Purdue University Fort Wayne

ECE 485

Lab 3: FreeRTOS Practice

Yilei (Eddie) Li, Rishi Mitra

11/12/2021

1.0 Objective

The objective of the lab is to use the given hardware design we have practiced on in the previous labs to achieve the following goals:

- Develop OS facilities for real-time applications
- Coordinate multiple foreground and background threads
- Understand FreeRTOS scheduler
- Ability to use semaphore, queue in FreeRTOS design

2.0 Hardware Design

The hardware used in this lab is a system built on breadboard using the TI TM4C123G LaunchPad board, µVision development system and the TM4C123 ARM Cortex-M4 microcontroller.

3.0 Software Design

In this lab, we design, implement and test operating system commands that implement a multiple thread environment. In real-time applications, the scheduling of tasks is critical for the proper operation of the system. There are five overall tasks to manage in Lab 3, Part 3a thru g. The word task in this lab is not a formal term, rather a general description of an overall function implemented with a combination of hardware, background threads (ISR), and foreground threads (main programs). Figure 3.1 shows the data flow for this part.

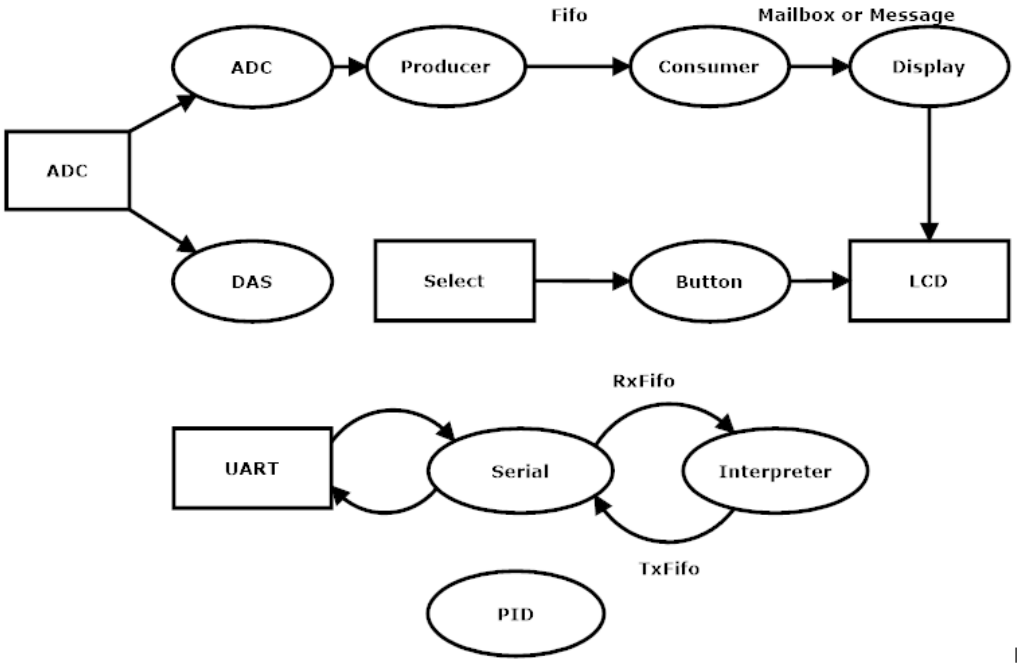

Figure 3.1 Data flow graph for the user programs

The results for all parts of the lab except for part 3e and 3f are included in the **4.0 Measurement Data** section. Hence, the result for part 3e and 3f are shown in this section.

3.1 Part 3e Result

In this part, we use FreeRTOS Task Notification API functions for communication between tasks, between task and ISR. A gatekeeper task, assigned with lowest priority, is created for LCD to display messages from push button status, FFT throughput. The resulting display on the LCD can be seen in the Figure 3.2 below.

