Lab 14
Submission February 07, 2021
Points 100

In this lab 14, you will 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, and Design a round-robin scheduler with blocking and sleeping. You need to do some changes on your Lab 13. In this lab you need to edit the code in Lab3_4C123.

The Lab 14 starter project using the LaunchPad and the Educational BoosterPack MKII (BOOSTXL-EDUMKII) is again a fitness device. Just like Lab 13, the starter project will not execute until you implement the necessary RTOS functions. Consider reusing codes from Lab 13. 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 Lab14.1 shows the data flow graph of Lab 14.

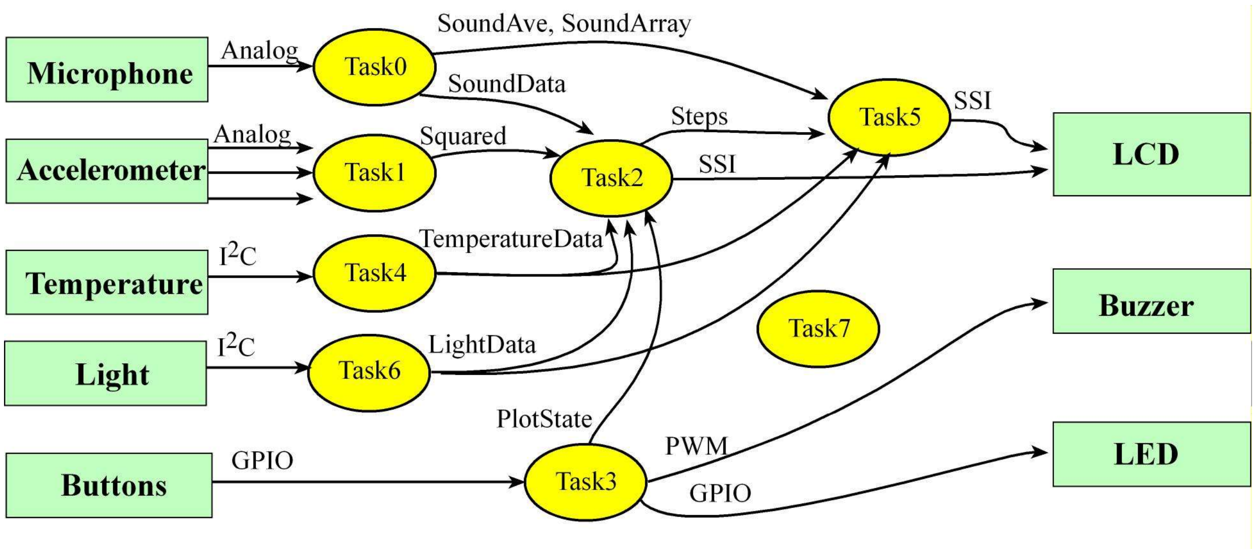


Figure 14.1: Data flow graph of Lab 14.

Your assignment is to understand OS_Suspend, blocking semaphores, first-in-first-out queue., sleeping mechanism and runing periodic tasks using the hardware timers. Your RTOS will run two periodic threads and six main threads. 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. Task7/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 can define the real-time performance of this manager by measuring the time between execution of tasks. For example, the grader will measure when Task0 is started for the first n times it is run, T0,i, for i=0 to n-1. From these measurements we calculate the time difference between starts. ΔT0,i = T0,i - T0,i-1, for i = 1 to n-1. Each task has a desired rate. Let Δtj be the desired time between executions for task j. The grader will generate these performance measures for each for Task j, j=0 to 5:

Minj = minimum ΔTj for Task j, j=0 to 5
Maxj = maximum ΔTj for Task j, j=0 to 5
Jitterj = Maxj - Minj for Task j, j=0 to 1
Avej = Average ΔTj for Task j, j=0 to 5
Errj = 100*(Avej - Δtj)/Δtj for Task j, j=0 to 1

Submission: You need to submit the screenshot of your GRADER and LOGICANALYZER from TExaSdisplay. If you don't have a working LCD please borrow from your friend who has a working one. Please submit the os.c, os.h and osasm.s file with your report.

Notes: You do not need to modify Lab3.c, the board support package in BSP.c, or the interface specifications in profile.h, Texas.h, BSP.h, or OS.h. Rather you need to implement the RTOS by writing code in the osasm.s and OS.c files as Lab 13.

Bug: There is a known bug in the given codes. This bug only applies to the BSPnotMKII.c on the TM4C123. In the BSPnotMKII.c, file, line 2,396 (the first instruction of BSP_Time_Init()), please add the line

```
  sr = StartCritical();
```

Also in Lab3.c file comment out line 890: //const unsigned short title2[] = {

Theories and procedures:

This Lab is fairly complicated. Therefore it has been divided into 6-steps. In these steps you need to implement a portion in OS.c and check the result with debugging main() function specially written to debug that edited portion. This main() function should replace the original main() in Lab3.c file.

STEP0:

Before you begin editing, and debugging, you should open up os.c from Lab 13 and the os.c for Lab 3 and copy C code from Lab 13 to Lab 3 (do not move the entire file, just some C functions as void SetInitialStack(int i), OS_AddThreads(……)). Similarly, copy the required portion from SysTick_Handler from Lab 13 osasm.s to your Lab 3 (our Lab 14) osasm.s. The Lab 13 SysTick_Handler should be sufficient for Lab 3. It is true for StartOS also.

