Lab 2 RTOS Kernel (Parts 1 & 2)

Goals
- Develop OS facilities for real-time applications,
- Coordinate multiple foreground and background threads,
- Design a round robin multi-thread scheduler,
- Implement spinlock semaphores and use them for thread synchronization,
- Implement inter-thread communication.

Starter files
- **`EdgeInterrupt_4C123`** project (input switch interrupt)
- **`PeriodicTimer0AInts_4C123`** project (periodic interrupts)
- **`RTOS_4C123`** project (basic OS)
- **`OS.h`** and **`Lab2.c`** (from class web site)

Downloads

1) http://www.ece.utexas.edu/~valvano/EE345M/STM32F10x_DSP_Lib_V2.0.0_setup.exe
   Copy **cr4_fft_64_stm32.s** file from **xx\Libraries\STM32F10x_DSP_Lib\src\asm\arm** to your project
   Copy **PID_stm32.s** files from **xx\Libraries\STM32F10x_DSP_Lib\src\asm\arm** to your project
2) Copy these files
   http://www.ece.utexas.edu/~valvano/EE345M/UM0585.pdf
   http://www.ece.utexas.edu/~valvano/EE345M/PID_stm32.s
   http://www.ece.utexas.edu/~valvano/EE345M/cr4_fft_64_stm32.s

Other References

1) https://www.micrium.com/download/µcos-ii-and-the-arm-cortex-m3-processors/
   Download, install, look at the *AppNotes* and *Software* files (choose *RealView* over *IAR*)
2) http://www.freertos.org/ Look at the Cortex M3 port

Background Part 1

In the first part of Lab 2 you will design and test a cooperative, and then you will design and test a preemptive thread scheduler. The requirements will be to run *Testmain1* and *Testmain2*, respectively.

Testmain1 needs a cooperative thread scheduler with no interrupts. Each thread will suspend itself each time through the loop by calling **`OS_Suspend()`**. When *Testmain1* executes, the PE0, PE1, PE2 outputs will look like Figure 2.1, and the three count variables will be equal (±1). In Figure 2.1, each toggle means a thread has started a pass through its main loop. For this system, these threads are running every 1.8 µs. Figure 2.1 shows no interrupts from the switch, timer, or SysTick.

```
unsigned long Count1;   // number of times thread1 loops
unsigned long Count2;   // number of times thread2 loops
unsigned long Count3;   // number of times thread3 loops
void Thread1(void){
  Count1 = 0;
  for(;;){
    PB2 ^= 0x04;       // heartbeat
    Count1++;
    OS_Suspend();      // cooperative multitasking
  }
}
void Thread2(void){
  Count2 = 0;
  for(;;){
    PB3 ^= 0x08;       // heartbeat
    Count2++;
    OS_Suspend();      // cooperative multitasking
  }
}
```

```
void Thread3(void){
  Count3 = 0;
  for(;;){
    PB4 ^= 0x10;       // heartbeat
    Count3++;
    OS_Suspend();      // cooperative multitasking
  }
}

int main(void){  // Testmain1
  OS_Init();          // initialize, disable interrupts
  PortB_Init();       // profile user threads
  NumCreated = 0 ;
  NumCreated += OS_AddThread(&Thread1,128,1);
  NumCreated += OS_AddThread(&Thread2,128,2);
  NumCreated += OS_AddThread(&Thread3,128,3);
  // Count1 Count2 Count3 should be equal or off by one at all times
  OS_Launch(TIME_2MS); // doesn't return, interrupts enabled in here
  return 0;            // this never executes
}
```

