ECE 4501/6501: Advanced Embedded Systems

Mini Project 2
Basic RTOS Kernel: TCB Management, Basic Scheduling, Timing Features
Due Date: Wednesday, Oct. 10, 2018, 11:59 PM
(Accept Until: Oct. 12, 2018, 11:59 PM)

In this mini project, you will develop and test the basic routines for implementing a real-time operating system (RTOS). Your RTOS will run multiple foreground threads using cooperative and preemptive scheduling. You will test the performance using a number of provided tests.

Go to the following link to accept the Mini Project 2 assignment in the GitHub Classroom: https://classroom.github.com/a/z1EFbIKb. You will get access to a private repository created for you in the @UVA-embedded-systems organization on GitHub which contains the starter files for the mini project. You can use this repository for developing and testing your code. Your submission will also be to this repository.

You are provided with skeleton code including a Thread Control Block (TCB) structure implemented as a node of a singly linked list. The TCB holds thread-specific information such as the thread stack pointer and the thread identifier. You will add more fields to the TCB structure as you add features to your RTOS.

```
#define NUMTHREADS  20      // Maximum number of threads
#define STACKSIZE   100     // Number of 32-bit words in stack

// TCB Data Structure
struct tcb {
  int32_t *sp;            // Pointer to stack (valid for threads not running)
  struct tcb *next;       // Pointer to the next TCB
  uint32_t id;            // Thread Identifier
  uint32_t available;     // Used to indicate if this tcb is available or not
};
typedef struct tcb tcbType;

tcbType *RunPt;
tcbType tcbs[NUMTHREADS];
int32_t Stacks[NUMTHREADS][STACKSIZE];
```

The skeleton code also includes the routines **`OS_Init`**, **`OS_Launch`**, and **`Scheduler`** in **`os.c`**. The routines **`OS_Start`** and **`SysTick_Handler`** are implemented in assembly in **`osasm.s`**. The pointer **`RunPt`** should point to the TCB of the currently running thread in the RTOS. It is modified by the **`Scheduler`** and used by the **`SysTick_Handler`** to switch between threads. The **`tcbs`** and **`Stacks`** arrays are statically allocated arrays that are used to store, respectively, the TCBs and the stacks of the threads. You should study these routines and data structures and understand how they work.

You will first add functionality to add and remove TCBs to your RTOS. You will then implement a cooperative thread scheduler where threads voluntarily give control back to the RTOS by calling the **OS_Suspend** function. You will compare this with the default round robin preemptive scheduler where each thread gets a time slice. The provided **MiniProject2Test.c** file contains **main** programs initializing and running a number of threads. You will use these threads and the Keil Logic Analyzer tool to measure timing information about the RTOS. Finally, you will add some timing features to your RTOS and compare their results to the logic analyzer measurements.

Mini Project Parts

Part 1) In this part, you will develop some basic RTOS functions.

A – Develop the **OS_AddThread** function in **os.c**:

int OS_AddThread(void(*task)(void), unsigned long stackSize, unsigned long priority)

where **task** is a pointer to a thread function, **stackSize** is the number of bytes to be allocated for its stack, and **priority** is the priority of the thread for scheduling purposes. This function attempts to add a TCB to a circular linked list of TCBs and initialize its stack. It returns 1 if successful and 0 if unsuccessful.

(**Hint:** You can ignore both the **stackSize** and **priority** fields in your implementation. The memory for the TCB and stack can be dynamically allocated by calling C functions **malloc** and **free**, but you might hit problems debugging such code. Therefore, we suggest you simply use the provided arrays **tcbs** and **Stacks** with finite (fixed size) set of elements as memory and use the **available** field in the TCBs for marking free (i.e., available) elements in the array and finding them when **OS_AddThread** needs a new TCB and a stack).

```
int OS_AddThreads(void(*task0)(void), void(*task1)(void), void(*task2)(void)){
  int32_t status;
  status = StartCritical();
  tcbs[0].next = &tcbs[1]; // 0 points to 1
  tcbs[1].next = &tcbs[2]; // 1 points to 2
  tcbs[2].next = &tcbs[0]; // 2 points to 0
  SetInitialStack(0);
  SetInitialStack(1);
  SetInitialStack(2);
  Stacks[0][STACKSIZE-2] = (int32_t)(task0); // PC
  Stacks[1][STACKSIZE-2] = (int32_t)(task1); // PC
  Stacks[2][STACKSIZE-2] = (int32_t)(task2); // PC
  RunPt = &tcbs[0];       // Thread 0 will run first
  EndCritical(status);
  return 1;               // Successful
}
```