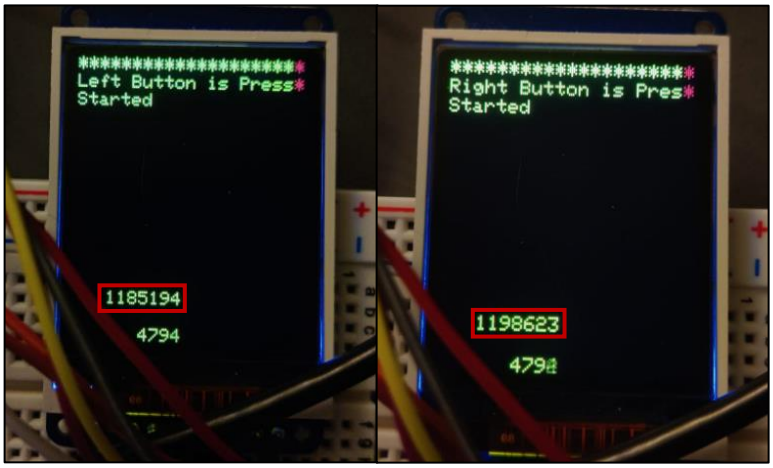

Figure 3.2 LCD display of messages from push button status and FFT throughput (in red rectangle)

3.2 Part 3f Result

The sixth task is the command line interpreter developed in Lab 1 and Lab2 using TivaWare. This task synchronizes with input and output from the UART. The execution results of available commands on the command line interpreter are shown in the figure below.

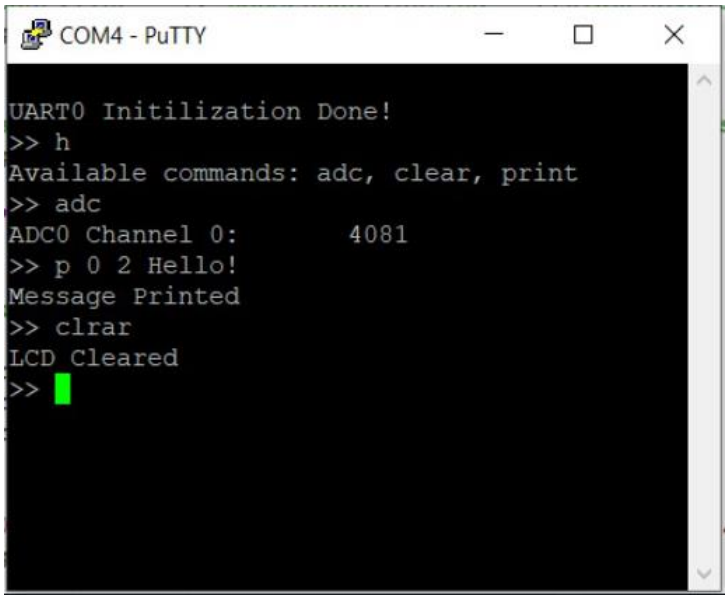

Figure 3.3 Commands successfully executed on the command line interpreter

4.0 Measurement Data

4.1 DAS System Jitter histogram (3.a)

The DAS task was activated using a general timer, periodic loading, interrupt enabled with a very high priority (=1). In the ISP of this timer, it triggers the ADC and saves it in a buffer. We generated a timestamp file using the function displaytimestamp() to measure and record the jitter for around 200 points and plot the histogram. A sample of the jitter histogram is shown below as Figure 3.4.

