

**Lab 5h Real-Time Operating System**

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, 2<sup>nd</sup> edition, by Jonathan W. Valvano, published by Thomson Publishing, copyright © 2006.

- Goals**
- Develop OS facilities for real time applications,
  - Coordinate multiple foreground and background threads,
  - Design a round robin multi-thread scheduler,
  - Implement semaphores and use them for thread synchronization,
  - Implement inter-thread communication.

- Review**
- Valvano Chapter 4 on FIFO queues and interrupts,
  - Valvano Chapter 5 on Threads and semaphores,
  - Valvano Chapter 6 on periodic interrupts using output compare

- Starter files**
- Lab 17 and Lab5h projects

**Background**

Your job is to design, implement and test operating system commands that implement a multiple thread environment. In real time applications, the scheduling of software tasks is critical for the proper operation of the system. For a data acquisition system, the software must start the analog-to-digital converter (ADC) and read the result at precise time intervals. Let  $\Delta t$  be the time interval, which is one divided by the sampling rate.

$$\Delta t = 1/f_s$$

For a control system, the software must read the sensors, perform the digital control equations, then output to the actuators at a fixed rate. For a system that generates signals, the software must output to the digital-to-analog converter (DAC) at a fixed rate. We can define time-jitter,  $\delta t$ , as the difference between when a periodic task is supposed to be run, and when it is actually run. Let  $t_n$  be the time the software task is actually run, and let  $n\Delta t$  be the time it was supposed to be run, then the time-jitter at sample  $n$  is

$$\delta t_n = t_n - n\Delta t$$

For a real time system with periodic tasks, we must be able to place an upper bound,  $k$ , on the time-jitter.

$$-k \leq \delta t_n \leq +k \text{ for all } n$$

Sometimes it is more important to control the time difference between periodic events rather than the absolute time itself. Let  $\Delta t_n$  be the actual time difference between two executions of a software task (e.g., starting the ADC). The desired time difference is  $1/f_s$ . For this situation, we define the time-jitter at sample  $n$  to be

$$\delta t_n = \Delta t_n - 1/f_s$$

Again, we must be able to place an upper bound,  $k$ , on the time-jitter.

$$-k \leq \delta t_n \leq +k \text{ for all } n$$

For example, in this lab the ADC is sampled at 500 Hz. This means the ADC should be activated every 2000 $\mu$ s. This sampling rate is fixed and should not be increased or decreased. Using a **ScanPoint**, the following data shows the TCNT values measured at the first six times the ADC was sampled.

```
TCNT=16660
TCNT=24653  24653-16660=8003  jitter= 3 cycles
TCNT=32655  32655-24653=8002  jitter= 2 cycles
TCNT=40652  40652-32655=7997  jitter= -3 cycles
TCNT=48652  48652-40652=8000  jitter= 0 cycles
TCNT=56651  56651-48652=7999  jitter= -1 cycle
```

Although the difference should have been exactly 8000 cycles (2000  $\mu$ s), there was some variability in when the signal was sampled. In this experiment, the measured time-jitter was less than  $\pm 1\mu$ s. Under most situations, this error is acceptable, and we are confident to specify this system as real time.

For a real time system with input/output devices, the software latency is important. For an input device, the software latency is the time delay between when the hardware says the input is ready and the time when the software reads the data. With a simple serial port like ones on the 9S12, the software must response to an input within 10 bit times or risk an overrun error (lost data.) For an output device, the software latency is the time delay between when the hardware says the output is idle and the time when the software write new data to the device. The software latency for the output task will affect the overall bandwidth, but there is usually no hard upper bound above which the system stops working.

Not all software tasks in a real time system require execution at specified times. For example, in a data acquisition and control systems, updating visual displays or saving the results in secondary storage can often be performed when the computer is free, i.e., not needed for time critical functions.

When there are a small number of real time tasks in a system, a simple software solution can usually be found. But as the number and complexity of the tasks increase, we will need a set of OS facilities to manage time. Foreground threads will be run using a priority thread scheduler. Background threads (interrupt service routines) will be run as a result of specific hardware conditions. In particular, an output compare interrupt will be used to sample data from the ADC. A thread is a “light weight” process. Threads share resources (global memory, I/O devices) but have separate registers, stack and local variables. Typically threads cooperate to achieve a common goal. Background threads will have the highest priority and they will be used to perform the most time-critical functions. It will be important to develop debugging tools to visualize software activity. Using these tools, you will develop performance measures to evaluate system efficiency. During the first week, you will first implement multitasking and spinlock semaphores, then you will extend the thread scheduler to implement blocking semaphores and priority. In this way, important foreground tasks can be run only when there is work to be done. Priority allows the more important software tasks to be performed first.

