Application Report
SPRACK3–February 2019

Enabling Wi-Fi® on Cortex®-M microcontrollers using CMSIS-RTOS2 and CMSIS-drivers

Timothy Logan

ABSTRACT

The CC3120R device is a companion microcontroller used to enable Wi-Fi radio capabilities on a host microcontroller that does not have integrated wireless capabilities. This application report provides guidance on how to enable CC3120 Wi-Fi functionality on microcontrollers using CMSIS-Drivers and CMSIS-RTOS2.

Project collateral discussed in this document can be downloaded from the following URL: http://www.ti.com/lit/zip/sprack3.

Contents

List of Figures

List of Tables

Trademarks

1 Introduction

While the SimpleLink SDK Ecosystem and SimpleLink SDK Wi-Fi Plugin enable the use of a CC3120R as a network processor (NWP) on a Texas Instruments SimpleLink™ device, it is often required to attach the NWP to other Cortex-M based microcontrollers. For these use cases, a generic solution of using the Arm® standard CMSIS-Drivers and CMSIS-RTOS2 software packages is provided and described in detail in this application report. This document is coupled with a supplemental software package that provides a version of the Wi-Fi host driver that has been ported over to use both CMSIS-RTOS2 and CMSIS-Drivers. By utilizing standard software interfaces defined directly by Arm, this ported host driver provides a way to enable Wi-Fi on a variety of Cortex-M microcontrollers spread across the entire Arm Cortex-M ecosystem. This enables Texas Instruments Wi-Fi solutions to be fully utilized on a wide array of host microcontrollers regardless of the host microcontroller's underlying vendor.

This implementation "casts a wide net" and covers numerous platforms across different architectures and vendors. While CMSIS-RTOS2 does not require any specific vendor implementation, if the host microcontroller does not support CMSIS-Drivers a level of manual integration work will have to be done to enable the use of proprietary drivers. It should also be noted that while this application report describes and provides an example of using CMSIS-RTOS2 and CMSIS-Drivers with the CC3120 Wi-Fi host driver, conceivably all real time operating systems and HAL/drivers can be used to communicate with the CC3120. The process of porting individual elements of the host driver to work with a generic host microcontroller implementation is described in detail in the *CC3120, CC3220 SimpleLink™ Wi-Fi® and Internet of Things Network Proces Programmer's Guide*.

2 CMSIS-Drivers

CMSIS-Drivers is a software interface defined by Arm aimed at abstracting out common embedded functionalities such as serial communication to a common API set that is compatible across numerous Arm Cortex-M vendors. This allows for a communication driver written for one vendor's Cortex-M4F, for example, to be ported to another vendor's Cortex-M33 solution with relative ease and minimal software porting overhead. In the software package provided by this application report, the USART and serial peripheral interface (SPI) CMSIS-Drivers are showcased.

In a typical setup, the CC3120 network processor is connected to the host microcontroller over a serial SPI connection. Due to the CC3120's SPI peripheral limitations, the SPI bit rate is limited to 20 MHz. In this software package, the SPI CMSIS-Driver is used to manage and completely control the communication to and from the CC3120 and host microcontroller. The USART CMSIS-Driver is used to manage serial port and terminal interaction to display relevant debug and application information. By using these two common software drivers, you have the ability to completely abstract out the communication software interface and easily swap the underlying CPU.

This software ecosystem is illustrated in Figure 1.

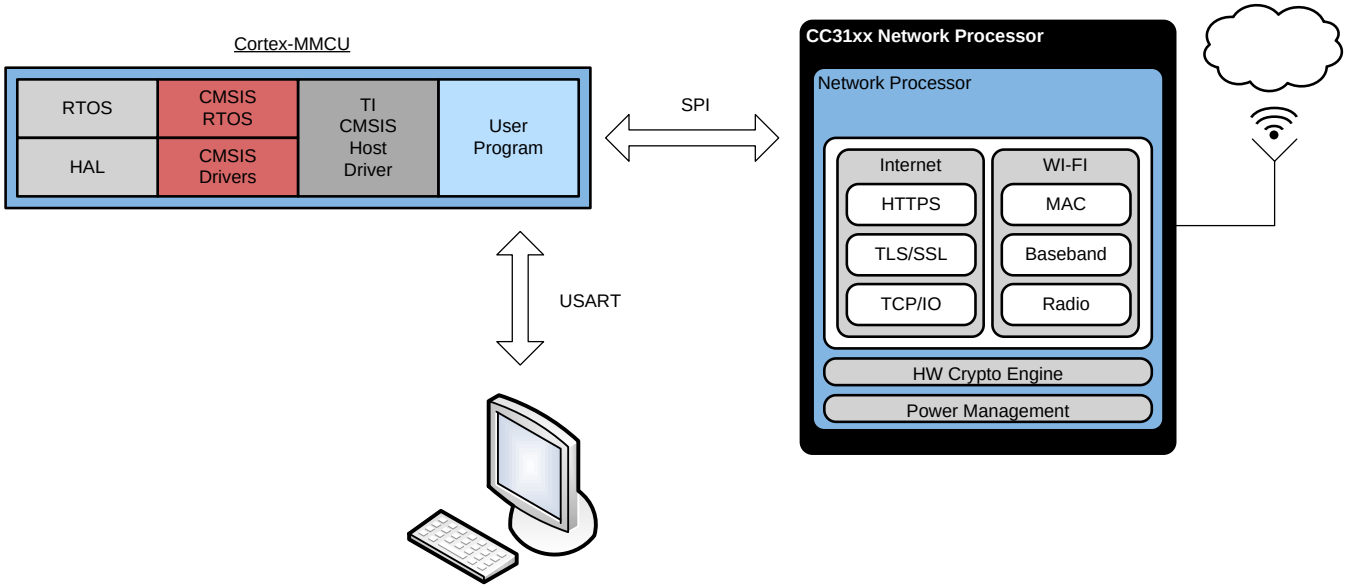

Figure 1. CMSIS-RTOS2/CMSIS-Driver Ecosystem

It is an important distinction to make that while CMSIS-Drivers are used to abstract out the common embedded functionalities such as serial communication, they are by no means meant to be a complete abstraction of the entire device's peripheral set. In each end application, the programmer needs to utilize the underlying microcontroller's software drivers to fully utilize the peripheral set of that device. This allows for differentiation and full hardware entitlement of the underlying peripheral set beyond what a "one size fits all" software driver would provide. Peripherals such as the analog-to-digital converter (ADC), digital-to-analog converter (DAC), and so forth will have differentiated features that are platform specific and not practical to abstract. For these instances, it is more productive and more powerful to use the device-specific software driver, whereas, common functionalities such as the inter-integrated circuit (I2C), universal asynchronous receiver/transmitter (UART), universal serial bus (USB), and so forth can be effectively abstracted out to a platform independent software driver.

