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)
- **`ADCT0ATrigger_4C123`** (timer-triggered ADC)
- **`RTOSVerySimple_4C123`** project (simplest OS)
- **`RTOS_4C123`** project (basic OS)
- **`RTOS_Lab2_RTOSkernel`** (starter project for Lab 2)
  - **`cr4_fft_64_stm32.s`** and **`PID_stm32.s`** (from **inc** folder)
  - **`OS.h`** and **`OS.c`** (from **RTOS_Labs_common** folder)
  - **`Lab2.c`** (in **RTOS_Lab2_RTOSkernel**)

References (good to look at for more background information, but not required)
1) Signal processing functions used in this lab: http://www.ece.utexas.edu/~valvano/EE345M/UM0585.pdf
2) Micrium µC/OS-II sample RTOS: https://github.com/weston-embedded/uC-OS2 (Cortex-M ARMv7 port)
3) FreeRTOS: http://www.freertos.org/ (look at the Cortex M3 port)

This lab has two parts. In Part 1, you will implement basic multi-threading support in your OS. In Part 2, we will extend the multi-tasking to support larger applications with inter-thread communication. Note the Hints at the end!

Background Part 1

In the first part of Lab 2 you will first design and test a cooperative, and then a preemptive thread scheduler. The requirements will be to run the *Testmain1* and *Testmain2* applications provided in **`Lab2.c`**, 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 PD0, PD1, PD2 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.

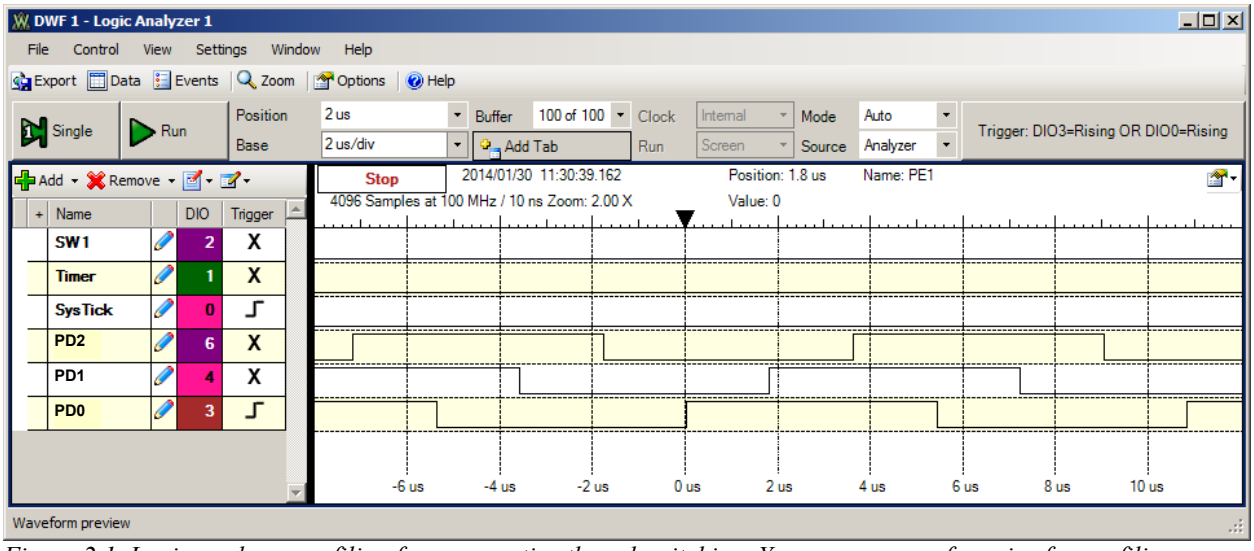

Figure 2.1. Logic analyzer profiling for cooperative thread switching. You may use any free pins for profiling.

By contrast, *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, PD0, PD1, PD2 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.