It will be important to implement reentrant code, because multiple threads will be executing the same software. You will have to carefully consider what information is local to each thread and what is global. A nonreentrant subroutine will have a section of code called a **vulnerable window**. An error occurs if

- 1) one thread calls the nonreentrant subroutine
- 2) is executing in the “vulnerable” window when interrupted by a second thread
- 3) the second thread calls the same subroutine
- 4) control is returned to the first thread
- 5) the first thread finishes the subroutine.

A vulnerable window may exist when two different subroutines access the same memory-resident data structure. Consider the following situation: **Thread1** increments a counter, and **Thread2** decrements it. Because the compiler implements the increment and decrement using multiple instructions, the read-modify-write access is nonatomic, and hence a vulnerable window exists. In particular, if **Thread1** is interrupted by **Thread2** in the middle of a read-modify-write access to the counter, then the counter will have an incorrect value.

**short counter; // shared global**

<pre>void Thread1(void){   while(1){     counter++;   } }</pre>	<pre>void Thread2(void){   while(1){     counter--;   } }</pre>
---	---

In Lab 5, we will pass data from a background thread (**Producer**) to a foreground thread (**Consumer**) using a buffered approach, see Figure 5.1. When **Consumer** needs information, it calls **RxFifo\_Get**. If the FIFO is empty, it will spin/block on the semaphore **RxAvailable** because it can not retrieve any information. On the other hand, **Producer** enters information by calling **RxFifo\_Put**. If the FIFO is full, then it will not wait because one can not block or spin in an interrupt service routine. The spinning/blocking occurs in the **OS\_Wait** routine.

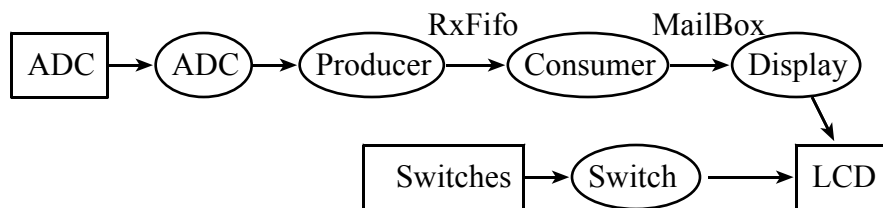


Figure 5.1. Data flow graph for the user programs.

With spinlock semaphores, a foreground thread can be in one of two states, see Figure 5.2. A foreground thread is in the **active state** if it ready to run but waiting for its turn. A foreground thread is in the **run state** if it is currently executing. With a single instruction stream computer like the 6812, at most one thread can be in the run state at a time. Therefore, the thread that is running uses the actual registers (CCR, A, B, X, Y, SP, and PC.) On the other hand, a foreground thread that is not running has its registers (CCR, A, B, X, Y, and PC) on top of its stack,

and has its stack pointer saved in its TCB. A circular linked list data structure holds the ready and active threads. Again, the background threads are the interrupt service routines, which are executed in response to specific hardware events.

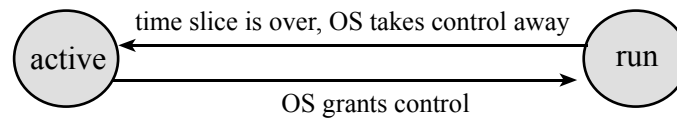


Figure 5.2. Thread state diagram.

An output compare interrupt feature will be used by your operating system (OS) to grant and take away execution control from the available active foreground threads. You will implement pre-emptive round robin scheduling. The basic steps required to switch threads is embodied in the following ten assembly instructions

#### OC3Handler

```

movb #$08,TFLG1 ; acknowledge
ldd  TC3
add  #4000      ; time slice
std  TC3
ldx  RunPt     ; pointer to TCB of running thread
sts  2,x       ; save SP into TCB
ldx  0,x       ; next thread
stx  RunPt
lds  2,x       ; restore stack of next thread
rti
  
```

The `0,x` addressing mode assumes the `next` parameter of the TCB at the first 16-bit item. The `2,x` addressing mode assumes the `saveSP` parameter of the TCB at the second 16-bit item. A second output compare channel (OC0) is used by the `Producer()` to implement the real-time data acquisition.