The example function **OS_AddThreads** above shows how to add three threads, initialize their stacks, and link their TCBs together to form a circular linked list. The function **SetInitialStack** shown below takes an integer corresponding to the index of a TCB in the **tcbs** array and initializes its stack. You may use this function in your implementation.

```
void SetInitialStack(int i){
  tcbs[i].sp = &Stacks[i][STACKSIZE-16]; // thread stack pointer
  Stacks[i][STACKSIZE-1] = 0x01000000;   // thumb bit
  Stacks[i][STACKSIZE-3] = 0x14141414;   // R14
  Stacks[i][STACKSIZE-4] = 0x12121212;   // R12
  Stacks[i][STACKSIZE-5] = 0x03030303;   // R3
  Stacks[i][STACKSIZE-6] = 0x02020202;   // R2
  Stacks[i][STACKSIZE-7] = 0x01010101;   // R1
  Stacks[i][STACKSIZE-8] = 0x00000000;   // R0
  Stacks[i][STACKSIZE-9] = 0x11111111;   // R11
  Stacks[i][STACKSIZE-10] = 0x10101010;  // R10
  Stacks[i][STACKSIZE-11] = 0x09090909;  // R9
  Stacks[i][STACKSIZE-12] = 0x08080808;  // R8
  Stacks[i][STACKSIZE-13] = 0x07070707;  // R7
  Stacks[i][STACKSIZE-14] = 0x06060606;  // R6
  Stacks[i][STACKSIZE-15] = 0x05050505;  // R5
  Stacks[i][STACKSIZE-16] = 0x04040404;  // R4
}
```

B – Develop the **OS_Suspend** function in **os.c**, which is called by a currently running thread when it wants to return control back to the RTOS. This function needs to make the SysTick interrupt trigger with software. It should also make sure the next thread starts with a new time slice. This implements a cooperative scheduler.

C – Implement the **OS_Kill** function in **os.c**, which kills the currently running thread by releasing its TCB and stack, making the memory available to be used for creating new threads. It should return control back to RTOS by calling the **OS_Suspend** function.

Deliverables for Part 1: Your code.

◆

Part 2) In this part, you will test your cooperative and preemptive thread schedulers and measure the context/thread switch overhead of the RTOS using the Keil Logic Analyzer tool. You should run the test functions provided in **MiniProject2Test.c** For each part, you should rename the relevant **TestmainX** function to **main**, as you can only have one **main** function.

A – **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 should look like Figure 1, and the three count variables will be equal (±1). In Figure 1, each toggle means a thread has started a pass through its main loop.