STEP1:

Implement the three blocking semaphore functions (`OS_InitSemaphore, OS_Wait, OS_Signal`) as requested in OS.c and OS.h as Lab2.c. You should use this simple main program to initially test the semaphore functions. Notice that this function will not block. There is a function called called int main_step1(void) in Lab3.c. Change this one to main(vaid) and change the actual main(void) to main_step1(void) for testing purposes. If it runs without error you are in right track. Now return the main() to main_step1(void) and main_step1(void) to main().

```
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
  }
}
```

STEP2:

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 (it is already in the Lab3.c code)

```
TExaS_Init(GRADESTEP2,1000);
```

STEP2 Theories and Implementation:

The basic idea of a blocking semaphore is to prevent a thread from running when the thread needs a resource that is unavailable. We classify these threads as blocked. There are three reasons to replace spin-lock semaphores with blocking semaphores. The first reason is an obvious inefficiency in having threads spin while there is nothing for them to do. Blocking semaphores will be a means to recapture this lost processing time. Essentially, with blocking semaphores, a thread will not run unless it has useful work it can accomplish. Even with spinlock/cooperation it is wasteful to launch a thread you know can't run, only to suspend itself 10 µs later. Spinlock semaphores with cooperation will become more inefficient as the number of threads increase. Spinlock semaphores with cooperation are practical for systems that have less than 10 threads, but if there are 50 threads and 47 of them are spinning, you will be wasting a lot of time trying to find the 3 threads that are active.

The second problem with spin-lock semaphores is a fairness issue. Consider the case with threads 1 2 3 running in round robin order. Assume thread 1 is the one calling Signal, and threads 2 and 3 call Wait. If threads 2 and 3 are both spinning waiting on the semaphore, and then thread 1 signals the semaphore, which thread (2 or 3) will be allowed to run? Because of its position in the 1 2 3 cycle, thread 2 will always capture the semaphore ahead of thread 3. It seems fair when the status of a resource goes from busy to available, that all threads waiting on the resource should get equal chance.

We define bounded waiting as the condition where once a thread begins to wait on a resource (the call to OS_Wait does not return right away), there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed. Bounded waiting does not guarantee a minimum time before OS_Wait will return, it just guarantees a finite number of other threads will go before this thread. For example, it is holiday time, and I want to mail a package to my mom. I walk into the post office and take a number, the number on the ticket is 251, I look up at the counter and the display shows 212, and I know there are 39 people ahead of me in line.

The third reason to develop blocking semaphores will be the desire to implement a priority thread scheduler. In Labs 2 and 3, you implemented a round-robin scheduler and assumed each thread had equal importance.

A thread is in the blocked state when it is waiting for some external event like input/output (keyboard input available, printer ready, I/O device available.) We use semaphores to implement communication and synchronization, and it is semaphore function OS_Wait that will block a thread if it needs to wait. For example, if a thread communicates with other threads then it can be blocked waiting for an input message or waiting for another thread to be ready to accept its output message. If a thread wishes to output to the display, but another thread is currently outputting, then it will block. If a thread needs information from a FIFO (calls Get), then it will be blocked if the FIFO is empty (because it cannot retrieve any information.) Also, if a thread outputs information to a FIFO (calls Put), then it will be blocked if the FIFO is full (because it cannot save its information.) The semaphore function OS_Signal will be called when it is appropriate for the blocked thread to continue. For example, if a thread is blocked because it wanted to print and the printer was busy, it will be signaled when the printer is free. If a thread is blocked waiting on a message, it will be signaled when a message is available. Similarly, if a thread is blocked waiting on an empty FIFO, it will be signaled when new data are put into the FIFO. If a thread is blocked because it wanted to put into a FIFO and the FIFO was full, it will be signaled when another thread calls Get, freeing up space in the FIFO.

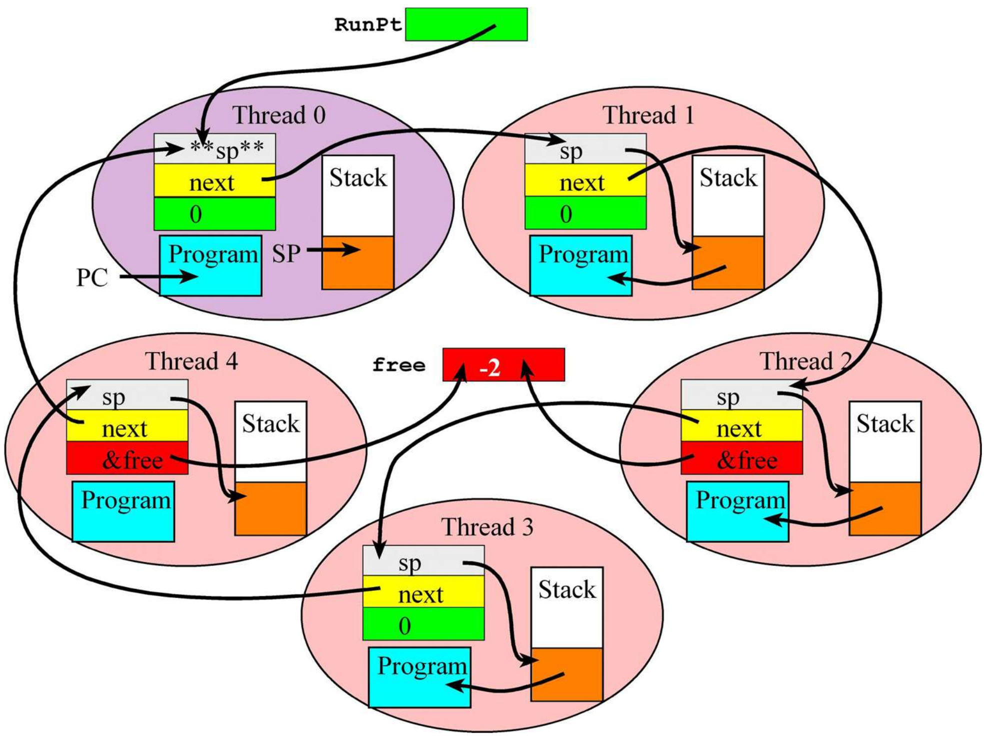


Figure 14.2. Threads 0, 1 and 3 are being run by the scheduler. Threads 2 and 4 are blocked on free and will not run until some thread signals free.