When passing data between two foreground threads, we could use a buffered approach (`DataFifo`). The buffered communication between two foreground threads will not be used in Lab5h; it is presented for information purposes only. When using a `DataFifo` buffer, the two counting semaphores called, `DataAvailable` and `DataRoomLeft`, contain the number of entries currently stored in the message `DataRoomLeft` and the number of spaces left in the `DataFifo` respectively. `DataAvailable` is initialized to zero, and `DataRoomLeft` is initialized to the maximum allowable number of elements in the `DataFifo`. The `Send` routine, called by the `Producer`, could execute the following steps:

```

OS_Wait(&DataRoomLeft)
Disable Interrupts
Enter data into the DataFifo structure
Enable Interrupts
OS_Signal(&DataAvailable)
  
```

The `Receive` routine, called by the `Consumer`, could execute the following steps:

```

OS_Wait(&DataAvailable)
Disable Interrupts
Remove data from the DataFifo structure
Enable Interrupts
OS_Signal(&DataRoomLeft)
  
```

A `Producer` thread creates data, and then sends the data to a consumer (calls `Send`). A `Consumer` thread receives the data from a producer (calls `Receive`). The `DataFifo` is used for interthread communication. In some applications there might be multiple producers and multiple consumers. In this lab however, we will have the simple situation of having a single producer and a single consumer. When one of the threads involved in the buffered producer-consumer communication is a background thread, we must remove the `OS_Wait` call from the ISR. This is because only foreground threads will be allowed to spin or block.

In Lab 5, we will pass data from a foreground thread (`Consumer`) to another foreground thread (`Display`) using an unbuffered approach, see Figure 5.1. A single global (called `MailBox`) will contain the data passed from

**Consumer** to **Display**. A binary semaphore, **DataValid**, is initialized to zero meaning the **MailBox** is empty. When **DataValid** equals one it means the mailbox has data in it placed by **Consumer** that has not been read by the **Display**. A second binary semaphore, **BoxFree**, is initialized to one meaning the **MailBox** is free. When **BoxFree** equals zero it means the mailbox is being used. When **Consumer** wishes to send data it first waits/blocks on the semaphore **BoxFree**. Next, **Consumer** puts its data into the **MailBox**, and sets **DataValid**. Setting **DataValid** will allow **Display** to proceed. When **Display** executes **Receive**, it first waits/blocks on the semaphore **DataValid**, gets the data from the **MailBox** and sets **BoxFree**. Setting **BoxFree** will allow **Consumer** to proceed. It doesn't matter whether the **Display** or **Consumer** executes first, they will wait for each other. The **Send** routine executes the following steps:

```
OS_bWait(&BoxFree)
Put data into the mailbox
OS_bSignal(&DataValid)
```

The **Receive** routine executes the following steps:

```
OS_bWait(&DataValid)
Retrieve the data from the mailbox
OS_bSignal(&BoxFree)
```

When we perform LCD output we will need a mechanism to share this resource. You will have to implement mutual exclusion (only one thread at a time can call the LCD output functions.) A traditional Computer Science term for this type of semaphore is **mutex**. We will call our semaphore **LCDFree**. It will prevent more than one thread from outputting at the same time. It is initialized to 1 that means there is 1 display available. When the **LCDFree** semaphore is zero, it means no displays are free (a thread is currently doing output.) Some operating systems provide special support for this true/false type of semaphore, calling it a binary semaphore. For example if you wished to output a message, then a thread could call a function like the following:

```
void Message(char letter, unsigned short data){
    OS_bWait(&LCDFree);    // capture resource
    LCD_OutChar(letter);
    LCD_OutUDec(data);
    LCD_OutChar(CR);
    OS_bSignal(&LCDFree); // release resource
}
```

