The objectives of Chapter 3 are implement:

- Cooperation using suspend
- Blocking semaphores
- Data flow with first in first out queues
- Thread sleeping
- Periodic interrupts to manage periodic tasks

An important aspect of real-time systems is managing time, more specifically minimizing wastage of time through an idle busy-wait. Such busy-wait operations were used in our simple implementation of semaphores in the last chapter. In this chapter we will see how we can recover this wasted time.

3.1.1. Spin-lock semaphore implementation with cooperation[edit]

Spin lock with cooperation

Play Video

Sometimes a thread knows it can no longer make progress. If a thread wishes to cooperatively release control of the processor it can call **OS_Suspend**, which will halt this thread and run another thread. Because all the threads work together to solve a single problem, adding cooperation at strategic places allows the system designer to greatly improve performance. When threads wish to suspend themselves, they call **OS_Suspend**. Again, the SysTick ISR must be configured as a priority 7 interrupt so that it does not attempt to suspend any hardware ISRs that may be running. **OS_Suspend** can only be called by a main thread. Note that it is possible to force a SysTick interrupt by bypassing the normal "count to zero" event that causes it. To do this, we write a 1 to bit 26 of the **INTCTRL** register, which causes the SysTick interrupt. Writing zeros to the other bits of this register has no effect. This operation will set the **Count** flag in SysTick and the ISR will suspend the current thread, runs the **SysTick_Handler** (which calls the scheduler), and then launch another thread. In this first implementation, we will not reset the SysTick timer from interrupting normally (count to zero). Rather we simply inject another execution of the ISR. If we were 75% through the 1-ms time slice when OS_Suspend is called, this operation will suspend the current thread and grant the remaining 0.25-ms time to the next thread.

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

One way to make a spin-lock semaphore more efficient is to place a thread switch in the while loop as it is spinning, as shown on the right of Figure 3.1 and as Program 3.1. This way, if the semaphore is not available, the thread stops running. If there are n other running threads and the time slice is Δt, then the semaphore is checked every $n*\Delta t$, and very little processor time is

wasted on the thread which cannot run. One way to suspend a thread is to trigger a SysTick interrupt.

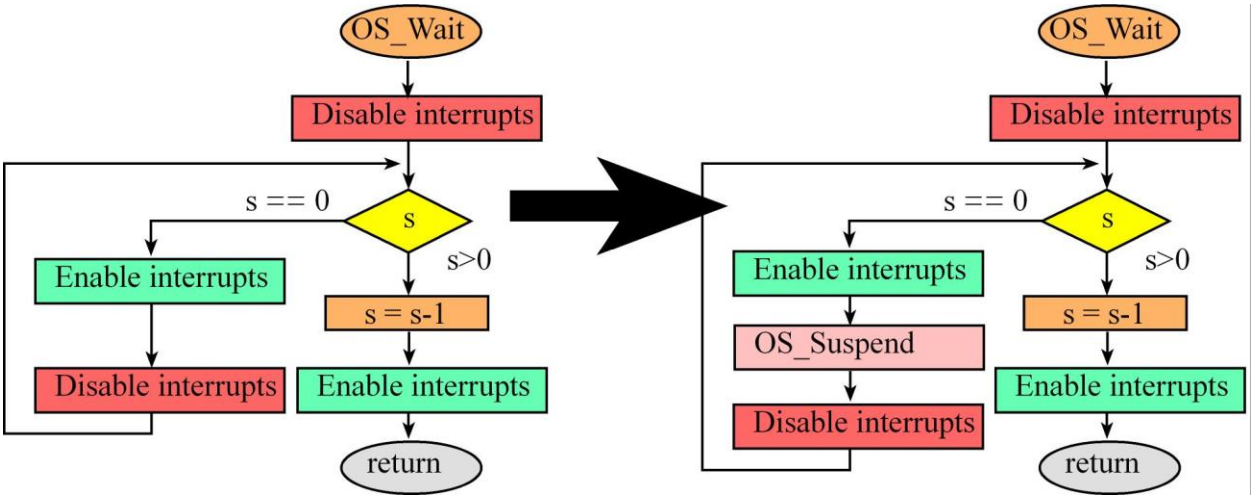


Figure 3.1. Regular and efficient implementations of spinlock wait.

```
void OS_Suspend(void){
  INTCTRL = 0x04000000; // trigger SysTick
}
void OS_Wait(int32_t *s){
  DisableInterrupts();
  while((*s) == 0){
    EnableInterrupts();
    OS_Suspend(); // run thread switcher
    DisableInterrupts();
  }
  (*s) = (*s) - 1;
  EnableInterrupts(); }
```

Program 3.1. A cooperative thread switch will occur if the software explicitly triggers a thread switch.

The implementation in Program 3.1 did not reset the SysTick counter on a cooperative thread switch. So it is a little unfair for the thread that happens to be run next. However, in this implementation, since SysTick interrupts are still triggered every 1 ms, SysTick can be used to perform periodic tasks. Once we shift the running of periodic tasks to another timer ISR, we will be able to add a more fair implementation of suspend:

```
void OS_Suspend(void){
  STCURRENT = 0;        // reset counter
  INTCTRL = 0x04000000; // trigger SysTick
}
```

Using this version of suspend, if we are 75% through the 1-ms time slice when **OS_Suspend** is called, this operation will suspend the current thread and grant a full 1-ms time to the next thread. We will be able to use this version of suspend once we move the periodic event threads away from SysTick and onto another periodic interrupt.

In particular, periodic event threads will be handled in Lab 3 using **BSP_PeriodicTask_Init**. This means the accurate running of event threads will not be disturbed by resetting the SysTick timer. Although you could use either version of **OS_Suspend** in Lab 3, resetting the counter will be fairer.

3.1.2. Cooperative Scheduler[edit]

Cooperative scheduler, show in cooperative project

Play Video

In this section we will develop a 3-thread cooperative round-robin scheduler by letting the tasks suspend themselves by triggering a SysTick interrupt.

You can find this cooperative OS as Cooperative_xxx, where xxx refers to the specific microcontroller on which the example was tested, Program 3.2. Figure 3.2 shows a profile of this OS. We can estimate the thread switch time to be about 1 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 1.3 µs, the 1-µs thread-switch overhead is significant. Even though SysTick interrupts are armed, the SysTick hardware never triggers an interrupt. Instead, each thread voluntarily suspends itself before the 1-ms interval.

```
void Task0(void){
  Count0 = 0;
  while(1){
    Count0++;
    Profile_Toggle0(); // toggle bit
    OS_Suspend();
  }
}
void Task1(void){
  Count1 = 0;
  while(1){
    Count1++;
    Profile_Toggle1(); // toggle bit
    OS_Suspend();
  }
}
void Task2(void){
  Count2 = 0;
```

```
  while(1){
    Count2++;
    Profile_Toggle2(); // toggle bit
    OS_Suspend();
  }
}
```

Program 3.2. User threads that use a cooperative scheduler.

