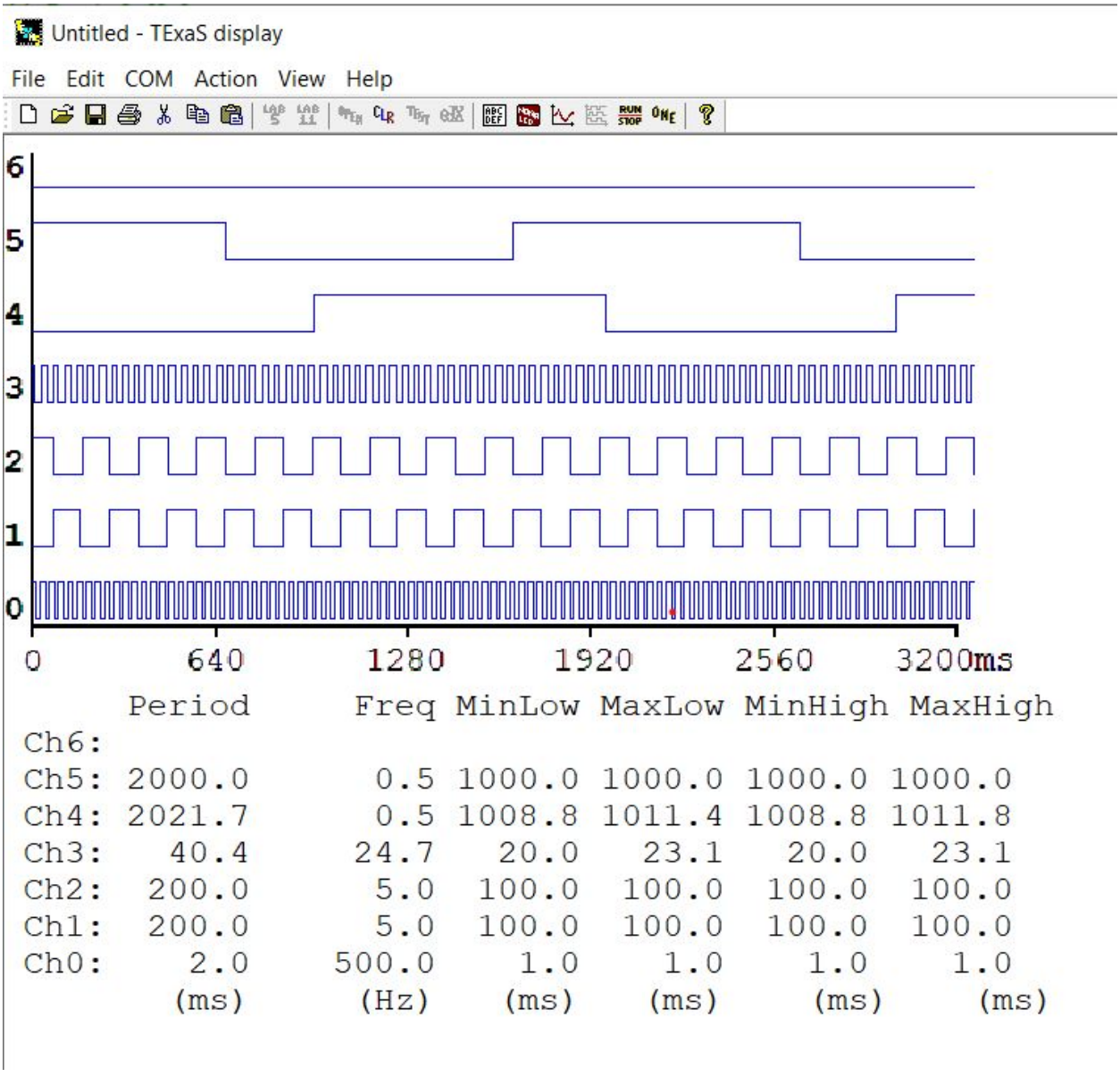
Riley Ruckman
TCES460, Wi21
Lab 13

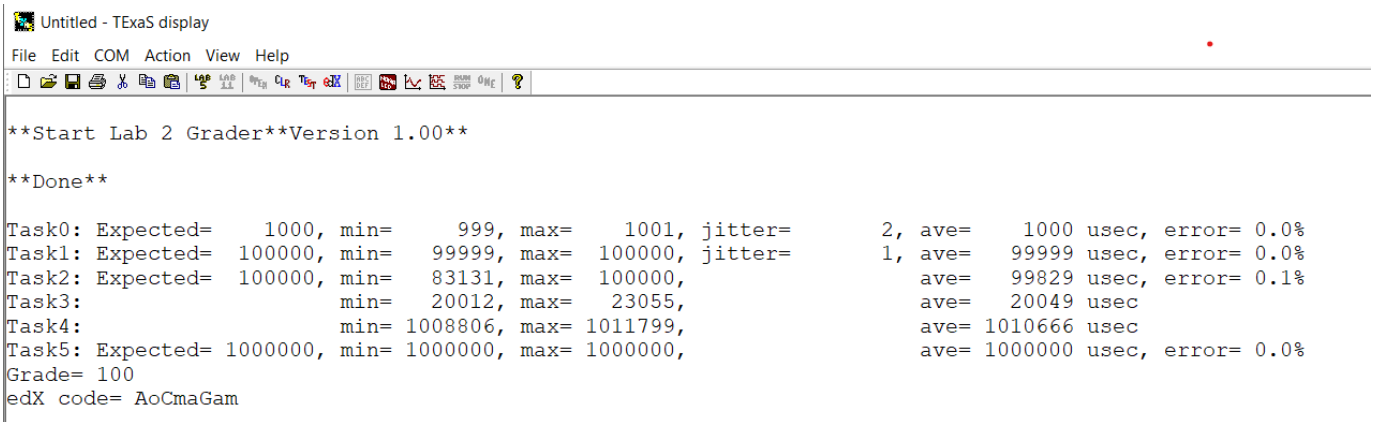
TExaS Display Screenshots:

LOGICANALYZER



GRADER

Lab 5:


Untitled - TExaS display
File Edit COM Action View Help
Start Lab 2 Grader Version 1.00**
Done
Task0: Expected= 1000, min= 999, max= 1001, jitter= 2, ave= 1000 usec, error= 0.0%
Task1: Expected= 100000, min= 99999, max= 100000, jitter= 1, ave= 99999 usec, error= 0.0%
Task2: Expected= 100000, min= 83131, max= 100000, ave= 99829 usec, error= 0.1%
Task3: min= 20012, max= 23055, ave= 20049 usec
Task4: min= 1008806, max= 1011799, ave= 1010666 usec
Task5: Expected= 1000000, min= 1000000, max= 1000000, ave= 1000000 usec, error= 0.0%
Grade= 100
edX code= AoCmaGam

Code:

os.c

```
// Riley Ruckman
// TCES460, Wi21
// Lab13 - os.c

// os.c
// Runs on LM4F120/TM4C123/MSP432
// Lab 2 starter file.
// Daniel Valvano
// February 20, 2016

#include <stdint.h>
#include "os.h"
#include "../inc/CortexM.h"
#include "../inc/BSP.h"

// function definitions in osasm.s
void StartOS(void);

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

// ******** OS_Init ************
// Initialize operating system, disable interrupts
// Initialize OS controlled I/O: systick, bus clock as fast as possible
// Initialize OS global variables
// Inputs:  none
// Outputs: none
void OS_Init(void){
  DisableInterrupts();
  BSP_Clock_InitFastest();// set processor clock to fastest speed
  // initialize any global variables as needed
  //***YOU IMPLEMENT THIS FUNCTION*****

}

void SetInitialStack(int i){
  //***YOU IMPLEMENT THIS FUNCTION*****
          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
}

//******** OS_AddThreads ***************
// Add four main threads to the scheduler
// Inputs: function pointers to four void/void main threads
// Outputs: 1 if successful, 0 if this thread can not be added
// This function will only be called once, after OS_Init and before OS_Launch
int OS_AddThreads(void(*thread0)(void),
                  void(*thread1)(void),
                  void(*thread2)(void),
                  void(*thread3)(void)){ int32_t status;
          // initialize TCB circular list
          // initialize RunPt
          // initialize four stacks, including initial PC
  //***YOU IMPLEMENT THIS FUNCTION*****
          status = StartCritical();
  tcbs[0].next = &tcbs[1]; // 0 points to 1
  tcbs[1].next = &tcbs[2]; // 1 points to 2
  tcbs[2].next = &tcbs[3]; // 2 points to 3
          tcbs[3].next = &tcbs[0]; // 3 points to 0

  SetInitialStack(0); Stacks[0][STACKSIZE-2] = (int32_t)(thread0); // PC
  SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32_t)(thread1); // PC
  SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32_t)(thread2); // PC
          SetInitialStack(3); Stacks[3][STACKSIZE-2] = (int32_t)(thread3); // PC
  RunPt = &tcbs[0];       // thread 0 will run first