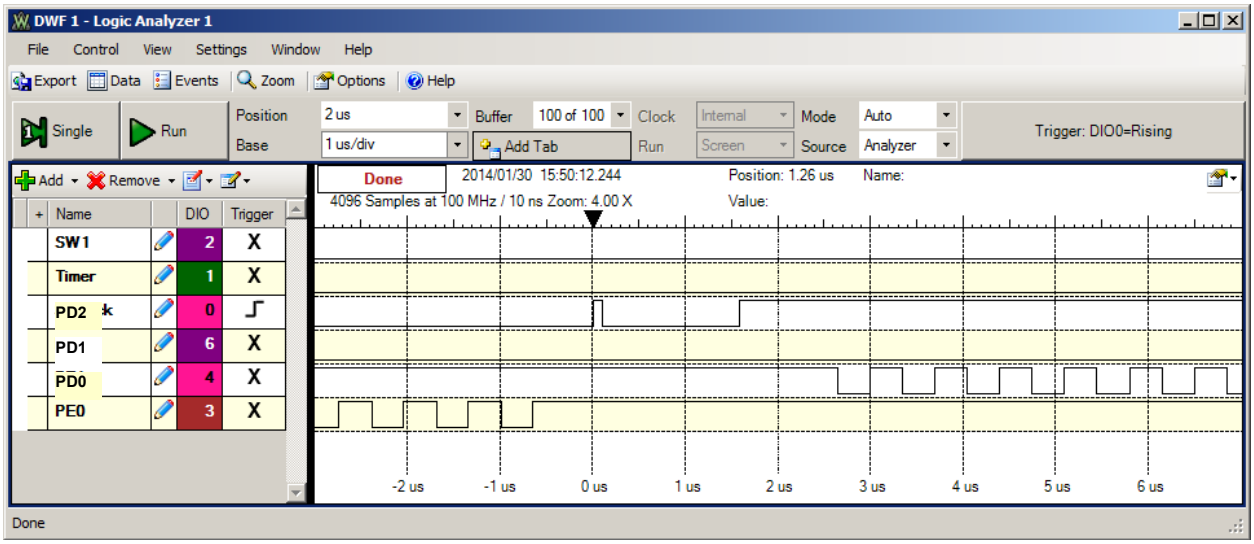

Figure 2.2. Logic analyzer profiling for preemptive thread switching (zoomed in). You may use any free pins.

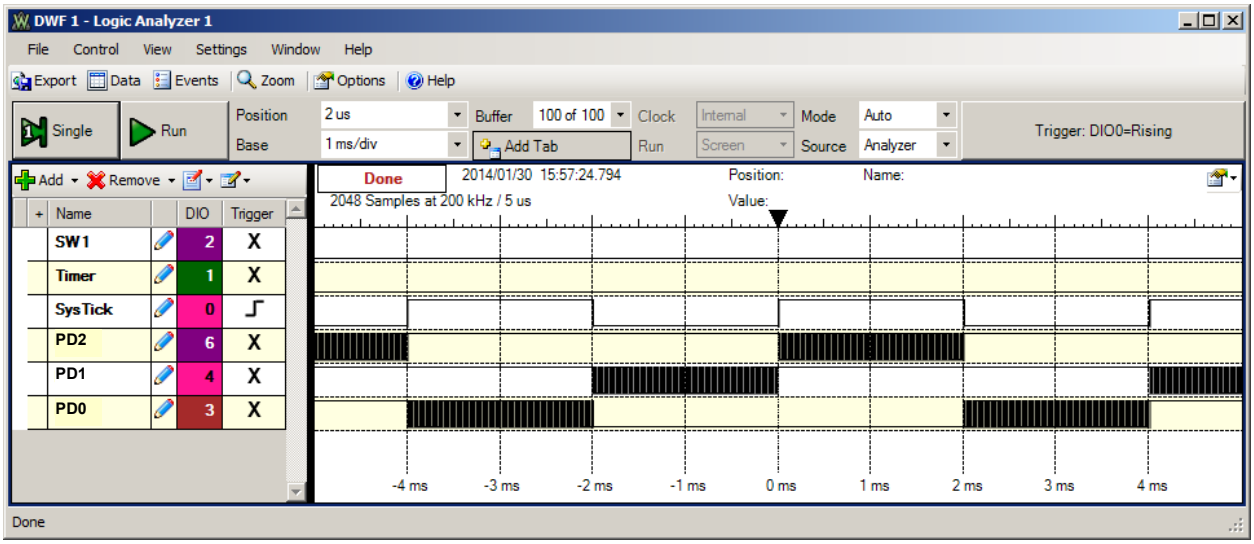

Figure 2.3. Logic analyzer profiling for preemptive thread switching (zoomed out). You may use any free pins.

