Objectives[edit]

- Definition of RTOS
- Interrupts, SysTick, and critical sections
- Threads and a robin preemptive scheduler
- Spin-lock semaphores
- Thread synchronization and communication

Play Video

2.1.1. Motivation[edit]

Introduction to threads

Play Video

Consider a system with one input task, one output tasks and two non I/O tasks, as shown in Figure 2.1. The non-I/O tasks are called function3 and function4. Here are two possible ways of structuring a solution to the problem. The left side of the figure shows a busy-wait solution, where a single main program runs through the tasks by checking to see if the conditions for running the task have occurred. Busy-wait solution is appropriate for problems where the execution patterns for tasks are fixed and well-known, and the tasks are tightly coupled. An alternative to busy-wait is to assign one thread per task. Interrupt synchronization is appropriate for I/O even if the execution pattern for I/O is unknown or can dynamically change at run time. The difficultly with the single-foreground multiple-background threaded solutions developed without an operating system stems from answering, "how to handle complex systems with multiple foreground tasks that are loosely coupled?" A **real-time operating system** (RTOS) with a thread scheduler allows us to run multiple foreground threads, as shown on the right side of the figure. As a programmer we simply write multiple programs that all "look" like main programs. Once we have an operating system, we write Task1, Task2, Task3, and Task4 such that each behaves like a main program. One of the features implemented in an RTOS is a **thread scheduler**, which will run all threads in a manner that satisfies the constraints of the system.

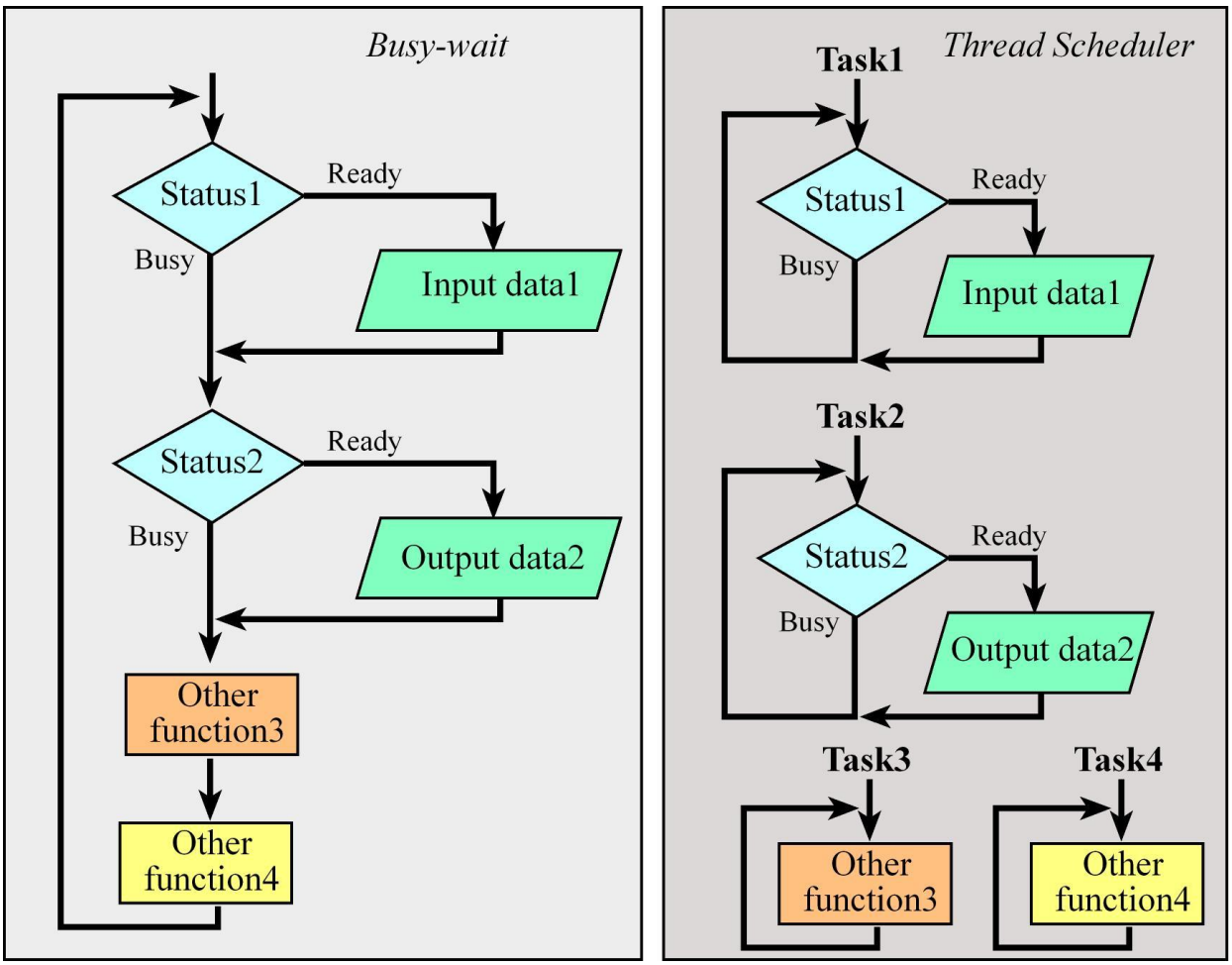


Figure 2.1. Flowcharts of a system with four loosely coupled tasks.

2.1.2. Introduction to Threads[edit]

Play Video

A **program** is a sequence of software commands connected together to affect a desired outcome. Programs perform input, make decisions, record information, and generate outputs. Programmers generate software using an editor with a keyboard and display. Programs are compiled and downloaded into the flash ROM of our microcontroller. Programs themselves are static and lifeless entities. However, when we apply power to the microcontroller, the processor executes the machine code of the programs in the ROM. A **thread** is defined as either execution itself or the action caused by the execution. Either way we see that threads are dynamic, and thus it is threads that breathe life into our systems. A thread therefore is a program in action, accordingly, in addition to the program (instructions) to execute it also has the state of the program. The thread state is captured by the current contents of the registers and the local variables, both of which are stored on the thread's stack. For example, Figure 2.2 shows a system

with four programs. We define Thread1 as the execution of Task1. Another name for thread is **light-weight process**. Multiple threads typically cooperate to implement the desired functionality of the system. We could use hardware-triggered interrupts to create multiple threads. However, in this class the RTOS will create the multiple threads that make up our system. Figure 2.2 shows the threads having separate programs. All threads do have a program to execute, but it is acceptable for multiple threads to run the same program. Since each thread has a separate stack, its local variables are private, which means it alone has access to its own local variables.

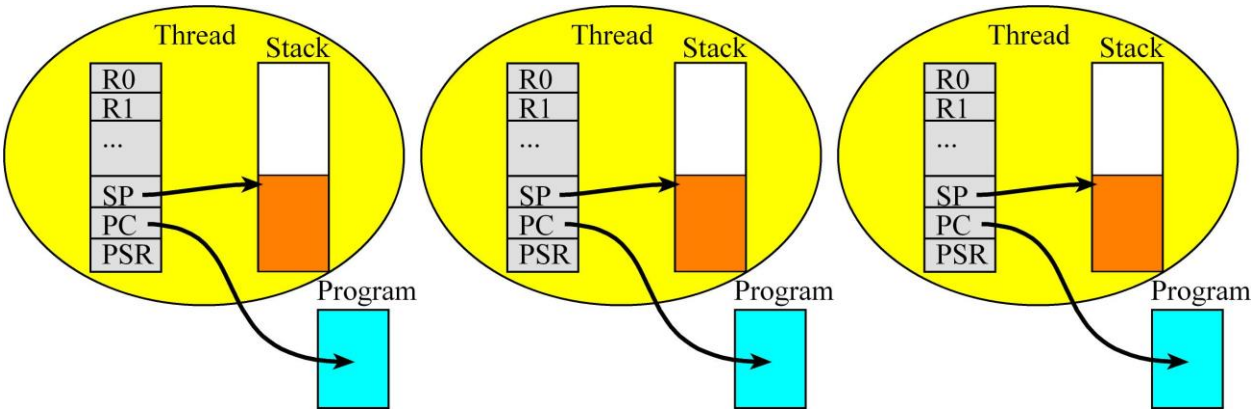


Figure 2.2. Each thread has its own registers and stack.

It looks like in Figure 2.2 that threads have physically separate registers. The stacks will be physically separate, but in reality there is just one set of registers that is switched between the threads as the thread scheduler operates. The thread switcher will suspend one thread by pushing all the registers on its stack, saving the SP, changing the SP to point to the stack of the next thread to run, then pulling all the registers off the new stack. Since threads interact for a common goal, they do share resources such as global memory, and I/O devices (Figure 2.3). However, to reduce complexity it is the best to limit the amount of sharing. It is better to use a well-controlled means to pass data and synchronize threads.

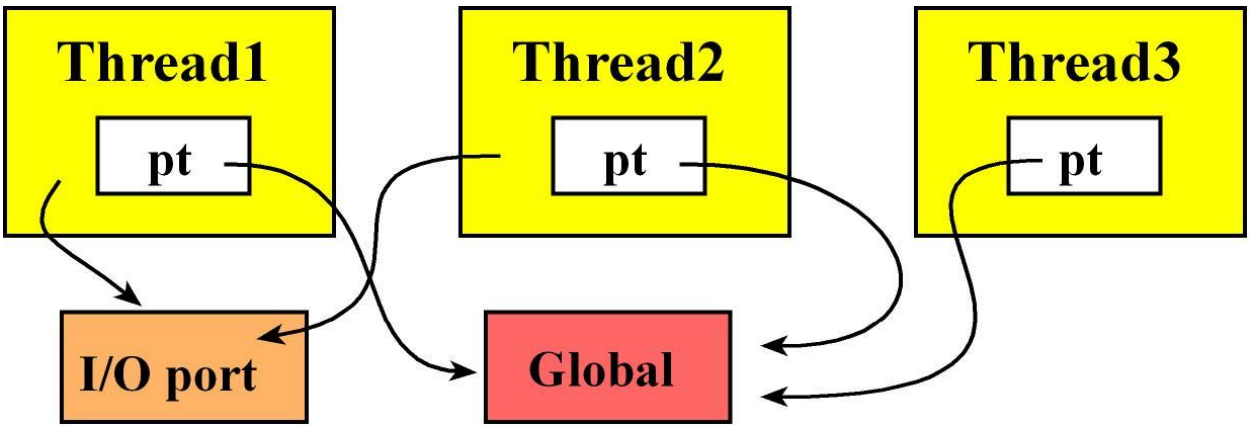


Figure 2.3. Threads share global memory and I/O ports.

Some simple examples of multiple threads are the interrupt-driven I/O. In each of these examples, the background thread (interrupt service routine) executes when the I/O device is done performing the required I/O operation. A single foreground thread (main program) executes during the times when no interrupts are needed. A global data structure is used to communicate between threads. Notice that data stored on the stack or in registers by one thread are not accessible by another thread.

2.1.3. States of a main thread[edit]

Play Video

A main thread can be in one of four states, as shown in Figure 2.4. The arrows in Figure 2.4 describe the condition causing the thread to change states. In Chapter 2, threads oscillate between the active and run states. In Chapter 2, we will create all main threads at initialization and these main threads will never block, sleep, or die.

A main thread is in the **run state** if it currently executing. On a microcontroller with a single processor like the Cortex M, there can be at most one thread running at a time. As computational requirements for an embedded system rise, we can expect microcontrollers in the future to have multicore processors, like the ones seen now in our desktop PC. For a multicore processor, there can be multiple threads in the run state.

A main thread is in the **active state** if it ready to run but waiting for its turn. In Lab 2, we will implement four threads that are either running or active.

Sometimes a main thread needs to wait for a fixed amount of time. The OS will not run a main thread if it is in the **sleep state**. 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. Sleeping will be presented in Chapter 3.

A main 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 will implement blocking in the next chapter.

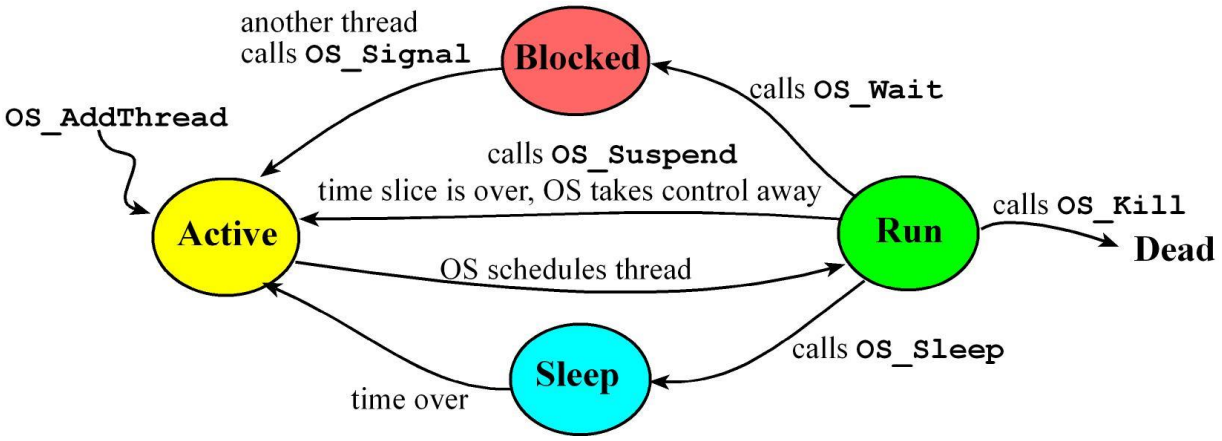


Figure 2.4. A main thread can be in one of four states.

The OS manages the execution of threads. An important situation to manage is when a thread is stuck and cannot make progress. For example, a thread may need data from another thread, a thread may be waiting on I/O, or a thread may need to wait for a specified amount of time. In Lab 3, when a thread is waiting because it cannot make progress it will **block**, meaning it will not run until the time at which it can make progress. Similarly, in Lab 3 when a thread needs to wait for a prescribed amount of time, it will **sleep**, meaning it will not run until the elapsed wait time has passed. Blocking and sleeping will free up the processor to perform actual work. In Lab 2 main threads will not block or sleep, but more simply we will **spin** if a thread is waiting on an event. A thread that is spinning remains in the active state, and wastes its entire time slice checking the condition over and over.

2.1.4. Real-time systems[edit]

Play Video

Designing a RTOS requires many decisions to be made. Therefore, it is important to have performance criteria with which to evaluate one alternative to another. A common performance criterion used in Real-Time Systems is **Deadline**, a timing constraint with many definitions in the literature. In this class we will define specific timing constraints that apply to design of embedded systems. **Bandwidth** is defined as the information rate. It specifies the amount of actual data per unit time that are input, processed, or output.

