SETTING UP AN OPEN-SOURCE, ECLIPSE-BASED DEVELOPMENT ENVIRONMENT FOR NXP ARM CORTEX MICROCONTROLLERS

ECE-633 Independent Study Report

Michael D'Argenio
mjdargen@ncsu.edu

TABLE OF CONTENTS

1 Project Overview 3

1.1 Introduction 3

1.2 Base Code GNU Transition 4

1.3 Processor Expert Project Transition 5

2 Development Environment Installation 7

2.1 Windows Indexing Issues 7

2.2 GNU ARM Embedded Toolchain 8

2.2.1 Overview 8

2.2.2 Installation Steps 8

2.3 MCUXpresso Setup – Windows/Linux/MAC 9

2.3.1 Overview 9

2.3.2 Installation Steps 9

2.3.3 Workspace Issues 11

2.4 GNU MCU Eclipse Plug-ins 12

2.4.1 Overview 12

2.4.2 Installation 12

2.5 NXP MCUXPresso SDK 13

2.5.1 Overview 13

2.5.2 Installation Steps 13

2.6 Processor Expert 15

2.6.1 Overview 15

2.6.2 Installation Steps 15

2.7 Eclox 20

2.7.1 Overview 20

2.7.2 Installation Steps 20

2.8 Debuggers 21

2.8.1 Overview 21

2.8.2 Installation Steps 21

2.9 PEmicro OpenSDA Bootloader/Application Firmware 22

2.9.1 Overview 22

2.9.2 Installation Steps 22
3 Usage 24
3.1 MCUXpresso 24
3.2 Image Info 25
3.3 Disassembly View 27
3.4 Peripheral Viewer 28
3.5 Register Viewer 30
3.6 Processor Expert 31
3.7 Doxygen Webpage Creation 32
3.8 Static Stack Depth Analysis 35
3.9 Dynamic Stack Depth Analysis & Task Profiling 36

1 PROJECT OVERVIEW

1.1 INTRODUCTION

Over the course of this semester, I migrated the existing code base for ECE 560/561 projects away from the ARM Keil Microcontroller Development Kit (MDK) and added a lot of additional functionality to the projects. It was a long learning process with a constantly learning scope. The initial catalyst for the project was code limit barrier. The Keil MDK comprises of Keil uVision IDE, ARM GCC Toolchain, and many other great features; however, the free educational version has a code limit of 32 kB. The full licenses are too expensive and not feasible for an undergraduate classroom. As the embedded systems courses continued to develop, there was a need to go beyond the 32 kB code limit.

The courses are presently designed around the FRDM-KL25Z which is an NXP controller built on an ARM Cortex-M0+ processor. We are using the MKL25Z128VLK4 which has 128 kB of flash memory meaning that we can only use about 25% of the code-size with Keil MDK. As the shield functionality has been built up, there is more need to utilize the full 128 kB to support additional features such as a FatFS, more extensive graphics/fonts, and more. Being able to support these extended features (in particular navigating a FatFS on an SD card which essentially provides us with seemingly endless memory) has become a necessity for further improving and developing new projects for the course.

I began to investigate what other tools could be used. At Dr. Dean's encouragement, I investigated moving to the GNU ARM Embedded Toolchain. Using the GNU toolchain would allow us to use GCC, GDB, GNU binutils, etc. as well as a number of other free and open-source tools that have been built for GCC. After reading more about the GNU MCU Eclipse project and Erich Styger's blog "MCU on Eclipse", I decided to move to an Eclipse-based IDE. Eclipse is a free and open-source IDE that is very heavily used. Due to its prevalence, a number of very helpful plugins have been developed. Thanks to projects like the GNU MCU Eclipse project, the design space has grown immensely for embedded systems within Eclipse.

Instead of going with a base installation of Eclipse and building it up from there, I went with MCUXpresso from NXP. MCUXpresso is an Eclipse-based IDE that already has a lot of built-in support for NXP microcontrollers, FreeRTOS, and many other things. Dr. Dean intends to continue using NXP microcontrollers in his courses. By using the vendor-provided Eclipse-based IDE, we could leverage more of the functionality and reduce the installation process for the students. I had also investigated other NXP/Freescale IDEs like CodeWarrior and Kinetis Design Studio. They had some unique and interesting tools like Processor Expert; however, I discovered these could be ported over to MCUXpresso.

I began porting the code over to MCUXpresso using the new GNU toolchain and compiling the project with GCC. This required a number of modifications (detailed in Section 1.2) and getting used to the new IDE/toolchain/compiler. Once I was finally able to successfully build and execute the project, I began to investigate what the new environment had to offer in terms of design and analysis tools. MCUXpresso has a lot of great native support for FreeRTOS including task-aware debugging, task stack size analysis, task profiling, heap usage analysis, and more. I modified the code to move from Keil RTX5 which used the CMSIS-RTOS API v2 API to support FreeRTOS to leverage this functionality.

As mentioned earlier, Processor Expert could be supported in MCUXpresso and adds a lot of very useful functionality. Processor Expert is a source code generation tool used to create and configure different software components. The components can represent a hardware abstraction, a peripheral driver, or a specific algorithm. I created a project with identical functionality as the existing shield project by creating and leveraging Processor Expert components. There was a large learning curve to get used to the creation and configuration of components to replicate identical functionality, but the Processor Expert interface is fairly intuitive and a great tool. The Processor Expert component setup for the shield project is detailed in section 1.3.

After successfully building and executing a Processor Expert project with identical functionality to the original Keil shield project, I began to extend some of the functionality of the project. I added support for a FatFS SD card over the SPI port and a UART command line interface with the system over the OpenSDA USB connection. The SD card can be used in future projects to decode a number of images for display or hold a .wav sound buffer for the speaker. The command line interface can be used to modify functionality of the system in real time and can be very helpful for debugging purposes. These two improvements pave the way for a number of new projects for the courses.

The following pages detail the new ECE 560/561 project code base as setup in the new environment. It describes the changes that were made and the required setup for this new style of project. I also thoroughly detail the steps for installing and configuring this new development environment and tool chain in Section 2. The details for using this new environment and overview of useful design and analysis tools are detailed in Section 3.

1.2 BASE CODE GNU TRANSITION

Below are the steps I followed to import the Keil MDK shield code as is into MCUXpresso. The following changes must be made in order to get the project to successfully build. The next section details how I re-created the same functionality using Processor Expert and FreeRTOS.

- Create new project
    - KL25Z source
    - Include nothing
    - Newlibnano(none)
    - enable printf/scanf float
- Need to update board source files, import from Keil projects
    - MKL25Z4.h (some discrepancies in base memory address pointers)
    - system_MKL25Z4.c
    - system_MKL25Z4.h
- Changed pragmas in I2C.c
    - #pragma noline --> **attribute** ((noinline))
- Properties->C/C++ Build ->Settings ->MCU Linker -> Managed Linker Script
    - Newlibnano(none)
- Properties->C/C++ Build ->Settings ->MCU Linker -> Managed Linker Script
    - enable printf/scanf float

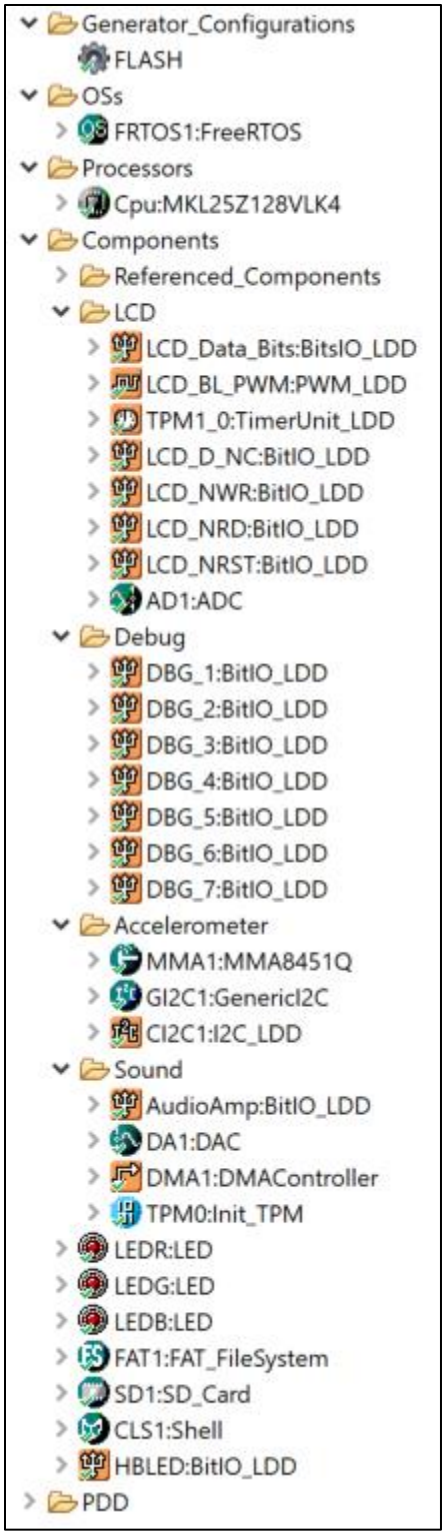
- Properties->C/C++ Build ->Settings ->MCU Linker -> General
    - no startup or standard libraries
- Import and build with CMSIS5
    - Source files available here -> https://github.com/ARM-software/CMSIS_5
    - The following files must be included
        - libRTX_CM0.a
        - cmsis_gcc.h -> replace native file with one from CMSIS5
        - cmsis_compiler
        - cmsis_os2.h
        - RTX_Config.c
        - RTX_Config.h
        - rtx_evr.h
        - rtx_os.h
        - rtx_lib.c
        - rtx_lib.h
- Add libRTX_CM0.a library
    - Properties -> C/C++ General -> Paths and Symbols -> Libraries
        - Add "RTX_CM0"
    - Properties -> C/C++ General -> Paths and Symbols -> Library Paths
        - Add "/${ProjName}/CMSIS/RTX5"
        - Check "Is a workplace path"
- Update Includes for all new directories
    - Properties -> C/C++ General -> Paths and Symbols -> Includes
    - Add all new paths just like already existing paths
- Enable clock to port A in LCD_GPIO_Init() ST7789.c
    - // Enable clock to ports
    - SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTC_MASK | SIM_SCGC5_PORTE_MASK;

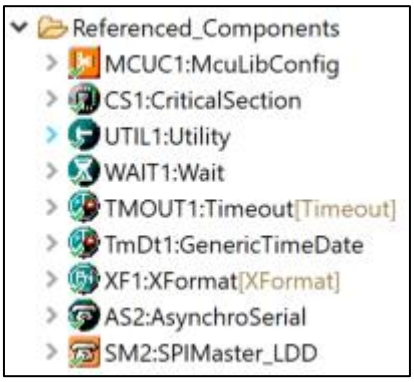
1.3 PROCESSOR EXPERT PROJECT TRANSITION

On the next page, you can view all the different components that I used to recreate the shield project code case. I organized the components in Processor Expert based on their function. To view the specific configurations, you can either open the project in MCUExpresso and look at the Processor expert components or go to the webpage below to view the documentation generated by Processor Expert which describes the configurations that were done.

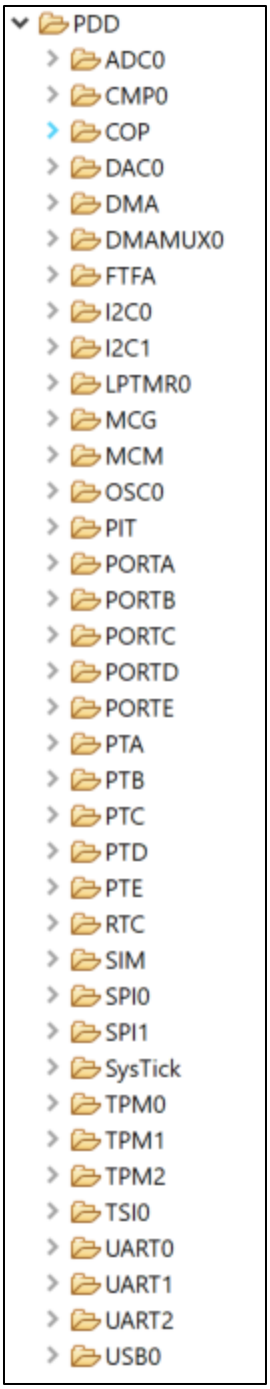
Project repository - https://github.com/mjdargen/MCUX_PE_KL25Z_FRTOS_ShieldwFatFS

Project repository's website - https://mjdargen.github.io/MCUX_PE_KL25Z_FRTOS_ShieldwFatFS





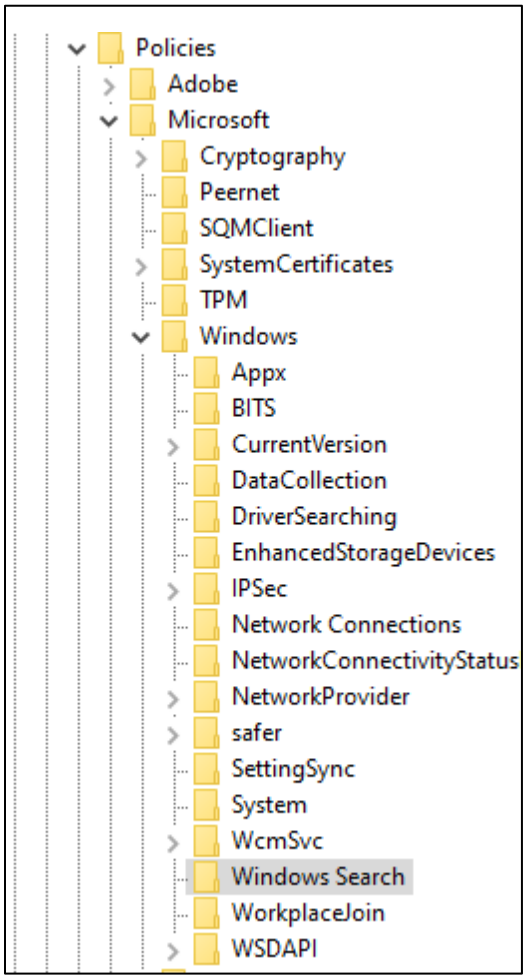
The top-level main Processor Expert components are shown to the left. The components are organized in a hierarchical structure based on their function (i.e. LCD, Sound). Above are the referenced components that the main components depend on. In addition, there are the Physical Device Drivers (PDD) shown on the right. These are macros which can be used to access the peripheral configuration registers.



2 DEVELOPMENT ENVIRONMENT INSTALLATION

2.1 WINDOWS INDEXING ISSUES

[*For Windows Only*] Windows may corrupt the debug MCU bootloader on the FRDM-KL25Z MCU by trying to index it. This will make the debug MCU unusable. To prevent this problem, you will need to disable indexing on removable storage devices.

