Lab 13

Submission February 25th 11:59 PM

Points 100

In this lab, you need to understand how an RTOS works and demonstrate your understanding by completing a set of activities. In this lab you need to edit the code in Lab2_4C123.

However, the given code will not execute until you implement a very simple RTOS. The user code inputs from the microphone, accelerometer, temperature sensor and switches. It performs some simple measurements and calculations of steps, sound intensity, and temperature. It outputs data to the LCD and it generates simple beeping sounds. Figure 13.1 shows the data flow graph of Lab 13. Your assignment is to first run and understand the concepts of the projects in folder RTOS_xxx and RoundRobin_xxx in specific. In RTOS_xxx project 3-main thread run without any event thread. In this lab your RTOS will run two periodic event threads and four main threads.

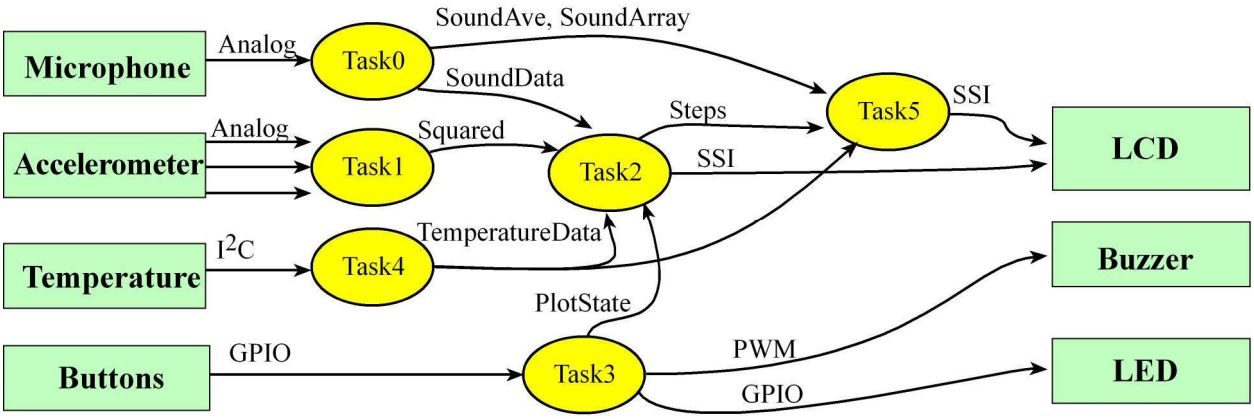

Figure 13.1: Data flow graph

In this lab as previous Lab 11 and 12, we are running a simple fitness device. This simple fitness device has six tasks: two periodic (event thread) and four main threads. Since we have two periodic threads to schedule, we could have used interrupts on two hardware timers to run the real-time periodic threads. However, Lab 13 will run with SysTick (same as TIMER/GPIO interrupts but manages by System) interrupts to run two periodic threads and to switch between the four main threads. These are the six tasks:

Task0: event thread samples microphone input at 1000 Hz

Task1: event thread samples acceleration input at 10 Hz

Task2: main thread detecting steps and plotting at on LCD, runs about 10 Hz

Task3: main thread inputs from switches, outputs to buzzer

Task4: main thread measures temperature, runs about 1 Hz

Task5: main thread output numerical data to LCD, runs about 1 Hz

Submission: After implementing the RTOS 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.

Notes: Your assignment is to implement the OS functions in OS.c and write the SysTick interrupt service routine in osasm.s. You don't need to edit the user code in Lab2.c, the board support package in BSP.c, or the interface specifications in profile.h, Texas.h, BSP.h, or OS.h. More specifically, you need to develop a real-time operating system.

Prelab

Before you begin editing, and debugging, you should first open up and run a couple of projects. The first project is RTOS_xxx. This project implements a very simple real-time operating system. Next you should open up the project RoundRobin_xxx. This project extends the simple real time operating system so the scheduler works.