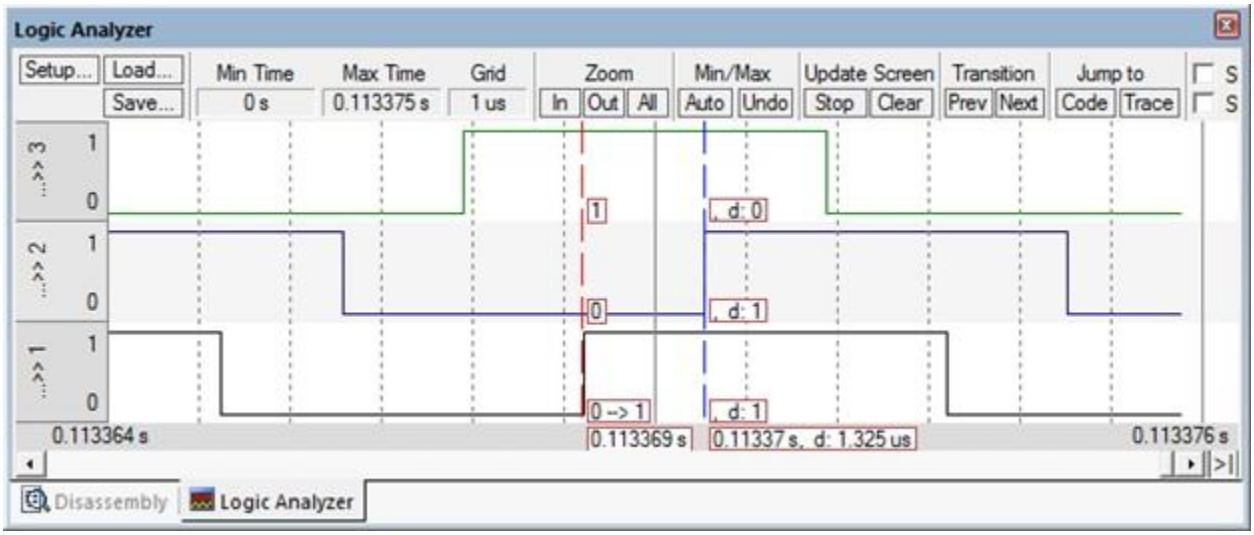


Figure 3.2. The OS runs three threads; each thread volunteers to suspend running in simulation mode on the TM4C123. The three profile pins from Program 3.2 are measured versus time using a logic analyzer.

We did not add cooperation to Lab 2 because it would have disturbed the ability of SysTick to run periodic tasks. In other words, in Lab 2 we had to maintain SysTick periodic interrupts at exactly 1000 Hz or one every 1 ms. However, we will be able to add cooperation in Lab 3, once we schedule periodic tasks using a separate hardware timer generating its own high-priority interrupts using **BSP_PeriodicTask_Init** and/or **BSP_PeriodicTask_Init16**.

3.2.1. The need for blocking[edit]

Need for blocking

Play Video

The basic idea of a **blocking semaphore** will be to prevent a thread from running (we say the thread is blocked) when the thread needs a resource that is unavailable. There are three reasons we will replace spin-lock semaphores with blocking semaphores. The first reason is an obvious **inefficiency** in having threads spin while there is nothing for them to do. Blocking semaphores will be a means to recapture this lost processing time. Essentially, with blocking

semaphores, a thread will not run unless it has useful work it can accomplish. Even with spinlock/cooperation it is wasteful to launch a thread you know can't run, only to suspend itself 10 µs later.

The second problem with spin-lock semaphores is a **fairness** issue. Consider the case with threads 1 2 3 running in round robin order. Assume thread 1 is the one calling Signal, and threads 2 and 3 call Wait. If threads 2 and 3 are both spinning waiting on the semaphore, and then thread 1 signals the semaphore, which thread (2 or 3) will be allowed to run? Because of its position in the 1 2 3 cycle, thread 2 will always capture the semaphore ahead of thread 3. It seems fair when the status of a resource goes from busy to available, that all threads waiting on the resource get equal chance. A similar problem exists in non-computing scenarios where fairness is achieved by issuing numbered tickets, creating queues, or having the customers sign a log when they enter the business looking for service. E.g., when waiting for a checkout clerk at the grocery store, we know to get in line, and we think it is unfair for pushy people to cut in line in front of us. We define bounded waiting as the condition where once a thread begins to wait on a resource (the call to **OS_Wait** does not return right away), there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed. **Bounded waiting** does not guarantee a minimum time before **OS_Wait** will return, it just guarantees a finite number of other threads will go before this thread. For example, it is holiday time, I want to mail a package to my mom, I walk into the post office and take a number, the number on the ticket is 251, I look up at the counter and the display shows 212, and I know there are 39 people ahead of me in line. We could implement bounded waiting with blocking semaphores by placing the blocked threads on a list, which is sorted by the order in which they blocked. When we wake up a thread off the blocked list, we wake up the one that has been waiting the longest. *Note: none of the labs in this class will require you to implement bounded waiting.* We introduce the concept of bounded waiting because it is a feature available in most commercial operating systems.

The third reason to develop blocking semaphores will be the desire to implement a **priority thread scheduler**. In Labs 2 and 3, you implemented a round-robin scheduler and assumed each thread had equal importance. In Lab 4 you will create a priority scheduler that will run the highest priority thread that is ready to run. For example, if we have one high priority thread that is ready, we will run it over and over regardless of whether or not there are any lower priority threads ready. We will discuss the issues of starvation, aging, inversion and inheritance in Chapter 4. A priority scheduler will require the use of blocking semaphores. I.e., we cannot use a priority scheduler with spin-lock semaphores.

3.2.2. The blocked state[edit]

The blocked state

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

Figure 3.3 shows five threads. In this simple implementation of blocking we add a third field, called blocked, to the TCB structure, defining the status of the thread. The **RunPt** points to the TCB of the thread that is currently running. The next field is a pointer chaining all five TCBs into a circular linked list. Each TCB has a **StackPt** field. Recall that, if the thread is running it is using the real SP for its stack pointer. However, the other threads have their stack pointers saved in this field. The third field is a blocked field. If the blocked field is null, there are no resources preventing the thread from running. On the other hand, if a thread is blocked, the blocked field contains a pointer to the semaphore on which this thread is blocked. In Figure 3.3, we see threads 2 and 4 are blocked waiting for the resource (semaphore free). All five threads are in the circular linked list although only three of them will be run.

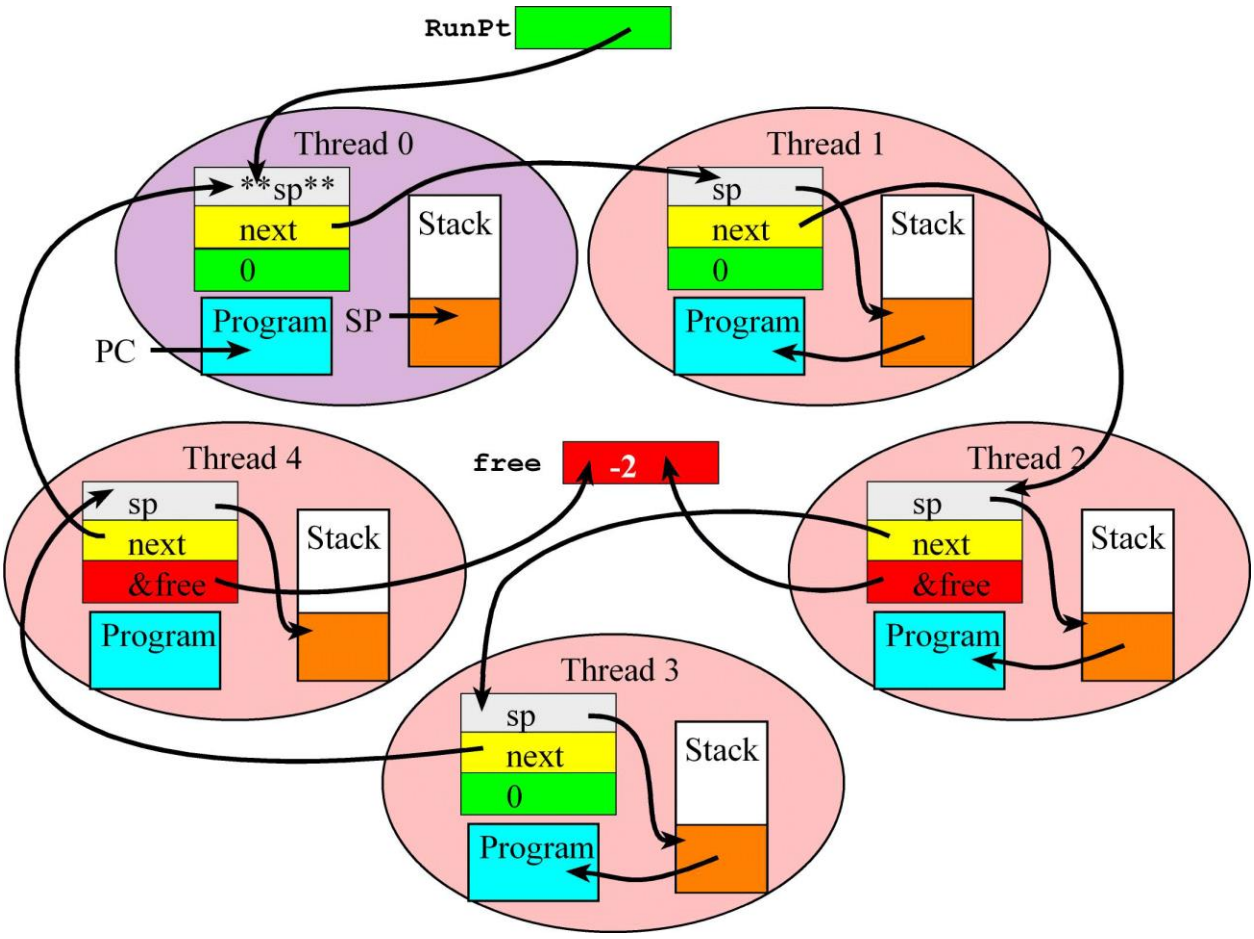


Figure 3.3. Threads 0, 1 and 3 are being run by the scheduler. Threads 2 and 4 are blocked on free and will not run until some thread signals free.