In a real-time system operations performed must meet logical correctness and also be completed on time (i.e., meet timing constraints). Non real-time systems require logical correctness but have no timing requirements. The tolerance of a real-time system towards failure to meet the timing requirements determines whether we classify it as **hard real time**, **firm real time**, or **soft real time**. If missing a timing constraint is unacceptable, we call it a hard real-time system. In a firm real-time system the value of an operation completed past its

timing constraint is considered zero but not harmful. In a soft real-time system the value of an operation diminishes the further it completes after the timing constraint.

Hard real time: For example, if the pressure inside a module in a chemical plant rises above a threshold, failure to respond through an automated corrective operation of opening a pressure valve within a timing constraint can be catastrophic. The system managing the operations in such a scenario is a hard real-time operating system.

Firm real time: An example of a firm real-time system is a streaming multimedia communication system where failure to render one video frame on time in a 30 frames per second stream can be perceived as a loss of quality but does not affect the user experience significantly.

Soft real time: An example of a soft real-time system is an automated stock trading system where excessive delay in formulating an automated response to buy/sell may diminish the monetary value one can gain from the trade. The delivery of email is usually soft real time, because the value of the information reduces the longer it takes.

Please understand that the world may not reach consensus of the definitions of hard, firm and soft. Rather than classify names to the real-time system, think of this issue is as a continuum. There is a continuous progression of the consequence of missing a deadline: catastrophic (hard) → zero effect and no harm (firm) → still some good can come from finishing after deadline (soft). Similarly: there is a continuous progression for the value of missing a deadline: negative value (hard), zero value (firm) and some but diminishing positive value (soft).

To better understand real-time systems, **timing constraints** can be classified into two types. The first type is **event-response**. The event is a software or hardware trigger that signifies something important has occurred and must be handled. The response is the system's reaction to that event. Examples of event-response tasks include:

- Operator pushes a button -> Software performs action
- Temperature is too hot -> Turn on cooling fan
- Supply voltage is too low -> Activate back up battery
- Input device has new data -> Read and process input data
- Output device is idle -> Perform another output

The specific timing constraint for this type of system is called latency, which is the time between the event and the completion of the response. Let E_i be the times that events occur in our system, and T_i be the times these events are serviced. **Latency** is defined as

$\Delta_i = T_i - E_i$ for $i = 0, 1, 2, \ldots, n\text{-}1$

where n is the number of measurements collected. The timing constraint is the maximum value for latency, Δ_i, that is acceptable. In most cases, the system will not be able to anticipate the event, so latency for this type of system will always be positive.

A second type of timing constraint occurs with prescheduled tasks. For example, we could schedule a task to run periodically. If we define f_s as the desired frequency of a periodic task, then the desired period is $\Delta t = 1/f_s$. Examples of prescheduled tasks include:

- Every 30 seconds -> Software checks for smoke
- At 22 kHz -> Output new data to DACs creating sound
- At 1 week, 1 month, 1 year -> Perform system maintenance
- At 300 Hz -> Input new data from ADC measuring EKG
- At 6 months of service -> Deactivate system because it is at end of life

For periodic, the desired time to run the i'th periodic instance of the task is given as

$$D_i = T_0 + i*\Delta t \text{ for } i = 0, 1, 2, \ldots, n\text{-}1$$

where T_0 is the starting time for the system. For prescheduled tasks, we define jitter as the difference between desired time a task is supposed to run and the actual time it is run. Let T_i be the actual times the task is run, so in this case jitter is

$$\delta t_i = T_i - D_i \text{ for } i = 0, 1, 2, \ldots, n\text{-}1$$

Notice for prescheduled tasks the jitter can be positive (late) or negative (early). For some situations running the task early is acceptable but being late is unacceptable. If I have the newspaper delivered to my door each morning, I do not care how early the paper comes, as long as it arrives before I wake up. In this case, the timing constraint is the maximum value for jitter δt_i that is acceptable. On the other hand, for some situations, it is unacceptable to be early and it is acceptable to be late. For example, with tasks involving DACs and ADCs, as shown in Figure 2.5, we can correlate voltage error in the signal to time jitter. If dV/dt is the slew rate (slope) of the voltage signal, then the voltage error (noise) caused by jitter is

$$\delta V_i = \delta t_i * dV/dt \text{ for } i = 0, 1, 2, \ldots, n\text{-}1$$

The error occurs because we typically store sampled data in a simple array and assume it was sampled at $f_s = 1/\Delta t$. In other words, we do not record exactly when the sample was actually performed.

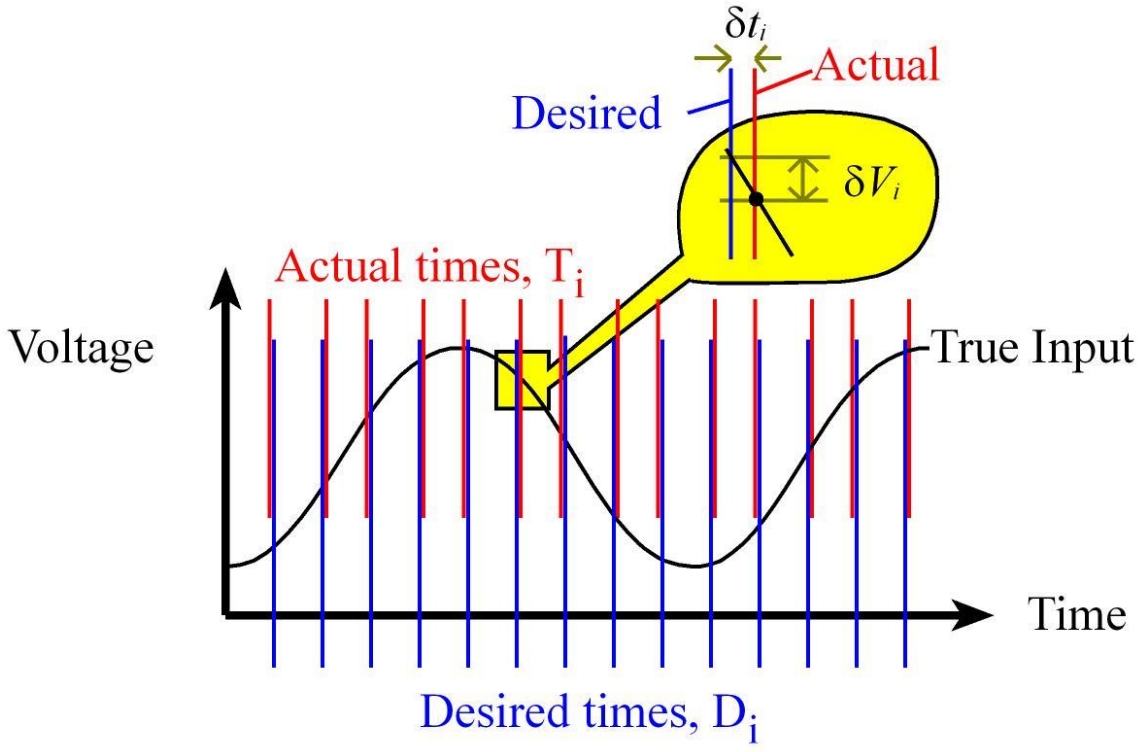


Figure 2.5. Effect of jitter on sampled data. True input is a sinusoidal. Blue lines depict when the voltage should be sampled. Red lines depict when the voltage was actually be sampled. There is time jitter such that every other sample is early and every other sample is late. In the zoomed in portion this sample is late; the consequence of being late is the actual sampled data is lowered than the correct value. Sampling jitter causes noise in the data.

For cases where the starting time, T_0, does not matter, we can simplify the analysis by looking at time differences between when the task is run, $\Delta T_i = (T_i - T_{i-1})$. In this case, jitter is simply

$\delta t_i = \Delta T_i - \Delta t$ for i = 0, 1, 2, …, n-1

We will classify a system with periodic tasks as real-time if the jitter is always less than a small but acceptable value. In other words the software task always meets its timing constraint. More specifically, we must be able to place an upper bound, k, on the time jitter.

$-k \le \delta t_i \le +k$ for all i

Previously in Lab 1, we defined a similar performance metric as

Min = minimum δt_i for all measurements i
Max = maximum δt_i for all measurements i
Jitter = *Max* - *Min* = (maximum δt_i – minimum δt_i)

In most situations, the time jitter will be dominated by the time the microcontroller runs with interrupts disabled. For lower priority interrupts, it is also affected by the length and frequency of higher priority interrupt requests.

To further clarify this situation we must clearly identify the times at which the T_i measurements are collected. We could define this time as when the task is started or when the task is completed. When sampling an ADC the important time is when the ADC sampling is started. More specifically, it is the time the ADC sample/hold module is changed from sample to hold mode. This is because the ADC captures or latches the analog input at the moment the sample/hold is set to hold. For tasks with a DAC, the important time is when the DAC is updated. More specifically, it is the time the DAC is told to update its output voltage.

2.1.5. Producer/Consumer problem[edit]

Play Video

One of the classic problems our operating system must handle is communication between threads. We define a **producer** thread as one that creates or produces data. A **consumer** thread is a thread that consumes (and removes) data. The communication mechanism we will use in this chapter is a mailbox (Figure 2.6). The mailbox has a *Data* field and a *Status* field. Mailboxes 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 task. The advantage of using a structure like a mailbox for a data flow problem is that we can decouple the producer and consumer threads. In chapter 3, we will replace the mailbox with a first in first out (FIFO) queue. The use of a FIFO can significantly improve system performance.

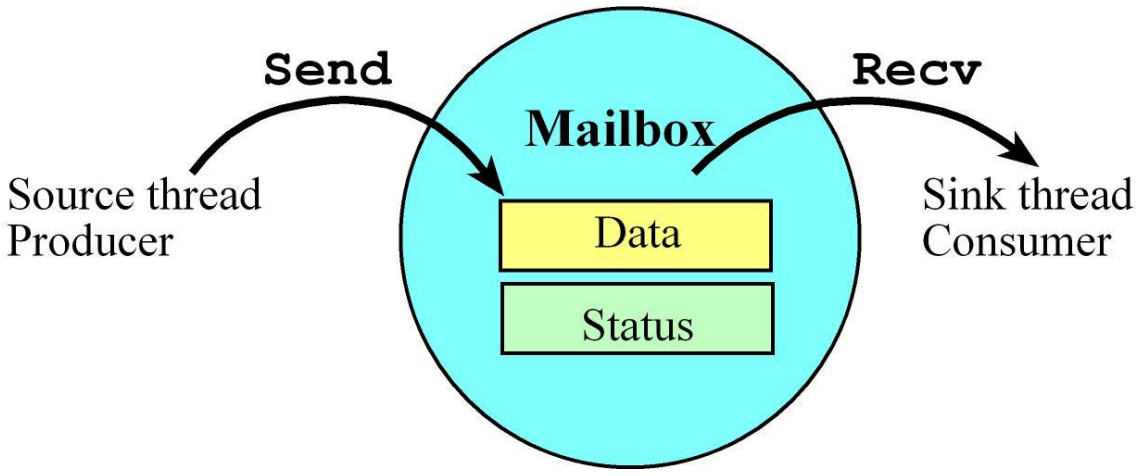


Figure 2.6. The mailbox is used to send data from the producer thread to the consumer thread.

There are many producer/consumer applications in the field of embedded systems. In Table 2.1 the threads on the left are producers that create data, while the threads on the right are consumers that process data.

Source/Producer	Sink/Consumer
Keyboard input	Program that interprets
Software that has data	Printer output
Software sends message	Software receives message
Microphone and ADC	Software that saves sound data
Software that has sound data	DAC and speaker

Table 2.1. Producer consumer examples.

Figure 2.7 shows how one could use a mailbox to pass data from a background thread (interrupt service routine) to a foreground thread (main program) if there were no operating system.

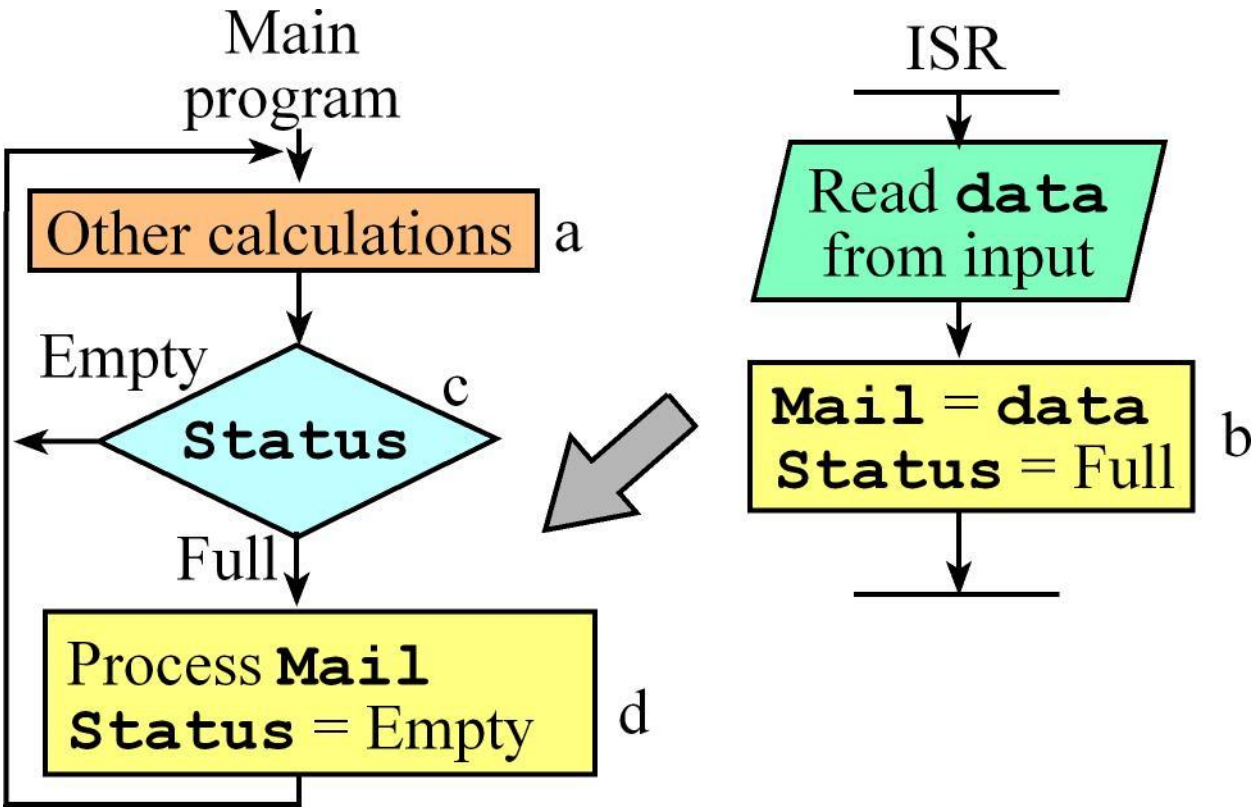


Figure 2.7. Use of a mailbox without an operating system.

