Application Note

Developing Basic Applications With FreeRTOS on TM4C MCUs

Ralph Jacobi and Charles Tsai

ABSTRACT

FreeRTOS is a real-time operating system for embedded systems. It has been widely ported to many architecture platforms due to its compact size and being distributed under free open source licensing. This application report will demonstrate how to use the FreeRTOS kernel on the Texas Instruments TM4C family of Arm® Cortex®-M4F microcontrollers. Example projects are provided to demonstrate how to use fundamental FreeRTOS features and how to build basic real-world applications for TM4C's peripherals.

The source code for all example projects discussed in this application report can be downloaded from the following link http://www.ti.com/lit/zip/spma085.

Table of Contents

Trademarks

1 Introduction

The microcontroller market continues to see a steady increase in the utilization of embedded real-time operating systems (RTOS) to manage applications. The increased focus and need for RTOS solutions has led to a number of open source RTOS offerings including the widely used FreeRTOS. This application report will introduce how to use FreeRTOS with the TM4C Arm Cortex-M4F microcontroller series and provides another supported option for users to select from when choosing an RTOS to use on TM4C devices.

Included with this application report are example projects for both the TM4C123x and the TM4C129x device families. Many of these projects are based on the bare metal examples provided in the TivaWare Software Development Kit (SDK). The TivaWare™ SDK includes Driver Library (DriverLib) APIs for all peripherals that are the building blocks for any application on a TM4C microcontroller. The provided example projects illustrate how to use DriverLib APIs within the FreeRTOS kernel to create simple real-world applications with basic device peripherals. Coverage for additional peripherals is planned for future collateral releases.

The kernel examples are centered on showcasing fundamental RTOS features using the FreeRTOS kernel on TM4C micrcontrollers. These examples keep the usage of TM4C peripherals to a minimum and provide a simple starting point for new users to learn how to use the following key features of the FreeRTOS kernel.

- Scheduler: Preemptive scheduler that guarantees the highest priority thread is running
- Communication Mechanism: Semaphores, Queues, Notifications
- Critical Region Mechanism: Mutexes
- Timing Services: Software Timers

The peripheral examples do not introduce new RTOS concepts but instead extend on those foundations to build basic real-world applications. These examples demonstrate key concepts such as how to plug in hardware interrupts into the kernel and how to minimize overhead when inside a hardware interrupt that is not controlled by the RTOS scheduler. The concepts used in these examples can be adapted to many other peripherals. The peripherals that are demonstrated in this report are as follows with additional peripheral examples planned for future collateral.

- Analog to Digital Converter (ADC)
- General-Purpose Input/Output (GPIO)
- Pulse Width Modulator (PWM)
- Timer
- Universal Asynchronous Receiver/Transmitter (UART)
- Watchdog

2 How to Install

The following steps demonstrate how to download the latest version of TivaWare and FreeRTOS:

1. Download the latest TivaWare SDK from here and follow the installer instructions. If using the default installation path, the TivaWare SDK will be installed at `C:\ti\TivaWare_C_Series-2.2.0.295`. For the TivaWare directory structure, see Figure 2-1 (and notice there is already a `third_party/FreeRTOS directory`). The existing FreeRTOS installation in the TivaWare library is version 8.2.3. This needs to be updated to the latest version.