In this simple approach, a main thread can only be blocked on one resource. In other words, when a thread calls **OS_Wait** on a semaphore with value 0 or less, that thread is blocked and stops running. Therefore, once blocked on one semaphore, it cannot block on a second semaphore. Figure 3.3 shows just one semaphore, but even when there are multiple semaphores, we need only one blocked field in the TCB. Since C considers zero as false and nonzero as true, the blocked field can also be considered as a Boolean, specifying whether or not the thread is blocked. This simple solution is adequate for systems with a small number of threads (e.g., less than 20).

Notice in this simple implementation we do not maintain a separate linked list of threads blocked on a specific semaphore. In particular, in Figure 3.3 we know threads 2 and 5 are blocked on the semaphore **free**, but we do not know which thread blocked first. The advantage of this implementation using one circular linked list data structure to hold the TCBs of all the threads will be speed and simplicity. Note that, we need to add threads to the TCB list only when created, and remove them from the TCB list if the thread kills itself. If a thread cannot run (blocked) we can signify this event by setting its blocked field like Figure 3.3 to point to the semaphore on which the thread is blocked.

In order to implement bounded waiting, we would have to create a separate blocked linked list for each reason why the thread cannot execute. For example, we could have one blocked list for threads waiting for the output display to be free, one list for threads waiting because a FIFO is full, and one lists for threads waiting because another FIFO is empty. In general, we will have one blocked list with each reason a thread might not be able to run. This approach will be efficient for systems with many threads (e.g., more than 20). These linked lists contain threads sorted in order of how long they have been waiting. To implement bounded waiting, when we signal a semaphore, we wake up the thread that has been waiting the longest. *Note: none of the labs in this class will require you to implement bounded waiting.*

In this more complex implementation, we unchain a TCB from the ready circular linked list when it is blocked. In this way a blocked thread will never run. We place the blocked TCBs on a linear linked list associated with the semaphore (the reason it was blocked). We can implement bounded waiting by putting blocked TCBs at the end of the list and waking up threads from the front of the list. There will be a separate linked list for every semaphore. This method is efficient when there are many threads that will be blocked at one time. The thread switching will be faster because the scheduler will only see threads that could run, and not have to look at blocked threads in the circular linked list. *We do not expect you to unchain threads when blocked and rechain them when they wake up in any of the labs.* We discuss it because this is how most commercial operating systems implement blocking.

3.2.3. Implementation[edit]

Implementation of blocking

Play Video

We will present the simple approach for implementing blocking semaphores, and we suggest you use this approach for Lab 3. Notice in Figure 3.4 that wait always decrements and signal always increments. This means the semaphore can become negative. In the example of using a semaphore to implement mutual exclusion, if **free** is 1, it means the resource is free. If free is 0, it means the resource is being used. If **free** is -1, it means one thread is using the resource and a second thread is blocked, waiting to use it. If **free** is -2, it means one thread is using the resource and two other threads are blocked, waiting to use it. In this simple implementation, the semaphore is a signed integer.

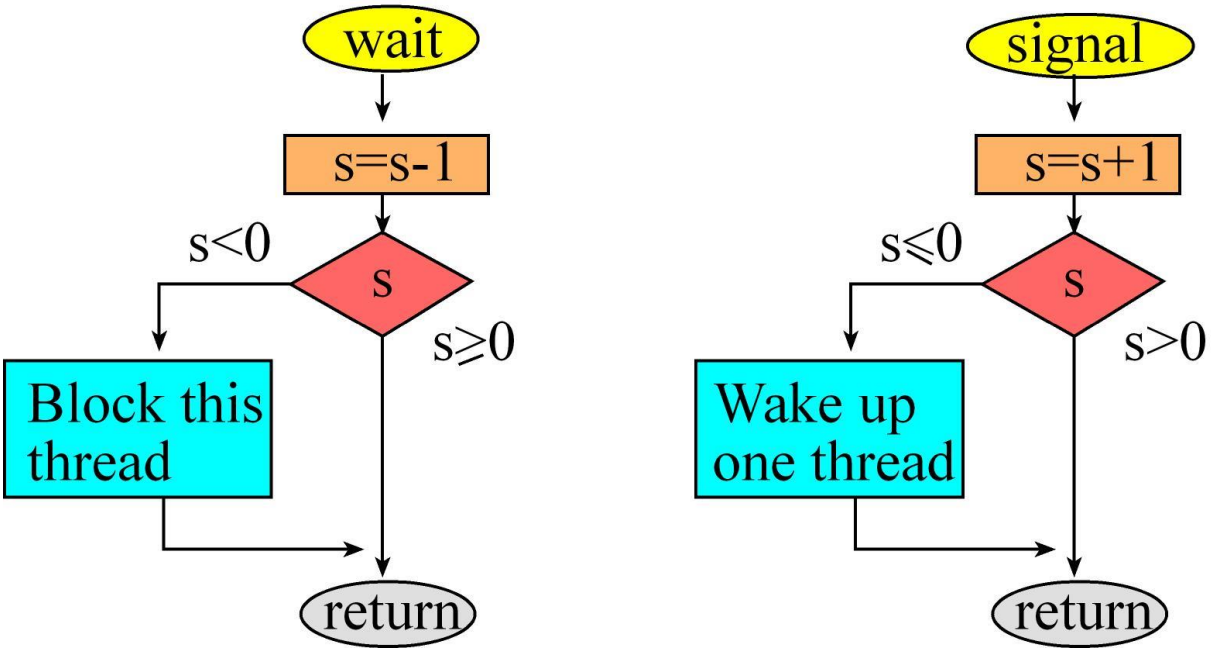


Figure 3.4. Flowcharts of a blocking counting semaphore.

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

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

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

```
void OS_Signal(int32_t *s){
  tcbType *pt;
  DisableInterrupts();
  (*s) = (*s) + 1;
  if((*s) <= 0){
```

```
    pt = RunPt->next;    // search for a thread blocked on this semaphore
    while(pt->blocked != s){
      pt = pt->next;
    }
    pt->blocked = 0;     // wakeup this one
  }
  EnableInterrupts();
}
```