Figure 14.2 shows five threads. In this simple implementation of blocking we add a third field, called blocked, to the TCB structure, defining the status of the thread (previously were int32_t *sp and struct tcb *next. Now add int32_t *blocked). The RunPt points to the TCB of the thread that is currently running. The next field is a pointer chaining all five TCBs into a circular linked list. Each TCB has a StackPt field. Recall that, if the thread is running it is using the real SP for its stack pointer. However, the other threads have their stack pointers saved in this field. The third field is a blocked field. If the blocked field is null, there are no resources preventing the thread from running. On the other hand, if a thread is blocked, the blocked field contains a pointer to the semaphore on which this thread is blocked. In Figure 14.2, we see threads 2 and 4 are blocked waiting for the resource (semaphore free). All five threads are in the circular linked list although only three of them will be run.

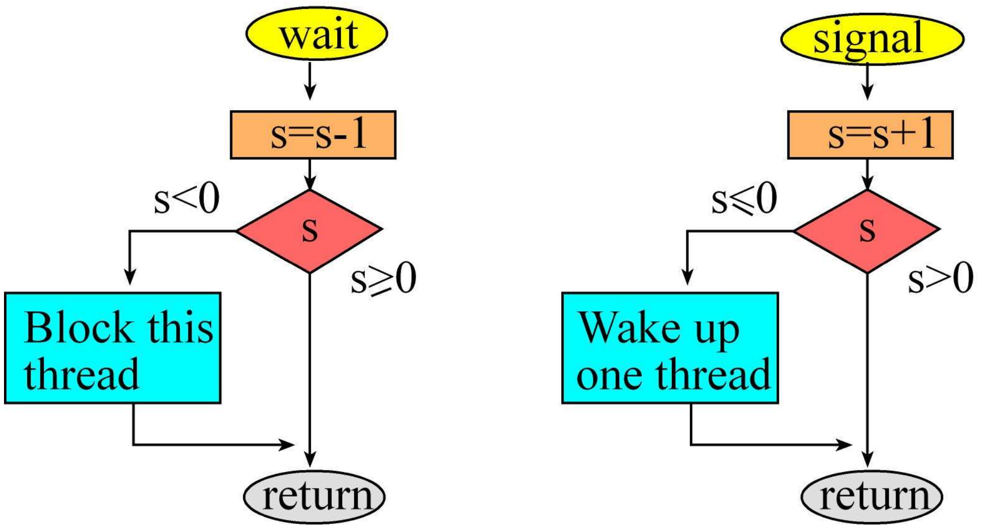


Figure 14.3. Flowcharts of a blocking counting semaphore.

Here is a simple approach for implementing blocking semaphores to use in Lab 14. Notice in Figure 14.3 that wait always decrements and signal always increments. This means the semaphore can become negative. In the example of using a semaphore to implement mutual exclusion, if free is 1, it means the resource is free. If free is 0, it means the resource is being used. If free is -1, it means one thread is using the resource and a second thread is blocked, waiting to use it. If free is -2, it means one thread is using the resource and two other threads are blocked, waiting to use it. In this simple implementation, the semaphore is a signed integer.

This simple implementation of blocking is appropriate for systems with less than 20 threads. In this implementation, a blocked field is added to the TCB. The type of this field is a pointer to a semaphore. The semaphore itself remains a signed integer. If blocked is null, the thread is not blocked. If the blocked field contains a semaphore pointer, it is blocked on that semaphore. The "Block this thread" operation will set the blocked field to point to the semaphore, then suspend the thread.

```
void OS_InitSemaphore(int32_t *s, int32_t v){
     *s = v;
}

void OS_Wait(int32_t *s){
  DisableInterrupts();
   (*s) = (*s) - 1;
  if((*s) < 0){
    RunPt->blocked = s; // reason it is blocked
    EnableInterrupts();
    OS_Suspend();       // run thread switcher
  }
  EnableInterrupts();
}
```

The "Wakeup one thread" operation will be to search all the TCBs for first one that has a blocked field equal to the semaphore and wake it up by setting its blocked field to zero.

```
void OS_Signal(int32_t *s){
  tcbType *pt;
  DisableInterrupts();
  (*s) = (*s) + 1;
  if((*s) <= 0){
    pt = RunPt->next;   // search for a thread blocked on
this semaphore
    while(pt->blocked != s){
      pt = pt->next;
    }
    pt->blocked = 0;    // wakeup this one
  }
  EnableInterrupts();
}
```

Notice in this implementation, calling the signal will not invoke a thread switch. During the thread switch, the OS searches the circular linked-list for a thread with a blocked field equal to zero (the woken up thread is a possible candidate).

```
void Scheduler(void){
  RunPt = RunPt->next;    // run next thread not blocked
  while(RunPt->blocked){  // skip if blocked
    RunPt = RunPt->next;
  }
}
```

Also OS_AddThreads (void(*thread0), ….) include the blocked state in thread definition as,

```
  SetInitialStack(i); Stacks[i][STACKSIZE-2] =
(int32_t)(threadi); tcbs[i].blocked = 0;// PC
```

Sometimes a thread knows it can no longer make progress. If a thread wishes to cooperatively release control of the processor it can call OS_Suspend, which will halt this thread and run another thread. Because all the threads work together to solve a single problem, adding cooperation at strategic places allows the system designer to greatly improve performance. When threads wish to suspend themselves, they call OS_Suspend.