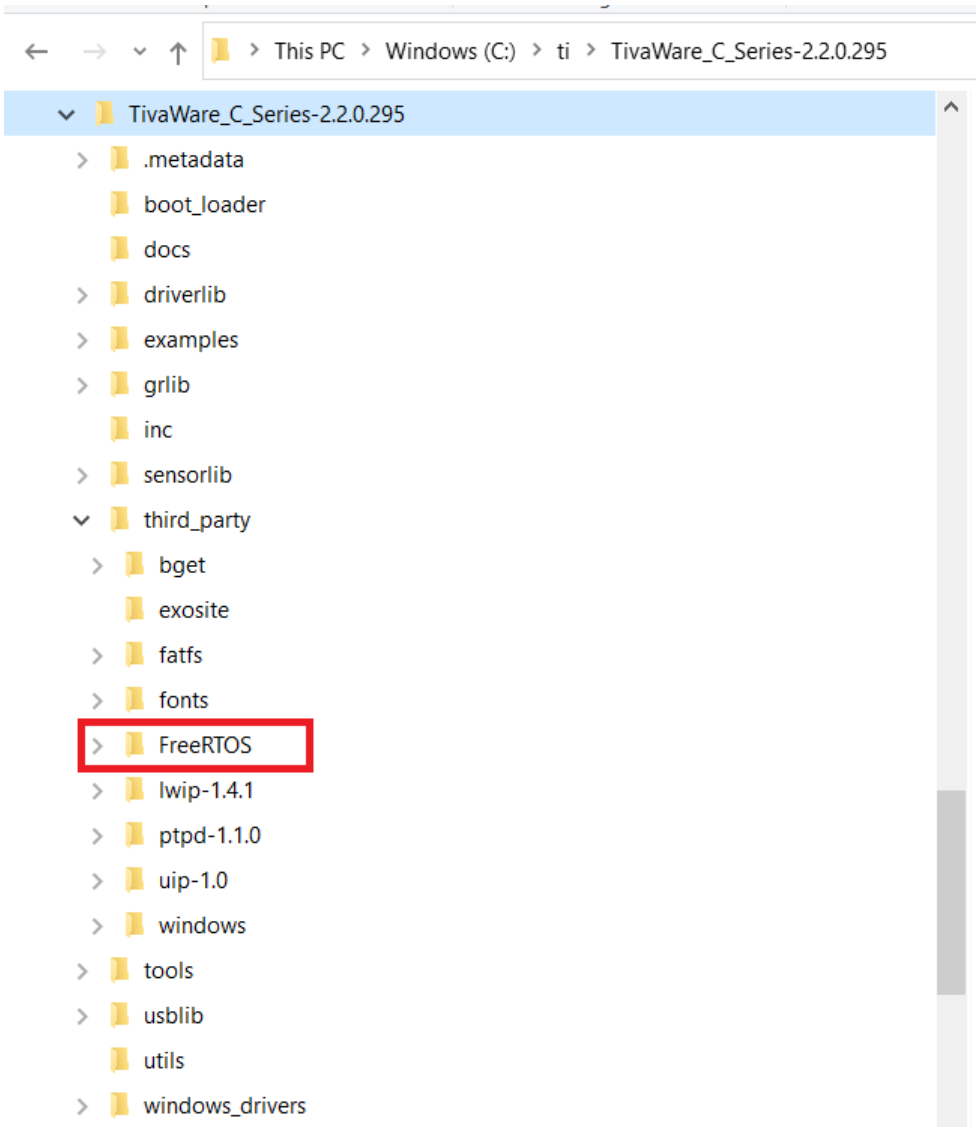

Figure 2-1. TivaWare Directory Structure

2. Download the latest version of FreeRTOS from the official download source here. At the time of the publishing of this application report, the latest version of FreeRTOS is Version 202112.00, as shown in Figure 2-2. All examples provided with this application report have been created and validated using that version.

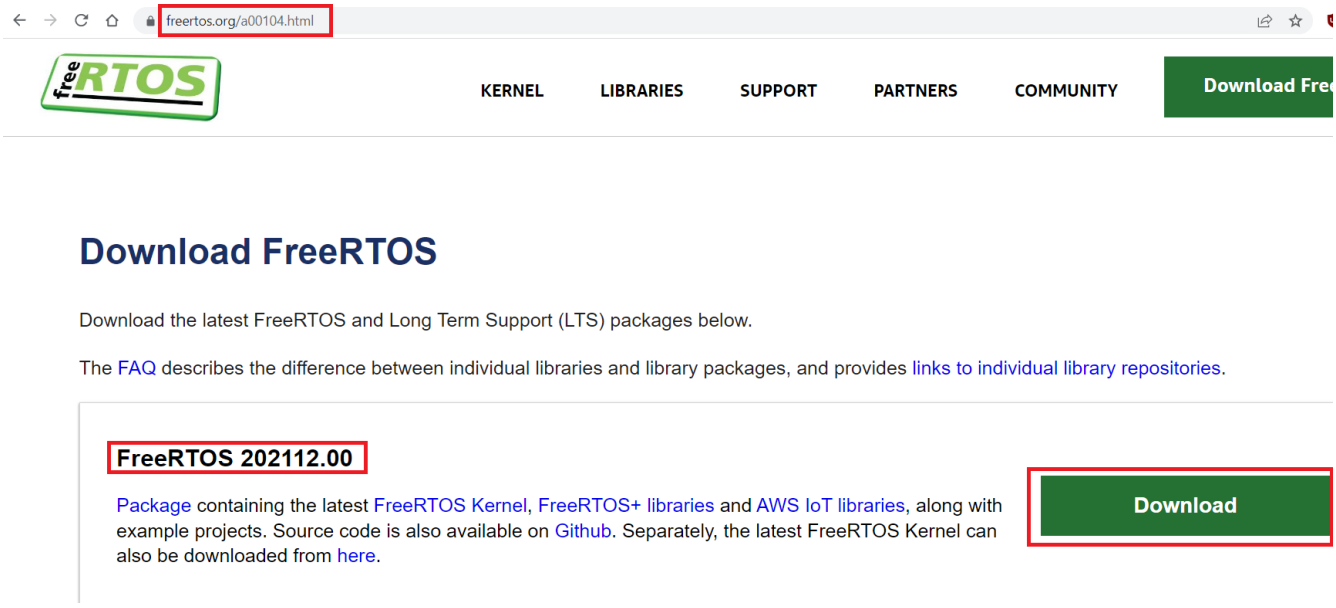

Figure 2-2. FreeRTOS Download Site

3. Unzip the FreeRTOS zip file and install to a temporary directory. View the FreeRTOS directory structure and notice the FreeRTOS folder and other folders and files, as shown in Figure 2-3.

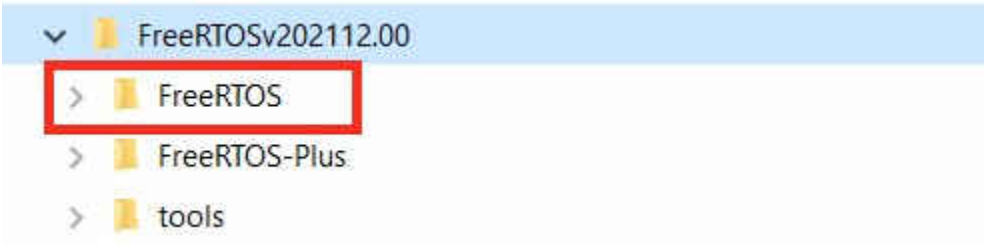

Figure 2-3. FreeRTOS version 202112.00 Directory Structure

2.1 Update the FreeRTOS Version in the TivaWare Directory

The FreeRTOS folder from the new download will need to be either moved or copied to the TivaWare library under `C:\ti\TivaWare_C_Series-2.2.0.295\third_party`. Make sure to first rename or move the existing FreeRTOS folder in the TivaWare library if there is a desire to retain it as a backup as shown in Figure 2-4.

