

## About Lab 3 [\[edit\]](#)

### Objectives

The objectives of Lab 3 are:

- Rework the semaphores from spinlock to blocking
- Implement period event tasks with dedicated timer interrupt(s)
- Develop a FIFO queue to implement data flow from producer to consumer
- Develop sleeping as a mechanism to recover wasted time
- Design a round-robin scheduler with blocking and sleeping

### Overview [\[edit\]](#)

As we progress through this class, your RTOS will become more and more complex. The Lab 3 starter project using the LaunchPad and the Educational BoosterPack MKII (BOOSTXL-EDUMKII) is again a fitness device. Just like Lab 2, the starter project will not execute until you implement the necessary RTOS functions. Consider reusing code from Lab 2. The user code inputs from the microphone, accelerometer, light sensor, temperature sensor and switches. It performs some simple measurements and calculations of steps, sound intensity, light intensity, and temperature. It outputs data to the LCD and it generates simple beeping sounds. Figure Lab3.1 shows the data flow graph of Lab 3. Your assignment is to first understand the concepts of the chapter in general and the software programs in specific. Your RTOS will run two periodic threads and six main threads. Section 3.1 develops cooperation using **OS\_Suspend**. Section 3.2 develops blocking semaphores. Section 3.3 explains how to implement a first-in-first-out queue. Section 3.4 shows how to implement sleeping. Section 3.5 presents the means to run periodic tasks using the hardware timers.

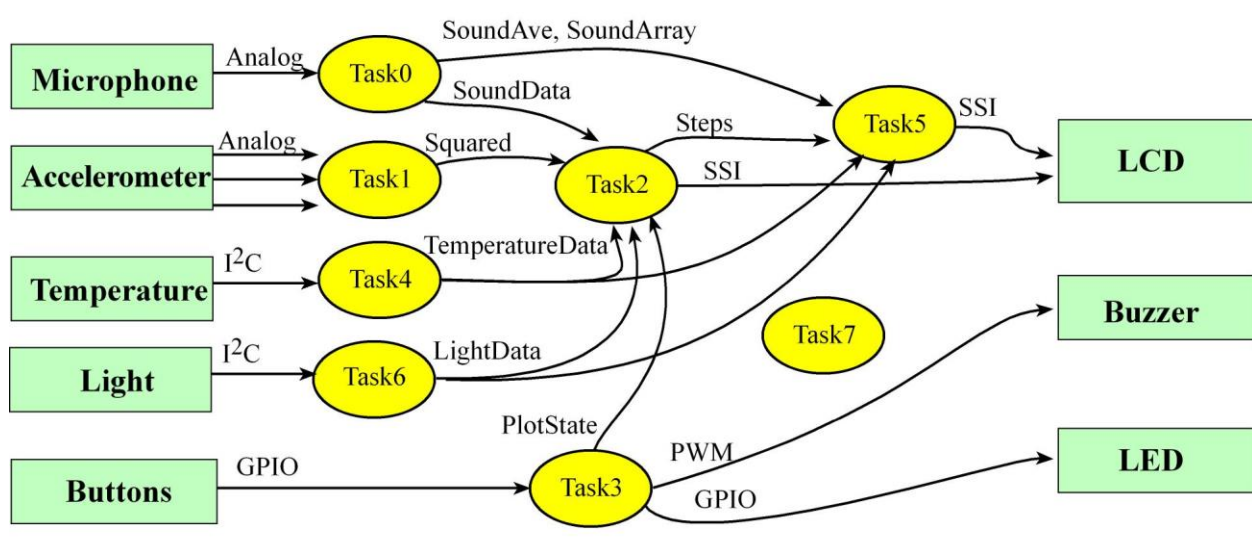


Figure Lab3.1. Data flow graph of Lab 3.

This simple fitness device has eight tasks: two periodic and six main threads. Since you have two periodic threads to schedule, you could use one hardware timer to run both tasks, or you could use two hardware timers, one for each task. You will continue to use SysTick interrupts to switch between the six main threads. These are the eight tasks:

- Task0: event thread samples microphone input at 1000 Hz
- Task1: event thread samples acceleration input at 10 Hz (calls **Put**)
- Task2: main thread detecting steps and plotting at on LCD, runs about 10 Hz (calls **Get**)
- Task3: main thread inputs from switches, outputs to buzzer (calls Sleep)
- Task4: main thread measures temperature, runs about 1 Hz (calls Sleep)
- Task5: main thread output numerical data to LCD, runs about 1 Hz
- Task6: main thread measures light, runs about 1.25 Hz (calls Sleep)
- Task7: main thread that does no work

Thread 7, which doesn't do any useful task, will never sleep or block. Adding this thread will make your RTOS easier to implement because you do not need to handle the case where all main threads are sleeping or blocked.