2.1.6. Scheduler[edit]

Play Video

A **scheduler** is a OS function that gives threads the notion of **Concurrent processing** where multiple threads are active. If we look from a distance (zoom out in time) it appears they are running simultaneously, when in fact only one thread is running at any time. On the Cortex-M with one processor only a single thread can run at any given time while other ready threads contend for processing. The scheduler therefore runs the ready threads one by one, switching between them to give us the illusion that all are running simultaneously.

In this class, the OS will schedule both main threads and event threads. However, in this section we will discuss scheduling main threads. To envision a scheduler, we first list the main threads that are ready to run. When the processor is free, the scheduler will choose one main thread from the ready list and cause it to run. In a **preemptive scheduler**, main threads are suspended by a periodic interrupt, the scheduler chooses a new main thread to run, and the return from interrupt will launch this new thread. In this situation, the OS itself decides when a running thread will be suspended, returning it to the active state. In Program 2.1, there exist four threads as illustrated in Figure 2.8. The preemptive scheduler in the RTOS runs the four main threads concurrently. In reality, the threads are run one at time in sequence.

```
void Task1(void){
  Init1();
  while(1){
    if(Status1())
      Input1();
  }
}
```

```
void Task2(void){
  Init2();
  while(1){
    if(Status2())
      Output2();
  }
}
```

```
void Task3(void){
  Init3();
  while(1){
    function3();
  }
}
```

Program 2.1. Four main threads run concurrently using a preemptive scheduler.

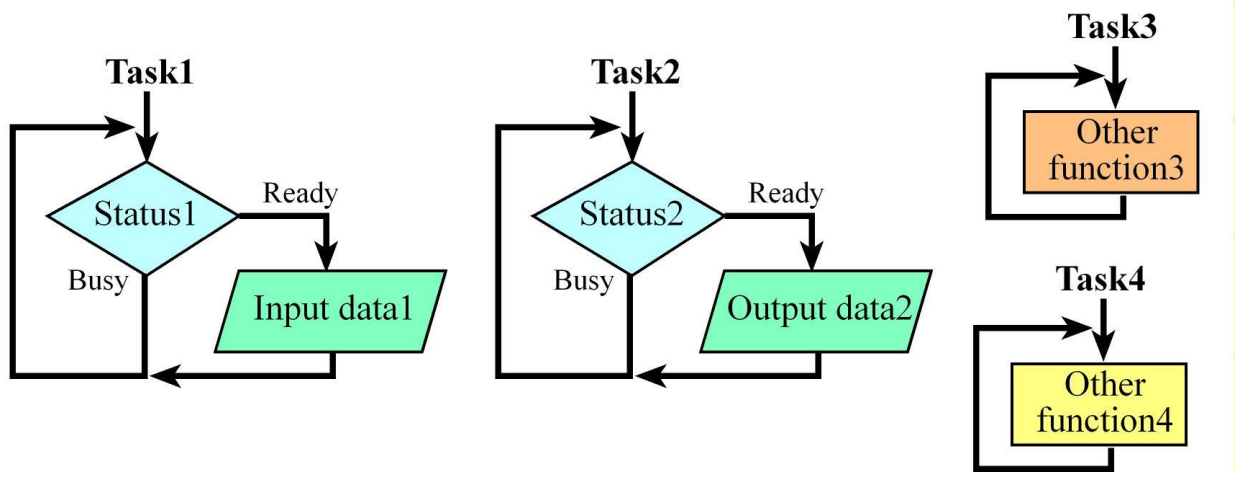


Figure 2.8. Four main threads.

In a **cooperative** or **nonpreemptive** scheduler, the main threads themselves decide when to stop running. This is typically implemented by having a thread call a function like OS_Suspend. This function will suspend the running thread (putting the old thread in the Active state), run the scheduler (which chooses a new thread), and launch the new thread. The new thread is now in the Run state. Although easy to implement because it doesn't require interrupts, a cooperative scheduler is not appropriate for real-time systems. In Program 2.2, the cooperative scheduler runs the four main threads in a cyclic manner.

```
void Task1(void){
 Init1();
 while(1){
  if(Status1()){
   Input1();
  }
  OS_Suspend();
 }
}
```

```
void Task2(void){
 Init2();
 while(1){
  if(Status2()){
   Output2();
  }
 OS_Suspend();
 }
}
```

```
void Task3(void){
 Init3();
 while(1){
   function3();
   OS_Suspend();
 }
}
```

Program 2.2. Four threads run in a cooperative manner.

There are many scheduling algorithms one can use to choose the next thread to run. A **round robin scheduler** simply runs the ready threads in circular fashion, giving each the same amount of time to execute. A **weighted round robin scheduler** runs the ready threads in circular fashion, but gives threads unequal weighting. One way to implement weighting is to vary the time each thread is allowed to run according to its importance. Another way to implement weighting is to run important threads more often. E.g., assume there are three threads 1 2 3, and thread 1 is more important. We could run the threads in this repeating pattern: 1, 2, 1, 3, 1, 2, 1, 3… Notice that very other time slice is given to thread 1. In this simple example, Thread 1 receives 50% of the processor time, and threads 2 and 3 each receive 25%. A **priority scheduler** assigns each thread a priority number (e.g., 1 is the highest). Two or more threads can have the same priority. A priority 2 thread is run only if no priority 1 threads are ready to run. Similarly, we run a priority 3 thread only if no priority 1 or priority 2 threads are ready. If all threads have the same priority, then the scheduler reverts to a round-robin system. The advantage of priority is that we can reduce the latency (response time) for important tasks by giving those tasks a high priority. The disadvantage is that on a busy system, low priority threads may never be run. This situation is called **starvation**.

Schedulers for real-time systems may use other metrics to decide thread importance/priority. A **deadline** is when a task should complete relative to when it is ready to run. The **time-to-deadline** is the time between now and the deadline. If you have a paper due on Friday, and it is Tuesday, the time-to-deadline is 3 days. Furthermore, we define **slack time** as the time-to-deadline minus the how long it will take to complete the task. If you have a paper due on Friday, it is Tuesday and it will take you one day to write the paper, your slack time is 2 days. Once the

slack time becomes negative, you will miss your deadline. There are many other ways to assign priority:

- Minimize latency for real-time tasks
- Assign a dollar cost for delayed service and minimize cost
- Give priority to I/O bound tasks over CPU bound tasks
- Give priority to tasks that need to run more frequently
- Smallest time-to-deadline first
- Smallest slack time first

A thread's priority may be statically assigned or can be changed dynamically as the system progresses. An **exponential queue** is a dynamic scheduling algorithm, with varying priorities and time slices. If a thread blocks on I/O, its priority is increased and its time slice is halved. If it runs to completion of a time slice, its priority is decreased and its time slice is doubled.

Another dynamic scheduling algorithm uses the notion of **aging** to solve starvation. In this scheme, threads have a permanent fixed priority and a temporary working priority. The permanent priority is assigned according the rules of the previous paragraph, but the temporary priority is used to actually schedule threads. Periodically the OS increases the temporary priority of threads that have not been run in a long time. Once a thread is run, its temporary priority is reset back to its permanent priority.

Assigning priority to tasks according to how often they are required to run (their periodicity) is called a **Rate Monotonic Scheduler**. Assume we have m tasks that are periodic, running with periods Tj ($0 \le j \le m$-1). We assign priorities according to these periods with more frequent tasks having higher priorities. Furthermore, let Ej be the maximum time to execute each task. Assuming there is little interaction between tasks, the **Rate Monotonic Theorem** can be used to predict if a scheduling solution exists. Tasks can be scheduled if

∑0≤j≤m−1EjTj≤m∗(21m−1)andlim

What this means is, as long as the total utilization of the set of tasks is below 69.32% (ln(2) ≈ 0.6932) RMS will guarantee to meet all timing constraints. The practical application of the Rate Monotonic Theorem is extremely limited because most systems exhibit a high degree of coupling between tasks. Nevertheless, it does motivate a consideration that applies to all real-time operating systems. Let E_j be the time to execute each task, and let T_j be the time between executions of each task. In general, E_j/T_j will be the percentage of time Task j needs to run. The sum of these percentages across all tasks yields a parameter that estimates processor **utilization**.

$$AverageUtilization \equiv \sum_{0 \le j \le m-1} \text{ave } E_j \text{ave } T_j \; MaximumUtilization \equiv \sum_{0 \le j \le m-1} \max E_j T_j$$

If utilization is over 100% there will be no solution. If utilization is below 5%, the processor may be too fast for your problem. The solution could be to slow down the clock and save power. As the sum goes over 50% and begins to approach 100%, it will be more and more difficult to

schedule all tasks. The solution will be to use a faster processor or simplify the tasks. An effective system will operate in the 5 to 50% range.

2.1.7. Function Pointers[edit]

Play Video

As we work our way towards constructing an OS there are some advanced programming concepts we require the reader to be familiar with. One such concept is "**function pointers**". Normally, when a software in module A wishes to invoke software in module B, module A simply calls a function in module B. The function in module B performs some action and returns to A. At this point, typically, this exchange is complete. A **callback** is a mechanism through which the software in module B can call back a preset function in module A at a later time. Another name for callback is **hook**. To illustrate this concept, let module A be the user code and module B be the operating system. To setup a callback, we first write a user function (e.g., **CallMe**), and then the user calls the OS passing this function as a parameter.

```
int count;
void CallMe(void){
  count++;
}
```

The OS immediately returns to the user, but at some agreed upon condition, the OS can invoke a call back to the user by executing this function.

As we initialize the operating system, the user code must tell the OS a list of tasks that should be run. More specifically, the user code will pass into the operating system pointers to user functions. In C on the Cortex M, all pointers are 32-bit addresses regardless of the type of pointer. A function pointer is simply a pointer to a function. In this class all tasks or threads will be defined as void-void functions, like **CallMe**. In other words, threads take no inputs and return no output. There are three operations we can perform on function pointers. The first is declaring a function pointer variable. Just like other pointers, we specify the type and add * in front of the name. We think it is good style to include **p**, **pt**, or **ptr** in pointer names. In this case, the syntax looks like this

```
void (*TaskPt)(void);
```

Although the above line looks a little bit like a prototype, we can read this declaration as **TaskPt** is a pointer to a function that takes no input and returns no output.

Just like other variables, we need to set its value before using it. To set a function pointer we assign it a value of the proper type. In this case, **TaskPt** is a pointer to a void-void function, so we assign it the address of a void-void function by executing this code at run time.

```
TaskPt = &CallMe; // TaskPt points to CallMe
```

Just like other pointers (to variables), to access what a pointer is pointing to, we dereference it using *. In this case, to run the function we execute

```
*TaskPt(); // call the function to which it points
```

As an example, let's look at one of the features in the BSP package. The function **BSP_PeriodicTask_Init** will initialize a timer so a user function will run periodically. Notice the user function is called from inside the interrupt service routine.

```
void (*PeriodicTask)(void); // user function
void BSP_PeriodicTask_Init(
  void(*task)(void), // user function
  uint32_t freq,     // frequency in Hz
  uint8_t priority){ // priority
// . . . PeriodicTask = task; // user function
// . . .
}
void T32_INT1_IRQHandler(void){
  TIMER32_INTCLR1 = 0x00000001; // acknowledge interrupt
  (*PeriodicTask)();            // execute user task
}
```

The user code creates a void-void function and calls **BSP_PeriodicTask_Init** to attach this function to the periodic interrupt:

```
BSP_PeriodicTask_Init(&checkbuttons, 10, 2);
```

You will NOT use **BSP_PeriodicTask_Init** in Lab 2, but will add it within Lab 3. However you will need to understand function pointers to implement Lab 2.

Another application of function pointers is a **hook**. A hook is an OS feature that allows the user to attach functions to strategic places in the OS. Examples of places we might want to place hooks include: whenever the OS has finished initialization, the OS is running the scheduler, or whenever a new thread is created. To use a hook, the user writes a function, calls the OS and passes a function pointer. When that event occurs, the OS calls the user function. Hooks are extremely useful for debugging.

2.2.1. NVIC[edit]

Play Video

On the ARM Cortex-M processor, exceptions include resets, software interrupts and hardware interrupts. Each exception has an associated 32-bit vector that points to the memory location

where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of memory. Program 2.3 shows the first few vectors as defined in the **startup_TM4C123.s** file for the TM4C123 and the **startup_msp432.s** file for the MSP432. DCD is an assembler pseudo-op that defines a 32-bit constant. ROM location 0x0000.0000 has the initial stack pointer, and location 0x0000.0004 contains the initial program counter, which is called the reset vector. It holds the address of a function called the reset handler, which is the first thing executed following reset. There are hundreds of possible interrupt sources and their 32-bit vectors are listed in order starting with location 0x0000.0008. From a programming perspective, we can attach ISRs to interrupts by writing the ISRs as regular assembly subroutines or C functions with no input or output parameters and editing the **startup_TM4C123.s** or **startup_msp432.s** file to specify those functions for the appropriate interrupt. In this class, we will write our ISRs using standard function names so that the startup files need not be edited. For example, we will simply name the ISR for SysTick periodic interrupt as **SysTick_Handler**. The ISR for this interrupt is a 32-bit pointer located at ROM address 0x0000.003C. Because the vectors are in ROM, this linkage is defined at compile time and not at run time. After the first 16 vectors, each processor will be different so check the data sheet.

```
        EXPORT  __Vectors
__Vectors                               ; address    ISR
        DCD     StackMem + Stack        ; 0x00000000 Top of Stack
        DCD     Reset_Handler           ; 0x00000004 Reset Handler
        DCD     NMI_Handler             ; 0x00000008 NMI Handler
        DCD     HardFault_Handler       ; 0x0000000C Hard Fault Handler
        DCD     MemManage_Handler       ; 0x00000010 MPU Fault Handler
        DCD     BusFault_Handler        ; 0x00000014 Bus Fault Handler
        DCD     UsageFault_Handler      ; 0x00000018 Usage Fault Handler
        DCD     0                       ; 0x0000001C Reserved
        DCD     0                       ; 0x00000020 Reserved
        DCD     0                       ; 0x00000024 Reserved
        DCD     0                       ; 0x00000028 Reserved
        DCD     SVC_Handler             ; 0x0000002C SVCall Handler
        DCD     DebugMon_Handler        ; 0x00000030 Debug Monitor Handler
        DCD     0                       ; 0x00000034 Reserved
        DCD     PendSV_Handler          ; 0x00000038 PendSV Handler
        DCD     SysTick_Handler         ; 0x0000003C SysTick Handler
```

Program 2.3. Software syntax to set the interrupt vectors for the first 16 vectors on the Cortex M processor.