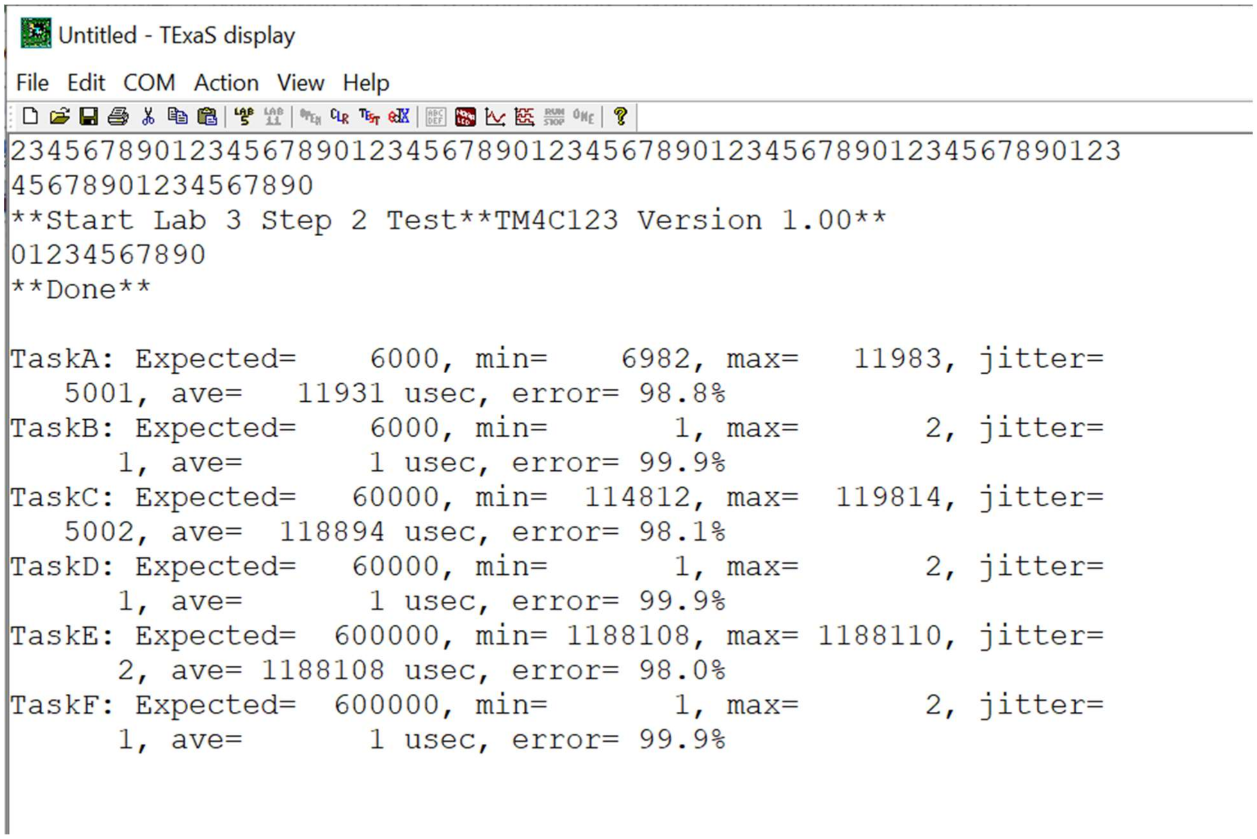
```
void OS_Suspend(void){
  INTCTRL = 0x04000000; // trigger SysTick, but not reset
timer
}
```

STEP2 output

Now you can test your code by this main_step2(void) in Lab3.c. As before you need to make this the only int main(void) function and make the actual one to main_step2(void). This main function look as following:

```
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
}
```

The output will look like Figure 14.4 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.

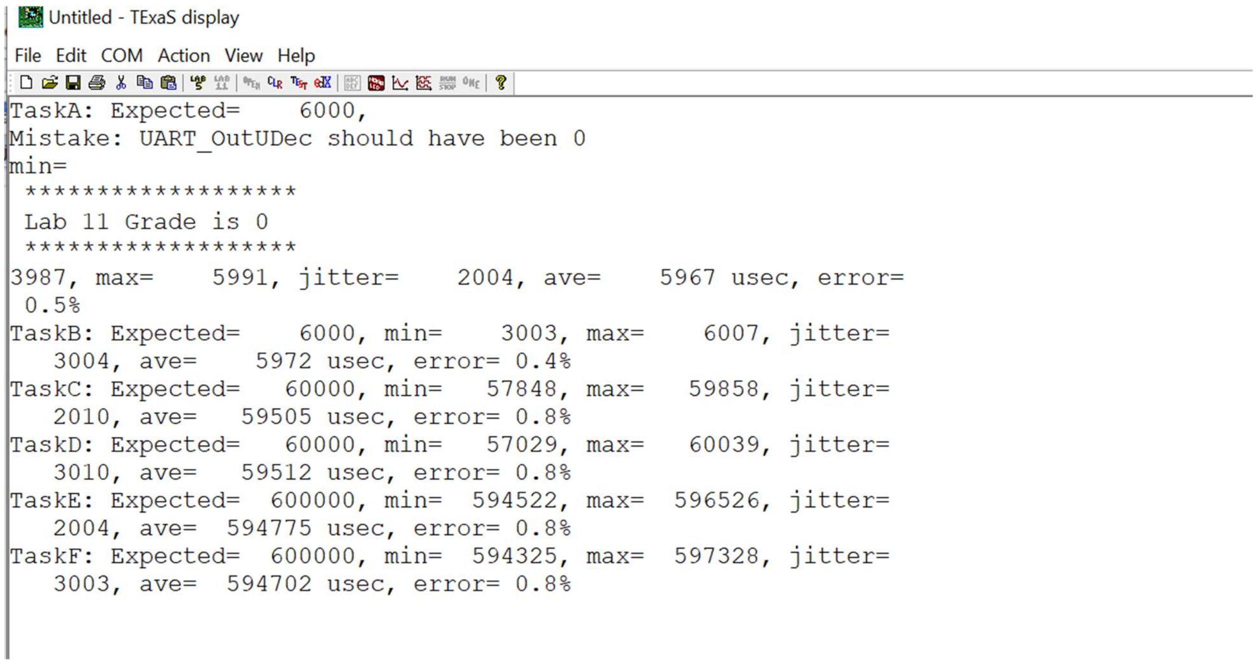


```
23456789012345678901234567890123456789012345678901234567890123
45678901234567890
**Start Lab 3 Step 2 Test**TM4C123 Version 1.00**
01234567890
**Done**

TaskA: Expected=    6000, min=    6982, max=   11983, jitter=
   5001, ave=   11931 usec, error= 98.8%
TaskB: Expected=    6000, min=       1, max=       2, jitter=
      1, ave=       1 usec, error= 99.9%
TaskC: Expected=   60000, min=  114812, max=  119814, jitter=
   5002, ave=  118894 usec, error= 98.1%
TaskD: Expected=   60000, min=       1, max=       2, jitter=
      1, ave=       1 usec, error= 99.9%
TaskE: Expected=  600000, min= 1188108, max= 1188110, jitter=
      2, ave= 1188108 usec, error= 98.0%
TaskF: Expected=  600000, min=       1, max=       2, jitter=
      1, ave=       1 usec, error= 99.9%
```