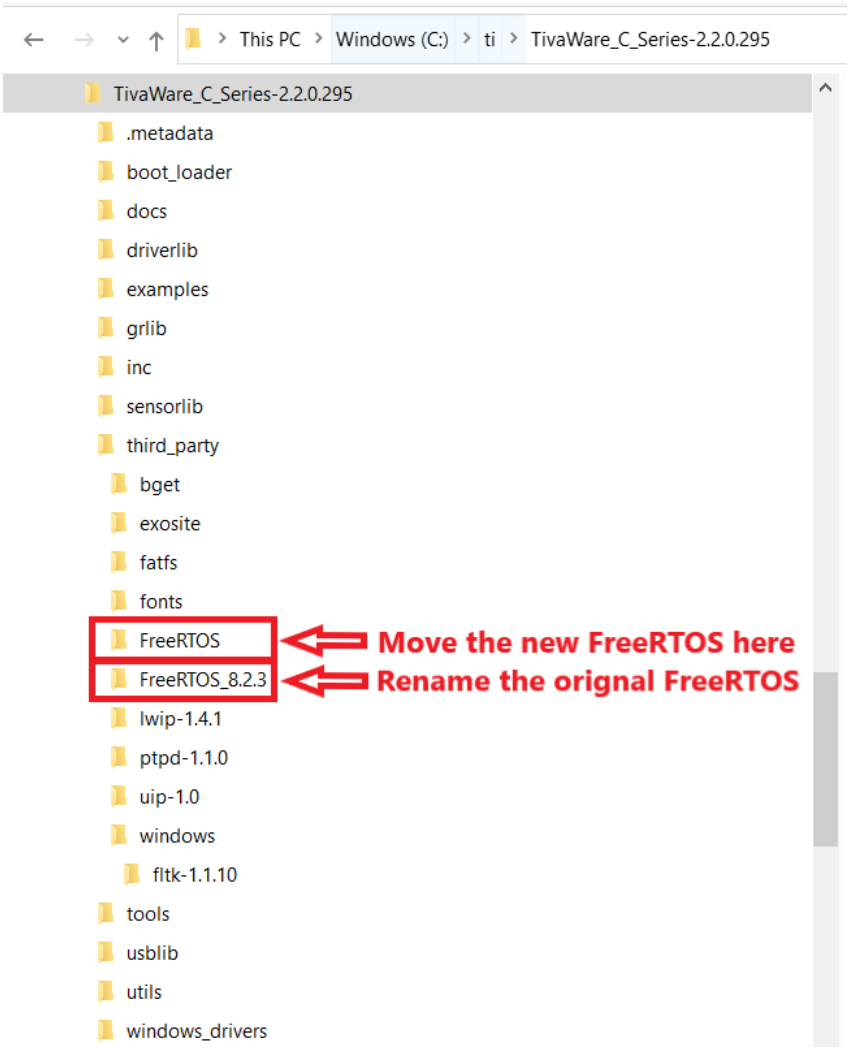

Figure 2-4. Updated TivaWare for New FreeRTOS Installation

2.2 Adding FreeRTOS Hardware Driver Files for TM4C LaunchPads

The collateral provided include new hardware driver files that were created to provide a single source for basic pinout configurations for each LaunchPad Evaluation Kit. The files are located under the `Driver Files` directory. These files are referenced in all examples that use device pins, so this step is required as part of the first time setup.

The *rtos_hw_drivers.c* and *rtos_hw_drivers.h* files for each specific LaunchPad should be copied into the TivaWare folders for that LaunchPad under the `drivers` directory. For EK-TM4C123GXL this path would be `TivaWare_C_Series-2.2.0.295\examples\boards\ek-tm4c123gxl\drivers` and for EK-TM4C1294XL this path would be `TivaWare_C_Series-2.2.0.295\examples\boards\ek-tm4c1294xl\drivers`.

3 Architecture for TM4C FreeRTOS Examples

The provided example projects are designed with the purpose of offering a compact and streamlined foundation that can be used to begin application development. In order to achieve this, the FreeRTOS kernel is used in conjunction with the TivaWare Driver Library (DriverLib) without any additional abstraction layers added. The configuration and handling of specific peripherals are contained within dedicated task files, which allow for easy re-use across multiple projects.

At this time, POSIX is not included as part of the architecture. The FreeRTOS + POSIX solution currently is a partial implementation that is only offered via GitHub. Details about that offering can be read about at this link.

In addition to DriverLib, TivaWare also includes some utilities to complement specific applications such as outputting UART messages over a terminal or using external memory sources. Be aware that these utilities were created for bare metal applications and may include APIs that were not designed to be thread safe. For the simplicity of demonstrations, there are times certain non-thread safe APIs are used in example projects. These are clearly highlighted as non-thread safe, and they are only used to demonstrate what is being executed as part of the example project. For UART communication, a thread safe UART example is provided which can be used for any application that requires utilizing UART console messaging for either debug purposes or the final product.

3.1 Proper Clock Configuration

To ensure that the FreeRTOS kernel is operating at the correct tick rate, care must be taken to assign the correct system clock frequency of the TM4C microcontroller to the FreeRTOS kernel. The *configCPU_CLOCK_HZ* variable in `FreeRTOSConfig.h` holds the value of the system clock frequency. This variable must be updated accordingly for any system clock frequency changes. All examples provided are using the maximum supported clock speed for the target microcontroller which is 80 MHz for TM4C123x devices and 120 MHz for TM4C129x devices.

The device-specific data sheets for each TM4C microcontroller provides information about the valid clock speed options under the *Clock Control* section. There are limitations to how precise a frequency can be set based on the clock dividers. If a frequency that cannot be supported is input in a TivaWare clock configuration function, typically the next closest frequency will be configured. Improper clock configurations in terms of not accurately reflecting external crystals or which oscillator should be used can also lock a device until reset to factory condition. Lastly, certain peripherals have requirements for minimum system clock frequencies to be able to operate to specification including the ADC (for high speed sampling), USB, and Ethernet (TM4C129x family only) peripherals.

There are differences in how to handle the clock configuration between the two TM4C device families.

For TM4C123x devices, the system clock frequency is set with the `SysCtlClockSet` API and is determined only by the divider that is passed with the function call. The actual rate of the system clock can be then acquired with the `SysCtlClockGet` API. However, since the `FreeRTOSConfig.h` file requires hardcoded definitions of the clock rate, any changes to the system clock frequency will need to be validated without the kernel running before plugging in the new value for *configCPU_CLOCK_HZ*.

For TM4C129x devices, the system clock frequency is set with the `SysCtlClockFreqSet` API, which requires an input for the desired system clock frequency. It then returns the actual system clock frequency that was set based on what is the closest setting to the requested value. In each example provided, that value is stored in *g_ui32SysClock*, which can be used to verify that the correct frequency has been set. The clock frequency input provided to `SysCtlClockFreqSet` is the *configCPU_CLOCK_HZ* variable. Therefore, as long as a valid TM4C129x clock frequency is defined, only one change is needed for the project.