Table 2.2 lists the interrupt sources we will use on the TM4C123 and Table 2.3 shows similar interrupts on the MSP432. Interrupt numbers 0 to 15 contain the faults, software interrupts and SysTick; these interrupts will be handled differently from interrupts 16 to 154.

Vector address	Number	IRQ	ISR name in Startup.s	NVIC p
0x00000038	14	-2	PendSV_Handler	SYS_PI

0x0000003C	15	-1	SysTick_Handler	SYS_PI
0x000001E0	120	104	WideTimer5A_Handler	NVIC_I

Table 2.2. Some of the interrupt vectors for the TM4C (goes to number 154 on the M4).

Vector address	Number	IRQ	ISR name in Startup.s	NVI
0x00000038	14	-2	PendSV_Handler	SYS
0x0000003C	15	-1	SysTick_Handler	SYS
0x000000A4	41	25	T32_INT1_IRQHandler	NVI

Table 2.3. Some of the interrupt vectors for the MSP432 (goes to number 154 on the M4).

Interrupts on the Cortex-M are controlled by the Nested Vectored Interrupt Controller (NVIC). To activate an interrupt source we need to set its priority and enable that source in the NVIC. SysTick interrupt only requires arming the SysTick module for interrupts and enabling interrupts on the processor (I=0 in the **PRIMASK**). Other interrupts require additional initialization. In addition to arming and enabling, we will set bit 8 in the **NVIC_EN3_R** to activate **WideTimer5A** interrupts on the TM4C123. Similarly, we will set bit 25 in the **NVIC_ISER0** to activate **T32_INT1** interrupts on the MSP432. This activation is in addition to the arm and enable steps.

Each interrupt source has an 8-bit priority field. However, on the TM4C123 and MSP432 microcontrollers, only the top three bits of the 8-bit field are used. This allows us to specify the interrupt priority level for each device from 0 to 7, with 0 being the highest priority. The priority of the SysTick interrupt is found in bits 31 – 29 of the **SYS_PRI3** register. Other interrupts have corresponding priority registers. The interrupt number (number column in Tables 2.2 and 2.3) is loaded into the **IPSR** register when an interrupt is being serviced. The servicing of interrupts does not set the I bit in the **PRIMASK**, so a higher priority interrupt can suspend the execution of a lower priority ISR. If a request of equal or lower priority is generated while an ISR is being executed, that request is postponed until the ISR is completed. In particular, those devices that need prompt service should be given high priority.

Figure 2.10 shows the context switch from executing in the foreground to running a SysTick periodic interrupt. The I bit in the **PRIMASK** is 0 signifying interrupts are enabled. Initially, the interrupt number (**ISRNUM**) in the **IPSR** register is 0, meaning we are running in **Thread mode** (i.e., the main program, and not an ISR). Handler mode is signified by a nonzero value in **IPSR**. When **BASEPRI** register is zero, all interrupts are allowed and the **BASEPRI** register is not active.

When a SysTick interrupt is triggered, the current instruction is finished. (a) Eight registers are pushed on the stack with **R0** on top. These registers are pushed onto the stack using whichever stack pointer is active: either the **MSP** or **PSP**. (b) The vector address is loaded into the **PC** ("Vector address" column in Tables 2.2 and 2.3). (c) The **IPSR** register is set to 15 ("Number" column in Tables 2.2 and 2.3) (d) The top 24 bits of **LR** are set to 0xFFFFFF, signifying the processor is executing an ISR. The bottom eight bits specify how to return from interrupt.

0xE1 Return to Handler mode MSP (using floating point state)
0xE9 Return to Thread mode MSP (using floating point state)
0xED Return to Thread mode PSP (using floating point state)
0xF1 Return to Handler mode MSP
0xF9 Return to Thread mode MSP ← we will mostly be using this one
0xFD Return to Thread mode PSP

After pushing the registers, the processor always uses the main stack pointer (**MSP**) during the execution of the ISR. Events b, c, and d can occur simultaneously.

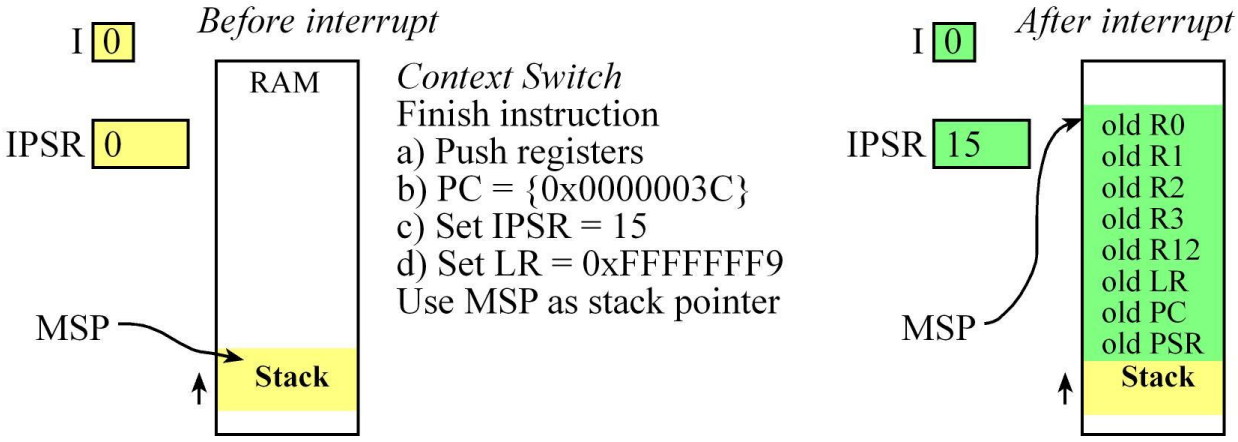


Figure 2.10. Stack before and after an interrupt, in this case a SysTick periodic interrupt.

To **return from an interrupt**, the ISR executes the typical function return statement: **BX LR**. However, since the top 24 bits of **LR** are 0xFFFFFF, it knows to return from interrupt by popping the eight registers off the stack. Since the bottom eight bits of LR in this case are 0b11111001, it returns to thread mode using the **MSP** as its stack pointer. Since the **IPSR**is part of the **PSR** that is popped, it is automatically reset to its previous state.

A **nested interrupt** occurs when a higher priority interrupt suspends an ISR. The lower priority interrupt will finish after the higher priority ISR completes. When one interrupt preempts another, the **LR** is set to 0xFFFFFFF1, so it knows to return to handler mode. **Tail chaining** occurs when one ISR executes immediately after another. Optimization occurs because the eight registers need not be popped only to be pushed once again. If an interrupt is triggered and is in the process of stacking registers when a higher priority interrupt is requested, this late arrival interrupt will be executed first.

On the Cortex-M4, if an interrupt occurs while in the floating point state, an additional 18 words are pushed on the stack. These 18 words will save the state of the floating point processor. Bits 7-4 of the LR will be 0b1110 (0xE), signifying it was interrupted during a floating point state. When the ISR returns, it knows to pull these 18 words off the stack and restore the state of the floating point processor. We will not use floating point in this class.

Priority determines the order of service when two or more requests are made simultaneously. Priority also allows a higher priority request to suspend a lower priority request currently being processed. Usually, if two requests have the same priority, we do not allow them to interrupt each other. NVIC assigns a priority level to each interrupt trigger. This mechanism allows a higher priority trigger to interrupt the ISR of a lower priority request. Conversely, if a lower priority request occurs while running an ISR of a higher priority trigger, it will be postponed until the higher priority service is complete.

2.2.2. SysTick periodic interrupts[edit]

Play Video

The SysTick Timer is a core device on the Cortex M architecture, which is most commonly used as a periodic timer. When used as a periodic timer one can setup the countdown to zero event to cause an interrupt. By setting up an initial reload value the timer is made to periodically interrupt at a predetermined rate decided by the reload value. Periodic timers as an interfacing technique are required for data acquisition and control systems, because software servicing must be performed at accurate time intervals. For a data acquisition system, it is important to establish an accurate sampling rate. The time in between ADC samples must be equal (and known) in order for the digital signal processing to function properly. Similarly, for microcontroller-based control systems, it is important to maintain both the input rate of the sensors and the output rate of the actuators. Periodic events are so important that most microcontrollers have multiple ways to generate periodic interrupts.

In this class our operating system will use periodic interrupts to schedule threads.

Assume we have a 1-ms periodic interrupt. This means the interrupt service routine (ISR) is triggered to run 1000 times per second. Let Count be a global variable that is incremented inside the ISR. Figure 2.11 shows how to use the interrupt to run Task 1 every *N* ms and run Task 2 every *M* ms.

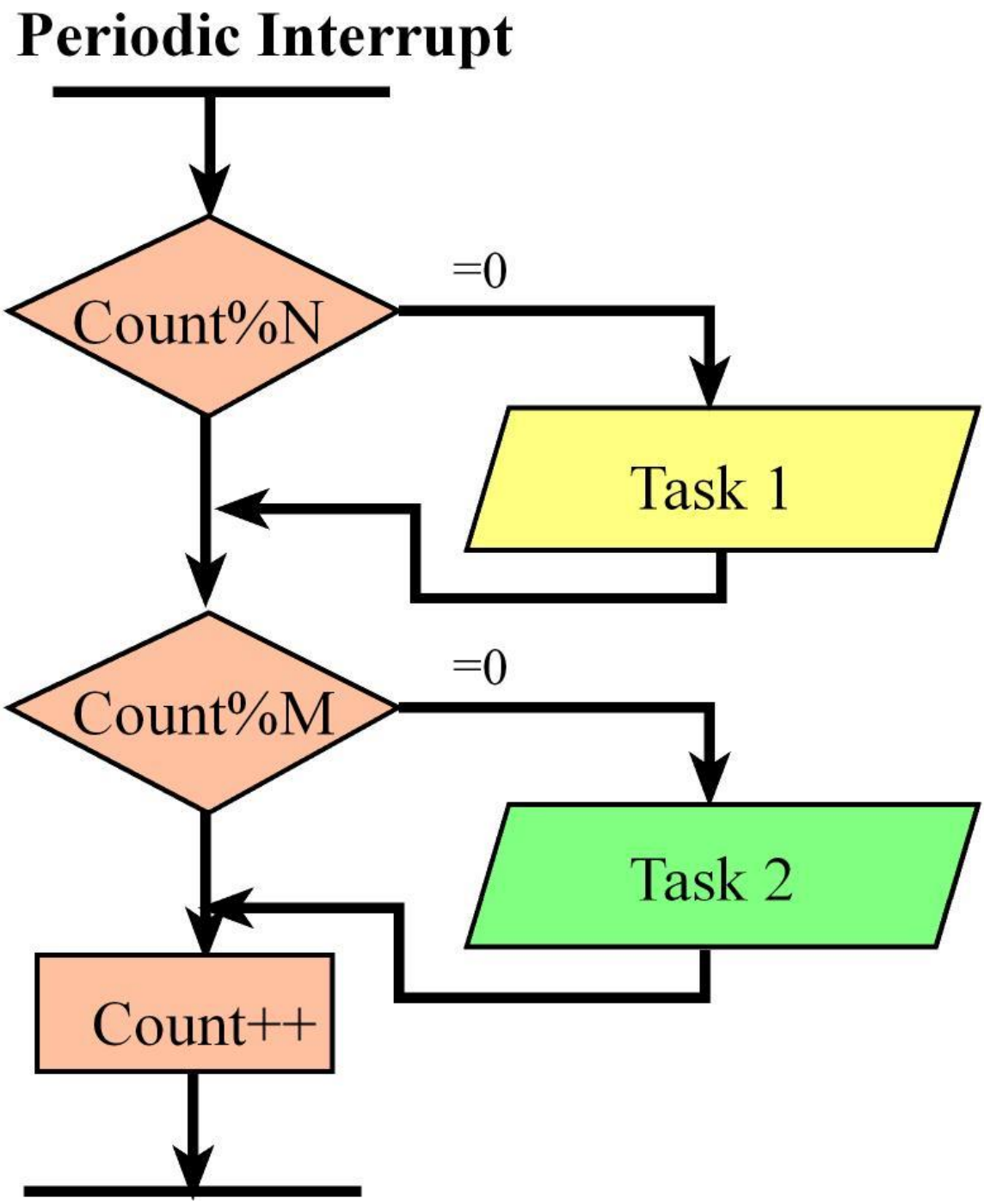


Figure 2.11. Using a 1-ms periodic interrupt to run Task 1 every N ms and run Task 2 every M ms.

The SysTick timer exists on all Cortex-M microcontrollers, so using SysTick means the system will be easy to port to other microcontrollers. Table 2.4 shows the register definitions for SysTick. The basis of SysTick is a 24-bit down counter that runs at the bus clock frequency. To configure SysTick for periodic interrupts we first clear the ENABLE bit to turn off SysTick during initialization, see Program 2.4. Second, we set the **STRELOAD** register. Third, we write any value

to the **STCURRENT**, which will clear the counter and the flag. Lastly, we write the desired clock mode to the control register **STCTRL**, also setting the **INTEN** bit to enable interrupts and enabling the timer (**ENABLE**). We establish the priority of the SysTick interrupts using the TICK field in the **SYSPRI3** register. When the **STCURRENT** value counts down from 1 to 0, the COUNT flag is set. On the next clock, the **STCURRENT** is loaded with the **STRELOAD** value. In this way, the SysTick counter (**STCURRENT**) is continuously decrementing. If the **STRELOAD** value is *n*, then the SysTick counter operates at modulo n+1:

…n, n-1, n-2 … 1, 0, n, n-1, …

In other words, it rolls over every *n*+1 counts. Thus, the **COUNT** flag will be configured to trigger an interrupt every *n*+1 counts. The main program will enable interrupts in the processor after all variables and devices are initialized.

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	STCTRL
$E000E014	0	24-bit RELOAD value						STRELOAD
$E000E018	0	24-bit CURRENT value of SysTick counter						STCURRENT

Address	31-29	28-24	23-21	20-8	7-5	4-0	Name
$E000ED20	TICK	0	PENDSV	0	DEBUG	0	SYSPRI3

Table 2.4. SysTick registers.

The SysTick counter decrements every bus cycle. So it is important to know the bus frequency when using SysTick. TM4C123 projects run at 16 MHz until the system calls **BSP_Clock_InitFastest**, at which time it will run at 80 MHz. MSP432 projects run at 3 MHz until the system calls **BSP_Clock_InitFastest**, at which time it will run at 48 MHz. In general, if the period of the core bus clock is *t* time units, then the **COUNT** flag will be set every (*n*+1)*t* time units. Reading the **STCTRL** control register will return the **COUNT** flag in bit 16, and then clear the flag. Also, writing any value to the **STCURRENT** register will reset the counter to zero and clear the **COUNT** flag. The **COUNT** flag is also cleared automatically as the interrupt service routine is executed.

