Getting Started with TivaWare™ for C Series

User's Guide

Literature Number: SPMU373A
MARCH 2021 – REVISED AUGUST 2022

Table of Contents

List of Figures

Preface

Read This First

About This Manual

Texas Instruments TivaWare™ software for Tiva-C Software Development Kit (SDK) provides users with device drivers, header files, application examples, utilities, and more, in order to speed development with the TM4C family of microcontroller devices. This User's Guide provides an overview of what is provided with the TivaWare SDK and covers fundamental topics including how to leverage TivaWare example projects, how to debug TivaWare libraries, and general software best practices to avoid common embedded programming errors. This guide uses TI's Code Composer Studio™ (CCS) Integrated Development Environment (IDE) for all examples, but the same concepts apply for other IDE's.

Glossary

TI Glossary	This glossary lists and explains terms, acronyms, and definitions.

Related Documentation From Texas Instruments

For a complete listing of related documentation and development-support tools for these devices, visit the Texas Instruments website at http://www.ti.com.

Support Resources

TI E2E™ support forums are an engineer's go-to source for fast, verified answers and design help — straight from the experts. Search existing answers or ask your own question to get the quick design help you need.

Linked content is provided "AS IS" by the respective contributors. They do not constitute TI specifications and do not necessarily reflect TI's views; see TI's Terms of Use.

Trademarks

TivaWare™, Code Composer Studio™, and TI E2E™ are trademarks of Texas Instruments.
Arm® is a registered trademark of Arm Limited (or its subsidiaries).
All trademarks are the property of their respective owners.

This page intentionally left blank.

Chapter 1

Introduction to TivaWare SDK

This document references TivaWare version 2.2.0.295 for all examples, but the information presented applies to both newer and older versions of TivaWare. Throughout this document, all directory paths mentioned are based on starting at the TivaWare installation folder, TivaWare_C_Series-2.2.0.295.

1.1 TivaWare SDK Folder Breakdown

This section covers a high-level walkthrough of what is provided within the TivaWare SDK. It additionally links various folders with the provided TivaWare documentation that further describes what is offered from the SDK.

In the base directory, there are thirteen folders provided along with a few files. The twelve highlighted folders in Figure 1-1 are where all the relevant collateral exists. The only files of note in the base directory are the three .txt licensing files. The remaining files in the base directory as well as the *.metadata* folder are build artifacts and TI Resource Explorer information and can be disregarded.

The *boot_loader* folder contains the source code required to execute the TivaWare Flash boot loader. Full details of the boot loader functionality are covered in the *TivaWare™ Boot Loader User's Guide (SW-TM4C-BOOTLDR-UG)*, including the differences between ROM and Flash boot loaders.

The *docs* folder contains all technical documentation pertaining to TivaWare examples, libraries, and utilities. It also includes the official TivaWare Release Notes that highlight changes made to TivaWare across each version of the SDK (*TivaWare™ for C Series Release Notes (SW-TM4C-RLN)*).

The *driverlib* folder contains the TivaWare Driver Library (DriverLib) source code that allows users to leverage TI validated functions. DriverLib functions provide all programmers with a simple means to configure the device and control peripherals without needing to operate at a register level. Full details of the DriverLib source code are covered in the *TivaWare™ Peripheral Driver Library User's Guide (SW-TM4C-DRL-UG)*.

The *examples* folder contains all the example projects provided with TivaWare. This is covered in more detail in Chapter 2.

The *inc* folder contains the device header files for each TM4C device as well as the hardware header files. The device header files provide defines for all registers, offsets, and bit fields for a specific device. The generic hardware header files are the .h files that start with the 'hw_' prefix and they contain the definitions for all register offsets and bit fields. TivaWare libraries leverage the generic hardware header files in order to ensure that all library functions are device agnostic, but for specific applications these can be replaced by a specific device header file.

> *If a code file uses #include for both a device header file and a generic hardware header file, there will be compiler errors for overlapped #define's. Only one or the other can be used in a code file.*

The *grlib* folder contains the TivaWare Graphics Library source code that provides a set of graphics primitives and a widget set for creating graphical user interfaces on various LCD displays. Graphics primitives are used to draw defined shapes and patterns. Widgets are used to draw various user interface elements. Additionally, all graphics applications will need a display driver, but that is handled at an application level and is provided within the TivaWare examples folder. Full details of the Graphics Library source code are covered in the *TivaWare™ Graphics Library User's Guide (SW-TM4C-GRL-UG)*.

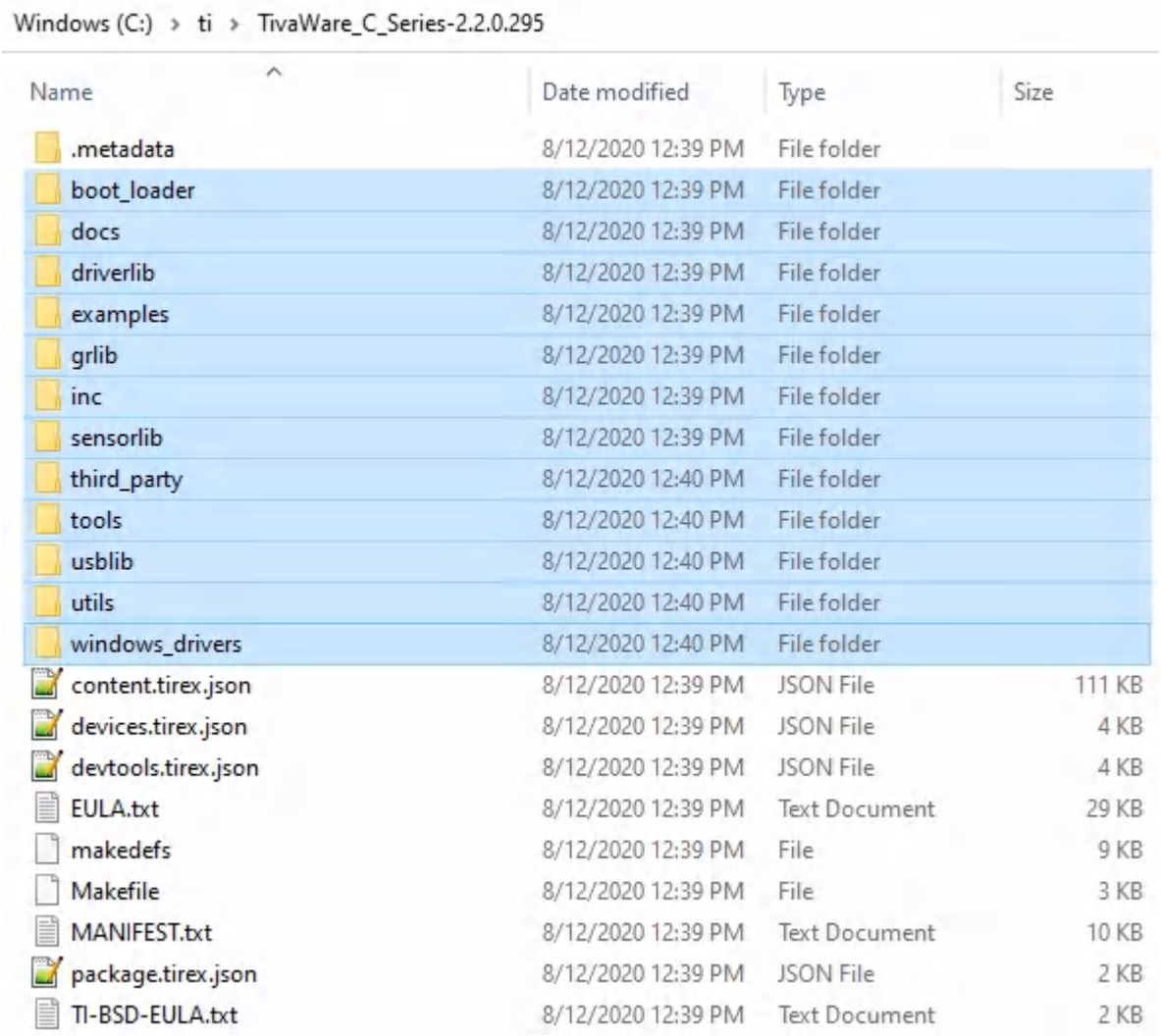

Figure 1-1. Complete TivaWare SDK Install Directory

The *sensorlib* folder contains the TivaWare Sensor Library source code that is provided for a variety of TivaWare supported sensors mainly from the original BOOSTXL-SENSHUB BoosterPack. Full details of the offerings are covered in the *TivaWare™ Sensor Library User's Guide (SW-TM4C-SENSORLIB-UG)*.

The *third_party* folder contains various open source resources from third party sources that are leveraged to support TivaWare examples. These include files for Exosite's cloud IoT demonstration (*IoT Demo Using EK-TM4C1294XL LaunchPad Application Report*), a generic FAT file system, FatFs, for SD cards, FreeRTOS support, and various Ethernet protocol related applications including lwIP 1.4.1.

The *tools* folder contains the TivaWare Host Tools that are run on the development system, not on the TM4C devices. These tools are provided to assist in the development of firmware for TM4C applications. Full details of the offerings are covered in the Tools User's Guide (SW-TM4C-TOOLS-UG).

The *usblib* folder contains the TivaWare USB Library source code which provides users with basic USB functionality for Device, Host, and On-The-Go modes. The USB Library provides descriptors for device enumeration, endpoint management, and class specific modules. Full details of the USB Library source code are covered in the *TivaWare™ USB Library User's Guide (SW-TM4C-USBL-UG)*.

The *utils* folder contains various TI-developed utilities to aid in the creation of application. These utilities are device agnostic and can be used to help build more advanced user applications. Full details of the offerings are covered in the Utilities Library User's Guide (SW-TM4C-UTILS-UG).

The *windows_drivers* folder contains signed Windows USB device drivers which are required for all TivaWare USB device examples.

This page intentionally left blank.

Chapter 2

TivaWare Example Projects

There are three sub-folders within *examples* and this section will cover the *boards* and *peripherals* folders in added detail. The *project* folder only contains a basic TivaWare project based on the TM4C123GH6PM to start software development on IDE's such as Code Composer Studio. A similar project is also provided within the boards folders for both TM4C123x and TM4C129x MCUs.

2.1 TivaWare Development Board Examples

Within the *examples/boards* folder are dedicated folders for each offered TM4C LaunchPad Evaluation Kit, the TM4C129x Development Kit, and for combinations with various BoosterPack's that can interface to those boards.

The folders for each standalone LaunchPad or Development Kit includes a *project0* example. That example is recommended to be used as the baseline code project for any fresh TivaWare development efforts.