Program 2.1. Test system for cooperative thread switching

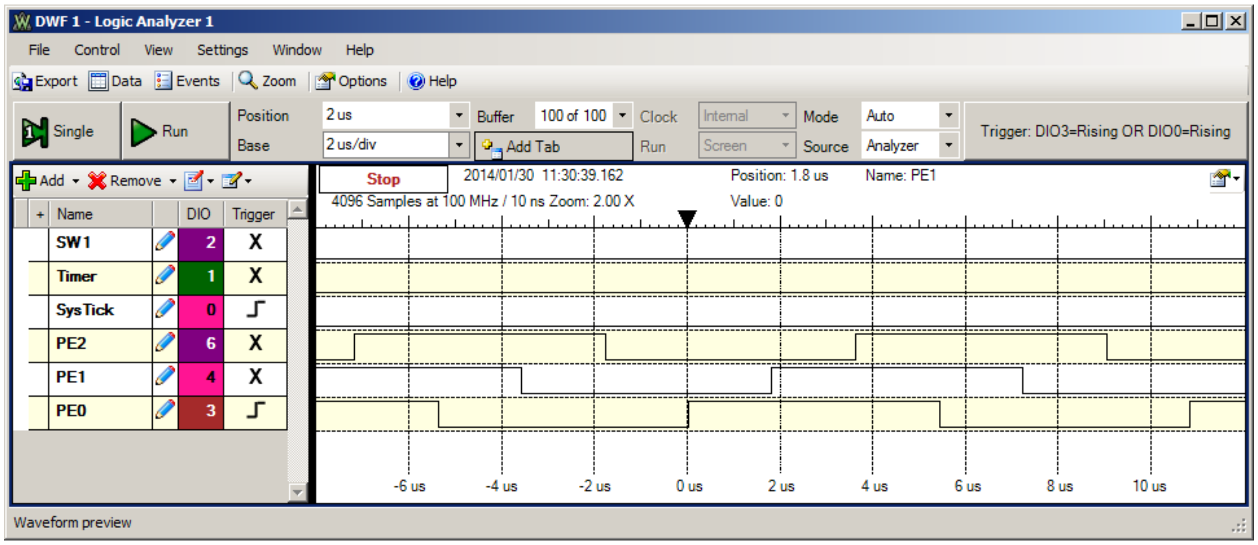


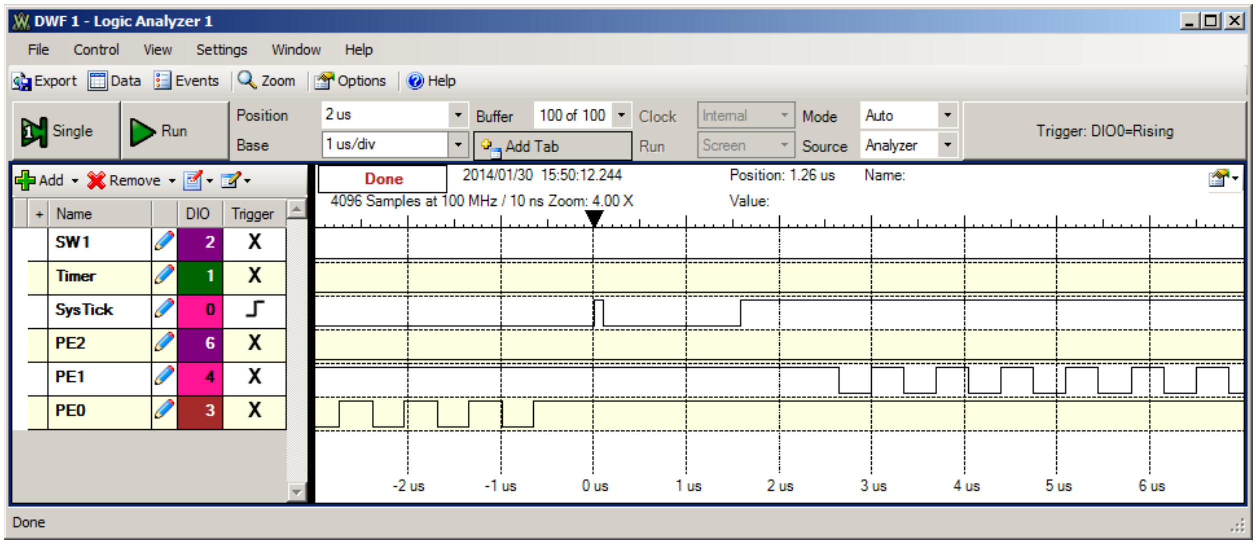
Figure 2.1. Logic analyzer profiling for cooperative thread switching.

Testmain2 needs a preemptive thread scheduler with SysTick interrupts. The SysTick ISR will suspend the running thread and run the next active thread in the list in a round robin fashion. When *Testmain2* executes, the SysTick, PE0, PE1, PE2 outputs will look like Figures 2.2 and 2.3, and the three count variables will be approximately equal. The values of the counters will be much higher than they were in *Testmain1*. Think of an explanation of why this is true. This OS toggles PF1 three times in each SysTick ISR as it is running the preemptive scheduler. In Figure 2.2, each toggle means a thread has started a pass through its main loop. For this system, the SysTick ISR runs in about 1.6 µs. Figure 2.3 shows the preemptive thread switch occurs every 2 ms.

```
void Thread1b(void){
  Count1 = 0;
  for(;;){
    PB2 ^= 0x04;       // heartbeat
    Count1++;
  }
}
```

```
void Thread2b(void){
  Count2 = 0;
  for(;;){
    PB3 ^= 0x08;       // heartbeat
    Count2++;
  }
}
void Thread3b(void){
  Count3 = 0;
  for(;;){
    PB4 ^= 0x10;       // heartbeat
    Count3++;
  }
}
int Testmain2(void){  // Testmain2
  OS_Init();           // initialize, disable interrupts
  PortB_Init();       // profile user threads
  NumCreated = 0 ;
  NumCreated += OS_AddThread(&Thread1b,128,1);
  NumCreated += OS_AddThread(&Thread2b,128,2);
  NumCreated += OS_AddThread(&Thread3b,128,3);
  // Count1 Count2 Count3 should be equal on average
  // counts are larger than testmain1
  OS_Launch(TIME_2MS); // doesn't return, interrupts enabled in here
  return 0;            // this never executes
}
```

Program 2.2. Test system for preemptive thread switching

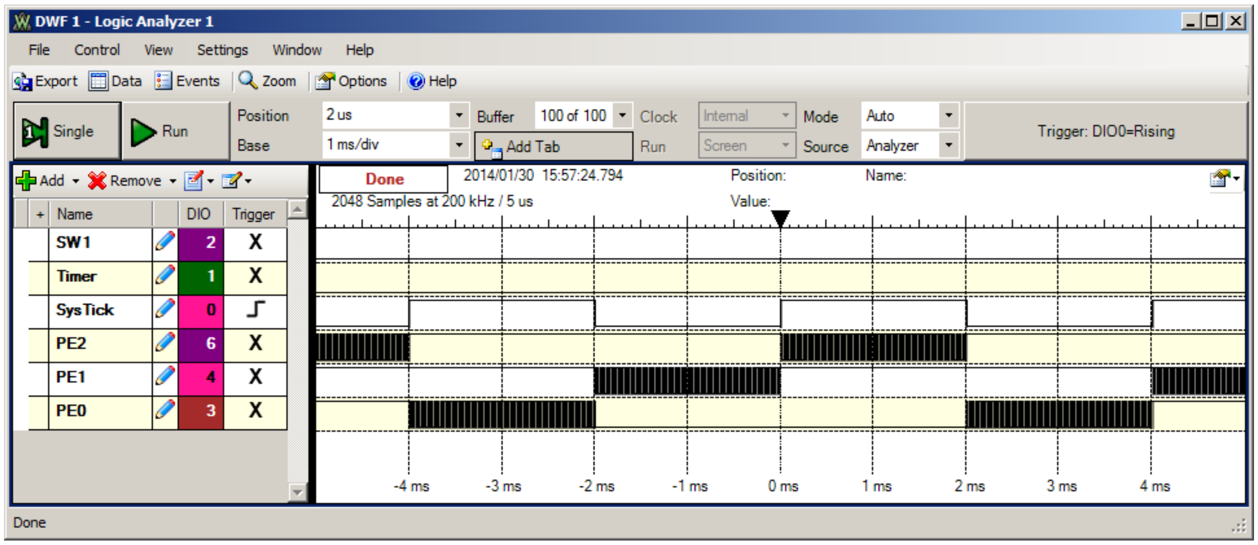

DWF 1 - Logic Analyzer 1
File Control View Settings Window Help
Export Data Events Zoom Options Help
Single Run
Add Tab
Add Remove
Done
-2 us -1 us 0 us 1 us 2 us 3 us 4 us 5 us 6 us
Done

Position 2 us
Base 1 us/div
Buffer 100 of 100
Clock Internal
Run Screen
Mode Auto
Source Analyzer
Trigger: DIO0=Rising

2014/01/30 15:50:12.244
4096 Samples at 100 MHz / 10 ns Zoom: 4.00 X
Position: 1.26 us
Name:
Value:

Name	DIO	Trigger
SW1	2	X
Timer	1	X
SysTick	0	⎍
PE2	6	X
PE1	4	X
PE0	3	X

Figure 2.2. Logic analyzer profiling for preemptive thread switching (zoomed in).