3 CMSIS-RTOS2

Just like CMSIS-Drivers abstracts out peripheral specific interaction in software, CMSIS-RTOS2 abstracts away the underlying real-time operating system (RTOS) implementation. Programming in the Arm Cortex-M ecosystem has the benefit of you being able to work on a common set of tools and software regardless of the underlying microcontroller. For applications that require an RTOS simply from the fact that you are programming on a Cortex-M device enables you to use a variety of device independent RTOS configurations. While each underlying RTOS has varying degrees of differentiation, the majority of user accessible APIs and core functionality will be similar regardless of the underlying implementation.

Components such as semaphores, mutexes, queues, and so forth will have implementations on virtually all RTOS configurations; however, only CMSIS-RTOS2 provides a device independent abstraction layer that standardizes these software interactions across multiple Arm Cortex-M vendors.

This level of abstraction and how the different layers tack up can be seen in Figure 2.

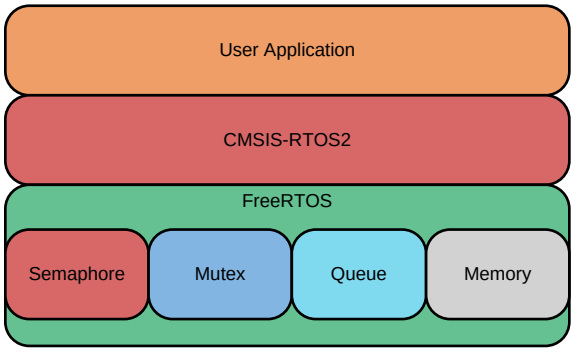

Figure 2. CMSIS-RTOS2 Organization

The software package provided with this application report includes a full distribution of FreeRTOS as well as an included port of the CMSIS-RTOS2 compatibility layer. The latest version of this software portability layer can be found on Arm's official GitHub at the following link: https://github.com/ARM-software/CMSIS-FreeRTOS.

It is recommended to use the software pack provided at the link above to import the latest version of the CMSIS-RTOS2 layer into your IDE of choice.

4 Porting Components

The software package included in this application report contains a full implementation of the CC3120 host driver and a standard Arm CMSIS-Pack that allows the you to easily add the host driver to your project in your IDE of choice. Additionally, a simple code example is provided in source that provides a templated solution for the programmer to adapt for their host microcontroller. The host driver includes pre-ported software interactions with both CMSIS-Drivers and CMSIS-RTOS2 and does not require any porting effort by you. The templated project contains a simple provisioning code example as well as templated configuration files that contain stubs where programmers will plug in their device-specific configuration and setup logic. Each relevant file and a small description of that file's purpose are described in Table 1.

Table 1. Host Driver and User Files

File	Description
cmsis_driver_config.c	Configuration settings specific to the SPI driver such as bit rate and polarity settings
user_port.c/user_port.h	Settings specific to the host microcontroller. Can contain initialization function for drivers, pin settings, clock rates, etc.
cc_pal.c/cc_pal.h	Ported implementation of CC3120 host driver to use CMSIS-Drivers and CMSIS-RTOS2 APIs
user.h	Contains macro definitions that map common API calls in the host driver to the specific implementation in cc_pal.c as well as any RTOS specific calls

5 Cross Platform Portability Considerations

While CMSIS-Drivers and CMSIS-RTOS2 do abstract out a significant amount of embedded functionality and RTOS interaction, there are certain device-specific components that need to be adapted in order to match the host microcontroller. These components include device-specific options such as pin configurations, device clocking, and the setup of hardware specific RTOS components. Each device dependent configuration component is described in detail below.

The main file that contains all relevant device-specific configuration and implementation logic is the *user_port.c* file that is included in the example project. It is recommended that all clock configuration, pin assignments, peripheral settings, and any other required device-specific configuration outside of the CMSIS-Driver layer is segregated to this file. Specifically in this file, the *CMSIS_CC31XX_userInit* function exists to execute any one-time device configuration logic that is required prior to the Wi-Fi NWP powering up.

For managing interrupts from the CC3120 and managing the synchronization signal from the network processor (the IRQ line), it is required for your application to provide logic to handle an interrupt on the IRQ line. General-purpose input/output (GPIO) interaction is a very device-centric function and does not make sense to abstract out to the CMSIS-Driver level due to the amount of software overhead that would be required to perform a relatively simple function. For that reason, it is required to provide an interrupt handler in the user_port.c file that in turn calls the appropriate interrupt handler in the host driver (which is exposed as a global function). An example of this interrupt handler can be seen in Figure 3.

```
void EXTI15_10_IRQHandler(void)
{
    /* Make sure that interrupt flag is set */
    if (__HAL_GPIO_EXTI_GET_IT(GPIO_PIN_12) != RESET)
    {
        /* Call the handler and clear interrupt flag */
        g_Host_irq_Hndlr(0);
        __HAL_GPIO_EXTI_CLEAR_FLAG(GPIO_PIN_12);
    }
}
```

Figure 3. IRQ Interrupt Handler Example

In this example, the *EXTI15_10_IRQHandler* is the function that is plugged into the NVIC table and is the function that is called by the CPU when an interrupt happens on the IRQ line (in this case GPIO_PIN_12). The interrupt handler makes device-specific HAL calls to clear the relevant interrupt flag as well as calls the *g_Host_irq_Hndlr* function that is exposed from the CC3120 host driver. This global function communicates back with the host driver that an interrupt occurred and proceeds to progress the state machine of the Wi-Fi host driver. Additionally, it is required for your application to define two separate API functions in the *user_port.c* file that tells the application how to turn on and off the nHIB line going from the host microcontroller to the CC3120. This signal is used by the host microcontroller to control the hibernation mode on the network processor. A reference implementation of the two functions required to be implemented can be seen in Figure 4.

```
void CMSIS_CC31XX_turnOnNWP(void)
{
    HAL_GPIO_WritePin(CMSIS_CC31XX_HIB_PORT,
                      CMSIS_CC31XX_HIB_PIN,
                      GPIO_PIN_SET);
    /* Wait 5msec */
    osDelay(5);
}

void CMSIS_CC31XX_turnOffNWP(void)
{
    HAL_GPIO_WritePin(CMSIS_CC31XX_HIB_PORT,
                      CMSIS_CC31XX_HIB_PIN,
                      GPIO_PIN_RESET);
    /* Wait 5msec */
    osDelay(5);
}
```

Figure 4. Turn On/Off NWP Implementation

This code example contains example implementations of both the *CMSIS_31XX_turnOnNWP* function as well as the *CMSIS_C31XX_turnOffNWP* function. Both of these functions are defined as extern functions in the CC3120 Wi-Fi driver and require definition in your application in order to link/build. These functions have two separate operations: an operation to drive the respective nHIB pin high/low and a delay function. The GPIO interaction is device specific and references the specific software HAL that is responsible for driving GPIO pins high and low. The delay function can be abstracted out to the *osDelay* CMSIS-RTOS2 function.