Let fBUS be the frequency of the bus clock, and let *n* be the value of the **STRELOAD** register. The frequency of the periodic interrupt will be

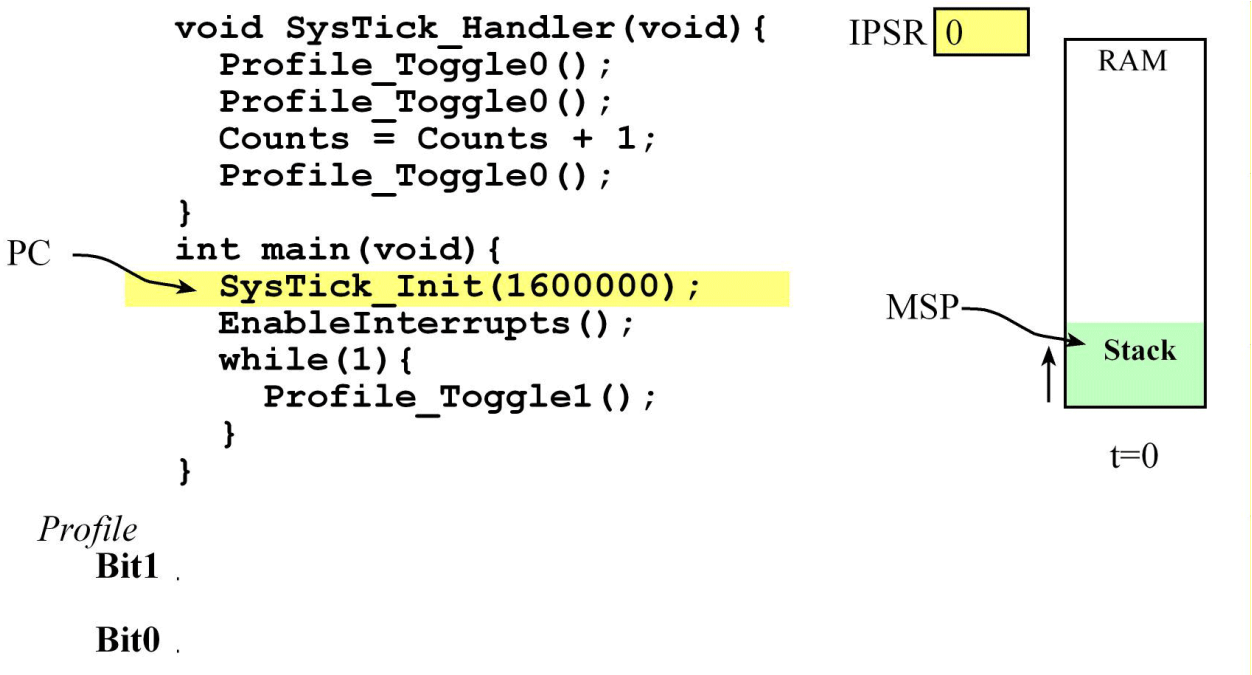
fBUSn+1

```
void SysTick_Init(uint32_t period){
  Profile_Init();
  Counts = 0;
  STCTRL = 0;          // disable SysTick during setup
  STRELOAD = period-1;// reload value
  STCURRENT = 0;       // any write to current clears it
  SYSPRI3 = (SYSPRI3&0x00FFFFFF)|0x40000000; // priority 2
```

```
          STCTRL = 0x07;      // enable, core clock, interrupts
}
// Interrupt service routine
// Executed every (bus cycle)*(period)
void SysTick_Handler(void){
  Profile_Toggle0();          // toggle bit
  Profile_Toggle0();          // toggle bit
  Counts = Counts + 1;
  Profile_Toggle0();          // toggle bit
}
int main(void){  // TM4C123 with bus clock at 16 MHz
  SysTick_Init(1600000);      // initialize SysTick timer
  EnableInterrupts();
  while(1){                   // interrupts every 100ms, 5 Hz flash
    Profile_Toggle1();        // toggle bit
  }
 }
```

Program 2.4. Implementation of a periodic interrupt using SysTick (SysTickInts_xxx).

Play Video



Question: If the bus clock is 80 MHz (12.5ns), what reload value yields a 100 Hz (10ms) periodic interrupt?

Answer: (80MHz/100Hz - 1) =**799999**. 10ms = (**799999**+1)*12.5ns. Reload should be **799999**.

Figure 2.12 shows a zoomed in view of the profile pin measured during one execution of the SysTick ISR. The first two toggles signify the ISR has started. The time from second to third toggle illustrates the body of the ISR takes 1.2 µs of execution time.

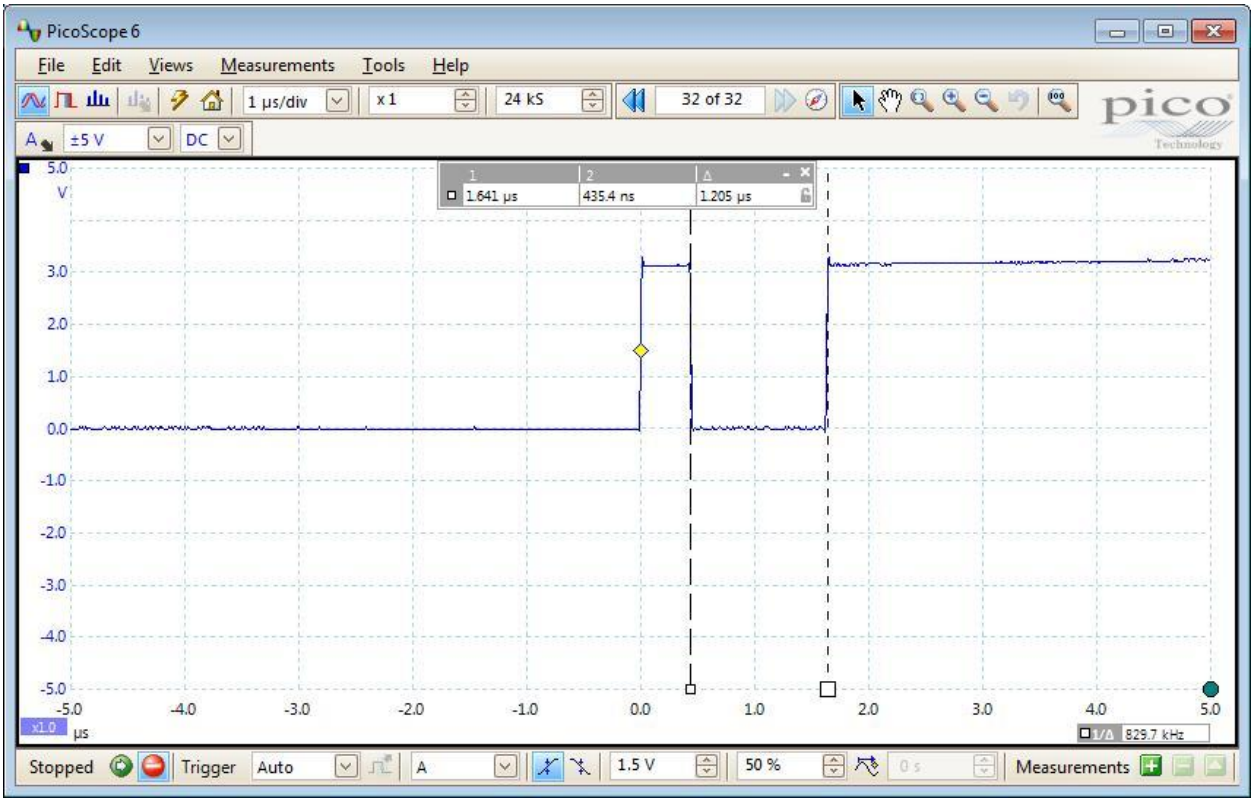


Figure 2.12. Profile of a single execution of the SysTick ISR measured on a TM4C123 running at 16 MHz.

Figure 2.13 shows a zoomed out view of the profile pin measured during multiple executions of the SysTick ISR. This measurement verifies the ISR runs every 100 ms. Because of the time scale, the three toggles appear as a single toggle. This **triple-toggle technique** (TTT) allows us to measure both the time to execution of one instance of the ISR and to measure the time between ISR executions.

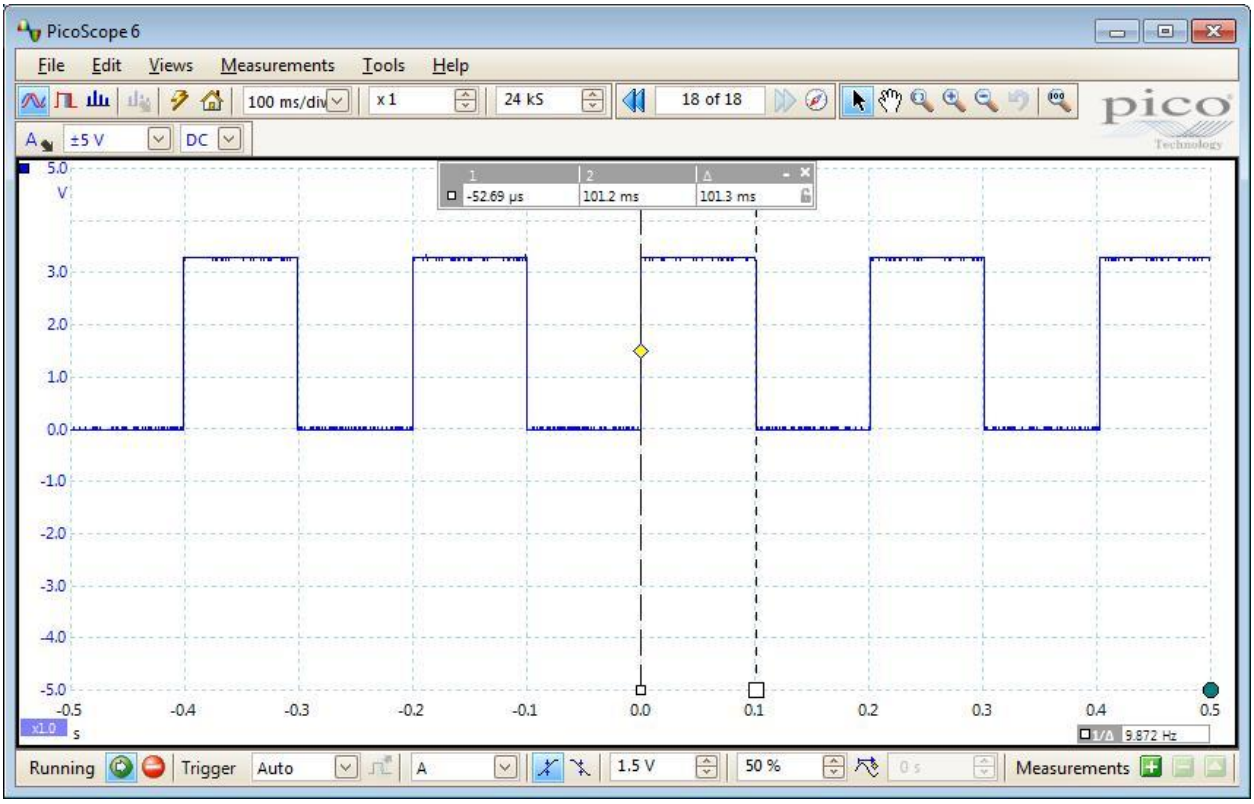


Figure 2.13. Profile of multiple executions of the SysTick ISR on a TM4C123 running at 16 MHz.

2.2.3. Critical sections[edit]

Play Video

An important consequence of multi-threading is the potential for the threads to manipulate (read/write) a shared object. With this potential comes the possibility of inconsistent updates to the shared object. A **race condition** occurs in a multi-threaded environment when there is a causal dependency between two or more threads. In other words, different behavior occurs depending on the order of execution of two threads. Consider a simple example of a race condition occurring where two thread initialize the same port in an unfriendly manner. Thread-1 initializes Port 4 bits 3 – 0 to be output using P4DIR = 0x0F; Thread-2 initializes Port 4 bits 6 – 4 to be output using **P4DIR** = 0x70; In particular, if Thread-1 runs first and Thread-2 runs second, then Port 4 bits 3 – 0 will be set to inputs. Conversely, if Thread-2 runs first and Thread-1 runs second, then Port 4 bits 6 – 4 will be set to inputs. This is a race condition caused by unfriendly code. The solution to this problem is to write the two initializations in a friendly manner, and make both initializations atomic.

In a second example of a race condition, assume two threads are trying to get data from the same input device. Both call the input function to receive data from the input device. When data arrives at the input, the thread that executes first will capture the data.

In general, if two threads access the same global memory and one of the accesses is a write, then there is a **causal dependency** between the execution of the threads. Such dependencies when not properly handled cause unpredictable behavior where the execution order may affect the outcome. Such scenarios are referred to as **race conditions**. While shared global variables are important in multithreaded systems because they are required to pass data between threads, they result in complex behavior (and hard to find bugs). Therefore, a programmer must pay careful attention to avoid race conditions.

A program segment is **reentrant** if it can be concurrently executed by two (or more) threads. Note that, to run concurrently means both threads are ready to run though only one thread is currently running. To implement reentrant software, we place variables in registers or on the stack, and avoid storing into global memory variables. When writing in assembly, we use registers, or the stack for parameter passing to create reentrant subroutines. Typically, each thread will have its own set of registers and stack. A non-reentrant subroutine will have a section of code called a **vulnerable window** or **critical section**. A critical section may exist when two different functions access and modify the same memory-resident data structure. An error occurs if

1. One thread calls a non-reentrant function
2. It is executing in the critical section when interrupted by a second thread
3. The second thread calls the same non-reentrant function.

There are a number of scenarios that can happen next. In the most common scenario, the second thread is allowed to complete the execution of the function, control is then returned to the first thread, and the first thread finishes the function. This first scenario is the usual case with interrupt programming. In the second scenario, the second thread executes part of the critical section, is interrupted and then re-entered by a third thread, the third thread finishes, the control is returned to the second thread and it finishes, lastly the control is returned to the first thread and it finishes. This second scenario can happen in interrupt programming if the second interrupt has higher priority than the first. Program 2.5 shows two C functions and the corresponding assembly codes. These functions have critical sections because of their read-modify-write nonatomic access to the global variable, count. If an interrupt were to occur just before or just after the **ADD** or **SUB** instruction, and the ISR called the other function, then count would be in error.

```
count    SPACE  4
Producer LDR  r1,[pc,#116] ; R0= &count
         LDR  r0,[r1]      ; R0=count
         ADD  r0,r0,#1
         STR  r0,[r1]      ; update
```

```
int32_t v
void Prod
  // othe
  count =
    // ot
```

```
        BX   lr
Consumer LDR  r1,[pc,#96]  ; R0= &count
        LDR  r0,[r1]       ; R0=count
        SUB  r0,r0,#1
        STR  r0,[r1]       ; update
        BX   lr
        DCD  num
```

```
}
void Cons
  // othe
  count =
  // othe
}
```

Program 2.5. These functions are nonreentrant because of the read-modify-write access to a global. The critical section is just before and just after the ADD and SUB instructions.