Position: 2 us
Base: 1 ms/div
Buffer: 100 of 100
Clock: Internal
Run: Screen
Mode: Auto
Source: Analyzer
Trigger: DIO0=Rising

Done 2014/01/30 15:57:24.794
2048 Samples at 200 kHz / 5 us
Position:
Value:
Name:

Name	DIO	Trigger
SW1	2	X
Timer	1	X
SysTick	0	⎍
PE2	6	X
PE1	4	X
PE0	3	X

Done

Figure 2.3. Logic analyzer profiling for preemptive thread switching (zoomed out).

Background Part 2

Your job is to 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 Labs 2 and 3. 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 2.4 shows the data flow for Lab 2.

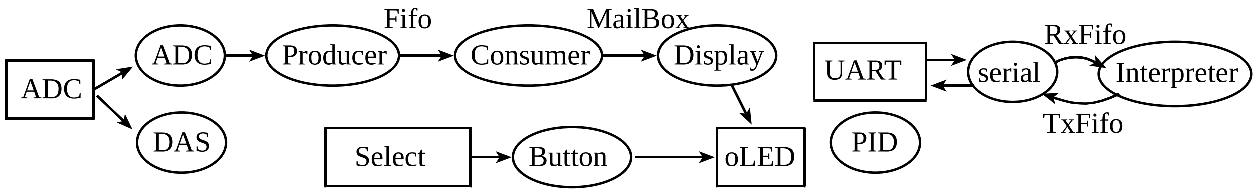


Figure 2.4. Data flow graph for the user programs.

Task 1: Software-triggered data acquisition and filtering

For the data acquisition system **DAS**, the software must start the analog-to-digital converter (ADC) and read the result at precise time intervals. Let **Δt** be the time interval, which is one divided by the sampling rate.

$$\Delta t = 1/f_s$$

In Labs 2 and 3, **Δt** is 500 µs and f_s is 2 kHz.

A second example of a software-triggered periodic task would be a digital control system. The software must read the sensors, perform the digital control equations, then output to the actuators at a fixed rate. A third example of a software-triggered periodic task would be signal generation. The software must output to the digital-to-analog converter (DAC) at a fixed rate.

We can define **time-jitter**, **δt**, as the difference between when a periodic task is supposed to be run, and when it is actually run. Let t_n be the time the software task is actually run, and let **nΔt** be the time it was supposed to be run, then the time-jitter at sample **n** is

$$\delta t_n = t_n - n\Delta t$$

For a **real time** system with periodic tasks, we must be able to place an upper bound, **k**, on the time-jitter.

$$-k \le \delta t_n \le +k \quad \text{for all } n$$

Sometimes it is more important to control the time difference between periodic events rather than the absolute time itself. Let $\mathbf{\Delta t_n}$ be the actual time difference between two executions of a software task (e.g., starting the ADC). The desired time difference is $1/\mathbf{f_s}$. For this situation, we define the time-jitter at sample **n** to be

$$\delta t_n = |\Delta t_n - 1/f_s|$$

Assuming the OS measures time with a resolution of 12.5 ns, the following line calculates $\mathbf{\delta t_n}$ with 0.1 µs units. **`PERIOD`** is a constant, in units of bus cycles, that represents the desired $1/\mathbf{f_s}$. **`diff`** is a measurement, also with units of bus cycles, that represents the actual $\mathbf{\Delta t_n}$. **`jitter`** is the calculation of $\mathbf{\delta t_n}$ with units of 0.1 µs.

```
unsigned long diff = OS_TimeDifference(LastTime,thisTime);
if(diff>PERIOD){
  jitter = (diff-PERIOD+4)/8;  // in 0.1 usec
}else{
  jitter = (PERIOD-diff+4)/8;  // in 0.1 usec
}
```

where **`OS_TimeDifference`** calculates the actual difference. The constant **`PERIOD`** is 40000 ($\mathbf{f_s}$ = 2kHz, bus = 80MHz) is the expected difference. The addition of 4 implements rounding during the divide by 8. To be classified as real-time, we must be able to place an upper bound, **k**, on the time-jitter.

$\delta t_n \le +k$ for all **n** or **`jitter`** ≤ **`MaxJitter`**

Assume that the DAS task was activated using a **`OS_AddPeriodicThread(&DAS,PERIOD,1)`**call. A time jitter experiment was performed on the professor's solution for Lab 2. The time resolution of this measurement was 0.1 µs. In a 20-sec experiment the DAS task ran 40000 times. Most of the time, the DAS task ran every 40000 cycles exactly. This means most of the time the DAS task ran at exactly equal time intervals. Following two histograms were collected as the ISR was run 40000 times without any interruptions from the switch or serial I/O. The histogram on the left was collected with an OS that disabled and enabled interrupts during **`OS_Wait`** and **`OS_Signal`**. The histogram on the right was collected with an OS that solved the critical sections in **`OS_Wait`** and **`OS_Signal`** using the **`LDREX`** and **`STREX`** instructions. Under these conditions, we can say the time jitter is less than 0.4 µs. With interpreter I/O, a time jitter of 0.6 µs. Under most situations, this error is acceptable, and we are confident to specify this system as real time.