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

1.1) 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 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)

1.2) 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.

1.3) 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.

1.4) Make a development plan showing how the components of this lab will be designed and tested. In particular, look ahead at Part 2 and the five test configurations at the end of **`Lab2.c`**.

Procedure Part 1

1.1) Implement the **`OS_AddThread`** function. The memory for the TCB and stack can in general be found by calling a dynamic memory manager such as **malloc**, or one can have a finite set of buffers, and when you need a TCB or a stack you could select a free buffer to use. We will implement a dynamic memory manager in Lab 5. Therefore, for this lab, you should write your own very simple TCB manager using pre-allocated fixed-sized buffers. Using the debugger, step through the first three calls to **`OS_AddThread`** and fill in the sketch (preparation 1.2) of the three TCBs with real data (e.g., specify a pointer as its actual hexadecimal values).

1.2) 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 1.3). 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 1.3).

1.3) Measure the context/thread switch overhead of the OS kernel. Figure 2.4 shows the logic analyzer occurring with only one active thread running (using *ThreadmainCS* provided in the starter code). If you define the thread-switch time as the lost time of the PD0 toggling, this OS requires about 2.1 µs to switch threads.

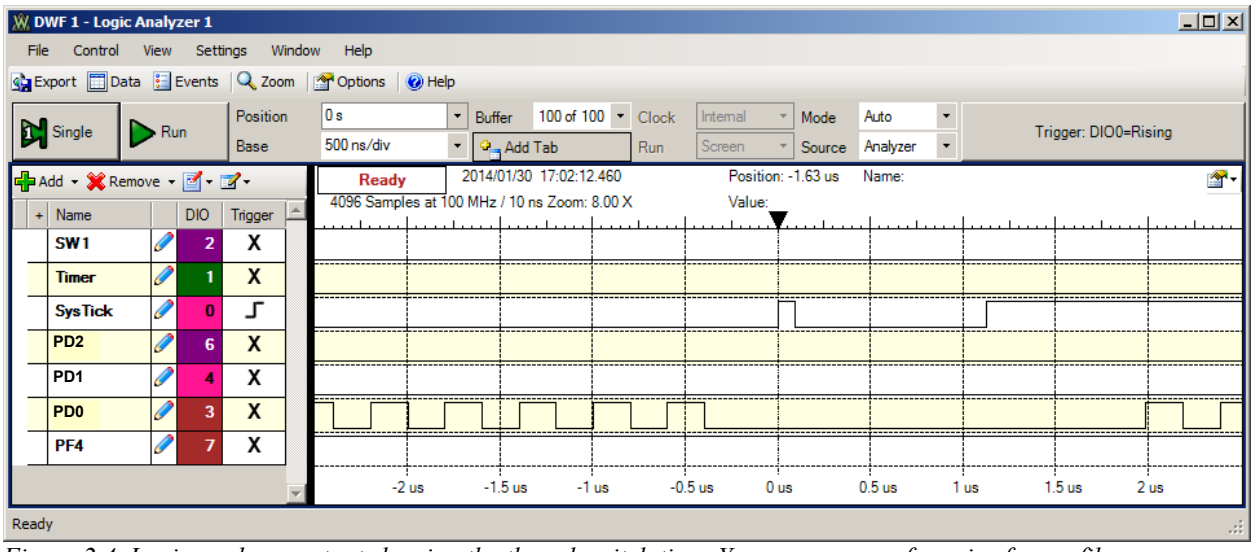

Figure 2.4. Logic analyzer output showing the thread-switch time. You may use any free pins for profile.

Checkout Part 1 (show this to the TA) (20 points performance, 20 points oral understanding)

1.1) Run *Testmain1*, *Testmain2* and *TestmainCS* and explain the profiling data to the TA.
1.2) Be prepared to discuss the sketches you created as part of the preparation, and procedure.
1.3) Discuss the TCB before and after a thread switch.