Your RTOS manages these eight tasks. We will use the same metrics as described as used in Labs 1 and 2, except jitter and error are only relevant for the two real-time event tasks:

$Min_j$  = minimum  $\Delta T_j$  for Task  $j$ ,  $j=0$  to 5

$Max_j$  = maximum  $\Delta T_j$  for Task  $j$ ,  $j=0$  to 5

$Jitter_j$  =  $Max_j - Min_j$  for Task  $j$ ,  $j=0$  to 1

$Ave_j$  = Average  $\Delta T_j$  for Task  $j$ ,  $j=0$  to 5

$Err_j$  =  $100 * (Ave_j - \Delta t_j) / \Delta t_j$  for Task  $j$ ,  $j=0$  to 1

In addition to the above quantitative measures, you will be able to visualize the execution profile of the system using a logic analyzer. Tasks 0 to 6 toggle both the virtual logic analyzer and a real logic analyzer when they start. For example, Task0 calls **TExaS\_Task0()**. The first parameter to the function **TExaS\_Init()** will be **GRADER** or **LOGICANALYZER**. Just like Labs 1 and 2, calling **TExaS\_Task0()** in grader mode performs the lab grading. However in logic analyzer mode, these calls implement the virtual logic analyzer and can be viewed with **TExaSdisplay**. Figure Lab3.2 shows the profile of one possible lab solution measured with the TExaS logic analyzer.

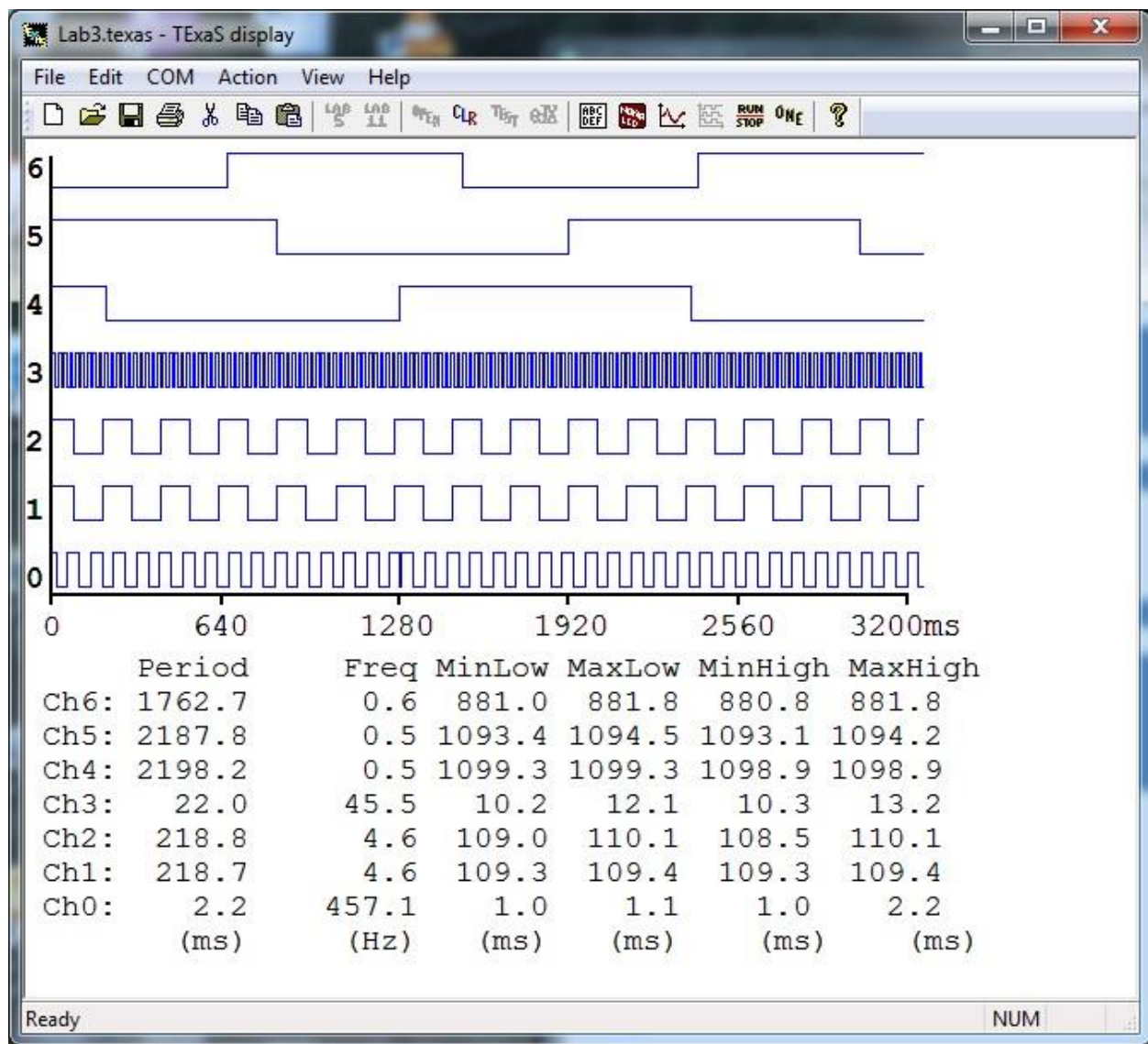


Figure Lab3.2. Task profile measured on a solution for Lab 3 using TExaS (zoom out). Channels 0 and 3 are oscillating too fast to be observed at this time scale.