However, the code is designed to implement in a LogicAnalyzer device. So, you need to add the option to display the result in TExaSdisplay. To add this option you need to add several lines in the user.c code:

#include "Texas.h"

Also add the following line in each void Taskn() functions. You should add this line where Profile_Togglen() is mentioned.

TExaS_Taskn();

Now add following two lines in the int main() function:

//TExaS_Init(GRADER, 1000);

TExaS_Init(LOGICANALYZER, 1000);

If you investigate the graph and text, you can see the data loss is almost zero (<0.1%) while running these three threads (count0, count1, count2).

Now thoroughly investigate the os.c and osasm.s code in these two projects.

A useful presentation on Logic analyzer of TExaSdisplay:
https://www.youtube.com/watch?v=kExg3F8UMr0

Coding Concepts and Theories

There are two types of threads in our simple OS as event thread and main thread. Event threads are attached to hardware and should execute changes in hardware status. Examples include periodic threads that should be executed at a fixed rate (like the microphone, accelerometer and light measurements in Lab 11/12), input threads that should be executed when new data are available at the input device (like the operator pushed a button), and output threads that should be executed when the output device is idle and new data are available for output. They are typically defined as void-void

functions. The time to execute an event thread should be short and bounded. In other words, event threads must execute and return. The time to execute an event thread must always be less than a small value (e.g., 10µs). In an embedded system without an OS, event threads are simply the interrupt service routines (ISRs). However, with a RTOS, we will have the OS manage the processor and I/O, and therefore the OS will manage the ISRs. The user will write the software executed as an event thread, but the OS will manage the ISR and call the appropriate event thread. Communication between threads will be managed by the OS.

```
void inputThread(void){
  data = ReadInput();
  Send(data);
}
void outputThread(void){
  data = Recv();
  WriteOutput(data);
}
void periodicThread(void){
  PerformTask();
}
```

Program 13.1: Periodic Thread

The second type of thread is a main thread. Without an OS, embedded systems typically have one main program that is executed on start up. This main initializes the system and defines the high level behavior of the system. In an OS however, we will have multiple main threads. Main threads execute like main programs that never return. These threads execute an initialization once and then repeatedly execute a sequence of steps within a while loop.

```
void mainThread(void){
  Init();
  while(1){
    Body();
  }
}
```

Program 13.2: Main thread

Each thread has a thread control block (TCB) encapsulating the state of the thread. For now, a thread's TCB we will only maintain a link to its stack and a link to the TCB of the next thread. The **RunPt** points to the TCB of the thread that is currently running. The next field is a pointer chaining all three TCBs into a circular linked list. Each TCB has an sp field. 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. Other fields that define a thread's state such as, status, Id, sleeping, age, and priority will be added later. However, for your first RTOS, the sp and next fields will be sufficient. The scheduler traverses the linked list of TCBs to find the

next thread to run. In this example there are three threads in a circular linked list. Each thread runs for a fixed amount of time, and a periodic interrupt suspends the running thread and switches **RunPt** to the next thread in the circular list. The scheduler then launches the next thread. The Thread Control Block (TCB) will store the information private to each thread. There will be a TCB structure and a stack for each thread. While a thread is running, it uses the actual Cortex M hardware registers.