3.2 How to use Hardware Interrupts Alongside the FreeRTOS Kernel

An important element of managing an RTOS on a microcontroller is correctly handling the use of hardware interrupts from the microcontroller without disrupting the RTOS kernel and scheduler. Because FreeRTOS aims to be chip and compiler agonistic, there is no unique plug-in for device specific hardware interrupts. Instead, hardware interrupt processing occurs outside of the kernel and scheduler and must be handled by using very lean interrupt service routines (ISRs). Minimizing the cycles spent executing the ISR will allow the scheduler to take control of the application quicker.

Because most ISRs require the processing of a specific event that has been triggered, the FreeRTOS kernel provides methods to defer the processing of events from an ISR to a task that is part of the kernel and runs according to the scheduler. These deferred tasks have configured priorities that allow developers the same flexibility as ISR processing where priorities can be set to ensure the most important interrupts are processed first. With this method, the hardware ISR processing can be minimalized while still retaining priorities as required by an application.

To learn more details about deferred interrupt processing as well as FreeRTOS interrupt safe API and how context switching is performed by the kernel, see the *Interrupt Management* chapter in Mastering the FreeRTOS Real Time Kernel.

From a TM4C microcontroller perspective, configuring interrupts includes mapping the ISR to the interrupt vector table so that the ISR is executed whenever the hardware interrupt is triggered. There are two methods to correctly register a new interrupt to the interrupt vector table. These are both documented in full detail in the *TivaWare Vector Tables and IntDefaultHandler* section of the *Getting Started with TivaWare™ User's Guide*.

4 Example Project Walkthroughs

This application report includes a zip file containing example projects for the EK-TM4C123GXL LaunchPad Evaluation Kit and EK-TM4C1294XL LaunchPad Evaluation Kit. Each board has fifteen Code Composer Studio™ projects. The projects demonstrate how to create applications on TM4C microcontrollers by using the TivaWare SDK with the FreeRTOS kernel. They are focused on demonstrating fundamental FreeRTOS features and a few basic Arm microcontroller peripherals. Table 4-1 shows a list of examples.

Table 4-1. FreeRTOS Application Examples

TM4C129	TM4C123
adc_multi_channel	adc_multi_channel
adc_timer_trigger	adc_timer_trigger
blinky_queue	blinky_queue
hello	hello
notify_example	notify_example
queue_example	queue_example
semaphore_example	semaphore_example
timer_hw_oneshot	timer_hw_oneshot
timer_hw_periodic	timer_hw_periodic
timer_hw_pwm	timer_hw_pwm
timer_sw_led_counter	timer_sw_rgb
timer_sw_oneshot	timer_sw_oneshot
timer_sw_periodic	timer_sw_periodic
uart_thread_safe	uart_thread_safe
watchdog	watchdog

4.1 Download and Import the Examples

The collateral provided with this application report can either be unzipped into a local directory or kept in the zip format. Both formats can be imported to the CCS.