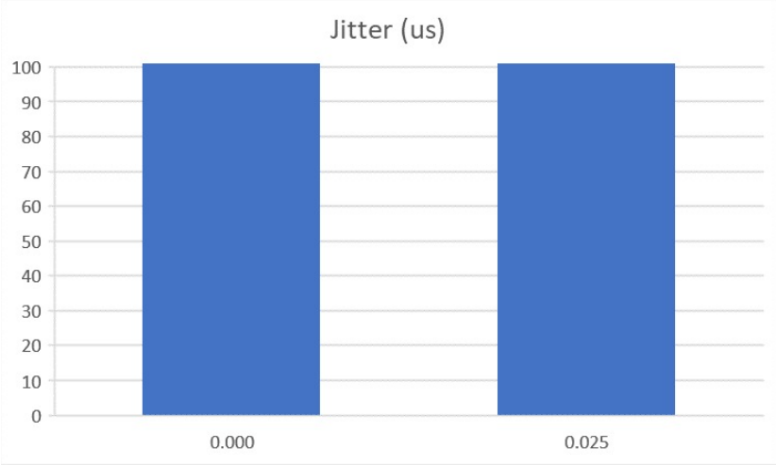

Figure 3.4 Jitter Histogram of ADC 2Khz Sampling

4.2 Push button response latency (3.b)

In this part, we created a button task which waits until a button (PF4, PF0) is pressed. We used a binary semaphore for the notification between switch ISR and a thread. A timestamp is saved for the timer in the starting of the switch ISR and the 2nd timestamp in the first instruction in the button thread after waiting for the semaphore. The time difference between r request and service is the interface latency, shown in Figure 3.5.

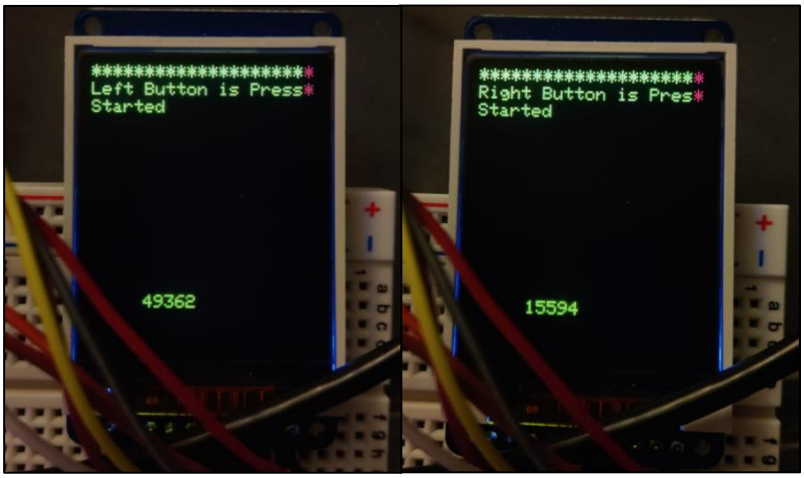

Figure 3.5 Latency measured between the button is pressed to Buttonpush thread responses

4.3 FFT calculation latency (3.c)

After 64 point is sampled, the ADC ISR notifies the FFT task through a semaphore to complete the sampled 64-point FFT calculation using cr4_fft_64_stm32.s. Similar to part 3b, we used timer 3 as the timestamp to find the time latency from 64-point completion in ADC ISR to the completion of 64-point FFT. We then recorded the latency in clock cycle. As shown in Figure 3.6, the FFT latency for this sample is **35664** clock cycles.

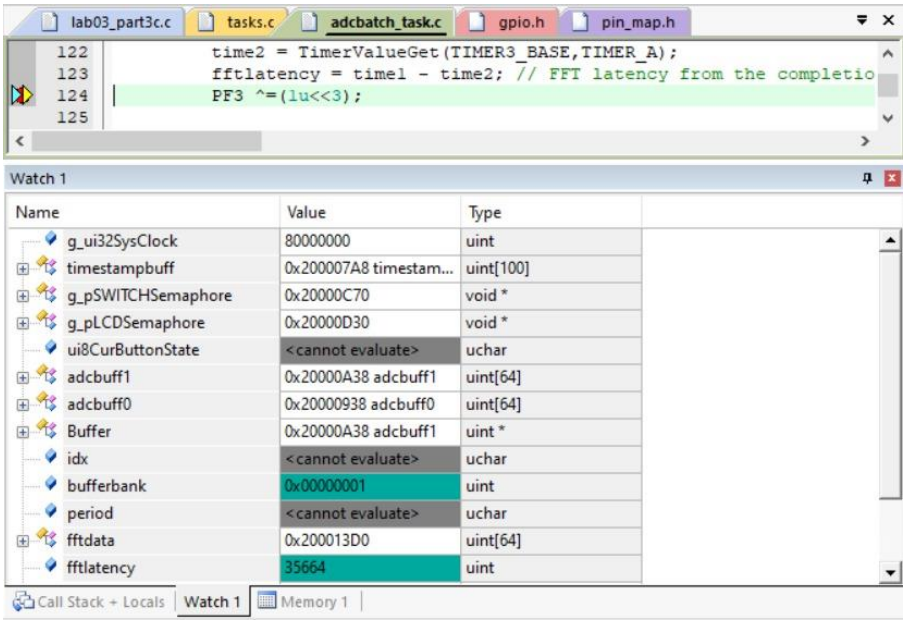

Figure 3.6 FFT Task Latency

4.4 The number of PID operations within 20 seconds (3.d)

In this lab, the variable PIDWork specifies the number of PID operations in the 20 second run. The value is displayed onto the LCD screen (Figure 3.7).

