EGH456 Embedded Systems

Laboratory Exercise 3

Multithreading and Intro to RTOS

Learning Objectives

After completing this laboratory, and following up using documentation you should be able to:

1. Become familiar with configuring and using freeRTOS libraries
2. Understand the use of a real-time kernel for embedded applications
3. Understand the need for multi-threading
4. Use multiple threads in a program
5. Understand resource sharing and semaphores
6. Use semaphores in a program

Reference Documents

- FreeRTOS Reference Manual
- FreeRTOS on TM4C MCUs
- EK-TM4C1294XL User's Guide
- TM4C1294NCPDT Datasheet

Pre-Lab Checklist (Setup)

Before creating a new project, ensure:

- TivaWare SDK is installed.
- FreeRTOS is downloaded and copied to "TivaWare_C_Series-2.2.0.295/third_party".
- Code Composer Studio recognizes FreeRTOS include paths.
- UART terminal is configured (115200 baud, 8N1).

Preparation Activities

Read the chapters on Multithreading and Shared Resources in the reference book by Lewis. Familiarise yourself with FreeRTOS and its integration with TM4C boards.

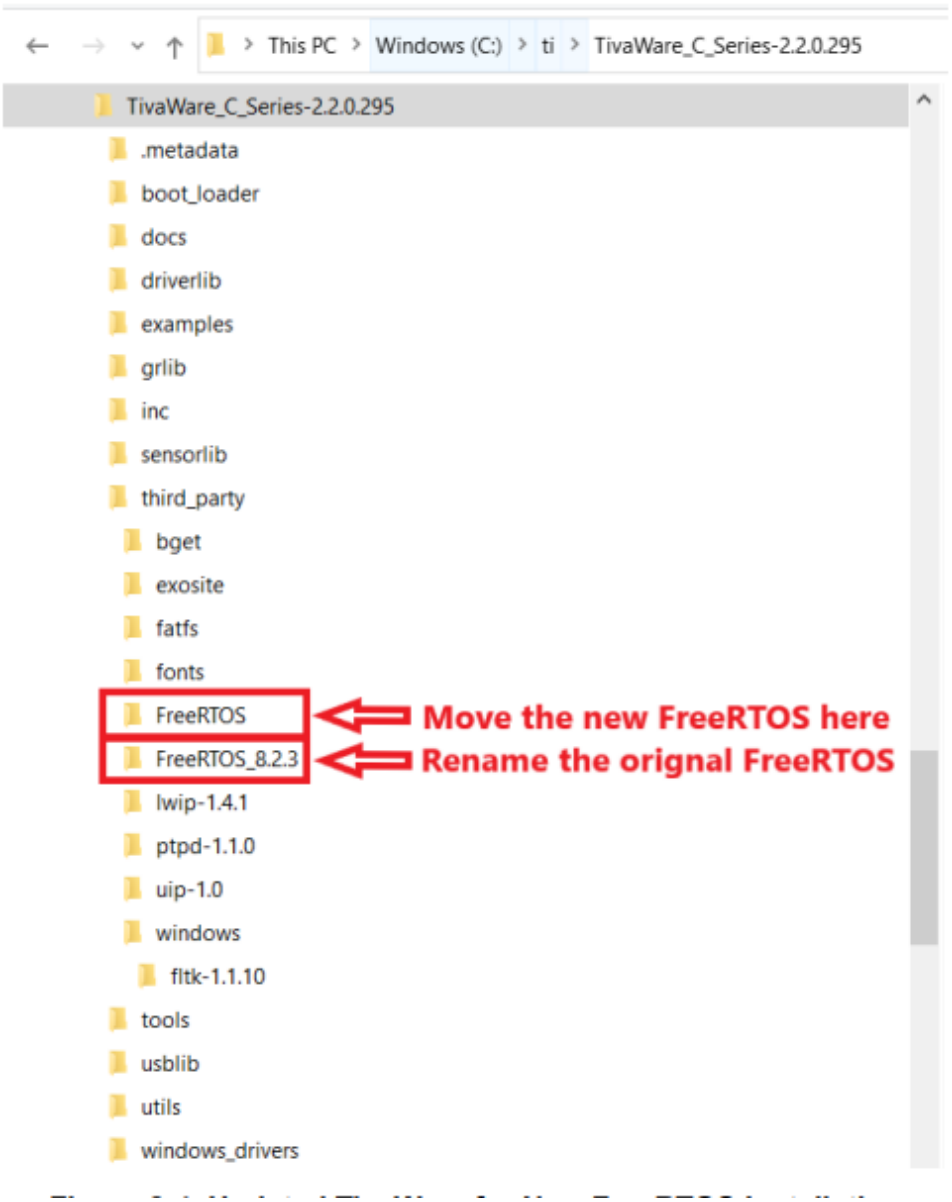
Preparation Activities

We will be using FreeRTOS to implement multi-threading and resource sharing throughout this semester. FreeRTOS is an open-source real-time operating kernel for embedded devices, which has been ported to 35 microcontroller platforms. It is important to familiarise yourself with the FreeRTOS user guide, which can be found in the reference documents. We also recommend reading the guide to using FreeRTOS on TM4C boards, which forms the basis for the workshop activities this semester.

Setup

Before creating a new project, you will need download the latest version of FreeRTOS and place it in the third party folder of the TivaWare SDK.

1. Download version 10.5.1 of FreeRTOS from the official github page link.
2. Unzip the FreeRTOS zip file and install to a temporary directory.
3. Copy the FreeRTOS folder that is within the extracted folder to the TivaWare SDK third party folder at "TivaWare_C_Series-2.2.0.295/third_party". Be sure to first remove or rename the existing FreeRTOS folder in the TivaWare library, as this is a previous version of freeRTOS that is unwanted.