Each example project includes project files for the following Integrated Development Environments (IDE): Code Composer Studio (Texas Instruments), IAR Embedded Workbench (IAR Systems), and µVision® IDE (Keil).

2.2 TivaWare Peripheral Examples

The *examples\peripherals* TivaWare folder contains basic example code for the most common TM4C peripherals. These examples are delivered purely as .c source files without corresponding IDE-specific projects. These files are configured to support both TM4C123x and TM4C129x devices.

To leverage these examples, first import the *project0* example for the desired EVM, rename the project in the IDE, and then copy and paste the entire contents of the peripheral source code file to replace the full contents of *project0.c*.

By using *project0* for the correct board, the predefined symbols for TM4C123x or TM4C129x will be automatically loaded into the project for the given IDE.

If the project uses additional utilities like **UARTprintf** then additional files may need to be linked. See Section 3.1 for details on how to add additional file links to a TivaWare project.

2.3 How to Import an Example Project into CCS

Code Composer Studio has an easy-to-use import feature that can be found under Project → Import CCS Projects. To import any TivaWare project click the 'Browse' button with the 'Select search-directory' option. This feature will look for **all** CCS projects in a provided directory. Due to the number of examples projects offered within the TivaWare SDK, it is strongly encouraged to navigate to at least a specific board level if not to the exact project desired. Figure 2-1 shows the options when navigating to *examples\boards\ek-tm4c123gxl*.

It is possible to import multiple projects at once. Simply scroll through the list and select all projects that should be imported and click 'Finish'. All TivaWare example projects will automatically be copied into the workspace as a new project so the original project will be left untouched and can be re-imported at any time.

Because the projects are copied into the workspace, if a previously imported project is removed from the workspace in CCS but not deleted from the file system then an error may indicate that the project still exists in the workspace. In this situation, either re-import the project back into CCS from the workspace directory or

delete the project folder from the workspace directory and try to import the project from the TivaWare directory again.

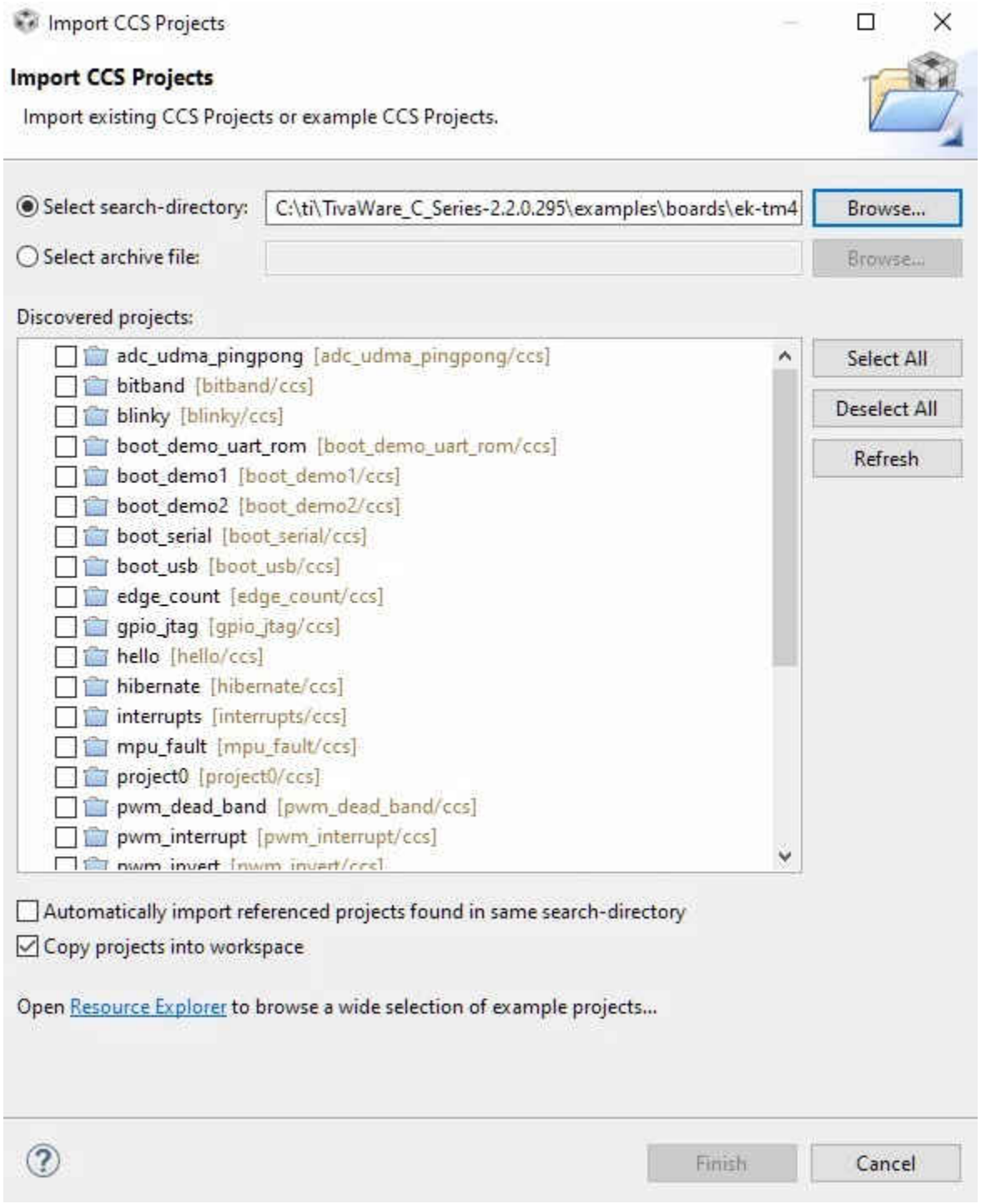

Figure 2-1. Example of the Import CCS Projects Utility

Chapter 3

Linking Files and Libraries into a TivaWare Project in Code Composer Studio

When expanding a TivaWare project to include additional files or libraries such as adding new utilities or including a library like *usblib* specific steps need to be taken to ensure the files and libraries are correctly added or else the project will return build errors. This section will cover how to avoid this common issue.

3.1 Linking Files in CCS

For this section, the *single_ended* peripheral examples under *examples/peripherals/adc* will be used as a reference.

Following the steps in Section 2.2, project0 for the EK-TM4C123GXL LaunchPad was used as the starter TivaWare project. The project name was renamed to 'single_ended', which allows project0 to be imported again in the future. After copying the source code into the *project0.c* file and trying to compile the project, the build errors seen in Figure 3-1 occur.

Figure 3-1. CCS Build Error - Unresolved Symbols

The 'unresolved symbol' error usually occurs when a function call is used without the proper .c source files added to the project. This usually occurs when a project uses files outside of a provided TivaWare library like DriverLib or UsbLib. If a header file is added with a #include from a TivaWare library directory then check to ensure the corresponding .lib file has been added to the project. If so, no further actions need to be taken. Section 3.2 will cover how to add a .lib file if it is missing from a project.

In this case, two UART related functions are causing the issue: **UARTprintf** and **UARTStdioConfig**. As this example is for an existing TivaWare example project, the quickest way to understand what file is missing would be to scroll up to the #include statements at the top of the code project. Upon doing so, the following lines related to UART can be found:

```
#include "driverlib/uart.h"
#include "utils/uartstdio.h"
```

As mentioned before, since *driverlib* files are already included with the .lib file, the *uartstdio.c* file would be the next file to check to see if it is missing. Looking through the list of files in the project indicates that there is no *uartstdio.c* file included in the project (Figure 3-2). Searching the contents of the *uartstdio.c* file in the *utils* directory of the TivaWare install will reveal that the missing functions are both inside of that file.

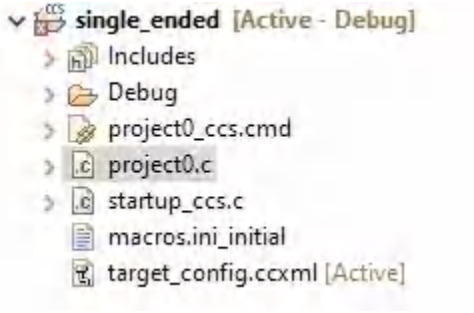

Figure 3-2. File List for the single_ended Project

Now that the file has been identified, it needs to be linked to the CCS project via the following steps:

1. Go to the CCS project and right click on the project name to bring up the options menu for the project (Figure 3-3).
2. Click "Add Files".

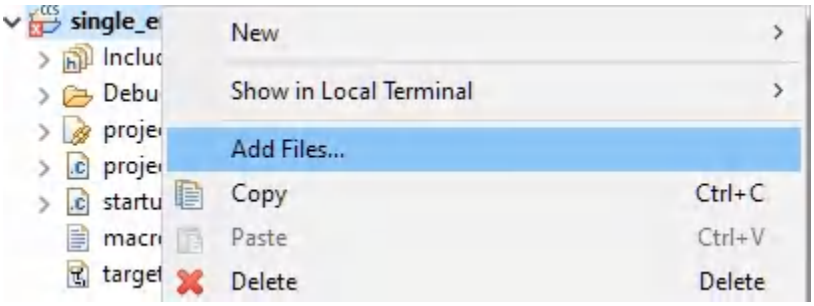

Figure 3-3. Add Files Option for CCS Project

3. This will bring up the file browser, and as identified before, the *uartstdio.c* file needs to be added from the *utils* directory. Navigate to that directory and select that file.

   It is possible to select multiple files by holding CTRL down and clicking each one that must be linked.

4. When adding the files, CCS will ask whether to copy or link the file(s). Select the 'Link to files' option (Figure 3-4). If adding another file from within the TivaWare SDK, it is also recommended to change the 'Create Link locations relative to:' path to 'SW_ROOT'. **SW_ROOT** is a TivaWare defined path for the base TivaWare SDK directory on the computer.

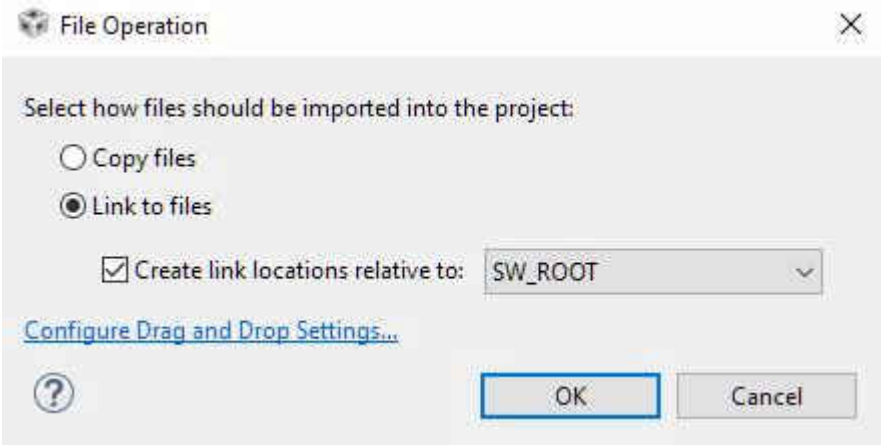

Figure 3-4. Recommended Link Location Path for Other TivaWare Files

The **SW_ROOT** definition can be found in the CCS Project Properties under Resources → Linked Resources → Path Variables tab view.