Background Part 2

For this part of the lab, you will implement a preemptive round robin scheduler, spinlock semaphores, and inter-thread communication. Threads will be dynamically activated at run time. An **`OS_Sleep`** function will be added to temporarily prevent the calling thread from being run. Furthermore, when a thread calls **`OS_Kill`** it will no longer be run, and its TCB and stack must be reclaimed and made available for adding other threads.

The desired endpoint of this lab is to design, implement and test operating system commands that implement a real multi-task application with I/O and inter-thread dependencies. Your job is to be able to run the provided system to completion without losing any data, and with a small time jitter. 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). In real-time applications, the scheduling of tasks is critical for the proper operation of the system. 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 overall tasks to manage in Lab 2 (and later in Lab 3). Figure 2.5 shows the data flow for Lab 2. 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.

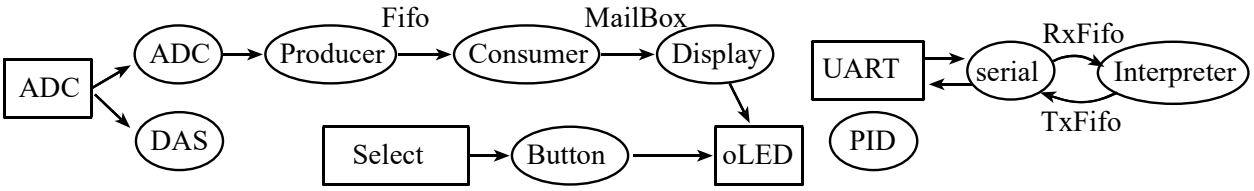

Figure 2.5. Data flow graph for the user program.

Task 1: Software-triggered periodic data acquisition and filtering

For the data acquisition system (**`DAS()`** task, 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 f_s: $\Delta t = 1/f_s$. In Labs 2 and 3, Δt is 500 μs and f_s is 2 kHz.

An example of software-triggered periodic tasks are digital control systems. The software must read the sensors, perform the digital control equations, and output to the actuators at a fixed rate. Another example 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 $n\Delta t$ 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 Δt_n be the actual time difference between two executions of a software task (e.g., starting the ADC). The desired time difference is $1/f_s$. 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 δt_n with 0.1 μs units. **PERIOD** is a constant, in units of bus cycles, that represents the desired $1/f_s$. **diff** is a measurement, also with units of bus cycles, that represents the actual Δt_n. **jitter** is the calculation of δt_n with units of 0.1 μs.

```
uint32_t 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 (f_s = 2kHz, bus = 80MHz). 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** $\le$ **MaxJitter**

Assume that the DAS task was activated using an **OS_AddPeriodicThread(&DAS,PERIOD,1)** call (to be developed in this lab). 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. 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**/**OS_Signal** using **LDREX** and **STREX** instructions. Most of the time, the DAS task ran every 40000 cycles exactly, i.e. at exactly equal time intervals. We can see that the time jitter is less than 0.4 μs under the measurement conditions. With interpreter I/O, it was 0.6 μs. This error is acceptable in most situations, 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, and 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 2.6 shows a measurement of latency. The falling edge of PF4 occurs when the switch is touched, and the rising edge of SW1 occurs in its ISR (a corresponding heartbeat is added in the **GPIOPortF_handler** – we recommend to use PC4, shown as SW1 in the figure). For this instance, the latency is 400ns, and the time to run the ISR is less than 5 μs. This OS toggles SW1 three times in the ISR.

When we perform LCD output in multiple threads, 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) to avoid access conflicts. The traditional term for this type of semaphore is **mutex**. We can call our mutex 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 (uint32_t device, uint32_t line,
                     char *string, int32_t value){
  OS_bWait(&LCDFree);          // capture resource
  // output as written in Lab 1
  OS_bSignal(&LCDFree);  // release resource
}
```