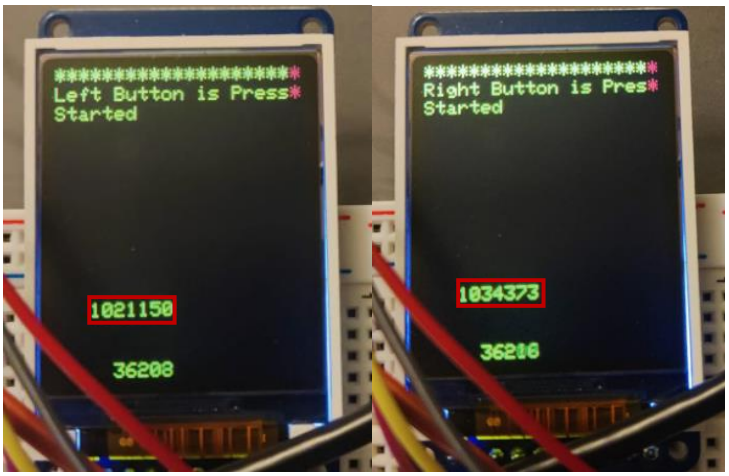

Figure 3.7 LCD display of PIDWork

4.5 Run-time statistics table, CPU usage percentage (3.g)

In this part, we created a runtime STAT task, for every 10 seconds and collect the run-time information of each task in the figure and table shown below.

