Lab 2 RTOS Kernel (Parts 1 & 2)

Michael Coulter and Jake Klovenski

A) Objectives (1/2 page maximum)

Develop the kernel of a RTOS. This includes implementing a round robin preemptive scheduler with optional cooperative functions such as sleep and suspend. The preemptive scheduler will function through SysTick interrupts at a period of determined at by the value set at the time the OS is launched. The SysTick will call the scheduler function which decides which thread runs next and control the sleeping threads. Afterwards the actual context switch will be handled by PendSV which is triggered by the SysTick. We will also implement periodic background threads which run off a periodic timer interrupt that has a higher priority than SysTick. The RTOS will also have the capability of having a GPIO triggered background thread.

B) Hardware Design (none for this lab)

C) Software Design (documentation and code of spinlock/round-robin operating system)

OS_AddThread works by creating a doubly linked list of TCB objects. An stack-like array called “thread_IDs” is initialized with values 0 through NUMTHREADS with index “thread_IDs_index” that points to the next unused ID. This array is utilized to distribute TCB IDs and to also indicate the place of each TCB in the “tcbs” array as threads are added. The first thread to be added is simply linked to itself, and is given the ID 0 from the 0th index of the thread_IDs array, and is placed into the 0th place of the tcbs array. thread_IDs_index is incremented, pointing to the next available thread ID. Each additional thread is given the next ID from thread_IDs while also being double linked to the end of the tcbs array. If a thread is killed, the killed thread’s ID is placed back into the thread_IDs array, decrementing thread_IDs_index, indicating that that thread’s ID is now available to be used by the thread that is created next. The killed thread is then removed from the doubly linked list of TCBs by doubly linking the killed TCBs next TCB with the killed TCBs previous TCB.

This doubly linked list keeps track of all running threads’ TCBs. Because the TM4C only has a single computational core we must progress through this linked list 1 thread per timeslice. Each call to the scheduler finds the next thread that is not currently sleeping. The scheduler also decrements all sleeping threads sleep counter by 1 and if it becomes 0 it becomes eligible for running that timeslice, if it is the next “awake” thread.

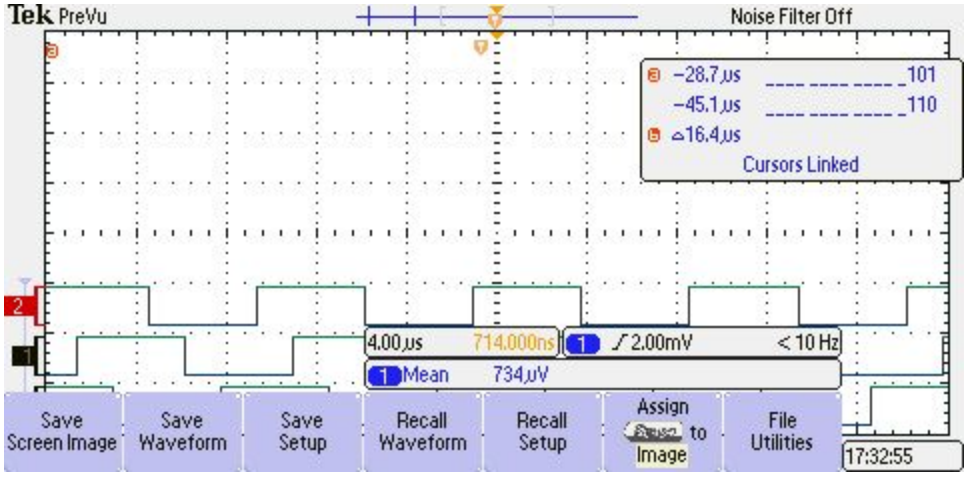
We have also implemented a spin lock semaphore that allows us to implement mutual exclusion and to some extent inter-thread communication. The semaphores are initialized to any given non-negative value the user desires. In the function OS_Wait, the OS atomically

