Inside Real-Time Kernels

This two-part class examines the inner workings of a real-time kernel. By understanding internals, you will have a better grasp on the issues involved when using a kernel. This class uses a combination of code, graphics, animations and, running examples on an actual target CPU to show you how a kernel works. You will see how a scheduler, a context switch, and many other common kernel services are typically implemented. This is the best class on the subject.

This two-part class examines the inner workings of a real-time kernel. A real-time kernel (also known as an RTOS) is software that manages the time of a microprocessor, microcontroller or DSP, allows multitasking and, provides valuable services to your embedded application. By understanding internals, you will have a better grasp on the issues involved when using a kernel. This class uses a combination of code, graphics, animations and, running examples on an actual target CPU to show you how a kernel works. You will experience the inner workings of a scheduler, see what's involved in a context switch, examines how semaphores, message queues, and many other common kernel services are typically implemented. This is the best class on the subject.

Jean J. Labrosse
Micriμm

Jean Labrosse is President of Micrium, a provider of high quality embedded software solutions. Mr. Labrosse is a regular speaker at the Embedded Systems Conferences and serves on the Advisory Board of the conference. Jean is the author of two books: MicroC/OS-II, The Real-Time Kernel and, Embedded Systems Building Blocks, Complete and Ready-to-Use Modules in C and has written numerous articles for magazines. He has an MSEE and has been designing embedded systems for many years.

INTRODUCTION

A kernel is software that manages the time of a microprocessor to ensure that all time critical events are processed as efficiently as possible. A kernel simplifies the design process of the system because it allows the system to be divided into multiple independent elements called *tasks*. A task is a simple program which thinks it has the microprocessor all to itself. Each task is given a priority based on its importance. The design process for a real-time system consists of splitting the work to be done into tasks which are responsible for a portion of the problem. The microprocessor still has the same amount of work to do but, now the work can be prioritized. A kernel also provides valuable services to your application such as time delays, system time, message passing, synchronization, mutual-exclusion and more.

Kernels come in two flavors: *non-preemptive* and *preemptive*. Non-preemptive kernels require that each task does something to explicitly give up control of the CPU (i.e. microprocessor). To maintain the illusion of concurrency, this process must be done frequently. Asynchronous events are handled by ISRs (Interrupt Service Routines). An ISR can make a higher-priority task ready to run but the ISR always returns to the interrupted task. The new higher-priority task will gain control of the CPU only when the current task gives up the CPU. Non-preemptive kernels are seldom used in real-time applications. On the other hand, a preemptive kernel is used when system responsiveness is important. With a preemptive kernel, the kernel itself ensures that the highest-priority task ready to run is always given control of the CPU. When a task makes a higher-priority task ready to run, the current task is *preempted* (suspended) and the higher-priority task is immediately given control of the CPU. If an ISR makes a higher-priority task ready, when the ISR completes, the interrupted task is suspended and the new higher-priority task is resumed. The execution profile of a system designed using a *preemptive* kernel is shown in figure 1. As shown, a low priority task is executing ①. An asynchronous event interrupts the microprocessor ②. The microprocessor services the event ③ which makes a higher priority task ready for execution. Upon completion, the ISR invokes the kernel which decides to run the more important task ④. The higher priority task executes to completion unless it also gets interrupted ⑤. At the end of the task, the kernel resumes the lower priority task ⑥. The lower priority task continues to execute ⑦.

A preemptive kernel can ensure that time critical tasks are performed first. In other words, without a kernel we may not get to perform a task that is time critical because we would be working on something that is not. Furthermore, execution of time critical tasks are deterministic and are almost insensitive to software changes.