Before you begin writing code for this lab, you can run an existing multithreaded system and “visualize” the execution pattern of the system. The Lab17 system has three foreground threads and two background threads. One background thread performs data acquisition (**Producer**). The other background thread uses SCI interrupts to perform serial I/O. The foreground threads are a display thread (**Consumer**), a thread calculating square roots (**Math**), and an interpreter thread (**Interpreter**). These threads are defined at compile/assembly time, and not created dynamically at run time. The Lab17 system uses spinlock semaphores to provide thread synchronization. These threads will run for a finite amount of time, then display performance measurements. There are three critical measurements in the Lab17 system. The first performance measure is time-jitter. The second measure is the number of lost data points. If the consumer is waiting for the SCI port, the **DataFifo** fills up, and the **Producer** would have no place to put the results. Lost data is particularly a problem when the **Interpreter** is being used. The last performance measurement is the number of mathematical calculations completed. This is the least important, but does give us a good indication of the efficiency of the operating system. Run the Lab17 system and observe the waveform patterns on Port T and Port M using a logic analyzer or scope. Capture the logic analyzer trace of PT2, PT1, PT0, PM4, PM3, PM2, PM1, and PM0. Read through the source code to label the significance of each of the eight debugging signals. I.e., what does it mean when each of the eight signals is high? There is a **#define DEBUG 1** definition at the top of **Lab17.C**. Change it to **#define DEBUG 0**, recompile and run the system again. This new system has all the debugging instruments removed. What conclusions can you make about the intrusiveness of these instruments?

During the first part of the lab, you will implement spinlock semaphores, and a preemptive round robin scheduler. Threads will be created dynamically, and memory for the TCB will be found by calling **malloc**. The priority field will not be used. The **OS\_sleep** function will inefficiently waste cycles by executing a delay loop. Similarly, the **OS\_kill** will spin in an infinite loop wasting time and memory. The desired endpoint of the first part is to be able to run the system to completion without losing any data, and with a small time jitter.

During the second part of the lab, you will implement blocking semaphores, and preemptive priority scheduler. Threads will be killed dynamically, and memory for the TCB will be returned by calling **free**. The priority field will be added to the TCB. The **OS\_Sleep** function will suspend a thread, so that it will not be run until the timeout has occurred. Now, the **OS\_Kill** will actually terminate a thread, removing it from the TCB list and release its memory back to the memory manager.

### First Preparation (do this before your lab period), due at the beginning of lab the week of 3/23

1) There are categories into which tasks may fall. First, there are **I/O bound** tasks, where the bandwidth (data processed each second) is limited by the I/O device. For example, in a data-entry task, it usually doesn't matter how fast the computer is, the amount of information entered into the system is limited by the input typing rate of the operator. In a similar fashion, the number of pages printed per second is usually limited by the printer speed, and not by the speed of the computer. The second category describes tasks with **fixed bandwidth**, and not limited by either software or hardware. For example, the weatherman collects temperature data every hour. Temperature measurements once an hour are all we need, so a faster ADC converter, a faster temperature sensor, or a faster computer would not enhance the performance of this system. The third category, **CPU bound**, describes tasks that are limited by the execution speed of the software. For these systems, a better software algorithm, a better compiler, and a faster computer will enhance the performance. There are three tasks in this system 1) data acquisition using **Producer Consumer** and **Display**, 2) operator interaction using **Switch**, and 3) calculations using **Math** and **LittleMath**. Look at the source code in the Lab5h starter files and categorize the type of these three tasks.

2) You will begin the design of the OS by defining the **TCB** structure. The following three items must be stored in the **TCB** for each thread:

- A pointer to the next TCB in the active list
- Current Stack Pointer, SP, for this thread (saved value when not actually running)
- Stack area for this thread

Inside the stack area, the local variables are stored. When a thread is suspended because of a time slice interrupt, the registers (CCR,A,B,X,Y,PC) are stored also on this stack. Other information, such as ID, priority, sleep state, and blocked state can be added.

3) Before designing **OS\_AddThread**, sketch the circular linked list of three TCBs that should be created after **main** calls **OS\_AddThread** three times, but before **main** calls **OS\_Launch**. Draw a second sketch of the four TCBs created after **Consumer** executes **OS\_AddThread** to create the **Display** thread. Leave room to plug in actual numbers that will be collected during the testing phase of the project.

4) Write spinlock implementations for the **OS\_InitSemaphore** **OS\_Signal** **OS\_Wait** **OS\_bSignal** and **OS\_bWait** routines. Be careful to consider critical sections.

5) Write code to implement the thread switch (an output compare interrupt service routine). Draw two rough sketches of the TCB linked list, before and after a thread switch. Leave room to plug in actual numbers that will be collected during the testing.