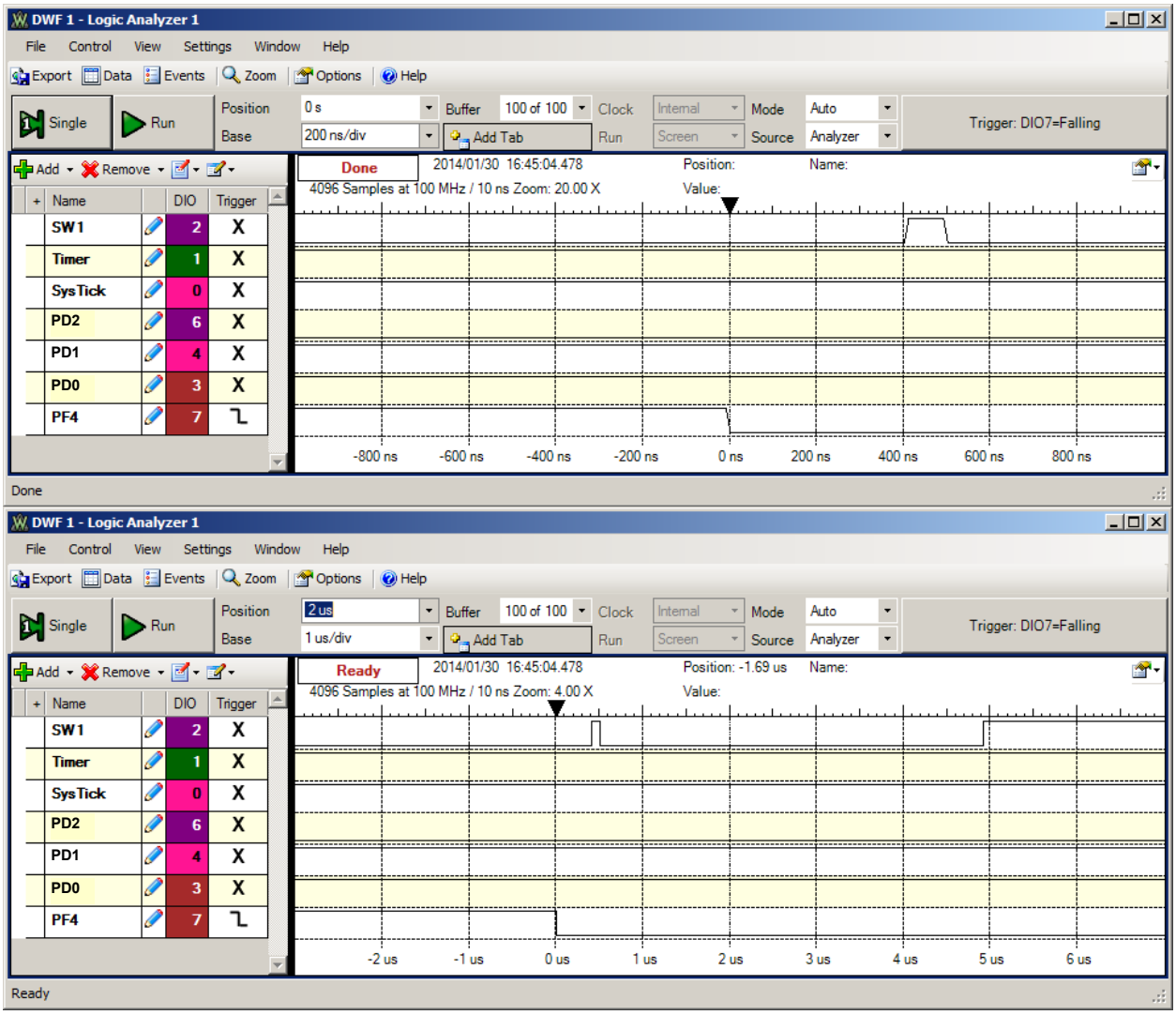

Figure 2.6. Logic analyzer output showing the latency of the SW1 real time thread. You may use any free pins.

Task 3: Hardware-triggered data acquisition and FFT

The third task also samples the ADC at a regular rate (using 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. 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 becomes full. In other words, a real time system does not loose data.

