A) Objectives (1/2 page maximum)

In Lab2, we will implement and test the following:
OS_AddThread: creates a new thread to run, should work while system is running
OS_Kill: remove a thread from running
OS_Suspend: trigger a cooperative context switch
OS_Sleep: a cooperative context switch with a certain wait time
OS_AddPeriodicThread: initialize and run a timer-triggered thread
OS_Wait, OS_bWait, OS_Signal, OS_bSignal: spinlock semaphores
OS_Time, OS_MsTime, OS_ClearMsTime: time tracking functions
OS_Fifo, OS_MailBox: for sending data between threads
ADC driver
Interpreter
Debugging instruments

In addition, we will measure the overhead of context switches (scheduler) and record the performance of our system.

All of our code is dependent on spinlocks, which in turn are dependent on spinning semaphores that we implemented. The lab was also an exercise in synchronization and mutual exclusion using semaphores and locks.

B) Hardware Design (none for this lab)

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

We implemented our spinlock in assembly, we wanted to use LDREX and STREX instructions as the basis of our lock. This avoids the need to disable and enable interrupts for the locks to work while waiting in a loop.

Our scheduler is round-robin, which is implemented by using a circularly linked list. Every time SysTick interrupts, we simply go to whatever RunPt→next points to.

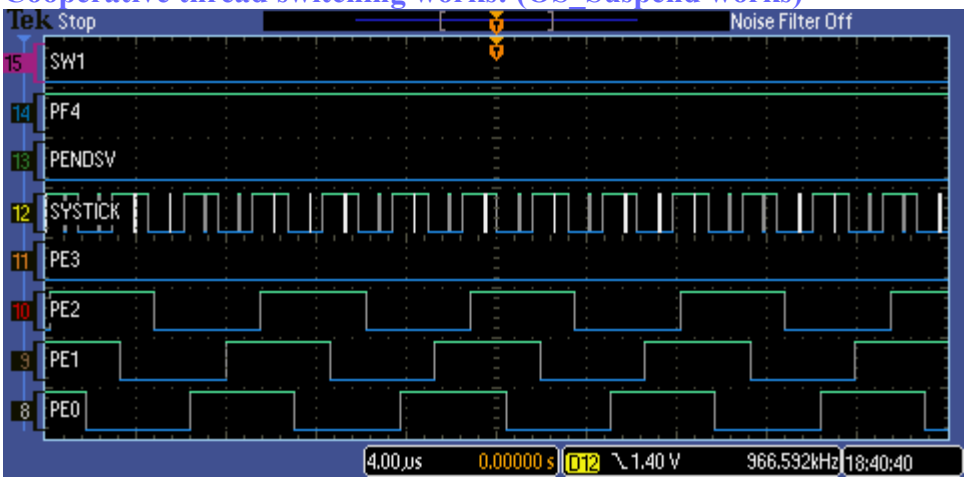
One of our most frustrating issue during development was getting OS_Sleep to work. The main issue was that we were trying to remove the TCB from a running list to a sleeping list; this caused a lot of havoc when we tried to reinsert our sleeping thread back into the running list as the thread may not wake up or cause the running list to corrupt resulting in a system crash.

In the end, we ended up rewriting our scheduler to check if the thread is sleeping or not and skipping to the next TCB instead. This simplifies the logic to simply checking the state of a thread instead of modifying the list. In addition, we use a 1ms timer to decrement the sleep counter in each thread.