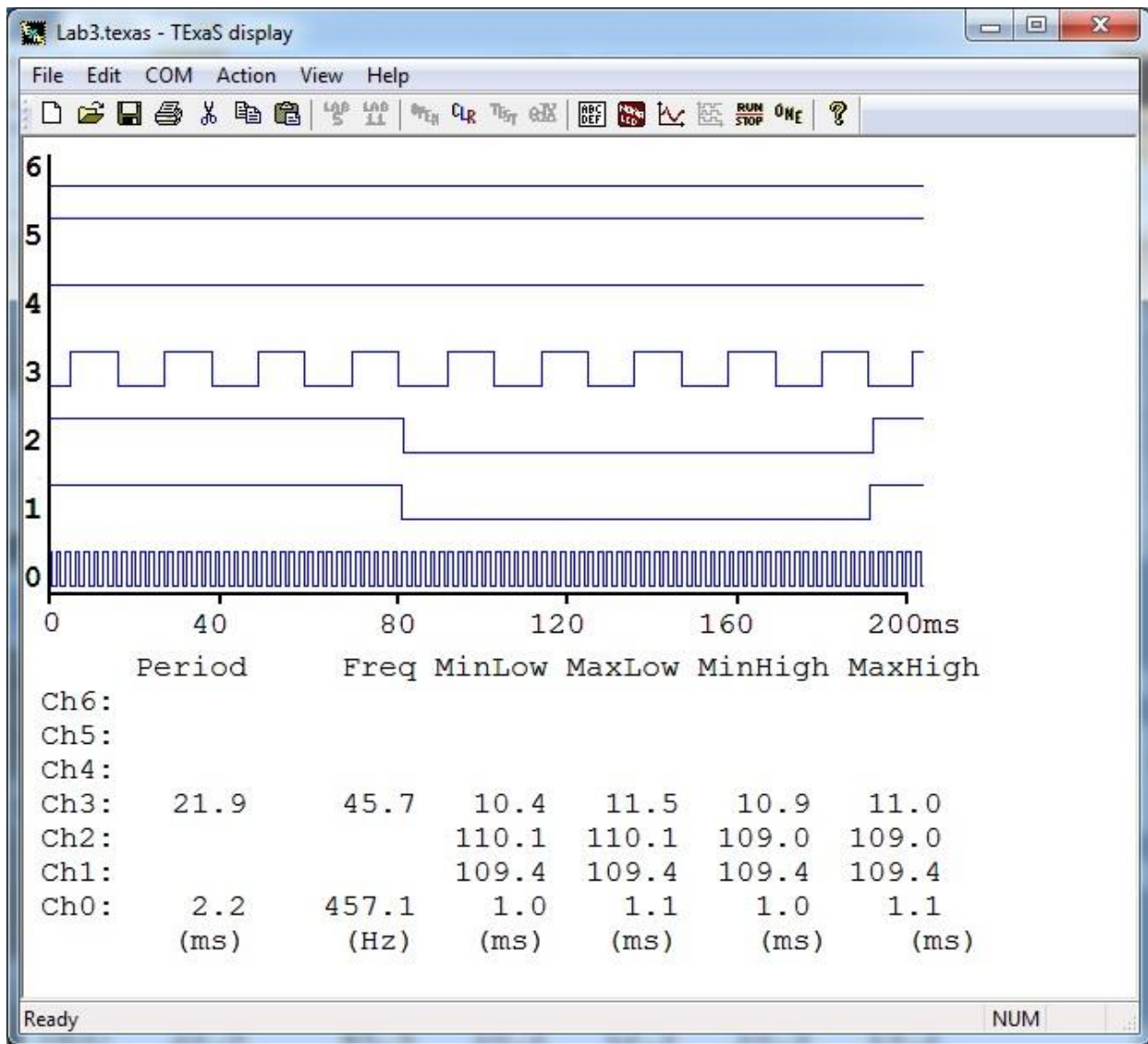


Figure Lab3.3. Task profile measured on a solution for Lab 3 using TExaS (zoom in).

At the start of each task it also toggles an actual pin on the microcontroller. For example, Task0 calls **Profile\_Toggle0()**. You do not need a real logic analyzer, but if you have one, it can be used.