1. To import the project into CCS, first select the “File” -> ”Import”.

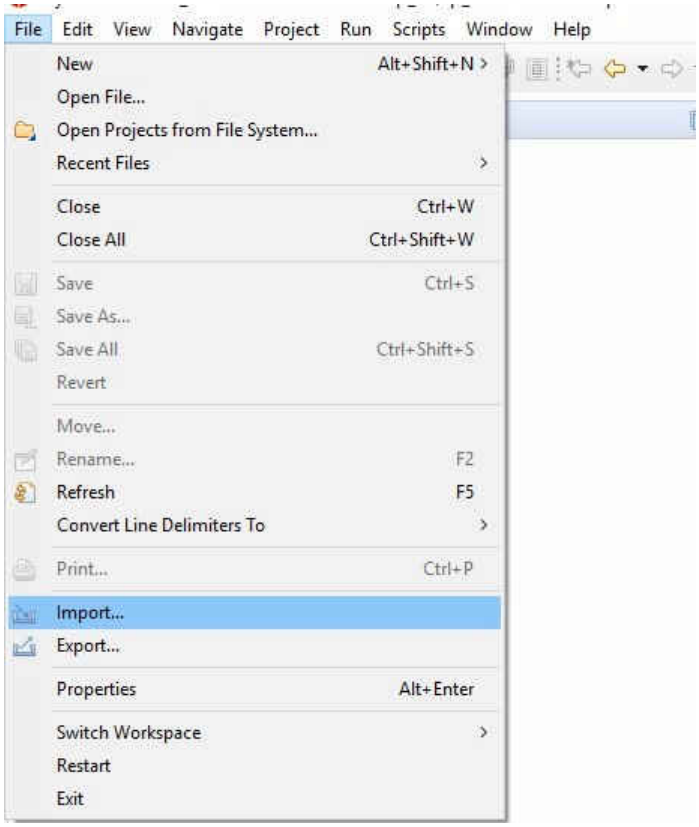

Figure 4-1. Import CCS Projects Step 1

2. Select “CCS Projects” to import the examples and then click “Next”.

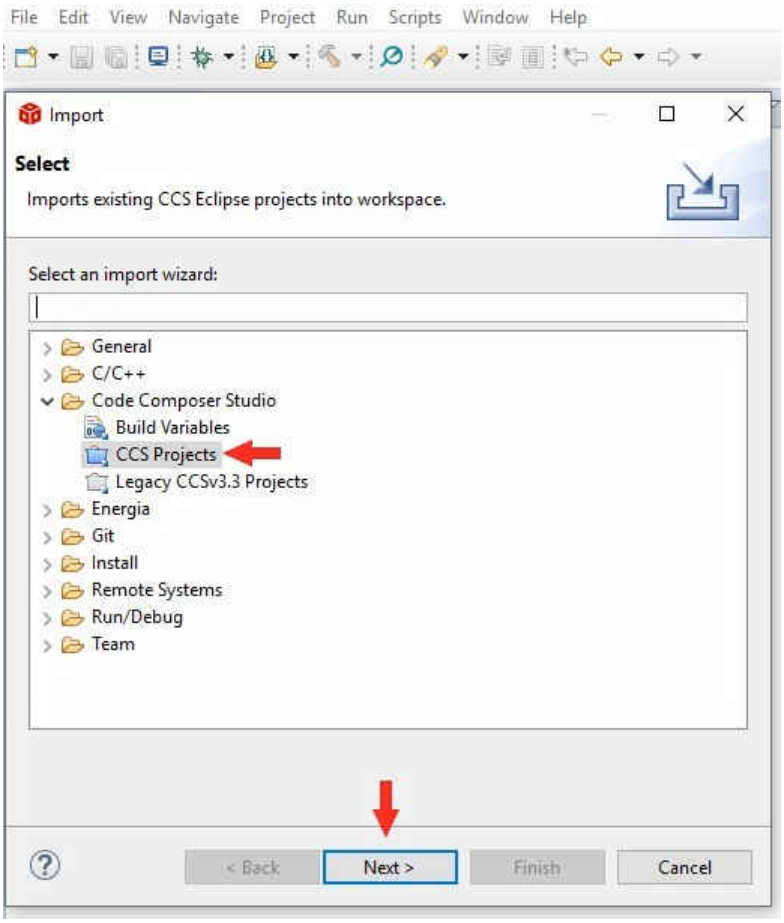

Figure 4-2. Import CCS Projects Step 2

3. Next, provide the path to either the unzipped project by selecting the first radio button or import the zip file directly by selecting the second radio button. Click the “Copy projects into workspace”.

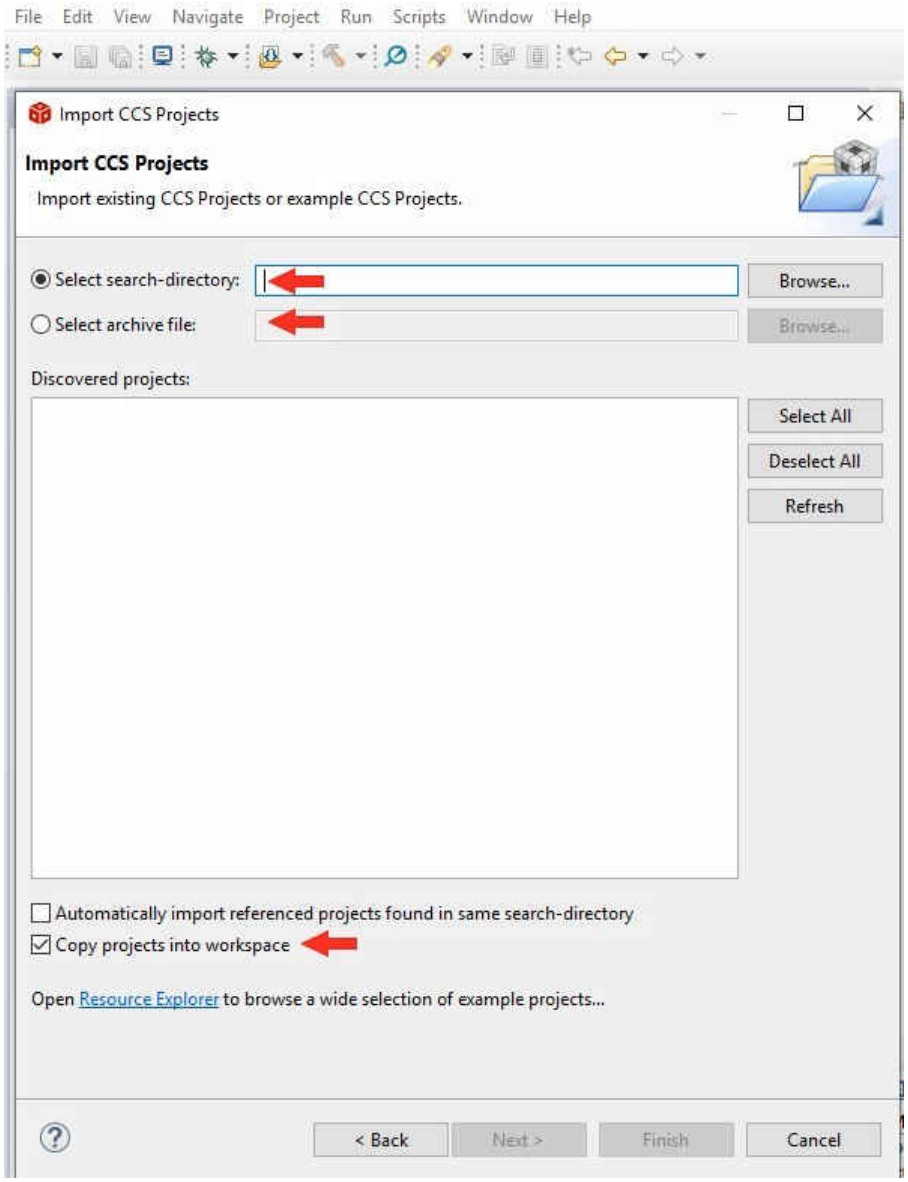

Figure 4-3. Import CCS Projects Step 3

4. After the project path is provided, a total of fifteen discovered projects will show up. First click “Select All” button and then click the “Finish” button to complete the import.