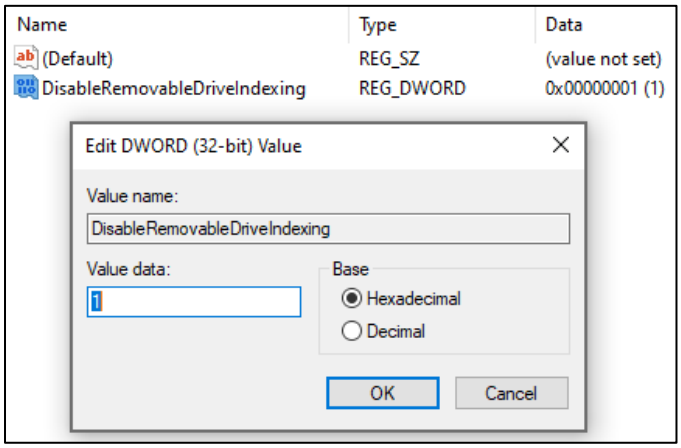


1. Run the Registry Editor aka regedit.exe by typing "regedit" into the Windows search bar.

2. Navigate to the registry: "HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\Windows Search"

If there is not a key named Windows Search in the Windows folder, navigate to "HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows", right-click in the window and select "New -> Key". Name the key "Windows Search". It should appear on the left under Windows as shown.

3. Create a new DWORD by right-clicking in the window and selecting "New->DWORD". Name it "DisableRemovableDriveIndexing" and give it a value of 1.



You can read about the problem and another approach to the solution on Erich Styger's website here: https://mcuoneclipse.com/2016/08/01/bricking_and_recovering_opensda_boards_in_windows_8_and_10/

2.2 GNU ARM EMBEDDED TOOLCHAIN

2.2.1 OVERVIEW

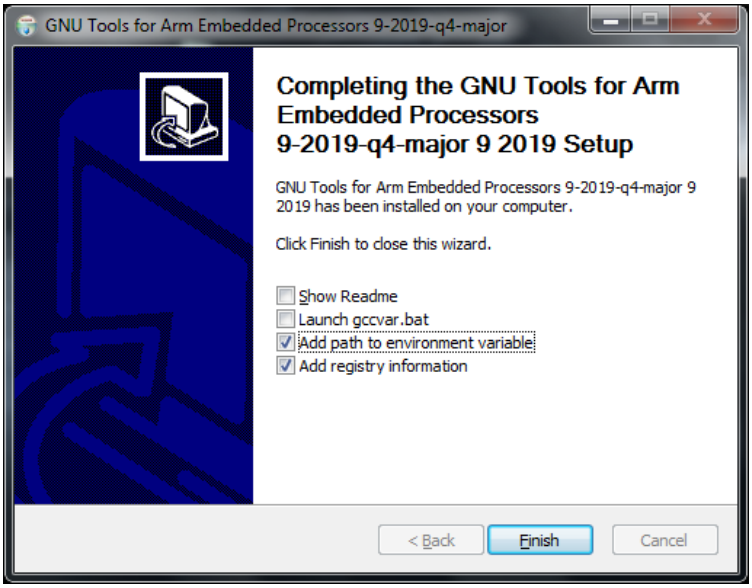
From GNU ARM Embedded Toolchain website: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm

"The GNU Arm Embedded toolchain contains integrated and validated packages featuring the Arm Embedded GCC compiler, libraries and other GNU tools necessary for bare-metal software development on devices based on the Arm Cortex-M and Cortex-R processors. The toolchains are available for cross-compilation on Microsoft Windows, Linux and Mac OS X host operating systems."

The toolchain includes: GNU C/C++ Compiler, Binutils, GDB, Newlib.

2.2.2 INSTALLATION STEPS

1. Go to the following URL and download the most recent GNU Embedded Toolchain for Arm for your operating system: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads
2. Launch the installation executable. Follow the standard installation steps.
3. Once the installation has completed, you will come to the last screen. Check the box to "Add path to environment variable" and "Add registry information" before clicking "Finish" button.

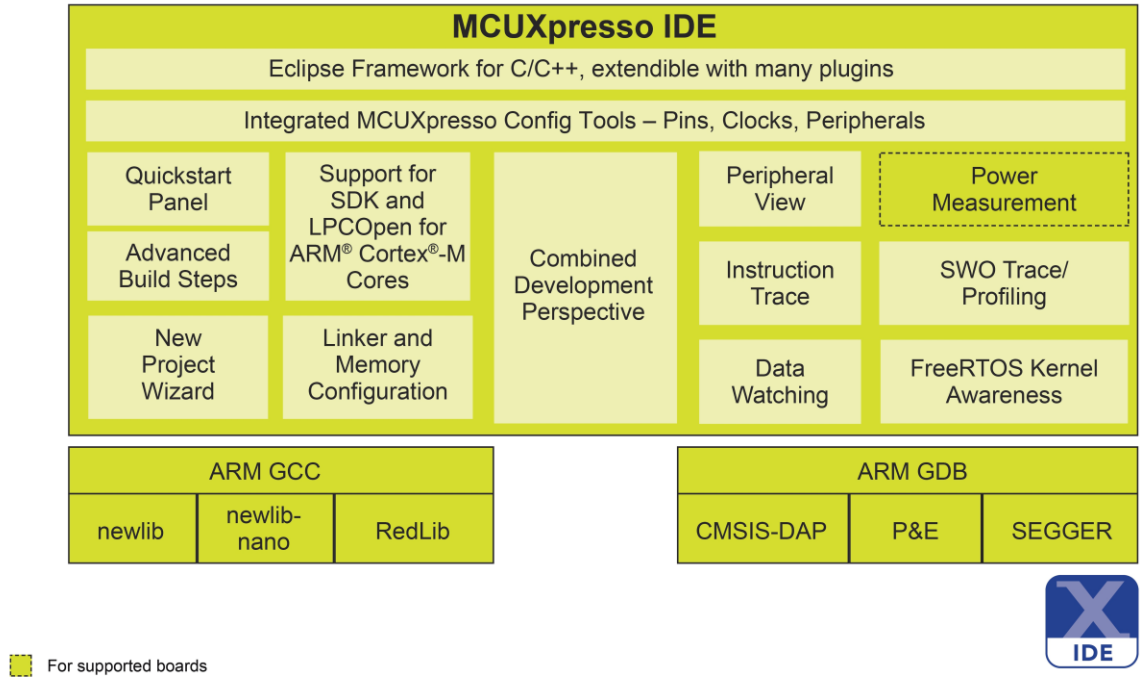


2.3 MCUXPRESSO SETUP – WINDOWS/LINUX/MAC

2.3.1 OVERVIEW

MCUXpresso is an Eclipse-based IDE released by NXP for their microcontrollers. It includes many helpful utilities for configuring your MCU, importing MCUXpresso SDK examples, and debugging. The installation includes many debug drivers in additional to tools for your debugger to view memory usage, heap usage, stack usage, code trace, profiling, and FreeRTOS statistics. More information about what is offered as a part of MCUXpresso IDE is detailed below in the diagram.

MCUXpresso IDE BLOCK DIAGRAM



As noted above, the installation of MCUXpresso contains an older version of the GNU ARM Embedded Toolchain. However, to utilize Processor Expert and some of the other useful plug-ins and extensions we will be need, it is useful to install a newer version of the GNU ARM Embedded Toolchain as a standalone installation as done above.

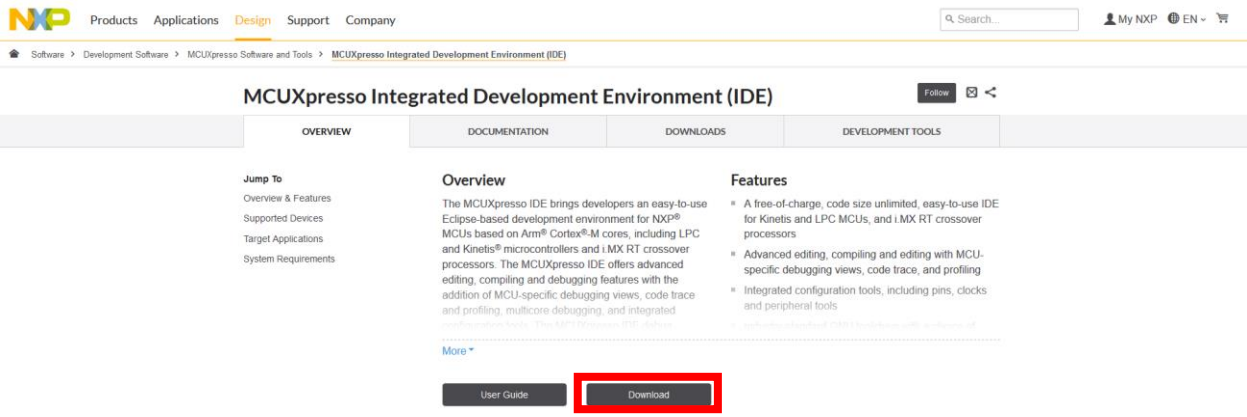
Depending upon your machine, the installation can take >10 minutes to install. The installation requires multiple reacknowledgements for additional drivers and software to be installed as a part of MCUXpresso, so you will have to keep an eye on the installation to continually advance the installation.

2.3.2 INSTALLATION STEPS

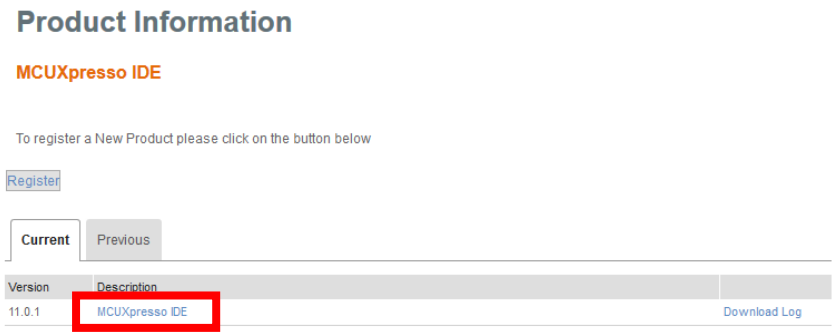
1. Go to the NXP website for the MCUXpresso IDE. Click on “Download” button.

Main link - https://www.nxp.com/mcuxpresso/ide

Alternate link - https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE

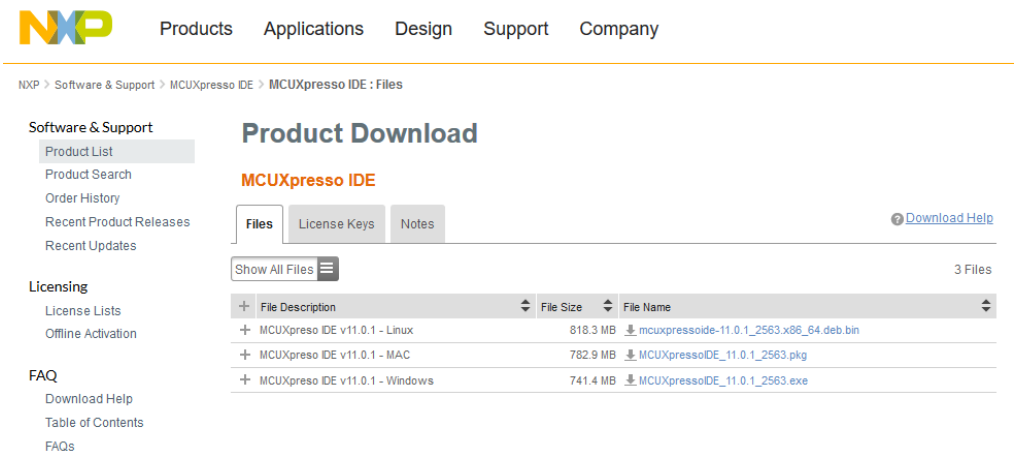


2. You will then be redirected to page to login or create an NXP account. If you need to create an NXP account, create an account with your .edu email address stating that you are a student as your job. You will need to verify your email address.
3. Once you have verified your email address and successfully logged in, you can return to the MCUXpresso IDE download page. You should now be able to click on the “Download” button and be redirected to the page below.
4. Click on MCUXpresso IDE. You will then be redirected to the Software Terms and Conditions. Once you have agreed to the terms, you should be redirected to the Product Download page. (If not, you may have to wait a couple minutes for approval. It should typically be instantaneous. You will receive an email when it has been added to your NXP Software Account.)



5. Once your software is ready to be downloaded, select your OS. There are MCUXpresso IDE installations for Windows/MAC/Linux.

NOTE: I will be focusing on the Windows steps for MCUXpresso IDE 11.0.1. These steps should be applicable for MAC and Linux as well as future versions of the IDE.



6. Follow the standard installation instructions for MCUXpresso IDE.
7. Throughout the installation, there may be a number of pop-ups from your operating system asking if you would like to install additional software. That is because the MCUXpresso installation has multiple installations bundled together including various device and debugger drivers. Allow these drivers and additional software to be installed. All software should come from NXP, Jungo Connectivity, P&E Micro, SEGGER, and ASHLING Microsystems.
8. There may also be some firewall issues/complaints the first time you open MCUXpresso or launch any of the debuggers. Be sure you list these programs on the whitelist to ensure that they will successfully run.

2.3.3 WORKSPACE ISSUES

Occasionally, you may run into issues when trying to launch MCUXpresso or type of Eclipse IDE. If the program locks up after you open it and select the workspace, there may be an issue in that particular workspace that is causing this issue. A common problem in Eclipse is that some of the different perspectives/views can prevent the workspace from loading. Since we will be using a handful of different plugins and extensions, this can occasionally occur.

The solution to this issue is removing a workbench metadata file that is responsible for the arrangements for the different window pespectives. By removing this file, you are able to refresh the workspace and launch MCUXpresso. The file is called “workbench.xmi” and can be found at the following location:

“$your workspace $/.metadata/.plugins/org.eclipse.e4.workbench/workbench.xmi”

2.4 GNU MCU ECLIPSE PLUG-INS

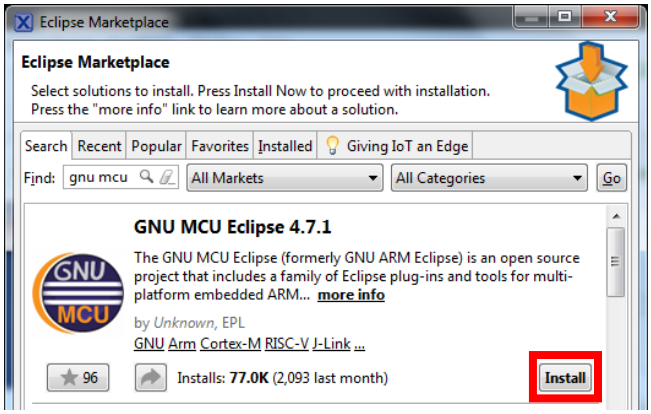
2.4.1 OVERVIEW

The GNU MCU Eclipse plug-in enables a lot of the tools from the GNU ARM Embedded Toolchain in Eclipse. This is required to have the appropriate compiler and linker for Processor Expert projects.

More information about the tools and plug-ins: https://gnu-mcu-eclipse.github.io/

2.4.2 INSTALLATION

1. In MCUXpresso, go to the “Help” dropdown in the main toolbar and select “Eclipse Marketplace”. Search for “GNU MCU Eclipse” in the marketplace. Find and install the latest version of GNU MCU Eclipse.



2. Follow the installation steps. When prompted, restart MCUXpresso.

NOTE: Windows users only beyond here. For more info on the tools, go here: https://gnu-mcu-eclipse.github.io/windows-build-tools/install/

3. To install certain additional command line programs like make and rm for Windows, there are some additional steps necessary. These are required by the external builder in Eclipse. These can be installed as an xPack if you have xpm set-up. Otherwise, visit the following URL to download the latest Windows build tools for your OS (32-bit/64-bit).
4. Unzip the file. You should have a top level directory called “GNU MCU Eclipse”. The file structure should be “GNU MCU Eclipse\Build Tools\2.12-20190422-1053\...”
5. Move the root folder “GNU MCU Eclipse” to “C:\Users*%user name%*\AppData\Roaming\” e.g. “C:\Users\Michael\AppData\Roaming\GNU MCU Eclipse”.
6. There is no need to update the user or system path.

