FreeRTOS Setup Guide

(FreeRTOSv8.0.1)

Rohit Dureja
rohit@cis.upenn.edu
University of Pennsylvania

Directory Structure

The main downloaded folder contains two subfolder folders, a quick start guide and a readme.

1. FreeRTOS – this folder contains the working files for the RTOS and demo applications.
    I. Demo – this folder contains the demo application for several MCU architectures on several different compiler flavors.
    II. License – this folder contains the licensing guidelines of FreeRTOS
    III. Source – this folder contains the actual raw working files for FreeRTOS. These need to be included in your workspace when creating a new project.
        i. include – this folder contains the kernel files for all FreeRTOS implementations.
        ii. portable – this folder contains specific files required by FreeRTOS for use with a specific microcontroller/compiler combination
        iii. Other files in this folder are list.c, timers.c, queue.c which are common to every port of FreeRTOS.

Creating a new project for Tiva Launchpad

The TivaWare which accompanies the Launchpad already consists of FreeRTOS ported to run on the ARM Cortex-M4F TM4C123GXL micro controller. The same project can be ported to other ARMM4F MCUs from the same family.

To get started, import the project into your workspace. We will be using Code Composer Studio v5.5 for this tutorial. To import go to File => Import => Code Composer Studio => Existing CCS Eclipse Projects and then navigate to the freertos_demo project in the board examples folder in TivaWare.

Once the project contents have been copied to your workspace, build the project to verify everything has been imported normally and works. If it doesn't build properly or shows errors, delete the project from the workspace and import again. Once it has built, upload the binary onto the device and use the pushbutton on the Launchpad to vary color and brightness of the onboard LED. This project has two tasks running; one is controlling the LED and the other is responding to user input on the pushbuttons.

If you are reading this it probably means your development environment is setup for writing applications using freeRTOS.

Let's go through the files in the project to get a feel of how tasks are organized in freeRTOS.

The project folder contains several .c and .h files. Of all the .c and .h files, the ones which control our application are:

1. freertos_demo – this is the main file of our project which initializes the MCU and OS, sets up the stack, and spawns tasks.

2. led_task – contains handlers for all functions responsible for the task controlling the LED brightness and color.
3. switch _task – contains handlers for all functions responsible for the task of handling user input.
4. priorities – header file used to set the priorities of running tasks.
5. FreeRTOSConfig.h – board specific file containing customizable parameters for clock speed and other MCU parameters as well as specific OS functions.
6. startup_ccs.c – startup and NVIC initialization file.
7. freertos_demo_ccs.cmd – linker file for the project.

Apart from all these files, the workspace has folders which contains the OS files, include files for device drivers among others.

Understanding the Code Structure of the Demo Application

freertos_demo.c

This file is the main file of our demo project. The file carries out the following tasks:

1. Set up the clock and source.
2. Configure the UART for sending debug data.
3. Register the semaphore for UART access.
4. Initialize the LED task.
5. Initialize the switch task.
6. Start the scheduler.

led_task.h

This header file contain the function prototype for initializing the LED task.

led_task.c

This is the working file for the LED task. It is responsible for:

1. Initializing the LED task and registering it with the OS scheduler.
2. Declaring and defining the LED task routine which is called whenever the scheduler chooses the LED task.

switch_task.h

This header file contains the function prototype for initializing the switch task.

switch_task.c

This is the working file for the switch task. It is responsible for:

1. Initializing the switch task and registering it with the OS scheduler.
2. Declaring the defining the switch task routine which is called whenever the scheduler chooses the LED task.

priorities.h

This header file contains the priority order of our two tasks; LED task and switch task.

All these files are critical for the operation of the freeRTOS demo. All projects utilizing freeRTOS as the scheduling OS need to have a similar file structure for correct operation.

Writing our first Application using freeRTOS

First of all, make a copy of the freertos_demo project in your workspace and give it any name you desire. In my case, I have named it rtos_led and will be using this name whenever I am referring to my project.