Now the *uartstdio.c* file should show up in the file list for the project (Figure 3-5) and attempting to re-build the project will now be successful!

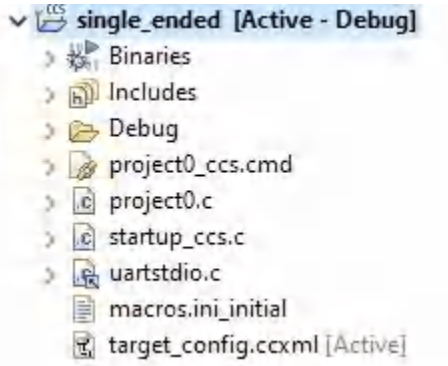

Figure 3-5. *uartstdio.c* Now Included in CCS Project

Another method to check if the process has been done correctly would be to go to the CCS Project Properties, navigate to the Resource → Linked Resources page and select the "Linked Resources" tab. Once there, look for the Variable Relative Location to be listed as seen in Figure 3-6.

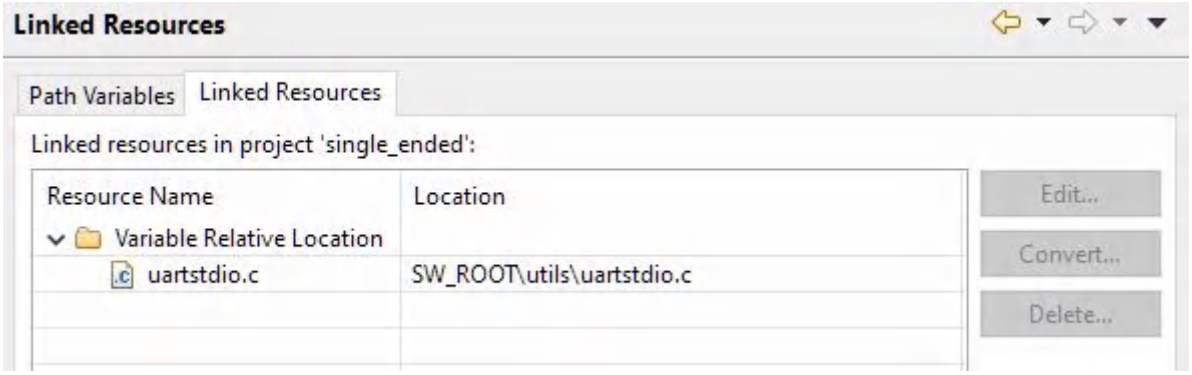

Figure 3-6. Linked Resources within CCS Project Properties

3.2 Linking Libraries in CCS

TivaWare example projects only include links for the TivaWare libraries relevant to the example project as provided. This means that while all projects will include a link to DriverLib, other libraries such as UsbLib or GrLib may not be linked to the project and a few steps need to be taken in order to link the library.

The library file that needs to be linked is the .lib file that is found within the library's folder in the TivaWare installation. For this example, the *usblib.lib* file will be used. The .lib file for a library is a build output from when the library is compiled and can be found under *usblib\ccs\Debug*.

In order to add a library to an existing project, go into the Project Properties and navigate to Build → ARM Linker → File Path Search. There will be a section for 'Include library file or command file as input'. Click the 'Add…' button seen circled in Figure 3-7.

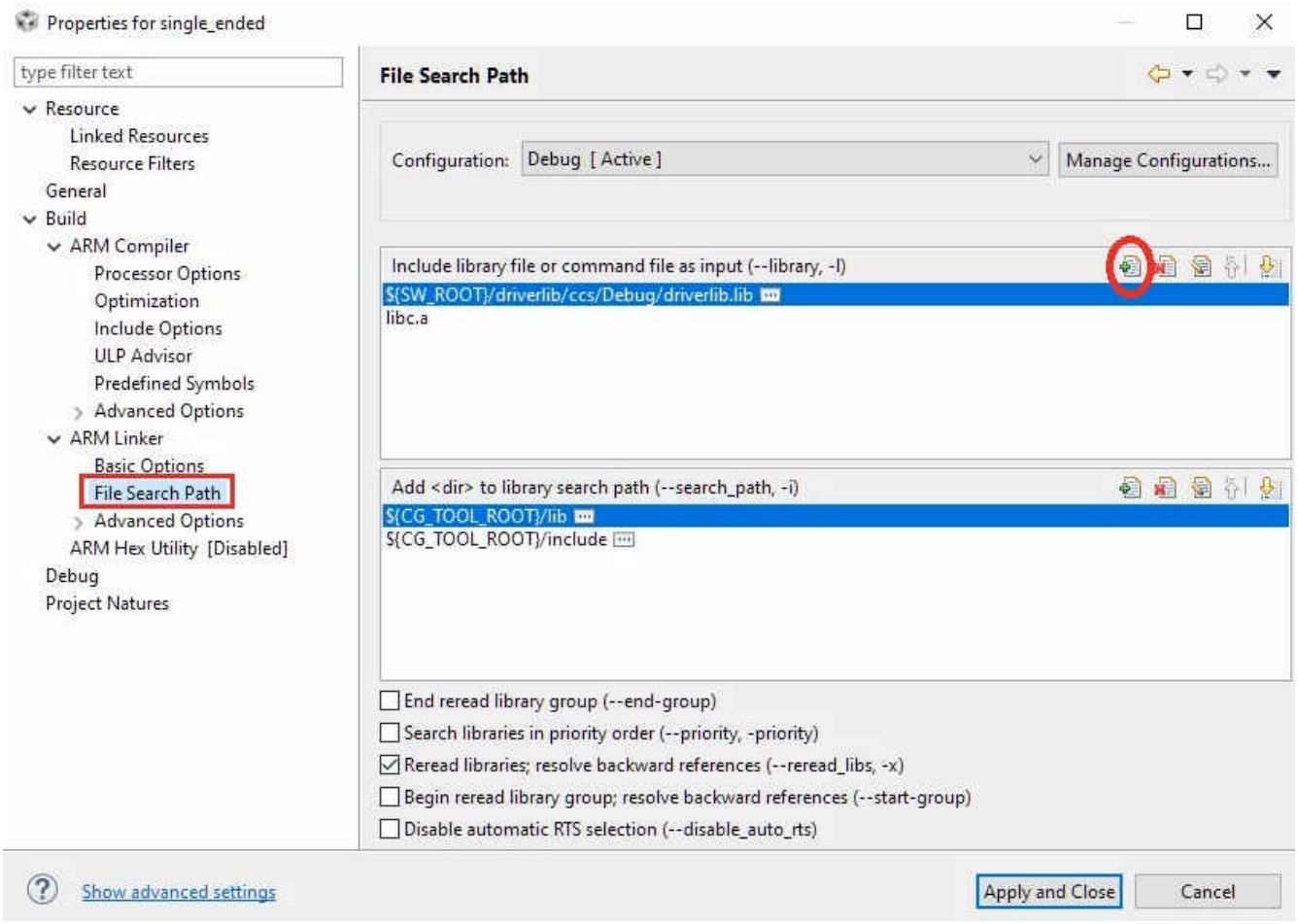

Figure 3-7. How to Add a Library Link in CCS Project Properties

Next, click on ‘Browse…’ and navigate to the folder with the .lib file, and either select the library file and click open or just double click the file. The path displayed should look similar to Figure 3-8. Click ‘OK’ and then ‘Apply and Close’ for the project properties and the library will be linked with the CCS project.

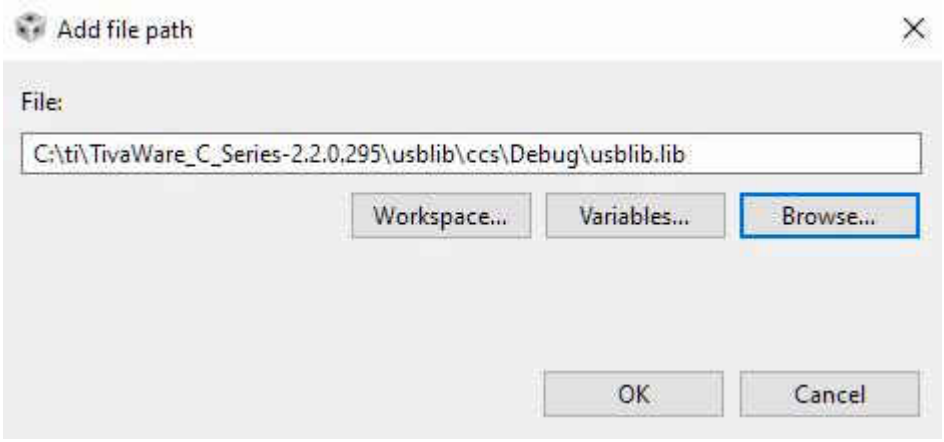

Figure 3-8. File Path to the UsbLib Libary File

While not necessary, the linked location can be edited further to use the **$SW_ROOT** linked resource path. Just replace the TivaWare directory path with **$SW_ROOT** as seen with how the the *driverlib.lib* link path is displayed on Figure 3-7.

This page intentionally left blank.

Chapter 4

How to Debug a TivaWare Library

It is uncommon to run into issues with TI library functions but in the event that a program needs to be debugged at a library level, there are a number of steps that need to be taken in order to do so. The provided .lib file will not allow for IDE debug features to look inside to the source code in order to step through library functions. In order to debug library code, there are a few options. The easiest would be to import the source file from the library folder into the project as covered in Section 3.1. The other two options that will be reviewed in this chapter are to direct the IDE to the source file when attempting C source debugging or recompiling the entire library so the correct path information for the local system is included to allow the IDE to find the C source files.

4.1 How to Direct Code Composer Studio to a Source File

If there is a need to access the source file for a library call while debugging code then CCS needs to be pointed to the source code file. This can be achieved during debug with a combination of using breakpoints and code stepping techniques to prompt the IDE to try and find the file that can then be manually located on the local file system.

The first step will be to set a breakpoint on the exact function call that needs to be debugged. This can be done by double-clicking on the grey space next to that line of code or by right-clicking the line of code and selecting 'Breakpoint' at the top of the menu that pops up. When the breakpoint is set, a small blue orb will appear in the grey space next to the line of code as seen in Figure 4-1.