  EndCritical(status);
  return 1;               // successful
}
```

```
//******** OS_AddThreads3 ***************
// add three foregound threads to the scheduler
// This is needed during debugging and not part of final solution
// Inputs: three pointers to a void/void foreground tasks
// Outputs: 1 if successful, 0 if this thread can not be added
int OS_AddThreads3(void(*task0)(void),
                   void(*task1)(void),
                   void(*task2)(void)){ int32_t status;
          // initialize TCB circular list (same as RTOS project)
          // initialize RunPt
          // initialize four stacks, including initial PC
  //***YOU IMPLEMENT THIS FUNCTION*****
          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); Stacks[0][STACKSIZE-2] = (int32_t)(task0); // PC
  SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32_t)(task1); // PC
  SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32_t)(task2); // PC
  RunPt = &tcbs[0];       // thread 0 will run first
  EndCritical(status);
  return 1;               // successful
}

//******** OS_AddPeriodicEventThreads ***************
// Add two background periodic event threads
// Typically this function receives the highest priority
// Inputs: pointers to a void/void event thread function2
//         periods given in units of OS_Launch (Lab 2 this will be msec)
// Outputs: 1 if successful, 0 if this thread cannot be added
// It is assumed that the event threads will run to completion and return
// It is assumed the time to run these event threads is short compared to 1 msec
// These threads cannot spin, block, loop, sleep, or kill
// These threads can call OS_Signal
void (*PeriodicEvent1)(void); // pointer to first periodic user function
uint32_t event1Period;                          // pointer to period of first periodic user
function
void (*PeriodicEvent2)(void); // pointer to second periodic user function
uint32_t event2Period;                          // pointer to period of second periodic user
function
int OS_AddPeriodicEventThreads(void(*thread1)(void), uint32_t period1,
  void(*thread2)(void), uint32_t period2){
  //***YOU IMPLEMENT THIS FUNCTION*****
          PeriodicEvent1 = thread1;
```

```
        event1Period = period1;
        PeriodicEvent2 = thread2;
        event2Period = period2;
  return 1;
}

//******** OS_Launch ***************
// Start the scheduler, enable interrupts
// Inputs: number of clock cycles for each time slice
// Outputs: none (does not return)
// Errors: theTimeSlice must be less than 16,777,216
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
  StartOS();                // start on the first task
}

uint32_t Counter = 0;
// runs every ms
void Scheduler(void){ // every time slice
  // run any periodic event threads if needed
  // implement round robin scheduler, update RunPt
  //***YOU IMPLEMENT THIS FUNCTION*****