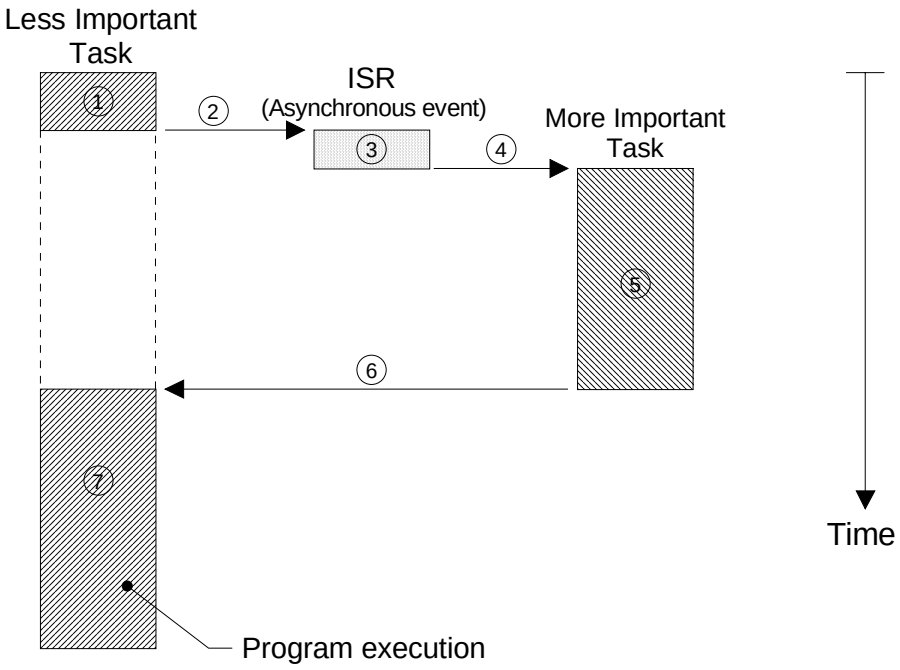

Figure 1

TASKS

A task is basically an infinite loop and looks as shown below. In order for the kernel to allow other tasks to perform work, you must invoke a service provided by the kernel to wait for some event to occur. The event can either be time to elapse or the occurrence of a signal from either another task or an ISR.

```
void Task(void)
{
    while (1) {
        /* Perform some work (Application specific)                    */
        /* Wait for event by calling a service provided by the kernel  */
        /* Perform some work (Application specific)                    */
    }
}
```

At any given time, a task is in one of six states: *Dormant*, *Ready*, *Running*, *Delayed*, *Waiting for an event* or *Interrupted*. The 'dormant' state corresponds to a task which resides in memory but has not been made available to the multitasking kernel. A task is 'ready' when it can execute but its priority is less than the current running task. A task is 'running' when it has control of the CPU. A task is 'delayed' when the task is waiting for time to expire. A task is 'waiting for an event' when it requires the occurrence of an event: waiting for an I/O operation to complete, a shared resource to be available, a timing pulse to occur, etc. Finally, a task is 'interrupted' when an interrupt occurred and the CPU is in the process of servicing the interrupt.

You must invoke a service provided by the kernel to have the kernel manage each task. This service *creates* a task by taking it from its 'dormant' state to the 'ready' state. Each task requires its own stack and where the task is concerned, it has access to most CPU registers. The function prototype for a task creation function is shown below.

```
void OSTaskCreate(void (*task)(void), void *stack, UBYTE priority)
```

'`task`' is the function name of the task to be managed by the kernel. '`stack`' is a pointer to the top-of-stack of the stack to be used by the task. '`priority`' is the task's relative priority - its importance with respect to all other tasks in the system.

The kernel keeps track of the state of each task by using a data structure called the *Task Control Block (TCB)*. The TCB contains the status of the task (Ready, Delayed or Waiting for an event), the task priority, the pointer to the task's top-of-stack and other kernel related information. Figure 2 shows the relationship between the task stacks, the TCBs and the CPU registers. Based on events, the kernel will *switch* between tasks. This process basically involves saving the contents of the CPU registers (i.e. the CPU *context*) onto the current task stack, saving the stack pointer into the current task's TCB, loading the stack pointer from the new task's TCB and finally, loading the CPU registers with the context of the more important task. This process is called a *context switch* (or *task switch*).