6 Porting SPI Interaction

For the SPI driver, the entirety of the interaction with the CC3120 network processor is handled within the *source/ti/drivers/net/wifi/porting/cc_pal.c* file. This file contains function definitions that tell the underlying Wi-Fi host driver how to communicate with the device's SPI peripheral. Typically, this would contain vendor-specific calls to the underlying software driver/HAL. However, in the implementation provided, the vendor proprietary calls have been switched over to use portable and standard CMSIS-Driver APIs. The SPI CMSIS-Driver initialization code from *spi_Open* is shown in Figure 5.

```
    int32_t status;
    ARM_DRIVER_SPI *spiHandle = curDeviceConfiguration->spiHandle;

    /* Opening the driver */
    status = spiHandle->Initialize(cc3120SpiCallback);

    if(status != ARM_DRIVER_OK)
    {
        return NULL;
    }

    /* Power up the SPI peripheral */
    status = spiHandle->PowerControl(ARM_POWER_FULL);

    if(status != ARM_DRIVER_OK)
    {
        return NULL;
    }

    /* Configure the SPI driver */
    status = spiHandle->Control(curDeviceConfiguration->spiConfigMask,
                                curDeviceConfiguration->spiBitRate);

    if(status != ARM_DRIVER_OK)
    {
        return NULL;
    }
```

Figure 5. Ported Initialization Code

In this code snippet, the SPI CMSIS-Driver is initialized using the *Initialize* function. The argument to this function is a pointer to a callback function that occurs when events such as a transaction complete occurs in the underlying SPI driver. After the initialization function is called, the power control function is invoked to power on the underlying SPI peripheral. After powered, the relevant user configuration is passed to the underlying SPI driver. These configuration settings reside in the *examples/provisioning/cmsis_driver_config.c* file that is provided in the software package.

```
WIFICMSIS_HWAttrsV1 wifiCMSISHWAttrs =
{
    .spiHandle = &Driver_SPIx,
    .spiConfigMask = ARM_SPI_MODE_MASTER | ARM_SPI_CPOL0_CPHA0 |
                     ARM_SPI_MSB_LSB | ARM_SPI_SS_MASTER_SW |
                     ARM_SPI_DATA_BITS(8),
    .spiBitRate = 20000000,
};
```

Figure 6. Wi-Fi Driver Settings

These SPI parameters are populated with default parameters that work with the CC3120's SPI peripheral. The Driver_SPIx parameter is used to specify which specific SPI peripheral is to be used with the application and is defined in the *examples/provisioning/user_port.h*. This parameter is dependent on the programmer's host microcontroller and is documented as a part of the device's CMSIS-Driver implementation. By default, the bit rate of the SPI clock is set to the CC3120's max bit rate of 20 MHz. This value will potentially change depending on the limitations of the host microcontroller's SPI peripheral and clocking setup.

All SPI transactions are done using both the *spi_Read* and *spi_Write* function declarations that are included in cc_pal.c. The definition of these functions can be seen in Figure 7.

```
int spi_Read(Fd_t fd, unsigned char *pBuff, int len)
{
    int32_t status;

    /* Queueing the transfer and waiting for completion */
    fd->Control(ARM_SPI_CONTROL_SS, ARM_SPI_SS_ACTIVE);
    status = fd->Receive(pBuff, len);
    if(status != ARM_DRIVER_OK)
    {
        return 0;
    }

    osSemaphoreAcquire(transferSemaphore, osWaitForever);
    fd->Control(ARM_SPI_CONTROL_SS, ARM_SPI_SS_INACTIVE);

    return fd->GetDataCount();
}

int spi_Write(Fd_t fd, unsigned char *pBuff, int len)
{
    int32_t status;

    /* Queuing the transfer and waiting for completion */
    fd->Control(ARM_SPI_CONTROL_SS, ARM_SPI_SS_ACTIVE);
    status = fd->Send(pBuff, len);

    if(status != ARM_DRIVER_OK)
    {
        return 0;
    }

    osSemaphoreAcquire(transferSemaphore, osWaitForever);
    fd->Control(ARM_SPI_CONTROL_SS, ARM_SPI_SS_INACTIVE);

    return fd->GetDataCount();
}
```

Figure 7. SPI Communication Functions

For both the read and write functions, the corresponding transaction is passed into the corresponding CMSIS-Driver *Receive* and *Send* functions. Before each function is called, a *Control* command is given to toggle the chip select (CS) pin either high or low depending if there is activity occurring on the line. Both the CMSIS-Driver transaction functions are non-blocking, meaning once the function has been called and the transaction has been queued the function returns before the transaction has been completed. To provide a mechanism of synchronization after the SPI transaction functions have been called, an immediate call to the CMSIS-RTOS2 function *osSemaphoreAcquire* is given. Once the SPI transaction has been completed, a call to *osSemaphoreRelease* is given from the SPI driver's callback function.

```
static void cc3120SpiCallback(uint32_t event)
{
    switch (event)
    {
    case ARM_SPI_EVENT_TRANSFER_COMPLETE:
        /* Success: Wakeup Thread */
        osSemaphoreRelease(transferSemaphore);
        break;
    case ARM_SPI_EVENT_DATA_LOST:
        __breakpoint(0);
        break;
    case ARM_SPI_EVENT_MODE_FAULT:
        __breakpoint(0);
        break;
    }
}
```

Figure 8. CMSIS Driver Callback

7 Porting RTOS Components

The majority of the RTOS interaction, in the included port of the CC3120 Wi-Fi driver and example application has already been ported to use CMSIS-RTOS2. Specifically, the included reference applications are setup to use the FreeRTOS implementation of CMSIS-RTOS2. While not required, for consistency, it is recommended to use CMSIS-RTOS2 calls in the user application to match what is implemented in the host driver. For example instead of the FreeRTOS specific *xSemaphoreTake* function it is recommended to use the CMSIS-RTOS2 *osSemaphoreAcquire* function.

One important component to be aware of is the *TimerGetCurrentTimestamp* function in the *source/ti/drivers/net/wifi/porting/cc_pal.c* porting file. The implementation provided with this document can be seen in Figure 9.

```
unsigned long TimerGetCurrentTimestamp(void)
{
    return osKernelGetTickCount();
}
```

Figure 9. Timestamp Implementation

The *osKernelGetTickCount* function is an abstracted function from CMSIS-RTOS2 that gets the tick count of the underlying RTOS's system timer. The system timer for the underlying RTOS can be the SysTick timer that is part of the Cortex-M, however, it can also be changed to be a device-specific timer (some vendors reserve the SysTick for HAL/driver interaction). Regardless of the time base that is used, the following #define statements in the *user.h* must be adjusted, see Figure 10.

```
#define _SlFd_t                    Fd_t
#define SL_TIMESTAMP_TICKS_IN_10_MILLISECONDS    (10000)
#define SL_TIMESTAMP_MAX_VALUE                   (0xFFFFFFFF)
```

Figure 10. Timestamp Setting

The *SL_TIMESTAMP_TICKS_IN_10_MILLISECONDS* define is critical to ensuring that the underlying host driver has a correct time base to work with. This definition is used to determine timeouts to the Wi-Fi NWP and an incorrect setting leads to a *SL_DEVICE_EVENT_FATAL_SYNC_LOSS* being returned by the Wi-Fi host driver. The value of this macro corresponds to how many ticks of the underlying RTOS's system timer are in 10 ms. In the example above, the system timer is running at 1 MHz so the value of the definition is set to 10000. To calculate the value for your system clock, take the period of your clock base (0.000001s for 1 MHz) and divide it into 0.010s (10 ms). In the case above, this would be 0.010/0.000001, which is equal to 10000.