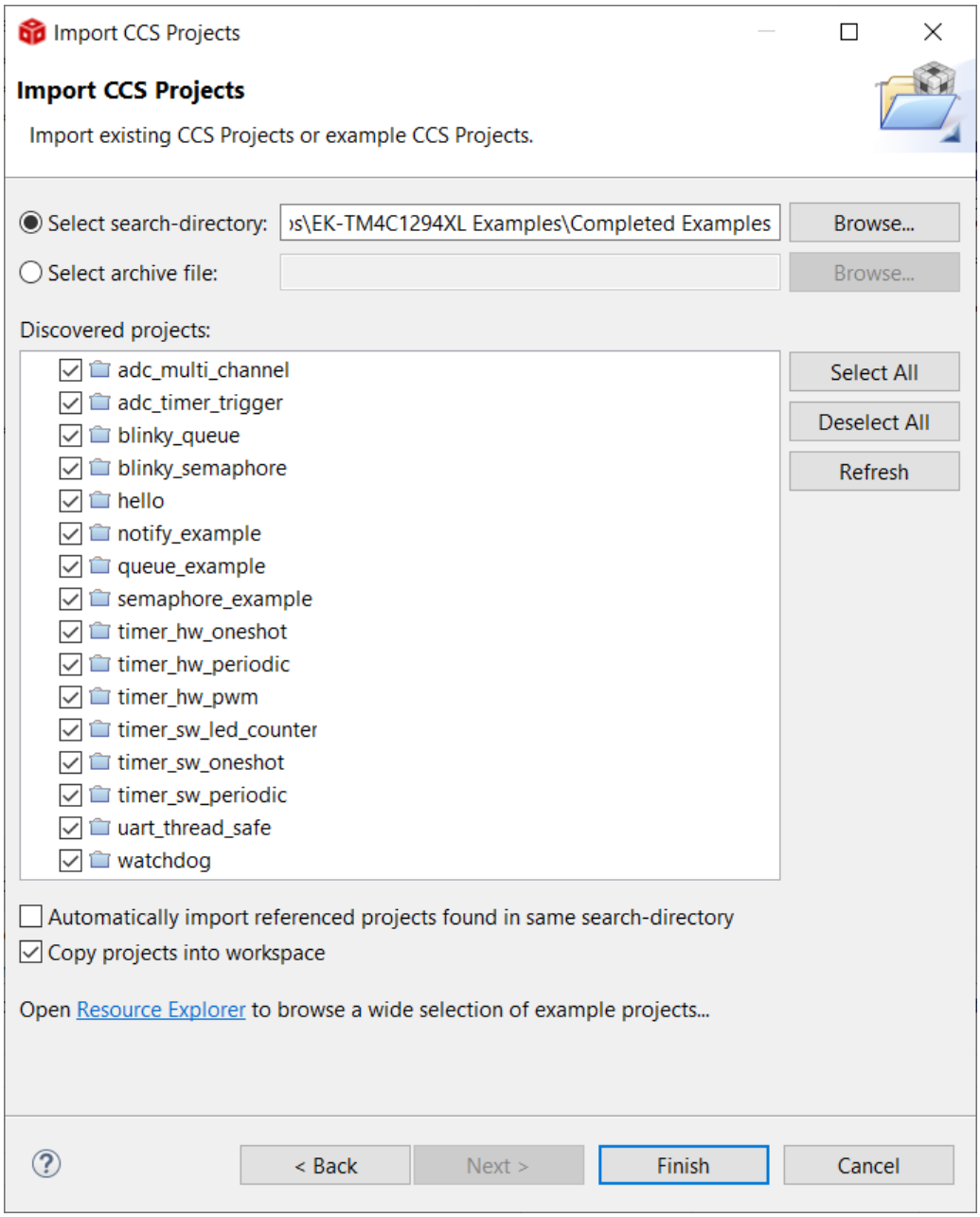

Figure 4-4. Import CCS Projects Step 4

4.2 Kernel Examples

The kernel examples are used to demonstrate many of the fundamental components of a multitasking system. The FreeRTOS APIs covered in these examples are summarized in Table 4-2. For more details on each API and additional capabilities offered by the kernel that are not covered in the provided examples, see the FreeRTOS reference manual.

Table 4-2. FreeRTOS APIs

xTaskCreate	Creates a task.
vTaskDelete	Deletes a task.
vTaskStartScheduler	Starts the FreeRTOS scheduler.
xSemaphoreCreateBinary	Creates a binary semaphore.
xSemaphoreCreateMutex	Creates a mutex type semaphore. A mutex is a special type of binary semaphore that is used to control access to a resource that is shared between tasks.
xSemaphoreTake	Obtains a semaphore that has previously been created. If a semaphore is not obtained then the task remains in BLOCKed state.
xSemaphoreGiveFromISR	Releases a semaphore that has been previously created. Calling xSemaphoreGiveFromISR makes the semaphore available to cause a task to leave the BLOCKed state.
xQueueCreate	Creates a new queue that allows items to be passed between tasks.
xQueueSend	Sends an item to a queue.
xQueueReceive	Receives an item from a queue.
ulTaskNotifyTake	A faster and lighter weight alternative to a binary semaphore, but comes with more limitations.
vTaskNotifyGiveFromISR	A faster and light weight version of xSemaphoreGiveFromISR for task synchronization.
XTimerCreate	Creates a software timer.
XTimerStart	Starts a software timer running.
xTaskGetTickCount	Returns the current tick count value which is the total number of tick interrupts that have occurred since the scheduler started.
vTaskDelay	Places the task that into the BLOCKed state for a fixed number of ticks.

4.2.1 Example: hello

The hello example illustrates the most basic FreeRTOS API which is to create a task. A task does not start until FreeRTOS scheduler starts by calling `vTaskStartScheduler`. In this example, `xTaskCreate` creates a task to print the "Hello World!" message five times on a terminal window at an one second interval. The one second interval is achieved using the `vTaskDelay` API. After five seconds, the task is then deleted using `vTaskDelete`.

While not directly related to this example, it is worth mentioning here that the FreeRTOS kernel must always have at least one task that is scheduled to run. When the scheduler is started, an idle task is created automatically to ensure there is always at least one task running. Once the *Hello* task is deleted, the kernel will run the idle task on each tick.