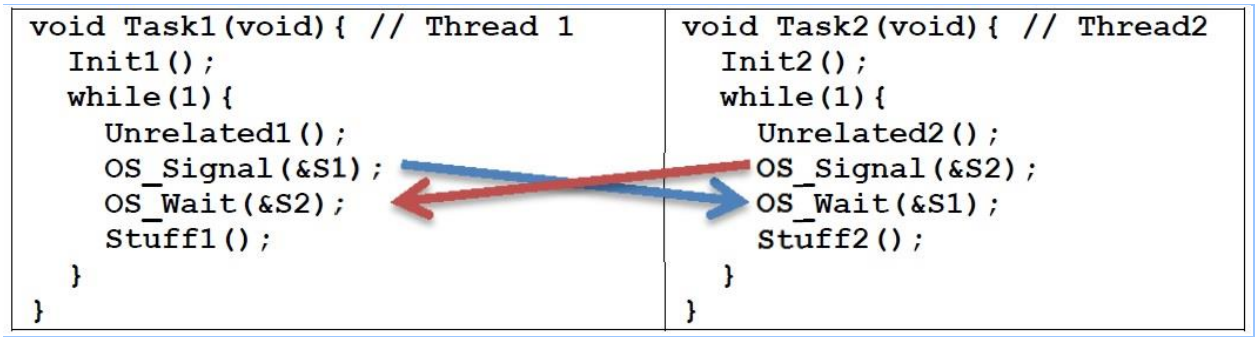
Notice in this implementation, calling the signal will not invoke a thread switch. During the thread switch, the OS searches the circular linked-list for a thread with a blocked field equal to zero (the woken up thread is a possible candidate). This simple implementation will not allow you to implement bounded waiting. *You do not need to implement bounded waiting in any of the labs.*

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

3.2.4. Thread synchronization or rendezvous[edit]

The objective of this example is to synchronize Threads 1 and 2 (Program 3.3). In other words, whichever thread gets to this part of the code first will wait for the other. Initially semaphores **S1** and **S2** are both 0. The two threads are said to **rendezvous** at the code following the signal and wait calls. The rendezvous will cause thread 1 to execute **Stuff1** at the same time (concurrently) as thread 2 executes its **Stuff2**. There are three scenarios the semaphores may experience and their significance is listed below:

```
  S1   S2    Meaning
  0    0     Neither thread has arrived at the rendezvous or both have passed
  -1   +1    Thread 2 arrived first and Thread 2 is blocked waiting for Thread 1
  +1   -1    Thread 1 arrived first and Thread 1 is blocked waiting for Thread 2
```



Program 3.3. Semaphores used to implement rendezvous.

3.3.1. Producer/Consumer problem using a FIFO[edit]

First in first out (FIFO) queue

Play Video

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

The first in first out circular queue (**FIFO**) is quite useful for implementing a buffered I/O interface (Figure 3.5). The function **Put** will store data in the FIFO, and the function **Get** will remove data. It operates in a first in first out manner, meaning the **Get** function will return/remove the oldest data. It can be used for both buffered input and buffered output. This order-preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). The class of FIFOs studied in this section will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be carefully shared by more than one program. The advantage of using a FIFO structure for a data flow problem is that we can decouple the producer and consumer threads. Without the FIFO we would have to produce one piece of data, then process it, produce another piece of data, then process it. With the FIFO, the producer thread can continue to produce data without having to wait for the consumer to finish processing the previous data. This decoupling can significantly improve system performance.

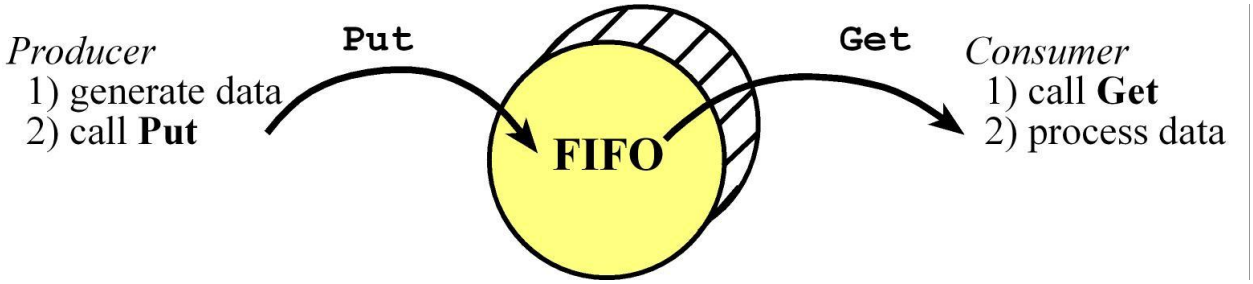


Figure 3.5. The FIFO is used to buffer data between the producer and consumer. The number of data stored in the FIFO varies dynamically, where Put adds one data element and Get removes/returns one data element.

You have probably already experienced the convenience of FIFOs. For example, a FIFO is used while streaming audio from the Internet. As sound data are received from the Internet they are stored (calls **Put**) into a FIFO. When the sound board needs data it calls **Get**. As long as the FIFO

never becomes full or empty, the sound is played in a continuous manner. A FIFO is also used when you ask the computer to print a file. Rather than waiting for the actual printing to occur character by character, the print command will put the data in a FIFO. Whenever the printer is free, it will get data from the FIFO. The advantage of the FIFO is it allows you to continue to use your computer while the printing occurs in the background. To implement this magic, our RTOS must be able to manage FIFOs. There are many producer/consumer applications, as we previously listed in Table 2.1, where the processes on the left are producers that create or input data, while the processes on the right are consumers which process or output data.

FIFOs can be statically allocated, where the buffer size is fixed at compile time, Figure 3.6. This means the maximum number of elements that can be stored in the FIFO at any one time is determined at design time. Alternately, FIFOs can be dynamically allocated, where the OS allows the buffer to grow and shrink in size dynamically. To allow a buffer to grow and shrink, the system needs a memory manager or **heap**. A heap allows the system to allocate, deallocate, and reallocate buffers in RAM dynamically. There are many memory managers (heaps), but the usual one available in C has these three functions. The function **malloc** creates a new buffer of a given size. The function **free** deallocates a buffer that is no longer needed. The function **realloc** allocates a new buffer, copies data from a previous buffer into the new buffer of different size, and then deallocates the previous buffer. **realloc** is the function needed to increase or decrease the allocated space for the FIFO statically-allocated FIFOs might result in lost data or reduced bandwidth compared to dynamic allocation.

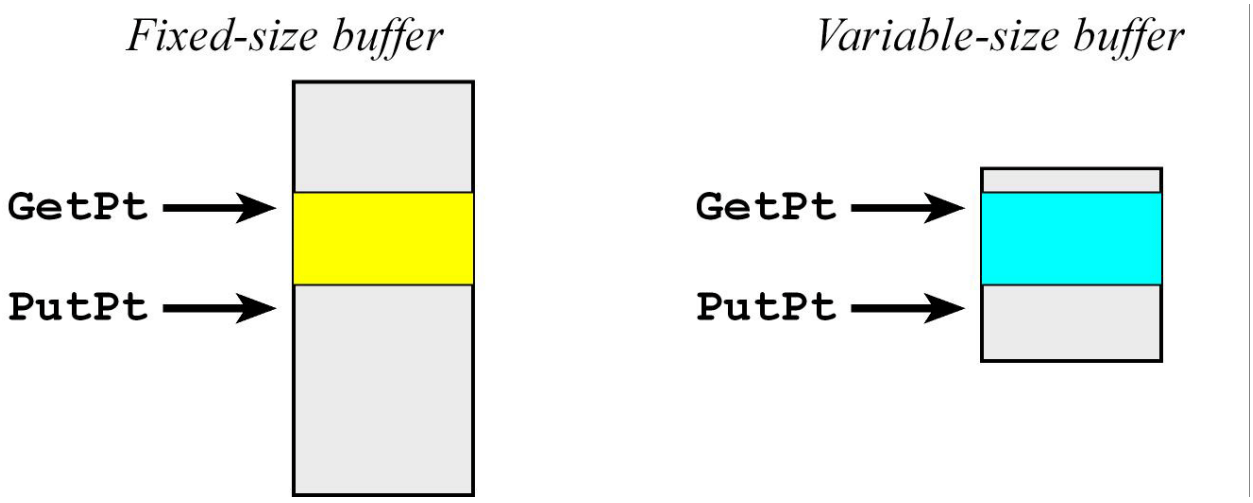


Figure 3.6. With static allocation, the maximum number of elements stored in the FIFO is fixed at compile time. With dynamic allocation, the system can call realloc when the FIFO is almost full to grow the size of the FIFO dynamically. Similarly, if the FIFO is almost empty, it can shrink the size freeing up memory.