Figure 14.4. GRADER output before blocking is implemented (no blocking in scheduler, threads, or in semaphores)

The output will look like Figure 14.5 after blocking is implemented. Look for low error percentages and for each task pair's average execution periods to be similar.



```
TaskA: Expected=    6000,
Mistake: UART_OutUDec should have been 0
min=
 *******************
 Lab 11 Grade is 0
 *******************
3987, max=    5991, jitter=    2004, ave=    5967 usec, error=
 0.5%
TaskB: Expected=    6000, min=    3003, max=    6007, jitter=
   3004, ave=    5972 usec, error= 0.4%
TaskC: Expected=   60000, min=   57848, max=   59858, jitter=
   2010, ave=   59505 usec, error= 0.8%
TaskD: Expected=   60000, min=   57029, max=   60039, jitter=
   3010, ave=   59512 usec, error= 0.8%
TaskE: Expected=  600000, min=  594522, max=  596526, jitter=
   2004, ave=  594775 usec, error= 0.8%
TaskF: Expected=  600000, min=  594325, max=  597328, jitter=
   3003, ave=  594702 usec, error= 0.8%
```

Figure 14.5. GRADER output after blocking is implemented (blocking in scheduler, threads, or in semaphores)

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.

STEP 3

Implement the three FIFO queue functions (OS_FIFO_Init, OS_FIFO_Put, OS_FIFO_Get) as defined in OS.c and OS.h. In this step you will test the FIFO using this user code.

STEP3 Theories and Implementation:

A common scenario in operating systems is where producer generates data and a consumer consumes/processes data. To decouple the producer and consumer from having to work in lock-step a buffer is used to store the data, so the producer thread can produce when it runs and as long as there is room in the buffer and the consumer thread can process data when it runs, as long as the buffer is non-empty. A common implementation of such a buffer is a FIFO which preserves the order of data, so that the first piece of data generated in the first consumed.

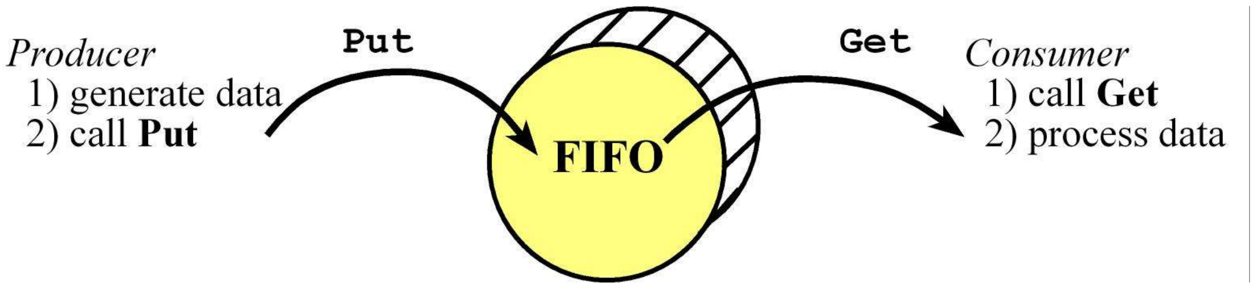


Figure 14.6. The FIFO is used to buffer data between the producer and consumer. The number of data stored in the FIFO varies dynamically, where Put adds one data element and Get removes/returns one data element.

The first in first out circular queue (FIFO) is quite useful for implementing a buffered I/O interface (Figure 14.6). The function Put will store data in the FIFO, and the function Get will remove data. It operates in a first in first out manner, meaning the Get function will return/remove the oldest data. It can be used for both buffered input and buffered output. This order-preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). The class of FIFOs studied in this section will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be carefully shared by more than one program. The advantage of using a FIFO structure for a data flow problem is that we can decouple the producer and consumer threads. Without the FIFO we would have to produce one piece of data, then process it, produce another piece of data, then process it. With the FIFO, the producer thread can continue to produce data without having to wait for the consumer to finish processing the previous data. This decoupling can significantly improve system performance.

For example, a FIFO is used while streaming audio from the Internet. As sound data are received from the Internet they are stored (calls Put) into a FIFO. When the sound board needs data it calls Get. As long as the FIFO never becomes full or empty, the sound is played in a continuous manner. A FIFO is also used when you ask the computer to print a file. Rather than waiting for the actual printing to occur character by character, the print command will put the data in a FIFO. Whenever the printer is free, it will get data from the FIFO. The advantage of the FIFO is it allows you to continue to use your computer while the printing occurs in the background. To implement this magic, our RTOS must be able to manage FIFOs.