### Specifications [\[edit\]](#)

A real-time system is one that guarantees the jitters are less than a desired threshold, and the averages are close to desired values. Now that we are using hardware timer interrupts we expect the jitter for the two event tasks to be quite low. For the six main threads, you will be graded only on minimum, maximum, and average time between execution of tasks. Your assignment is implement the OS functions in **OS.c** and write the SysTick interrupt service routine in **osasm.s**. We do not expect you to edit the user code in **Lab3.c**, the board support package in **BSP.c**, or

the interface specifications in **profile.h**, **Texas.h**, **BSP.h**, or **OS.h**. More specifically, we are asking you to develop and debug a real-time operating system, such that

- Task0: jitter between executions should be less than or equal to 10 $\mu$ s
- Task1: jitter between executions should be less than or equal to 30 $\mu$ s
- Task2: average time between executions should be 100 ms within 5%
- Task3: average time between executions should be less than 50 ms
- Task4: average time between executions should be less than 1.2 s
- Task5: average time between executions should be 1.0 s within 5%
- Task6: average time between executions should be less than 1.0 s
- Task7: no specifications

## Debugging Lab 3 [\[edit\]](#)

### Approach [\[edit\]](#)

First, we encourage you to open up the project Lab3\_xxx and fully understand the system from the user perspective by reading through **Lab3.c**. Lab3 requires both the LaunchPad and the Educational BoosterPack MKII. (*Although just like Labs 1 and 2, you could do Lab 3 without the MKII*). Next, read through **OS.c** and **OS.h** to learn how your operating system will support the user system. Since this is a class on operating systems, and not personal fitness devices, we do not envision you modifying Lab3.c at all. Rather you are asked to implement the RTOS by writing code in the **osasm.s** and **OS.c** files. To activate the logic analyzer, initialize TExaS with

**TExaS\_Init(LOGICANALYZER,1000);**

Do not worry about the number 1000; you will fill in a valid number once you are done with Lab 3. To activate the grader, initialize TExaS with

**TExaS\_Init(GRADER,1000);**

When you run the starter code in grading mode, you should see this output on TExaSdisplay. Note the numbers on the MSP432 running at 48 MHz will be slightly different than the numbers generated by the TM4C123 running at 80 MHz.

Before you begin editing, and debugging, we encourage you to open up os.c from Lab 2 and the os.c for Lab 3 and copy C code from Lab 2 to Lab 3 (do not move the entire file, just some C functions). Similarly, copy the SysTick ISR from Lab 2 **osasm.s** to your Lab 3 **osasm.s**. The Lab 2 SysTick ISR should be sufficient for Lab 3.

Step 1) Implement the three blocking semaphore functions as defined in **OS.c** and **OS.h**. For more information on semaphores review Section 3.2. You should use this simple main program to initially test the semaphore functions. Notice that this function will not block.

```
int32_t s1,s2;
int main(void){
    OS_InitSemaphore(&s1, 0);
    OS_InitSemaphore(&s2, 1);
```

```

while(1){
    OS_Wait(&s2); // now s1=0, s2=0
    OS_Signal(&s1); // now s1=1, s2=0
    OS_Signal(&s2); // now s1=1, s2=1
    OS_Signal(&s1); // now s1=2, s2=1
    OS_Wait(&s1); // now s1=1, s2=1
    OS_Wait(&s1); // now s1=0, s2=1
}
}

```

## Lab 3 Step2

Play Video

Step 2) Extend your **OS\_AddThreads** from Lab 2 to handle six main threads, add a blocked field to the TCB, and rewrite the scheduler to handle blocking. In this step you will test the blocking feature of your OS using the following user code. TaskA, TaskC, and TaskE are producers running every 6, 60, and 600 ms respectively. Each producer thread signals a semaphore. TaskB, TaskD, and TaskF are consumers that should run after their respective producer. Observe the profile on the TExaS logic analyzer. To activate the grader, initialize TExaS with

**TExaS\_Init(GRADESTEP2,1000);**

The output will look like this before blocking is implemented. Note that TaskB, TaskD, and TaskF are running constantly, and TaskA, TaskC, and TaskE are running half as frequently as expected. This is because the **BSP\_Delay1ms** function implements delay simply by decrementing a counter. When a thread is not running, its counter is not being decremented although time is still passing. Therefore, the actual amount of delay is the parameter of **BSP\_Delay1ms** multiplied by the number of running (unblocked) threads, which should be three once blocking is implemented. In other words, the delay parameters 2, 20, and 200 are consistent with about a 6, 60, and 600 ms delay for a system with three running threads and three blocked threads.

```

**Start Lab 3 Step 2 Test**TM4C123 Version 1.00**
**Done**
TaskA: Expected= 6000, min= 6987, max= 11989, jitter= 5002, ave= 11937 usec, error= 98.9%
TaskB: Expected= 6000, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 99.9%
TaskC: Expected= 60000, min= 114860, max= 119865, jitter= 5005, ave= 119200 usec, error= 98.6%
TaskD: Expected= 60000, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 99.9%
TaskE: Expected= 600000, min= 1188612, max= 1193616, jitter= 5004, ave= 1191739 usec, error= 98.6%
TaskF: Expected= 600000, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 99.9%