4.2.2 Example: notify_example

This example is used to introduce the concept of a task notification which is similar to the concept of semaphores. FreeRTOS offers task notifications as a faster and lighter weight method for synchronization between tasks in place of the binary semaphore. Two new APIs, `ulTaskNotifyTake` and `vTaskNotifyGiveFromISR`, are utilized in this example. A task calls `ulTaskNotifyTake` as to wait for the notification to arrive. Until the arrival of the notification the task is put in a BLOCKed state. In this example, the task waits for the notification in order to toggle a LED. A press on a switch on the LaunchPad board will cause an interrupt to generate from which the ISR will release the notification using `vTaskNotifyGiveFromISR`. After the notification is received, the task is unblocked and toggle the on-board LED.

4.2.3 Example: queue_example

This example is used to introduce the concept of a queue. Similar to semaphores, queues are another synchronization mechanism used for data communication between tasks. The synchronization using queues ensures that data passed between tasks are not overwritten or entered into a race condition. Three new APIs: `xQueueCreate`, `xQueueSend` and `xQueueReceive` are utilized in this example. The example also demonstrates how to pass data to the queue by value as well as by pointer (aka by reference). In this example, a total of six tasks and two queues are created. Four tasks send data with different delays to two queues while two tasks receive data and process the incoming data.

The tasks and queues present a basic producer-consumer model where one task acts as a producer that generates data which is sent to the queue while another task acts as a consumer which consumes the data coming from the queue. `xQueueCreate` API is used to create a queue object. The consumer tasks call the `xQueueReceive` API which will put the task in a BLOCKed state until data arrives in the queue. The producer tasks will send data to the queue by calling `xQueueSend`. When data is sent by a producer task, it acts as a signal for a consumer task to unblock and process the data. The consumer tasks in this example output a UART message over the terminal using the data provided from the producer task.

4.2.4 Example: semaphore_example

This example is used to introduce the concept of a semaphore. A semaphore is a synchronization method used to control access to a common resource by multiple threads or tasks in a multitasking operating sytem. Three new APIs: `xSemaphoreCreateBinary`, `xSemaphoreTake`, and `xSemaphoreGiveFromISR` are utilized in this example. `xSemaphoreCreateBinary` is used to create a semaphore. A task uses the `xSemaphoreTake` API to enter a BLOCKed state and wait for the requested semaphore to be made available. For this example, pressing a switch on the LaunchPad board will trigger an interrupt and the associated ISR will release the semaphore using `xSemaphoreGiveFromISR`. Once the semaphore is received, the task is unblocked and toggles the on-board LED(s) before going back to the BLOCKed state to wait for the next semaphore.

4.2.5 Example: blinky_queue

This example creates a simple blinky application by using two tasks and a queue. One task controls the blink rate by delaying sending data to the queue at a rate that is determined by button presses on the LaunchPad. One button reduces the delay time and speeds up the LED blink rate, and the other button increases the delay time to slow down the blink rate. The button inputs are processed with an ISR that adjusts a global variable based on the input and includes switch debouncing. The other task simply toggles the LED each time it receives data from the queue.

4.2.6 Software Timer Examples

Timers are an important feature for microcontroller as they are often used to schedule the execution of a function at a set time in the future or with a fixed frequency periodically. For this reason, almost all embedded micrcontrollers have built-in hardware timers. FreeRTOS, as part of the kernel, provides software timers that are similar in nature to the hardware timers but do not require built-in support. These software timers are not plugged into any hardware timers, instead they are sourced from the RTOS tick and fully controlled by the kernel. When a software timer expires, it executes a specified callback function. A software timer callback function should involve simple code processing to it can execute without any loops or delays. It can be assigned priority levels as needed. Because the callback is triggered each time the timer expires, it is essential that a callback function must never call FreeRTOS API functions that will result in the callback function in a BLOCKed state.

4.2.6.1 Example: timer_sw_oneshot

This example illustrates how to configure a FreeRTOS software timer to only expire once. A one-shot software timer is created using the `xTimerCreate` API with the *uxAutoReload* parameter set to *pdFALSE*. `xTimerStart` API is then called to start the timer. The `xTaskGetTickCount` API returns the total number of tick interrupts that have occurred since the scheduler was started. The callback function when executed after the oneshot timer expires will call `xTaskGetTickCount` to return the current tick count and display it on the terminal window.

4.2.6.2 Example: timer_sw_periodic

This example illustrates how to configure a FreeRTOS software timer to run periodically. A periodic software timer is created using the `xTimerCreate` API with the *uxAutoReload* parameter set to *pdTRUE*. `xTimerStart` API is then called to start the timer. Upon expiration of the timer, the callback function is called. The callback function toggles the LED on the LaunchPad.

4.2.6.3 Example: timer_sw_led_counter/timer_sw_rgb

In this example, three (*rgb*) or four (*counter*) periodic software timers are created using the `xTimerCreate` API and each timer is configured with a different frequency. `xTimerStart` API is called to start the timers. As each timer expires an associated callback function is called. These callback functions toggle the different LEDs on the LaunchPad board. For the *rgb* example on the EK-TM4C123GXL LaunchPad, this will cycle through all the LED color combinations for the on-board RGB LED. For the *counter* example on the EK-TM4C1294XL LaunchPad, this will cycle through the four LED's on the LaunchPad which act as a four-bit binary up counter.

4.3 ADC Examples

The analog-to-digital converter (ADC) module is one of the most widely used features on TM4C microcontrollers. It is a highly flexible peripheral with many configuration options and use cases. The two examples that are provided with this application report will provide fundamental building blocks that can be used for many applications by demonstrating how to sample multiple ADC channels and how to implement both polling and interrupt based ADC sampling while using FreeRTOS.

4.3.1 Example: adc_multi_channel

For this example, two tasks and one queue are created. The ADC is configured to sample a total of four channels. The start of the conversion is controlled by the processor rather than using a timer module. The first task kicks off the start of the conversion periodically using the `vTaskDelay` API, waits for the conversions to complete, and then fetches and sends the ADC data to the queue. The second task is unblocked from the BLOCKed state only when there is data available in the queue. Once the data is received, it is processed and the conversion results are sent to the terminal window.