> If the function is called with a MAP or ROM prefix, remove the prefix to avoid any ROM library calls as these cannot be accessed by the debugger.

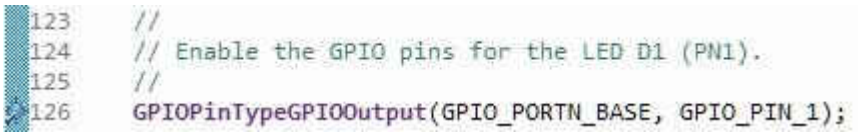

Figure 4-1. Setting a Breakpoint on a DriverLib Call

Once the breakpoint is set, execute code until the breakpoint is triggered. Once that occurs, the execution of the code will stop, and the toolbar at the top will have the options seen in Figure 4-2. Use the 'Step Into' feature, highlighted in the figure or keyboard shortcut F5, to trigger the IDE to search for the source code file.

Figure 4-2. CCS Debug Control Options

Once the IDE is prompted to look for the source file, it will not be able to locate the file and will pop-up a notification (Figure 4-3). As part of this pop-up, the user is given the option to 'Locate File...' and this is what is used to help direct the IDE to the right files. After clicking on 'Locate File...', navigate to the DriverLib folder on the local file system and select it. The entire folder is selected for this, not an individual file, so there is no need to know exactly what source code file is needed. The IDE will search all files in the folder and bring up the correct one automatically.

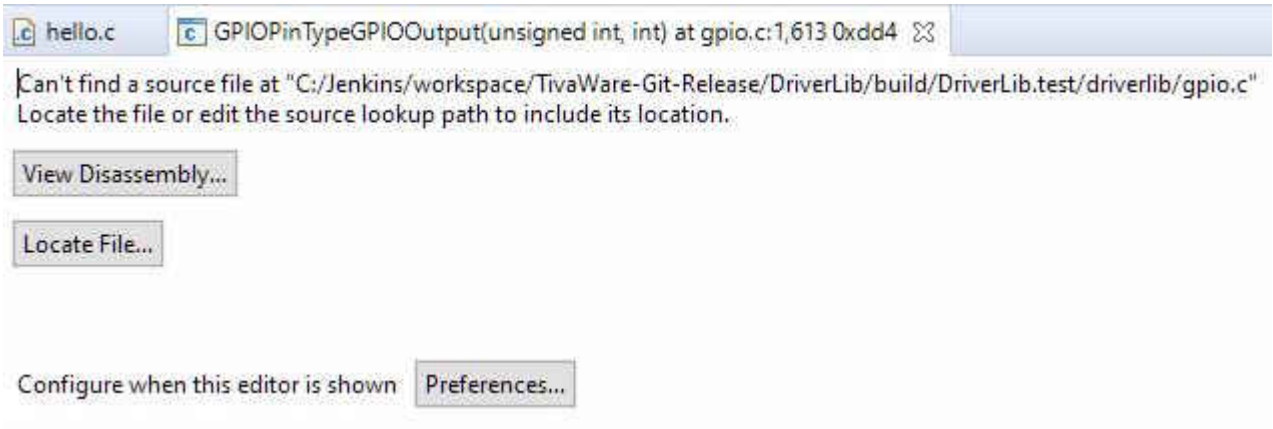

Figure 4-3. Locate File Pop-Up

After providing the directory for the source code, then the pop-up will go away and the source code file will be opened up and the cursor will be placed inside of the function that was being stepped into as shown in Figure 4-4. Debugging can continue from here like normal.

```
1610 //*****************************************************************************
1611 void
1612 GPIOPinTypeGPIOOutput(uint32_t ui32Port, uint8_t ui8Pins)
1613 {
1614     //
1615     // Check the arguments.
1616     //
1617     ASSERT(_GPIOBaseValid(ui32Port));
1618
1619     //
1620     // Set the pad(s) for standard push-pull operation.
1621     //
1622     GPIOPadConfigSet(ui32Port, ui8Pins, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD);
1623
1624     //
1625     // Make the pin(s) be outputs.
1626     //
1627     GPIODirModeSet(ui32Port, ui8Pins, GPIO_DIR_MODE_OUT);
1628 }
```

Figure 4-4. DriverLib Source Code File Successfully Located and Loaded

4.2 How to Rebuild TivaWare Libraries

There are a few situations where recompiling the library files to get a new .lib file output is necessary. One situation would be the case mentioned at the start of this chapter to enable debug of the library. A more common situation would be to change the optimization of the library in terms of speed / size trade-offs. A rare circumstance may be to reflect a change in library files where the .lib file is recompiled in order to reflect the modifications made. The risk with that approach over keeping the modified source file in the project is that it requires maintaining the library over time to ensure the updates are not lost across versions.

In order to recompile a TivaWare library, it must be imported into CCS and rebuilt. The same steps as outlined in Section 2.3 apply except for one key difference. The box 'Copy projects into workspace' must be unchecked. This keeps the library project and the build outputs located in the TivaWare folder to which all TivaWare projects are linked.

Once imported, simply rebuild the project and the changes will be applied in the new .lib file within the TivaWare directory.

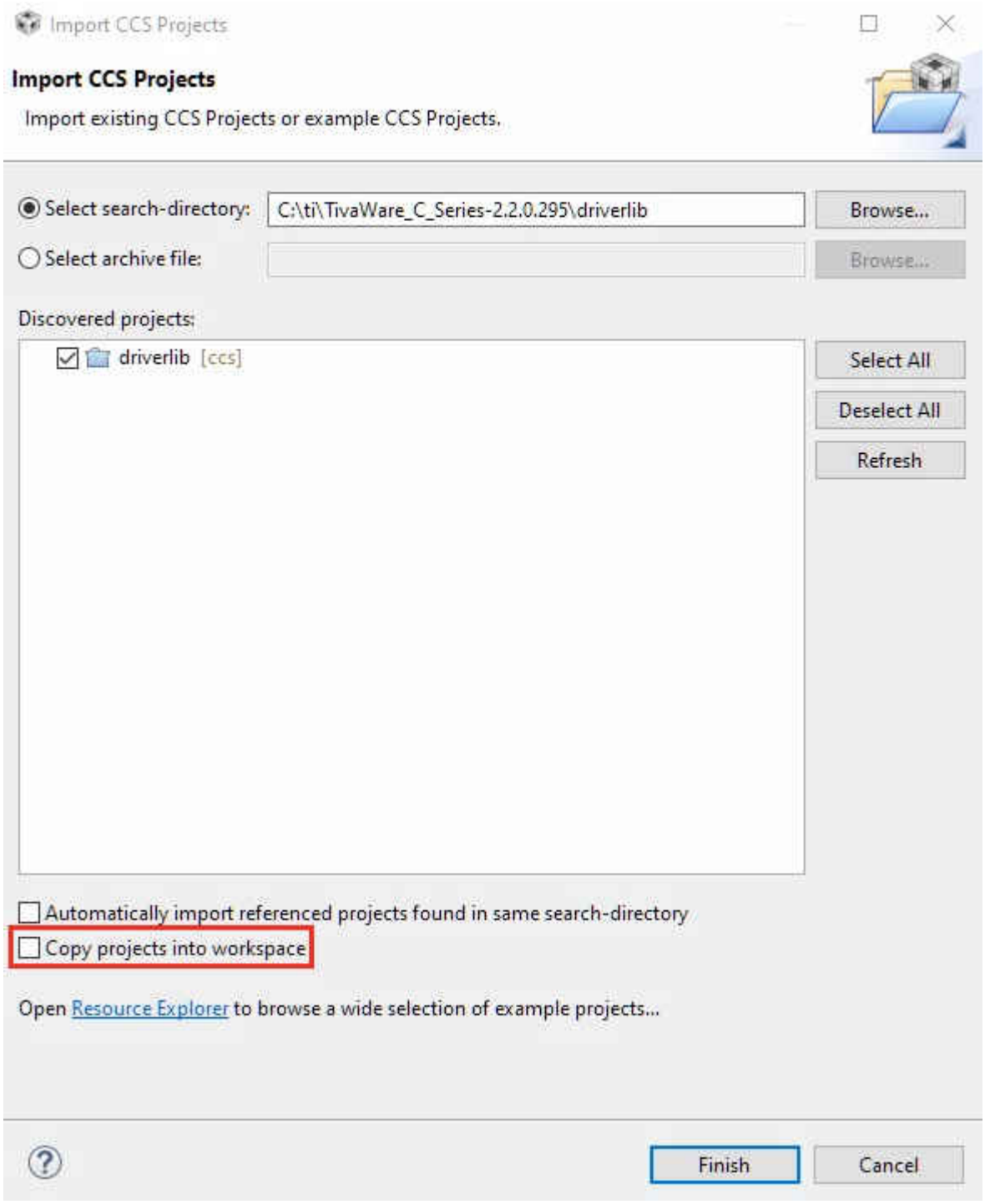

Figure 4-5. Proper Method to Import DriverLib into CCS

This page intentionally left blank.

Chapter 5

How to Add TivaWare to an Existing CCS Project

It is always recommended to start new projects from an existing TivaWare example, but there are times that an existing project needs to be modified to add TivaWare into the project. The process to do so is more involved than just linking libraries and this section will walk through all the other considerations that need to be done.

The images shown during these steps will be for a custom CCS project that is trying to include the functionality of the peripheral example *single_ended*. Each of these changes will be made within the CCS Project Properties. If in doubt about a location referenced, start with the Project Properties and then follow the steps outlined.

5.1 Path Variables

The first error that would occur if trying to add TivaWare to an existing project is that the paths to specific files and folders would not be correctly linked inside of the project properties. Inside of each TivaWare project, there are several Include Paths that allow the compiler to find where specific files are located. The most important path is to the TivaWare root directory. This is defined as the path variable **SW_ROOT** in TivaWare projects, but it does not necessarily have to be added as a path variable. It is possible to add the include option just navigating to the folder. However, as **SW_ROOT** can be re-used for multiple other situations, it is recommended to add the path variable.

To add the path variable, go into the Project Properties and navigate to Resource → Linked Resources. There should be a list of existing folders that are linked as seen in Figure 5-1. To add the **SW_ROOT** path, click ‘New’. Name the path ‘SW_ROOT’ and then click ‘Folder’. Navigate to your TivaWare installation folder (on Windows by default this will be C:\ti\TivaWare_C_Series-2.2.0.295) and click ‘Select Folder’. Verify the path is correct and then click ‘Ok’.