As the default freertos_demo contains additional files which are not relevant to our first application, you can delete them. This will help minimize build time. The folder you can delete from rtos_led are drivers and files you can delete are led_task.c/led_task.h and switch_task.c/switch_task.c. We will add new files for tasks in our application. You can rename freertos_demo.c to main.c in the rtos_led project.

The application we are going to write blinks two LEDs at different rates (250ms and 125ms), each running in a task of their own.

Before we start our application, there are a few functions in freeRTOS which you should know. These are:

vTaskStartScheduler

This function appears in your main() function and usually marks the point when control to carry out the specified tasks is handled to the OS. All code before this line, is bare-metal code.

xTaskCreate

This function is usually called when the task initializer comes into scope of is called from main. It registers the task routine with the scheduler and configures other parameters for the task.

Let's start by first writing our source and header files for the LED blinking at 250ms period. Let's call these files task_led1.c and task_led1.h

task_led1.h

```
#ifndef __LED1_TASK_H__
#define __LED1_TASK_H__

// Prototypes for the LED1 task.
extern uint32_t LED1TaskInit(void);

#endif // __LED1_TASK_H__
```

This file just declares the prototype of the LED 1 task initializer function which will be called from our main() function.

task_led1.c

```
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/rom.h"
#include "drivers/rgb.h"
#include "driverlib/sysctl.h"
#include "drivers/buttons.h"
#include "utils/uartstdio.h"
#include "task_led1.h"
#include "priorities.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"


// The stack size for the LED1  task.
#define LED1TASKSTACKSIZE        128         // Stack size in words

// This task blinks LED1 with a period of 250ms
static void LED1Task(void *pvParameters)
{

    while(1)
    {
       ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);
       ROM_SysCtlDelay(ROM_SysCtlClockGet()/12);
       ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);
       ROM_SysCtlDelay(ROM_SysCtlClockGet()/12);
    }
}

// Initializes the LED1 task.
uint32_t LED1TaskInit(void)
{
       ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
       ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1);

       // Create the LED task.
    if(xTaskCreate(LED1Task, (signed portCHAR *)"LED1", LED1TASKSTACKSIZE, NULL,
                   tskIDLE_PRIORITY + PRIORITY_LED1_TASK, NULL) != pdTRUE)
    {
        return(1);
    }

    // Success.
    return(0);
}
```

Well, a lot of stuff is happening in this file. The function LED1TaskInit() is the same for which we have declared the prototype for in task_led1.h. The function configures the GPIO port and pin to be used with the LED. It also

creates a new task by calling xTaskCreate() and assigning LED1Task to it (function defined above LED1TaskInit()). The LED1Task() function uses the TivaWare API to blink an LED with a delay of 250ms.

task_led2.h

```
#ifndef __LED2_TASK_H__
#define __LED2_TASK_H__

// Prototypes for the LED2 task.
extern uint32_t LED2TaskInit(void);

#endif // __LED2_TASK_H__
```

task_led2.c

```
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/rom.h"
#include "drivers/buttons.h"
#include "utils/uartstdio.h"
#include "task_led2.h"
#include "task_led1.h"
#include "priorities.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"

// The stack size for the LED2 task.
#define LED2TASKSTACKSIZE 128         // Stack size in words

// This task blinks LED2 with a period of 125ms
static void LED2Task(void *pvParameters)
{
    while(1)
    {
       ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
       ROM_SysCtlDelay(ROM_SysCtlClockGet()/24);
       ROM_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
       ROM_SysCtlDelay(ROM_SysCtlClockGet()/24);
    }
}

// Initializes the LED2 task.
uint32_t LED2TaskInit(void)
{
```

```
      ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
      ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);

      // Create the switch task.
    if(xTaskCreate(LED2Task, (signed portCHAR *)"LED2",
                   LED2TASKSTACKSIZE, NULL, tskIDLE_PRIORITY +
                   PRIORITY_LED2_TASK, NULL) != pdTRUE)
    {
        return(1);
    }

    // Success.
    return(0);
}
```

Once the program is uploaded on to the board, two of the three onboard LEDs should blink simultaneously with the programmed period.