checks the value of the semaphore and if it is greater than 0 decrements the pointer and continues on into the section. If the value is equal to 0 the thread suspends itself allowing another thread to run while it waits for the value of the semaphore to incremented. OS_Signal atomically increments the value of the semaphore passed to it and continues on with the program. OS_bWait sets the value of the semaphore to 0 if it is not already or waits for the semaphore to become 1 then sets it to 0 and continues. OS_bSignal atomically sets the value of the semaphore to 1 allowing other threads that might be waiting on the semaphore to access the critical section of code. A semaphore can also be used for interthread communication, such as in the OS_Fifo functions. When a new value is loaded into the Fifo OS_Signal is called, and when a thread is pulling a value from the OS_Fifo it calls OS_Wait on the Semaphore and decrements. In this case the semaphore keeps track of the number of items in the Fifo and forces a consumer thread to wait if the Fifo is empty. We have implemented two forms of background threads, one being a periodic thread that is controlled by a hardware periodic timer, and the other being triggered by a GPIO Interrupt caused by one of the TM4Cs onboard switches. These background threads will interrupt any foreground thread, AKA a thread being controlled by the SysTick preemptive scheduler.

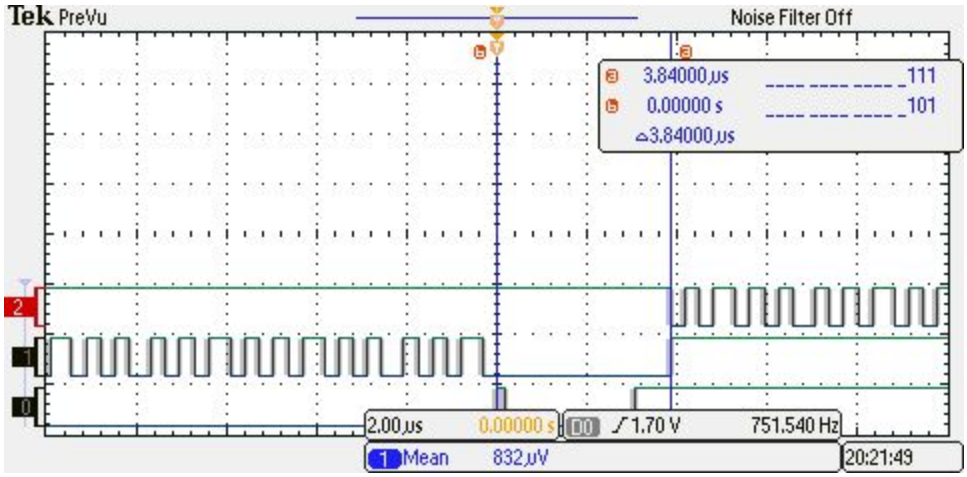
D) Measurement Data

1) plots of the logic analyzer like Figures 2.1, 2.2, 2.3, 2.4, and 2.8

2.1 Cooperative scheduler

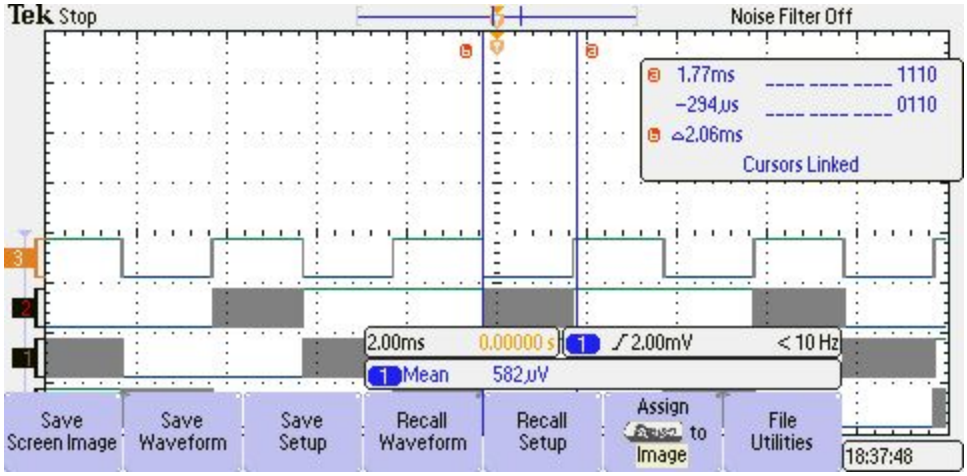


2.2 Preemptive scheduler

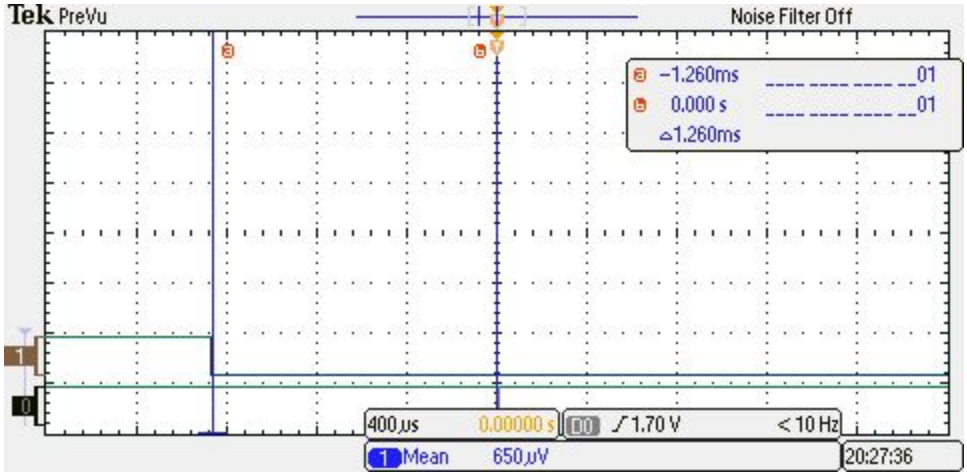


Time for contact switch is about 3.84 us

2.3 Preemptive scheduler zoomed out

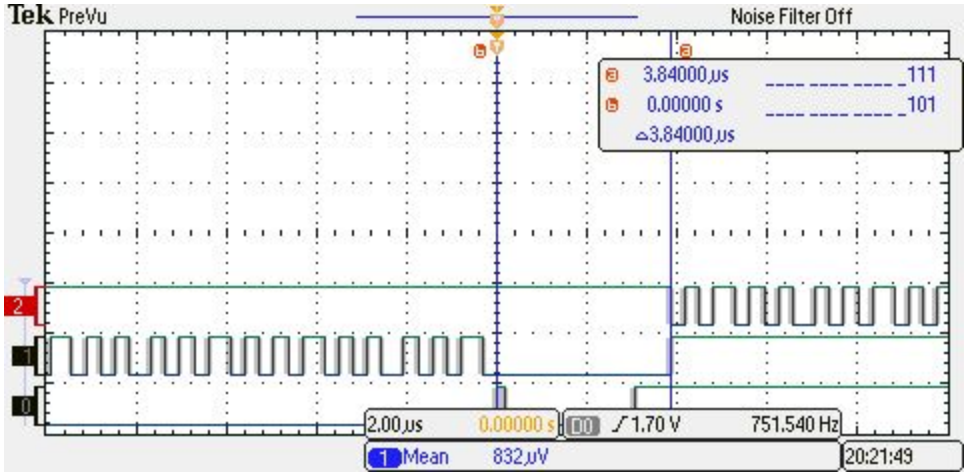


2.4 Latency of Real time thread



Cursor b shows the toggle of the pin in the button task. The time between when the switch is pressed and when the button task is run is about 1.26 ms.