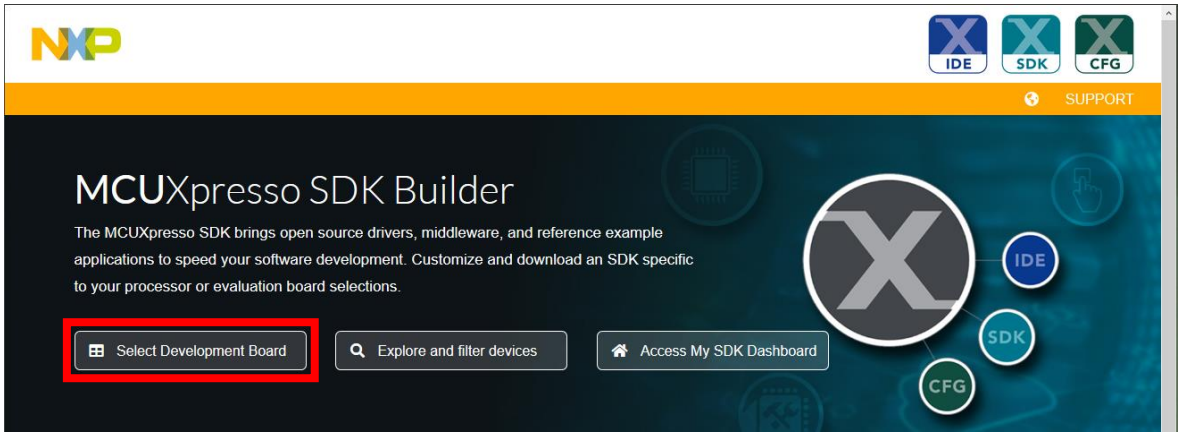
2.5 NXP MCUXPRESSO SDK

2.5.1 OVERVIEW

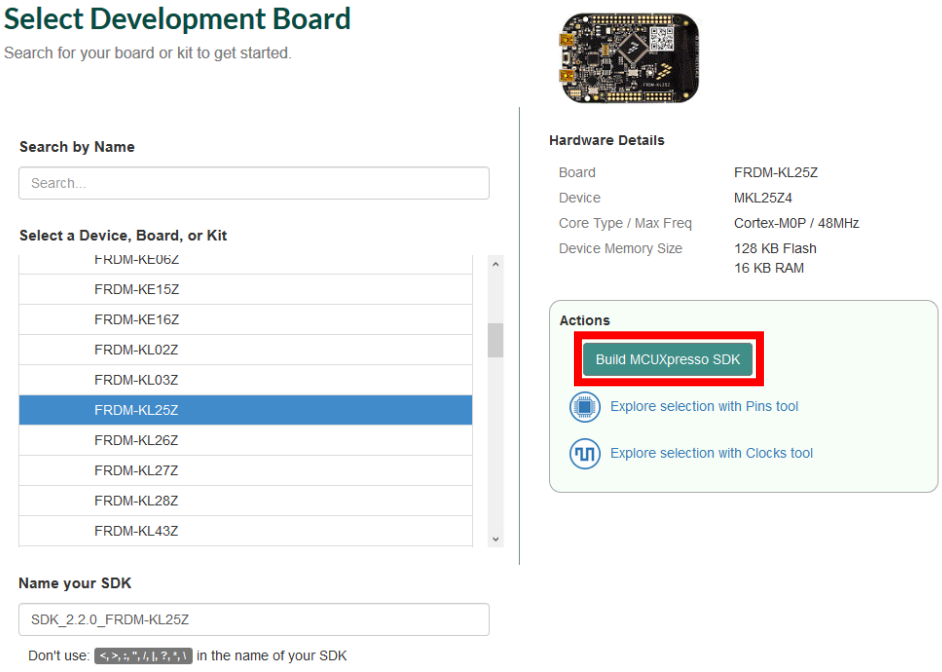
The MCUXpresso SDK does not work with Processor Expert meaning that the example projects provided by the SDK do not have Processor Expert components in them. However, the SDK provides useful examples and the necessary startup and linker files for non-Processor Expert projects.

2.5.2 INSTALLATION STEPS

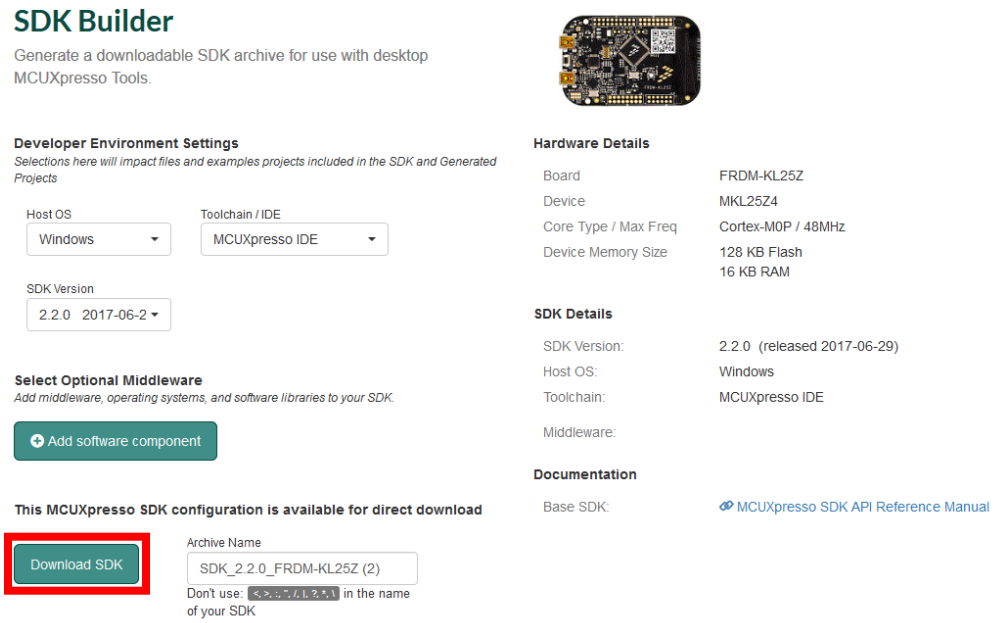
1. Go to the MCUXpresso SDK Builder: https://mcuxpresso.nxp.com/en/welcome. Click “Select Development Board”.



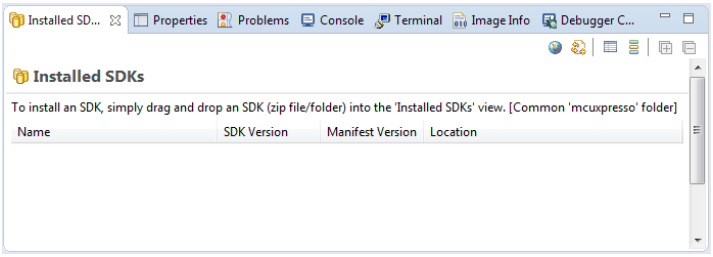
2. You will then be prompted to log back into your NXP account. Upon logging in, you will be directed to the SDK Builder. Select your board as shown below then click “Build MCUXpresso SDK”.

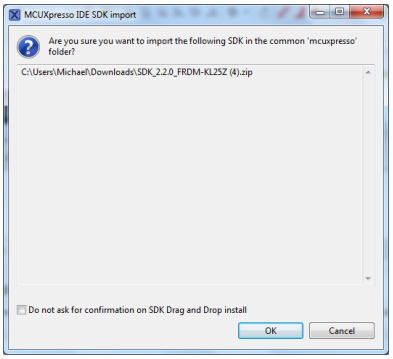


3. Now select your OS, IDE, and SDK version and click “Download SDK”. You can also select optional middleware if you would like.

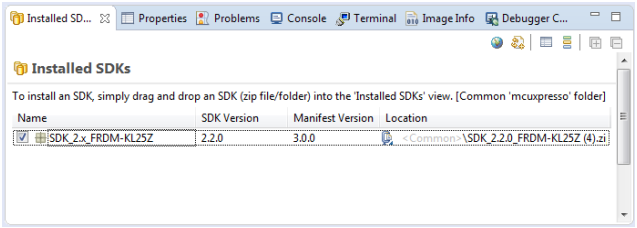


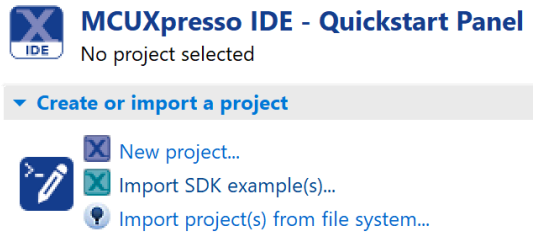
4. To import the SDK into MCUXpresso IDE, navigate to the “Installed SDKs” window in the IDE. You can drag the zip file you downloaded from the online MCUXpresso SDK tool into this window to import.





5. The SDK is installed. You can now view SDK examples and create new SDK projects.





2.6 PROCESSOR EXPERT

2.6.1 OVERVIEW

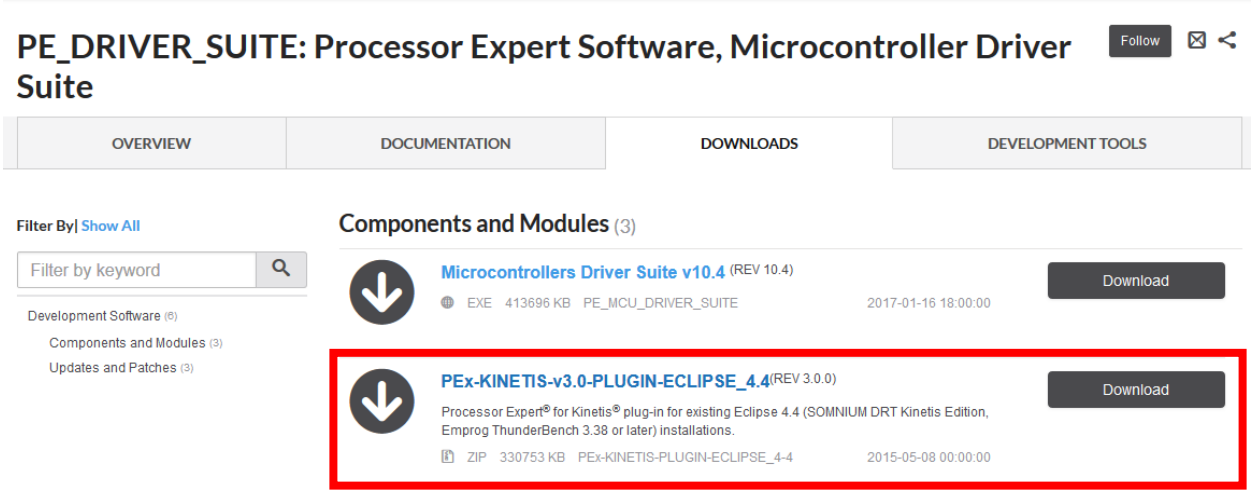
Processor Expert is an NXP microcontroller configuration and code generation tool. It provides an intuitive interface for configuring the processor, peripherals, and RTOS. In addition to configuration, it also provides an API for every component that is added. For example, you can add an ADC "component" in Processor Expert. The component viewer allows you to configure the ADC by selecting the channel, bit resolution, sampling time, etc. Once you have configured the component, you can generate code which creates functions that will initialize the peripheral and an entire API with a whole variety of methods to call. Methods that will start a conversion, retrieve a value, etc. etc.

Processor Expert is not natively supported by the MCUXpresso IDE. It was created for a deprecated NXP IDE, but it can very easily be added using the following steps.

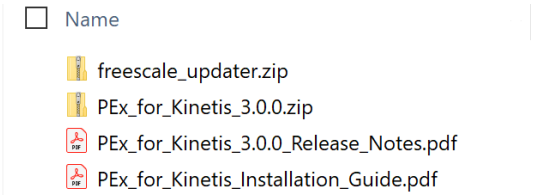
2.6.2 INSTALLATION STEPS

1. Go to the following URL and download PEx-KINETIS-v3.0-PLUGIN-ECLIPSE_4.4 (or Processor Expert Kinetis v3.0 Plugin for Eclipse 4.4 or later).

   https://www.nxp.com/design/software/development-software/processor-expert-software/processor-expert-software-microcontroller-driver-suite:PE_DRIVER_SUITE?&&tab=Design_Tools_Tab

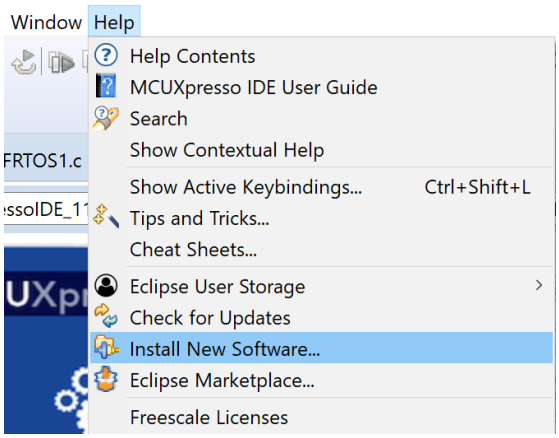


2. Unzip "PEx_for_Kinetis_3.0.0_Install_into_Eclipse_4.4_Unzip_me.zip". You should the directory structure below:

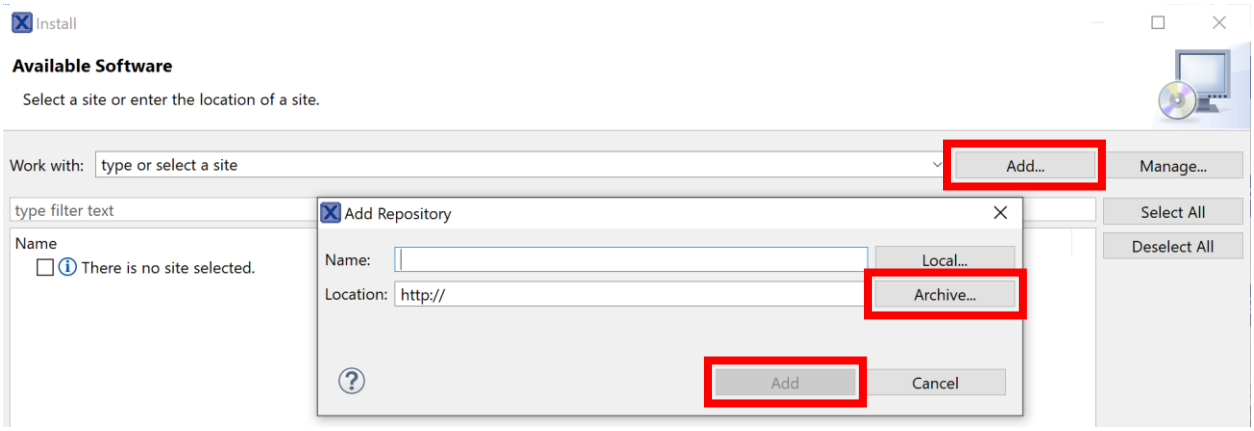


3. PEx_for_Kinetis_Installation_Guide.pdf has all the information to install Processor Expert 3.0.0. I will summarize this information in the following steps and show the additional steps required to update Processor Expert to the latest version 3.0.2.