```

The output will look like this after blocking is implemented. Look for low error percentages and for each task pair's average execution periods to be similar.

```

**Start Lab 3 Step 2 Test**TM4C123 Version 1.00**
**Done**
TaskA: Expected= 6000, min= 3991, max= 5996, jitter= 2005, ave= 5972 usec, error= 0.4%
TaskB: Expected= 6000, min= 3003, max= 6007, jitter= 3004, ave= 5973 usec, error= 0.4%
TaskC: Expected= 60000, min= 57898, max= 59908, jitter= 2010, ave= 59637 usec, error= 0.6%
TaskD: Expected= 60000, min= 57031, max= 60040, jitter= 3009, ave= 59636 usec, error= 0.6%
TaskE: Expected= 600000, min= 595018, max= 597025, jitter= 2007, ave= 596522 usec, error= 0.5%
TaskF: Expected= 600000, min= 594339, max= 597344, jitter= 3005, ave= 596592 usec, error= 0.5%

```



The test should complete in about 5 seconds. The automatic grader is looking for at least 100 calls to **TExaS\_Task0**, **TExaS\_Task1**, **TExaS\_Task2**, and **TExaS\_Task3** and for at least 10 calls to **TExaS\_Task4** and **TExaS\_Task5**. If the numbers keep counting for more than about 30 seconds, the test may never complete. The most likely explanation is that you did not extend your **OS\_AddThreads** function from Lab 2 to handle all six main threads or there is a problem with your thread scheduler. The TExaS logic analyzer or an oscilloscope on the Profile pins may give more insight about which threads are running and when.

```
int32_t sAB,sCD,sEF;
int32_t CountA,CountB,CountC,CountD,CountE,CountF;
void TaskA(void){ // producer
    CountA = 0;
    while(1){
        CountA++;
        TExaS_Task0();
        Profile_Toggle0();
        OS_Signal(&sAB); // TaskB can proceed
        BSP_Delay1ms(2);
    }
}
void TaskB(void){ // consumer
    CountB = 0;
    while(1){
        CountB++;
        OS_Wait(&sAB); // signaled by TaskA
        TExaS_Task1();
        Profile_Toggle1();
    }
}
void TaskC(void){ // producer
    CountC = 0;
    while(1){
        CountC++;
        TExaS_Task2();
        Profile_Toggle2();
        OS_Signal(&sCD); // TaskD can proceed
        BSP_Delay1ms(20);
    }
}
void TaskD(void){ // consumer
    CountD = 0;
    while(1){
        CountD++;
        OS_Wait(&sCD); // signaled by TaskC
        TExaS_Task3();
        Profile_Toggle3();
    }
}
```

```

void TaskE(void){ // producer
    CountE = 0;
    while(1){
        CountE++;
        TExaS_Task4();
        Profile_Toggle4();
        OS_Signal(&sEF); // TaskF can proceed
        BSP_Delay1ms(200);
    }
}

void TaskF(void){ // consumer
    CountF = 0;
    while(1){
        CountF++;
        OS_Wait(&sEF); // signaled by TaskE
        TExaS_Task5();
        Profile_Toggle5();
    }
}

int main(void){
    OS_Init();
    Profile_Init(); // initialize the 7 hardware profiling pins
    OS_InitSemaphore(&sAB, 0);
    OS_InitSemaphore(&sCD, 0);
    OS_InitSemaphore(&sEF, 0);
    OS_AddThreads(&TaskA, &TaskB, &TaskC, &TaskD, &TaskE, &TaskF);
    // TExaS_Init(LOGICANALYZER, 1000); // initialize the Lab 3 grader
    TExaS_Init(GRADESTEP2, 1000); // initialize the Lab 3 grader
    OS_Launch(BSP_Clock_GetFreq()/1000);
    return 0; // this never executes
}

```

### Lab3 step3

[Play Video](#)

Step 3) Implement the three FIFO queue functions as defined in **OS.c** and **OS.h**. In this step you will test the FIFO using this user code. **TaskG** is a producer running every 100 ms. The producer thread puts 1 to 5 elements into the FIFO. **TaskH** is a consumer that should accept each data from the producer. The other tasks are dummy tasks not related to the FIFO test. Notice the data is a simple sequence so we can tell if data is lost. The data variables are made global so you can place them in the watch window of the debugger. Observe the profile on the TExaS logic analyzer. To activate the grader, initialize TExaS with

**TExaS\_Init(GRADESTEP3,1000);**

The output will look like this before blocking is implemented in **OS\_FIFO\_Get**. Note



that **TaskH** is running constantly instead of blocking while waiting for data from **TaskG** in the FIFO.

```

**Start Lab 3 Step 3 Test**TM4C123 Version 1.00** 01234567890
**Done**
TaskG: Expected= 100000, min= 114861, max= 119870, jitter= 5009, ave= 119201 usec, error= 19.2%
TaskH: Expected= 45667, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 99.9%
TaskI: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%
TaskJ: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%
TaskK: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%
TaskL: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%