4.3.2 Example: adc_timer_trigger

For this example, one ADC channel is configured to convert an ADC input by using a timer to trigger the conversion to start. Two tasks are created in this example. The first task is designed to receive the ADC conversion data from a queue and print it onto the terminal window. The second task is designed to wait for a notification from the timer interrupt. When the timer expires, an interrupt is generated and the notification is released. Upon receiving the notification, the second task grabs the conversion data from the ADC FIFO to store into one of two local buffers. Once a buffer is filled, it is passed to the queue as a pointer which allows the first task to access the data without transferring the entire buffer into the first task. Meanwhile the other buffer gets filled during this time to ensure there is no data corruption.

4.4 Hardware Timer Examples

As introduced earlier, the software timers are under the control of the FreeRTOS scheduler. The software timer resolution is basically the same as that for tasks. Therefore, the period of the software timers can never be smaller than the FreeRTOS ticks. Since the software timers are managed by FreeRTOS as tasks, they will consume some task stack on RAM. On the other hand, the resolution of hardware timers depend on clocks available on the hardware the application is running on. Therefore, if an application requires an execution of a future event that has a finer resolution than what the software timer can offer, then the hardware time should be used. TM4C hardware timers not only offer the normal timer functionalities like the FreeRTOS software timers, they have additional capabilities such as to generate PWM waveforms and perform timing measurement for input captures. For further details on the timer features, see the device-specific data sheet.

4.4.1 Example: timer_hw_oneshot

The *timer_hw_oneshot* example is the equivalent of *timer_sw_oneshot* implemented with the on-chip timer. In this example, one task is created to process the interrupt and it waits for a notification to unblock and execute. The expiration of the hardware timer triggers an interrupt and the ISR sends a notification to the interrupt processing task. This is done to defer the interrupt processing to a task and minimize time in the hardware ISR that is not part of the RTOS kernel scheduling. After the notification is received, the task is unblocked and prints the current tick count on the terminal window.

4.4.2 Example: timer_hw_periodic

The *timer_hw_periodic* example is the equivalent of *timer_sw_periodic* implmented with the on-chip timer. In this example, one task is created to process the interrupt and it waits for a notification to unblock and execute. The expiration of the hardware timer triggers an interrupt and the ISR sends a notification to the interrupt processing task. This is done to defer the interrupt processing to a task and minimize time in the hardware ISR that is not part of the RTOS kernel scheduling. After the notification is received, the task is unblocked and toggles the on-board LED before re-entering the BLOCKed state to wait for the next notification.

4.4.3 Example: timer_hw_pwm

This example demonstrates how to configure a single TM4C timer to operate as a split pair in PWM mode with different periods and duty cycles. It also creates a duty cycle management task that will periodically alter the duty cycle for each PWM by updating the timer match value. The task will block for a set period of time using `vTaskDelay` to ensure each duty cycle change occurs at the specified interval.

4.5 UART Example

UART is a popular asynchronous serial communication interface that is frequently used to send data between different processors or from hardware to a computer. For example, each TM4C LaunchPad has a UART to USB bridge that can be connected to a terminal / console to receive data from the UART0 port of the TM4C microcontroller. This is often used with example projects to indicate configurations, output data, or provided status updates. For developers, this can be used to output real-time debug information as well.

While UART is a very simple and easy to use serial communication method, one drawback can be the slow speeds to transfer data. Baud rates are typically set at most to be 115,200 which equates to effectively 11.5kBytes/sec. Because of this slower communication, it can take longer than expected to output a string of text. This is why even for bare metal applications, it is not recommended to output full strings of UART data within an ISR. For a system running an RTOS, the UART interface needs to be properly managed so that it is not being accessed by multiple resources at once because otherwise the output could become corrupted if the last message sent takes longer to output than the user anticipates.

4.5.1 Example: uart_thread_safe

This example highlights how to create a thread safe function by using a mutex to ensure that only one task is accessing the resource-sensitive portion of a function at a given time regardless of priorities. A thread safe function assures that it can be safely executed by multiple tasks without any potential corruption of data or resources. A mutex is useful for ensuring thread safety as it is a special type of binary semaphore that is designed to control access to resources which are shared between multiple tasks.

This example creates two tasks that send data to a terminal window through UART interface. One task will use a standard TivaWare DriverLib UART API to send data, while the other will use the TivaWare utility UART standard I/O (`uartstdio.c`) to send a string with the `UARTvprintf` API call. A mutex is created by using the `xSemaphoreCreateMutex` API to ensure the UART peripheral is only accessed by one of these tasks at a given time so each UART message is output completely before the next task is given access. The two tasks are created with different priorities and send their character strings to the terminal window at a random interval.

Without the mutex, the characters from the two tasks would have mixed together resulting in a unreadable message displayed on the terminal window. Using a mutex allows for segments of code, such as using the UART peripheral to send a message, to be guarded from access until the task acquires a token (a mutually exclusive flag) which grants access. Once acquired, the task is able to access and execute the guarded section of code. Since the token can only be acquired by a single task at a given time, this protects the UART peripheral from access until the current task is finished transmitting the message. This protection cannot be bypassed by task priority so even a higher priority task must wait until the token is released. Therefore it is critical that the task releases the mutex token immediately after the execution of the guarded section is concluded.

The example also demonstrates two different methods of how to output data on UART to give users different options. One method is a simple output of a message with minimal execution time. The other method uses a more advanced API that can process receiving variables from the application code to include in the message.

4.6 Watchdog Example

All TM4C12x microcontrollers have an on-chip watchdog timer module which can be used to generate either a non-maskable interrupt signal (NMI) or a system reset. The intention of the watchdog timer is to provide a safeguard against severe application malfunctions such as hanging or runaway code.

4.6.1 Example: watchdog

In this example, one task is created that tries to continuously clear the watchdog status register to prevent the watchdog from generating a NMI. A flag is used to determine whether or not the watchdog status register gets cleared. The default state is the flag is false so the task does not clear the watchdog status register and that results in the watchdog generating a NMI. The NMI service routine will toggle an LED each time it is entered. A press on the button will toggle the flag, which when set to true prompts the task to clear the watchdog status register continuously to prevent the watchdog expiration.

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