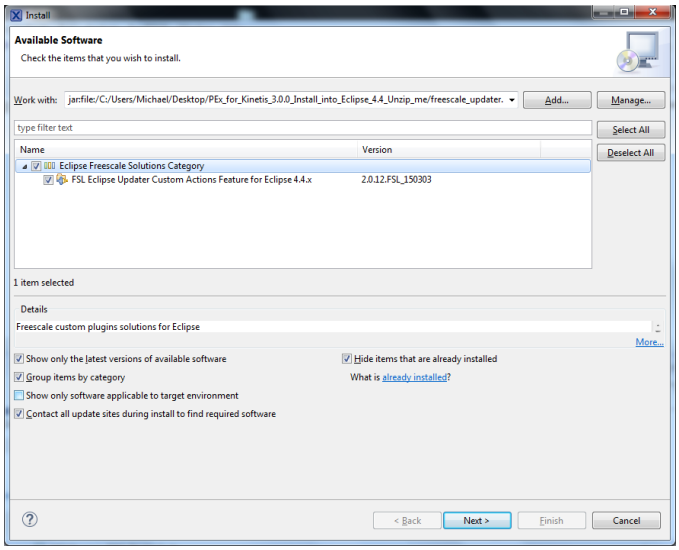
4. First you will need to install freescale_updater.zip. You need this plugin to be able to install and use Processor Expert. In MCUXpresso IDE, select “Help → Install New Software” from the main toolbar.



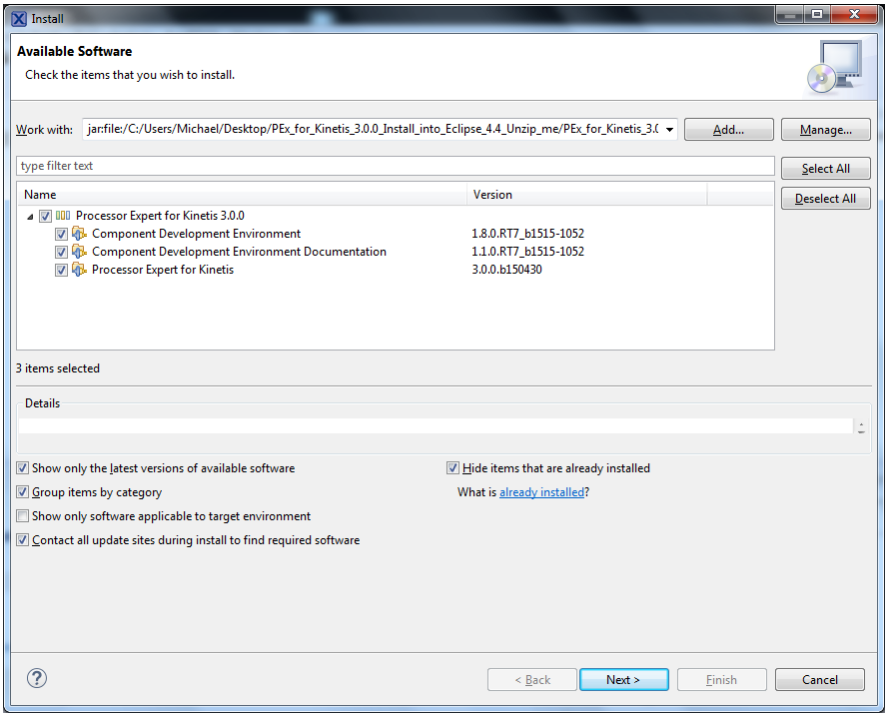
5. Drag zip file “freescale_updater.zip” to the “Work with:” dialog box. This should prepare the zip file to be imported and installed.
   Or if the drag and drop method does not work, you can click “Add” Button next to “Work with:” dialog box. “Next” and install the Freescale Eclipse Updater. Click “Archive” button in “Add Repository” pop-up window. Navigate to the directory and select “freescale_updater.zip” and click “Open” button.



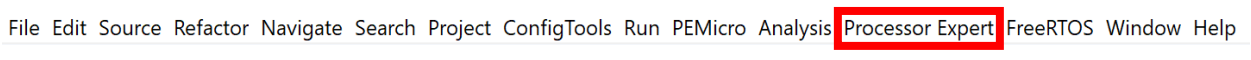
6. If you have added the zip file repository successfully, it should automatically populate the name and other information as shown below.



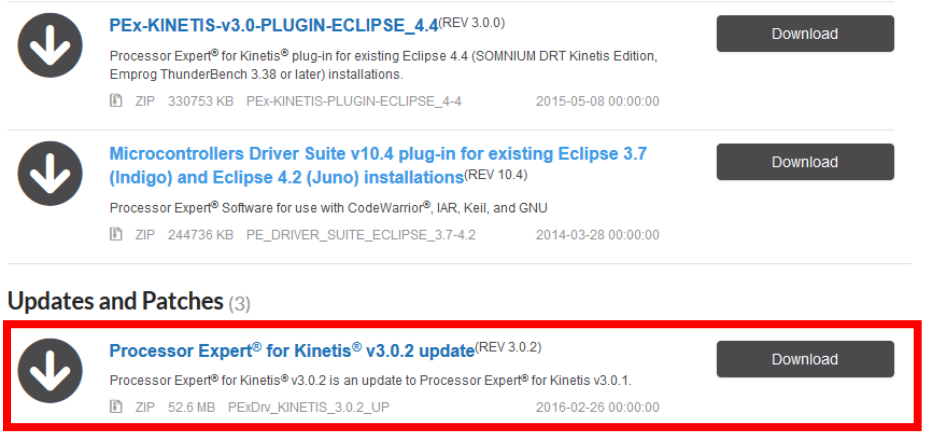
7. Click “Next” and proceed through the installation steps. Accept the license agreement. You will receive a warning that you are installing software with unsigned content. Click “Install anyway”. Accept the certificates and proceed until the installation is complete. It will prompt you to restart MCUXpresso. Be sure that you restart it.
8. After MCUXpresso has successfully restarted, you will repeat the same steps to install PEx_for_Kinetis_3.0.0.zip. After successfully adding the zip file repository, it should look like this.



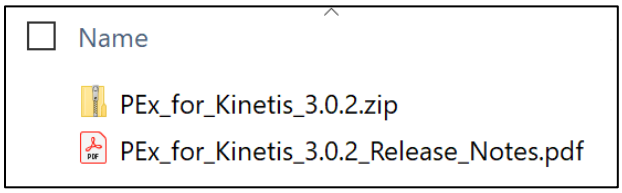
9. Proceed through the installation steps as before and restart MCUXpresso when prompted.
10. At this point, you should now see a Processor Expert dropdown in the main toolbar in the IDE.



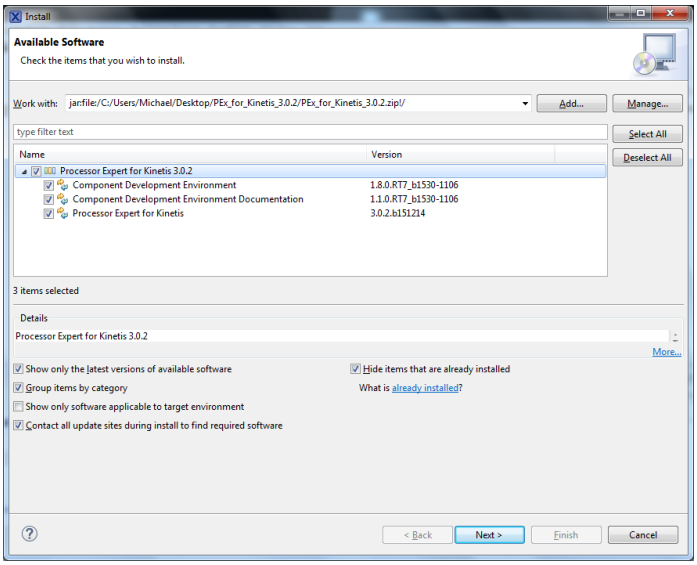
11. Now it is time to download and install the updated to version 3.0.2 of Processor Expert. Visit the following URL and download “Processor Expert for Kinetis v3.0.2.zip”. https://www.nxp.com/design/software/development-software/processor-expert-software/processor-expert-software-microcontroller-driver-suite:PE_DRIVER_SUITE?&&tab=Design_Tools_Tab



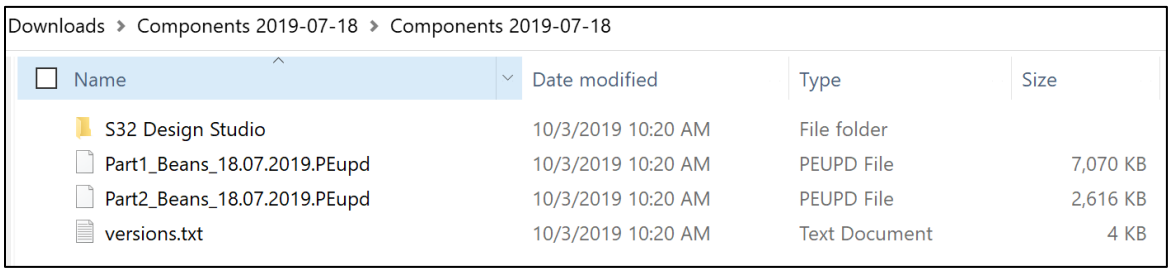
12. Unzip “PEx_for_Kinetis_3.0.2.zip”. You should now see the directory structure below. (Yes, there is a file called PEx_for_Kinetis_3.0.2.zip in the file PEx_for_Kinetis_3.0.2.zip. Make sure that you install the zip file that was extracted from the downloaded zip file.)



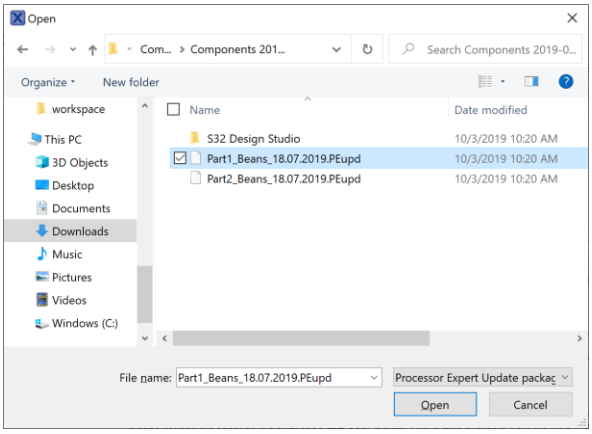
13. Now install Processor Expert 3.0.2 following the same steps you did for the previously. You can make sure that you are installing the correct zip file by confirming that it shows up like this.



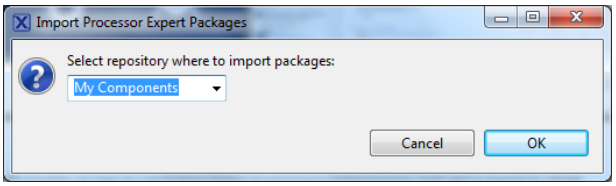
14. Once MCUXpresso has successfully restarted again, it is time to update the components in Processor Expert. Download the most recent component updates from Sourceforge at this link: https://sourceforge.net/projects/mcuoneclipse/files/PEx%20Components/
15. Once you have downloaded the zip file of the most recent components, unzip them. The files in the zip folder should appear as shown below.

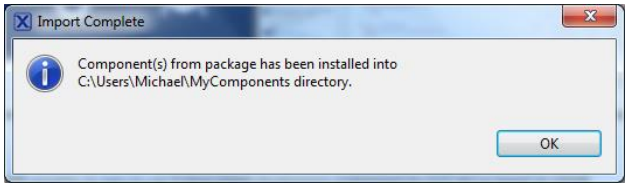


16. In MCUXpresso, select the Processor Expert dropdown menu and click “Import Component(s)”. Navigate to the directory where your .PEupd files are located.

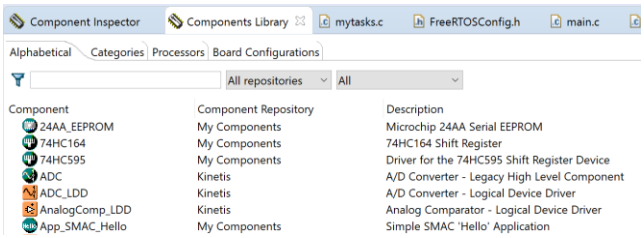


17. Add them one at a time to “My Components” (you can make your own repository if you want).





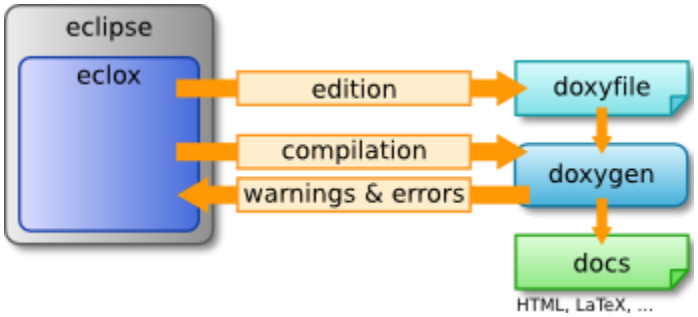
18. Return to the “Processor Expert” dropdown menu and select “Show Views.” In the Components Library window, you should see a collection of Processor Expert components from Kinetis and from My Components. Processor Expert has been successfully set up!
19. For more information on setting up different component repositories (if you have multiple versions of components), you can read more on Erich Styger’s website here: https://mcuoneclipse.com/2015/07/06/processor-expert-component-repositories/



2.7 ECLOX

2.7.1 OVERVIEW

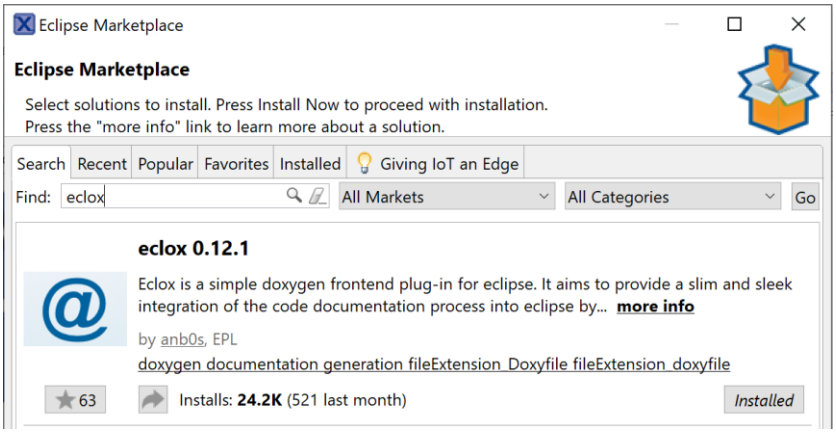
Eclox is a plugin for Eclipse that enables doxygen. Processor Expert generates a very useful doxyfile. By using eclox, you can automatically generate some incredibly helpful documentation for your projects.



For more information about eclox, visit the documentation website: https://anb0s.github.io/eclox/

2.7.2 INSTALLATION STEPS

1. In MCUXpresso, go to the “Help” dropdown in the main toolbar and select “Eclipse Marketplace”. Search for “eclox” in the marketplace. Find and install the latest version of eclox.