FIFOs can be statically allocated, where the buffer size is fixed at compile time. This means the maximum number of elements that can be stored in the FIFO at any one time is determined at design time. Alternately, FIFOs can be dynamically allocated, where the OS allows the buffer to grow and shrink in size dynamically. To allow a buffer to grow and shrink, the system needs a memory manager or heap. A heap allows the system to allocate, deallocate, and reallocate buffers in RAM dynamically. There are many memory managers (heaps), but the usual one available in C has these three functions. The function malloc creates a new buffer of a given size. The function free deallocates a buffer that is no longer needed. The function realloc allocates a new buffer, copies data from a previous buffer into the new buffer of different size, and then deallocates the previous buffer. realloc is the function needed to increase or decrease the allocated space for the FIFO statically-allocated FIFOs might result in lost data or reduced bandwidth compared to dynamic allocation.

There are many ways to implement a statically-allocated FIFO. We can use either two pointers or two indices to access the data in the FIFO. We can either use or not use a counter that specifies how many entries are currently stored in the FIFO. There are even hardware implementations. We will implementations FIFO using semaphores.

We can implement a FIFO based on Figure 14.6. This structure is used to pass data from a single producer to a single consumer. The producer is an event thread and the consumer is a main thread.

```
#define FIFOSIZE 10  // can be any size
uint32_t PutI;       // index of where to put next
uint32_t GetI;       // index of where to get next
uint32_t Fifo[FIFOSIZE];
```

```
int32_t CurrentSize; // 0 means FIFO empty, FIFOSIZE
means full
uint32_t LostData;   // number of lost pieces of data
// initialize FIFO
void OS_FIFO_Init(void){
  PutI = GetI = 0;   // Empty
  OS_InitSemaphore(&CurrentSize, 0);
  LostData = 0;
}
int OS_FIFO_Put(uint32_t data){
  if(CurrentSize == FIFOSIZE){
    LostData++;
    return -1;         // full
  } else{
    Fifo[PutI] = data; // Put
    PutI = (PutI+1)%FIFOSIZE;
    OS_Signal(&CurrentSize);
    return 0;          // success
  }
}
uint32_t OS_FIFO_Get(void){uint32_t data;
  OS_Wait(&CurrentSize);    // block if empty
  data = Fifo[GetI];        // get
  GetI = (GetI+1)%FIFOSIZE; // place to get next
  return data;
}
```

Program 14.1. Two-index one-semaphore implementation of a FIFO. This implementation is appropriate when a single producer is running as an event thread and a single consumer is running as a main thread.

Now you can test your code by this main_step3(void) in Lab3.c. As before you need to make this the only int main(void) function and make the actual one to main_step3(void). 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. This main function look as following:

```
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
}
```

STEP 4

Implement sleeping (OS_Sleep()) 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 (BCP.c). Observe the profile on the TExaS logic analyzer.

STEP4 Theories and Implementation

Sometimes a thread needs to wait for a fixed amount of time. We will implement an OS_Sleep function that will make a thread dormant for a finite time. A thread in the sleep state will not be run. After the prescribed amount of time, the OS will make the thread active again. Sleeping would be used for tasks that are not real-time.

```
void Task(void){
  InitializationStuff();
  while(1){
    PeriodicStuff();
    OS_Sleep(ONE_SECOND); // go to sleep for 1 second
  }
}
```

Program 14.2. This thread uses sleep to execute its task approximately once a second.

```
void OS_Sleep(uint32_t sleepTime){
    RunPt->sleep = sleepTime;
    OS_Suspend;
}
```

Program 14.3. A sleep function

To implement the sleep function, we could add a counter to each TCB and call it Sleep. If Sleep is zero, the thread is not sleeping and can be run, meaning it is either in the run or active state. If Sleep is nonzero, the thread is sleeping. We need to change the scheduler so that RunPt is updated with the next thread to run that is not sleeping and not blocked,

```
void Scheduler(void){
  RunPt = RunPt->next; // skip at least one
  while((RunPt->Sleep)||(RunPt-> blocked)){
    RunPt = RunPt->next; // find one not sleeping and not
blocked
  }
}
```

Program 14.4 Round-robin scheduler that skips threads if they are sleeping or blocked.

In this way, any thread with a nonzero Sleep counter will not be run. The user must be careful not to let all the threads go to sleep, because doing so would crash this implementation. Next, we need to add a periodic task that decrements the Sleep counter for any nonzero counter. When a thread wishes to sleep, it will set its Sleep counter and invoke the cooperative scheduler. The period of this decrementing OS operation will determine the resolution of the TCB field Sleep.

STEP 5

Implement the two periodic event (OS_AddPeriodicEventThread(), runperiodicevents()) 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.

STEP5 Theories and Implementation:

There are 5-basic components to create a periodic interrupt with a timer.