Assume there are two concurrent threads, where the main program calls **Producer** and a background ISR calls **Consumer**. Concurrent means that both threads are ready to run. Because there is only one computer, exactly one thread will be running at a time. Typically, the operating system switches execution control back and forth using interrupts. There are two places in the assembly code of **Producer** at which if an interrupt were to occur and the ISR called the **Consumer** function, the end value of count will be inconsistent. Assume for this example count is initially 4. An error occurs if:

1. The main program calls **Producer**
2. The main executes **LDR r0,[r1]**
    - making R0 = 4
3. The OS suspends the main (using an interrupt) and starts the ISR
4. The ISR calls **Consumer**
    - Executes **count=count-1;** making **count** equal to 3
5. The OS returns control back to the main program
    - R0 is back to its original value of 4
6. The producer finishes (adding 1 to **R0**) Making **count** equal to 5

The expected behavior with the producer and consumer executing once is that count would remain at 4. However, the race condition resulted in an inconsistency manifesting as a lost consumption. As the reader may have observed, the cause of the problem is the non-atomicity of the read-modify-write operation involved in reading and writing to the count (**count=count+1** or **count=count-1**) variable. An **atomic operation** is one that once started is guaranteed to finish. In most computers, once an assembly instruction has begun, the instruction must be finished before the computer can process an interrupt. The same is not the case with C instructions which themselves translate to multiple assembly instructions. In general, nonreentrant code can be grouped into three categories all involving 1) nonatomic sequences, 2) writes and 3) global variables. We will classify I/O ports as global variables for the consideration of critical sections. We will group registers into the same category as local variables because each thread will have its own registers and stack.

The first group is the **read-modify-write** sequence:

1. The software reads the global variable producing a copy of the data
2. The software modifies the copy (original variable is still unmodified)

3. The software writes the modification back into the global variable.

In the second group, we have a **write followed by read**, where the global variable is used for temporary storage:

1. The software writes to the global variable (only copy of the information)
2. The software reads from the global variable expecting the original data to be there.

In the third group, we have a **non-atomic multi-step write** to a global variable:

1. The software writes part of the new value to a global variable
2. The software writes the rest of the new value to a global variable.

Observation: When considering reentrant software and vulnerable windows we classify accesses to I/O ports the same as accesses to global variables.

Observation: Sometimes we store temporary information in global variables out of laziness. This practice is to be discouraged because it wastes memory and may cause the module to not be reentrant.

Sometimes we can have a critical section between two different software functions (one function called by one thread, and another function called by a different thread). In addition to above three cases, a **non-atomic multi-step** read will be critical when paired with a **multi-step write**. For example, assume a data structure has multiple components (e.g., hours, minutes, and seconds). In this case, the write to the data structure will be atomic because it occurs in a high priority ISR. The critical section exists in the foreground between steps 1 and 3. In this case, a critical section exists even though no software has actually been reentered.

Foreground thread Background thread
1. The main reads some of the data

2. ISR writes to the data st

3. The main reads the rest of the data
In a similar case, a **non-atomic multi-step write** will be critical when paired with a **multi-step read**. Again, assume a data structure has multiple components. In this case, the read from the data structure will be atomic because it occurs in a high priority ISR. The critical section exists in the foreground between steps 1 and 3.

Foreground thread Background
1. The main writes some of the data

2. ISR reads

3. The main writes the rest of the data Background thread
When multiple threads are active, it is possible for two threads to be executing the same program. For example, the system may be running in the foreground and calls a function. Part

way through execution of the function, an interrupt occurs. If the ISR also calls the same function, two threads are simultaneously executing the function.

If critical sections do exist, we can either eliminate them by removing the access to the global variable or implement mutual exclusion, which simply means only one thread at a time is allowed to execute in the critical section. In general, if we can eliminate the global variables, then the subroutine becomes reentrant. Without global variables there are no "vulnerable" windows because each thread has its own registers and stack. Sometimes one must access global memory to implement the desired function. Remember that all I/O ports are considered global. Furthermore, global variables are necessary to pass data between threads. Program 2.6 shows four functions available in the starter projects for this class that can be used to implement mutual exclusion. The code is in the startup file and the prototypes are in the **CortexM.h**file.

```
;*********** DisableInterrupts ***************
; disable interrupts
; inputs:  none
; outputs: none
DisableInterrupts
        CPSID  I
        BX     LR
;*********** EnableInterrupts ***************
; disable interrupts
; inputs:  none
; outputs: none
EnableInterrupts
       CPSIE  I
       BX     LR
;*********** StartCritical ************************
; make a copy of previous I bit, disable interrupts
; inputs:  none
; outputs: previous I bit
StartCritical
        MRS    R0, PRIMASK  ; save old status
        CPSID  I            ; mask all (except faults)
        BX     LR
;*********** EndCritical ************************
; using the copy of previous I bit, restore I bit to previous value
; inputs:  previous I bit
; outputs: none
EndCritical
        MSR    PRIMASK, R0
        BX     LR
```

Program 2.6. Assembly functions needed for interrupt enabling and disabling.

A simple way to implement mutual exclusion is to disable interrupts while executing the critical section. It is important to disable interrupts for as short a time as possible, so as to minimize the

effect on the dynamic performance of the other threads. While we are running with interrupts disabled, time-critical events like power failure and danger warnings cannot be processed. The assembly code of Program 2.6 is in the startup file in our projects that use interrupts. Program 2.7 illustrates how to implement mutual exclusion and eliminate the critical section.

```
uint32_t volatile count;
void Producer(void){   // simple option
  DisableInterrupts();
  count = count + 1;
  EnableInterrupts();
}
void Producer(void){   // safer option
long sr;
  sr = StartCritical();
  count = count + 1;
  EndCritical(sr);
}
```

Program 2.7. This function is reentrant because of the read-modify-write access to the global is atomic. Use the simple option only if one critical section is not nested inside another critical section.

When making code atomic with this simple method, make sure one critical section is not nested inside another critical section.

Play Video

2.3.1. Two types of threads[edit]

Play Video

A fundamental concept in Operating Systems is the notion of an execution context referred to as a **thread**. We introduced threads and their components in section 2.1.2, we will now look at the types of threads and how they are treated differently in the OS.

We define two types of threads in our simple OS. **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 1), 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. For example, threads could use a FIFO to pass data.

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

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. Here in chapter 2, we will specify all the main threads at initialization and these threads will exist indefinitely. However, in later chapters we will allow main threads to be created during execution, and we will allow main threads to be destroyed dynamically.

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

Table 2.5 compares event and main threads. For now, main threads will run indefinitely, but later in the class we will allow main threads to be terminated if their task is complete. Also, in Chapter 2 we will create all the main threads statically at the time the OS launches. In subsequent chapters we will allow the user to create main threads dynamically at run time.

Event Thread	*Main Thread*
Triggered by hardware	Created when OS launch
Must return	Runs indefinitely
Short execution time	Unbounded execution ti

No waiting	Allowed to wait
Finite number of loops (definite)	Indefinite or infinite loop

Table 2.5. Comparison of event and main threads.

2.3.2. Thread Control Block (TCB)[edit]

Play Video

Figure 2.14 shows three threads. 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.

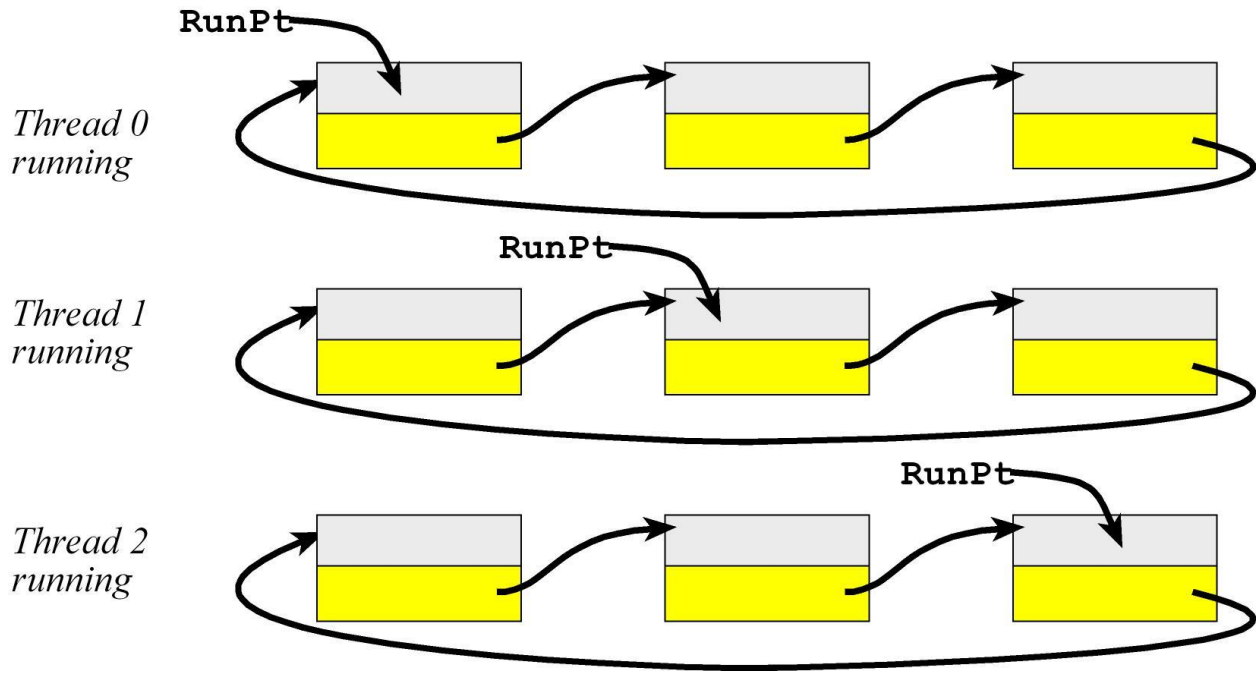


Figure 2.14. Three threads have their TCBs in a circular linked list.

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 2.15). Program 2.8 shows a TCB structure with the necessary components for three threads:

1. A pointer so it can be chained into a linked list
2. The value of its stack pointer

In addition to these necessary components, the TCB might also contain:

3. Status, showing resources that this thread has or wants
4. A sleep counter used to implement sleep mode
5. Thread number, type, or name
6. Age, or how long this thread has been active
7. Priority (not used in a round robin scheduler)

```
#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 2.8. TCBs for up to 3 threads, each stack is 400 bytes.

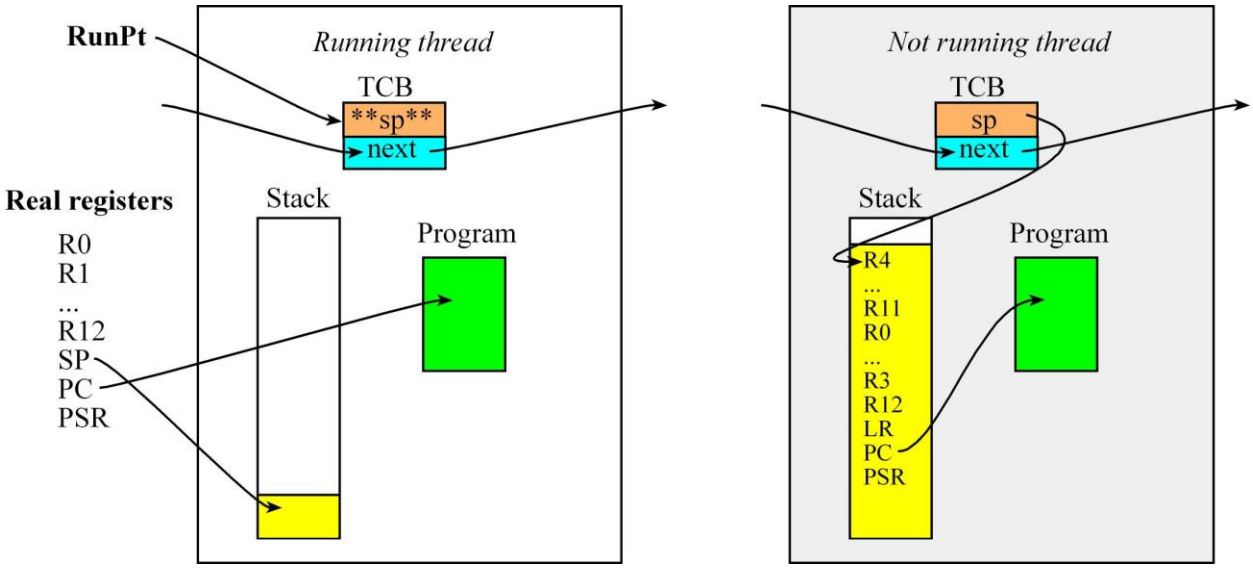


Figure 2.15. The running thread uses the actual registers, while the other threads have their register values saved on the stack. For the running thread the sp field is not valid, while the sp field on other threads points to the top of its stack.

2.3.3. Creation of threads[edit]

Play Video

Program 2.9 shows how to create three TCBs that will run three programs. First, the three TCBs are linked in a circular list. Next the initial stack for each thread is created in such a way that it looks like it has been running already and has been previously suspended. The PSR must have the T-bit equal to 1 because the Arm Cortex M processor always runs in Thumb mode. The PC field on the stack contains the starting address of each thread. The initial values for the other registers do not matter, so they have been initialized to values that will assist in debugging. This idea came from the **os_cpu_c.c** file in Micrium µC/OS-II. The allocation of the stack areas must be done such that the addresses are double-word aligned.

```
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 2.9. OS code used to create three active threads.

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. Notice that the initial value loaded into the PSR when the thread runs for the first time has T=1.Program 2.10 shows simple user software that can be run on this RTOS. Each thread increments a counter and toggles an output pin. The three counters should be approximately equal. Profile bit 0 toggles quickly while thread 0 is running. Profile bits 1 and 2 toggle when running threads 1 and 2 respectively.

```
void Task0(void){
  Count0 = 0;
  while(1){
    Count0++;
    Profile_Toggle0(); // toggle bit
  }
}
void Task1(void){
  Count1 = 0;
  while(1){
    Count1++;
    Profile_Toggle1(); // toggle bit
  }
}
void Task2(void){
  Count2 = 0;
  while(1){
    Count2++;
    Profile_Toggle2(); // toggle bit
  }
}
#define THREADFREQ 500 // frequency in Hz
int main(void){ // testmain2
  OS_Init(); // initialize, disable interrupts
  Profile_Init(); // enable digital I/O on profile pins
  OS_AddThreads(&Task0, &Task1, &Task2);
  OS_Launch(BSP_Clock_GetFreq()/THREADFREQ); // interrupts enabled in here
  return 0; // this never executes
}
```

Program 2.10. Example user code with three threads.