2. Follow the installation steps. When prompted, restart MCUXpresso.

2.8 DEBUGGERS

2.8.1 OVERVIEW

The MCUXpresso IDE installation includes all of the necessary drivers and supporting software for 3 different debuggers: Native LinkServer, P&E Micro, and SEGGER J-Link.

Native LinkServer (or RedLink) supports a variety of debug probes including OpenSDA programmed with CMSIS-DAP firmware. https://community.nxp.com/thread/389157

P&E Micro supports a variety of debug probes including OpenSDA programmed with P&E compatible firmware, MultiLink, and Cyclone debug probes. http://www.pemicro.com/

SEGGER J-Link supports a variety of debug probes including OpenSDA programmed with J-Link compatible firmware and J-Link debug probes. https://www.segger.com/

2.8.2 INSTALLATION STEPS

1. No installation required. Installed as a part of the MCUXpresso installation. The installation may not include the most up-to-date installations of the debuggers. You can update them by managing the drivers in Eclipse or visiting the websites above to download the drivers from the manufacturers.

2.9 PEMICRO OPENSDA BOOTLOADER/APPLICATION FIRMWARE

2.9.1 OVERVIEW

I will only detail how to update the OpenSDA firmware and debug application for PEmicro. The instructions will be much the same for other applications.

To download the latest PEMicro OpenSDA Debug Firmware, go to the follow website and download the "Firmware Apps" zip file. You will need

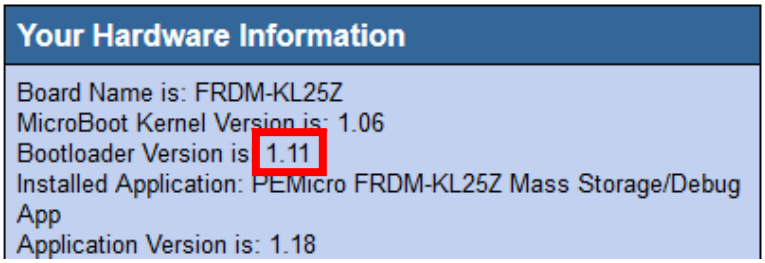
http://www.pemicro.com/opensda/

2.9.2 INSTALLATION STEPS

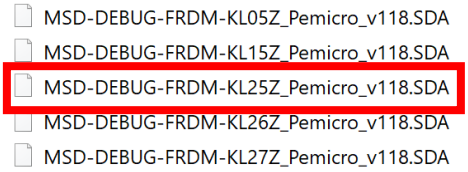
1. First confirm if your bootloader needs to be updated (needs to be version 1.11). The current version of bootloader software can be seen by entering bootloader mode and opening the SDA_INFO.HTM file.
2. To enter bootloader mode, hold down the reset button the target board and then power on the board by connecting it via USB to the OpenSDA port. After the board has powered on, you can release the reset button.

NOTE: Be sure you have corrected the Windows indexing issue before connecting the board.

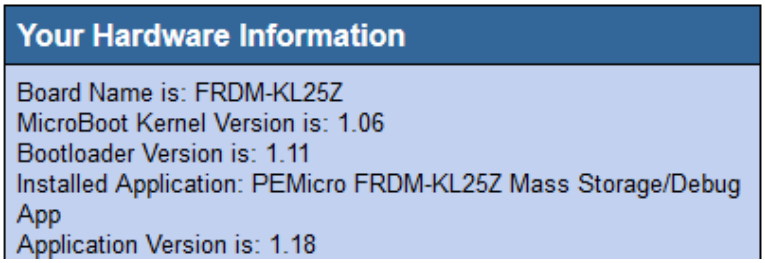
3. Make sure that your bootloader is version 1.11. If it is not, follow the steps below.



4. To download the latest PEMicro OpenSDA Debug Firmware, go to the follow website and download the "Firmware Apps" zip file. http://www.pemicro.com/opensda/
5. Extract the zip file you downloaded and extract the zip file contained in that directory that has the BOOTUPDATEAPP_Pemicro_v111.SDA file.
6. Re-enter bootloader mode. A new removable mass storage device should show up on your computer called "BOOTLOADER". Drag and drop BOOTUPDATEAPP_Pemicro_v111.SDA to the "BOOTLOADER" drive. Wait for the file to finish copying.
7. Once it has finished, remove power from the board. Then power up again normally (do not hold the reset button). Check SDA_INFO.HTM to confirm that the Bootloader Version is now 1.11.
8. Now it is time to update the debugger application. Go back to the Firmware Apps zip file that you unzipped from PEMicro. Find the correct debug application for your board.



9. You will update the debug application in the same way you updated the bootloader. Enter bootloader mode by holding down the reset button and connecting power to the OpenSDA USB port. Make sure your computer recognizes the board as a storage device called “BOOTLOADER”.
10. Drag and drop the debug application firmware onto the “BOOTLOADER” drive. Wait for it to copy, remove power, and power back on normally without pressing the reset button.
11. Re-open SDA_INFO.HTM and check the application version. Confirm that it matches the application version of the debug firmware you just loaded onto the board.

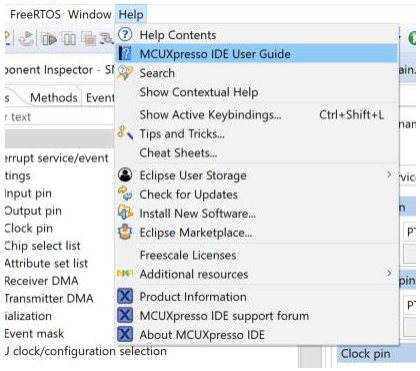


12. Now you board is already ready to be flashed and debugged!

3 USAGE

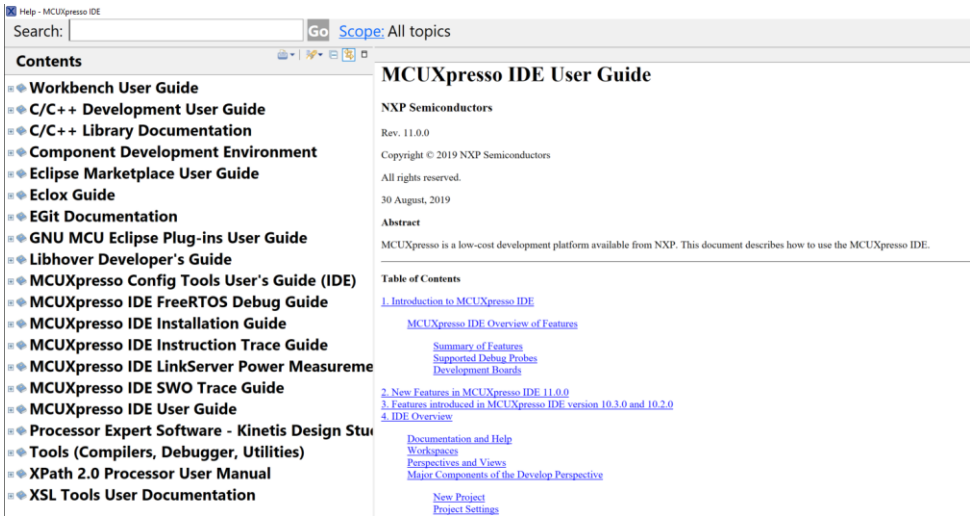
3.1 MCUXPRESSO

I will not provide an exhaustive overview of the MCUXpresso and the functionality provided as this would be redundant. There is an incredibly thorough User Guide embedded in the IDE which can be found as shown below.



However, in the following sections, I will highlight some of the most useful design and analysis features for our purposes. If you ever have any questions about MCUXpresso, I would always start with the User Guide then search the NXP forums (https://community.nxp.com/) for help.

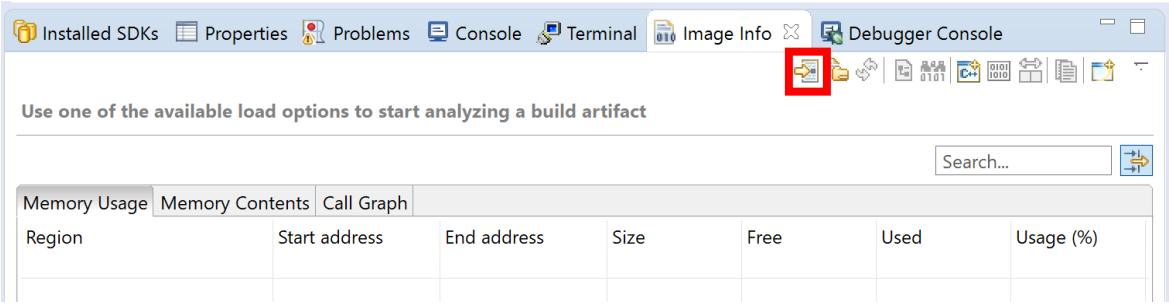
Please note that the User Guide will be updated with any additional plugins you install if they also have help documentation. See below the references to eclox, Processor Expert, GNU MCU Eclipse, and git documentation.



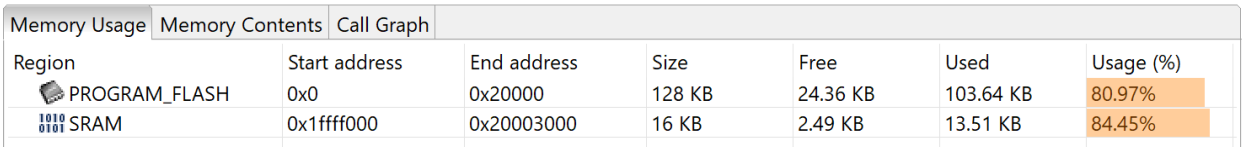
3.2 IMAGE INFO

Through MCUXpresso, you have the ability to inspect the .elf image file to see the memory usage, memory contents, and a call graph. Below are the instructions on how to view this information. In order to successfully view this image information, you must have link time optimizations disabled. Otherwise, you will not be able to view static stack size or the callgraph.

1. Go to the Image Info tab. Make sure you have successfully built the current project.
2. Click the “Load the build artifact associated with the current build configuration” button below.

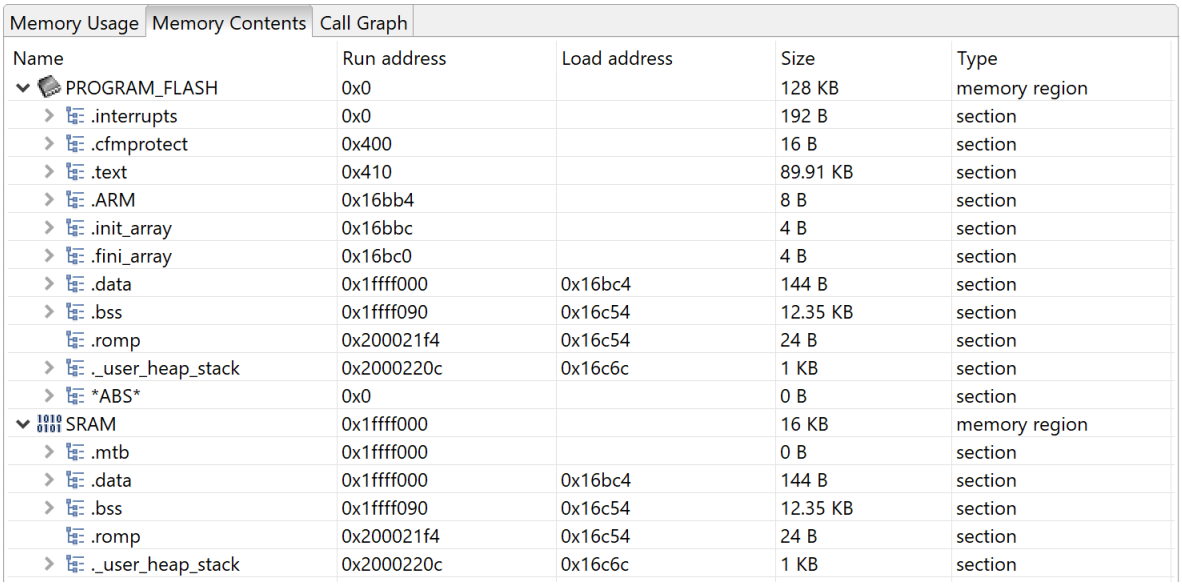


3. The image should have been successfully imported. You should now be able to view the ROM and RAM memory usage.



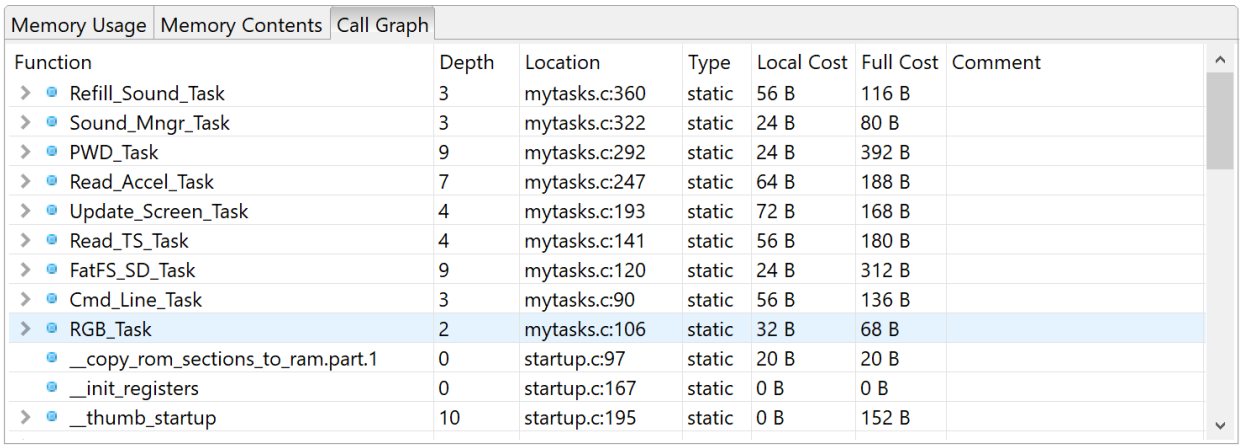
Region	Start address	End address	Size	Free	Used	Usage (%)
PROGRAM_FLASH	0x0	0x20000	128 KB	24.36 KB	103.64 KB	80.97%
SRAM	0x1ffff000	0x20003000	16 KB	2.49 KB	13.51 KB	84.45%

4. Click on the memory contents tab to view the size different objects and variables take in memory in addition. This is a great tool if you are running out of ROM or RAM and need to find some of the biggest memory hogs.