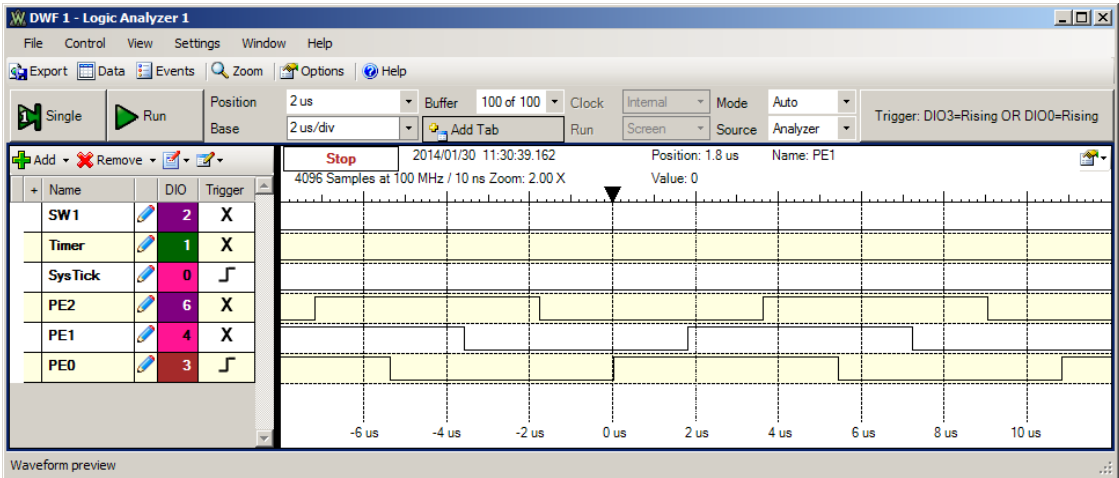

Figure 1: Logic Analyzer Profiling for Cooperative Thread Switching

B – **`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 and 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. In Figure 2, each toggle means a thread has started a pass through its main loop.

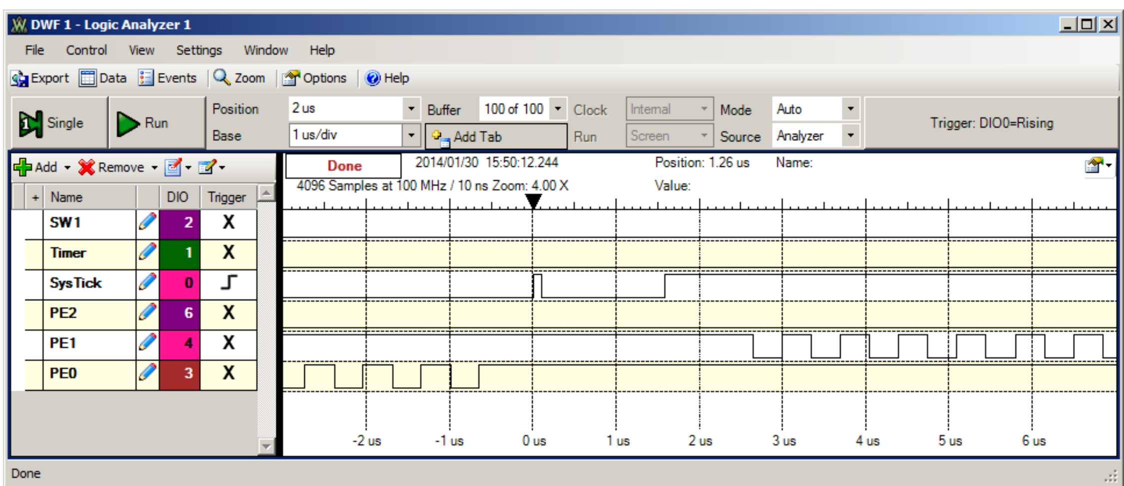

Figure 2: Logic Analyzer Profiling for Preemptive Thread Switching (Zoomed In)