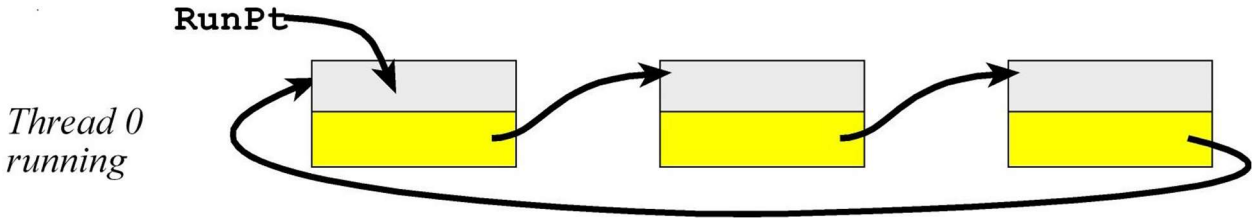

Figure 13.2: Three threads have their TCBs in a circular linked list

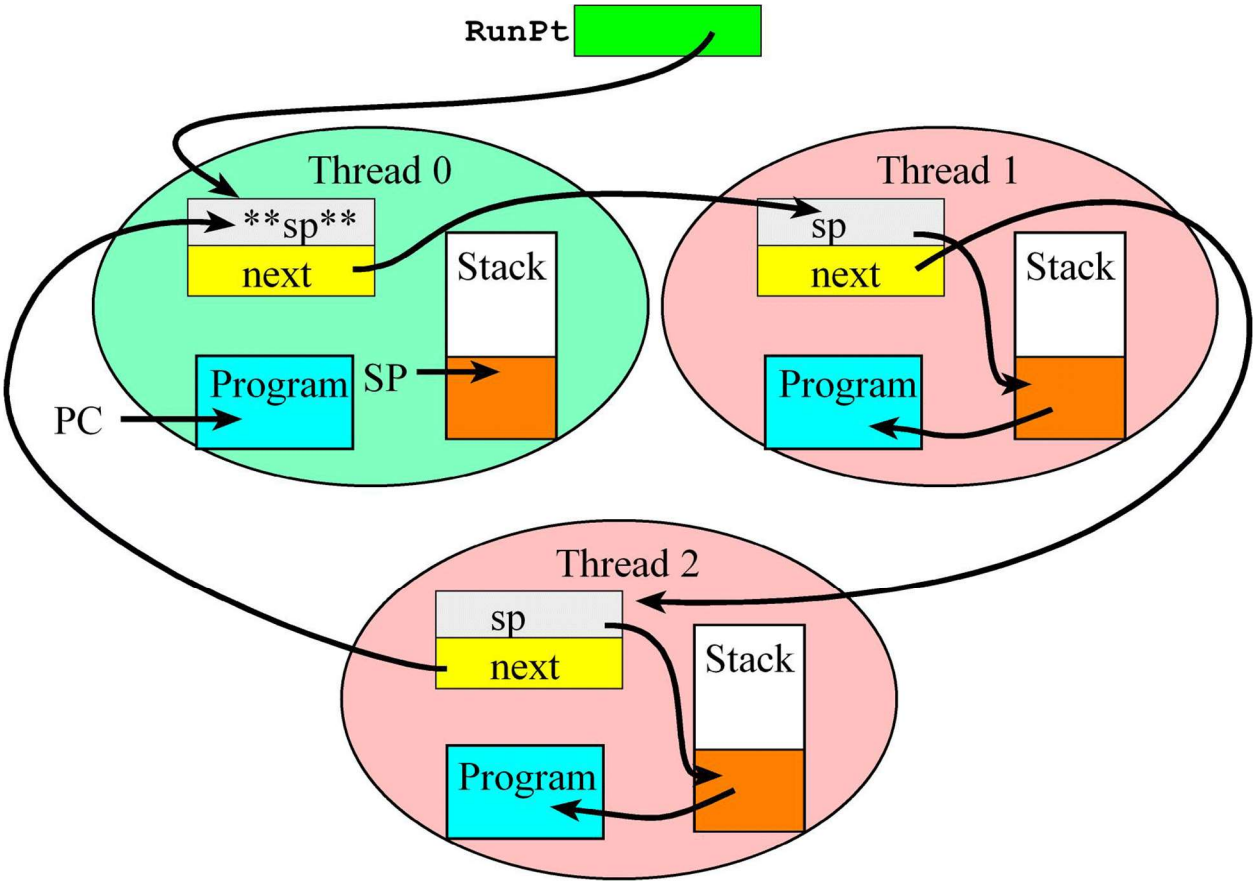

Figure 13.3: Three threads have their TCBs in a circular linked list. "**sp**" means this field is invalid for the one thread that is actually running.