Name	Run address	Load address	Size	Type
PROGRAM_FLASH	0x0		128 KB	memory region
.interrupts	0x0		192 B	section
.cfmprotect	0x400		16 B	section
.text	0x410		89.91 KB	section
.ARM	0x16bb4		8 B	section
.init_array	0x16bbc		4 B	section
.fini_array	0x16bc0		4 B	section
.data	0x1ffff000	0x16bc4	144 B	section
.bss	0x1ffff090	0x16c54	12.35 KB	section
.romp	0x200021f4	0x16c54	24 B	section
._user_heap_stack	0x2000220c	0x16c6c	1 KB	section
ABS	0x0		0 B	section
SRAM	0x1ffff000		16 KB	memory region
.mtb	0x1ffff000		0 B	section
.data	0x1ffff000	0x16bc4	144 B	section
.bss	0x1ffff090	0x16c54	12.35 KB	section
.romp	0x200021f4	0x16c54	24 B	section
._user_heap_stack	0x2000220c	0x16c6c	1 KB	section

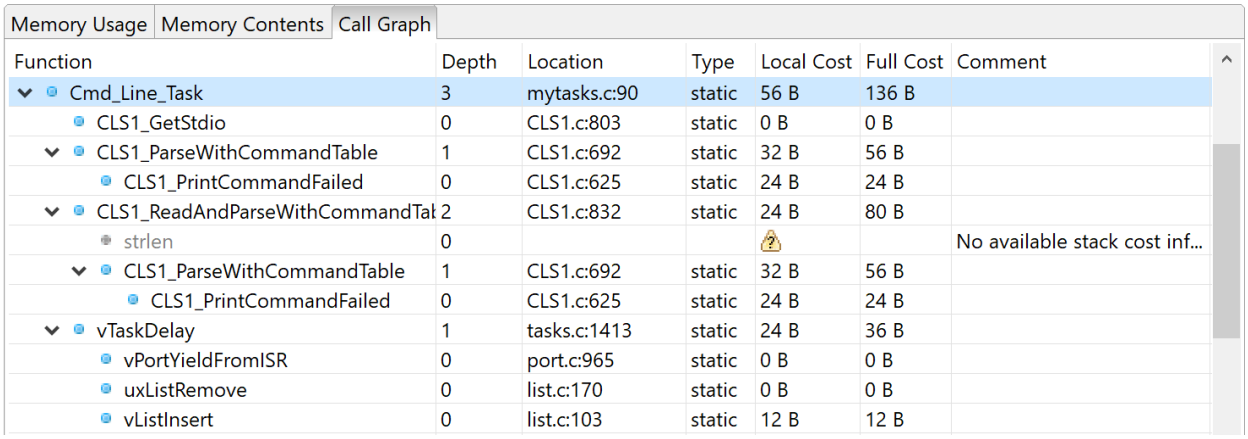
5. Finally, you can view a call graph along with the static stack depth of each function. The call graph is RTOS task aware and creates a hierarchical structure showing all callee functions nested underneath.



Memory Usage | Memory Contents | Call Graph

Function	Depth	Location	Type	Local Cost	Full Cost	Comment
Refill_Sound_Task	3	mytasks.c:360	static	56 B	116 B	
Sound_Mngr_Task	3	mytasks.c:322	static	24 B	80 B	
PWD_Task	9	mytasks.c:292	static	24 B	392 B	
Read_Accel_Task	7	mytasks.c:247	static	64 B	188 B	
Update_Screen_Task	4	mytasks.c:193	static	72 B	168 B	
Read_TS_Task	4	mytasks.c:141	static	56 B	180 B	
FatFS_SD_Task	9	mytasks.c:120	static	24 B	312 B	
Cmd_Line_Task	3	mytasks.c:90	static	56 B	136 B	
RGB_Task	2	mytasks.c:106	static	32 B	68 B	
__copy_rom_sections_to_ram.part.1	0	startup.c:97	static	20 B	20 B	
__init_registers	0	startup.c:167	static	0 B	0 B	
__thumb_startup	10	startup.c:195	static	0 B	152 B	

6. However, there is one important issue to be aware of. The call graph is not able to calculate the stack size of some functions (mainly library functions). As shown below, strlen() has a note “No available stack cost information.” It displays a question mark for the “Local Cost”. The issue is that this unknown stack size does not percolate up through the call graph. Instead of stating that Cmd_Line_Task() has some unknown stack size, it just states 136 B. It should be understood that the stack size is 136 B + some unknown size for strlen().

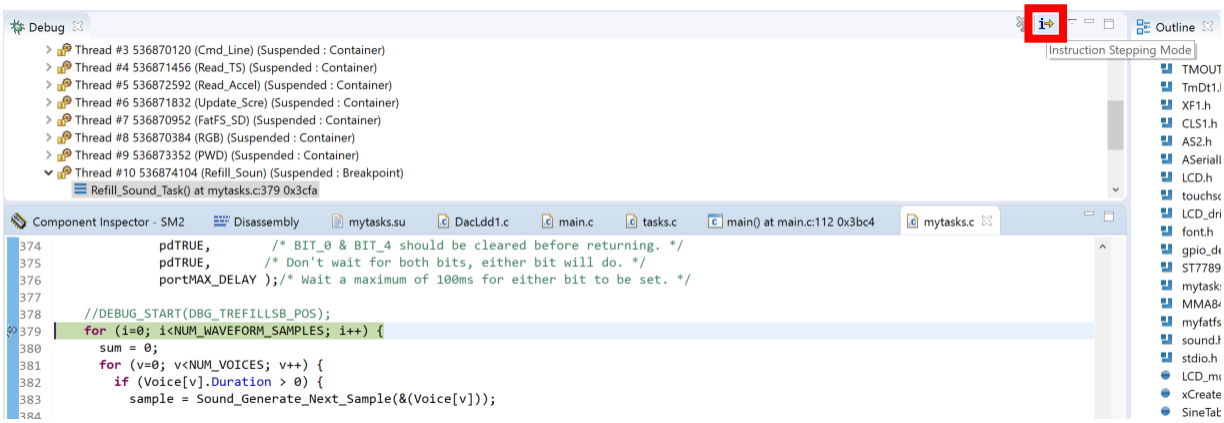


Memory Usage | Memory Contents | Call Graph

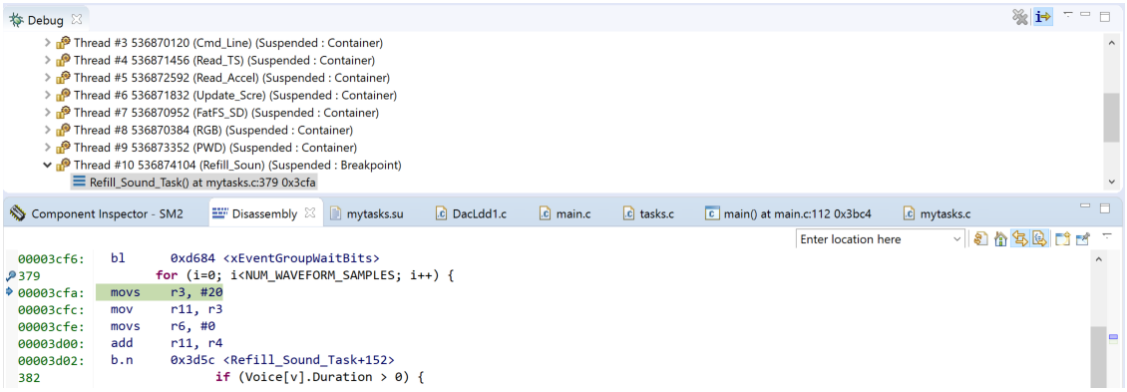
Function	Depth	Location	Type	Local Cost	Full Cost	Comment
Cmd_Line_Task	3	mytasks.c:90	static	56 B	136 B	
CLS1_GetStdio	0	CLS1.c:803	static	0 B	0 B	
CLS1_ParseWithCommandTable	1	CLS1.c:692	static	32 B	56 B	
CLS1_PrintCommandFailed	0	CLS1.c:625	static	24 B	24 B	
CLS1_ReadAndParseWithCommandTal	2	CLS1.c:832	static	24 B	80 B	
strlen	0			?		No available stack cost inf...
CLS1_ParseWithCommandTable	1	CLS1.c:692	static	32 B	56 B	
CLS1_PrintCommandFailed	0	CLS1.c:625	static	24 B	24 B	
vTaskDelay	1	tasks.c:1413	static	24 B	36 B	
vPortYieldFromISR	0	port.c:965	static	0 B	0 B	
uxListRemove	0	list.c:170	static	0 B	0 B	
vListInsert	0	list.c:103	static	12 B	12 B	

3.3 DISASSEMBLY VIEW

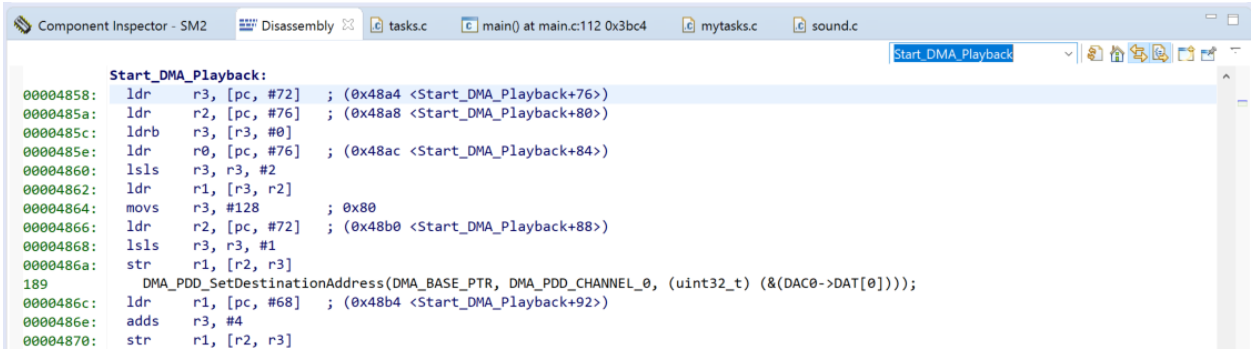
1. Sometimes for optimization or debugging, it may be helpful to view object code in a disassembly viewer. In order to do this, you must run your program in debugger mode.
2. Once in debugger mode, you can set a breakpoint where you would like to inspect the object code. Once the program hits the breakpoint, it will halt execution. At this point, you can hit the "Instruction Stepping Mode" button shown below.



3. Once you have pressed this button, it should bring the disassembly window into focus. It shows which instruction was presently executing. It also shows the associated source code for the object code instructions.

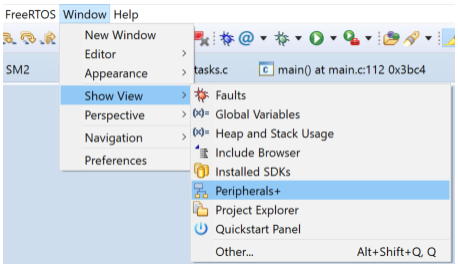


4. If you want to view the object code at a particular location in source code, you can search by source code in the search field in the disassembly window. It will navigate to the object code.

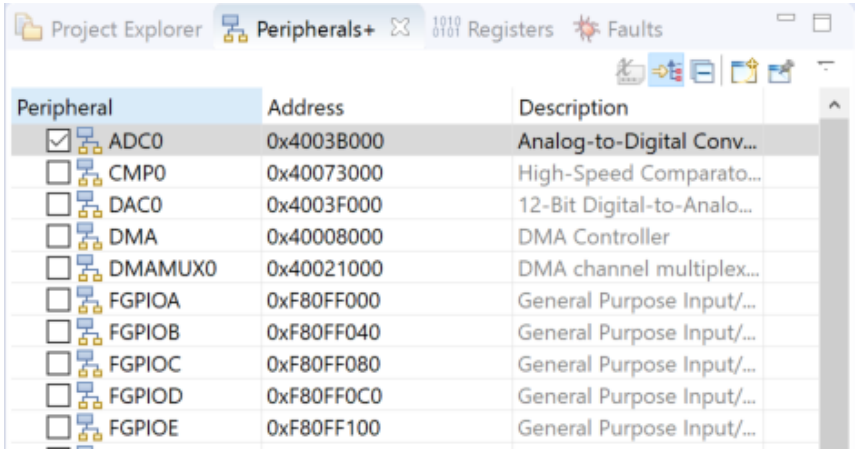


3.4 PERIPHERAL VIEWER

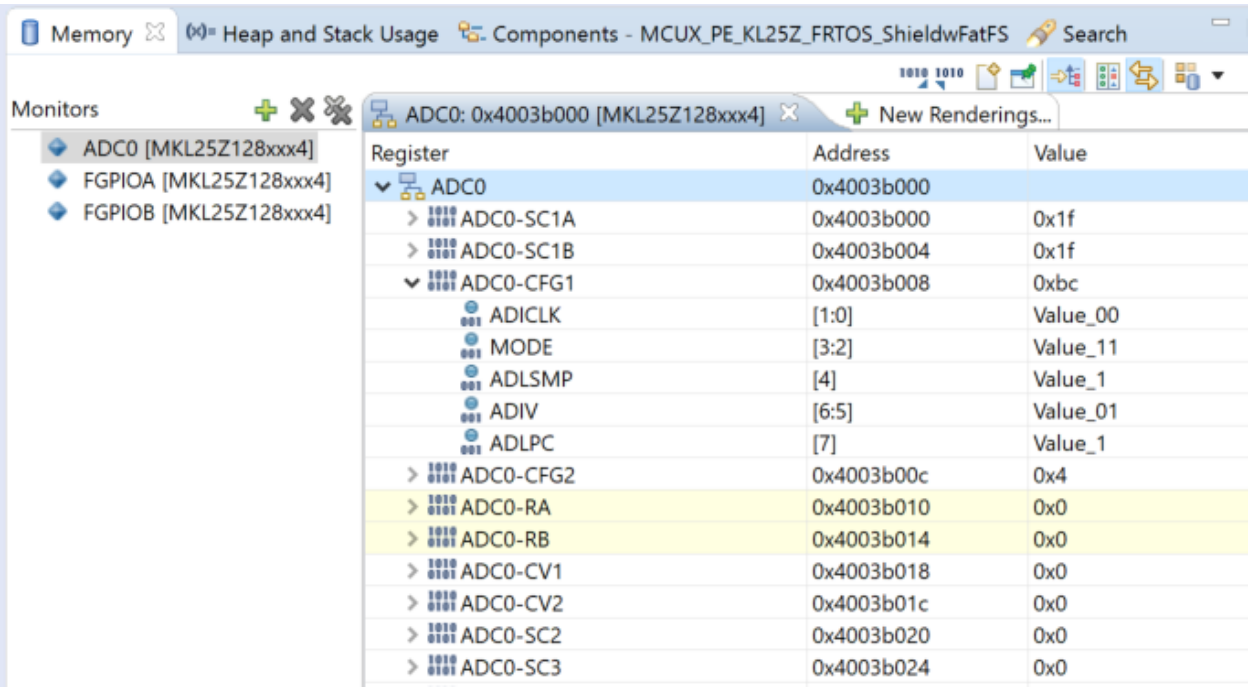
1. Sometimes for debugging purposes, it may be helpful to view the peripheral configuration settings at a particular point in execution.
2. In order to do this, you must run your program in debugger mode. You can view and even change the values of the peripheral configuration registers when execution is halted.
3. To do this, open the Peripheral Viewer as shown.



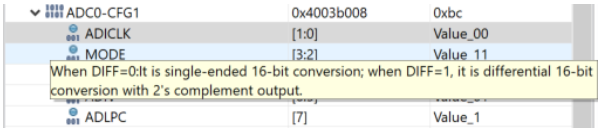
4. This will bring up the Peripherals+ window shown below. To view the configuration registers of a specific peripheral, check the box next to that peripheral.



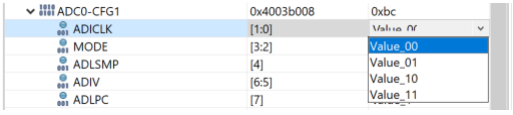
5. Once you check the box, a new window called “Memory” should come into view. In this window, you can view all of the registers within each of the peripherals. You can expand the registers to show the breakdown of the bits into the various fields and see their present values.