```






The output will look like this after the FIFOs and blocking are implemented. Look for **TaskH** to have a maximum execution period that is similar to that of **TaskG**. **TaskH** will still have a very small minimum and average execution period since 80% of the time it gets two to five elements in quick succession before the FIFO is empty and it blocks until **TaskG** runs again.




```

**Start Lab 3 Step 3 Test**TM4C123 Version 1.00** 012345678
**Done**
TaskG: Expected= 100000, min= 95869, max= 99890, jitter= 4021, ave= 99348 usec, error= 0.6%
TaskH: Expected= 45667, min= 4, max= 100006, jitter= 100002, ave= 38525 usec, error= 15.6%
TaskI: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%
TaskJ: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%
TaskK: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%
TaskL: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%

```

When observing the global variables in the debugger, expect to see no lost data. The variable **LostData** is from the implementation of the function **OS\_FIFO\_Put**. See Program 3.7.

Watch 1		
Name	Value	Type
 TaskGdata	16421	int
 TaskHexpected	16421	int
 TaskHactual	16420	int
 TaskHLostData	0	int
 LostData	0	unsigned int
<Enter expression>		

 Call Stack + Locals
  Watch 1
  Memory 1

```

int32_t TaskGdata;
void TaskG(void){ // producer
    TaskGdata=0;
    while(1){ int i; int num = (TaskGdata%5)+1;

```

```

TExaS_Task0();
Profile_Toggle0();
for(i=0; i<num; i++){
    OS_FIFO_Put(TaskGdata); // TaskH can proceed
    TaskGdata++;
}
BSP_Delay1ms(20);
}
}
int32_t TaskHexpected,TaskHactual,TaskHLostData;
void TaskH(void){ // consumer
    TaskHexpected = 0;
    TaskHLostData = 0;
    while(1){
        TaskHactual = OS_FIFO_Get(); // signaled by TaskG
        if(TaskHactual != TaskHexpected){
            TaskHLostData++;
            TaskHexpected = TaskHactual;
        }else{
            TaskHexpected++;
        }
        TExaS_Task1();
        Profile_Toggle1();
    }
}
int32_t CountI;
void TaskI(void){ // dummy
    CountI = 0;
    while(1){
        CountI++;
        TExaS_Task2();
        Profile_Toggle2();
    }
}
int32_t CountJ;
void TaskJ(void){ // dummy
    CountJ = 0;
    while(1){
        CountJ++;
        TExaS_Task3();
        Profile_Toggle3();
    }
}
int32_t CountK;
void TaskK(void){ // dummy
    CountK = 0;
    while(1){
        CountK++;
        TExaS_Task4();

```

```

    Profile_Toggle4();
}
}
int32_t CountL;
void TaskL(void){ // dummy
    CountL = 0;
    while(1){
        CountL++;
        TExaS_Task5();
        Profile_Toggle5();
    }
}
int main(void){
    OS_Init();
    Profile_Init(); // initialize the 7 hardware profiling pins
    OS_FIFO_Init();
    OS_AddThreads(&TaskG, &TaskH, &TaskI, &TaskJ, &TaskK, &TaskL);
    // TExaS_Init(LOGICANALYZER, 1000); // initialize the Lab 3 logic analyzer
    TExaS_Init(GRADESTEP3, 1000); // initialize the Lab 3 grader
    OS_Launch(BSP_Clock_GetFreq()/1000);
    return 0; // this never executes
}

```

#### Lab 3 step 4

Step 4) Implement sleeping as defined in OS.c and OS.h. You will need to add a Sleep parameter to the TCB, and check the scheduler to skip sleeping threads. Do not use SysTick to count down the sleeping threads; rather use one of the hardware timers included in the board support package. The data variables are made global so you can place them in the watch window of the debugger. Observe the profile on the TExaS logic analyzer. To activate the grader, initialize TExaS with

**TExaS\_Init(GRADESTEP4,1000);**









The output will look like this after sleeping is implemented. Look for low error percentages for the tasks that sleep: **TaskM**, **TaskO**, **TaskP**, and **TaskQ**.

```

**Start Lab 3 Step 4 Test**TM4C123 Version 1.00** 01
**Done**
TaskM: Expected= 10000, min= 9029, max= 10031, jitter= 1002, ave= 10008 usec, error= 0.0%
TaskN: Expected= 4567, min= 2, max= 10021, jitter= 10019, ave= 3884 usec, error= 14.9%
TaskO: Expected= 20000, min= 19030, max= 20046, jitter= 1016, ave= 20016 usec, error= 0.0%
TaskP: Expected= 30000, min= 29050, max= 30065, jitter= 1015, ave= 30013 usec, error= 0.0%
TaskQ: Expected= 50000, min= 50087, max= 50103, jitter= 16, ave= 50093 usec, error= 0.1%
TaskR: Expected= 1, min= 1, max= 2, jitter= 1, ave= 1 usec, error= 0.0%

```

When observing the global variables in the debugger, expect to see **CountO**, **CountP**, and **CountQ** in the proper ratios. In other words, multiplying **CountO** by 20, **CountP** by 30, and **CountQ** by 50 should all yield approximately the same number. Of course, **TaskNLostData** should still be zero, which is a repeat of the FIFO test in Step 3.