        Counter = (Counter + 1) % (event1Period*event2Period);
        if (Counter%event1Period == 0)
                (*PeriodicEvent1)();

        if (Counter%event2Period == 1)
                (*PeriodicEvent2)();
        RunPt = RunPt->next;// Round Robin scheduler
}

// ******** OS_InitSemaphore ************
// Initialize counting semaphore
// Inputs:  pointer to a semaphore
//          initial value of semaphore
// Outputs: none
void OS_InitSemaphore(int32_t *semaPt, int32_t value){
  //***YOU IMPLEMENT THIS FUNCTION*****
        *semaPt = value;
}
```

```
// ******** OS_Wait ************
// Decrement semaphore
// Lab2 spinlock (does not suspend while spinning)
// Lab3 block if less than zero
// Inputs:  pointer to a counting semaphore
// Outputs: none
void OS_Wait(int32_t *semaPt){
        DisableInterrupts();
        while((*semaPt) == 0) {
                EnableInterrupts();
                DisableInterrupts();
        }
        (*semaPt) = (*semaPt) - 1;
        EnableInterrupts();
}

// ******** OS_Signal ************
// Increment semaphore
// Lab2 spinlock
// Lab3 wakeup blocked thread if appropriate
// Inputs:  pointer to a counting semaphore
// Outputs: none
void OS_Signal(int32_t *semaPt){
//***YOU IMPLEMENT THIS FUNCTION*****
        DisableInterrupts();
        (*semaPt) = (*semaPt) + 1;
        EnableInterrupts();
}

// ******** OS_MailBox_Init ************
// Initialize communication channel
// Producer is an event thread, consumer is a main thread
// Inputs:  none
// Outputs: none
int32_t Mail;                           // Shared data
int32_t Send;           // semaphore for sent data
int32_t Ack;            // semaphore for data received acknowledgement
void OS_MailBox_Init(void){
  // include data field and semaphore
  //***YOU IMPLEMENT THIS FUNCTION*****
        Mail = 0;
        Send = 0;
        Ack = 0;
```

```
}

// ******** OS_MailBox_Send ************
// Enter data into the MailBox, do not spin/block if full
// Use semaphore to synchronize with OS_MailBox_Recv
// Inputs:  data to be sent
// Outputs: none
// Errors: data lost if MailBox already has data
uint32_t Lost = 0;
void OS_MailBox_Send(uint32_t data){
  //***YOU IMPLEMENT THIS FUNCTION*****
        Mail = data;
        if(Send) {
                Lost++;
        } else {
                OS_Signal(&Send);
        }
}

// ******** OS_MailBox_Recv ************
// retreive mail from the MailBox
// Use semaphore to synchronize with OS_MailBox_Send
// Lab 2 spin on semaphore if mailbox empty
// Lab 3 block on semaphore if mailbox empty
// Inputs:  none
// Outputs: data retreived
// Errors:  none
uint32_t OS_MailBox_Recv(void){
  //***YOU IMPLEMENT THIS FUNCTION*****
        OS_Wait(&Send);
  return Mail;
}
```

os.h

```
// os.h
// Runs on LM4F120/TM4C123/MSP432
// A very simple real time operating system with minimal features.