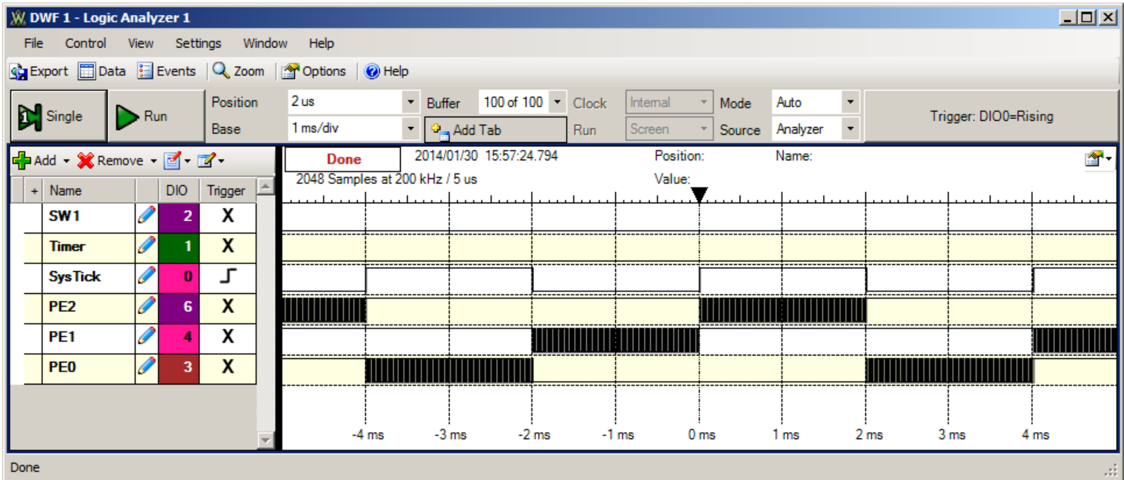

Figure 3: Logic Analyzer Profiling for Premptive Thread Switching (Zoomed Out)

C – Use **`Testmain0`** and Keil logic analyzer to measure the context/thread switch overhead of the RTOS kernel. Figure 4 shows the logic analyzer occurring with only one active thread running and toggling PE0. We can define the thread-switch time as the lost time of the PE0 toggling by the thread.

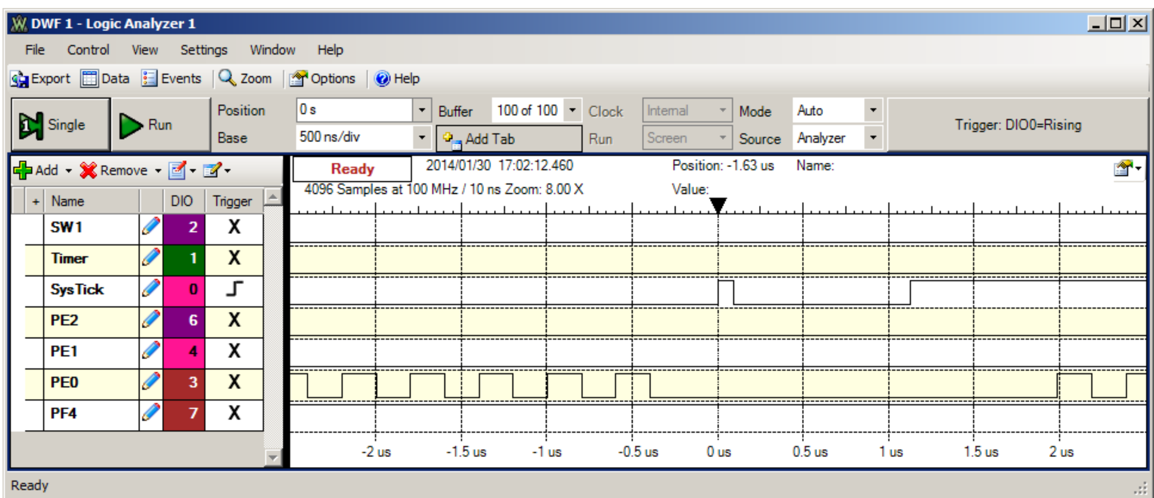

Figure 4: Logic Analyzer Output Showing the Thread-Switch Time

Deliverables for Part 2:

- **Snapshots** of the logic analyzer similar to Figures 1 – 4. They must include measurements needed to calculate the **time slice** and **thread frequency** of the threads for the cooperative (**`Testmain1`**) and preemptive (**`Testmain2`**) scheduler and the **thread-switch time** of the RTOS kernel from **`Testmain0`**. Show your calculations and any other snapshots containing measurements. Note that **thread frequency** is defined as the reciprocal of the time between invocations of a thread by the scheduler.
- **Explain** why the values of the counters will be much higher in **`Testmain2`** than they were in **`Testmain1`**.