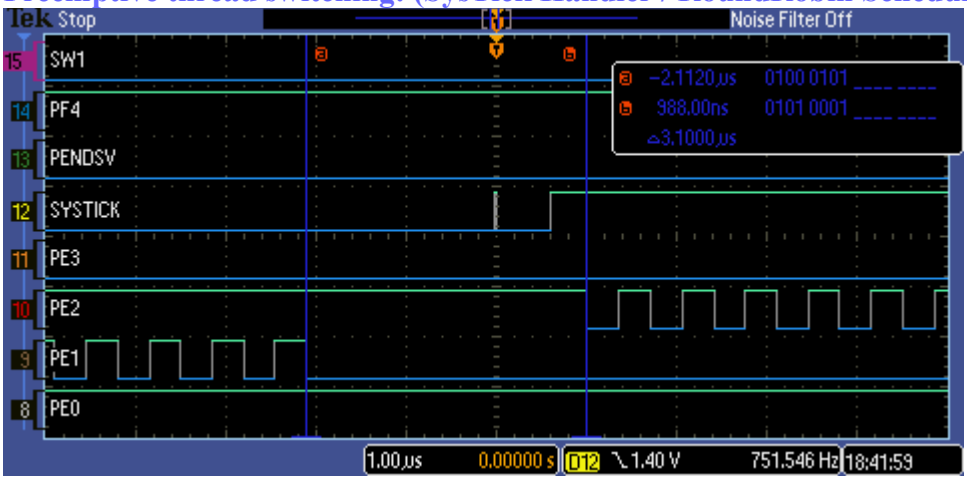
D) Measurement Data

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

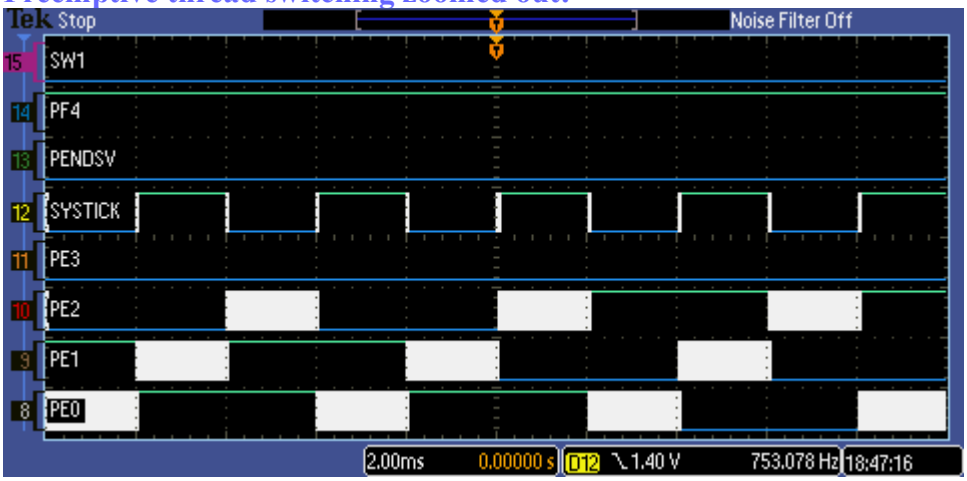
Cooperative thread switching works: (OS_Suspend works)



Preemptive thread switching: (SysTick Handler / RoundRobin Scheduler)

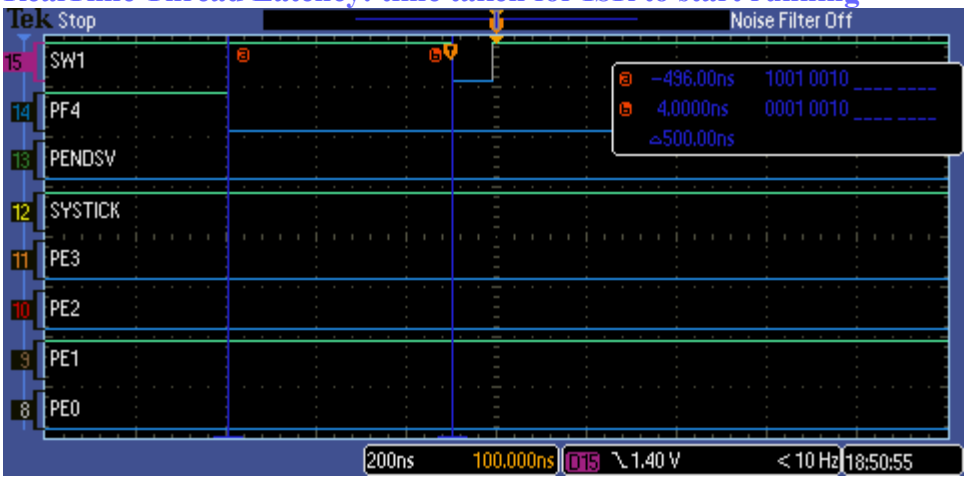


Preemptive thread switching zoomed out:



RealTime Thread Latency: this shows how long it takes to service ButtonWork

RealTime Thread Latency: time taken for ISR to start running



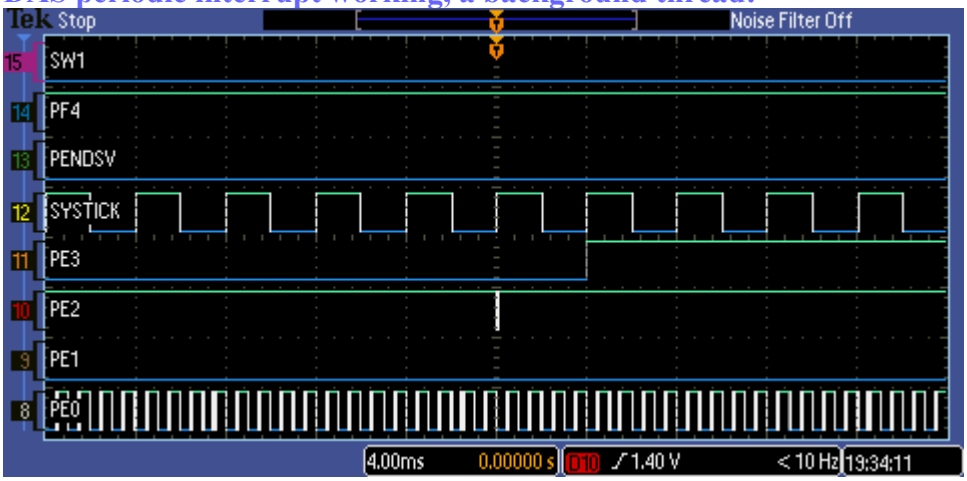
ContextSwitch overhead: 1.2040us – 204.00ns = 1.0000us overhead

2) measurement of the thread-switch time

It takes 1.0000us to thread switch.

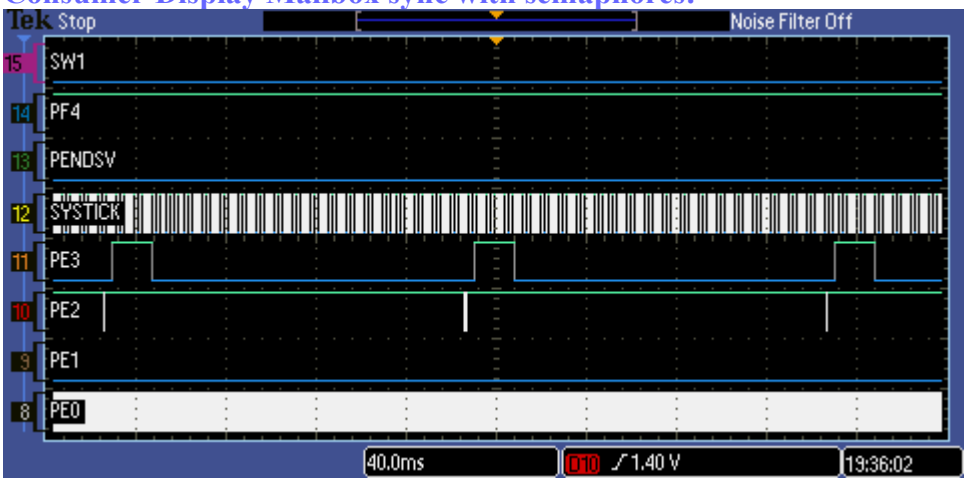
3) plot of the logic analyzer running the spinlock/round-robin system (profile data)

DAS periodic interrupt working, a background thread:



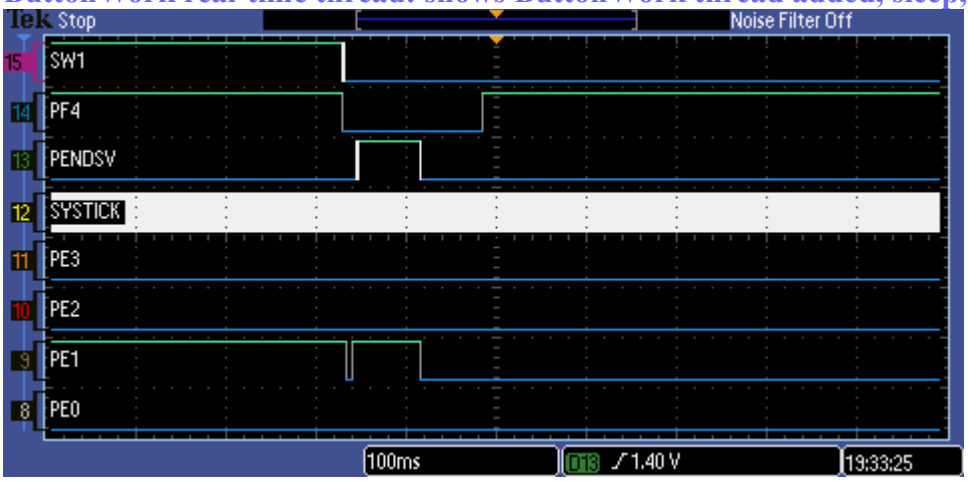
The DAS execution is independent of SysTick.

Consumer-Display Mailbox sync with semaphores:



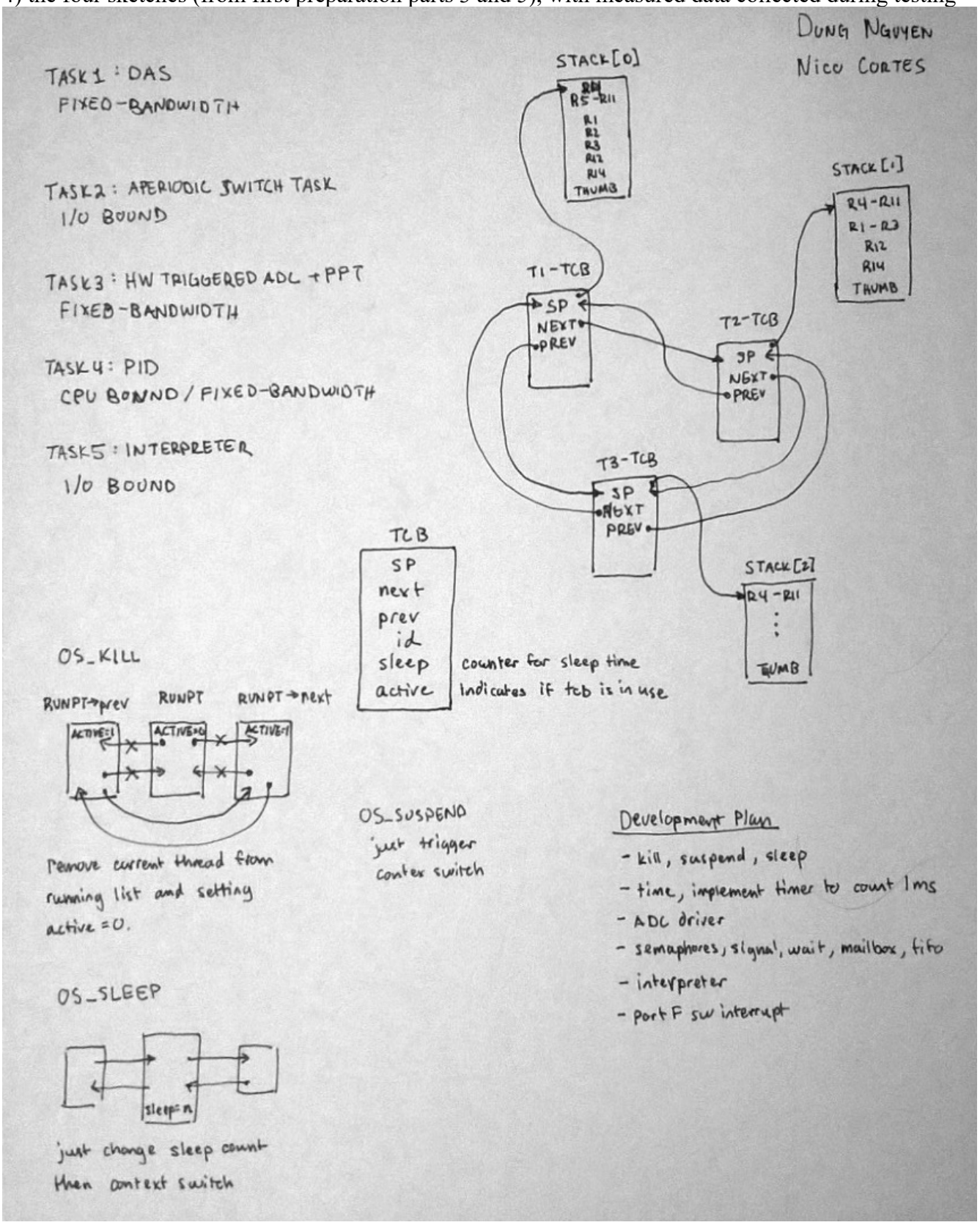
We can see the display receiving the message after the consumer sends the data.