6. You can hover over each of the registers, fields, and values to view more information about what each of them are.

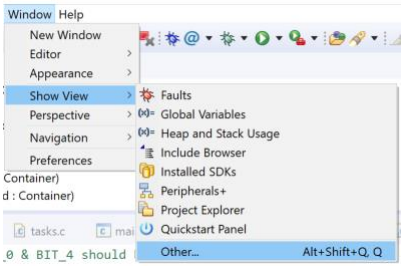


7. You can also modify the register values in debug mode and resume execution to see how the change in register values effect the program for debugging.

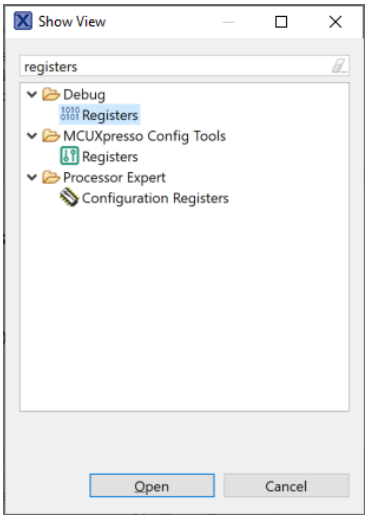


8. In addition, you can also export the register configurations for reference later or to even import them into the program later.

3.5 REGISTER VIEWER



1. Similarly for debugging purposes, it may be helpful to view the contents of the registers at a specific point in time during execution.



2. The registers window should already be open by default. If you cannot find it, go to Window→Show View→Other.

3. Once there, you can search “registers” or navigate to Debug→Registers and click “Open”.

4. That should bring up the registers view where you can view the values of all of the CPU registers, stack pointers, and the status registers as shown on the following page.

Project Explorer | Peripherals+ | Registers | Faults

Name	Value	Description
MKL25Z128xxx4 (co		**MCUX_PE_KL25Z_FRTOS_ShieldwFatFS.elf registers**
r0	0x1	Argument/Scratch Register 1
r1	0x0	Argument/Scratch Register 2
r2	0x20001844	Argument/Scratch Register 3
r3	0x1	Argument/Scratch Register 4
r4	0x200017a4	Variable Register 1
r5	0xffffffffa5a5a5a5	Variable Register 2
r6	0xffffffffa5a5a5a5	Variable Register 3
r7	0xffffffffa5a5a5a5	Variable Register 4
r8	0x10000	Variable Register 5
r9	0x20001844	Variable Register 6
r10	0x1ffff2fc	Variable Register 7
r11	0xffffffffa5a5a5a5	Variable Register 8
r12	0xffffffffa5a5a5a5	Intra-Procedure-Call Scratch Register
sp	0x20000c30 <ucHeap+5384>	Stack Pointer (r13)
lr	0xd705	Link Register (r14)
pc	0x3cfa <Refill_Sound_Task+54>	Program Counter (r15)
xpsr	0x1000000	Program Status Register
msp	0x20002fc8	Main Stack Pointer
psp	0x20000c30	Process Stack Pointer
primask	0 '\000'	Interrupt/Exception Mask Register
basepri	0 '\000'	Interrupt/Exception Mask Register
faultmask	0 '\000'	Interrupt/Exception Mask Register
control	2 '\002'	Control Register
Status Registers		**Status Registers for cortex-m0plus**
apsr	nzcvq	Application Program Status Register
ipsr	no fault	Interrupt Program Status Register
epsr	T	Execution Program Status Register

3.6 PROCESSOR EXPERT

Processor Expert is a source code generation tool used to create and configure different software components. The components can represent a hardware abstraction, a peripheral driver, or a specific algorithm. Once added, they can be in an intuitive GUI to generate configured source code. The source code oftentimes comes with many different macros and functions that can be used as an API to utilize that specific component. Examples of Processor Expert components include timer, ADC, DMA, ring buffer, I2C, RTOS, etc. The tool is incredibly user friendly and is great for teaching embedded systems capabilities. It gives a high-level overview of the many different peripherals and functionalities that can be used as well as giving you a lot of control over low-level configurations.

There is a lot of very useful documentation provided with the installation of Processor Expert. The User Guide shows you how to setup a project, add components, configure components, and generate code. In addition to this documentation, you can right click on any component and select “Help on Component”. A window will pop up showing you an overview of the components, a description of the different properties that can be configured, an overview of all the methods and events, and finally any additional application notes. If you still have questions about actually implementing Processor Expert projects, you can view the project and the website below. The shield project has a number of different Processor Expert components and can be a good example to pull up alongside your project for examples in how to configure things. If you still have questions, Erich Styger’s blog MCU on Eclipse and his Github contains many more examples on Processor Expert projects.

Project repository - https://github.com/mjdargen/MCUX_PE_KL25Z_FRTOS_ShieldwFatFS

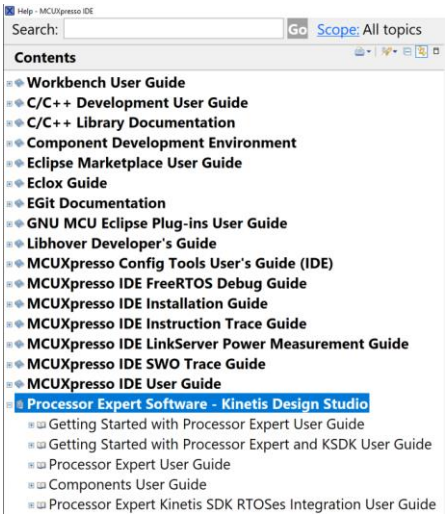
Project repository’s website - https://mjdargen.github.io/MCUX_PE_KL25Z_FRTOS_ShieldwFatFS

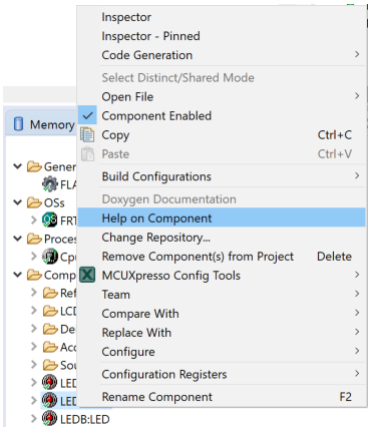
MCU on Eclipse

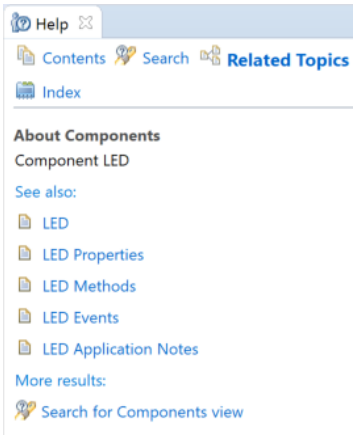
https://mcuoneclipse.com/category/processor-expert/

https://mcuoneclipse.com/tag/processor-expert/

https://github.com/ErichStyger/mcuoneclipse/



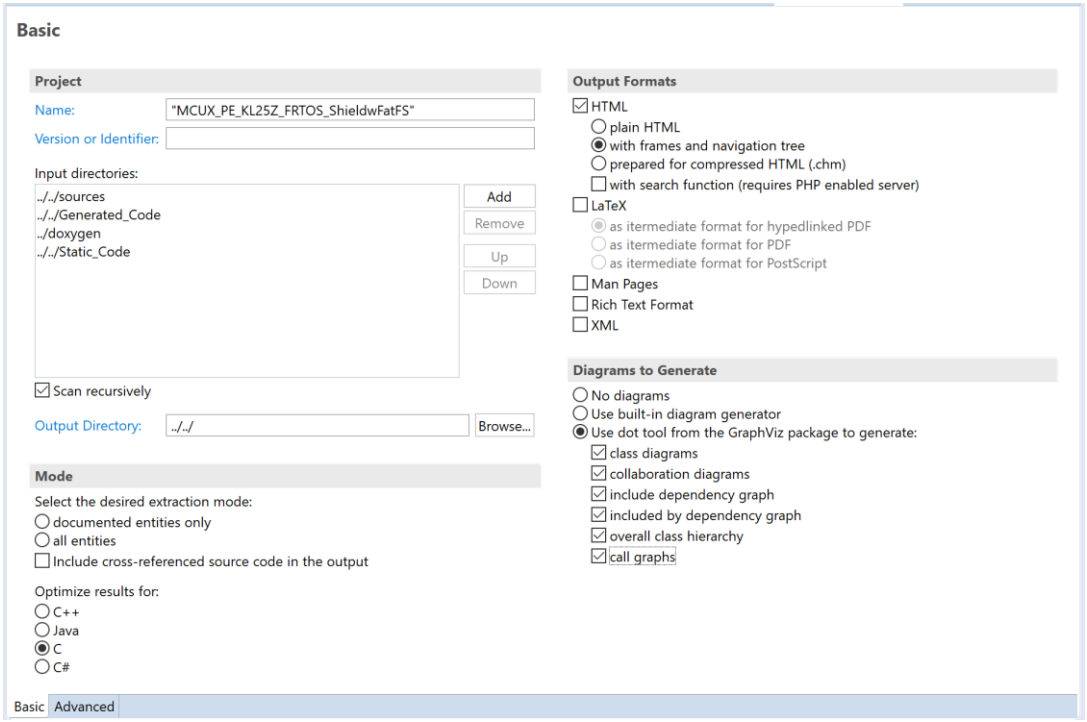




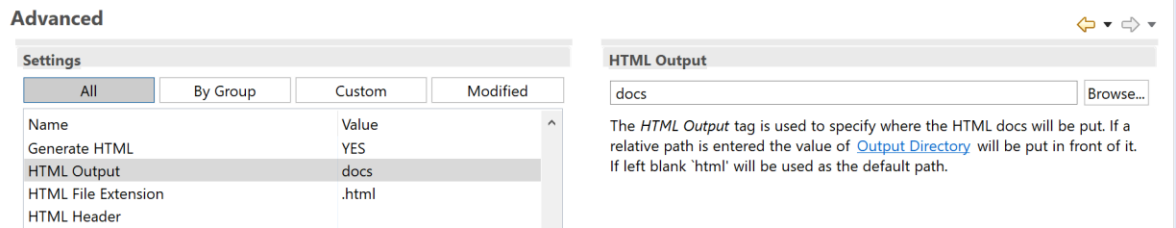
3.7 DOXYGEN WEBPAGE CREATION

If you are hosting your Processor Expert MCUXpresso project on Github, you can easily configure it to display a nice webpage detailing your project and the Processor Expert configurations using Doxygen. You will need to modify the pex.doxyfile and change the settings of your Github repository as shown below. A sample webpage is here: https://mjdargen.github.io/MCUX_PE_KL25Z_FRTOS_ShieldwFatFS

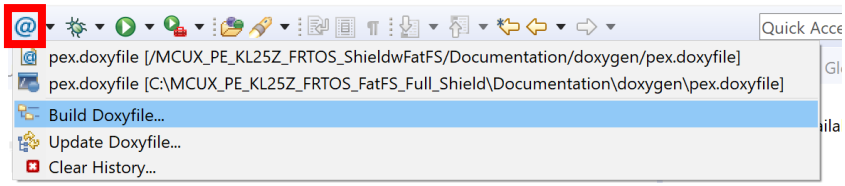
1. The doxygen file created by Processor Expert is located at "<ProjDirPath>/Documentation/…doxygen/pex.doxyfile". Double-click on the file to open the doxyfile editor.
2. Set the "Output Directory:" field to "../../" or wherever your root directory for Github is. This field is in reference to where the doxyfile is. So "../../" directs the program to the root project directory which is also my root Github directory.
3. Change the Output Formats so that it only outputs html.
4. Your basic settings should appear as below. (You can modify diagrams as you prefer.)



5. After configuring the basic settings, click on the Advanced tab in the bottom left corner of the doxyfile editor to switch to the advanced settings.
6. Select the HTML Output setting and change the field to say "docs". This will put all the html files into "../../docs" which creates a folder called docs in your root project directory. In order to successfully display these files as a website on Github, all of your html files need to be in a directory called "docs" located in the root folder of your Github repository.



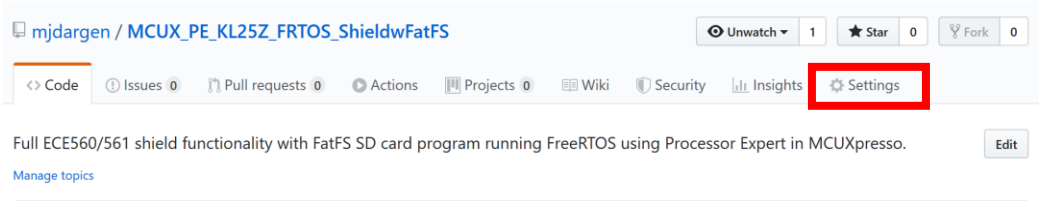
7. Once you have configured these settings, save your doxyfile. When your doxyfile has saved, click the blue @ icon and select “Build Doxyfile”. In the case the button is not visible, go to “Window →Customize perspective→Commands” and check Doxygen in “Available command groups”.



8. Once the doxygen files have been created, you can push your repository to Github.

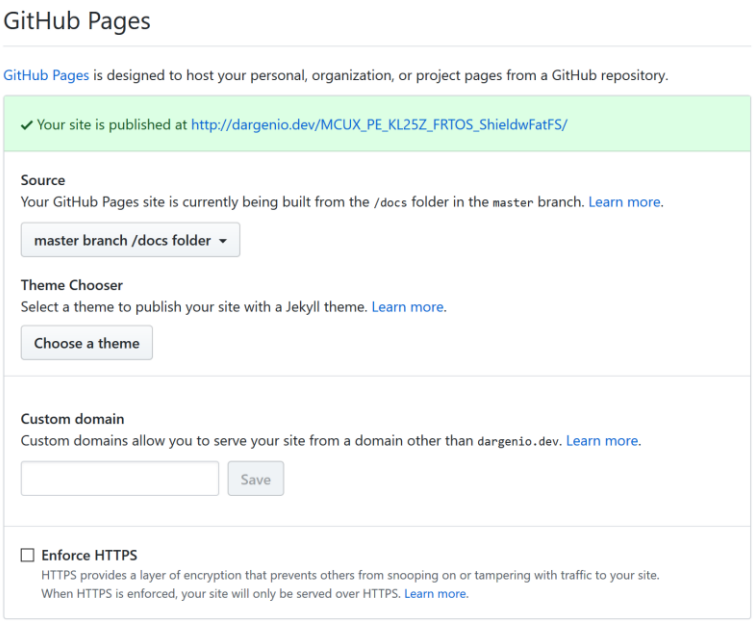
NOTE: Anytime you hit “Generate Code” for Processor Expert, it will overwrite your doxyfile. You will have to modify the doxyfile as shown above anytime you regenerate code if you wish to publish your project with a webpage on Github.

9. To configure your Github repository to publish the doxygen output as a website, go to your repository and select settings.