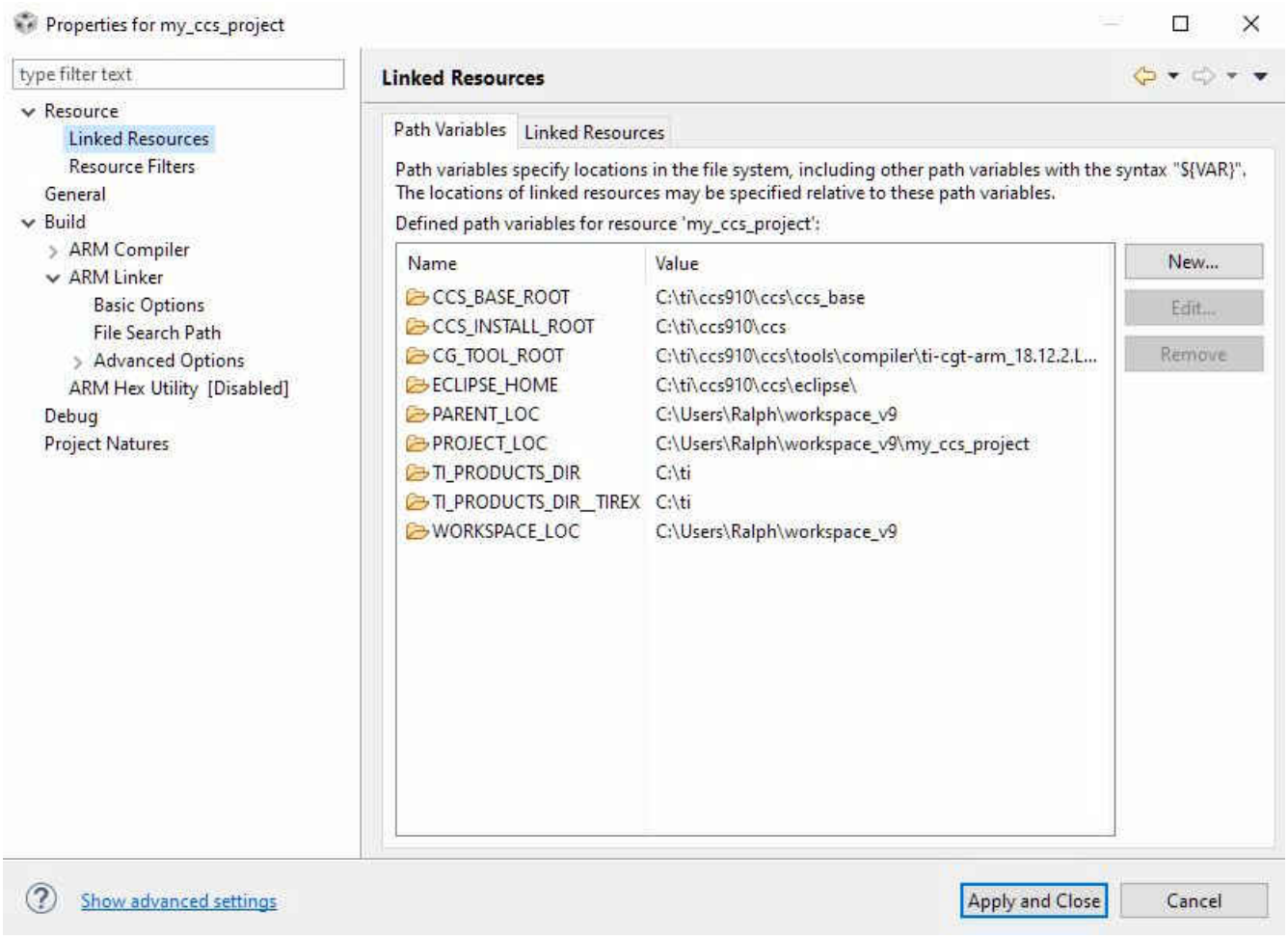

Figure 5-1. Path Variables Menu in CCS Project Properties

5.2 Include Paths

With the path variable for **SW_ROOT** added, now it is simple to add the links to various include paths. The first include path needed is for **SW_ROOT**, which is the TivaWare base directory. That alone will resolve a lot of build errors related to file linking.

In the CCS Project Properties, navigate to Build → ARM Compiler → Include Options. There will be two boxes here, but the one for 'preinclude file' will not be used. Looking at the box for 'Add dir to #include search path' should reveal the current path editions that usually at a minimum includes one for **PROJECT_ROOT** and one for **CG_TOOL_ROOT**. To add **SW_ROOT** to this, click on the 'Add…' button (Figure 5-2) and input *${SW_ROOT}* into the 'Directory' field and click 'OK'.

Some projects may also require a LaunchPad folder to be added. The path required to be added for those follows this pattern: *${SW_ROOT}/examples/boards/ek-tm4c123gxl*, with the final folder name being that of the LaunchPad being used.

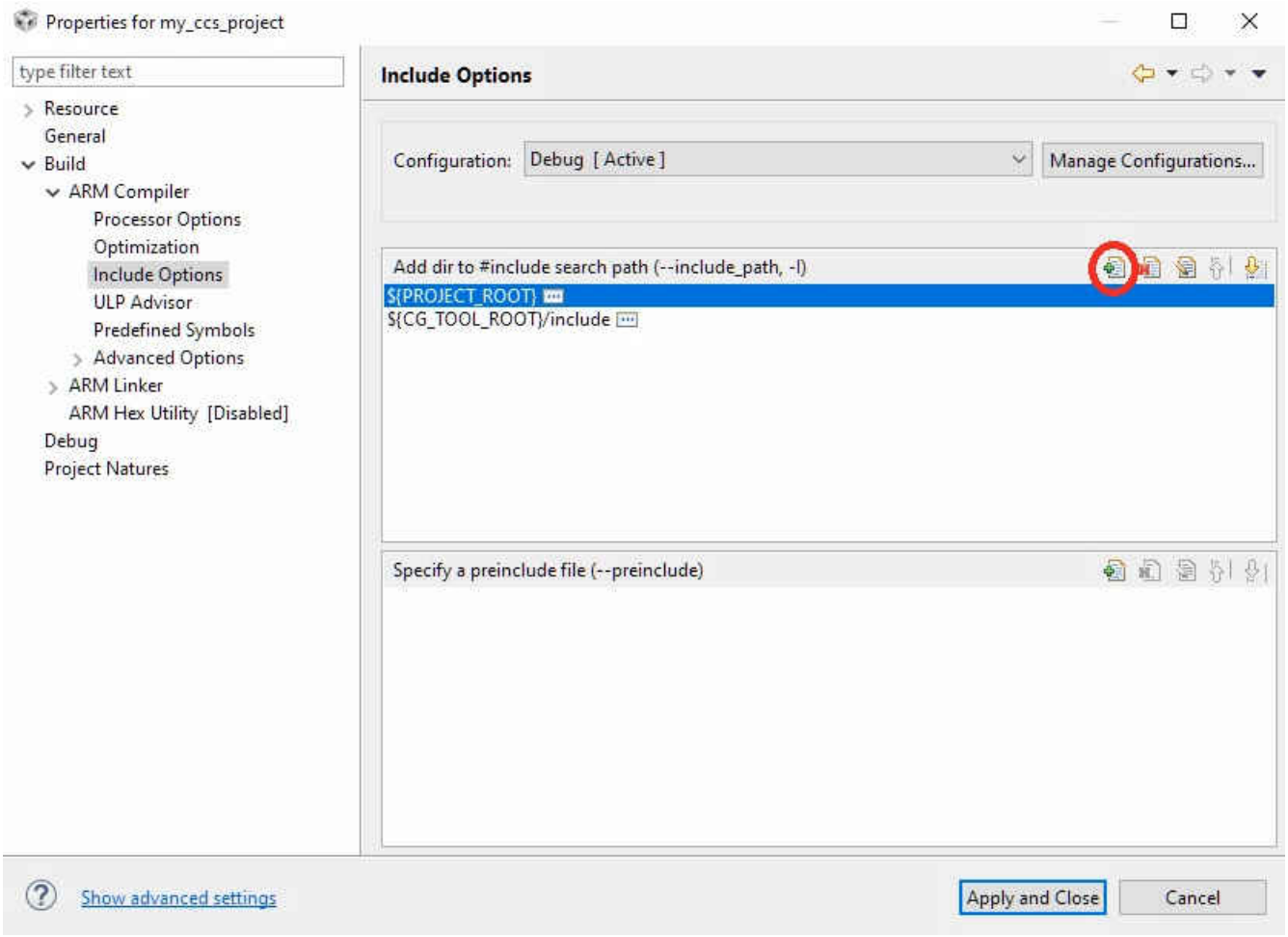

Figure 5-2. How to Add a New Include Path in CCS Project Properties

5.3 Predefined Variables

The next step is to add in a predefined symbol into the project. Predefined symbols are used by TivaWare files including in DriverLib to select code blocks based on the TM4C MCU family used. To add a symbol first go to CCS Project Properties and then navigate to Build → ARM Compiler → Predefined Symbols. There should already be at least one symbol defined for the TM4C microcontroller being used such as the **PART_TM4C123GH6PM** shown in Figure 5-3. If for some reason that is not present, it must be added as well. The part number that should be used after '**PART_**' would be the device that was chosen in the 'General' section of the Project Properties.

To add a new symbol, click the 'Add…' button and type in the value into the box that pops up. The symbol that needs to be added depends on the MCU family used. If a TM4C123x device is used, then add **TARGET_IS_TM4C123_RB1**. If a TM4C129x device is used, then add **TARGET_IS_TM4C129_RA2**. Click 'OK' to add the new symbol.