The code to initialize three TCBs are as following:

```
#define NUMTHREADS 3  // maximum number of threads
#define STACKSIZE 100 // number of 32-bit words in stack
```

```
struct tcb{
  int32_t *sp;      // pointer to stack, valid for
threads not running
  struct tcb *next; // linked-list pointer
};
typedef struct tcb tcbType;
tcbType tcbs[NUMTHREADS];
tcbType *RunPt;
int32_t Stacks[NUMTHREADS][STACKSIZE];
```

Program 13.3. TCBs for up to 3 threads,

These threads have their private TCBs. They use their private registers. The three TCBs are linked in a circular list. The OS code to create three active threads and link them are as following:

```
void SetInitialStack(int i){
  tcbs[i].sp = &Stacks[i][STACKSIZE-16]; // thread stack
pointer
  Stacks[i][STACKSIZE-1] = 0x01000000; // Thumb bit
  Stacks[i][STACKSIZE-3] = 0x14141414; // R14
  Stacks[i][STACKSIZE-4] = 0x12121212; // R12
  Stacks[i][STACKSIZE-5] = 0x03030303; // R3
  Stacks[i][STACKSIZE-6] = 0x02020202; // R2
  Stacks[i][STACKSIZE-7] = 0x01010101; // R1
  Stacks[i][STACKSIZE-8] = 0x00000000; // R0
  Stacks[i][STACKSIZE-9] = 0x11111111; // R11
  Stacks[i][STACKSIZE-10] = 0x10101010; // R10
  Stacks[i][STACKSIZE-11] = 0x09090909; // R9
  Stacks[i][STACKSIZE-12] = 0x08080808; // R8
  Stacks[i][STACKSIZE-13] = 0x07070707; // R7
  Stacks[i][STACKSIZE-14] = 0x06060606; // R6
  Stacks[i][STACKSIZE-15] = 0x05050505; // R5
  Stacks[i][STACKSIZE-16] = 0x04040404; // R4
}
int OS_AddThreads(void(*task0)(void), void(*task1)(void),
        void(*task2)(void)){
int32_t status;
  status = StartCritical();
  tcbs[0].next = &tcbs[1]; // 0 points to 1
  tcbs[1].next = &tcbs[2]; // 1 points to 2
  tcbs[2].next = &tcbs[0]; // 2 points to 0
```

```
  SetInitialStack(0); 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
}
```

Program 13.4. OS code used to create three active threads.

The THUMB bit is 1 in PSR. The PC field on the stack contains the starting address of each thread. The initial values for the other registers do not matter. Even though the thread has not yet been run, it is created with an initial stack that "looks like" it had been previously suspended by a SysTick interrupt. SysTick will be used to perform the preemptive thread switching. We will set the SysTick to the lowest level so we know it will only suspend foreground threads.

OS initialization: OS_Init(void) code initialize the OS.

```
void OS_Init(void){
  DisableInterrupts();
  BSP_Clock_InitFastest();// set processor clock to
fastest speed
}
```

Program 13.5. RTOS initialization.

After initialization you need to Launch the OS with OS_launch() code. To start the RTOS, the following code arms the SysTick interrupts and unloads the stack as if it were returning from an interrupt. The units of theTimeSlice are in bus cycles. The bus cycle time on the TM4C123 is 12.5ns.

```
void OS_Launch(uint32_t theTimeSlice){
  STCTRL = 0;                  // disable SysTick during
setup
  STCURRENT = 0;               // any write to current cl
ears 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
}
```

Program 13.6. RTOS launch.

osasm.s File: This file contains the assembly code to describe the SysTick interrupt and OS start.