2.8 Thread Switch time

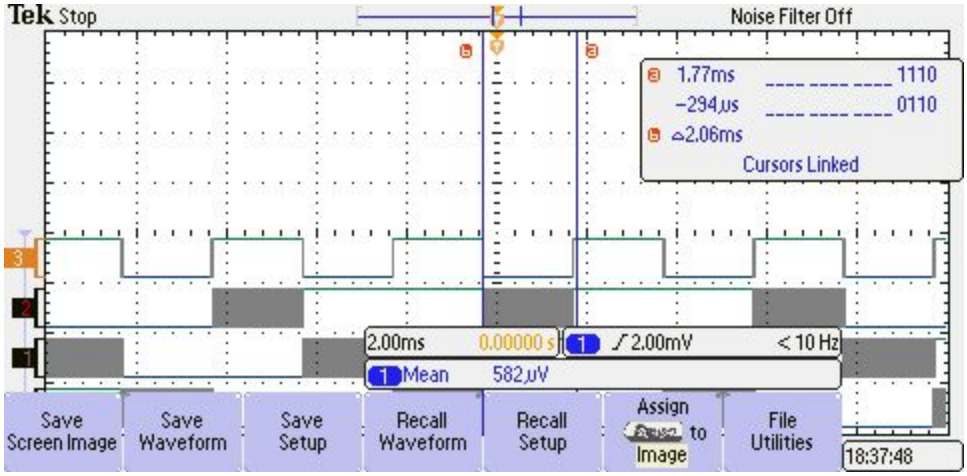


It takes about 3.84us for the contact switcher to switch threads.

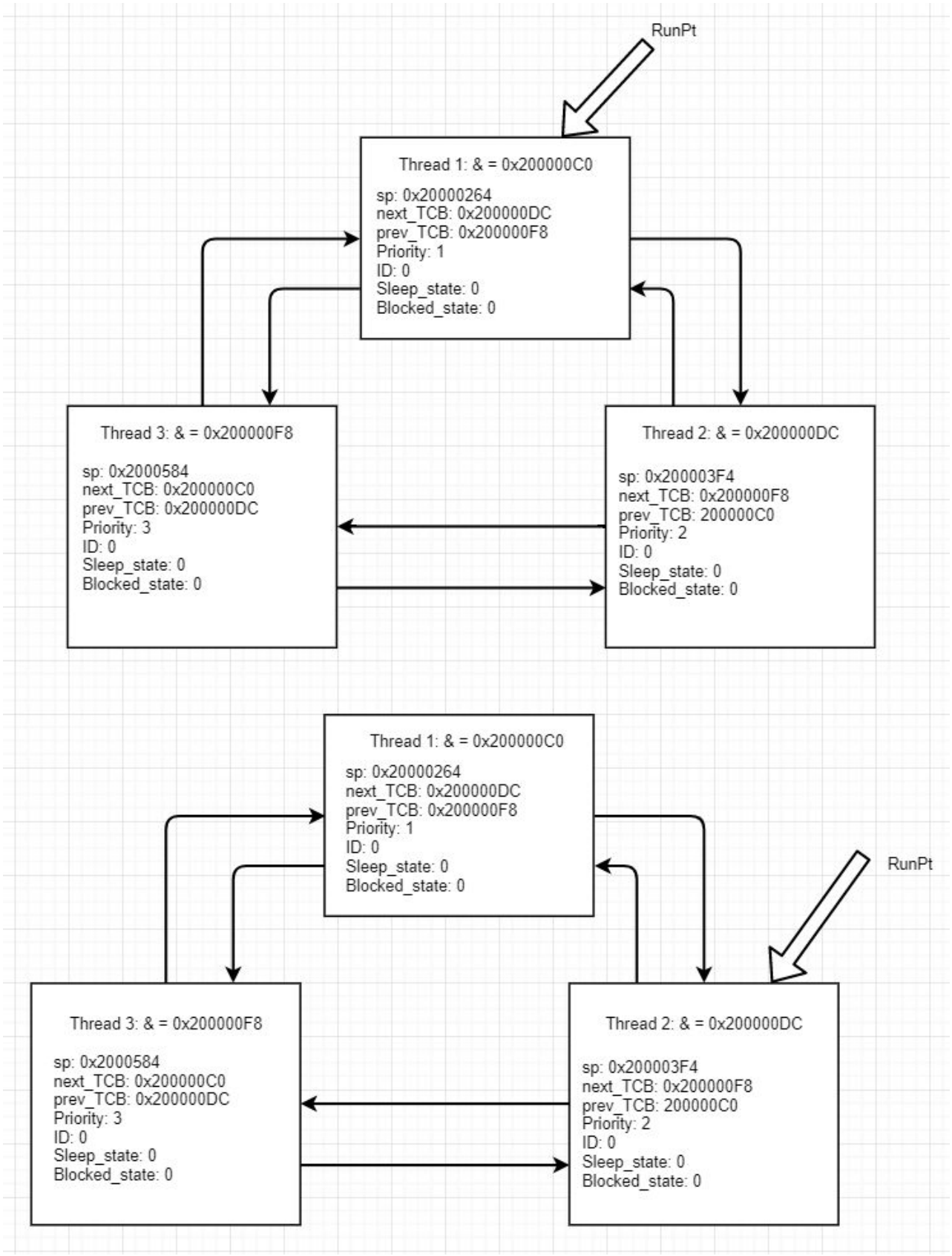
2) measurement of the thread-switch time