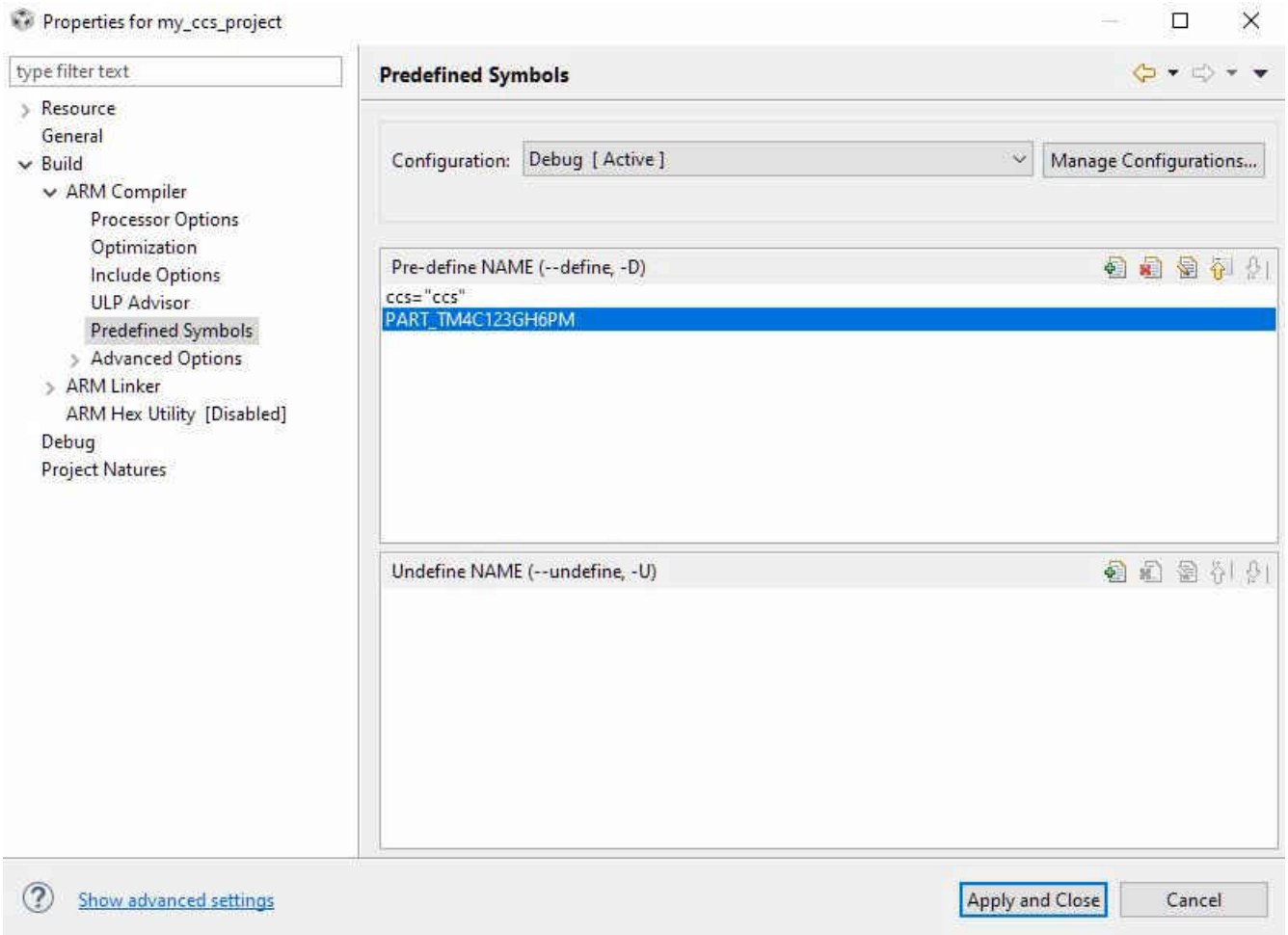

Figure 5-3. Predefined Symbols Menu in CCS Project Properties

5.4 Library Linking

The final step is to follow the instructions in Section 3.2 to link all relevant TivaWare libraries to the CCS project. At minimum *driverlib.lib* will be required, but other libraries may be as well. Depending on what is added, also remember that files can be linked as laid out in Section 3.1.

With that, all steps that are required should be completed. Try to build the project from there and if any errors occur, check for missing files that may need to be linked. If issues persist, check the linked locations for an official TivaWare project for comparison.

This page intentionally left blank.

Chapter 6

TivaWare Boot Loader

An important feature offered with TM4C microcontrollers is the ability to use multiple interfaces in order to update the firmware on the device via a boot loader. There are both ROM and flash boot loader options available. A comparison of the differences between these modes, how to use the ROM boot loader, and all the details of supported interfaces can be found in the *TivaWare™ Boot Loader User's Guide (SW-TM4C-BOOTLDR-UG)*.

This section focuses on how to use the flash boot loader in order to review how to configure the boot loader in CCS, before reviewing the tools that can be used to then execute the boot loading process.

6.1 Modifying a TivaWare Project for Boot Loading in Code Composer Studio

For this section, the TivaWare *boot_serial* example for the EK-TM4C123GXL LaunchPad Development Kit will be used as the boot loader firmware, and the *hello* example will be used for the application code.

The *boot_serial* example contains only a *bl_config.h* file as the flash boot loader itself is part of the TivaWare *boot_loader* folder. Any modifications to the boot loader are done through the *bl_config* file where the serial interface can be selected and modified, including selecting which modules and pins are used (unlike the ROM boot loader that is locked to specific pins). Another important feature is the ability to set the starting address. This is defined as **APP_START_ADDRESS**, and that determines where the application that is loaded by the boot loader starts in flash memory. This is required because the boot loader needs to be placed in flash memory starting at address 0x0000.0000. The size of **APP_START_ADDRESS** is defined based on the size of the boot loader code. It is used at the end of the boot loading process to jump to the start of the application code and begin executing it.

```
#define APP_START_ADDRESS       0x2800
```

The **APP_START_ADDRESS** differs between the TM4C123x devices and TM4C129x devices due differences in the flash architecture between the families. In general there are two key considerations for where to place the starting address. The first is that the vector table must be aligned to a 1024 byte page boundary. The second is that the boot loader should not be erased while programming the application code since flash memory is erased in sectors.

For TM4C123x devices, the flash sectors are 1kB in size, but for TM4C129x devices, they are 16kB in size. In order to avoid having the boot loader and applications in the same sector, the simplest solution is to start the application on the first sector boundary beyond the end of the boot loader. This will also meet the requirement for having the vector table aligned with a 1024 byte page boundary as well. For TM4C123x devices using the provided TivaWare flash boot loader, the starting address is set to be 0x2800, which is the first sector without any boot loader code present. For TM4C129x devices, the starting address is set to be 0x4000 because of the 16kB sector size.

After the starting address is defined, the next step is to compile the application code that will be boot loaded so the start address is reflected in the generated binary file. This is done by modifying the linker command file for the CCS project. After opening or importing the *hello* project into CCS, locate the *hello_ccs.cmd* file and double-click to open the file in the CCS linker command file editor. There are two important sections of code in this file, the address bases set by the #define statements, and the system memory map.

```
#define APP_BASE 0x00000000
#define RAM_BASE 0x20000000

/* System memory map */

MEMORY
{
     /* Application stored in and executes from internal flash */
     FLASH (RX) : origin = APP_BASE, length = 0x00040000
     SRAM (RWX) : origin = RAM_BASE, length = 0x00008000
}
```

To prepare *hello* for use with the boot loader, first update the **APP_BASE** memory address to reflect the **APP_START_ADDRESS** from *bl_config*. Because the boot loader takes up the memory space from 0x0000.0000 to **APP_BASE**, the *length* field of the system memory map must be updated to reflect the available flash space for the application. The length field should be the total device flash memory minus the space needed for the boot loader. In this case, *0x0004.0000 – 0x0000.2800* would be *0x0003.D800*. A simple and error minimizing way to reflect this is to simply subtract **APP_BASE** from the defined *length* value.

```
#define APP_BASE 0x00002800
#define RAM_BASE 0x20000000

/* System memory map */

MEMORY
{
     /* Application stored in and executes from internal flash */
     FLASH (RX) : origin = APP_BASE, length = 0x00040000 - APP_BASE
     SRAM (RWX) : origin = RAM_BASE, length = 0x00008000
}
```

After making these changes, save the updated .cmd file and then re-compile the project to generate the updated .out and .bin files in the Debug folder. Now the *hello* project is ready to be loaded in via the *boot_serial* boot loader.

The last step using Code Composer Studio is to now program the target device with the *boot_serial* project. Because the boot loader does not have a *main* function the IDE will not have an option to ‘run’ the firmware after programming and means that the boot loader was successful programmed.

6.2 How to Boot Load with LM Flash Programmer

The TI offered LM Flash Programmer utility tool enables boot loading for TM4C devices over UART, USB, and Ethernet interfaces. In order to use the tool for boot loading, select the Manual Configuration option in the Quick Set section of the Configuration tab (Figure 6-1). After Manual Configuration is selected, then the Interface section can be used to select one of the three boot loading options of *Serial (UART)*, *Ethernet*, and *USB DFU*.

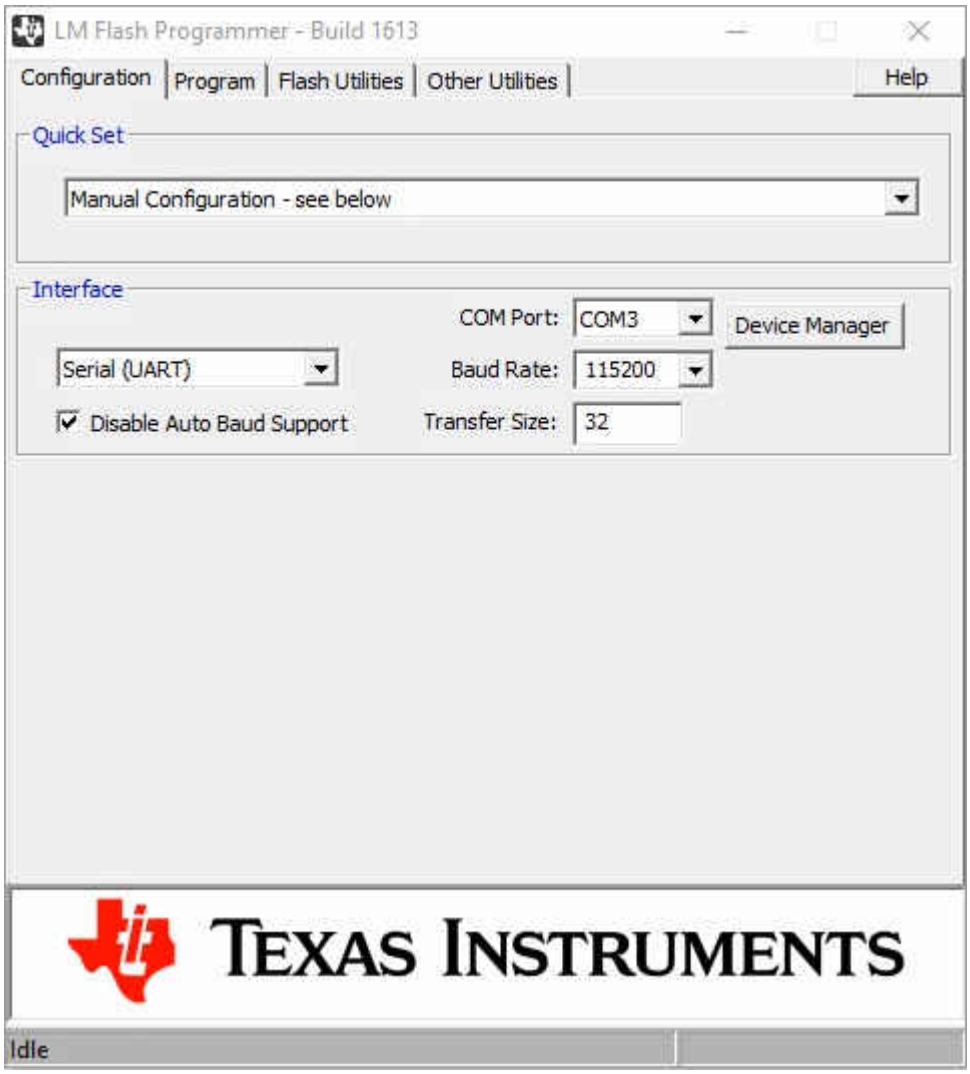

Figure 6-1. LM Flash Programmer Manual Configuration

For the Ethernet boot loader, it is important to ensure the correct Client MAC Address is entered. When using the *USB DFU* mode, the TM4C microcontroller must be in a state where the USB boot loader is running so that the PC can recognize the USB DFU port. If no devices show up on the list, invoke the USB boot loader on the TM4C microcontroller and then use the refresh button to trigger LM Flash Programmer to search for the USB DFU interface again.