6) Make a development plan showing how the components of this first part will be designed and tested.

### First Part Procedure

1) One way to test the semaphores is to call **OS\_Signal** from a periodic interrupt. Place **OS\_Wait** in the body main program, then count the number of interrupts and the number of times the body of the loop is executed.

2) Implement the **OS\_AddThread** function. Using the debugger, step through the first three calls to **OS\_AddThread** and fill in the sketch of the three TCBs with real data (e.g., specify a pointer as its actual 16-bit hexadecimal value).

3) Implement and test the **OS\_Launch** function. Using the debugger, step through this function, until the first thread starts to execute. Show what the three TCBs look like when one thread is running and the other threads are active. Fill in with real data the first sketch from preparation 5). Set a breakpoint in the thread switch ISR. Single step through the ISR until the next thread is running. Fill in with real data the second sketch from preparation 5).

4) Add debugging features to facilitate visualization. A profile collects information of both time and place (when and which thread is executing). One approach to profiling can be found in Lab 17.

5) Increase the size of the **RxFifo** until no data is lost. Test the system with operator input to the **Switch** too. Take measurements for three **RxFifo** sizes and three time slices (5 runs) when no input occurs in the **Switch**. Make a table showing the three performance parameters (time-jitter, number of data points lost, number of math calculations performed) versus the **TIME\_SLICE** and the **RXFIFOSIZE**.

### First Part Checkout (show this to the TA), due in second lab period the week of 3/23

1) Run the first part of Lab 5 software system and explain the profiling data to the TA,

- 2) Be prepared to discuss the sketches you created as part of the preparation, and procedure
- 3) Discuss the TCB before and after a thread switch.
- 4) Identify inefficiencies in this implementation.

### Description of the Second Part (no preparation is required)

The basic idea of the second part is to replace the spinlock semaphores with blocking semaphores. You will also convert the round-robin scheduler to a priority scheduler. 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.) If a thread communicates with other threads then it can be blocked waiting for receive data or waiting for there to be room in the transmit buffer. Both types of blocking that will be implemented in the part of this lab. If a thread wishes to output to the display, but another thread is currently outputting, it will block. We will use a blocking semaphore to implement the sharing of the display output among multiple threads.

All of these features can be implemented by modifying **OS\_wait** and **OS\_signal**. One possible way to implement blocking semaphores is described in Chapter 5 of the book. This implementation uses linked-list data structures to hold the ready and blocked threads. You will need to create multiple blocked linked lists. In general, we will have one blocked list with each blocking semaphore. You will extend the semaphore structure to include both the semaphore value and a pointer to a TCB list containing threads that are blocked on the semaphore. The semaphore initialization should be extended to clear the linked-list of blocked threads on that semaphore. Except for the semaphore structure, everything else in the user program (**Lab5h.c**) should remain exactly the same. In this implementation, a **status** field and semaphore pointer are added to the TCB. There are other ways to implement blocking, and you are free to implement other blocking schemes.

#### New OS\_wait

- 1) Save the CCR, then disable interrupts
- 2) Decrement the semaphore counter,  $(*pt \rightarrow Value) = (*pt \rightarrow Value) - 1$
- 3) If the semaphore counter is less than zero then this thread will be blocked
  - set the **status** of this thread to blocked,
  - specify this thread is to be blocked onto the linked list of this semaphore (semaphore pointer)
  - suspend thread using **TC3=TCNT+15;**
- 4) Restore CCR

#### New Threadswitch

- 1) Save SP into TCB
- 2) If this thread is to be blocked
  - move the **TCB** of this thread from the active list to the blocked list of the specified semaphore
- 3) Find the next active thread from the active list
- 4) Acknowledge C3F, set **TC3**, and launch next thread

#### New OS\_signal

- 1) Save the CCR, then disable interrupts
- 2) Increment the semaphore counter,  $(*pt \rightarrow Value) = (*pt \rightarrow Value) + 1$
- 3) If the semaphore counter is less than or equal to zero then
  - wake up one thread from the **TCB** linked list (the one waiting the longest)
  - do not suspend execution of the thread that called **OS\_signal**
  - simply move the **TCB** of the “wakeup” thread from the blocked list to the active list
- 4) Restore interrupt status

You will implement a blocking scheduler. If multiple threads are blocked, when it is time to wakeup a thread, the OS will wakeup the one that has been blocked the longest. If a thread requests a resource that is unavailable, your system should move the thread to the appropriate blocked linked-list. Careful thought should go into when to remove a thread from the blocked list. Careful thought should go into what information should be placed into the **TCB** of each thread (e.g., register values, program counter, local variable space etc.)