Results with Wait/Signal disabling interrupts

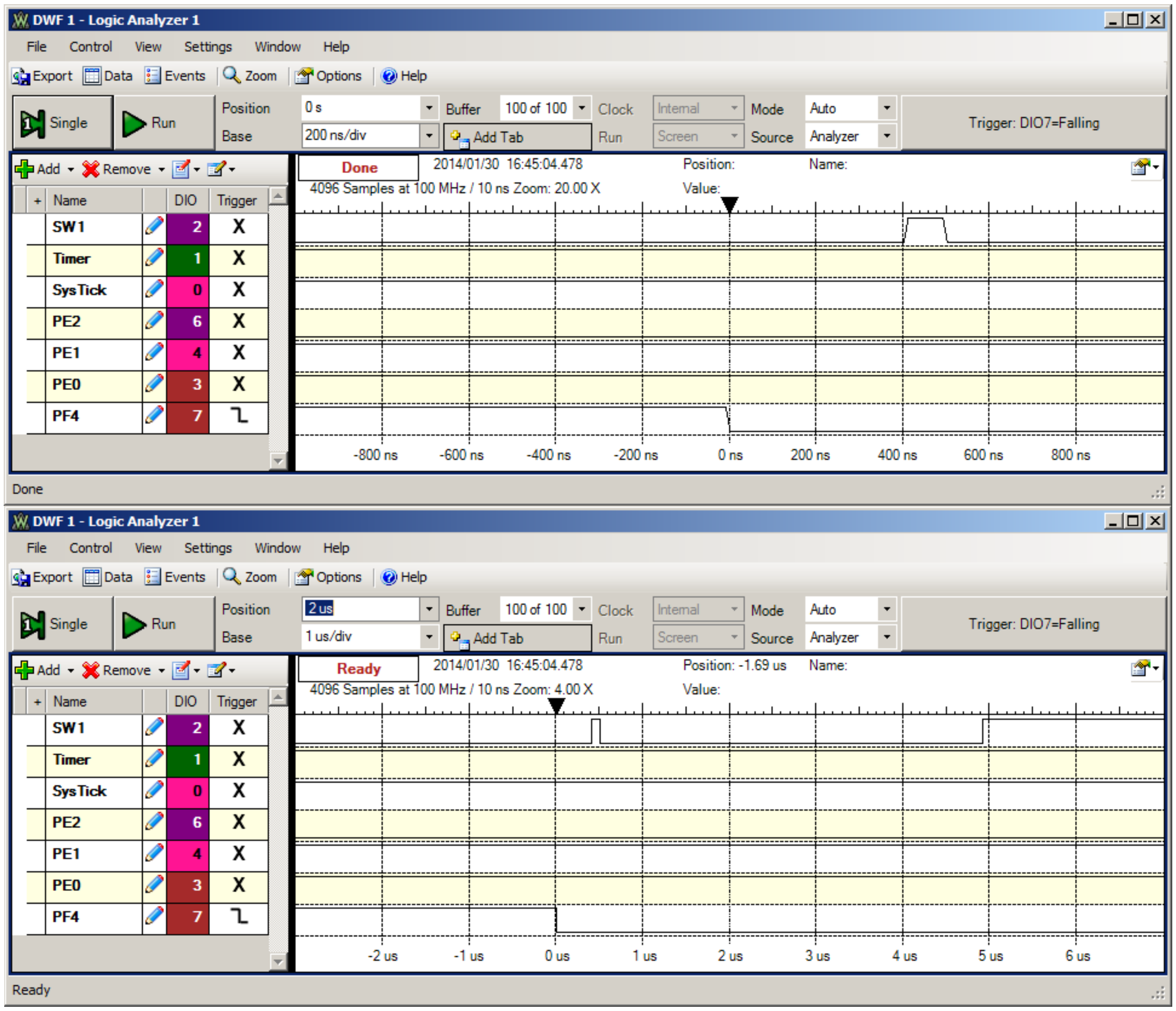
Time(us)	Frequency
0.0	22213
0.1	2770
0.2	12900
0.3	2118
0.4	1

Results with Wait/Signal using **`LDREX STREX`**

Time(us)	Frequency
0.0	38686
0.1	1312
0.2	2
0.3	0
0.4	2

Task 2: Aperiodic task triggered by the select switch

An **aperiodic** thread is one that executes frequently, but the rate is not deterministic. In Lab 2, the background thread **`ButtonPush`** should be run whenever the user touches the select button. This thread will create a foreground thread that outputs to the LCD, sleeps for 50ms, outputs again the LCD, then kills itself. If the user pushes the button faster than once every two seconds, then multiple foreground threads will be running at the same time. In this case, the time of the request is defined as the touching of the switch (fall of PF4), and the time of the service is defined as the time when the PF4 ISR is run. The time difference between request and service is the **interface latency**. If this latency is short and bounded, then this thread is classified as **real time**. Figure 2.5 shows a measurement of latency. The falling edge of PF4 occurs when the switch is touched, and the rising edge of PF3 (SW1 in figure) occurs in its ISR. For this instance, the latency is 400ns, and the time to run the ISR is less than 5 µs. This OS toggles PF3 three times in the ISR.


DWF 1 - Logic Analyzer 1

Single | Run | Position 0 s | Base 200 ns/div | Buffer 100 of 100 | Clock Internal | Run Screen | Mode Auto | Source Analyzer | Trigger: DIO7=Falling

Done | 2014/01/30 16:45:04.478 | 4096 Samples at 100 MHz / 10 ns Zoom: 20.00 X | Position: | Value: | Name:

Name	DIO	Trigger
SW1	2	X
Timer	1	X
SysTick	0	X
PE2	6	X
PE1	4	X
PE0	3	X
PF4	7	⌐

Done

DWF 1 - Logic Analyzer 1

Single | Run | Position 2 us | Base 1 us/div | Buffer 100 of 100 | Clock Internal | Run Screen | Mode Auto | Source Analyzer | Trigger: DIO7=Falling

Ready | 2014/01/30 16:45:04.478 | 4096 Samples at 100 MHz / 10 ns Zoom: 4.00 X | Position: -1.69 us | Value: | Name:

Name	DIO	Trigger
SW1	2	X
Timer	1	X
SysTick	0	X
PE2	6	X
PE1	4	X
PE0	3	X
PF4	7	⌐

Ready

File Control View Settings Window Help
Export Data Events Zoom Options Help
Add Remove
-800 ns -600 ns -400 ns -200 ns 0 ns 200 ns 400 ns 600 ns 800 ns
-2 us -1 us 0 us 1 us 2 us 3 us 4 us 5 us 6 us

Figure 2.4. Logic analyzer output showing the latency of this real time thread.