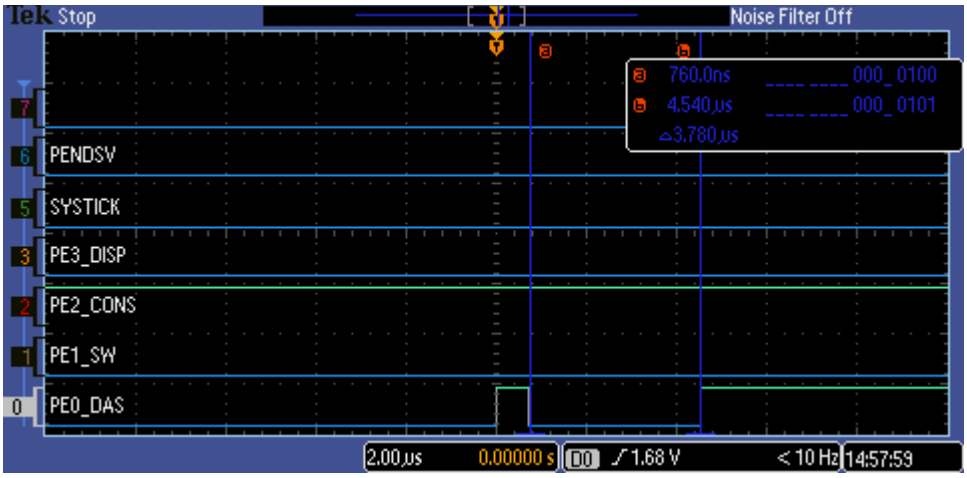
ButtonWork real-time thread: shows ButtonWork thread added, sleep, wakeup, and killed



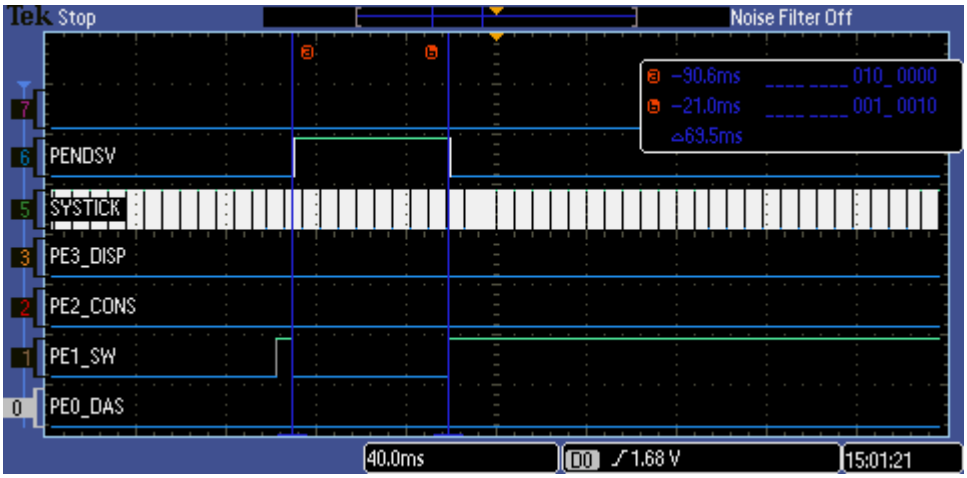
PendSV toggles twice indicating OS_Sleep call and OS_Kill call. OS_Kill also indicates the thread has woken up.
ButtonWork is quickly added after the PortF ISR runs.