The **StartOS** is written in assembly as following. In this simple implementation, the first user thread is launched by setting the stack pointer to the value of the first thread, then pulling all the registers off the stack explicitly. The stack is initially set up like it had been running previously, was interrupted (8 registers pushed), and then suspended (another 8 registers pushed). When launch the first thread for the first time we do not execute a return from interrupt (we just pull 16 registers from its stack). Thus, the state of the thread is initialized and is now ready to run.

```
StartOS
    LDR     R0, =RunPt    ; currently running thread
    LDR     R1, [R0]      ; R1 = value of RunPt
    LDR     SP, [R1]      ; new thread SP; SP = RunPt->sp;
    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
```

Program 13.7. Assembly code for the thread switcher.

The SysTick ISR (SysTick_Handler) is written in assembly, performs the preemptive thread switch. Because SysTick is priority 7, it cannot preempt any background threads. This means SysTick can only suspend foreground threads. The processor automatically saves eight registers (R0-R3,R12, LR,PC and PSR) on the stack as it suspends execution of the main program and launches the ISR. 2) Since the thread switcher has read-modify-write operations to the SP and to RunPt, we need to disable interrupts to make the ISR atomic. 3) Here we explicitly save the remaining registers (R4-R11). Notice the 16 registers on the stack match exactly the order of the 16 registers established by the OS_AddThreads function. 4) Register R1 is loaded with RunPt, which points to the TCB of the thread in the process of being suspended. 5) By storing the actual SP into the sp field of the TCB, we have finished suspending the thread. To repeat, to suspend a thread we push all its registers on its stack and save its stack pointer in its TCB. 6) To implement round robin, we simply choose the next thread in the circular linked list and

update RunPt with the new value. The #4 is used because the next field is the second entry in the TCB. We will change this step later to implement sleeping, blocking, and priority scheduling. 7) The first step of launching the new thread is to establish its stack pointer. 8) We explicitly pull eight registers from the stack. 9) We enable interrupts so the new thread runs with interrupts enabled. 10) The LR contains 0xFFFFFFF9 because a main program using MSP was suspended by SysTick. The BX LR instruction will automatically pull the remaining eight registers from the stack, and now the processor will be running the new thread.

The SysTick ISR calls the C function in order to find the next thread to run. We must save R0 and LR because these registers will not be preserved by the C function. IMPORT is an assembly pseudo-op to tell the assembler to find the address of Scheduler from the linker when all the files are being stitched together. We had to save the LR before calling the function because the BL instruction uses LR to save its return address. The POP instruction restores LR to 0xFFFFFFF9. We need to push/pop an even number of registers (8-byte alignment) and functions are allowed to freely modify R0-R3, R12. For these two reasons, we also pushed and popped R0.

```
IMPORT Scheduler
SysTick_Handler                 ; 1) Saves R0-
R3,R12,LR,PC,PSR
    CPSID   I                   ; 2) Prevent interrupt dur
ing 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

PUSH    {R0,LR}
    BL      Scheduler
    POP     {R0,LR}
    LDR     R1, [R0]            ; 6) R1 = RunPt, new threa
d
    LDR     SP, [R1]            ; 7) new thread SP; SP = R
unPt->sp;
    POP     {R4-R11}            ; 8) restore regs r4-11
    CPSIE   I                   ; 9) tasks run with interr
upts enabled
    BX      LR                  ; 10) restore R0-
R3,R12,LR,PC,PSR
```

Program 13.8. Assembly code for the thread switcher with call to the scheduler written in C.

The assembly SysTick ISR call a C function, as shown in following program. The purpose of the C function is to run the scheduler and update the **RunPt** with the thread to run next. You can find this simple RTOS SysTick_Handler in RoundRobin_xxx project.

```
void Scheduler(void){
  RunPt = RunPt->next;  // Round Robin
}
```

Program 13.9. Round robin scheduler written in C.