Task 3: Hardware-triggered data acquisition and FFT

The third task also samples the ADC at a regular rate (choose a different sequencer and a different hardware pin). In this task, the ADC sampling rate is established with a hardware-triggered timer. This task will continuously sample data at 400 Hz. This means the ADC is started every 2.5 ms. It takes 64 ms to collect the block of 64 samples. An FFT is calculated on this block. This sampling rate is fixed and should not be increased or decreased. A timer runs at 400 Hz and is used to trigger the ADC. This timer does not interrupt. When each ADC is done, an ADC interrupt will be requested. This ADC ISR (**`Producer`**) will pass data to the foreground using a FIFO queue, by calling **`OS_Fifo_Put`**. The **`Consumer`** foreground thread calculates the FFT, and the **`Display`** foreground thread outputs to the LCD. A mailbox is used to synchronize the two foreground threads. Because the ADC is triggered in hardware, this sampling process has no jitter. We will classify this task as real time if neither the ADC hardware FIFO nor the OS software FIFO become full. In other words, a real time system does not loose data. Read the errata about timer-triggered ADC sampling. YOU MUST USE 16-BIT MODE FOR TIMER-TRIGGERED ADC SAMPLING.

Task 4: CPU bound task

The fourth task is a single foreground thread **`PID`**, which performs mathematical calculations with synchronizing to input or output hardware. This thread does not wait, signal, sleep or kill. Most operating systems must have at least one thread like this, so that something can run if all the other threads are blocked, dead, or sleeping. As mentioned earlier, a digital controller is typically a real time periodic task that has both input and output. However, in this lab, this PID controller runs continuously without I/O. This thread is not real time, so we do not calculate any latency or jitter for it. However, one performance measure for non real-time threads is the rate at which is performs calculations. In this lab, the variable **`PIDWork`** will specify the number of PID operations in the 20 second run. The larger **`PIDWork`** is, the more efficient is your OS. FYI, we expect **`FilterWork`** to be about 40000, because it runs at 2000 samples/sec.

Task 5: User interface with a command line interpreter

The fifth task is the command line interpreter developed in Lab 1. This task synchronizes with input and output from the UART. For a real-time system with input/output devices, the interface latency is important. For an input device, the **interface latency** is the time delay between when the hardware says the input is ready and the time when the software reads the data. Because of the 16-element hardware FIFO in the UART, the software must response to an input within 160 bit times or risk an overrun error (lost data.) For an output device, the interface latency is the time delay between when the hardware says the output is idle and the time when the software write new data to the device. The interface latency for the output task will affect the overall bandwidth, but there is usually no hard upper bound above which the system stops working. You are free to implement the UART I/O synchronization however you wish. However, you may also use an approach that uses no interrupts, as illustrated in Figure 2.5. The busy-wait synchronization uses the 16-element hardware FIFO and checks status bits (RxFE, TxFF) in the **`UART0_FR_R`** register. If busy, the foreground thread waits by calling **`OS_Sleep`**.

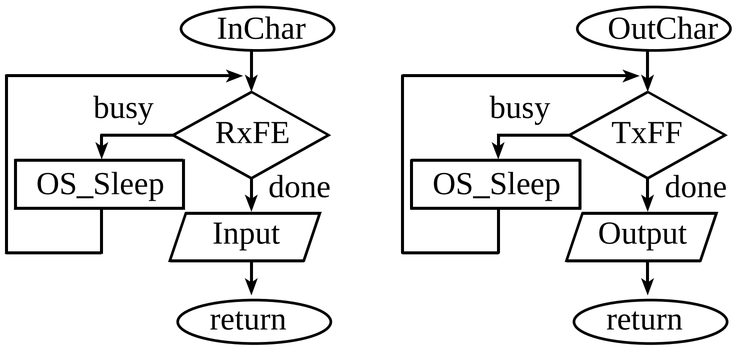


Figure 2.5. One possible synchronization method for the serial input/output.

If you use interrupts and software FIFO queues, have the UART interrupt on TXRIS, RXRIS or RTRIS. The receive operation is illustrated in Figure 2.6. There are two possible ways to get a receive interrupt. The RXRIS bit is set if the receive hardware FIFO gets big (e.g., 1/8 full means receive hardware FIFO goes from 1 to 2 elements). The RTRIS bit is set if there is a timeout (some data in the receive hardware FIFO and no input for 32 bit times). The ISR will copy as much data as it can from the hardware FIFO to the software FIFO. There is a counting semaphore, called RxDataAvailable and is initialized to 0. The RxFifo_Put will call signal if data is successfully entered, and RxFifo_Get will call wait. To acknowledge the receive interrupt, we write ones to the RXIC and RTIC bits in UART0_ICR_R.

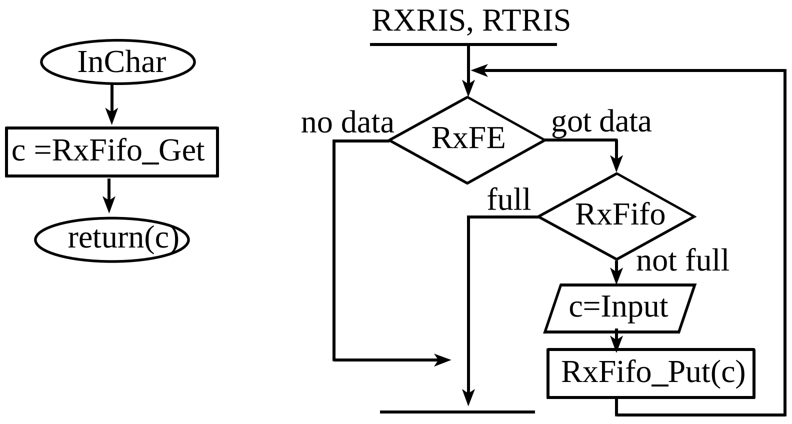


Figure 2.6. One possible interrupt synchronization for the serial input.

For the transmission channel there is a software FIFO (TxFifo) and the transmit hardware FIFO. The ISR is triggered by TXRIS, which occurs when the hardware FIFO goes from 3 to 2 elements (1/8 full). The software FIFO has a counting semaphore, TxRoomLeft, which is initialized to the maximum number of elements that can be saved in the software FIFO. The TxFifo_Put will wait on the semaphore and TxFifo_Get will signal the semaphore when data is successfully removed. The helper function TxCopy will transfer as much data as it can from the software FIFO to the hardware FIFO. The TxFifo_Get function is not reentrant, so OutChar disarms TXRIS before calling TxCopy. To acknowledge the transmit interrupt, we write 1 to the TXIC bit in UART0_ICR_R.