// Daniel Valvano
// February 20, 2016

/* This example accompanies the book

   "Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers",
   ISBN: 978-1466468863, , Jonathan Valvano, copyright (c) 2016
   Programs 4.4 through 4.12, section 4.2

 Copyright 2016 by Jonathan W. Valvano, valvano@mail.utexas.edu
    You may use, edit, run or distribute this file
    as long as the above copyright notice remains
 THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS,
IMPLIED
 OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS
SOFTWARE.
 VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL,
INCIDENTAL,
 OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 For more information about my classes, my research, and my books, see
 http://users.ece.utexas.edu/~valvano/
 */


#ifndef __OS_H
#define __OS_H  1
// grader needs access to TCBs and stacks
#define NUMTHREADS  4        // maximum number of threads
#define STACKSIZE   100      // number of 32-bit words in stack per thread
struct tcb{
  int32_t *sp;       // pointer to stack (valid for threads not running
  struct tcb *next;  // linked-list pointer
};
typedef struct tcb tcbType;

// ******** OS_Init ************
// Initialize operating system, disable interrupts
// Initialize OS controlled I/O: systick, bus clock as fast as possible
```

```
// Initialize OS global variables
// Inputs:  none
// Outputs: none
void OS_Init(void);



//******** OS_AddThreads ***************
// Add four main threads to the scheduler
// Inputs: function pointers to four void/void main threads
// Outputs: 1 if successful, 0 if this thread can not be added
// This function will only be called once, after OS_Init and before OS_Launch
int OS_AddThreads(void(*thread0)(void),
                  void(*thread1)(void),
                  void(*thread2)(void),
                  void(*thread3)(void));

//******** OS_AddThreads3 ***************
// add three foregound threads to the scheduler
// This is needed during debugging and not part of final solution
// Inputs: three pointers to a void/void foreground tasks
// Outputs: 1 if successful, 0 if this thread can not be added
int OS_AddThreads3(void(*task0)(void),
                   void(*task1)(void),
                   void(*task2)(void));

//******** OS_AddPeriodicEventThreads ***************
// Add two background periodic event threads
// Typically this function receives the highest priority
// Inputs: pointers to a void/void event thread function2
//         periods given in units of OS_Launch (Lab 2 this will be msec)
// Outputs: 1 if successful, 0 if this thread cannot be added
// It is assumed that the event threads will run to completion and return
// It is assumed the time to run these event threads is short compared to 1 msec
// These threads cannot spin, block, loop, sleep, or kill
// These threads can call OS_Signal
int OS_AddPeriodicEventThreads(void(*thread1)(void), uint32_t period1,
  void(*thread2)(void), uint32_t period2);

//******** OS_Launch ***************
// Start the scheduler, enable interrupts
// Inputs: number of clock cycles for each time slice
// Outputs: none (does not return)
// Errors: theTimeSlice must be less than 16,777,216
```

```
void OS_Launch(uint32_t theTimeSlice);


// ******** OS_InitSemaphore ************
// Initialize counting semaphore
// Inputs:  pointer to a semaphore
//          initial value of semaphore
// Outputs: none
void OS_InitSemaphore(int32_t *semaPt, int32_t value);

// ******** OS_Wait ************
// Decrement semaphore
// Lab2 spinlock (does not suspend while spinning)
// Lab3 block if less than zero
// Inputs:  pointer to a counting semaphore
// Outputs: none
void OS_Wait(int32_t *semaPt);

// ******** OS_Signal ************
// Increment semaphore
// Lab2 spinlock
// Lab3 wakeup blocked thread if appropriate
// Inputs:  pointer to a counting semaphore
// Outputs: none
void OS_Signal(int32_t *semaPt);

// ******** OS_MailBox_Init ************
// Initialize communication channel
// Producer is an event thread, consumer is a main thread
// Inputs:  none
// Outputs: none
void OS_MailBox_Init(void);

// ******** OS_MailBox_Send ************
// Enter data into the MailBox, do not spin/block if full
// Use semaphore to synchronize with OS_MailBox_Recv
// Inputs:  data to be sent
// Outputs: none
// Errors: data lost if MailBox already has data
void OS_MailBox_Send(uint32_t data);

// ******** OS_MailBox_Recv ************
// retreive mail from the MailBox
