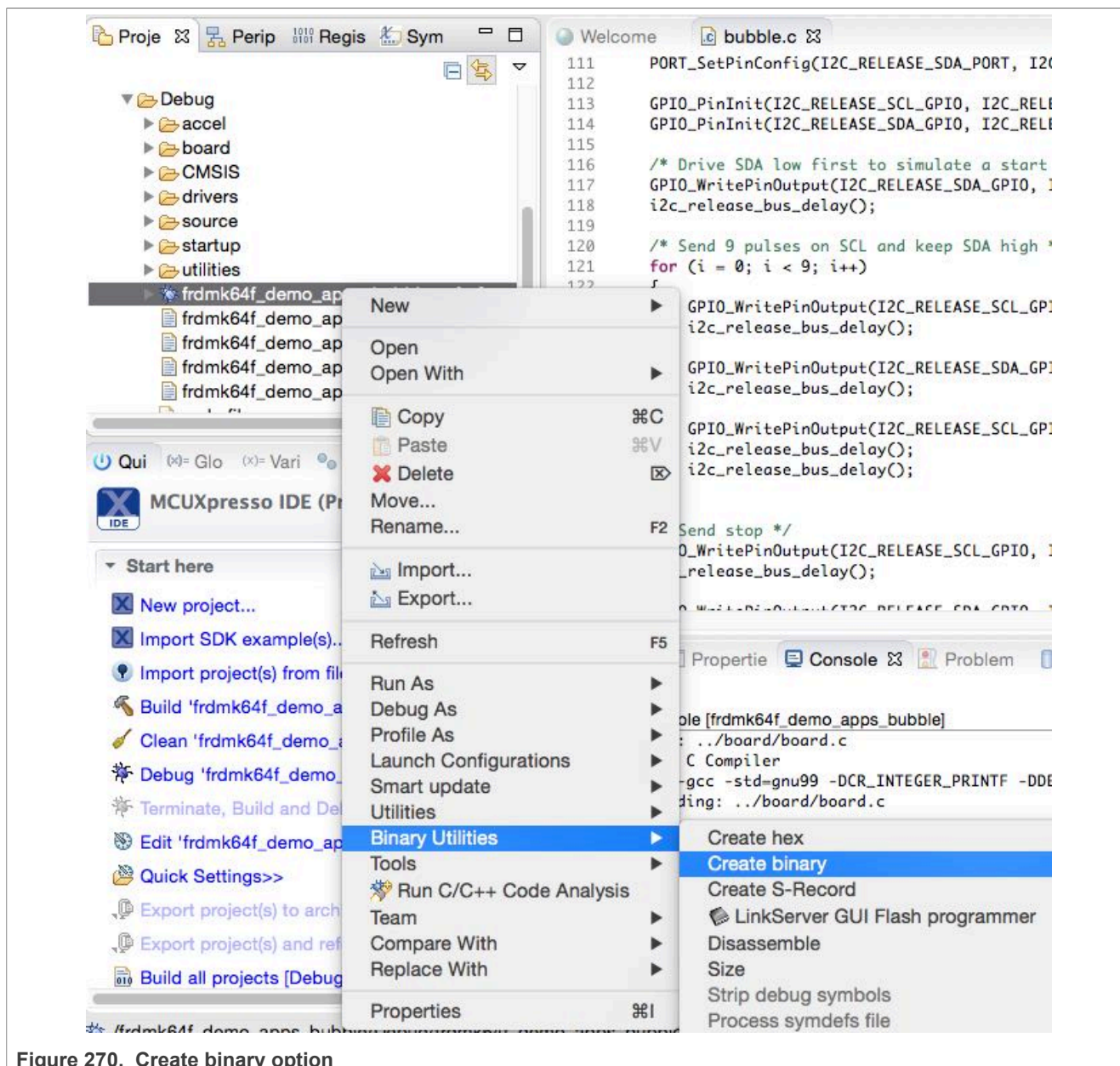
Figure 270. Create binary option

You can also change the underlying commands and options that are called by these menu entries from the "**Preferences -> MCUXpresso IDE -> Utilities**" preference page.