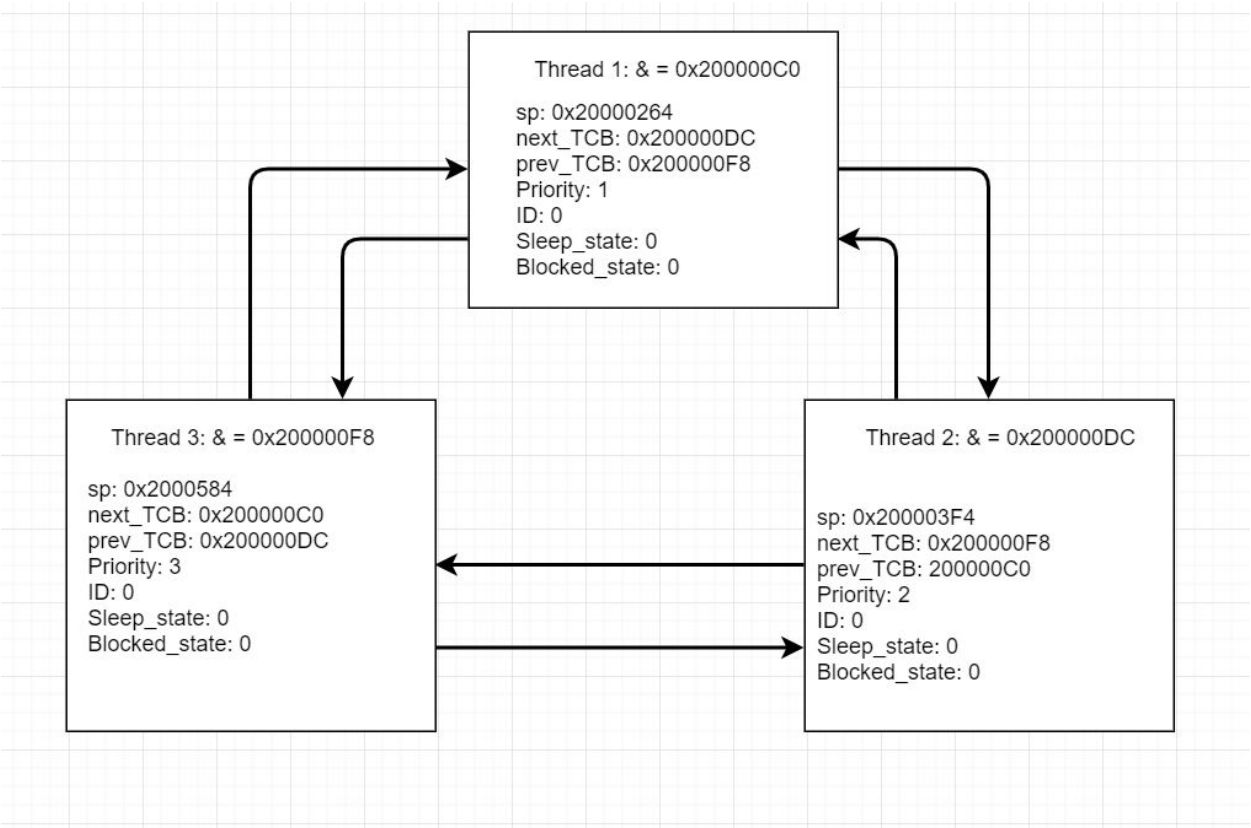
Measured as part of D.1, put value here: 3.84us

3) plot of the logic analyzer running the spinlock/round-robin system (profile data)



4) the four sketches (from first preparation parts 3 and 5), with measured data collected during testing
    a) We did not save the graphs before adding the values.