Periodic Task:

A very appropriate feature of a RTOS is scheduling periodic tasks. If the number of periodic tasks is small, the OS can assign a unique periodic hardware timer for each task. Another simple solution is to run the periodic tasks in the scheduler. For example, assume the thread switch is occurring every 1 ms, and we wish to run the function PeriodicUserTask() every 10 ms, then we could modify the scheduler. Assume the OS initialized the counter to 0. In order for this OS to run properly, the time to execute the periodic task must be very short and always return. These periodic tasks cannot spin or block.

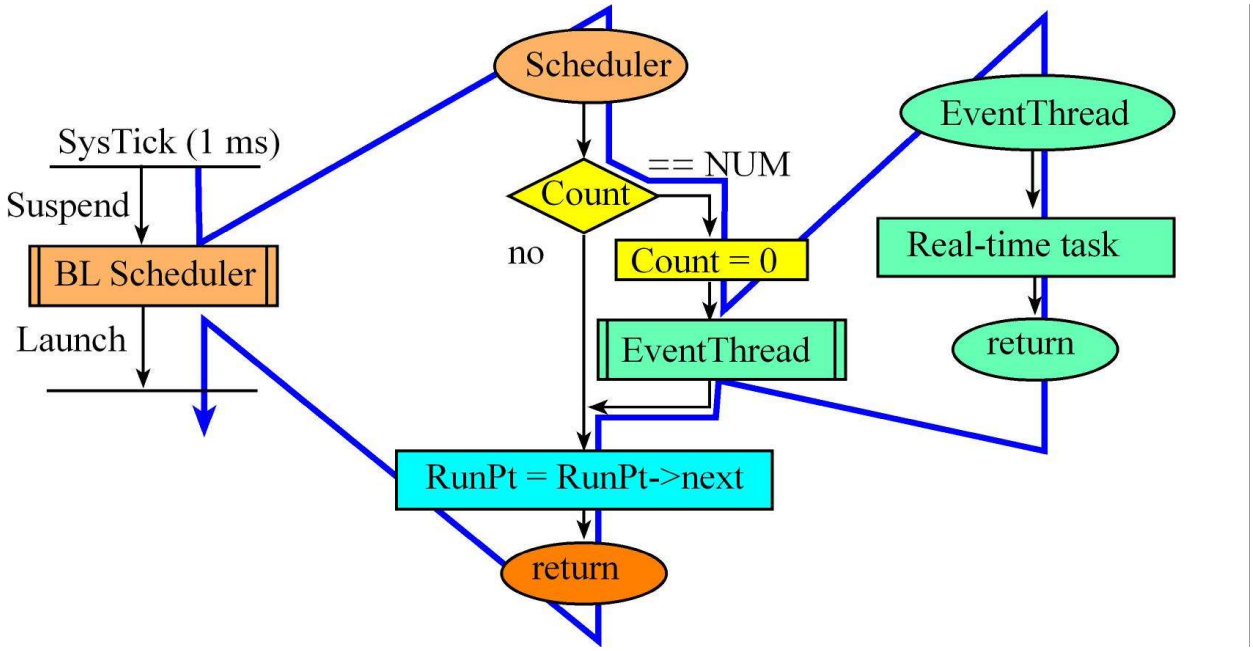

Figure 13.4. Simple mechanism to implement periodic event threads is to run them in the scheduler.

```
uint32_t Counter;
#define NUM 10
```

```
void (*PeriodicTask1)(void);  // pointer to user function
void Scheduler(void){
  if((++Counter) == NUM){
    (*PeriodicTask1)();        // runs every NUM ms
    Counter = 0;
  }
  RunPt = RunPt->next;          // Round Robin scheduler
}
```

Program 13.10. Round robin scheduler with periodic tasks.