A system is considered to be **deterministic** if when the system is run with the same set of inputs, it produces identical responses. Most real-time systems often require deterministic behavior, because testing can be used to certify performance. Dynamically-allocated FIFOs cause

the behavior of one subsystem (that might allocate large amounts of RAM from the heap) to affect behavior in another unrelated subsystem (our FIFO that wishes to increase buffer size). It is better for real-time systems to be reliable and verifiable than to have higher performance. As the heap runs, it can become fragmented; meaning the free memory in the heap has many little pieces, rather than a few big pieces. Since the time to reallocate a buffer can vary tremendously, depending on the fragmentation of the heap, it will be difficult to predict execution time for the FIFO functions. Since a statically allocated FIFO is simple, we will be able to predict execution behavior. For these reasons, we will restrict FIFO construction to static allocation. In other words, you should not use **malloc** and **free** in your RTOS.

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

3.3.2. Three-semaphore FIFO implementation[edit]

FIFO used by main threads

Play Video

The first scenario we will solve is where there are multiple producers and multiple consumers. In this case all threads are main threads, which are scheduled by the OS. The FIFO is used to pass data from the producers to the consumers. In this situation, the producers do not care to which consumer their data are passed, and the consumers do not care from which producer the data arrived. These are main threads, so we will block producers when the FIFO is full and we will block consumers when the FIFO is empty.

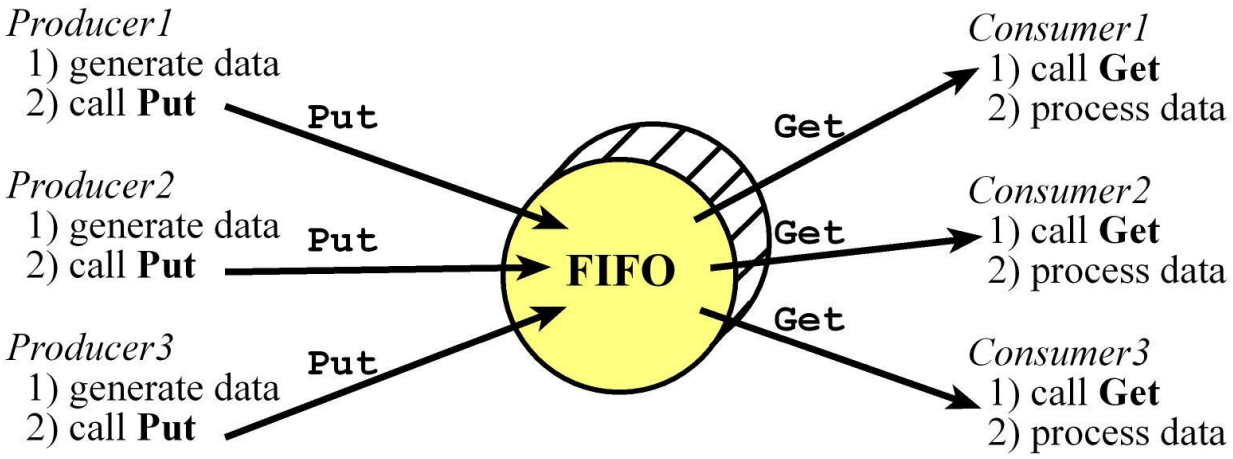


Figure 3.7. FIFO used to pass data from multiple producers to multiple consumers. All threads are main threads.

The producer puts data into the FIFO. If the FIFO is full and the user calls **Fifo_Put**, there are two responses we could employ. The first response would be for the **Fifo_Put** routine to block assuming it is unacceptable to discard data. The second response would be for the **Fifo_Put** routine to discard the data and return with an error value. In this subsection we will block the producer on a full FIFO. This implementation can be used if the producer is a main thread, but cannot be used if the producer is an event thread or ISR. The consumer removes data from the FIFO. For most applications, the consumer will be a main thread that calls **Fifo_Get** when it needs data to process. After a get, the particular information returned from the get routine is no longer saved in the FIFO. If the FIFO is empty and the user tries to get, the **Fifo_Get** routine will block because we assume the consumer needs data to proceed. The FIFO is order preserving, such that the information returned by repeated calls to **Fifo_Get**give data in the same order as the data saved by repeated calls of **Fifo_Put**.

The two-pointer implementation has, of course, two pointers. If we were to have infinite memory, a FIFO implementation is easy (Figure 3.8). **GetPt** points to the data that will be removed by the next call to **Fifo_Get**, and **PutPt** points to the empty space where the data will stored by the next call to **Fifo_Put**, see Program 3.4.

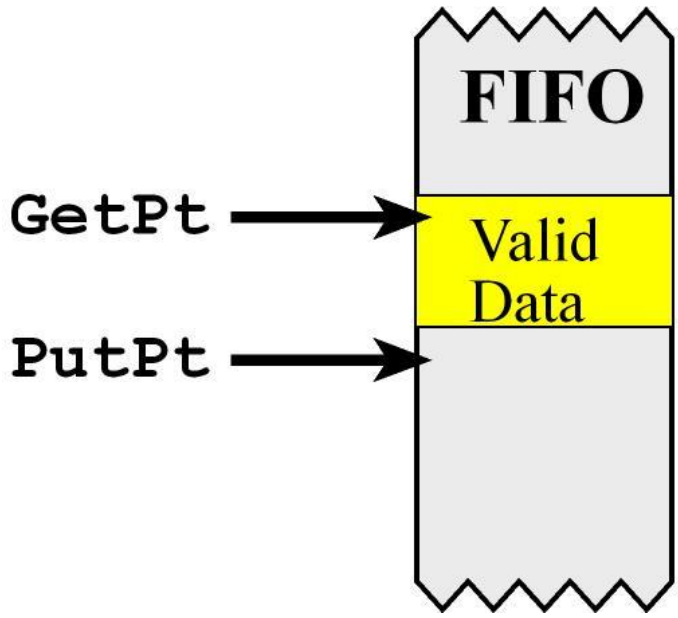


Figure 3.8. The FIFO implementation with infinite memory.

```
uint32_t volatile *PutPt; // put next
uint32_t volatile *GetPt; // get next
void Fifo_Put(uint32_t data){ // call by value
  *PutPt = data; // Put
  PutPt++;       // next
}
uint32_t Fifo_Get(void){ uint32_t data;
  data = *GetPt; // return by reference
```

```
  GetPt++;        // next
  return data;  // true if success
}
```

Program 3.4. Code fragments showing the basic idea of a FIFO.

There are four modifications that are required to the above functions. If the FIFO is full when **Fifo_Put** is called then the function should block. Similarly, if the FIFO is empty when **Fifo_Get** is called, then the function should block. *PutPt* must be wrapped back up to the top when it reaches the bottom (Figure 3.9).

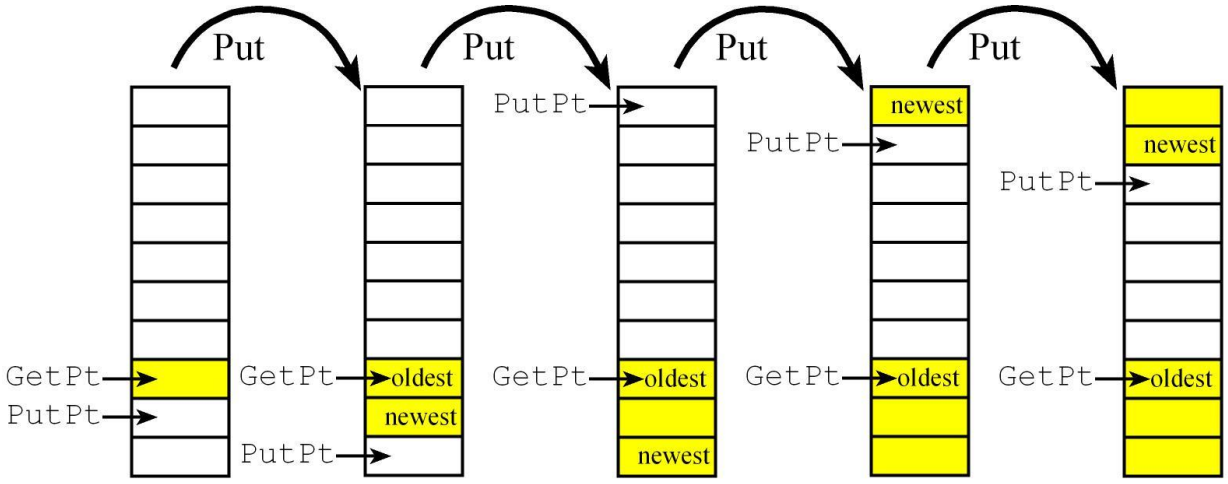


*Figure 3.9. The FIFO **Fifo_Put** operation showing the pointer wrap.*

The **GetPt** must also be wrapped back up to the top when it reaches the bottom (Figure 3.10).