One component of the portability layer that needs to be specifically adapted to the underlying RTOS implementation is the way dynamic memory is allocated. For the CC3120 host Wi-Fi driver, there are several dynamically allocated structures that are used internal to the driver. While CMSIS supports static memory object, dynamic memory allocations that are allocated in heap space are not explicitly supported. Memory pools are supported by CMSIS-RTOS2, however, C style mallocs that accept variable lengths for allocation are not. For this reason, you have to specify your own implementations of both allocating memory and deallocating memory in the *cc_pal.h* file. By default, the correct implementations that work with FreeRTOS are used, as shown in Figure 11.

```
/*!
    \brief
    \sa
    \note           belongs to \ref configuration_sec
    \warning
*/
#define sl_Malloc(Size)                                 pvPortMalloc(Size)

/*!
    \brief
    \sa
    \note           belongs to \ref configuration_sec
    \warning
*/
#define sl_Free(pMem)                                   vPortFree(pMem)
```

Figure 11. Memory Allocation Example

The *pvPortMalloc* and *vPortFree* functions are specific to FreeRTOS and are defined in the *heap_4.c* source file. If a different RTOS configuration than FreeRTOS (such as RTX) is used these values must be changed to match the new RTOS.

8 Example Provisioning Application

To showcase the functionality of the ported Wi-Fi host driver, an example provisioning application is provided in source format. Additionally, an Arm standard CMSIS-Pack file is provided that enables you to seamlessly import the host driver into their IDE of choice. The provisioning example provided is identical to the one provided with the SimpleLink SDK Wi-Fi plugin, however, it uses the Arm standard CMSIS-Drivers and CMSIS-RTOS2 abstraction layers as opposed to the Texas Instruments SDK software ecosystem. The following steps show how to start a new project and build the provided code example in the Keil uVision 5 IDE.

1. With the Keil uVision 5 IDE, open the *Pack Installer* window using the icon on the top tool bar, as shown in Figure 12.

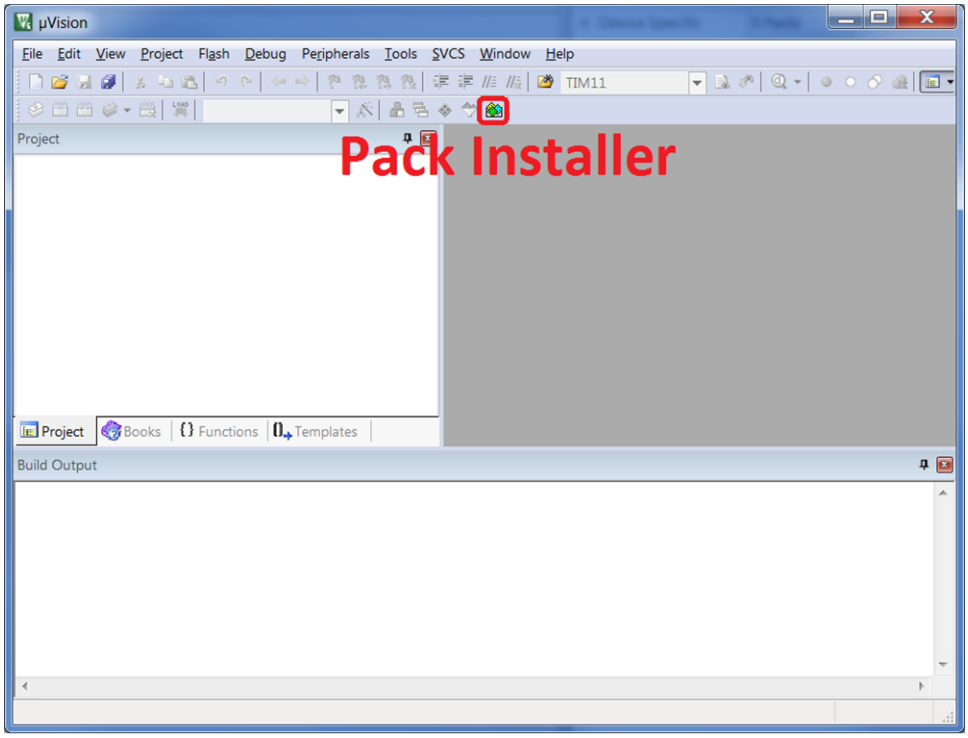

Figure 12. Pack Installer

2. From the Pack installer screen, go to *File → Import*. Then, navigate to and open the included *TexasInstruments.CC31xxHostDriver.1.0.0.pack* file that is provided with the software package, as shown in Figure 13.

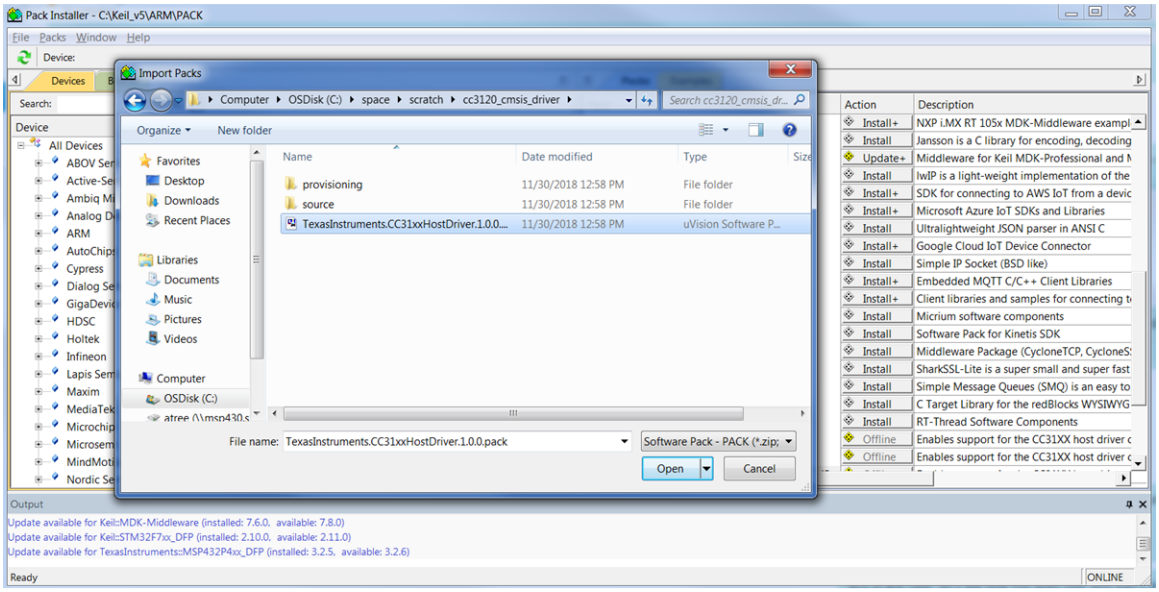

Figure 13. CC3120 Software Pack

3. Once opened, accept the licensing terms and follow the prompts to install the software pack. Once installed, the relevant item shows up in the pack menu, as shown in Figure 14.