If there are multiple real-time periodic tasks to run, then you should schedule at most one of them during each SysTick ISR execution. For example, assume the thread switch is occurring every 1 ms, and we wish to run PeriodicUserTask1() every 10 ms, and run PeriodicUserTask2() every 25 ms. In this simple approach, the period of each task must be a multiple of the thread switch period. I.e., the periodic tasks must be multiples of 1 ms. First, we find the least common multiple of 10 and 25, which is 50. We let the counter run from 0 to 49, and schedule the two tasks at the desired rates, but at non-overlapping times.

```
uint32_t Counter;
void Scheduler(void){
 Counter = (Counter+1)%50; // 0 to 49
 if((Counter%10) == 1){ // 1, 11, 21, 31 and 41
  PeriodUserTask1();
 }
 if((Counter%25) == 0){ // 0 and 25
  PeriodUserTask2();
 }
 RunPt = RunPt->next; // Round Robin scheduler
}
```

Program 13.11. Round robin scheduler with two periodic tasks.

Semaphore:

We will use semaphores to implement synchronization, sharing and communication between threads. A semaphore is a counter with three functions: OS_InitSemaphore, OS_Wait, and OS_Signal. Initialization occurs once at the start, but wait and signal are called at run time to provide synchronization between threads.

In this lab we will use the simplest implementation of semaphore called “spin-lock”. If the thread calls OS_Wait with the counter equal to zero it will “spin” (do nothing) until the counter goes above zero. Once the counter is greater than zero, the counter is decremented, and the wait function returns. In this simple implementation, the OS_Signal just increments the counter. In the context of the previous round

robin scheduler, a thread that is “spinning” will perform no useful work, but eventually will be suspended by the SysTick handler, and then other threads will execute. It is important to allow interrupts to occur while the thread is spinning so that the software does not hang. The read-modify-write operations on the counter, s, is a critical section. So the read-modify-write sequence must be made atomic, because the scheduler might switch threads in between any two instructions that execute with the interrupts enabled.