In this lab, we will pass data from a background thread (**Producer**) to a foreground thread (**Consumer**) using a buffered approach, see Figure 2.5. You will need to implement a FIFO with thread synchronization that is properly integrated with and tied into your OS. In particular, a counting semaphore (e.g. **FifoAvailable**) contains the number of spaces left in the FIFO and is initialized to the maximum allowable number of elements in the FIFO. when the **Consumer** needs information, it calls **OS_Fifo_Get**, which will try to acquire and decrement the semaphore. If the FIFO is empty, the semaphore will be zero and the **Consumer** will need to wait and spin/block on the semaphore to let other threads run until it can retrieve any information. The **Producer** enters information by calling **OS_Fifo_Put**, which notifies the semaphore. However, if the FIFO is full, it must not wait because one can not block or spin in an interrupt service routine.

When passing data between two foreground threads, we can use a similar buffered approach. Another name often used for this type of first in first out (FIFO) queue is **pipe**. The buffered FIFO communication between two foreground threads will not be used in this lab. Instead of using a buffered FIFO, we will pass data from a foreground thread (**Consumer**) to another foreground thread (**Display**) using an unbuffered approach, see Figure 2.5. A single global **MailBox** will contain the data passed from **Consumer** to **Display**. You will need to implement **OS_MailBox_Send** and **OS_MailBox_Recv** routines that use two binary semaphores to synchronize thread accesses to a mailbox. The first semaphore (e.g. **DataValid**) is initialized to zero and indicates whether the mailbox has new data that has not been picked up yet. The second semaphore (e.g. **BoxFree**) is initialized to one and indicates whether the mailbox is empty and can accept data. The **OS_MailBox_Send** routine first waits/blocks on the **BoxFree** semaphore, then puts the data into the mailbox and signals **DataValid**:

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

The **OS_MailBox_Recv** routine in turn first waits/blocks on the semaphore **DataValid**, then gets the data from the **MailBox** and signals **BoxFree** to allow the next sender to proceed.

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

Task 4: Foreground PID controller 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 first 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 respond 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. You may use an approach that uses no interrupts, as illustrated in Figure 2.7. 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**.

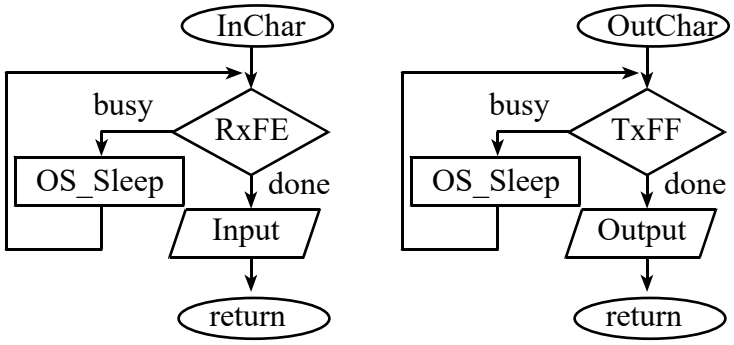

Figure 2.7. One possible synchronization method for the serial input/output.

The Lab 1 starter code provided a UART implementation that uses interrupts and software FIFO queues (**`UART0int.h`** and **`UART0int.c`** in **`RTOS_Labs_common`**). The UART interrupts on TXRIS, RXRIS or RTRIS. 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) while the RTRIS bit is set if there is a timeout (some data in the receive hardware FIFO and no input for 32 bit times). In either case, the ISR will copy as much data as it can from the hardware receive FIFO to the software receive FIFO (RxFifo). To acknowledge the receive interrupt, we write ones to the RXIC and RTIC bits in UART0_ICR_R. For the transmission channel, there is a software transmit FIFO (TxFifo) and the hardware transmit FIFO. The ISR is triggered by TXRIS, which occurs when the hardware FIFO goes from 3 to 2 elements (1/8 full). It will transfer as much data as it can from the software transmit FIFO to the hardware transmit FIFO. To acknowledge the transmit interrupt, we write 1 to the TXIC bit in UART0_ICR_R. In addition to the ISR, the **`UART_OutChar`** routine itself will also try to copy as much data as it can from the software into the hardware FIFO after putting the data to send into the transmit FIFO. Note that since the copying can be called from both the ISR and the foreground threat but **`TxFifo_Get`** is not reentrant, **`UART_OutChar`** disarms TXRIS before copying.