This PC
Windows (C:)
ti
TivaWare_C_Series-2.2.0.295
TivaWare_C_Series-2.2.0.295
.metadata
boot_loader
docs
driverlib
examples
grlib
inc
sensorlib
third_party
bget
exosite
fatfs
fonts
FreeRTOS
FreeRTOS_8.2.3
Move the new FreeRTOS here
Rename the orignal FreeRTOS
lwip-1.4.1
ptpd-1.1.0
uip-1.0
windows
fltk-1.1.10
tools
usblib
utils
windows_drivers

Laboratory Tasks

Task A: Hello FreeRTOS (1.5 Marks)

Download and build the example project `hello_freertos` from Canvas. Ensure you have done the preparation activity for installing freeRTOS described previously. Upload/run the example on your microcontroller dev kit and verify the hello world task is printing to the serial monitor. You may need to push the reset button as the example only runs for 5 seconds before finishing. Familiarise yourself with how the example project works and then complete the following tasks:

1. Modify the hello task so that it no longer terminates after five prints, but instead continues to print the message every 500ms forever using the vTaskDelay(). Then, add a second task with the same priority that prints a different message (e.g., "Secondary task running") every 750ms using vTaskDelay(). Observe the output from both tasks. What pattern do you notice in the combined output? **(0.5 marks)**
2. Change the priority of the secondary task to be one level higher than the hello task. Observe the output again. How does task priority influence the behavior of the system? **(0.5 marks)**
3. Remove the call to `vTaskDelay()` from both task threads. Rebuild and run the program with both tasks active. What do you observe in the terminal output? Explain what is happening in terms of task scheduling and freeRTOS. **(0.5 marks)**

Task B: Semaphores (1.5 Mark)

Download and build the example project `semaphore` from Canvas. This program demonstrates the use of a binary semaphore to synchronize a hardware interrupt with a task that controls the onboard LEDs.