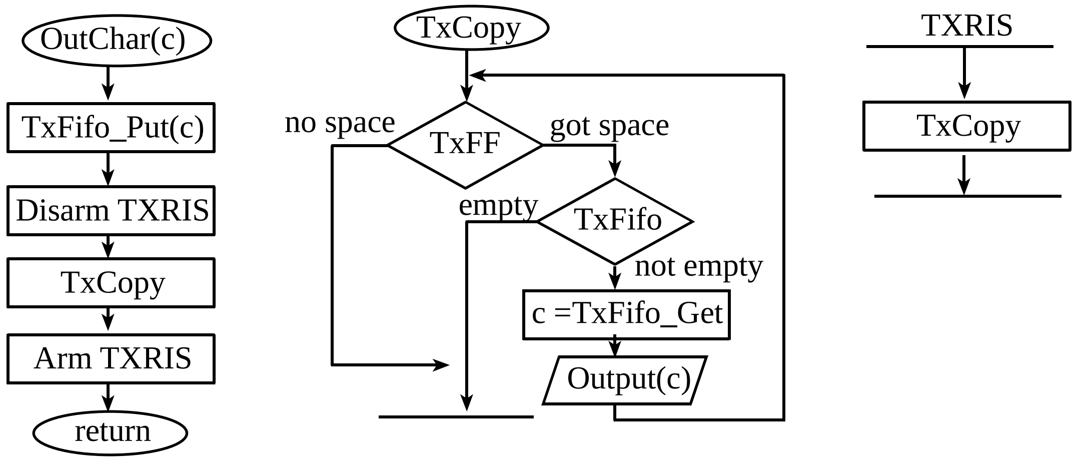


Figure 2.7. One possible interrupt synchronization for the serial output.

Not all software tasks in a real-time system require execution at specified times. For example, in a data acquisition and control systems, updating visual displays or saving the results in secondary storage can often be performed when the computer is free, i.e., not needed for time critical functions. Specifically in Lab 2, the threads **`DAS`** and **`ButtonPush`** must be real time, and the others are not. Task 3 will be real time if no data is lost. Losing data at the hardware level can be found making sure the appropriate bit in **`ADC0_OSTAT_R`** is never set. The number of data lost at the software level is calculated in the variable **`DataLost`**. Task 5 will be real time if the hardware FIFO in the receiver never becomes full. We will not specifically test or attempt to overflow the UART receive buffer in this lab. However, later in the CAN lab we will attempt to create this type of input overflow.

In Lab 2, we will pass data from a background thread (**`Producer`**) to a foreground thread (**`Consumer`**) using a buffered approach, see Figure 2.4. When **`Consumer`** needs information, it calls **`OS_Fifo_Get`**. If the FIFO is empty, it will spin/block on a semaphore **`FifoAvailable`** because it can not retrieve any information. On the other hand, **`Producer`** enters information by calling **`OS_Fifo_Put`**. If the FIFO is full, then it will not wait because one can not block or spin in an interrupt service routine. The spinning/blocking occurs in the **`OS_Wait`** routine.

When passing data between two foreground threads, we could use a buffered approach (**`DataFifo`**). Another name for this first in first out queue is **pipe**. The buffered communication between two foreground threads will not be used in this lab; it is presented for information purposes only. When using a **`DataFifo`** buffer, the two counting semaphores called, **`DataAvailable`** and **`DataRoomLeft`**, contain the number of entries currently stored in the message **`DataRoomLeft`** and the number of spaces left in the **`DataFifo`** respectively. **`DataAvailable`** is initialized to zero, and **`DataRoomLeft`** is initialized to the maximum allowable number of elements in the **`DataFifo`**. The **`Send`** routine, called by a foreground thread when it wishes to transmit data, could execute the following steps:

```
OS_Wait(&DataRoomLeft)
OS_bWait(&mutex)
Enter data into the DataFifo structure
OS_bSignal(&mutex)
OS_Signal(&DataAvailable )
```

The **`Receive`** routine, called by a foreground thread when it wishes to receive data, could execute the following steps:

```
OS_Wait(&DataAvailable)
OS_bWait(&mutex)
Remove data from the DataFifo structure
OS_bSignal(&mutex)
OS_Signal(&DataRoomLeft)
```

One thread creates data, and then sends the data to a second thread by calling **`Send`**. The second thread receives the data from the first thread by calling **`Receive`**. The **`DataFifo`** is used for interthread communication. In some applications there might be multiple producers and multiple consumers. When one of the threads involved in the buffered producer-consumer communication is a background thread, we must remove the **`OS_Wait`** call from the ISR. This is because only foreground threads will be allowed to spin or block.

In this lab, we will pass data from a foreground thread (**`Consumer`**) to another foreground thread (**`Display`**) using an unbuffered approach, see Figure 2.1. A single global (called **`MailBox`**) will contain the data passed from **`Consumer`** to **`Display`**. A binary semaphore, **`DataValid`**, is initialized to zero meaning the **`MailBox`** is empty. When **`DataValid`** equals one it means the mailbox has data in it placed by **`Consumer`** that has not been read by the **`Display`**. A second binary semaphore, **`BoxFree`**, is initialized to one meaning the **`MailBox`** is free. When **`BoxFree`** equals zero it means the mailbox contains data that has not yet been received. When **`Consumer`** wishes to send data it first waits/blocks on the semaphore **`BoxFree`**. Next, **`Consumer`** puts its data into the **`MailBox`**, and signals **`DataValid`**. Setting **`DataValid`** will allow **`Display`** to proceed. When **`Display`** executes **`Receive`**, it first waits/blocks on the semaphore **`DataValid`**, gets the data from the **`MailBox`** and signals **`BoxFree`**. Setting **`BoxFree`** will allow **`Consumer`** to proceed. It doesn't matter whether the **`Display`** or **`Consumer`** executes first, they will wait for each other. The **`OS_MailBox_Send`** routine executes the following steps:

```
OS_bWait(&BoxFree)
Put data into the mailbox
OS_bSignal(&DataValid)
```

The **`OS_MailBox_Recv`** routine executes the following steps:

```
OS_bWait(&DataValid)
Retrieve the data from the mailbox
OS_bSignal(&BoxFree)
```