// Use semaphore to synchronize with OS_MailBox_Send
```

```
// Lab 2 spin on semaphore if mailbox empty
// Lab 3 block on semaphore if mailbox empty
// Inputs:  none
// Outputs: data retreived
// Errors:  none
uint32_t OS_MailBox_Recv(void);

#endif
```

osasm.s

```
; Riley Ruckman
; TCES460, Wi21
; Lab13 - os.c

;/*****************************************************************************/
; OSasm.s: low-level OS commands, written in assembly                       */
; Runs on LM4F120/TM4C123/MSP432
; Lab 2 starter file
; February 10, 2016
;


      AREA |.text|, CODE, READONLY, ALIGN=2
      THUMB
      REQUIRE8
      PRESERVE8

      EXTERN  RunPt          ; currently running thread
      EXPORT  StartOS
      EXPORT  SysTick_Handler
      IMPORT  Scheduler


SysTick_Handler               ; 1) Saves R0-R3,R12,LR,PC,PSR
   CPSID   I                  ; 2) Prevent interrupt during switch
   PUSH    {R4-R11}           ; 3) Save remaining regs r4-11
   LDR     R0, =RunPt         ; 4) R0=pointer to RunPt, old thread
   LDR     R1, [R0]           ;    R1 = RunPt
   STR     SP, [R1]           ; 5) Save SP into TCB
;   LDR     R1, [R1,#4]       ; 6) R1 = RunPt->next
;   STR     R1, [R0]          ;    RunPt = R1
   PUSH    {R0,LR}
   BL      Scheduler
   POP     {R0,LR}
   LDR     R1, [R0]           ; 6) R1 = RunPt, new thread
   LDR     SP, [R1]           ; 7) new thread SP; SP = RunPt->sp;
   POP     {R4-R11}           ; 8) restore regs r4-11
   CPSIE   I                  ; 9) tasks run with interrupts enabled
   BX      LR                 ; 10) restore R0-R3,R12,LR,PC,PSR

StartOS
   LDR     R0, =RunPt         ; currently running thread
```

```
LDR     R2, [R0]          ; R2 = value of RunPt
LDR     SP, [R2]          ; new thread SP; SP = RunPt->stackPointer;
POP     {R4-R11}          ; restore regs r4-11
POP     {R0-R3}           ; restore regs r0-3
POP     {R12}
ADD     SP,SP,#4          ; discard LR from initial stack
POP     {LR}              ; start location
ADD     SP,SP,#4          ; discard PSR
CPSIE   I                 ; Enable interrupts at processor level
BX      LR                ; start first thread

ALIGN
END
```

Code Screenshots:

os.c

```
// Riley Ruckman
// TCES460, Wi21
// Lab13 - os.c

// os.c
// Runs on LM4F120/TM4C123/MSP432
// Lab 2 starter file.
// Daniel Valvano
// February 20, 2016

#include <stdint.h>
#include "os.h"
#include "../inc/CortexM.h"
#include "../inc/BSP.h"

// function definitions in osasm.s
void StartOS(void);

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

// ******** OS_Init ************
// Initialize operating system, disable interrupts
// Initialize OS controlled I/O: systick, bus clock as fast as possible
// Initialize OS global variables
// Inputs:  none
// Outputs: none
void OS_Init(void){
  DisableInterrupts();
  BSP_Clock_InitFastest();// set processor clock to fastest speed
  // initialize any global variables as needed
  //***YOU IMPLEMENT THIS FUNCTION*****

}

void SetInitialStack(int i){
  //***YOU IMPLEMENT THIS FUNCTION*****
  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
```

```
54    Stacks[i][STACKSIZE-16] = 0x04040404;  // R4
55  }
56
57  //******** OS_AddThreads ***************
58  // Add four main threads to the scheduler
59  // Inputs: function pointers to four void/void main threads
60  // Outputs: 1 if successful, 0 if this thread can not be added
61  // This function will only be called once, after OS_Init and before OS_Launch
62  int OS_AddThreads(void(*thread0)(void),
63                    void(*thread1)(void),
64                    void(*thread2)(void),
65                    void(*thread3)(void)){ int32_t status;
66    // initialize TCB circular list
67    // initialize RunPt
68    // initialize four stacks, including initial PC
69    //***YOU IMPLEMENT THIS FUNCTION*****
70    status = StartCritical();
71    tcbs[0].next = &tcbs[1]; // 0 points to 1
72    tcbs[1].next = &tcbs[2]; // 1 points to 2
73    tcbs[2].next = &tcbs[3]; // 2 points to 3
74    tcbs[3].next = &tcbs[0]; // 3 points to 0
75    SetInitialStack(0); Stacks[0][STACKSIZE-2] = (int32_t)(thread0); // PC
76    SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32_t)(thread1); // PC
77    SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32_t)(thread2); // PC
78    SetInitialStack(3); Stacks[3][STACKSIZE-2] = (int32_t)(thread3); // PC
79    RunPt = &tcbs[0];       // thread 0 will run first
80    EndCritical(status);