### Second Part Procedure (no preparation is required)

- 1) Debug the blocking semaphore implementation.
- 2) Implement and test the **OS\_Sleep** and **OS\_Kill** functions
- 3) Run the blocking sleeping round-robin system for the same 5 cases (FIFO sizes and timeslices) as you used to make the table in first part. Make a second table showing the three performance parameters for the new system. Again take these measurements when no input occurs in the **Switch** thread.

4) Add the thread priority feature. Lower priority threads will run only if all higher priority threads are blocked or sleeping. Assign appropriate priority levels for each foreground thread.

5) Run the blocking sleeping priority system for the same 5 cases as you used to make the other two. Make a third table showing the three performance parameters for the final system. Again take these measurements when no input occurs in the **Switch** thread. Compare the performance of the three systems.

### Second Part Checkout (show this to the TA), due in second lab period the week of 3/30

- 1) Demonstrate the final system to the TA.
- 2) Show the TA where in your final system the worst case time jitter arises.
- 3) Demonstrate your method to visualize the real time execution pattern.
- 4) Discuss your results of the three tables.

### Deliverables (exact components of the lab report)

- A) Objectives (1/2 page maximum)
- B) Hardware Design (none for this lab)
- C) Software Design (printout of these software components, show only parts that are different)
  - 1) spinlock/round-robin system, part 1)
  - 2) blocking/sleeping/killing/round-robin system, part 2)
  - 3) blocking/sleeping/killing/priority system, part 2)
- D) Measurement Data
  - 1) plot of the scope window running the spinlock/round-robin system (profile data)
  - 2) plot of the scope window running the blocking/sleeping/killing/round-robin system (profile data)
  - 3) plot of the scope window running the blocking/sleeping/killing/priority system (profile data)
  - 4) the four sketches (from first preparation parts 3 and 5), with measured data collected during testing
  - 5) the three tables each showing performance measurements versus sizes of RxFifo and timeslices
- E) Analysis and Discussion (2 page maximum)

### Hints

1) You do have to use the TExaS simulator, but good debugging skills are essential. Install the licensed version of TExaS that came with the book. Get the latest version of the Lab17 and Lab5h projects from the web page. The system includes C files, H files, and TExaS simulator files. This lab may be developed on a real 9S12 or on the TExaS simulator. The starter files are configured for the simulator. To run the Lab5h programs on *a real 9S12* change the following definitions to

```
#define TEXAS 0
```

To run the systems in TExaS, you will use Metrowerks to compile, then you will import the object code into TExaS. Within Metrowerks, open the project and create an object code image (Project->Make). This creates an S19 file called **monitor.sx**, and a symbol table called **monitor.map**. If you are simulating, you start the TExaS application, then open the microcomputer file **Lab17.uc** or **Lab5h.uc**. Next, you execute **OpenS19...** and import the **monitor.sx** object code.

- 2) Make small changes and save the changes using new file names, so that when something doesn't work you can go back to a version that does work and try something new. You will have to debug this system in small very parts. A mechanism to visualize the real time execution will be helpful.
- 3) Avoid infinite loops with the interrupts disabled (a crash).
- 4) Avoid using breakpoints and single stepping on the real 9S12. Remember to use the minimally intrusive debugging techniques that you have learned. If you store data into global memory, the information should be available for viewing even after a crash or a hardware reset. (Interesting note: because the TExaS simulator models both the hardware and software, breakpoints and single stepping are appropriate in this setting. A simulator breakpoint halts both the external hardware and software.)
- 5) The compiler may allocate local variables within the OC3 handler (even if you didn't explicitly define any yourself.) This causes the data to be allocated on one stack and deallocated on another. If this is the case, put the complex software into a subroutine and call it from the ISR. Check the listing file created by the compiler.
- 6) *Look for the most recent files on the network.*
- 7) It is very essential to study the details of the entire Lab17 and Lab5h code for the OS using the TExaS simulator or by any other means before starting the lab. Please ask the instructor or the TA for help in clarifying any details you don't understand.