When we perform LCD output we will need a mechanism to share this resource. You will have to implement mutual exclusion (only one thread at a time can call the LCD output functions.) The traditional term for this type of semaphore is **mutex**. We will call our semaphore **`LCDFree`**. It will prevent more than one thread from outputting at the same time. It is initialized to 1 that means the LCD display available. When the **`LCDFree`** semaphore is zero, it means no displays are free (a thread is currently doing output.) Some operating systems provide special support for this true/false type of semaphore, calling it a binary semaphore. For example if you wished to output a message, then a thread could call a function like the following (you are free to implement whatever syntax you wish for the LCD):

```
void ST4773_Message (int device, int line, char *string, long value){
  OS_bWait(&LCDFree);         // capture resource
// output
  OS_bSignal(&LCDFree);  // release resource
}
```

Before you begin writing code for this lab, please review the user programs that your OS will be running (i.e., the starter file **`Lab2.c`**). You are free to change the syntax of the I/O functions and the OS calls, as long as the fundamental approach is unchanged.

For this part of the lab, you will implement spinlock semaphores, and a preemptive round robin scheduler. Threads will be activated at run time. The memory for the TCB and stack can be found by calling **malloc**, or you could have a finite set of buffers and when you need a TCB or a stack you could select a free buffer to use. Many students have had problems debugging their software that used malloc. Therefore, we suggest you write your own very simple memory manager that handles fixed-sized blocks. The priority field will not be used until Lab 3. The **`OS_Sleep`** function will temporarily prevent this thread from being run. When a thread calls **`OS_Kill`** it will no longer be run, and its TCB and stack will made available for adding other threads. The desired endpoint of the first part is to be able to run the system to completion without losing any data, and with a small time jitter.

Preparation Part 1 (do this before your lab period) (20 points Prep Part 1)

1) There are categories into which tasks may fall. First, there are **I/O bound** tasks, where the bandwidth (data processed each second) is limited by the I/O device. For example, in a data-entry task, it usually doesn't matter how fast the computer is, the amount of information entered into the system is limited by the input typing rate of the operator. In a similar fashion, the number of pages printed per second is usually limited by the printer speed, and not by the speed of the computer. The second category describes tasks with **fixed bandwidth**, and not limited by either software or hardware. For example, the weatherman collects temperature data every hour. Temperature measurements once an hour are all we need, so a faster ADC converter, a faster temperature sensor, or a faster computer would not enhance the performance of this system. The third category, **CPU bound**, describes tasks that are limited by the execution speed of the software. For these systems, a better software algorithm, a better compiler, and a faster computer will enhance the performance. There are five tasks in this system. Look at the user code and categorize the type of these five tasks.

2) Begin the design of the OS by defining the **TCB** data structure (in C). You may place the stack inside the TCB, or place it outside as shown in Program 4.4 of the book. The following items should be stored in the **TCB** for each thread:

- Current Stack Pointer, SP, for this thread (saved value when not actually running)
- One or two pointers to the next/previous TCB in the active list
- Id (a unique identifier for the thread, used for debugging)
- Sleep state (used to suspend execution)
- Priority (used in Lab 3)
- Blocked state (used in Lab 3)

3) Design **`OS_AddThread`**. Sketch the circular linked list of three TCBs that should be created after **`Testmain1`** calls **`OS_AddThread`** three times, but before **`Testmain1`** calls **`OS_Launch`**. Leave room to plug in actual numbers that will be collected during the testing phase of the project.

4) Write code to implement the thread switch triggered by **`OS_Suspend`** or a periodic interrupt service routine. Draw two rough sketches of the TCB linked list, before and after a thread switch. Leave room to plug in actual numbers that will be collected during the testing.

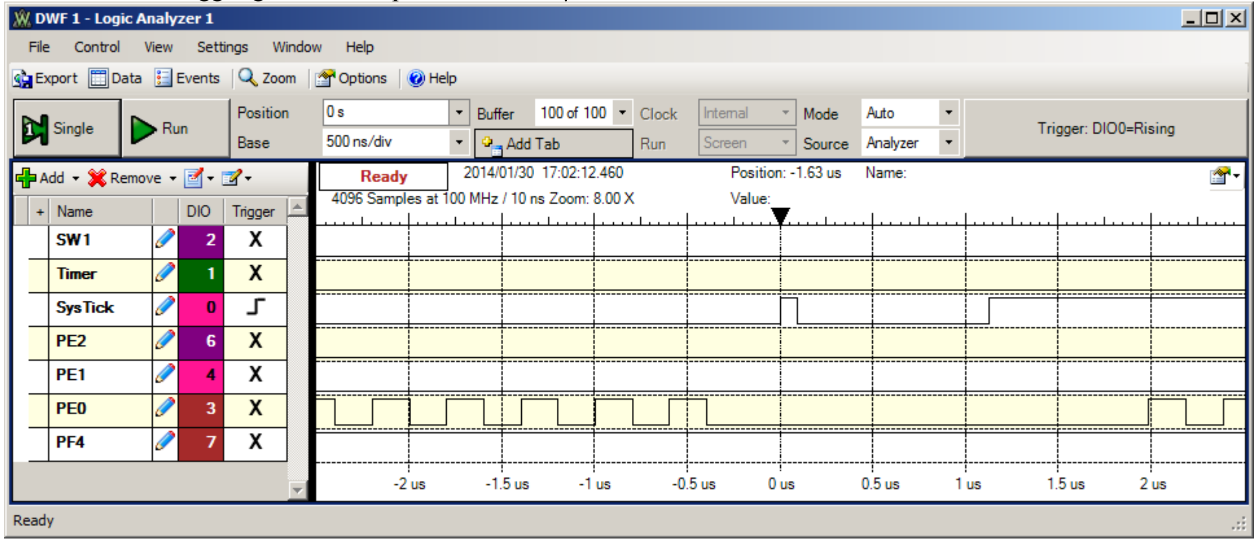
5) Make a development plan showing how the components of this lab will be designed and tested. In particular, look at the four test configurations at the end of **`Lab2.c`**.

Preparation Part 2 (do this before your lab period) (10 points Prep Part 2)
6) Write spinlock implementations for the **`OS_InitSemaphore`** **`OS_Signal`** **`OS_Wait`** **`OS_bSignal`** and **`OS_bWait`** routines. Add these synchronization utilities to your display driver. If you use interrupting serial port I/O, add semaphores to the software FIFOs. Be careful to consider critical sections.