In this lab, you will have to modify the interrupt-based and buffered UART implementation to replace any busy waiting with proper FIFO synchronization on top of your OS. On the receive side, a counting semaphore, e.g. called **`RxDataAvailable`** indicates the number of entries currently stored in the receive FIFO and is initialized to 0. The **`RxFifo_Put`** will call signal if data is successfully entered, and **`RxFifo_Get`** will call wait. On the transmit side, the software FIFO has a counting semaphore (e.g. **`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.

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.

Preparation Part 2 (do this before your lab period) (10 points Prep Part 2)

2.1) There are five tasks in this system. Look at the user code and categorize the type of these five tasks as I/O bound, fixed bandwidth, or CPU bound.

2.2) Design your implementations of the **`OS_Sleep`** and **`OS_Kill`** functionality.

2.3) 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 2

2.1) Implement the **OS_AddPeriodicThread** routine that executes a background task at a periodic rate:

```
int OS_AddPeriodicThread(void(*task)(void),
                         uint32_t period, uint32_t priority);
```

where **task** is a pointer to the function to execute every **period** milliseconds, and **priority** is the value to be specified in the NVIC for this thread. You are free to specify the units of **period** however you wish. You can simply start with the code provided in the Lab 1 project to launch the **DAStask** there (see **Lab1.c**). Note, however, that for later labs, we will need to generalize the function to be able to launch multiple periodic background threads running at the same time.

2.2) Similarly, implement a **OS_AddSW1Task** routine that launches a given task whenever SW1 is pressed:

```
int OS_AddSW1Task(void(*task)(void), uint32_t priority);
```

You can refer to the **EdgeInterrupt_4C123** starter project for example code. In Lab 2, we will only be using SW1, but in Lab 3 we will add a similar routine for SW2.

2.3) Implement and test your **OS_Sleep** and **OS_Kill** functionality. You can use the provided *Testmain3* to test your sleep and kill functionality, but you should also develop your own test cases and variants.

2.4) 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 *Testmain4* and *Testmain5*.

2.5) Implement and test **OS_MailBox_Init**, **OS_MailBox_Send**, **OS_MailBox_Recv**, **OS_Fifo_Init**, **OS_Fifo_Get** and **OS_Fifo_Put** inter-thread communication primitives. You can use the provided *TestmainFIFO* to test the FIFO primitives. A similar simple test case can be developed to test the mailbox.

2.6) Add **Interpreter** commands to display performance metrics. These metrics include timer-jitter, the number of data points lost, or the number of calculations performed. At minimum, you must be able to display **NumCreated** (number of threads created) and **MaxJitter** (maximum time jitter) but additional information may be useful for debugging.

2.7) Implement, test and profile the complete system (*realmain*). 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.

2.8) 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.

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 2.7 (collected with no interpreter or switch input).

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

Deliverables (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 2.6
2) measurement of the thread-switch time
3) plot of the logic analyzer running the spinlock/round-robin system (profile data)
4) the three sketches (from first preparation parts 1.2 and 1.3), 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
0) Start early! Lab 2 (especially the checkout for part 2) is much harder than Lab 1 because you'll undoubtedly get unexpected hard faults that can stump you for several hours at a time. Many students had to checkout late in previous years. Even if you pass *Testmain4* and *Testmain5*, *realmain* will be a lot harder.

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. The comments in the starter code tell you what the expected output for the testmains are and lots of other useful information. A mechanism to visualize the real time execution will also be extremely helpful.

3) *Look for the most recent files on the class website. (there may be corrections)*

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

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

6) 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.

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

10) The starter project already includes the necessary code for hardware-triggered ADC sampling (in the **Consumer** task). You are free to modify the code, but read the errata about timer-triggered ADC sampling. YOU MUST USE 16-BIT MODE FOR TIMER-TRIGGERED ADC SAMPLING.

11) Make sure your other timers are in 32-bit mode. Do not use code from the **ADCT0ATrigger_4C123** project for normal timers, start with any of the periodic timer projects (such as **PeriodicTimer0AInts_4C123**).

12) The starter code also already has some switch debouncing code, but feel free to write your own.

13) 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. However, if you have a robot sensor board and an IR sensor, then you can measure distance similar to the robot competition at the end of the class.

14) Be careful of any pins that you choose since some are being used by the sensor board already (see the sensor board schematic on the *Resources* tab of the class website).