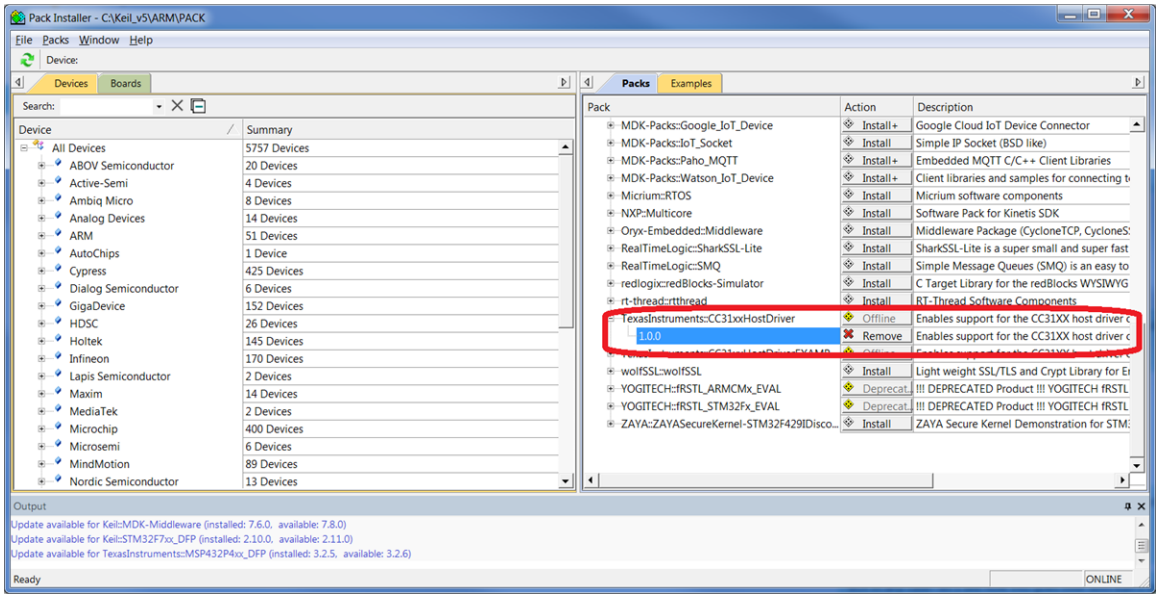

Figure 14. Installed Pack

4. Shut the Pack Installer noting that when you do uVision will prompt you to reload the packs (select yes). Create a new project and select your microcontroller of choice from the device selection screen. After you select your device in the new project wizard, the Run-Time Environment options window will open. From here, check the *Device* → *SimpleLink* → *Wi-Fi* box, as shown in Figure 15.

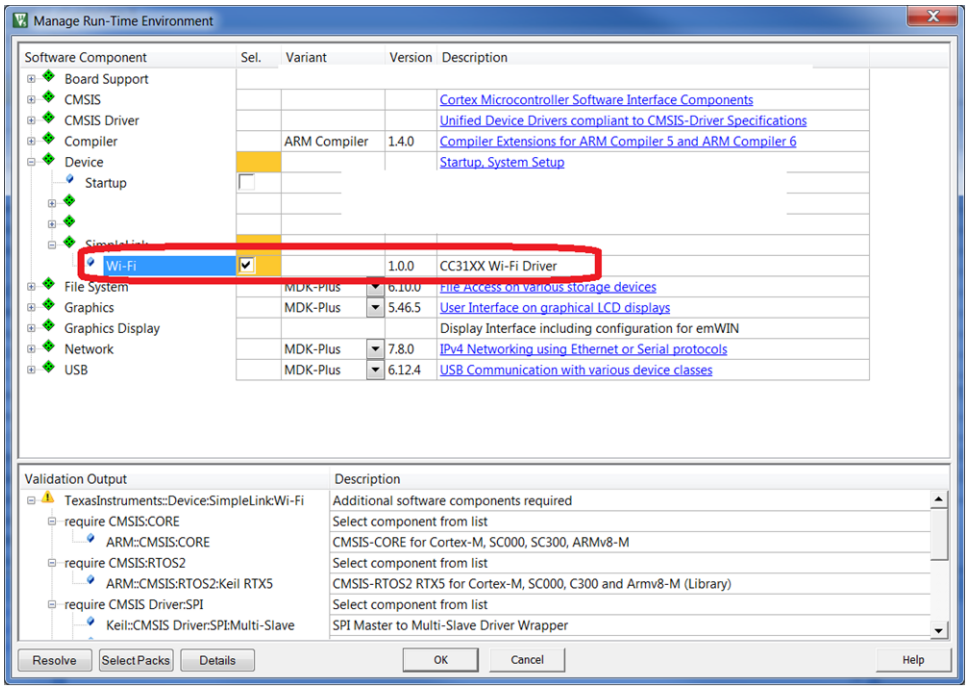

Figure 15. Wi-Fi Component

5. Note that once selected, a few validation warnings will be flagged stating that there are unresolved dependencies that need to be fixed before creating the project. As a first step to fix these dependency warnings, press the *Resolve* button on the bottom left of the window, as shown in Figure 16.

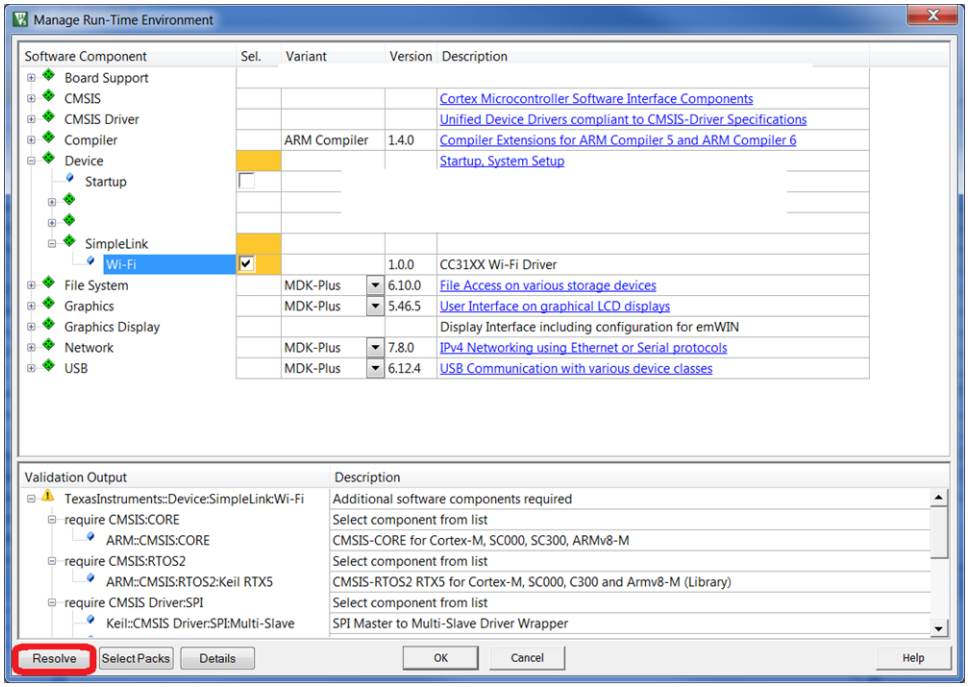

Figure 16. Resolve Dependencies

6. After you do the initial resolution of dependencies, the next step will be selecting which RTOS implementation is desired for the project. Navigate to and select *CMSIS → RTOS2 (API) → FreeRTOS*, as shown in Figure 17.