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 (Program 2.11).

```
void OS_Init(void){
  DisableInterrupts();
```

```
  BSP_Clock_InitFastest();// set processor clock to fastest speed
}
```

Program 2.11. RTOS initialization.

2.3.4. Launching the OS[edit]

Play Video

To start the RTOS, we write code that arms the SysTick interrupts and unloads the stack as if it were returning from an interrupt (Program 2.12). The units of **theTimeSlice** are in bus cycles. The bus cycle time on the TM4C123 is 12.5ns, and on the MSP432 the bus cycle time is 20.83ns.

```
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
}
```

Program 2.12. RTOS launch.

The **StartOS** is written in assembly (Program 2.13). 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 (address of RunPt)
    LDR     R1, [R0]      ; R1 = value of RunPt, pointing to thread control block
(tcb)
    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 2.13. Assembly code for the thread switcher.

2.3.5. Switching theads[edit]

Play Video

The SysTick ISR, written in assembly, performs the preemptive thread switch (Program 2.14). SysTick interrupts will be triggered at a fixed rate (e.g., every 2 ms in this example. Because SysTick is priority 7, it cannot preempt any background threads. This means SysTick can only suspend foreground threads. 1) 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.

```
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
    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
```

Program 2.14. Assembly code for the thread switcher.

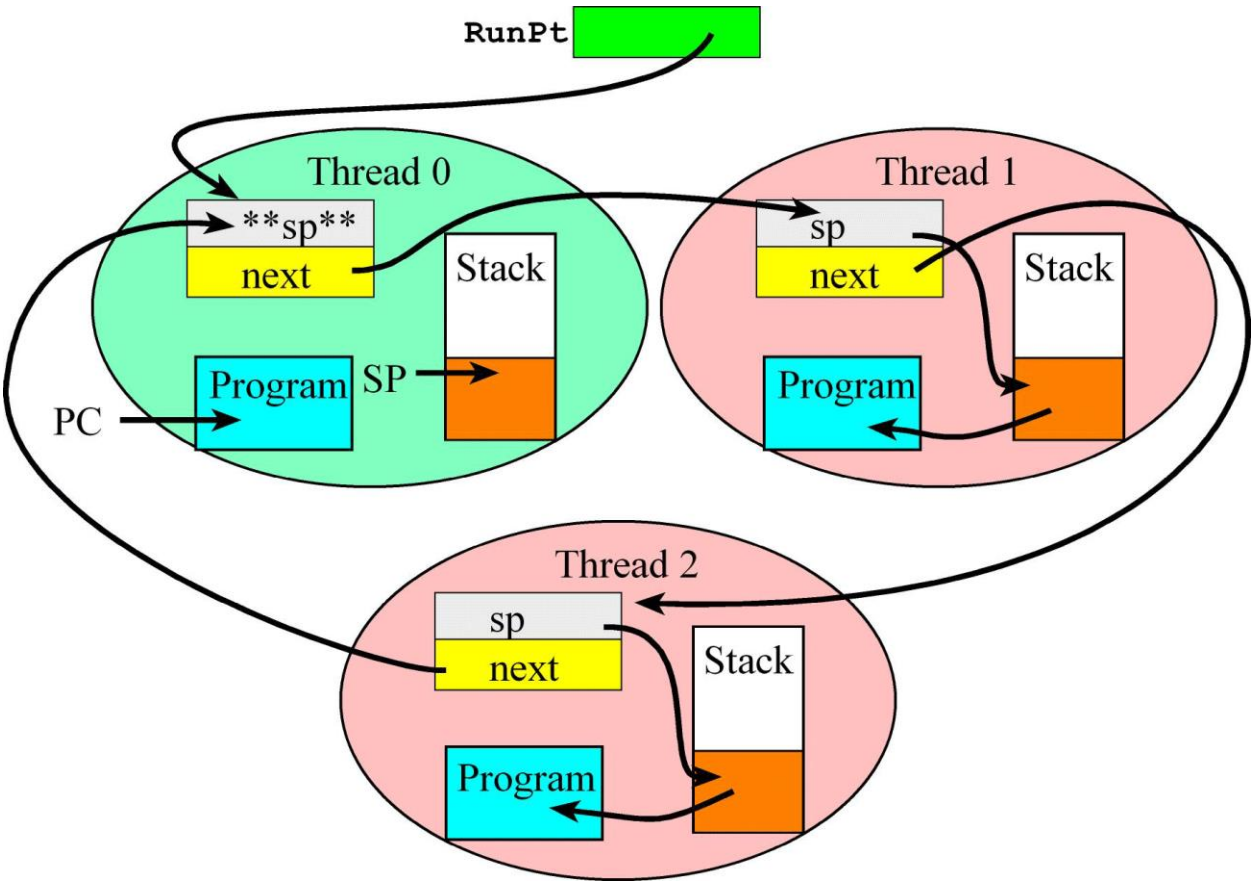


*Figure 2.16. Three threads have their TCBs in a circular linked list. "**sp**" means this field is invalid for the one thread that is actually running.*

The first time a thread runs, the only registers that must be set are PC, SP, the T-bit in the PSR (T=1), and the I-bit in the PSR (I=0). For debugging purposes, we do initialize the other registers the first time each thread is run, but these other initial values do not matter. We learned this trick of setting the initial register value to the register number (e.g., R5 is initially 0x05050505) from Micrium uC/OS-II. Notice in this simple example, the first time **Task0** runs it will be executed as a result of **StartOS**. However, the first time **Task1** and **Task2** are run, it will be executed as a result of running the **SysTick_Handler**. In particular, the initial LR and PSR for Task0 are set explicitly in **StartOS**, while the initial LR and PSR for **Task1** and **Task2** are defined in the initial stack set in **SetInitialStack**. An alternative approach to launching would have been to set the SP to the R4 field of its stack, set the LR to 0xFFFFFFF9 and jump to line 8 of the scheduler. Most commercial RTOS use this alternative approach because it makes it easier to change. But we decided to present this **StartOS** because we feel it is easier to understand the steps needed to launch.

2.3.6. Profiling the OS[edit]

Play Video

You can find this simple RTOS in the starter projects as RTOS_xxx, where xxx refers to the specific microcontroller on which the example was tested. Figures 2.17 and 2.18 show profiles of this RTOS at different time scales. We can estimate the thread switch time to be about 0.8 µs, because of the gap between the last edge on one pin to the first edge on the next pin. In this case because the thread switch occurs every 2 ms, the 0.8-µs thread-switch overhead is not significant.

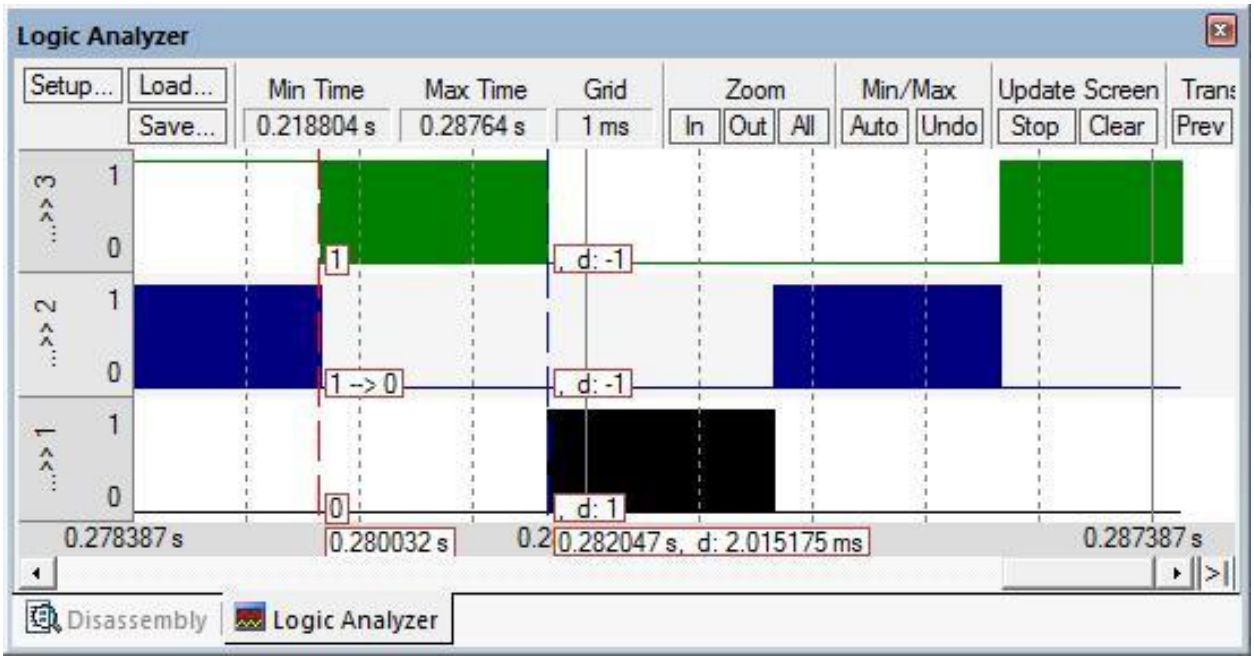


Figure 2.17. The RTOS runs three threads by giving each a 2ms, measured in simulator for the TM4C123.

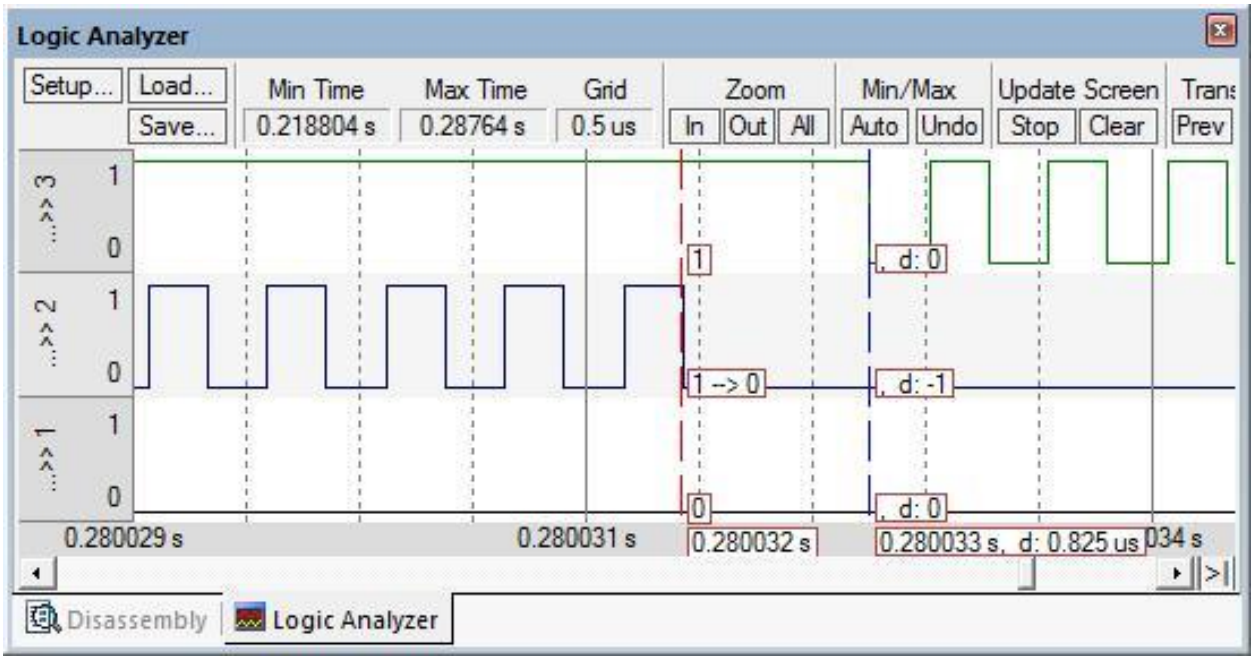


Figure 2.18. Profile showing the thread switch time is about 0.8 µs, measured in simulator for the TM4C123.

2.3.7. Linking assembly to C[edit]

Play Video

One of the limitations of the previous scheduler is that it's written entirely in assembly. Although fast, assembly programming is hard to extend and hard to debug. One simple way to extend this round robin scheduler is to have the assembly SysTick ISR call a C function, as shown in Program 2.15. 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 as **RoundRobin_xxx**, where xxx refers to the specific microcontroller on which the example was tested.

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

Program 2.15. Round robin scheduler written in C.

The new SysTick ISR calls the C function in order to find the next thread to run, Program 2.16. 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. Since this is an ISR, recall that LR contains 0xFFFFFFF9, signifying we are running an ISR. 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. According to AAPCS, 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. Note that the other registers, R1,R2,R3 and R12 are of no consequence to us, so we don't bother saving them.

```
    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
```

Program 2.16. Assembly code for the thread switcher with call to the scheduler written in C.

In this implementation, we are running the C function **Scheduler** with interrupts disabled. On one hand this is good because all read-modify-write operations to shared globals will execute atomically, and not create critical sections. On the other hand, since interrupts are disabled, it will delay other possibly more important interrupts from being served. Running with interrupts disabled will cause time jitter for periodic threads and latency for event-response threads. In Lab 2 we will manage this problem by running all the real-time tasks inside this Scheduler function itself.

2.3.8. Periodic threads[edit]

Play Video

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 as shown in Figure 2.19 and Program 2.17. 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.