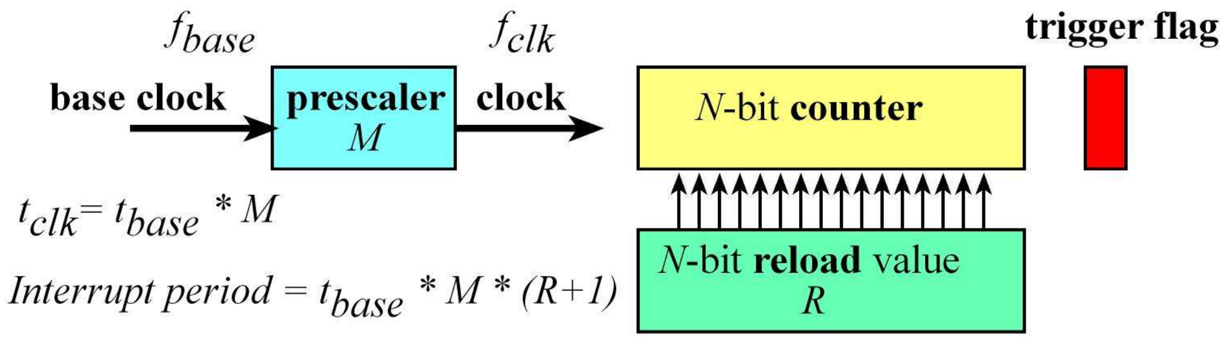


Figure 14.7. Fundamental hardware components used to create periodic interrupts.

The central component for creating periodic interrupts is a hardware counter. The counter may be 16, 24, 32, 48, or 64 bits wide. Let N be the number of bits in the counter. When creating periodic interrupts, it doesn't actually matter if the module counts up or counts down. I think we discussed about counter on last quarter.

Just like SysTick, as the counter counts down to 0, it sets a trigger flag and reloads the counter with a new value. The second component will be the reload value, which is the N-bit value loaded into the counter when it rolls over. Typically the reload value is a constant set once by the software during initialization. Let R be this constant value.

The third component is the trigger flag, which is set when the counter reaches 0. This flag will be armed to request an interrupt. Software in the ISR will execute code to acknowledge or clear this flag.

The fourth component will be the base clock with which we control the entire hardware system. On the TM4C123, we will select the 80-MHz system clock. The clock is derived from the crystal; hence timing will be both accurate and stable. Let f_{base} be the frequency of the base clock (80 MHz) and t_{base} be the period of this clock (12.5 ns).

The fifth component will be a prescaler, which sits between the base clock and the clock used to decrement the counter. Most systems create the prescaler using a modulo-M counter, where M is greater than or equal to 1. This way, the frequency and period of the clock used to decrement the counter will be

$f_{clk} = f_{base} /M \quad t_{clk} = t_{base} *M$

Software can configure the prescaler to slow down the counting. However, the interrupt period will be an integer multiple of t_{clk}. In addition, the interrupt period must be less than 2N * t_{clk}. Thus, the smaller the prescale M is, the more fine control the software has in selecting the interrupt period. On the other hand, the larger prescale M is, the longer the interrupt could be. Thus, the prescaler allows the software to control the trade-off between maximum interrupt period and the fine-tuning selection of the interrupt period.

Because the counter goes from the reload value down to 0, and then back to the reload value, an interrupt will be triggered every R+1 counts. Thus the interrupt period, P, will be

$P = t_{base} * M * (R + 1)$

Solving this equation for R, if we wish to create an interrupt with period P, we make

$R = (P / (t_{base} * M)) – 1$

Remember R must be an integer less than 2N. Most timers have a limited choice for the prescale M. Luckily, most microcontrollers have a larger number of timers. The TM4C123 has six 32-bit timers and six 64-bit timers. The board support package provides support for two independent periodic interrupts. The BSP uses a separate 32-bit timer to implement the BSP_Time_Get feature.

The BSP provides three timers. For each timer there are two functions, one to start and one to stop. The initialization will not clear the I bit, but will set up the timer for periodic interrupts, arm the trigger in the timer, set the priority in the NVIC, and arm the timer in the NVIC. The prototypes for these functions (found in the BSP.h file) are as follows:

```
// ------------BSP_PeriodicTask_Init------------
// Activate an interrupt to run a user task periodically.
// Input: task is a pointer to a user function
//        freq is number of interrupts per second 1 Hz to
10 kHz
//        priority is a number 0 to 6
// Output: none
void BSP_PeriodicTask_Init(void(*task)(void), uint32_t
freq, uint8_t priority);

// ------------BSP_PeriodicTask_Stop------------
// Deactivate the interrupt running a user task
periodically.
// Input: none
// Output: none
void BSP_PeriodicTask_Stop(void);
```

[You don’t need to implement/change anything on BSP.c]

To initialize a periodic task you need to send the task with BSP_PeriodicTask_Init(void(*task)(void), uint32_t freq, uint8_t priority) function. In this lab you have two periodic tasks which activity is summarized in runperiodicevents() function. So, at the time of OS_Launch() you can send the run runperiodicevents() function as a task of BSP_PeriodicTask_Init(….). The activity of multiple periodic tasks in runperiodicevents() has been described as a schedular in Lab13 manual.

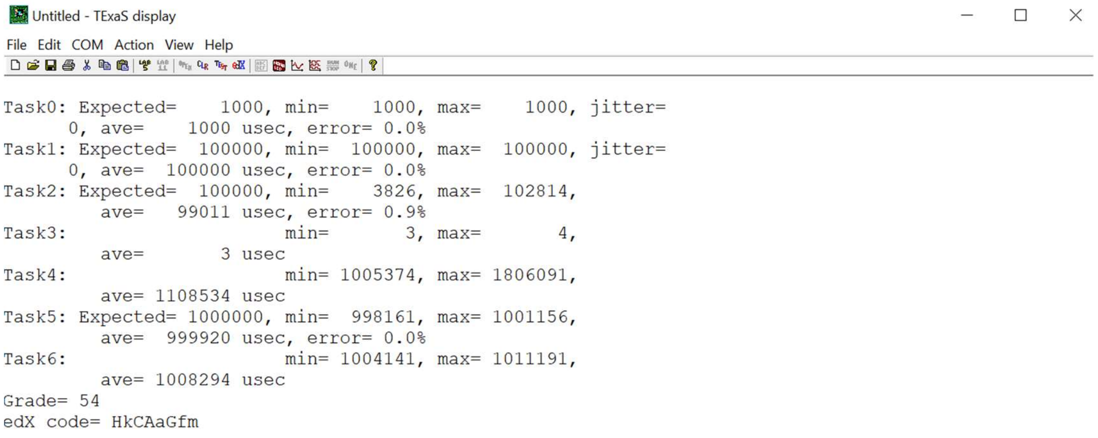
```
void OS_Launch(uint32_t theTimeSlice){
  STCTRL = 0;                  // disable SysTick
during setup
  STCURRENT = 0;               // any write to current
clears it
  SYSPRI3 =(SYSPRI3&0x00FFFFFF)|0xE0000000; // priority
7
  STRELOAD = theTimeSlice - 1; // reload value
  STCTRL = 0x00000007;         // enable, core clock
and interrupt arm
  BSP_PeriodicTask_Init(&runperiodicevents,1000,4); //
check with priority 1
    StartOS();                   // start on the first
task
}
```