22.11.2 From the command line

The above "Binary Utilities" option within the IDE GUI simply invokes the command line objcopy tool (arm-none-eabi-objcopy). Objcopy can convert into the following formats:

- srec (Motorola S record format)
- binary
- ihex (Intel hex)
- tekhex

For example, to convert an example.axf into binary format, use the following command:

arm-none-eabi-objcopy -O binary example.axf example.bin

If you ctrl-click the project name on the right-hand side of the bottom bar of the IDE, this launches a command prompt in the project directory with appropriate tool paths set up. You can also use the Project Explorer right-click "Utilities->Open command prompt here" option to do this.

All you need to do before running the objcopy command is change into the directory of the required Build configuration.

22.11.3 Automatically converting the file during a build

Objcopy may be used to convert automatically an axf file during a build. To do this, create an appropriate Post-build step.

22.11.4 Binary files and checksums

When creating a binary file for most LPC MCUs, you also have to ensure that you apply a checksum to it - so that the LPC bootloader sees the image as being valid. Generally, the linker script does this if the managed linker script mechanism is used. Otherwise, the "checksum" utility found in the `ide/binaries` subdirectory of your MCUXpresso IDE installation can be used.

22.12 GCC 13 and freestanding environments

GCC 13 brings a major change when dealing with freestanding and hosted environments. In other words, more restrictions have been added, and GCC no longer allows inclusion of non-freestanding library headers in freestanding environments. As a result, newly created projects and example projects imported from SDK 2.16.000 are no longer built with the `-ffreestanding` flag.

Problems view shows information markers for all the projects that still use the `-ffreestanding` flag at build time.
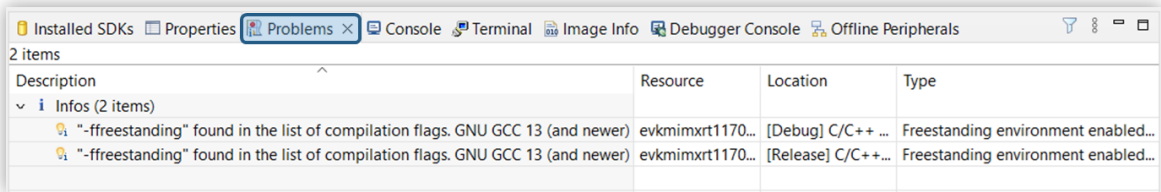
Figure 271. Problems view with `-ffreestanding` information marker

You can remove or ignore the marker, but the IDE also offers a Quick Fix action.
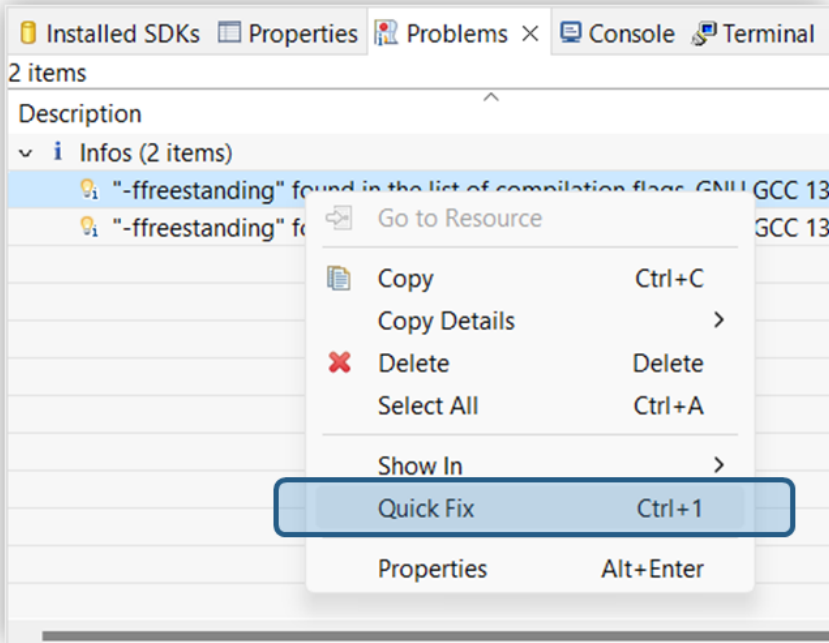
Figure 272. Problems view with Quick Fix for `-ffreestanding` information marker