4) the four sketches (from first preparation parts 3 and 5), with measured data collected during testing

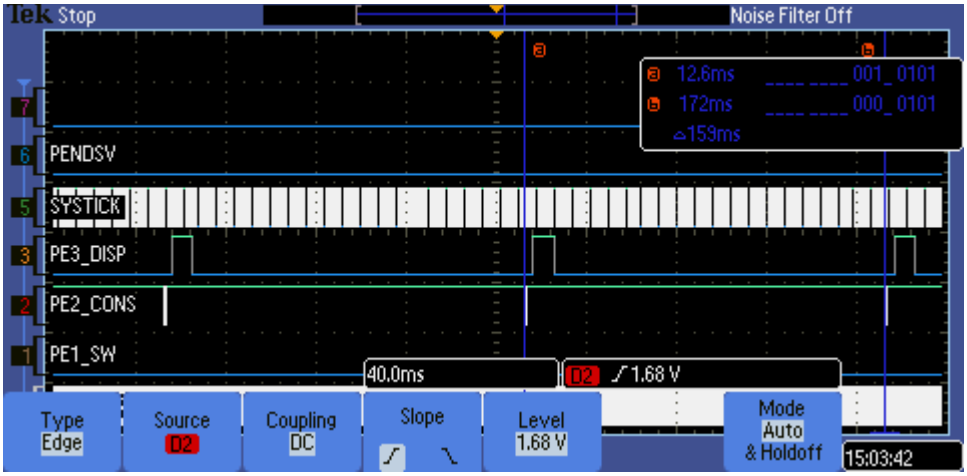




The DAS takes 3.780us to complete ADC sampling and do Filterwork.



The switch takes 69.5ms to complete, 50ms is spent sleeping.



The consumer takes 159ms to collect and calculate 64 samples.

Our PID performance is 3396 / 20s which is 169.8 PID work/sec or 5.889ms for each PID work.

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

FIFOSize	TIMESLICE	DataLost	Jitter (us)	PIDWork
4	2	0	0	3395
4	1	0	0	3393
4	0.5	0	0	3382
4	3	0	0	3396
128	4	0	0	3396
32	4	0	0	3396
8	4	0	0	3396
2	2	1000	0	3396

When we try large timeslices, the system would stop working until we increase the fifosize.

6) a table showing performance measurements with and without debugging instruments
We get the same performance as above without debugging instruments, (our debug are just pin toggles).

E) Analysis and Discussion

1) Why did the time jitter in my solution jump from 4 to 6 µs when interpreter I/O occurred?
The interpreter (UART ISR) interrupted the DAS during execution and takes 2 µs to complete.

2) Justify why Task 3 has no time jitter on its ADC sampling.
The producer is always called by the ADC ISR, in which the ADC sampling is triggered by a timer. Since the Timer and ADC hardware is not affected by interrupts, Timer will always trigger the ADC at a regular interval therefore no jitter.

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?
From low to high: Foreground → SysTick (7) → PendSV(6) → PortF, 1msTimer (5) → ADC (2) → DAS (1)

SysTick and PendSV are the lowest since we never want a context switch during an ISR. If SysTick interrupted an ISR, 1) The ISR never finishes 2) we will context switch into a new thread with an ISR privileges 3) possibly be unable to exit ISR, since LR must have a specific value to BX LR from ISR and so on…

DAS gets the highest priority as we want to reduce time jitter.

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?
Stack overflow will occur if the stack size is too small. We can detect stack overflow by comparing our Stack Pointer to the stack size. One way of preventing stack overflow is to preemptively kill the thread if it reaches a certain bound right before the SP writes into unallocated space. (Make the stack size = needed size + some buffer area)

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

Data goes from Producer to Consumer to Display. If the producer does not work correctly, then OS_Kill() will never be reached. OS_Kill() is only reached when Consumer and Display gets enough data.

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 there is a strict ordering in which threads input to or obtain from the fifo. The consume is I/O bound since it has to wait to print to the display.

7) Without going back and actually measuring it, do you think the Consumer ever waits when it calls OS_MailBox_Send? Explain.
No, the Consumer does not wait because of RoundRobin scheduling; the Display will always call OS_MailBox_Recv() to get the data sent by the Consumer before the Consumer calls OS_MailBox_Send().