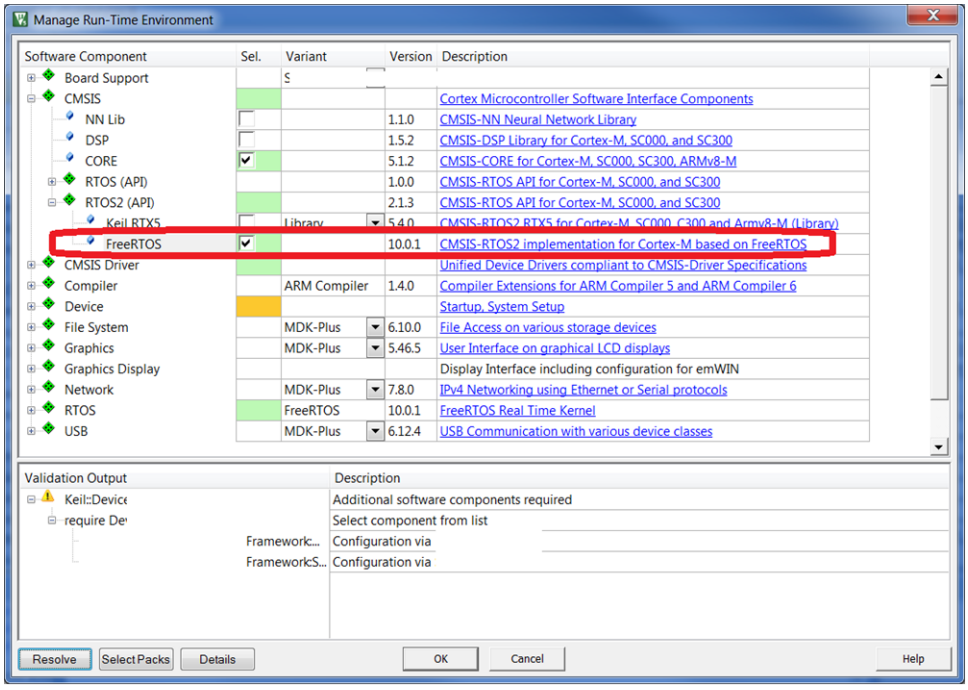

Figure 17. FreeRTOS Component

7. Any RTOS can be used (such as RTX), however the provided example project is built to run with FreeRTOS. After selecting FreeRTOS you will need to click *Resolve* again to resolve the necessary dependencies. The next step to setup the project is to enable the relevant CMSIS-Drivers configuration for your microcontroller. To do this expand both the *CMSIS Driver → SPI* and *CMSIS Driver → USART* components in the configuration tree and enable the relevant *SPI* and *USART* CMSIS-Drivers. Note that these can be enabled by default, however, take care in making sure that the implementation specific to your device is selected and not the generic Arm implementations.

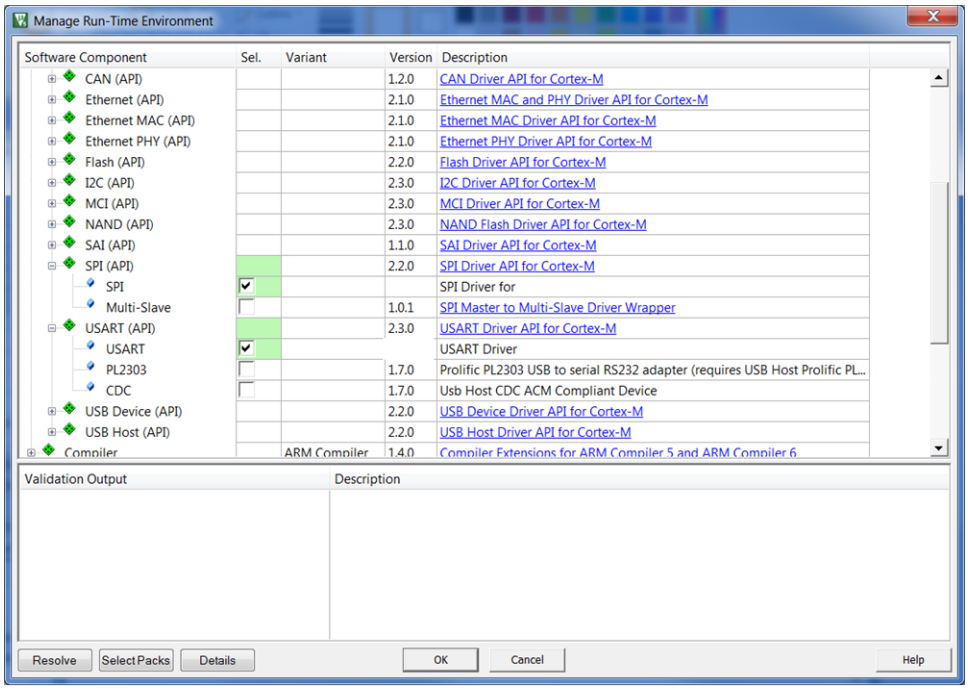

Figure 18. CMSIS-Driver Component

8. By including the CMSIS-Driver components there will likely have to be some device-specific components that will have to be included in the project such as device-specific HALs or drivers. Click *Resolve* to automatically resolve these and manually configure any vendor specific configuration components needed. After adding the relevant CMSIS-Drivers, click *OK* to create the new project and be taken to your new project. You will notice that in the project bar to the left that all of the relevant Wi-Driver source files will show up under the *Device* tree item, as shown in Figure 19.