RunPt
Thread 1: & = 0x200000C0
sp: 0x20000264
next_TCB: 0x200000DC
prev_TCB: 0x200000F8
Priority: 1
ID: 0
Sleep_state: 0
Blocked_state: 0
Thread 3: & = 0x200000F8
sp: 0x2000584
next_TCB: 0x200000C0
prev_TCB: 0x200000DC
Priority: 3
ID: 0
Sleep_state: 0
Blocked_state: 0
Thread 2: & = 0x200000DC
sp: 0x200003F4
next_TCB: 0x200000F8
prev_TCB: 200000C0
Priority: 2
ID: 0
Sleep_state: 0
Blocked_state: 0
Thread 1: & = 0x200000C0
sp: 0x20000264
next_TCB: 0x200000DC
prev_TCB: 0x200000F8
Priority: 1
ID: 0
Sleep_state: 0
Blocked_state: 0
RunPt
Thread 3: & = 0x200000F8
sp: 0x2000584
next_TCB: 0x200000C0
prev_TCB: 0x200000DC
Priority: 3
ID: 0
Sleep_state: 0
Blocked_state: 0
Thread 2: & = 0x200000DC
sp: 0x200003F4
next_TCB: 0x200000F8
prev_TCB: 200000C0
Priority: 2
ID: 0
Sleep_state: 0
Blocked_state: 0



5) a table like Table 2.1 each showing performance measurements versus sizes of OS_Fifo and timeslices

FIFOSize	TIMESLICE(ms)	DataLost	Jitter(us)	PIDWork
4	2	0	NA	617
8	2	0	NA	700
32	2	0	NA	5704
32	1	0	NA	226
32	10	0	NA	764

E) Analysis and Discussion (2 page maximum). In particular, answer these questions

1) Why did the time jitter in my solution jump from 4 to 6s when interpreter I/O occurred?

The interpreter utilizes UART interrupts to send and receive I/O which preempt the periodic thread.

2) Justify why Task 3 has no time jitter on its ADC sampling.

Note to Malek: we did not have jitter working properly for this lab.

3) There are four (or more) interrupts in this system DAS, ADC, Select, and SysTick (thread switch). Justify your choice of hardware priorities in the NVIC?

The SysTick needs to lower than all other interrupts except PendSV because it triggers PendSV. The Periodic Timer requires a fairly high priority so that is able to interrupt other interrupts and attain a more accurate jitter reading. The GPIO interrupt needs to be somewhere between SysTick and the periodic timer. The ADC interrupt also needs to have a priority between the periodic thread and the SysTick, it should also be lower priority than the GPIO priority.

4) 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 is too small we could have a the thread's stacks colliding, resulting in data being corrupted and local variables in those threads being changed. If the LRs in the stacks are corrupted this could lead to a total system crash. We can prevent this by storing a proverbial stop sign in the form of a "magic number" at the bounds of the thread's stack that would prevent that thread from writing outside of it stack.

5) Both `Consumer` and `Display` have an `OS_Kill()` at the end. Do these `OS_Kill`s always execute, sometime execute, or never execute? Explain.

The OS kills will always execute in Consumer and Display given that the system runs long enough that ADC_Collect can generate a RUNLENGTH amount of data. They run in the while loop with the condition NumSamples < RUNLENGTH and NumSamples is incremented when a new value is read from the Sequencer interrupt.

6) The interaction between the producer and consumer is deterministic. What does deterministic mean? Assume for this question that if the `OS_Fifo` has 5 elements

data is lost, but if it has 6 elements no data is lost. What does this tell you about the timing of the consumer plus display?

Deterministic means that the program will always run a certain way. If we assume that OS_Fifo has 5 elements and loses data, but if it has 6 elements if will not lose data means that the producer creating data at a rate faster than it takes for the consumer to process the data and display it. With the addition of the sixth space the producer has more storage to put data while the consumer is displaying the data, leading to no loss.

7) Without going back and actually measuring it, do you think the Consumer ever waits when it calls OS_MailBox_Send? Explain.

I do not believe that Consumer ever has to wait on OS_MailBox_Send. It is only called every 160ms (not measured, it's in the comments for the function call), while the threads have a time slice of 2ms. This means that for approximately every 1 call to OS_MailBox_Send 80 context switches occur, meaning Display, the thread the calls OS_MailBox_Recieve, runs approximately 20-30 times.