The Quick Fix dialog is displayed and allows you to update the project or simply dismiss the information marker and keep the build settings of the project unchanged.
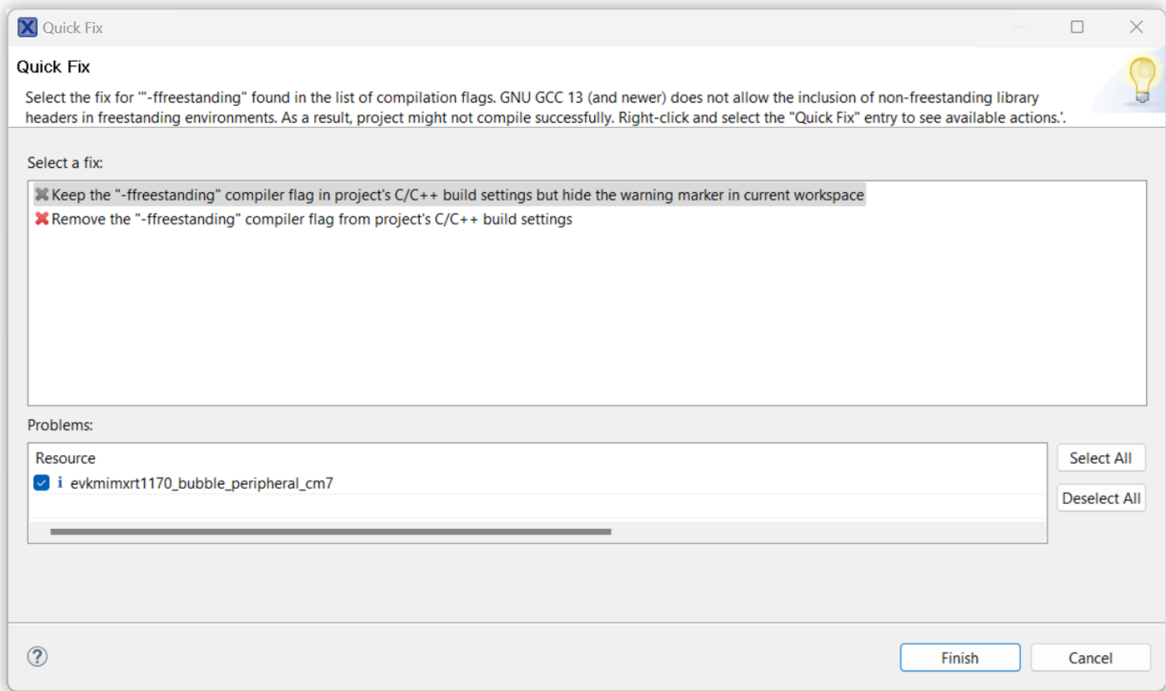
Figure 273. Quick Fix actions for `-ffreestanding` information marker

22.13 Building projects with DWARF 4 debug support

When creating or importing a new project, MCUXpresso IDE forces GCC to produce DWARF 4 debug information. GDB does not seem to fully support DWARF 5, which is the default in GCC 13 (and newer). By switching to DWARF 4, the IDE is able to offer a much smoother debugging experience.

Once the IDE detects that Debug Level is not set to *None* inside the project's build settings, `-gdwarf-4` is automatically added in the Other debugging flags section of the C/C++ compiler and Assembler. Moreover, the Problems view also contains an information marker that describes the change made to the project.
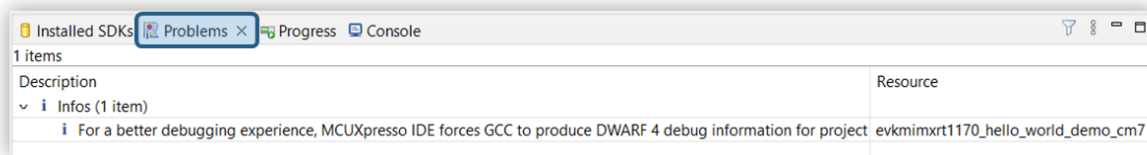
Figure 274. Problems view with DWARF 4 information marker

22.14 Post-build (and pre-build) steps

It is sometimes useful to be able to post-process automatically your linked application, typically to run one or more of the GNU 'binutils' on the generated AXF file.