Procedure Part 1
1) Implement the **`OS_AddThread`** function. Using the debugger, step through the first three calls to **`OS_AddThread`** and fill in the sketch (preparation 3) of the three TCBs with real data (e.g., specify a pointer as its actual hexadecimal values).

2) Implement and test the **`OS_Launch`** and thread switch functions. Using the debugger, step through the launch function, until the first thread starts to execute. Show what the three TCBs look like when one thread is running and the other threads are active. Fill in with real data the first sketch from preparation 4). Set a breakpoint in the thread switch routine. Single step through the routine until the next thread is running. Fill in with real data the second sketch from preparation 4).

Measure the context/thread switch overhead of the OS kernel. Figure 2.8 shows the logic analyzer occurring with only one active thread running (*Threadmain7* from the starter code). If you define the thread-switch time as the lost time of the PE0 toggling, this OS requires about 2.1 µs to switch threads.



Position: 0 s
Base: 500 ns/div
Buffer: 100 of 100
Clock: Internal
Run: Screen
Mode: Auto
Source: Analyzer
Trigger: DIO0=Rising

Ready 2014/01/30 17:02:12.460
4096 Samples at 100 MHz / 10 ns Zoom: 8.00 X
Position: -1.63 us
Name:
Value:

Name	DIO	Trigger
SW1	2	X
Timer	1	X
SysTick	0	↑
PE2	6	X
PE1	4	X
PE0	3	X
PF4	7	X

Ready

Figure 2.8. Logic analyzer output showing the thread-switch time.

Procedure Part 2
3) Implement and test your semaphores. One way to test them is to call **`OS_Signal`** from a periodic interrupt. Place **`OS_Wait`** in the body main program, then count the number of interrupts and the number of times the body of the loop is executed. In particular, see **`Testmain3`**.

4) Implement, test and profile the system. Connect unused output pins to a logic analyzer and add minimally intrusive debugging instruments to profile the RTOS. This profile collects information of both time and place (when and which thread is executing). Add a compile time switch to remove these profiling instruments. In this way you can determine the degree to which the debugging instruments themselves affect system performance. In particular, measure **`PIDWork`** with and without the debugging instruments. In addition, once you get to the robot in Lab 7, you will need all the I/O pins for interfacing components.

5) Try running the system with lots of select pushes. Increase the size of the **`OS_Fifo`** until no data is lost. Test the system with operator input to the **`Interpreter`** too. Take measurements for three **`OS_Fifo`** sizes and three time slices (5 runs) when no input occurs in tamper or the **`Interpreter`**. Make a table showing the three performance parameters (time-jitter, number of data points lost, number of **`PIDWork`** calculations performed) versus the **`TIMESLICE`** and the **`FIFOSIZE`**. Table 2.1 shows an example of the 5 runs required for this part.

Checkout Part 1 (show this to the TA) (20 points performance, 20 points oral understanding)
1) Run *Testmain1*, *Testmain2* and *Testmain7* and explain the profiling data to the TA,
2) Be prepared to discuss the sketches you created as part of the preparation, and procedure
3) Discuss the TCB before and after a thread switch.

Checkout Part 2 (show this to the TA) (25 points performance, 25 points oral understanding)
1) Run the software system and explain the profiling data to the TA,
2) Be prepared to discuss the sketches you created as part of the preparation, and procedure
3) Discuss the TCB before and after a thread switch.
4) Identify inefficiencies in your implementation.

FIFOSize	TIMESLICE (ms)	DataLost	Jitter (us)	PIDWork
4	**2**	3763	7	1285
8	**2**	0	7	1285
32	**2**	0	7	1285
32	**1**	0	7	1283
32	**10**	0	7	1287

Table 2.1. Example performance data for procedure 5 (collected with no interpreter or switch input).

Deliverables Part 1 (none)

Deliverables Part 2 (components of lab report and submission) (20 points report, 10 points software quality)
A) Objectives (1/2 page maximum)
B) Hardware Design (none for this lab)
C) Software Design (documentation and code of spinlock/round-robin operating system)
D) Measurement Data
    1) plots of the logic analyzer like Figures 2.1, 2.2, 2.3, 2.4, and 2.8
    2) measurement of the thread-switch time
    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
    5) a table like Table 2.1 each showing performance measurements versus sizes of **`OS_Fifo`** and timeslices
    6) a table showing performance measurements with and without debugging instruments
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 6 μs when interpreter I/O occurred?
  2) Justify why Task 3 has no time jitter on its ADC sampling.
  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?
  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?
  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.
  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?
  7) Without going back and actually measuring it, do you think the **`Consumer`** ever waits when it calls **`OS_MailBox_Send`**? Explain.

Hints
1) Review how the μCOS-II or μCOS-III thread switching occurs. Please reference all software you copy/paste.

2) Make small changes and save the changes using new file names, so that when something doesn't work you can go back to a version that does work and try something new. You will have to debug this system in small very parts. A mechanism to visualize the real time execution will be helpful.

3) You do not have to implement **`OS_Wait`** and **`OS_Signal`** with **`LDREX`** and **`STREX`**, but you do have to remove critical sections.

4) Avoid using breakpoints and single stepping on the real microcontroller. Remember to use the minimally intrusive debugging techniques that you have learned. If you store data into global memory, the information should be available for viewing even after a crash or a hardware reset. Debuggers get very confused when you change the stack pointer.

5) *Look for the most recent files on the network.*

6) Please ask the instructor or the TA for help in clarifying any details you don't understand.

7) The **`OS.h`** starter file includes both binary and counting semaphores. If there is no efficiency advantage of binary semaphores over counting semaphores, you are free to delete the binary semaphores and just use counting semaphores.

8) You can ignore the thread stack size parameter and simply create a fixed number of fixed size TCB/stacks. This will assist you in debugging. You do not have to use **`malloc`** and **`free`**, but once a thread calls **`OS_Kill`**, its TCB and stack should be available to use if the user calls **`OS_AddThread`**.

9) Feel free to sample the internal temperature rather than soldering wires, or to sample the real input without connecting an analog signal. There are no action items on the ADC data you collect in Labs 1, 2, and 3.

10) Make sure your OS does not return when a thread calls **`OS_Kill`**.