To add a periodic task you need to add the periodicfunction (*Periodic_TaskPt) and period in tcb.

```
int OS_AddPeriodicEventThread(void(*task)(void),
uint32_t p){
        periodEv[EventIndx].Periodic_TaskPt = task;
        periodEv[EventIndx].period = p;
  return 1;}
```

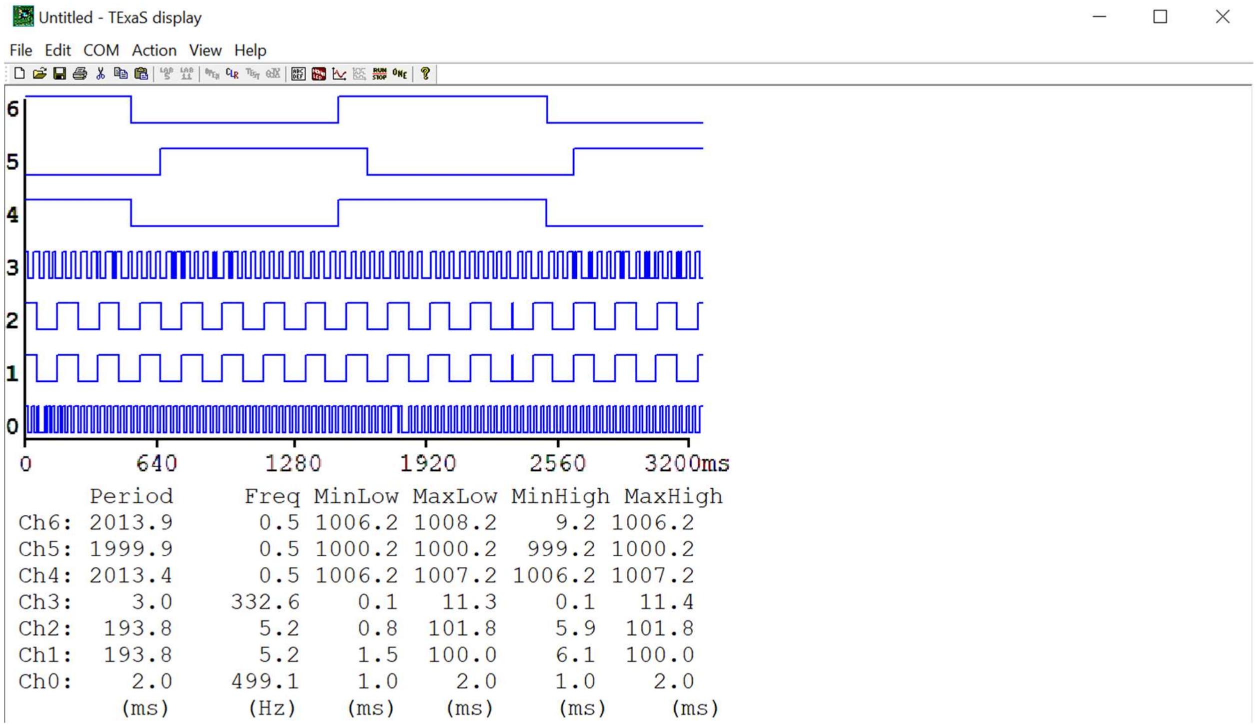
Final Result:

Now after these steps if you run the code with actual main function you should get a the performance in GRADER as:



```
Task0: Expected=    1000, min=    1000, max=    1000, jitter=
      0, ave=    1000 usec, error= 0.0%
Task1: Expected=  100000, min=  100000, max=  100000, jitter=
      0, ave=  100000 usec, error= 0.0%
Task2: Expected=  100000, min=    3826, max=  102814,
         ave=   99011 usec, error= 0.9%
Task3:                    min=       3, max=       4,
         ave=       3 usec
Task4:                    min= 1005374, max= 1806091,
         ave= 1108534 usec
Task5: Expected= 1000000, min=  998161, max= 1001156,
         ave=  999920 usec, error= 0.0%
Task6:                    min= 1004141, max= 1011191,
         ave= 1008294 usec
Grade= 54
edX code= HkCAaGfm
```

The waveform in LOGICANALYZER is expected to show as:



```
       Period      Freq MinLow MaxLow MinHigh MaxHigh
Ch6:  2013.9        0.5 1006.2 1008.2     9.2 1006.2
Ch5:  1999.9        0.5 1000.2 1000.2   999.2 1000.2
Ch4:  2013.4        0.5 1006.2 1007.2  1006.2 1007.2
Ch3:     3.0      332.6    0.1   11.3     0.1   11.4
Ch2:   193.8        5.2    0.8  101.8     5.9  101.8
Ch1:   193.8        5.2    1.5  100.0     6.1  100.0
Ch0:     2.0      499.1    1.0    2.0     1.0     2.0
         (ms)      (Hz)   (ms)   (ms)    (ms)    (ms)
```