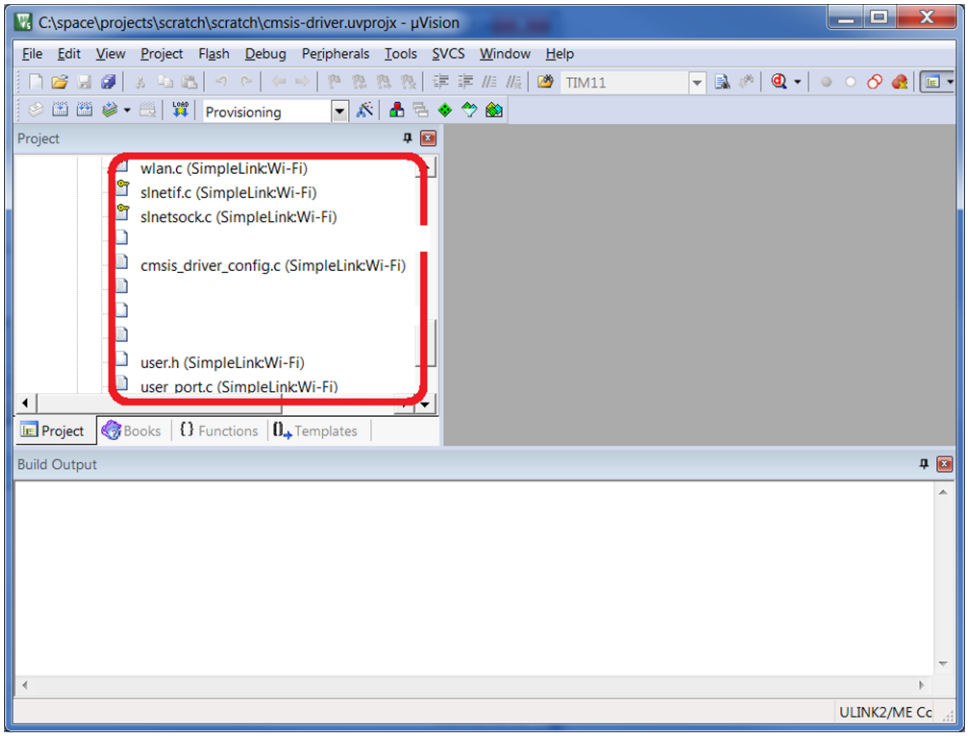

Figure 19. Device Configuration Files

9. In your new project, rename your target and source group to something more relevant, if desired. Then, right click on your target and select the *Options for Target* item, as shown in Figure 20.

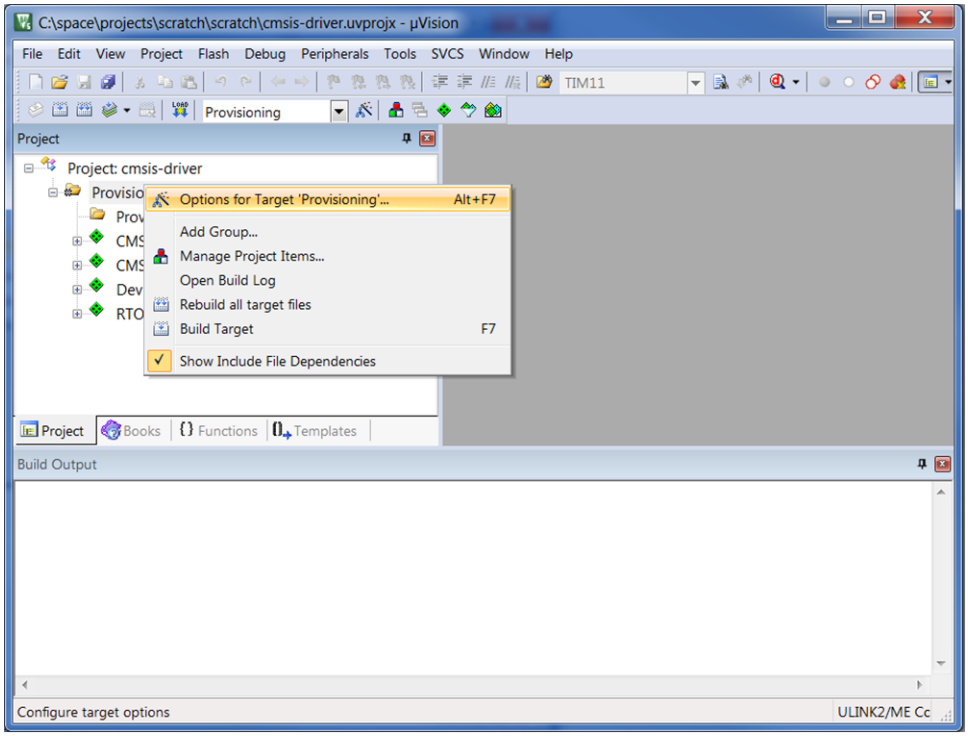

Figure 20. Options for Target

10. In the options window, select the *C/C++* tab and add the *SL_PLATFORM_MULTI_THREADED,SL_MEMORY_MGMT_DYNAMIC* defines to the *Define* text box under the *Preprocessor Symbols*, as shown in Figure 21.

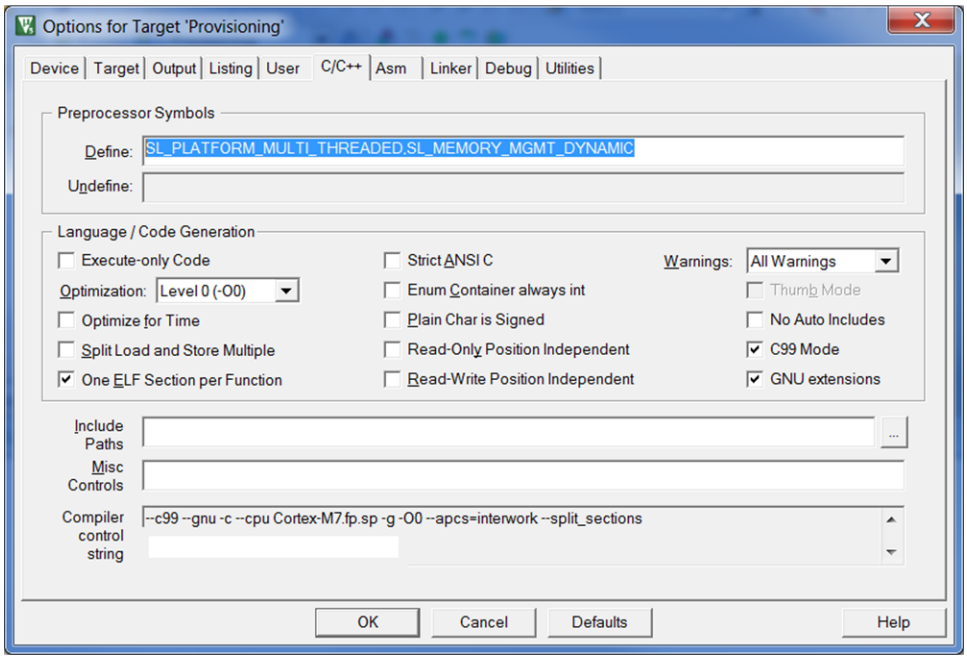

Figure 21. Additional Symbols

11. Click *OK* to save the settings and return to your project. Double click your source group, navigate to the provisioning directory included in the provided software package, and add the *uart_term.h*, *uart_term.c*, *main.c*, *main.h*, *provisioning.c*, and *provisioning.h* files, as shown in Figure 22.