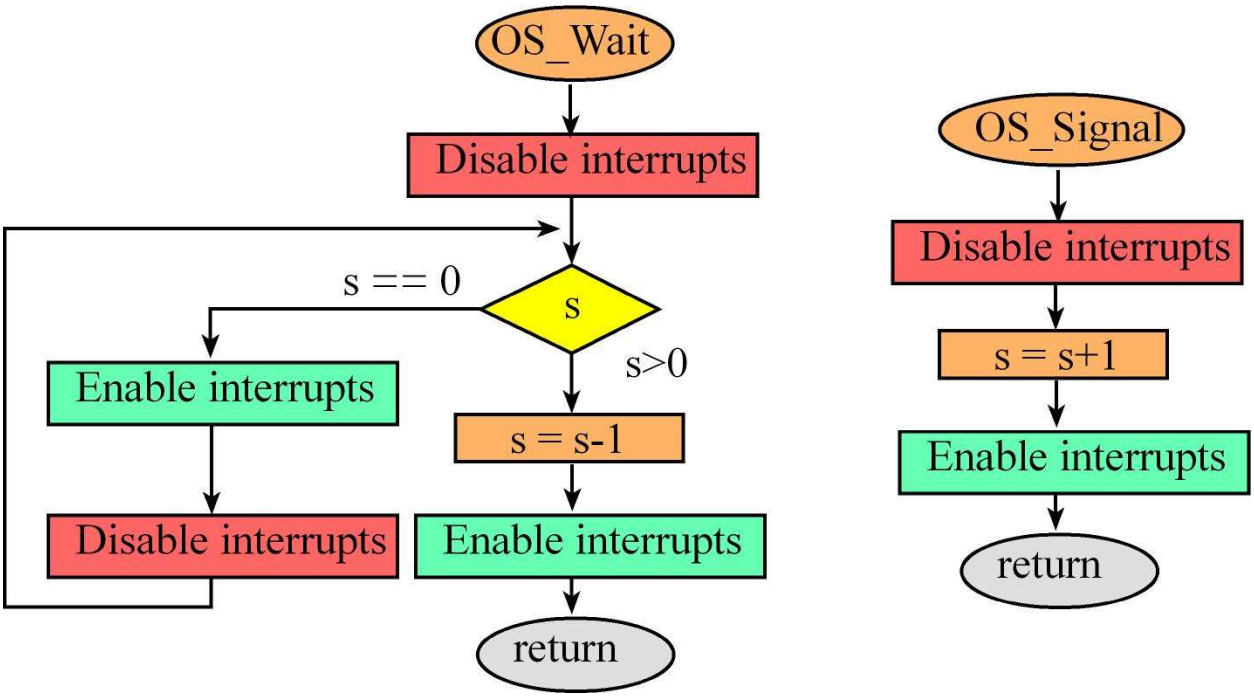

Figure 13.5. Flowcharts of a spinlock counting semaphore.

```
void OS_InitSemaphore(int32_t *s, int32_t v){

     *s = v;

}

void OS_Wait(int32_t *s){
  DisableInterrupts();
  while((*s) == 0){
    EnableInterrupts(); // interrupts can occur here
    DisableInterrupts();
  }
  (*s) = (*s) - 1;
  EnableInterrupts();
}
void OS_Signal(int32_t *s){
  DisableInterrupts();
```

```
  (*s) = (*s) + 1;
  EnableInterrupts();
}
```

Program 13.12. A spinlock counting semaphore.

Communication between two threads using a mailbox

The producer first generates data, and then it calls SendMail(). Consumer first calls RecvMail(), and then it processes the data. Mail is a shared global variable that is written by a producer thread and read by a consumer thread. In this way, data flows from the producer to the consumer. The Send semaphore allows the producer to tell the consumer that new mail is available. The Ack semaphore is a mechanism for the consumer to tell the producer, the mail was received. If Send is 0, it means the shared global does not have valid data. If Send is 1, it means the shared global does have valid data. If Ack is 0, it means the consumer has not yet read the global. If Ack is 1, it means the consumer has read the global. The sequence of operation depends on which thread arrives first. Initially, semaphores Send and Ack are both 0. Consider the case where the producer executes first.

```
uint32_t Mail;  // shared data
int32_t Send=0; // semaphore
int32_t Ack=0;  // semaphore

void SendMail(uint32_t data){
  Mail = data;
  OS_Signal(&Send);
  OS_Wait(&Ack);
}
void Producer(void){
  Init1();
  while(1){ uint32_t int myData;
    myData = MakeData();
    SendMail(myData);
    Unrelated1();
  }
}

uint32_t RecvMail(void){
uint32_t theData;
  OS_Wait(&Send);
  theData = Mail; // read mail
  OS_Signal(&Ack);
  return theData;
}
void Consumer(void){
  Init2();
  while(1){ uint32_t thisData;
    thisData = RecvMail();
    Unrelated2();
```

```
  }
}
```

Program 13.13. Semaphores used to implement a mailbox. Both Producer and Consumer are main threads.

Remember that only main threads can call OS_Wait, so the above implementation works only if both the producer and consumer are main threads. If producer is an event thread, it cannot call OS_Wait. For this scenario, we must remove the Ack semaphore and only use the Send semaphore. Initially, the Send semaphore is 0. If Send is already 1 at the beginning of the producer, it means there is already unread data in the mailbox. In this situation, data will be lost. In this implementation, the error count, Lost, is incremented every time the producer calls SendMail() whenever the mailbox is already full.

```
uint32_t Lost=0;
void SendMail(uint32_t data){
  Mail = data;
  if(Send){
    Lost++;
  }else{
    OS_Signal(&Send);
  }
}
void Producer(void){
  Init1();
  while(1){ uint32_t int myData;
    myData = MakeData();
    SendMail(myData);
    Unrelated1();
  }
}

uint32_t RecvMail(void){
  OS_Wait(&Send);
  return Mail; // read mail
}


void Consumer(void){
  Init2();
  while(1){ uint32_t thisData;
    thisData = RecvMail();
    Unrelated2();
  }
}
```

Program 13.14. Semaphores used to implement a mailbox. Producer is an event thread and Consumer is a main thread.