For example, any application project that you create using the Project wizard has at least one such "post-build step" - typically to display the size of your application.
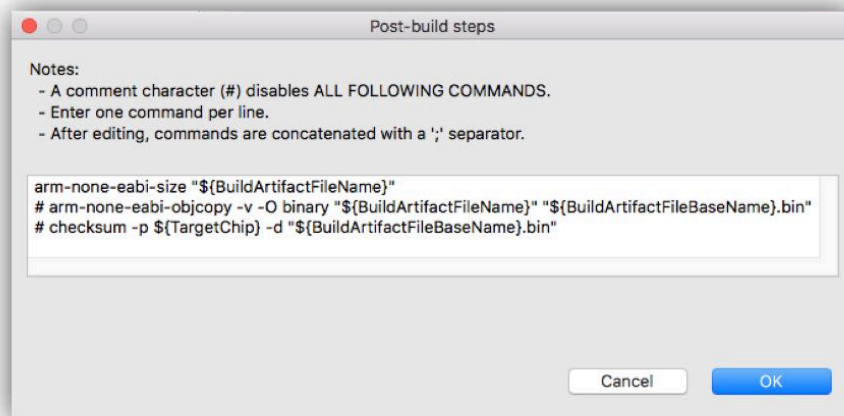
Figure 275. Post-build steps details

Note: Extra commands may also be listed (for example, to create a binary and to run a checksum command), but can be commented out by use of a # character and therefore not executed. Any commands following a comment #command is ignored.

Adding additional steps is simple. In the below example, we are going to carry out three post-link steps:

• Displaying the size of the application
• Generate an interleaved C/assembler listing
• Create a hex version of the application image

To do this:

- Open the Project properties. There are various ways of doing this. For example, make sure that the Project is highlighted in the Project Explorer view then open the menu "Project -> Properties".
- In the left-hand list of the Properties window, open "C/C++ Build" and select "Settings".
- Select the "Build steps" tab.
- In the "Post-build steps - Command" field, click 'Edit...'
  - Paste in the lines below and click 'OK'

```
arm-none-eabi-size ${BuildArtifactFileName};
arm-none-eabi-objdump -S ${BuildArtifactFileName} >
 ${BuildArtifactFileBaseName}.lss;
arm-none-eabi-objcopy -O ihex ${BuildArtifactFileName}
 ${BuildArtifactFileBaseName}.hex;
```

- Click apply
- Repeat for your other Build Configurations (Debug/Release)

Next time you do a build, this set of post-build steps will run, displaying the application size in the console, creating an interleaved C/assembler listing file called *.lss* and a hex file called *hex*.

Note: Pre-build steps can be added to a project in the same way if required.

22.14.1 Temporarily removing post-build steps

If you want to remove temporarily a step from your post-build process, rather than deleting it completely - move that entry to the end of the line and prefix it with a "#" (hash) character. This acts as a comment, causing the rest of the post-build steps to be ignored.

22.15 Save info for support

When reporting an issue, it is recommended to send to the product team all necessary information to easily reproduce the error you experience in the IDE working environment. For this reason, a new option ("**Help -> MCUXpresso IDE Save Info for Support**") is introduced in the Eclipse Help menu that helps you to gather and pack all information related to workspace, logs, and consoles:
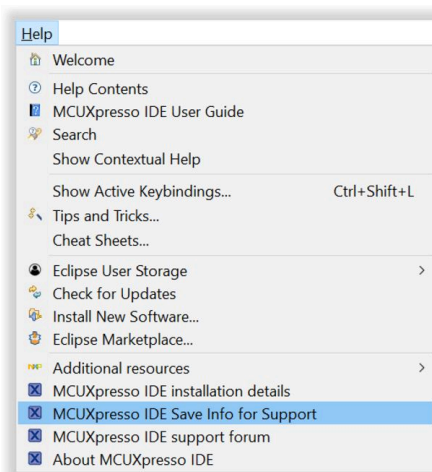
Figure 276. MCUXpresso IDE Save Info for Support