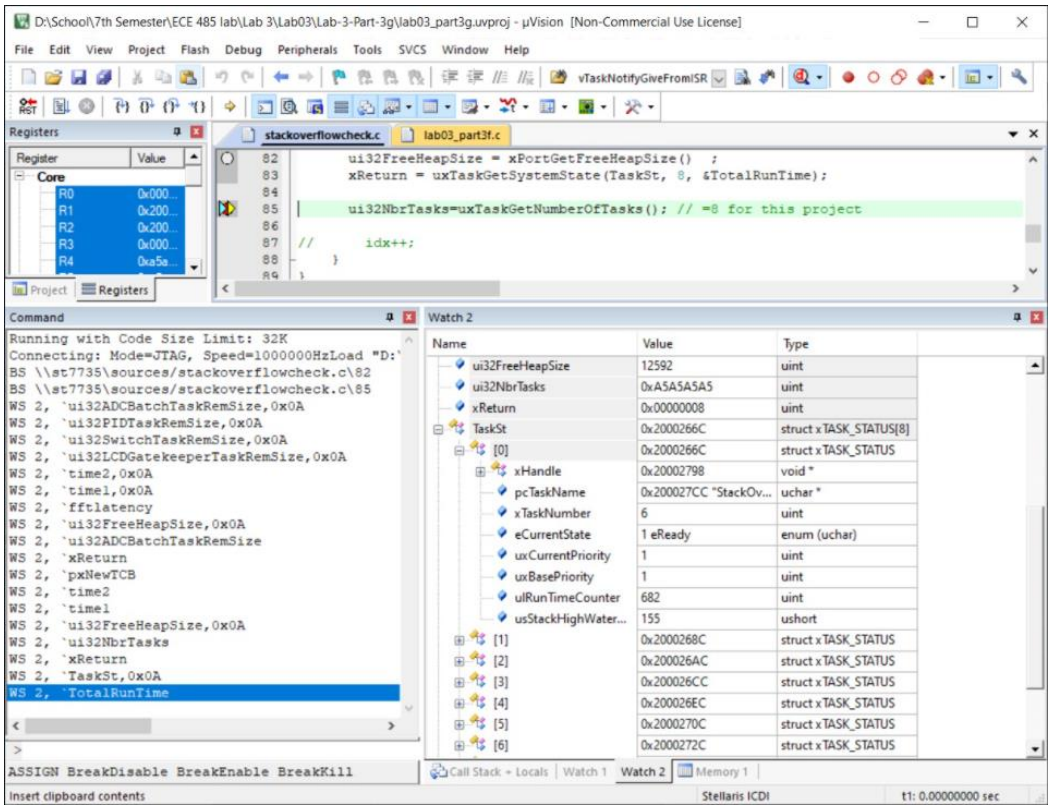

Figure 3.8 A Sample of Task Run-Time STAT

Table 1. A Sample of Task Run-Time STAT

Task Handle	Task Name	Task Number	Current State	Current Priority	RunTime Counter	StackHigh WaterMark
0x20002798	Stackoverflow	6	eReady	1	682	155
0x20002B28	IDLE	7	eReady	0	43182594	189
0x20002328	PID Work	5	eBlocked	3	73756	215
0x200031D8	Tmr Svc	8	eBlocked	7	374546	368
0x20001EB8	CMD RUN TAS	4	eBlocked	4	15242	209
0x20001988	ADC Batch	3	eSuspended	5	2795300	159
0x200010A8	LCD GateKeep	1	eBlocked	2	755029676	193
0x20001518	Switch	2	eSuspended	6	480	223

Total number of tasks: 8

Total running time since kernel started: 801554776

System tick = 10 ms

CPU usage percentage = (total run time – IDLE)/total run time

CPU usage percentage = (801554776 – 43182594)/801554776 = 94.61%

5.0 Analysis and Discussion

1) **Justify why Task 3.a DAS has no time jitter on its ADC sampling.**

   Because the ADC is triggered in hardware, this sampling process has no jitter. The DAS is software-triggered, and the software must output to the DAC at a fixed rate. The the difference between when a periodic task is supposed to be run is defined with calculation based on sampling rate in Task 3.a.

2) **There are four (or more) interrupts in this system DAS : Push button, UART, SysTick, PendSV. Justify your choice of hardware priorities in the NVIC?**

   The interrupt of Push Button Port F is high, but lower than the ADC sampling so that we will still see the sample value being displayed. Typically, the priority of system tick which is used for FreeRTOS is low, and the pendSV is the lowest, other interrupt priorities can be setup higher than these two exceptions.

3) **Explain what happens if your stack size is too small. How could you detect stack overflow? How could you prevent stack overflow from crashing the OS?**

   If the stack size is too small for a thread to run it may cause silent data corruption in neighboring threads, or segmentation faults. In this lab, we created a runtime STAT task and collect run-time information for every 10 seconds. Task run-time statistics provide information on the amount of processing time each task has received and the availability of stack size of each task. Collecting run-time statistics will increase the task context switch time during debugging and developing stage. Based on the information, we can adjust the size of task stacks, and priorities of each task.

   There are two FreeRTOS methods of catching stack overflow before it crashes the OS:

   Stack Overflow Detection - Method 1

   It is likely that the stack will reach its greatest (deepest) value after the RTOS kernel has swapped the task out of the Running state because this is when the stack will contain the task context. At this point the RTOS kernel can check that the processor stack pointer remains within the valid stack space. The stack overflow hook function is called if the stack pointer contains a value that is outside of the valid stack range.

   This method is quick but not guaranteed to catch all stack overflows. Set configCHECK_FOR_STACK_OVERFLOW to 1 to use this method.

   Stack Overflow Detection - Method 2

   When a task is first created its stack is filled with a known value. When swapping a task out of the Running state the RTOS kernel can check the last 16 bytes within the valid stack range to ensure that these known values have not been overwritten by the task or interrupt activity. The stack overflow hook function is called should any of these 16 bytes not remain at their initial value.

This method is less efficient than method one, but still fairly fast. It is very likely to catch stack overflows but is still not guaranteed to catch all overflows.

Set configCHECK_FOR_STACK_OVERFLOW to 2 to use this method.