Watch 1			
Name	Value	Type	
 TaskMdata	48761	int	
 TaskNexpected	48761	int	
 TaskNactual	48760	int	
 TaskNLostData	0	int	
 CountO	9756	int	
 CountP	6505	int	
 CountQ	3903	int	
< Enter expression >			
<div>  Call Stack + Locals           <div> <div>Watch 1</div> <div>Memory 1</div> </div> </div>			

```

int32_t TaskMdata;
void TaskM(void){ // producer
    TaskMdata=0;
    while(1){ int i; int num = (TaskMdata%5)+1;
        TExaS_Task0();
        Profile_Toggle0();
        for(i=0; i<num; i++){
            OS_FIFO_Put(TaskMdata); // TaskN can proceed
            TaskMdata++;
        }
        OS_Sleep(10);
    }
}
int32_t TaskNexpected, TaskNactual, TaskNLostData;
void TaskN(void){ // consumer
    TaskNexpected = 0;
    TaskNLostData = 0;
    while(1){
        TaskNactual = OS_FIFO_Get(); // signaled by Task M
        if(TaskNactual!= TaskNexpected){
            TaskNLostData++;
            TaskNexpected = TaskNactual;
        }else{
            TaskNexpected++;
        }
        TExaS_Task1();
        Profile_Toggle1();
    }
}

```

```

}
int32_t CountO;
void TaskO(void){ // sleeping 20 ms
    CountO = 0;
    while(1){
        CountO++;
        TExaS_Task2();
        Profile_Toggle2();
        OS_Sleep(20);
    }
}
int32_t CountP;
void TaskP(void){ // sleeping 30 ms
    CountP = 0;
    while(1){
        CountP++;
        TExaS_Task3();
        Profile_Toggle3();
        OS_Sleep(30);
    }
}
int32_t CountQ;
void TaskQ(void){ // sleeping 50 ms
    CountQ = 0;
    while(1){
        CountQ++;
        TExaS_Task4();
        Profile_Toggle4();
        OS_Sleep(50);
    }
}
int32_t CountR;
void TaskR(void){ // dummy
    CountR = 0;
    while(1){
        CountR++;
        TExaS_Task5();
        Profile_Toggle5();
    }
}
int main(void){
    OS_Init();
    Profile_Init(); // initialize the 7 hardware profiling pins
    OS_FIFO_Init();
    OS_AddThreads(&TaskM, &TaskN, &TaskO, &TaskP, &TaskQ, &TaskR);
    // TExaS_Init(LOGICANALYZER, 1000); // initialize the Lab 3 logic analyzer
    TExaS_Init(GRADESTEP4, 1000); // initialize the Lab 3 grader
    OS_Launch(BSP_Clock_GetFreq()/1000);
}

```

```

    return 0;           // this never executes
}

```

## Lab 3 step 5

Play Video

Step 5) Implement the two periodic event threads as defined in OS.c and OS.h. You could use the same hardware timer interrupt as you used for sleeping or you could use additional hardware interrupts. Do not use SysTick to run periodic event threads. The data variables are made global so you can place them in the watch window of the debugger. Observe the profile on the TExaS logic analyzer. To activate the grader, initialize TExaS with

**TExaS\_Init(GRADESTEP5,1000);**

The output will look like this after the periodic event threads are implemented. Look for **TaskU** and **TaskV** to have similar average execution periods, since they are linked by a semaphore. Look for low error percentages for the tasks that sleep: **TaskW**, **TaskX**, and **TaskY**. Finally, verify that the new periodic event threads **TaskS** and **TaskU** have average execution periods of exactly 10,000  $\mu$ sec and 100,000  $\mu$ sec, respectively. **TaskS** and **TaskU** should have jitter less than or equal to 18  $\mu$ sec. **Jitter** that is small and bounded is a requirement of a real-time system.

```

**Start Lab 3 Step 5 Test**TM4C123 Version 1.00** 012345678
**Done**
TaskS: Expected= 10000, min= 10000, max= 10000, jitter= 0, ave= 10000 usec, error= 0.0%
TaskT: Expected= 4567, min= 2, max= 10014, jitter= 10012, ave= 3881 usec, error= 15.0%
TaskU: Expected= 100000, min= 99999, max= 100001, jitter= 2, ave= 100000 usec, error= 0.0%
TaskV: Expected= 100000, min= 99105, max= 100117, jitter= 1012, ave= 100003 usec, error= 0.0%
TaskW: Expected= 30000, min= 30028, max= 30040, jitter= 12, ave= 30034 usec, error= 0.1%
TaskX: Expected= 40000, min= 40043, max= 40051, jitter= 8, ave= 40045 usec, error= 0.1%
TaskY: Expected= 50000, min= 50050, max= 50064, jitter= 14, ave= 50056 usec, error= 0.1%