1. Explain how the binary semaphore prevents the task thread from running continuously in a loop. Why is this considered good practice in an embedded RTOS system? **(0.5 marks)**
2. Modify the program to split the existing LED task into two separate tasks: one that shifts the LEDs one direction when SW1 is pressed, and another that shifts them in the opposite direction when SW2 is pressed. Each task should be triggered by its own semaphore, signaled from the button's interrupt service routine (ISR). Make sure to use semaphores to coordinate safely between interrupts and tasks, ensuring protection of any critical sections. **(0.5 marks)**

3. Add a third software-only task that runs every second and prints a status message (e.g., "System OK") over UART. Ensure this task does not interfere with the button/LED behavior. What design choices did you make to ensure system responsiveness? **(0.5 marks)**

Task C: freeRTOS Game - Falling Fruit (4 Marks)

In this task, you will implement a simple arcade-style game where a player-controlled basket moves left and right at the bottom of the screen to catch falling fruit. The game will demonstrate the use of interrupts, periodic timers, task synchronization with semaphores, and display rendering within FreeRTOS.

The player controls the basket using the onboard buttons. Fruit will spawn at random positions at the top of the screen and fall down over time. The player gains one point for each fruit caught. The player starts the game with three lives, and one life is lost each time a fruit reaches the bottom of the screen without being caught. The game ends when all lives are lost. We have provided an example project (*game_example*) as a starting point for this task. Please download and use this example project to complete this task.

Your implementation must use two task threads:

- One task dedicated to managing the game logic (collision detection, scoring, lives, game state, etc.)
- One task dedicated to updating the display using the graphics library

All push-button input and timer-based events should be implemented using interrupt service routines (ISRs). ISR routines must be kept as short as possible and should only update shared state variables such as fruit position, movement direction or flags. Critical sections around shared resources should be protected via semaphores or appropriate methods.

Complete the following subtasks to implement the game:

1. Implement an intro screen that shows instructions and waits for a button press to begin the game. (**0.5 marks**)
2. Use two push buttons to move the player character (basket) left or right. This must be handled via ISRs. The ISR should only update a shared direction variable and must not perform any drawing or complex logic. Ensure to choose a reasonable speed at which the basket can move left and right (**1 mark**)

3. Use a periodic timer and it's ISR to spawn a fruit at a random horizontal position every 2 seconds. Use the same (or second) timer and timer interrupt to update the fruit's position so that it falls downward at a speed that takes approximately 4 seconds to fall from the top to the bottom of the screen. Ensure to choose a size (in pixels) of the fruit and basket that is suitable for this speed (**1 mark**)
4. Within the game logic task, detect if a falling fruit reaches the same horizontal and vertical (i.e. within the drawn basket) position as the players basket. If caught, increment the score. If missed and when the fruit reaches the bottom of the screen, decrement the player's remaining lives. End the game and return to the intro screen when all lives are lost. (**1 mark**)
5. Protect all shared variables (player position, fruit positions, score, lives, etc.) using semaphores to prevent race conditions. You do not have to use one semaphore per variable but can utilise the same semaphore if sharing these multiple variables from the same thread and ISR. (**0.25 marks**)
6. Ensure that all display updates are performed only within the display task. No drawing should be done in ISRs or the game logic task thread. Include a score and lives remaining section displayed somewhere on the screen while the game is running. (**0.25 marks**)

Fruit Falling Extension Task (1 Mark)

Implement the following advanced functions of the game to add some fun for the user.

1. Increase the falling speed of fruit every 10 seconds.
2. Allow multiple fruits to fall at once in a way that makes it possible to catch.
3. Add a temporary "basket enlargement" power-up fruit to help catch fruit more easily.

Debugging Tips

- **vTaskDelay not delaying?** Check tick rate and 'vTaskStartScheduler()'.
- **No output on terminal?** Ensure correct COM port and BAUD rate settings.
- **Build errors?** Check include paths for FreeRTOS.
- **Random crashes?** Check for stack size, objects have been initialised, correct semaphore usage or task priority conditions.

Assessment Instructions

Show your modified programs and written answers to your tutor during the lab. You may work in groups of two. Each group member must be actively participating.