Once the desired interface is setup, move to the Program tab to select the binary file for programming. Browse the local file system to find the .bin file for your project (Figure 6-2). This will always be located in the Debug folder of a CCS project.

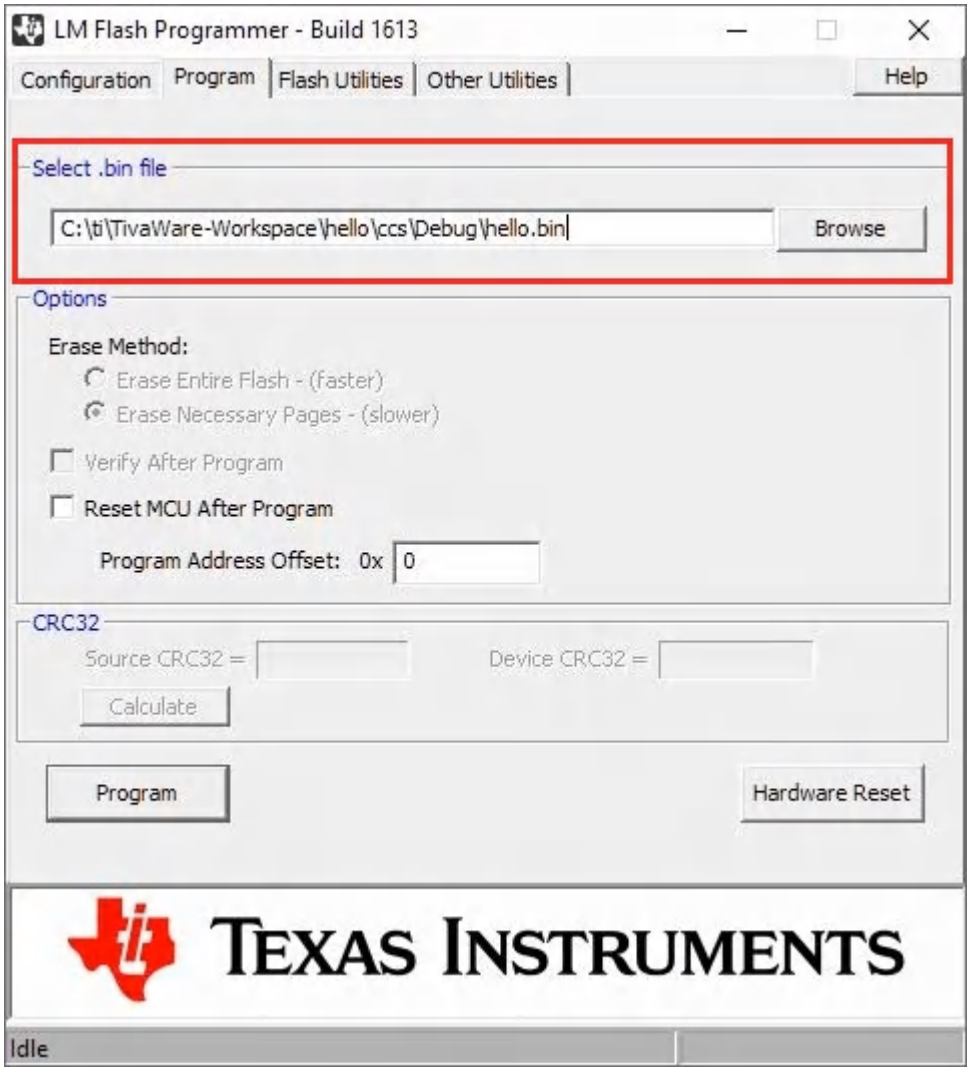

Figure 6-2. Browse for .bin File

Once the file is selected, set the correct *Program Address Offset* (Figure 6-3). This will be equal to the **APP_START_ADDRESS**. Once these steps are completed, press the Program button to trigger the boot loading process and upload the application firmware onto the device.

Figure 6-3. Set the Program Address Offset Value

Chapter 7

Software Best Practices

There are many different elements that must come together when architecting a fully functional embedded code project and it can be easy to miss one or two key details. This section will go through the most commonly seen issues that cause issues within embedded systems and how to avoid those problems.

7.1 Stack / Heap Settings and Stack Overflow

Ensuring there is sufficient stack size for a program is an important consideration to ensure that programs are not susceptible to stack overflow. The program stack exists in RAM memory and is used to store information like current register values and return addresses during program execution. A stack overflow is a programming error where the stack pointer exceeds the allocated space for the program stack. When that occurs, it is possible for data read back by the stack to have been overwritten by the application and the corrupted data would then cause a program crash.

In order to adjust the stack settings for a Code Composer Studio project, open the Project Properties and go to Build → ARM Linker → Basic Options. Towards the bottom of the list there is a box to 'Set C system stack size' (Figure 7-1). In order to ensure the memory is aligned, always use a multiple of 4 bytes for the stack setting.

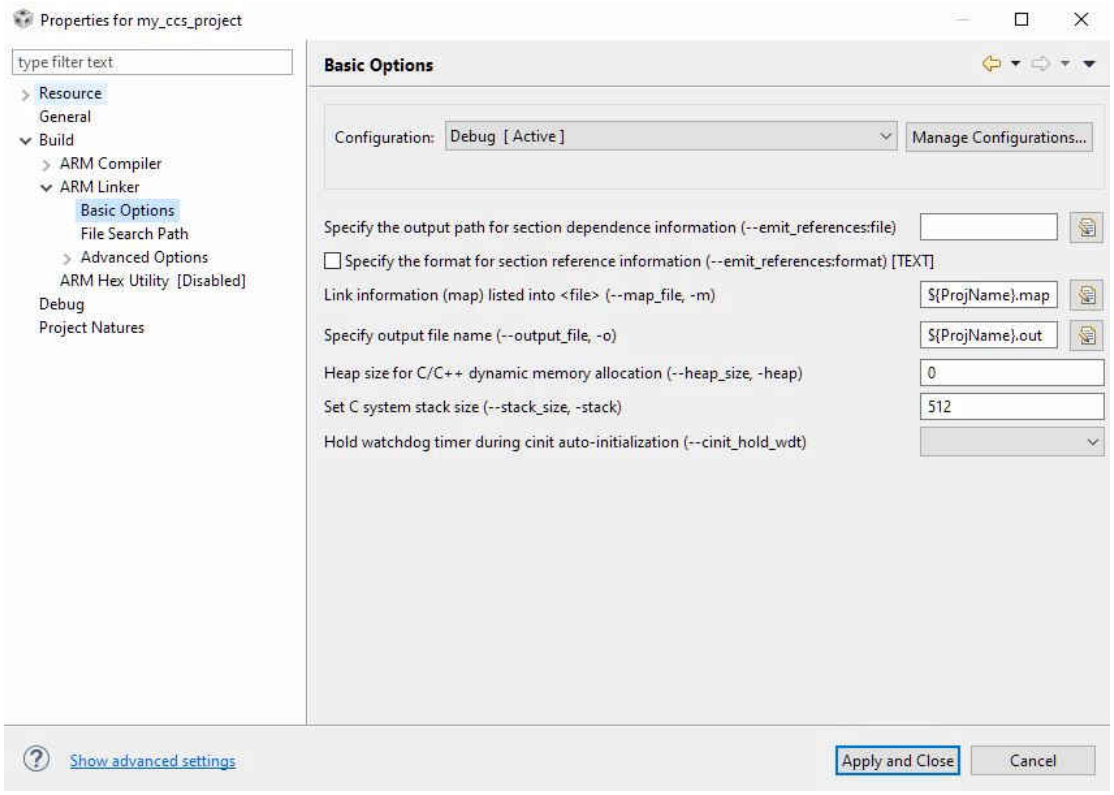

Figure 7-1. Stack and Heap Settings in CCS Project Properties

Another location to modify the stack size is within the linker command file (.cmd) for the project. On the bottom of the file there should be a line item for *__STACK_TOP* and the numerical value that follows can be edited to adjust the stack size.

```
__STACK_TOP = __stack + 512;
```

For programs that use dynamic memory allocation such as *malloc* and when using dynamically created tasks within an RTOS, heap memory needs to be allocated as well. Heap is a separate section of memory that is reserved in order to support dynamic memory allocation. The heap setting for a Code Composer Studio project can be found directly above the stack option, 'Heap size for C/C++ dynamic memory allocation'. This will typically be set to zero by default in TivaWare examples.

7.2 Interrupt Service Routines

A fundamental building block of any embedded program, the Interrupt Service Routine (ISR) is a block of code that is entered due to system hardware interrupts. There are right and wrong ways to use ISRs and incorrect usage of an ISR can cause a lot of system wide issues. This section will go over some best practices and then cover how to debug a couple of basic errors that users stumble upon related to ISRs.

7.2.1 Best Practices

As the name implies, an Interrupt Service Routine will interrupt what a program is currently doing in order to execute the code within the routine. In the simplest of programs that behavior is generally harmless regardless of how an ISR is written, but as the size and scope of the program increases it becomes increasingly important to write efficient ISR's in order to minimize the amount of time that typical program execution is being interrupted.

ISR's are best used to quickly receive and store or fetch and send data, and/or reflect a change in state for the rest of the program. When receiving or sending data, the goal should be to process data from the hardware peripheral that triggers the ISR only and use flags or changes in state to inform the primary program application that data has been processed in order for the main program to control what is done with that data outside of the ISR.