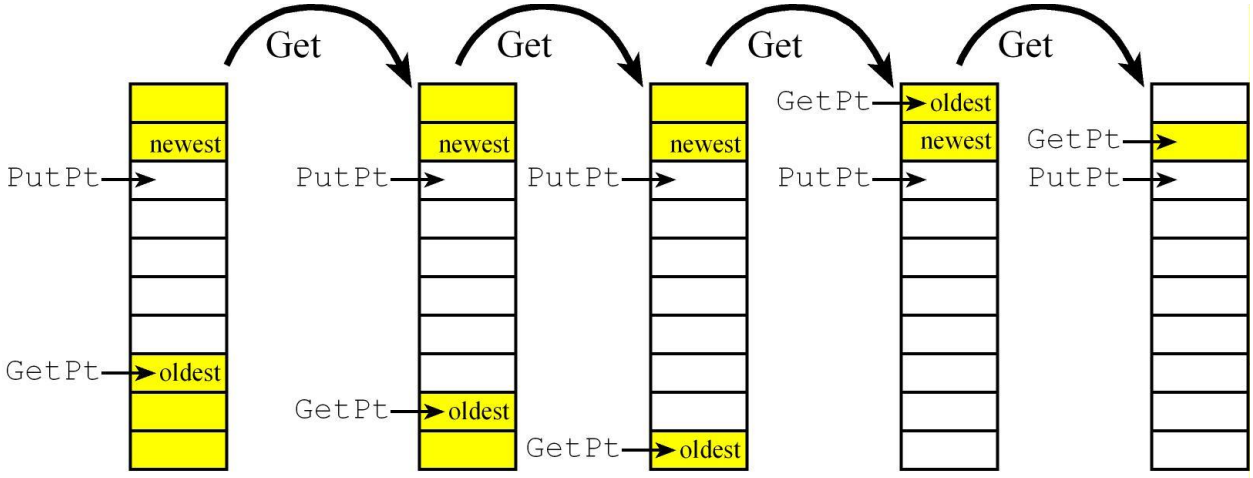


*Figure 3.10. The FIFO **Fifo_Get** operation showing the pointer wrap.*

We will deploy two semaphores to describe the status of the FIFO, see Program 3.5. In this FIFO, each element is a 32-bit integer. The maximum number of elements, **FIFOSIZE**, is determined at compile time. In other words, to increase the allocation, we first change **FIFOSIZE**, and then recompile.

The first semaphore, **CurrentSize**, specifies the number of elements currently in the FIFO. This semaphore is initialized to zero, meaning the FIFO is initially empty, it is incremented by **Fifo_Put** signifying one more element, and decremented by **Fifo_Get** signifying one less element.

The second semaphore, **RoomLeft**, specifies the how many more elements could be put into the FIFO. This semaphore is initialized to **FIFOSIZE**, it is decremented by **Fifo_Put** signifying there is space for one less element, and incremented by **Fifo_Get** signifying there is space for one more element. When **RoomLeft** is zero, the FIFO is full.

Race conditions and **critical sections** are important issues in systems using interrupts. If there are more than one producer or more than one consumer, access to the pointers represent a critical section, and hence we will need to protect the pointers using a **FIFOmutex** semaphore.

```
#define FIFOSIZE 10       // can be any size
uint32_t volatile *PutPt; // put next
uint32_t volatile *GetPt; // get next
uint32_t static Fifo[FIFOSIZE];
int32_t CurrentSize;   // 0 means FIFO empty
int32_t RoomLeft;      // 0 means FIFO full
int32_t FIFOmutex;     // exclusive access to FIFO
// initialize FIFO
void OS_Fifo_Init(void){
  PutPt = GetPt = &Fifo[0]; // Empty
  OS_InitSemaphore(&CurrentSize, 0);
  OS_InitSemaphore(&RoomLeft, FIFOSIZE);
  OS_InitSemaphore(&FIFOmutex, 1);
}
void OS_Fifo_Put(uint32_t data){
  OS_Wait(&RoomLeft);
  OS_Wait(&FIFOmutex);
  *(PutPt) = data;     // Put
  PutPt++;             // place to put next
  if(PutPt == &Fifo[FIFOSIZE]){
    PutPt = &Fifo[0];  // wrap
  }
  OS_Signal(&FIFOmutex);
  OS_Signal(&CurrentSize);
}
uint32_t OS_Fifo_Get(void){ uint32_t data;
  OS_Wait(&CurrentSize);
  OS_Wait(&FIFOmutex);
```

```
  data = *(GetPt);      // get data
  GetPt++;              // points to next data to get
  if(GetPt == &Fifo[FIFOSIZE]){
    GetPt = &Fifo[0];   // wrap
  }
  OS_Signal(&FIFOmutex);
  OS_Signal(&RoomLeft);
  return data;
}
```

Program 3.5. Two-pointer three-semaphore implementation of a FIFO. This implementation is appropriate when producers and consumers are main threads.

3.3.3. Two-semaphore FIFO implementation[edit]

If there is one producer as an event thread coupled with one or more consumers as main threads (Figure 3.11), the FIFO implementation shown in the previous section must be changed, because we cannot block or spin an event thread. If the FIFO is full when the producer calls **Put**, then that data will be lost. The number of times we lose data is recorded in **LostData**.
The **Put** function returns an error (-1) if the data was not saved because the FIFO was full.
This **Put** function cannot be called by multiple producers because of the read-modify-write sequence to **PutPt**. See Program 3.6. To tell if the FIFO is full, we simply compare the **CurrentSize** with its maximum. This is a statically allocated FIFO, so the maximum size is a constant.