81    return 1;               // successful
82  }
83
84  //******** OS_AddThreads3 ***************
85  // add three foregound threads to the scheduler
86  // This is needed during debugging and not part of final solution
87  // Inputs: three pointers to a void/void foreground tasks
88  // Outputs: 1 if successful, 0 if this thread can not be added
89  int OS_AddThreads3(void(*task0)(void),
90                     void(*task1)(void),
91                     void(*task2)(void)){ int32_t status;
92    // initialize TCB circular list (same as RTOS project)
93    // initialize RunPt
94    // initialize four stacks, including initial PC
95    //***YOU IMPLEMENT THIS FUNCTION*****
96    status = StartCritical();
97    tcbs[0].next = &tcbs[1]; // 0 points to 1
98    tcbs[1].next = &tcbs[2]; // 1 points to 2
99    tcbs[2].next = &tcbs[0]; // 2 points to 0
100   SetInitialStack(0); Stacks[0][STACKSIZE-2] = (int32_t)(task0); // PC
101   SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32_t)(task1); // PC
102   SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32_t)(task2); // PC
103   RunPt = &tcbs[0];       // thread 0 will run first
104   EndCritical(status);
```

```
105   return 1;               // successful
106 }
107
108 //******** OS_AddPeriodicEventThreads ***************
109 // Add two background periodic event threads
110 // Typically this function receives the highest priority
111 // Inputs: pointers to a void/void event thread function2
112 //         periods given in units of OS_Launch (Lab 2 this will be msec)
113 // Outputs: 1 if successful, 0 if this thread cannot be added
114 // It is assumed that the event threads will run to completion and return
115 // It is assumed the time to run these event threads is short compared to 1 msec
116 // These threads cannot spin, block, loop, sleep, or kill
117 // These threads can call OS_Signal
118 void (*PeriodicEvent1)(void);   // pointer to first periodic user function
119 uint32_t event1Period;         // pointer to period of first periodic user function
120 void (*PeriodicEvent2)(void);   // pointer to second periodic user function
121 uint32_t event2Period;         // pointer to period of second periodic user function
122 int OS_AddPeriodicEventThreads(void(*thread1)(void), uint32_t period1,
123   void(*thread2)(void), uint32_t period2){
124   //***YOU IMPLEMENT THIS FUNCTION*****
125   PeriodicEvent1 = thread1;
126   event1Period = period1;
127   PeriodicEvent2 = thread2;
128   event2Period = period2;
129   return 1;
130 }
131
132 //******** OS_Launch ***************
133 // Start the scheduler, enable interrupts
134 // Inputs: number of clock cycles for each time slice
135 // Outputs: none (does not return)
136 // Errors: theTimeSlice must be less than 16,777,216
137 void OS_Launch(uint32_t theTimeSlice){
138   STCTRL = 0;                  // disable SysTick during setup
139   STCURRENT = 0;               // any write to current clears it
140   SYSPRI3 =(SYSPRI3&0x00FFFFFF)|0xE0000000; // priority 7
141   STRELOAD = theTimeSlice - 1; // reload value
142   STCTRL = 0x00000007;         // enable, core clock and interrupt arm
143   StartOS();                   // start on the first task
144 }
145
146 uint32_t Counter = 0;
147 // runs every ms
148 void Scheduler(void){ // every time slice
149   // run any periodic event threads if needed
150   // implement round robin scheduler, update RunPt
151   //***YOU IMPLEMENT THIS FUNCTION*****
152   Counter = (Counter + 1) % (event1Period*event2Period);
153   if (Counter%event1Period == 0)
154     (*PeriodicEvent1)();
155
```

```
156     if (Counter%event2Period == 1)
157       (*PeriodicEvent2)();
158     RunPt = RunPt->next;  // Round Robin scheduler
159   }
160
161   // ******** OS_InitSemaphore ************
162   // Initialize counting semaphore
163   // Inputs:  pointer to a semaphore
164   //          initial value of semaphore
165   // Outputs: none
166   void OS_InitSemaphore(int32_t *semaPt, int32_t value){
167     //***YOU IMPLEMENT THIS FUNCTION*****
168     *semaPt = value;
169   }
170
171   // ******** OS_Wait ************
172   // Decrement semaphore
173   // Lab2 spinlock (does not suspend while spinning)
174   // Lab3 block if less than zero
175   // Inputs:  pointer to a counting semaphore
176   // Outputs: none
177   void OS_Wait(int32_t *semaPt){
178     DisableInterrupts();
179     while((*semaPt) == 0) {
180       EnableInterrupts();
181       DisableInterrupts();
182     }