```

```

int32_t TaskSdata=0;
void TaskS(void){ // producer as event thread every 10 ms
    int i; int num = (TaskSdata%5)+1;
    TExaS_Task0();
    Profile_Toggle0();
    for(i=0; i<num; i++){
        OS_FIFO_Put(TaskSdata); // TaskT can proceed
        TaskSdata++;
    }
}

int32_t TaskTexpected, TaskTactual, TaskTlostData;
void TaskT(void){ // consumer as main thread
    TaskTexpected = 0;
    TaskTlostData = 0;
    while(1){
        TaskTactual = OS_FIFO_Get(); // signaled by Task S
        if(TaskTactual!= TaskTexpected){
            TaskTlostData++;
        }
    }
}

```

```

        TaskTexpected = TaskTactual;
    }else{
        TaskTexpected++;
    }
    TExaS_Task1();
    Profile_Toggle1();
}
}
int32_t CountU=0;
int32_t sUV;
void TaskU(void){ // event thread every 100 ms
    CountU++;
    TExaS_Task2();
    Profile_Toggle2();
    OS_Signal(&sUV);
}
int32_t CountV;
void TaskV(void){ // connected to TaskU
    CountV = 0;
    while(1){
        OS_Wait(&sUV);
        CountV++;
        TExaS_Task3();
        Profile_Toggle3();
    }
}
int32_t CountW;
void TaskW(void){ // sleeping 30 ms
    CountW = 0;
    while(1){
        CountW++;
        TExaS_Task4();
        Profile_Toggle4();
        OS_Sleep(30);
    }
}
int32_t CountX;
void TaskX(void){ // sleeping 40 ms
    CountX = 0;
    while(1){
        CountX++;
        TExaS_Task5();
        Profile_Toggle5();
        OS_Sleep(40);
    }
}
int32_t CountY;
void TaskY(void){ // sleeping 50 ms
    CountY = 0;

```



```

while(1){
    CountY++;
    TExaS_Task6();
    Profile_Toggle6();
    OS_Sleep(50);
}
}
int32_t CountZ;
void TaskZ(void){ // dummy
    CountZ = 0;
    while(1){
        CountZ++;
    }
}
int main(void){
    OS_Init();
    Profile_Init(); // initialize the 7 hardware profiling pins
    OS_FIFO_Init();
    OS_InitSemaphore(&sUV, 0);
    OS_AddThreads(&TaskT, &TaskV, &TaskW, &TaskX, &TaskY, &TaskZ);
    OS_AddPeriodicEventThread(&TaskS, 10);
    OS_AddPeriodicEventThread(&TaskU, 100);
    // TExaS_Init(LOGICANALYZER, 1000); // initialize logic analyzer
    TExaS_Init(GRADESTEP5, 1000); // initialize the Lab 3 grader
    OS_Launch(BSP_Clock_GetFreq()/1000);
    return 0; // this never executes
}

```

## Lab 3 step 6

Play Video

Step 6) Debug your Lab3 using debugging windows and the TExaS logic analyzer. You should hear the buzzer when you press a switch. You should be able to see seven of the eight tasks running on the TExaS logic analyzer, and you should be able to see global variables **PlotState** change with buttons, see **TemperatureData** set by Task4, and see **LightData** set by Task6. To activate the grader, initialize TExaS with

**TExaS\_Init(GRADER,1000);**

Remember to change the second parameter to your 4-digit number in order to get credit for this lab. The output will look like this after the periodic event threads are implemented.

```

**Start Lab 3 Grader**TM4C123 Version 1.00** 012345678
**Done**
Task0: Expected=    1000, min=    1000, max=    1000, jitter=    0, ave=    1000 usec, error= 0.0%
Task1: Expected=  100000, min=   99998, max=  100002, jitter=    4, ave=  100000 usec, error= 0.0%
Task2: Expected=  100000, min=   17754, max=  101781,      ave=   99164 usec, error= 0.8%
Task3:              min=    9750, max=    11406,      ave=   10021 usec
Task4:              min= 1003371, max= 1005158,      ave=  1003813 usec
Task5: Expected= 1000000, min=   999398, max= 1000874,      ave= 1000022 usec, error= 0.0%
Task6:              min=   820806, max=   821679,      ave=   821117 usec
Grade= 100
edX code= koCieGbm

```

Lab 3 real logic analyzer