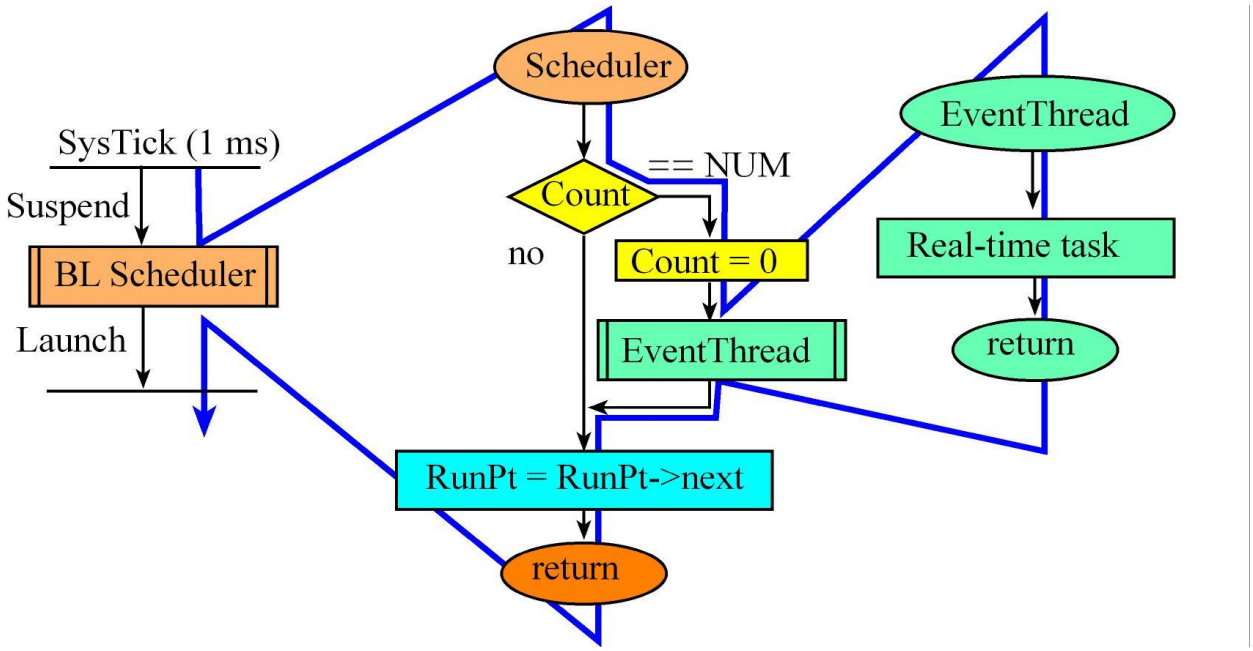


Figure 2.19. Simple mechanism to implement periodic event threads is to run them in the scheduler.

```
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 2.17. Round robin scheduler with periodic tasks.

This approach has very little time jitter because SysTick interrupts occur at a fixed and accurate rate. The SysTick ISR calls the Scheduler, and then the Scheduler calls the user task. The execution delay from the SysTick trigger to the running of the user task is a constant, so the time between executions of the user task is fixed and exactly equal to the SysTick trigger period.

If there are multiple real-time periodic tasks to run, then you should schedule at most one of them during each SysTick ISR execution. This way the time to execute one periodic task will not affect the time jitter of the other periodic tasks. 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 as illustrated in Program 2.18.

```
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 2.18. Round robin scheduler with two periodic tasks.

Consider a more difficult example, where we wish to run Task0 every 1 ms, Task1 every 1.5 ms and Task2 every 2 ms. In order to create non-overlapping executions, we will need a thread switch period faster than 1 kHz, so we don't have to run Task0 every interrupt. So, let's try working it out for 2 kHz, or 0.5 ms. The common multiple of 1, 1.5 and 2 is 6 ms. So we use a counter from 0 to 11, and try to schedule the three tasks. Start with Task0 running every other, and then try to schedule Task1 running every third. There is a conflict at 4 and 10.

Task0: runs every 1 ms at counter values 0, 2, 4, 6, 8, and 10
Task1: runs every 1.5 ms at counter values 1, 4, 7, and 10

So, let's try running faster at 4 kHz or every 0.25 ms. The common multiple is still 6 ms, but now the counter goes from 0 to 23. We can find a solution

Task0: runs every 1 ms at counter values 0, 4, 8, 12, 16, and 20
Task1: runs every 1.5 ms at counter values 1, 7, 13, and 19
Task2: runs every 2 ms at counter values 2, 10, and 18

In order this system to operate, the maximum time to execute each task must be very short compared to the period used to switch threads.

2.4.1. Introduction to semaphores[edit]

Play Video

Remember that when an embedded system employs a real-time operating system to manage threads, typically this system combines multiple hardware/software objects to solve one dedicated problem. In other words, the components of an embedded system are tightly coupled. For example, in lab all threads together implement a personal fitness device. The fact that an embedded system has many components that combine to solve a single problem leads

to the criteria that threads must have mechanisms to interact with each other. The fact that an embedded system may be deployed in safety-critical environments also implies that these interactions be effective and reliable.

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. Other names for wait are **pend** and **P**(derived from the Dutch word *proberen*, which means to test). Other names for signal are **post** and **V** (derived from the Dutch word *verhogen*, which means to increment).

The concept of a semaphore was originally conceived by the Dutch computer scientist Edsger Dijkstra in 1965. He received many awards including the 1972 Turing Award. He was the Schlumberger Centennial Chair of Computer Sciences at The University of Texas at Austin from 1984 until 2000. Interestingly he was one of the early critics of the GOTO instruction in high-level languages. Partly due to his passion, structured programming languages like C, C++ and Java have almost completely replaced non-structured languages like BASIC, COBOL, and FORTRAN.

In this class we will develop three implementations of semaphores, but we will begin with the simplest implementation called “spin-lock” (Figure 2.20). Each semaphore has a counter. If the thread calls **OS_Wait** with the counter equal to zero it will “spin” (do nothing) until the counter goes above zero (Program 2.20). 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. Program 2.20 shows this simple implementation the semaphore functions, which we will use in Lab 2.

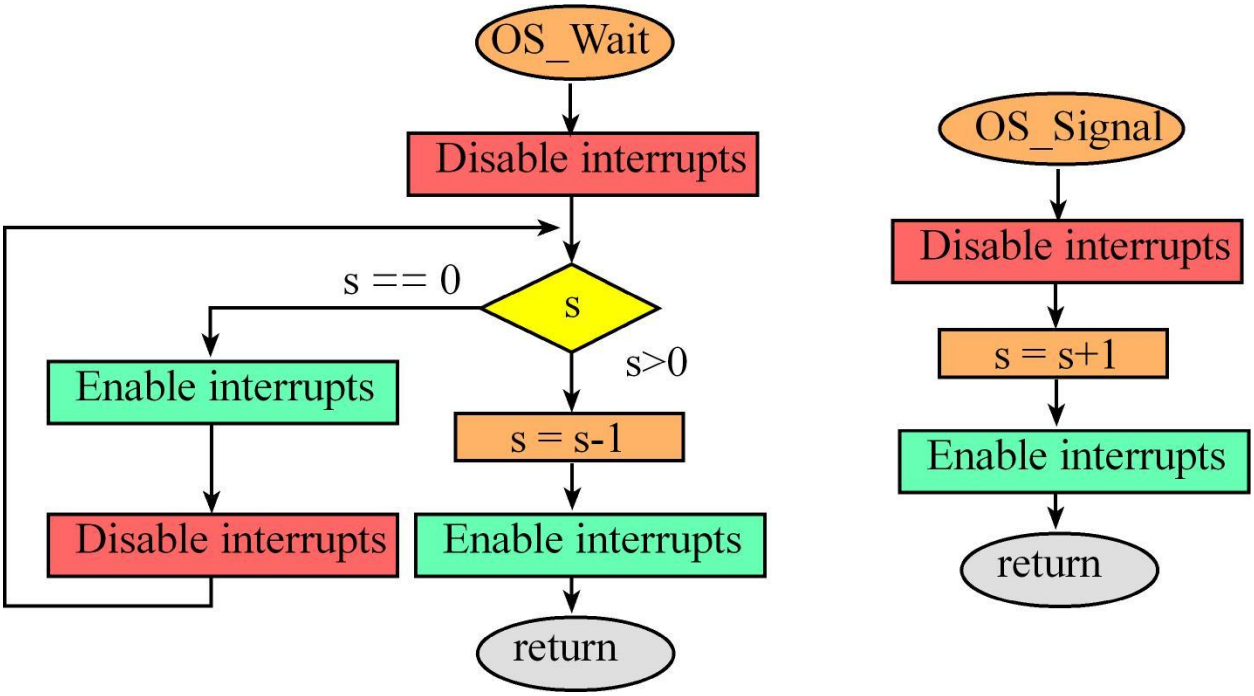


Figure 2.20. Flowcharts of a spinlock counting semaphore.

In the C implementation of spinlock semaphores, the tricky part is to guarantee all read-modify-write sequences are atomic. The while-loop reads the counter, which is always run with interrupts disabled. If the counter is greater than 0, it will decrement and store, such that the entire read-modify-write sequence is run with interrupts disabled. The while-loop must spend some time with interrupts enabled to allow other threads an opportunity to run, and hence these other threads have an opportunity to call signal.

```
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 2.20. A spinlock counting semaphore.

Spinlock semaphores are inefficient, wasting processor time when they spin on a counter with a value of zero. In the subsequent chapters we will develop more complicated schemes to recover this lost time.

2.4.2. Applications of semaphores[edit]

Play Video

When we use a semaphore, we usually can assign a meaning or significance to the counter value. In the first application we could use a semaphore as a **lock** so only one thread at a time has access to a shared object. Another name for this semaphore is **mutex**, because it provides **mutual exclusion**. If the semaphore is 1 it means the object is free. If the semaphore is 0 it means the object is busy being used by another thread. For this application the initial value of the semaphore (**x**) is 1, because the object is initially free. A thread calls **OS_Wait** to capture the object (decrement counter) and that same thread calls **OS_Signal** to release the object (increment counter).

```
void Thread1(void){
  Init1();
  while(1){
    OS_Wait(&x);
    // exclusive access to object
    OS_Signal(&x);
    // other processing
  }
}
void Thread2(void){
  Init2();
  while(1){
    OS_Wait(&x);
    // exclusive access to object
    OS_Signal(&x);
    // other processing
  }
}
```

In second application we could use a semaphore for synchronization. One example of this synchronization is a**condition variable**. If the semaphore is 0 it means an event has not yet happened, or things are not yet ok. If the semaphore is 1 it means the event has occurred and things are ok. For this application the initial value of the semaphore is 0, because the event is yet to occur. A thread calls OS_Wait to wait for the event (decrement counter) and another thread calls **OS_Signal** to signal that the event has occurred (increment counter). Let y be a semaphore with initial value of 0.

```
void Thread1(void){
  Init1();
  OS_Wait(&y); // wait for event
       // event to occur
  while(1){
       // other processing
  }
}
void Thread2(void){
  Init2();
// this thread knows the event has occurred
  OS_Signal(&y); // signal event
  while(1){
     // other processing
  }
}
```

2.5.1. Resource sharing, nonreentrant code or mutual exclusion

[edit]

Play Video

This section can be used in two ways. First it provides a short introduction to the kinds of problems that can be solved using semaphores. In other words, if you have a problem similar to one of these examples, then you should consider a thread scheduler with semaphores as one possible implementation. Second, this section provides the basic approach to solving these particular problems. An important design step when using semaphores is to ascribe a meaning to each semaphore and a meaning to each value that semaphore can have.

The objective of this example is to share a common resource on a one at a time basis, also referred to as "mutually exclusive" fashion. The critical section (or vulnerable window) of nonreentrant software is that region that should only be executed by one thread at a time. As an example, the common resource we will consider is a display device (LCD). Mutual exclusion in this context means that once a thread has begun executing a set of LCD functions, then no other thread is allowed to use the LCD. See Program 2.21. In other words, whichever thread starts to output to the LCD first will be allowed to finish outputting. The thread that arrives second will simply wait for the first to finish. Both will be allowed to output to the LCD, however, they will do so on a one at a time basis. The mechanism to create mutual exclusion is to initialize the semaphore to 1, execute **OS_Wait** at the start of the critical section, and then execute **OS_Signal** at the end of the critical section. In this way, the information sent to one part of the LCD is not mixed with information sent to another part of the LCD.

```
void Task2(void){
  Init2();
  while(1){
```

```
void Task5(void){
  Init5();
  while(1){
```

```
    Unrelated2();
    OS_Wait(&LCDmutex);
    BSP_LCD_PlotPoint(Data, COLOR);
    BSP_LCD_PlotIncrement();
    OS_Signal(&LCDmutex);
  }
}
```

```
    Unrelated5();
    OS_Wait(&LCDmutex);
    BSP_LCD_SetCursor(5, 0);
    BSP_LCD_OutUDec4(Time/10,CO
    BSP_LCD_SetCursor(5, 1);
    BSP_LCD_OutUDec4(Steps,COLO
    BSP_LCD_SetCursor(16, 0);
    BSP_LCD_OutUFix2_1(TempData
    BSP_LCD_SetCursor(16, 1);
    BSP_LCD_OutUDec4(SoundRMS,C
     OS_Signal(&LCDmutex);
  }
}
```

Program 2.21. Semaphores used to implement mutual exclusion, simplified from usage in Lab 2.

Initially, the semaphore is 1. If **LCDmutex** is 1, it means the LCD is free. If **LCDmutex** is 0, it means the LCD is busy and no thread is waiting. In this chapter, a thread that calls **OS_Wait** on a semaphore already 0 will wait until the semaphore becomes greater than 0. For a spinlock semaphore in this application, the possible values are only 0 (busy) or 1 (free). A semaphore that can only be 0 or 1 is called a **binary semaphore**.

2.5.2. Thread communication between two threads using a mailbox[edit]

Play Video

The objective of this example is to communicate between two main threads using a mailbox. In this first implementation both the producer and consumer are main threads, which are scheduled by the round robin thread scheduler (Program 2.22). 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.

Execution	Mail	Send	Ack	Comments
Initially	none	0	0	
Producer sets Mail	valid	0	0	Producer gets here first
Producer signals Send	valid	1	0	
Producer waits on Ack	valid	1	0	Producer spins because Ack =0
Consumer waits on Send	valid	0	0	Returns immediately because Send was 1

Consumer reads Mail	none	0	0	Reading once means Mail not valid
Consumer signals Ack	none	0	1	Consumer continues to execute
Producer finishes wait	none	0	0	Producer continues to execute

Consider the case where the consumer executes first.

Execution	Mail	Send	Ack	Comments
Initially	none	0	0	
Consumer waits on send	none	0	0	Consumer spins because Send =0
Producer sets Mail	valid	0	0	Producer gets here second
Producer signals Send	valid	1	0	
Producer waits on Ack	valid	1	0	Producer spins because Ack =0
Consumer finishes wait	valid	0	0	Consumer continues to execute
Consumer reads Mail	none	0	0	Reading once means Mail not valid
Consumer signals Ack	none	0	1	Consumer continues to execute
Producer finishes wait	none	0	0	Producer continues to execute

```
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
  OS_Signal(&Ack);
  return theData;
}
void Consumer(void){
  Init2();
  while(1){ uint32_t this
    thisData = RecvMail()
    Unrelated2();
  }
}
```

Program 2.22. 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 (Program 2.24). 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 ma
}

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

Program 2.24. Semaphores used to implement a mailbox. Producer is an event thread and Consumer is a main thread.

A mailbox forces the producer and consumer to execute lock-step {producer, consumer, producer, consumer,…}. It also suffers from the potential to lose data. Both of these limitations will motivate the **first in first out (FIFO) queue**presented in the next chapter.