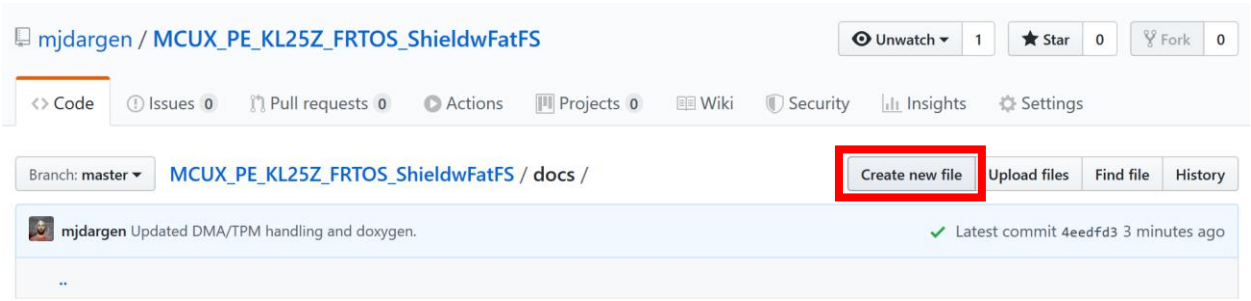


10. Scroll down to the Github Pages section. Set your repository to publish a website by selecting “master branch /docs folder” as the source. After applying the settings and pushing the repository, it should publish the website at “<github username>.github.io/<repository>”. (Mine shows as being published elsewhere because I host my personal website on Github and point it to a custom domain name, so it adopts this domain name.)

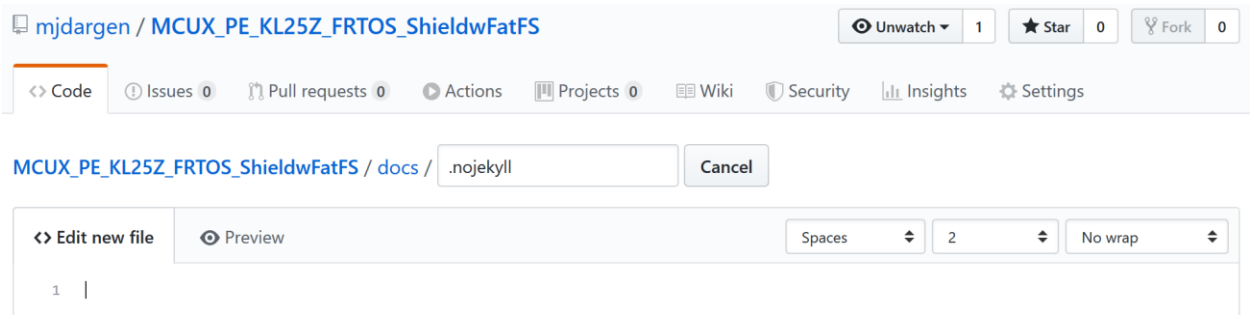
Note: It can take 15 minutes or so for your website to show as published. If you have a previous version of the doxygen webpage cached by your browser, it may not show up properly. You may have to clear your cache or use another browser/device to confirm the changes. Or you can try to view your website through one of the many available online web proxies.



11. There is one issue with the new Github pages publisher. Github interprets html files that start with an underscore ("_") as Jekyll files (Jekyll is a blog-aware, static site generator that Github has adopted). Many of the html pages generated by Processor Expert begin with an underscore.
12. To fix this issue, you just need to create an empty file called ".nojekyll" in your "docs" directory in your Github repository. This file tells Github to bypass Jekyll. Github will now see all of your webpages beginning with underscores as valid webpages.
13. Navigate to your "docs" directory and click "Create new file".



14. Name the file ".nojekyll", leave the file blank, and click "Commit new file".

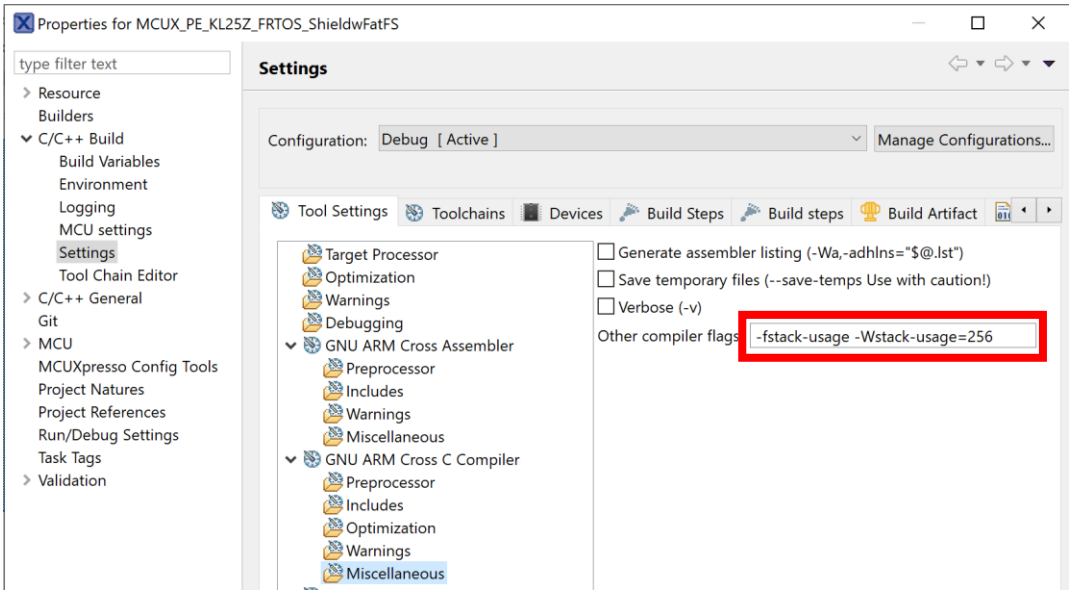


15. Your project should now have a website at "<github username>.github.io/<repository>".

3.8 STATIC STACK DEPTH ANALYSIS

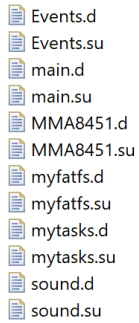
In addition to the stack depth information provided by MCUXpresso detailed in the Image Info Section 3.2, GCC provides static stack depth information.

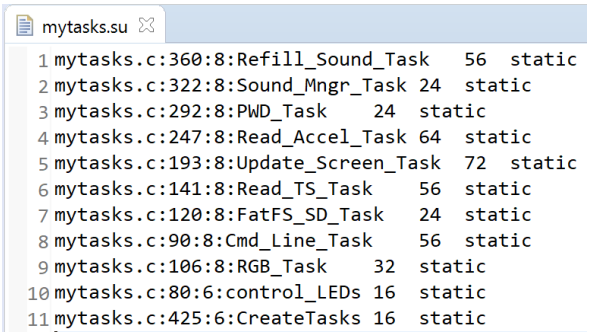
1. To turn on the static stack depth analysis, you must add the “-fstack-usage” flag. Go to the miscellaneous C compiler settings and add “-fstack-usage” to the “other compiler flags” field.
2. You can also set an additional compiler “-Wstack-usage=256” flag which gives a warning when a function has a stack depth over a specific threshold. The number following the equals sign is the threshold number of bytes. In this instance if the stack exceeds 256 bytes, the compiler will throw a warning stating the offending function.



3. When you build your project, there will be a .su file for every source file in your project in the target debug folder alongside the .d file. If you open the .su file, you will see a list of all of the functions in that source file. It lists the following information:

Source_file.c:line:column:function_name Stack Size in Bytes Whether static/dynamic/bounded



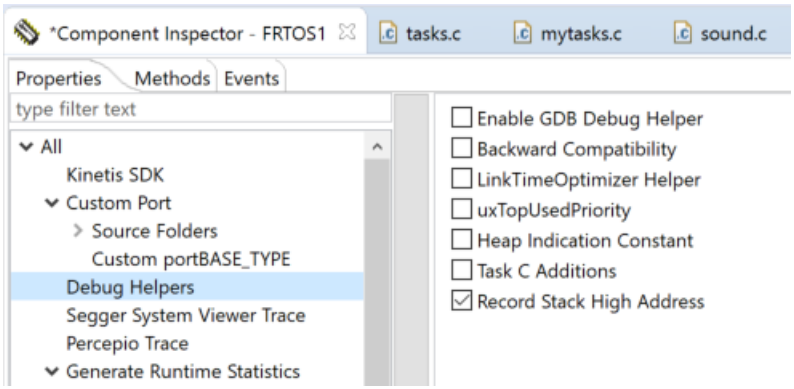


4. Below shows an example compiler warning caused by a function with a stack over 256 B.

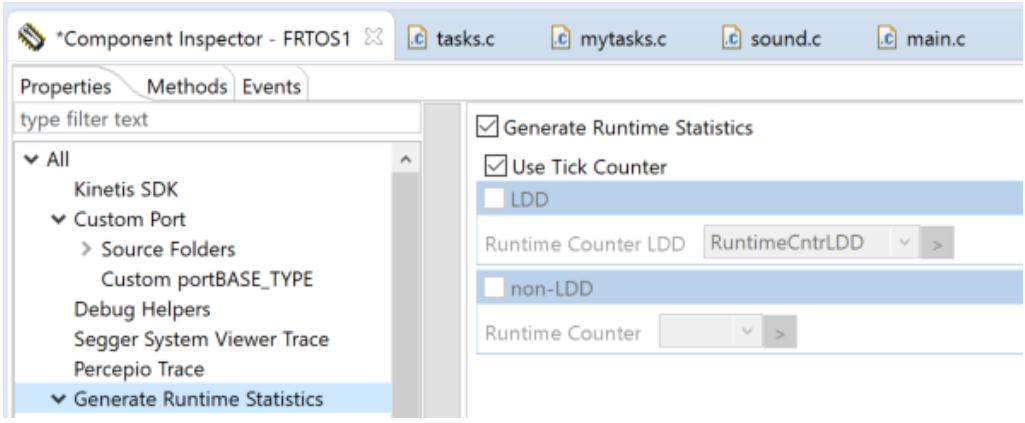
```
Building file: ../Generated_Code/FAT1.c
Invoking: GNU ARM Cross C Compiler
arm-none-eabi-gcc -mcpu=cortex-m0plus -mthumb -O3 -fmessage-length=0 -fsigned-char -ffunction-sections -fdata-sections  -g3
../Generated_Code/FAT1.c: In function 'FAT1_CreateFile.part.11':
../Generated_Code/FAT1.c:1965:9: warning: stack usage is 584 bytes [-Wstack-usage=]
 uint8_t FAT1_CreateFile(const uint8_t *fileName, const CLS1_StdIOType *io)
         ^~~~~~~~~~~~~~~
```

3.9 DYNAMIC STACK DEPTH ANALYSIS & TASK PROFILING

1. By using MCUXpresso with a FreeRTOS program, we are able to do profiling and dynamic stack depth analysis. This can be incredibly helpful for debugging and analysis purposes.
2. In order to do this, there are some configurations that must be made in the FreeRTOS Processor Expert component, so the debugger has appropriate RTOS awareness.
3. The first is to go to the Debug Helpers tab and check “Record Stack High Address”. This allows the debugger to make a high-water mark at the largest memory address for the stack so we know the maximum size the stack grew to during execution.

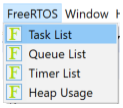


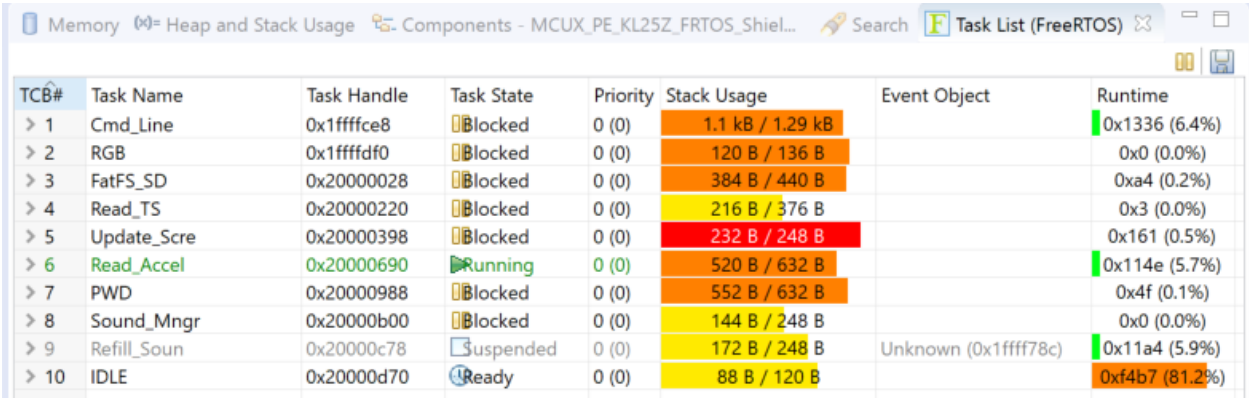
4. In order to enable profiling, we need to go to the Generate Runtime Statistics tab and check the “Generate Runtime Statistics” box and the “Use Tick Counter” box. This allows for profiling using the FreeRTOS systick timer. We will now know how many systicks occur in each thread.



5. To use the dynamic stack size analysis tool and profiler, we need to run the program in debug mode. (It can also be done without the debugger through the command line interface. More information about that will be provided later.)
6. Once in debug mode, run the program for a while. Be sure to manipulate whatever inputs are going to the microcontroller to model real-case and worst-case scenarios. This will give you more accurate information on stack size and profiling and help you design a more robust embedded system. Remember that the processor is recording a high-water mark so there is no need to pause the program at a particular point.
7. After you are satisfied with the rigorousness of your program execution, you can pause the debugger to view the profiler and stack size analysis results.

8. Open the task list view by going to FreeRTOS→Task List.
9. A new window should come into focus showing a list of all the tasks with their name, memory address, state, priority, stack usage information, event objects, and runtime statistics.





TCB#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Object	Runtime
1	Cmd_Line	0x1ffffce8	Blocked	0 (0)	1.1 kB / 1.29 kB		0x1336 (6.4%)
2	RGB	0x1ffffdf0	Blocked	0 (0)	120 B / 136 B		0x0 (0.0%)
3	FatFS_SD	0x20000028	Blocked	0 (0)	384 B / 440 B		0xa4 (0.2%)
4	Read_TS	0x20000220	Blocked	0 (0)	216 B / 376 B		0x3 (0.0%)
5	Update_Scre	0x20000398	Blocked	0 (0)	232 B / 248 B		0x161 (0.5%)
6	Read_Accel	0x20000690	Running	0 (0)	520 B / 632 B		0x114e (5.7%)
7	PWD	0x20000988	Blocked	0 (0)	552 B / 632 B		0x4f (0.1%)
8	Sound_Mngr	0x20000b00	Blocked	0 (0)	144 B / 248 B		0x0 (0.0%)
9	Refill_Soun	0x20000c78	Suspended	0 (0)	172 B / 248 B	Unknown (0x1ffff78c)	0x11a4 (5.9%)
10	IDLE	0x20000d70	Ready	0 (0)	88 B / 120 B		0xf4b7 (81.2%)

10. As you can see, the stack size that was allocated for each task has been pretty heavily used. However, the processor is still not that heavily used with 81.2% of the systicks occurring in the IDLE task. This is not always truly indicative of actual processor usage as the systicks occur every 1 ms and may sync up with certain tasks and screw the statistics. Nevertheless, it gives us a pretty good idea of what tasks are using the processor the most.
11. We can expand a specific task to see more information about the stack.

6	Read_Accel	0x20000690	Running	0 (0)	520 B / 632 B		0x114e (5.7%)
	Task Number:	0x0					
	Stack Base:	0x20000408					
	Stack Top:	0x20000680					
	Stack High Water Mark:	0x20000478					