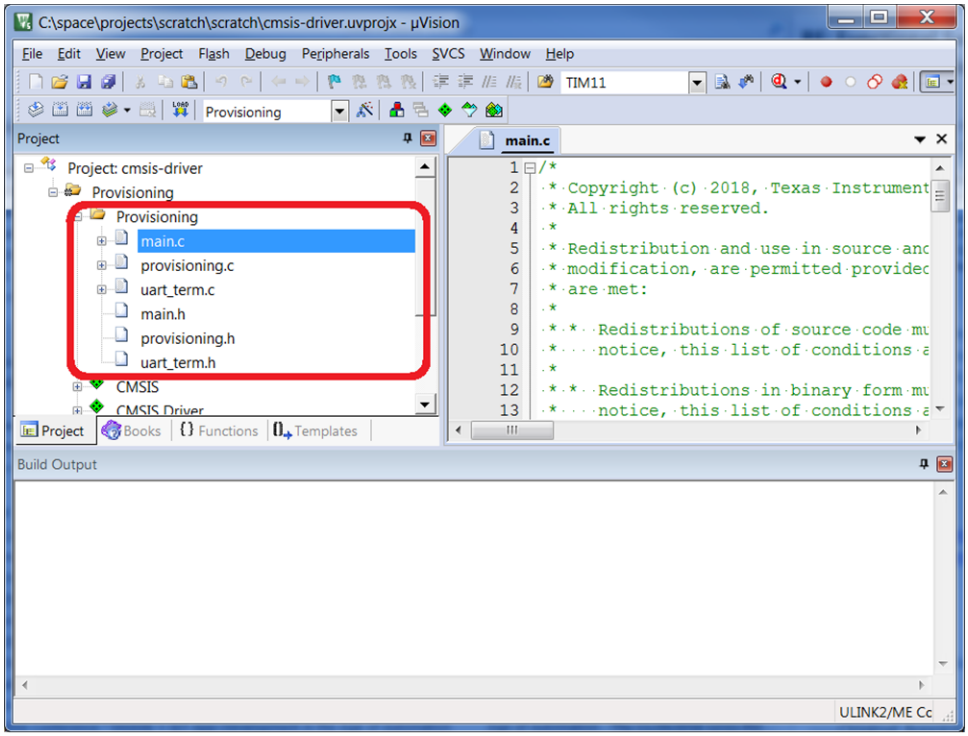

Figure 22. Example Source Files

12. Note that the *user_port.c* and *user_port.h* files that are in the *provisioning* directory do not have to be added as they are included in the CMSIS-Pack. To edit the relevant platform specific configuration files, such as *user_port.c*, navigate to them under the *Device* tree item in the project explorer.

    Once the relevant project setting changes are made and source files are added, the project is now ready to be adapted to match the underlying microcontroller. For a detailed guide on which components need to be explicitly added to the project to enable the project to build, see Section 4. The project is architected to be identical to the provisioning example that is included in the SimpleLink CC3220 SDK and the steps to run/enable the project are identical. For a reference on how to use the provisioning application, see the Wi-Fi Provisioning module on SimpleLink Academy.

9 Arduino to BoosterPack Layout

When using the CC3120 device with a host microcontroller, the CC3120BOOST BoosterPack module is the recommended hardware platform. While a BoosterPack is designed to work with the LaunchPad and SimpleLink SDK ecosystems, the host microcontrollers that support CMSIS-Drivers and CMSIS-RTOS2 usually are on evaluation kits that have Arduino Uno headers. It is possible to jumper wire the relevant connections from the CC3120BOOST to your host microcontroller, however, long jumper wires will introduce parasitic capacitance issues that will cause usability issues at higher SPI bitrates. For this reason, an example layout is provided that routes the relevant signals from the CC3120 BoosterPack to the underlying Arduino headers.

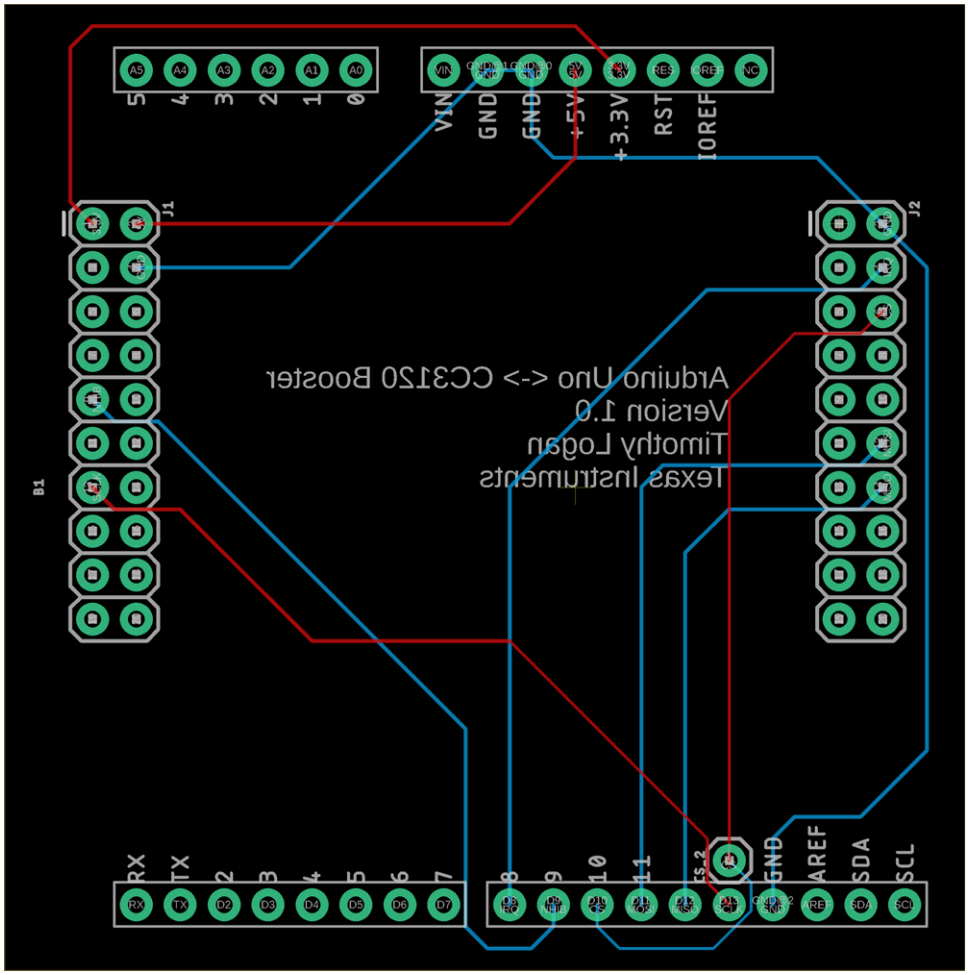

Figure 23. CC3120 BoosterPack to Arduino Uno Schematic

This layout is a passive design that simply routes the relevant SPI, GPIO, and power signals from the CC3120 BoosterPack to the Arduino Uno headers on the host microcontroller. It is provided by in both EagleCAD project format as well as Gerber files.

10 References

- SimpleLink SDK Ecosystem
- SimpleLink SDK Wi-Fi Plugin
- Texas Instruments: *CC3120, CC3220 SimpleLink™ Wi-Fi® and internet of things network process programmer's guide*

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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, 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 (www.ti.com/legal/termsofsale.html) 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.

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