Note: SysTick on TM4C123G is not directly connected to a pin. You may ignore SysTick in your submission or toggle a pin multiple times in **`Scheduler`** which is called by the SysTick handler. This will introduce additional overhead to the system.

◆

Part 3) Use the timers provided to you in **`os.c`** to implement the functions **`OS_Time`**, **`OS_TimeDifference`**, **`OS_ClearMsTime`**, and **`OS_MsTime`**.

(**Hint:** Study the provided timers and use the appropriate ones for each routine. Keep in mind that a timer may either count up or down. You may need to modify the timer ISRs).

Deliverables for Part 3: Your code.

Part 4) Test the **`OS_Time`**, **`OS_TimeDifference`**, and **`OS_MsTime`** functions by using them for measuring the thread time slice and context/thread switch time in the **`Testmain1`** and **`Testmain2`** functions in **`MiniProject2Test.c`**. Time slice is the amount of time the scheduler allocates to each thread and is specified by passing a parameter to the **`OS_Launch`** function. You need to figure out where is the right location for calling these functions from the **`Testmain1`** and **`Testmain2`** in order to measure these times. Define the global variables **`TimeSlice`** and **`ContextSwitchTime`** and print their values in the Watch window inside the Keil debugger.

Deliverables for Part 4:

- Your code in **`MiniProject2Test.c`**.
- **Snapshots** measuring the **`TimeSlice`** and **`ContextSwitchTime`** values using **`OS_Time`** and printing them in the Watch window inside the Keil debugger for cooperative (**`Testmain1`**) and preemptive (**`Testmain2`**) scheduler.
- Compare these measurements with the time slice and context switch time you measured using the logic analyzer in Part 2 and answer the following **questions**:
  - Q1: Are there any differences between the values you measured using logic analyzer versus the measurements using **`OS_Time`** functions? Explain why.
  - Q2: Are there any differences between the values you measured in **`Testmain1`** versus **`Testmain2`** function? Explain why.
- Q3: Explain the purpose of using Timer1A, Timer2A, Timer3A, and Timer4A in this mini project.

◆

Mini Project 2 Deliverables and Grading

The following table lists the deliverables and their corresponding points. Commit and push your changes to your private repository on **GitHub** before the submission deadline. Include a PDF report containing all the deliverables (any calculations, snapshots, answers to questions, and links to videos). The first page of your report should include your name and computing ID. You may upload any required videos to a platform such as YouTube or Google Drive. **Note** that the latest commit before the deadline will be considered your submission. Any major changes to your submitted code and report after the deadline and before posting the grades will be considered a late submission.

Survey: After completion of this mini project, please go to the following link and complete the survey: https://virginia.az1.qualtrics.com/jfe/form/SV_eK9D5HpmioNhSQZ. (Copy the link into browser)
This will be anonymous feedback but will be counted towards your class participation. Attach a **snapshot** of the completion page to your project report.

Deliverables	**Points**
1) Your **code** for the following functions:	
• **`OS_AddThread`**	15
• **`OS_Kill`**	10
• **`OS_Suspend`**	10
• **`OS_Time`**	10
• **`OS_TimeDifference`**	10
• **`OS_MsTime`**	2.5
• **`OS_ClearMsTime`**	2.5
• **`MiniProject2Test.c`**	10
2) Deliverables for **Part 2:**	
• **Snapshots** of logic analyzer (similar to Figures 1-4) measuring:	
▪ The **time slice** and **frequency** of the threads in cooperative thread scheduler	15
▪ The **time slice** and **frequency** of the threads in preemptive thread scheduler	15
▪ The **thread-switch time** of the RTOS kernel	15
• Explain why the values of the counters will be much higher in **`Testmain2`** than they were in **`Testmain1`**	5
3) Deliverables for **Part 4:**	
• **Snapshots** measuring the **`TimeSlice`** and **`ContextSwitchTime`**	10
• Question 1	5
• Question 2	5
• Question 3	10
Total	150