183     (*semaPt) = (*semaPt) - 1;
184     EnableInterrupts();
185   }
186
187   // ******** OS_Signal ************
188   // Increment semaphore
189   // Lab2 spinlock
190   // Lab3 wakeup blocked thread if appropriate
191   // Inputs:  pointer to a counting semaphore
192   // Outputs: none
193   void OS_Signal(int32_t *semaPt){
194   //***YOU IMPLEMENT THIS FUNCTION*****
195     DisableInterrupts();
196     (*semaPt) = (*semaPt) + 1;
197     EnableInterrupts();
198   }
199
200   // ******** OS_MailBox_Init ************
201   // Initialize communication channel
202   // Producer is an event thread, consumer is a main thread
203   // Inputs:  none
204   // Outputs: none
205   int32_t Mail;       // Shared data
206   int32_t Send;   // semaphore for sent data
```

```
207  int32_t Ack;    // semaphore for data received acknowledgement
208  void OS_MailBox_Init(void){
209    // include data field and semaphore
210    //***YOU IMPLEMENT THIS FUNCTION*****
211    Mail = 0;
212    Send = 0;
213    Ack = 0;
214  }
215
216  // ******** OS_MailBox_Send ************
217  // Enter data into the MailBox, do not spin/block if full
218  // Use semaphore to synchronize with OS_MailBox_Recv
219  // Inputs:  data to be sent
220  // Outputs: none
221  // Errors: data lost if MailBox already has data
222  uint32_t Lost = 0;
223  void OS_MailBox_Send(uint32_t data){
224    //***YOU IMPLEMENT THIS FUNCTION*****
225    Mail = data;
226    if(Send) {
227      Lost++;
228    } else {
229      OS_Signal(&Send);
230    }
231  }
232
233  // ******** OS_MailBox_Recv ************
234  // retreive mail from the MailBox
235  // Use semaphore to synchronize with OS_MailBox_Send
236  // Lab 2 spin on semaphore if mailbox empty
237  // Lab 3 block on semaphore if mailbox empty
238  // Inputs:  none
239  // Outputs: data retreived
240  // Errors:  none
241  uint32_t OS_MailBox_Recv(void){
242    //***YOU IMPLEMENT THIS FUNCTION*****
243    OS_Wait(&Send);
244    return Mail;
245  }
246
```

os.h

```
// os.h
// Runs on LM4F120/TM4C123/MSP432
// A very simple real time operating system with minimal features.
// Daniel Valvano
// February 20, 2016

/* This example accompanies the book

   "Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers",
   ISBN: 978-1466468863, , Jonathan Valvano, copyright (c) 2016
   Programs 4.4 through 4.12, section 4.2

 Copyright 2016 by Jonathan W. Valvano, valvano@mail.utexas.edu
    You may use, edit, run or distribute this file
    as long as the above copyright notice remains
 THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS, IMPLIED
 OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
 VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,
 OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 For more information about my classes, my research, and my books, see
 http://users.ece.utexas.edu/~valvano/
 */


#ifndef __OS_H
#define __OS_H  1
// grader needs access to TCBs and stacks
#define NUMTHREADS  4        // maximum number of threads
#define STACKSIZE   100      // number of 32-bit words in stack per thread
struct tcb{
  int32_t *sp;       // pointer to stack (valid for threads not running
  struct tcb *next;  // linked-list pointer
};
typedef struct tcb tcbType;

// ******** OS_Init ************
// Initialize operating system, disable interrupts
// Initialize OS controlled I/O: systick, bus clock as fast as possible
// Initialize OS global variables
// Inputs:  none
// Outputs: none
void OS_Init(void);



//******** OS_AddThreads ***************
// Add four main threads to the scheduler
// Inputs: function pointers to four void/void main threads
// Outputs: 1 if successful, 0 if this thread can not be added
// This function will only be called once, after OS_Init and before OS_Launch
int OS_AddThreads(void(*thread0)(void),
                  void(*thread1)(void),
```

```
                  void(*thread2)(void),
                  void(*thread3)(void));

//******** OS_AddThreads3 ***************
// add three foregound threads to the scheduler
// This is needed during debugging and not part of final solution
// Inputs: three pointers to a void/void foreground tasks
// Outputs: 1 if successful, 0 if this thread can not be added
int OS_AddThreads3(void(*task0)(void),
                   void(*task1)(void),
                   void(*task2)(void));

//******** OS_AddPeriodicEventThreads ***************
// Add two background periodic event threads
// Typically this function receives the highest priority