Once the wizard is open, various types of information can be individually selected (by default, all categories are preselected):
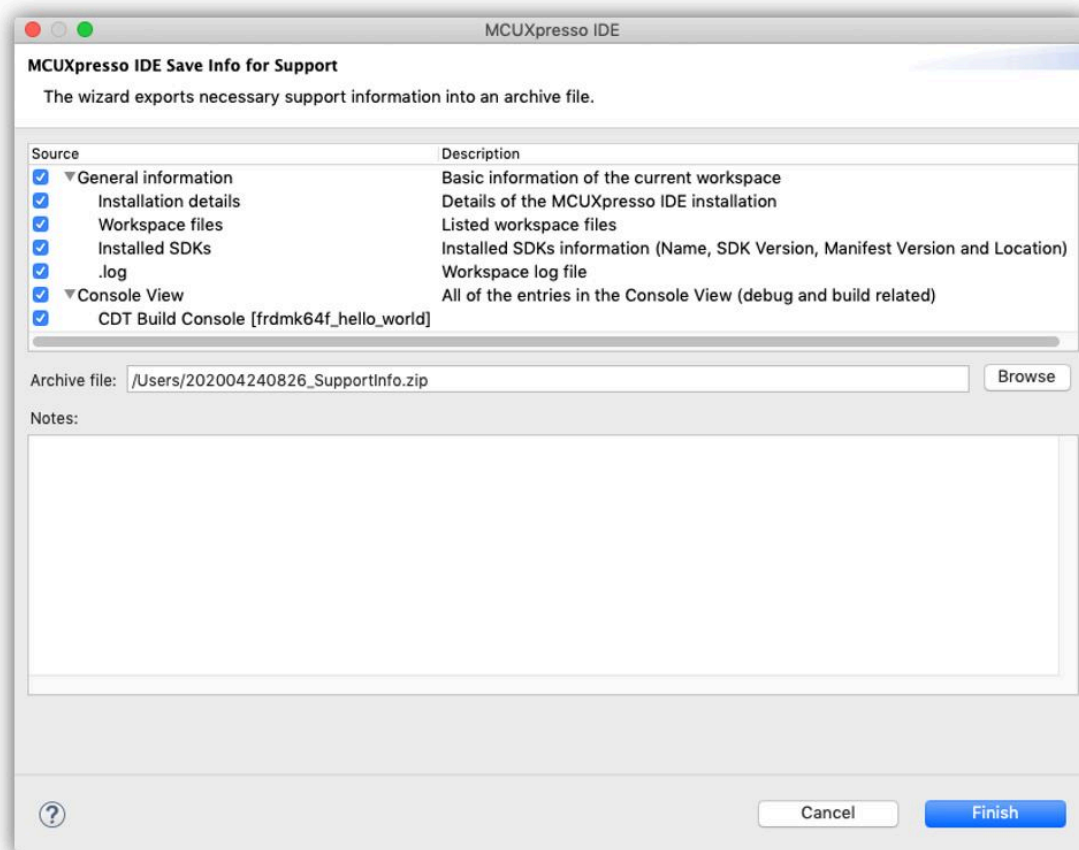
Figure 277. MCUXpresso IDE Save Info for Support wizard

- Installation details - information about installation paths, tool version, OS and Java versions, and IDE plugin versions
- Workspace files - a list with all filenames and directory structures within the workspace
- Installed SDKs - a list with generic information about all installed SDKs: name, SDK and manifest versions, and location path
- .log - can be one or more files and contain Eclipse logs generated in <workspace dir>/.metadata directory
- Console view - all consoles (Console Eclipse views) content: build log, debug and flash programming logs, standard input/output console, and so on

Note: No sensitive information like source code or file content (other than eclipse/build/debug logs) is included!

Optionally, add details in the "Notes:" edit box, located at the bottom of the "Save Info for Support" window, regarding any information you think might be helpful: steps to reproduce, errors you observe, expected results, and so on. If used, the text content is added to the final zip file.

After the selection is done and a proper zip filename and path are chosen, press the 'Finish' button to generate the archived support information. Afterward, use this zip as an attachment when reporting the problem.

23   Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024-2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

24 Revision history

Table 6. Revision history

Document ID	Release date	Description
UG10055 v.4	14 January 2025	24.12 - major release version update. See chapter 2 for details.
UG10055 v.3	1 July 2024	11.10.0 - major release version update. See chapter 2 for details.
UG10055 v.2	17 January 2024	11.9.0 - major release version update. See chapter 2 for details.
UG10055 v.1	31 July 2023	11.8.0 - major release version update. See chapter 2 for details.

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Tables

Figures

Contents