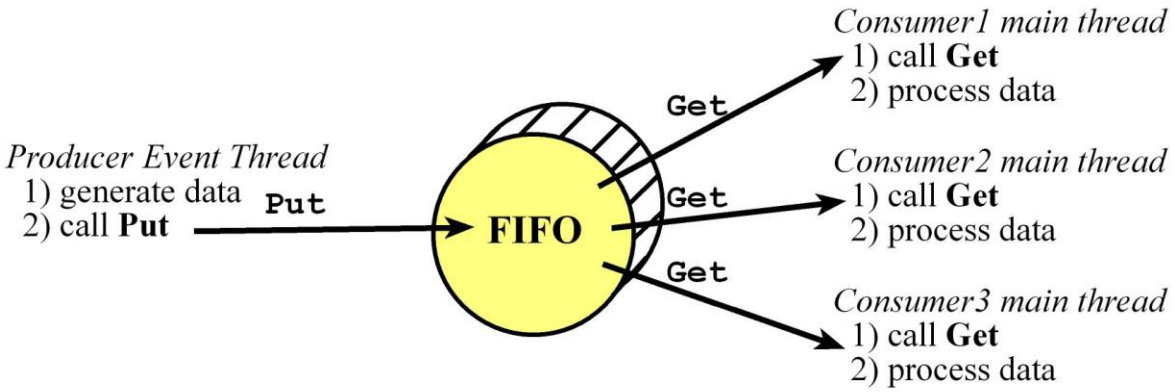


Figure 3.11. FIFO used to pass data from a single producer to multiple consumers. The producer is an event thread and the consumers are main threads.

```
#define FIFOSIZE 10       // can be any size
uint32_t volatile *PutPt; // put next
uint32_t volatile *GetPt; // get next
uint32_t static Fifo[FIFOSIZE];
int32_t CurrentSize;      // 0 means FIFO empty
int32_t FIFOmutex;        // exclusive access to FIFO
```

```
uint32_t LostData;
// initialize FIFO
void OS_Fifo_Init(void){
  PutPt = GetPt = &Fifo[0]; // Empty
  OS_InitSemaphore(&CurrentSize, 0);
  OS_InitSemaphore(&FIFOmutex, 1);
  LostData=0;
}
int OS_FIFO_Put(uint32_t data){
  if(CurrentSize == FIFOSIZE){
    LostData++;       // error
    return -1;
  }
  *(PutPt) = data;    // Put
  PutPt++;            // place for next
  if(PutPt == &Fifo[FIFOSIZE]){
    PutPt = &Fifo[0]; // wrap
  }
  OS_Signal(&CurrentSize);
  return 0;
}
uint32_t OS_FIFO_Get(void){uint32_t data;
  OS_Wait(&CurrentSize); // block if empty
  OS_Wait(&FIFOmutex);
  data = *(GetPt);    // get data
  GetPt++;            // points to next data to get
  if(GetPt == &Fifo[FIFOSIZE]){
    GetPt = &Fifo[0]; // wrap
  }
  OS_Signal(&FIFOmutex);
  return data;
}
```

Program 3.6. Two-pointer two-semaphore implementation of a FIFO. This implementation is appropriate when a single producer is running as an event thread and multiple consumers are running as main threads.

Note that, in this solution we no longer need the **RoomLeft** semaphore which was used to protect the multiple changes to **PutPt** that multiple producers would entail. A single producer does not have this problem. We still need the **CurrentSize** semaphore because we have multiple consumers that can change the **GetPt** pointer. The **FIFOmutex** semaphore is needed to prevent two consumers from reading the same data.

3.3.4. One-semaphore FIFO implementation[edit]

Streaming data from an event thread to a main thread

If there is one producer as an event thread coupled with one consumer as a main thread (Figure 3.12), we can remove the **mutex** semaphore. This **Get** function cannot be called by multiple consumers because of the read-modify-write sequence to **GetI**. In the previous FIFO implementations, we used pointers, but in this example we use indices, see Program 3.7. Whether you use pointers versus indices is a matter of style, and our advice is to use the mechanism you understand the best. As long as there is one event thread calling **Put** and one main thread calling **Get**, this implementation does not have any critical sections.

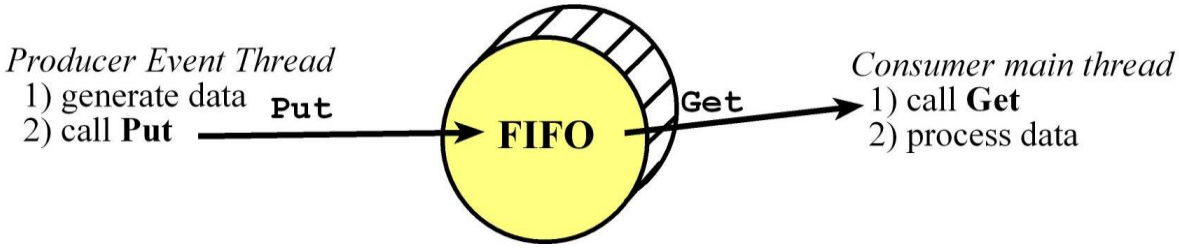


Figure 3.12. FIFO used to pass data from a single producer to a single consumer. The producer is an event thread and the consumer is a main thread.

```
#define FIFOSIZE 10  // can be any size
uint32_t PutI;       // index of where to put next
uint32_t GetI;       // index of where to get next
uint32_t Fifo[FIFOSIZE];
int32_t CurrentSize; // 0 means FIFO empty, FIFOSIZE means full
uint32_t LostData;   // number of lost pieces of data
// initialize FIFO
void OS_FIFO_Init(void){
  PutI = GetI = 0;   // Empty
  OS_InitSemaphore(&CurrentSize, 0);
  LostData = 0;
}
int OS_FIFO_Put(uint32_t data){
  if(CurrentSize == FIFOSIZE){
    LostData++;
    return -1;          // full
  } else{
    Fifo[PutI] = data; // Put
    PutI = (PutI+1)%FIFOSIZE;
    OS_Signal(&CurrentSize);
    return 0;           // success
  }
}
uint32_t OS_FIFO_Get(void){uint32_t data;
  OS_Wait(&CurrentSize);    // block if empty
  data = Fifo[GetI];        // get
```

```
  GetI = (GetI+1)%FIFOSIZE; // place to get next
  return data;
}
```

Program 3.7. Two-index one-semaphore implementation of a FIFO. This implementation is appropriate when a single producer is running as an event thread and a single consumer is running as a main thread.

This is the approach we recommend for Lab 3. The use of indexes rather than pointers also means all index arithmetic is a simple modulo the size of the FIFO to implement the wraparound.

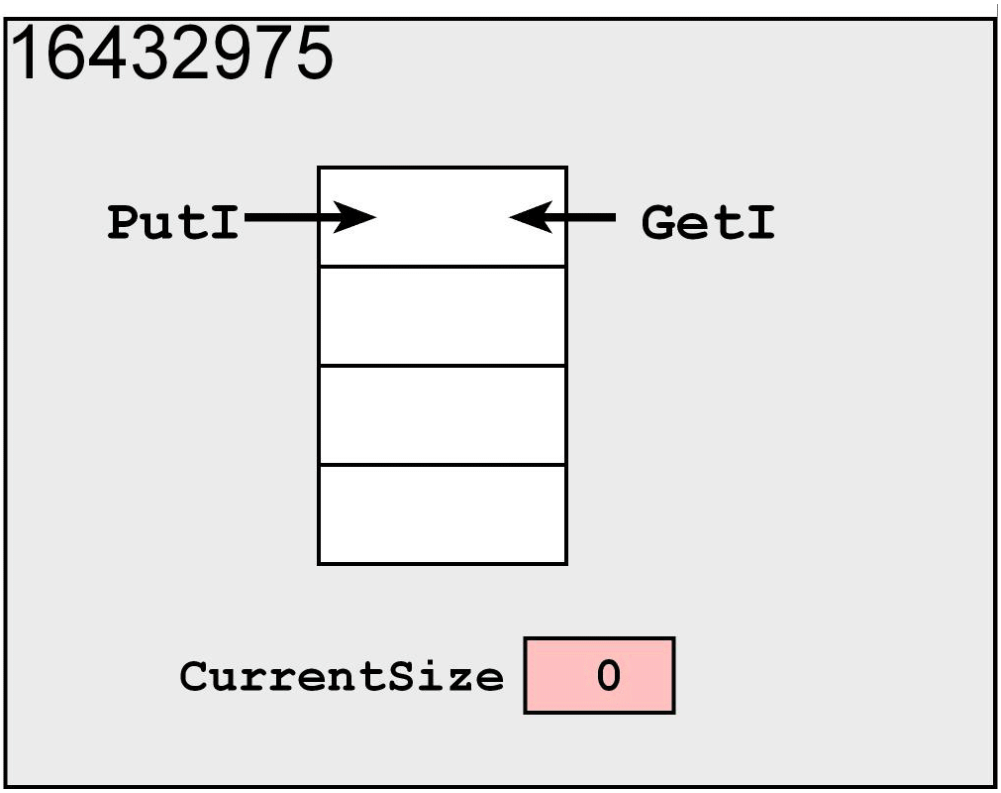


Figure 3.13. This FIFO can store a maximum of four elements, using one semaphore. ***Put*** *is called from an event thread, so it cannot block or spin.* ***Get*** *is called from a main thread, so it will block on the semaphore* ***CurrentSize*** *if the FIFO is empty.*

Thread sleeping[edit]

3.4. sleeping

Play Video

Sometimes a thread needs to wait for a fixed amount of time. We will implement an **OS_Sleep** function that will make a thread dormant for a finite time. A thread in the sleep state will not be run. After the prescribed amount of time, the OS will make the thread active again. Sleeping would be used for tasks which are not real-time. In Program 3.8, the **PeriodicStuff** is run approximately once a second.