On the topic of flags and best practices, when using global variables inside of an ISR to act as flag, counter, or any other variable that will be modified inside of the ISR, the global variable must be declared as a **volatile** variable. If the **volatile** keyword is not used, the compiler may determine the variable is not being used inside of an ISR and optimize it out of the ISR. This would prevent the variable from being modified when the ISR executes and the program will not run as intended. A missing volatile declaration for a global variable is one of the most common programming errors when using interrupts and should always be part of a code review checklist.

Key common mistakes that cause issues with ISR's include: doing data crunching within an ISR, running loops in an ISR that prevent it from quickly exiting, or sending data with blocking commands such as **UARTprintf** where the ISR cannot exit until the data is transmitted properly. Each of these impede the ISR from quickly executing and cause latency that can impact system performance, and in each case, these should be handled within the main program execution instead.

If a system is experiencing unexpected behaviors and there is suspicion a slow ISR may be to blame, there is a simplistic method to measure the duration of ISR's using an oscilloscope. Toggling a GPIO at the entry and exit of an ISR will allow for a measurement of how much time passes each time the ISR executes. If the ISR has multiple branches, additional GPIO can be added to help shed light on which branch is executing.

7.2.2 TivaWare Vector Tables and IntDefaultHandler

Within the TivaWare *startup* file that is included in every project and for each compiler, there is a data section for the interrupt vector table for the microcontroller. The vector table entries are populated based on the interrupt provided within the device that can be found in the *Exception Types* section of the device data sheet. Comments in the *startup* file will indicate which line corresponds to which hardware interrupt. This vector table is then mapped within the device to link all ISR's within the application to the Nested Vectored Interrupt Controller (NVIC) on the TM4C microcontroller.

The vector table is filled with default ISR handlers to begin with. Aside from the **ResetISR**, which is used to soft reset the device, each other default ISR handler contains an empty while(1) loop. The **FaultISR** indicates that a fault has occurred in the MCU and debug will be required to understand what went wrong. There is an application report offered by TI that is focused on Diagnosing Software Faults, which shows the debug process for system faults. The **NmiSR** will be entered when the processor receives an NMI. Every other interrupt is associated to the catch all **IntDefaultHandler**. A program getting stuck in the **IntDefaultHandler** would indicate that an interrupt has been enabled but that the ISR handler was not properly associated with the NVIC vector table.

There are two ways to associate an application ISR with the appropriate entry for the NVIC vector table. The first is to use the **IntRegister** API, which will copy the vector table stored in Flash to RAM and then add the new ISR association during code execution. This allows for a dynamic method to create a new ISR association and offers more flexibility. The second method is to manually add an **extern** call for the application's ISR handler in the *startup* file and replace the **IntDefaultHandler** instance for the intended peripheral with the ISR handler. This is a static method that will make the association at compile time and provides the advantage of less memory usage and code execution that makes it generally preferred.

To add an application-defined ISR handler into the *startup* file, the **extern** keyword is used in front of the declaration. The following code is from the *interrupts* example on the EK-TM4C123GXL LaunchPad. As seen, each ISR handler is added with an **extern** keyword preceding the rest of the declaration.

```
//*****************************************************************************
//
// External declarations for the interrupt handlers used by the application.
//
//*****************************************************************************
extern void IntGPIOa(void);
extern void IntGPIOb(void);
extern void IntGPIOc(void);
```

Once the ISR handler has been properly added to the *startup* file, find the peripherals in the vector table that correspond to the ISR. For example, searching through the table to find the GPIO peripherals for Port A, B, and C will uncover the following segment in the vector map.

```
    IntDefaultHandler,                      // The PendSV handler
    IntDefaultHandler,                      // The SysTick handler
    IntDefaultHandler,                      // GPIO Port A
    IntDefaultHandler,                      // GPIO Port B
    IntDefaultHandler,                      // GPIO Port C
    IntDefaultHandler,                      // GPIO Port D
    IntDefaultHandler,                      // GPIO Port E
    IntDefaultHandler,                      // UART0 Rx and Tx
```

Not all peripherals are grouped together sequentially. Searching the file with a Ctrl+F can be a quick way to skim through the whole table.

Replace **IntDefaultHandler** for each corresponding entry on the interrupt vector table with the function name for the desired application-defined ISR handler and save the file.

```
    IntDefaultHandler,                      // The PendSV handler
    IntDefaultHandler,                      // The SysTick handler
    IntGPIOa,                               // GPIO Port A
    IntGPIOb,                               // GPIO Port B
    IntGPIOc,                               // GPIO Port C
    IntDefaultHandler,                      // GPIO Port D
    IntDefaultHandler,                      // GPIO Port E
    IntDefaultHandler,                      // UART0 Rx and Tx
```

7.3 TivaWare Hardware Header Files

A common issue that can occur with compiling code after merging examples together or trying to add new functionality to an existing project is to get build errors like the 'unresolved symbol' error seen in Section 3.1. There are many situations that can cause the error to occur and that section covers some of the basic ones based on TivaWare API's. However, this issue can also arise when using hardware-based declarations that are harder to track down.

The declarations for TM4C hardware access macros, register offsets, and hardware bit fields are provided within the set of hardware header files. These files are located within the *inc* folder of the TivaWare SDK and have filenames that start with the 'hw_' prefix. These declarations are then used by TivaWare API's to control the MCU at a register programming level. While TivaWare provides a broad set of API's for every peripheral, there are cases where a few register level programming calls may be needed such as to optimize the speed of a process or in the event an API does not offer the correct settings.

A majority of the hardware header files are specific to peripherals and contain information for the registers (both offsets and bit fields) of that specific peripheral. These files are occasionally needed in an application when receiving information such as peripheral status flags that need to be parsed. If additional code is added and an unresolved symbol seems to be a definition from a specific peripheral, check if that peripheral's hardware header file is included in the project or search the *inc* folder in TivaWare to see which file(s) contain that definition.

In addition to the peripheral specific files, there are four additional hardware header files that provide more general-purpose declarations:

The *hw_memmap.h* file must be used in every TivaWare project as it provides the memory base addresses for all peripherals that are accessed. This is the only mandatory hardware file required.

The *hw_ints.h* file includes the definitions needed to enable peripheral ISR's in the NVIC vector table. This file is needed for any TivaWare project that uses an ISR and it is a good practice to always have it included.

The *hw_nvic.h* file has the definitions for everything concerning the NVIC. It is included in every TivaWare project within the *startup_ccs* file, but if the application code needs to use any NVIC-specific definitions such as with a boot loader application then it must be included within the application file as well.

The *hw_types.h* file provides the macros for directly modifying a register. This is required only when making direct register modifications as part of the application code.

7.4 ROM and MAP TivaWare Prefixes

In order to help minimize Flash space, TM4C microcontrollers have a version of TivaWare's DriverLib loaded into ROM memory. However, the DriverLib included within the ROM is an older version and any updated functions or new functions must be executed via Flash using the latest TivaWare DriverLib. To minimize the burden of understanding which functions that are loaded in ROM are still up to date and which ones must be executed from Flash, TivaWare includes a mapping file that is used to determine whether to use a ROM function or a Flash function. With this setup, there are three possible function calls for each DriverLib function. The generic call, a ROM prefixed call (functions starting with 'ROM_'), and a MAP prefixed call (functions starting with 'MAP_').

For all ROM prefixed calls, the *rom.h* header file is required, and it will select the correct function from the ROM memory map to execute. If the function does not exist in ROM, then a compiler error will indicate that function is not available. In some cases, older ROM functions that have errata associated with them are removed from *rom.h* to avoid misuse, and in those cases the DriverLib function from the latest TivaWare should be used. In other cases, the ROM function may not exist at all. This adds a level of complexity that may be undesirable for programmers.

To simplify this process, the MAP prefix is offered as the third option for a function call. All DriverLib function calls have an equivalent defined within the *rom_map.h* header file based on whether a ROM version exists or not. Therefore, using the MAP prefix takes all the guesswork out of when to use ROM or Flash DriverLib functions while gaining the benefit of minimizing the Flash footprint for DriverLib. Once *rom_map.h* is included in an application file simply add the MAP prefix to all DriverLib functions to leverage the benefits of all available ROM functions.

This page intentionally left blank.

Chapter 8

TM4C Resources

Device Software

- TivaWare
- TI-RTOS for Tiva-C

Software Tools

- Code Composer Studio
- LM Flash Programmer
- Uniflash

Hardware Development Kits

- TM4C123x LaunchPad
- TM4C129x LaunchPad
- TM4C129x Development Kit

Documentation

- General Information
  - *TM4C Microcontrollers Product Selection Guide*
- Software – TivaWare
  - *TivaWare™ Peripheral Driver Library User's Guide*
  - *TivaWare™ USB Library User's Guide*
  - *TivaWare™ Boot Loader User's Guide*
- Software – Miscellaneous
  - *Diagnosing Software Faults in Stellaris® Microcontrollers Application Report*
  - *TI-RTOS 2.20 for TivaC Getting Started Guide*
  - *TI-RTOS 2.20 User's Guide*
- Hardware Design Application Reports
  - *System Design Guidelines for the TM4C123x Family of Tiva™ C Series Microcontrollers Application Report*
  - *System Design Guidelines for the TM4C129x Family of Tiva™ C Series Microcontrollers Application Report*
  - *Using TM4C12x Devices Over JTAG Interface Application Report*
- Hardware User's Guides
  - *Tiva™ C Series TM4C123G LaunchPad Evaluation Board EK-TM4C123GXL User's Guide*
  - *Tiva™ C Series TM4C1294 Connected LaunchPad Evaluation Kit EK-TM4C1294XL User's Guide*
  - *Tiva™ TM4C129X Development Board DK-TM4C129X User's Guide*

- Application Reports
  - *IoT Demo Using EK-TM4C1294XL LaunchPad Application Report*
  - *Using the Edde Flex CAN Controller on the EKTM4C123GXL LaunchPad Application Report*
  - *Using the Edde Flex CAN Controller on the EKTM4C1294XL LaunchPad Application Report*
  - *Using Feature Set of I2C Master on TM4C129x Microcontrollers Application Report*
  - *Using USB Host Mode on the EK-TM4C123GXL LaunchPad Application Report*
  - *ADC Oversampling Techniques for Stellaris® Family Microcontrollers Application Report*

Technical Support

- E2E Forums
  - Arm®-based Microcontrollers Forum
- FAQ's
  - TM4C General Information & FAQ
  - Comprehensive List of TI-RTOS FAQs
  - Original TI-RTOS Workshop
  - Third-Party Support for TM4C

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (March 2021) to Revision A (August 2022)	Page
• Updated the numbering format for tables, figures and cross-references throughout the document	7
• Updated Section 6.2	31
• Updated Section 7.2.2	34

This page intentionally left blank.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265