// Inputs: pointers to a void/void event thread function2
//         periods given in units of OS_Launch (Lab 2 this will be msec)
// Outputs: 1 if successful, 0 if this thread cannot be added
// It is assumed that the event threads will run to completion and return
// It is assumed the time to run these event threads is short compared to 1 msec
// These threads cannot spin, block, loop, sleep, or kill
// These threads can call OS_Signal
int OS_AddPeriodicEventThreads(void(*thread1)(void), uint32_t period1,
  void(*thread2)(void), uint32_t period2);

//******** OS_Launch ***************
// Start the scheduler, enable interrupts
// Inputs: number of clock cycles for each time slice
// Outputs: none (does not return)
// Errors: theTimeSlice must be less than 16,777,216
void OS_Launch(uint32_t theTimeSlice);


// ******** OS_InitSemaphore ************
// Initialize counting semaphore
// Inputs:  pointer to a semaphore
//          initial value of semaphore
// Outputs: none
void OS_InitSemaphore(int32_t *semaPt, int32_t value);

// ******** OS_Wait ************
// Decrement semaphore
// Lab2 spinlock (does not suspend while spinning)
// Lab3 block if less than zero
// Inputs:  pointer to a counting semaphore
// Outputs: none
void OS_Wait(int32_t *semaPt);

// ******** OS_Signal ************
// Increment semaphore
// Lab2 spinlock
```

```
105 // Lab3 wakeup blocked thread if appropriate
106 // Inputs:  pointer to a counting semaphore
107 // Outputs: none
108 void OS_Signal(int32_t *semaPt);
109
110 // ******** OS_MailBox_Init ************
111 // Initialize communication channel
112 // Producer is an event thread, consumer is a main thread
113 // Inputs:  none
114 // Outputs: none
115 void OS_MailBox_Init(void);
116
117 // ******** OS_MailBox_Send ************
118 // Enter data into the MailBox, do not spin/block if full
119 // Use semaphore to synchronize with OS_MailBox_Recv
120 // Inputs:  data to be sent
121 // Outputs: none
122 // Errors: data lost if MailBox already has data
123 void OS_MailBox_Send(uint32_t data);
124
125 // ******** OS_MailBox_Recv ************
126 // retreive mail from the MailBox
127 // Use semaphore to synchronize with OS_MailBox_Send
128 // Lab 2 spin on semaphore if mailbox empty
129 // Lab 3 block on semaphore if mailbox empty
130 // Inputs:  none
131 // Outputs: data retreived
132 // Errors:  none
133 uint32_t OS_MailBox_Recv(void);
134
135 #endif
136
```

osasm.s

```
; Riley Ruckman
; TCES460, Wi21
; Lab13 - os.c

;/*****************************************************************************/
; OSasm.s: low-level OS commands, written in assembly                       */
; Runs on LM4F120/TM4C123/MSP432
; Lab 2 starter file
; February 10, 2016
;


        AREA |.text|, CODE, READONLY, ALIGN=2
        THUMB
        REQUIRE8
        PRESERVE8

        EXTERN  RunPt            ; currently running thread
        EXPORT  StartOS
        EXPORT  SysTick_Handler
        IMPORT  Scheduler


SysTick_Handler                ; 1) Saves R0-R3,R12,LR,PC,PSR
    CPSID   I                  ; 2) Prevent interrupt during switch
    PUSH    {R4-R11}           ; 3) Save remaining regs r4-11
    LDR     R0, =RunPt         ; 4) R0=pointer to RunPt, old thread
    LDR     R1, [R0]           ;    R1 = RunPt
    STR     SP, [R1]           ; 5) Save SP into TCB
;    LDR     R1, [R1,#4]       ; 6) R1 = RunPt->next
;    STR     R1, [R0]          ;    RunPt = R1
    PUSH    {R0,LR}
    BL      Scheduler
    POP     {R0,LR}
    LDR     R1, [R0]           ; 6) R1 = RunPt, new thread
    LDR     SP, [R1]           ; 7) new thread SP; SP = RunPt->sp;
    POP     {R4-R11}           ; 8) restore regs r4-11
    CPSIE   I                  ; 9) tasks run with interrupts enabled
    BX      LR                 ; 10) restore R0-R3,R12,LR,PC,PSR

StartOS
    LDR     R0, =RunPt         ; currently running thread
    LDR     R2, [R0]           ; R2 = value of RunPt
    LDR     SP, [R2]           ; new thread SP; SP = RunPt->stackPointer;
    POP     {R4-R11}           ; restore regs r4-11
    POP     {R0-R3}            ; restore regs r0-3
    POP     {R12}
    ADD     SP,SP,#4           ; discard LR from initial stack
    POP     {LR}               ; start location
    ADD     SP,SP,#4           ; discard PSR
    CPSIE   I                  ; Enable interrupts at processor level
    BX      LR                 ; start first thread

```

```
54      ALIGN
55      END
56
```