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

Program 3.8. This thread uses sleep to execute its task approximately once a second.

To implement the sleep function, we could add a counter to each TCB and call it **Sleep**. If **Sleep** is zero, the thread is not sleeping and can be run, meaning it is either in the run or active state. If **Sleep** is nonzero, the thread is sleeping. We need to change the scheduler so that **RunPt** is updated with the next thread to run that is not sleeping and not blocked, see Program 3.9.

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

Program 3.9. Round-robin scheduler that skips threads if they are sleeping or blocked.

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

Notice that this implementation is not an exact time delay. When the sleep parameter is decremented to 0, the thread is not immediately run. Rather, when the parameter reaches 0, the thread is signified ready to run. If there are *n* other threads in the TCB list and the thread switch time is Δt, then it may take an additional $n*\Delta t$ time for the thread to the launched after it awakens from sleeping.

3.5.1. Basic principles[edit]

Basics of periodic timers

Play Video

Because time is a precious commodity for embedded systems there is a rich set of features available to manage time. If you connect a digital input to the microcontroller you could measure its

- Period, time from one edge to the next
- Frequency, number of edges in a fixed amount of time
- Pulse width, time the signal is high, or time the signal is low

If there are multiple digital inputs, then you can measure more complicated parameters such as frequency difference, period difference or phase.

Alternately, you can create a digital output and have the software set its

- Period
- Frequency
- Duty cycle (pulse-width modulation)

If there are multiple digital outputs, then you can create more complicated patterns that are used in stepper motor and brushless DC motor controllers. For examples of projects that manage time on the TM4C123 see examples at
http://users.ece.utexas.edu/~valvano/arm/#Timer
href="http://edx-org-utaustinx.s3.amazonaws.com/UT601x/ValvanoWareTM4C123.zip" target="_blank" rel="noreferrer noopener">http://edx-org-utaustinx.s3.amazonaws.com/UT601x/ValvanoWareTM4C123.zip
For all the example projects on the TM4C123/MSP432 download and unzip these projects:
http://edx-org-utaustinx.s3.amazonaws.com/UT601x/ValvanoWare.zip
However in this section, we present the basic principles needed to create periodic interrupts using the timer. We begin by presenting five hardware components needed as shown in Figure 3.14.

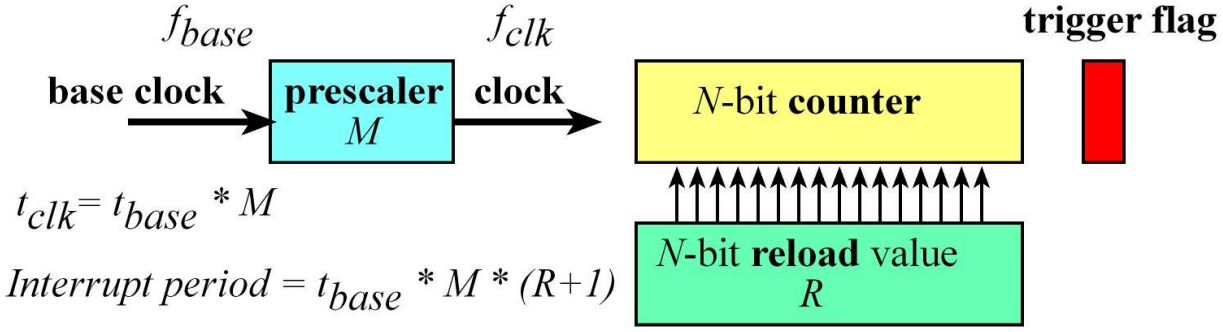


Figure 3.14. Fundamental hardware components used to create periodic interrupts.

The central component for creating periodic interrupts is a hardware counter. The counter may be 16, 24, 32, 48, or 64 bits wide. Let *N* be the number of bits in the counter. When creating periodic interrupts, it doesn't actually matter if the module counts up or counts down. However, most of the software used in this class will configure the counter to decrement.

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

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

The fourth component will be the base clock with which we control the entire hardware system. On the TM4C123, we will select the 80-MHz system clock. On the MSP432, we will select the 12-MHz **SMCLK**. In both cases, these clocks are derived from the crystal; hence timing will be both accurate and stable. Let f_{base} be the frequency of the base clock (80 MHz or 12MHz) and t_{base} be the period of this clock (12.5 ns or about 83.33 ns). The fifth component will be a prescaler, which sits between the base clock and the clock used to decrement the counter. Most systems create the prescaler using a modulo-M counter, where *M* is greater than or equal to 1. This way, the frequency and period of the clock used to decrement the counter will be

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

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

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

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

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

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

Remember R must be an integer less than 2*N*. Most timers have a limited choice for the prescale **M**. Luckily, most microcontrollers have a larger number of timers. The TM4C123 has six 32-bit timers and six 64-bit timers. The MSP432 has four 16-bit timers and two 32-bit timers. The board support package, presented in the next section, provides support for two

independent periodic interrupts. The BSP uses a separate 32-bit timer to implement the **BSP_Time_Get** feature.

Initialization software follows these steps.

1. Activate the base clock for the timer
2. Set the timer mode to continuous down counting with automatic reload
3. Set the prescale, *M*
4. Set the reload value, *R*
5. Arm the trigger flag in the timer
6. Arm the timer in the NVIC
7. Set the priority in the NVIC
8. Clear trigger flag
9. Enable interrupts (I=0)

3.5.2. Board support package

Periodic Interrupts in the BSP

Play Video

When using the periodic interrupt features in the BSP, the user or operating system writes a regular **void-void** C function. During initialization, we activate the periodic interrupt by passing a pointer to this **void-void** function. Furthermore, we specify the desired interrupt frequency (Hz) and hardware priority for this timer. The passed function is registered as a callback with the BSP, which it invokes at the specified frequency.

The priority is limited to 0 to 6, because priority 7 is reserved for the SysTick ISR used to switch main threads. We assign priority such that lower numbers signify higher priority. If an interrupt service routine is running at level *p* when another interrupt with priority less than p occurs, the processor will suspend the first ISR and immediately execute the higher priority ISR. If an interrupt service routine is running at level p when another interrupt with priority greater than or equal to p occurs, the processor will finish the higher priority ISR first, and then it will execute the lower/equal priority ISR. Thus, interrupts that have equal priority are handled sequentially in a first come first served manner.

One timer on the MSP432 uses 32 bits and the others use only 16 bits. Thus, the slowest frequency available on some of the MSP432 timers is 8 Hz. All timers on the TM4C123 support frequencies as slow as 1 Hz. The fastest frequency was capped at 10 kHz, in order to prevent one ISR from monopolizing the processor. Regardless of the interrupt rate, it is important to estimate the utilization of each periodic task. Maximum utilization is defined as the ratio of the maximum execution time of the ISR (Δt) divided by the interrupt period, P:

Max utilization = 100 * Δt / P (in percent)

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

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

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

Each timer uses a base clock, prescaler, and finite number of bits in the counter as shown in Table 3.2. Let *freq* be the desired interrupt frequency, *freq*=1/*P*. The initialization routine will calculate the reload value, *R*, using base clock frequency, f_{base} and prescale *M*. The frequency of the counter clock, f_{clk}, is the base clock divided by the prescale

$$f_{clk} = f_{base} / M$$

Because the reload value must be an integer, best results occur if *fclk/freq* is an integer. The reload value will be:

$$R = fclk/freq - 1$$

BSP function	processor	*N*	f_{base}
BSP_PeriodicTask_Init	TM4C123	64	80 MHz
BSP_PeriodicTask_InitB	TM4C123	64	80 MHz
BSP_PeriodicTask_InitC	TM4C123	64	80 MHz
BSP_PeriodicTask_Init	MSP432	32	12 MHz
BSP_PeriodicTask_InitB	MSP432	16	12 MHz
BSP_PeriodicTask_InitC	MSP432	16	12 MHz

Table 3.2. Internal parameters of the periodic task feature implemented in the BSP.