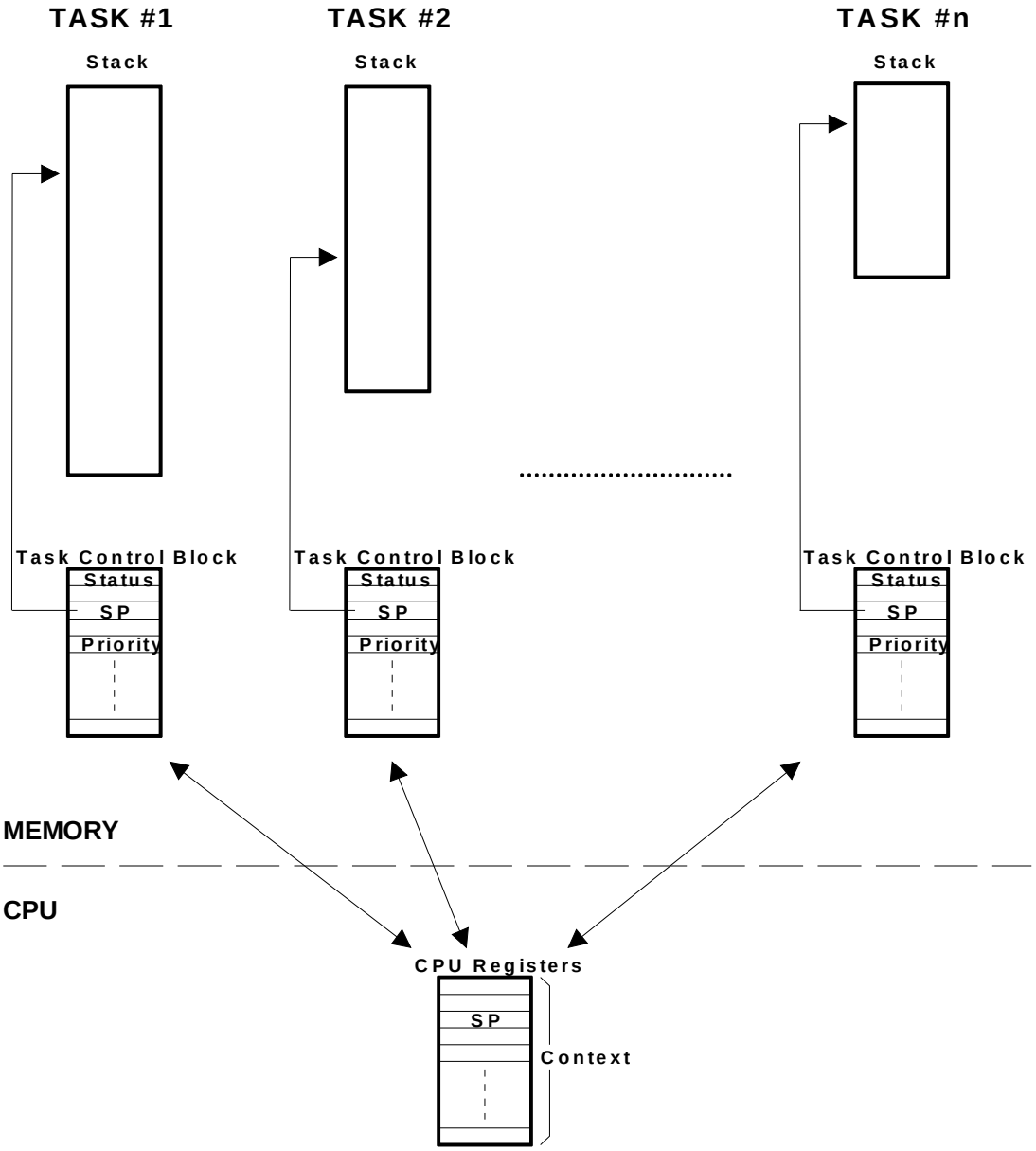

Figure 2

THE READY LIST

One of the kernel's function is to maintain a list of all the tasks that are ready-to-run in order of priority as shown in figure 3. This particular list is called the *Ready List*. When the kernel decides to run a more important task it basically picks the TCB at the head of the ready list. This process is called *Scheduling*. As you can see, finding the most important task in the list is fairly simple. The problem with using a singly-linked

list as shown in figure 3 is that the kernel may have to go through the whole list in order to insert a low priority TCB. This actually happens every time a high priority task preempts a lower priority task. Linked lists are only used here for purpose of illustration. There are actually better techniques to insert and remove TCBs from a list instead of using linked lists. This is, however, a topic that is beyond the scope of this paper.

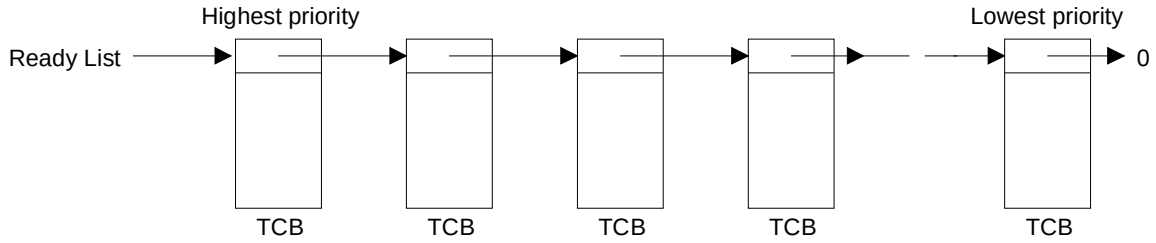

Figure 3

EVENT MANAGEMENT

The kernel provides services to allow a task to suspend execution until an event occurs. The most common type of event to wait for is the semaphore. A semaphore is used to either control access to a shared resource (mutual exclusion), signal the occurrence of an event or allow two tasks to synchronize their activities. A semaphore generally consist of a value (an unsigned 16-bits variable) and a waiting list (see figure 4).

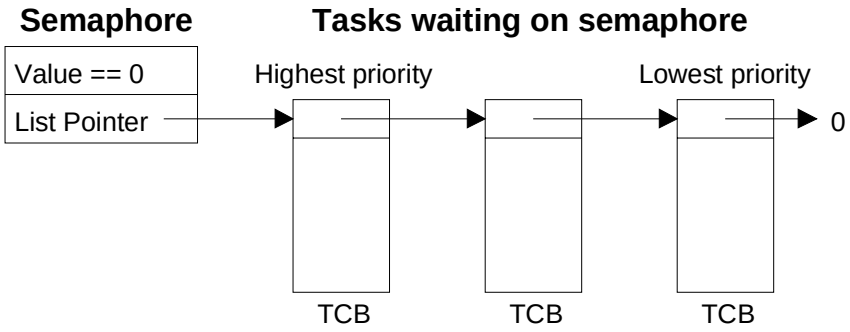

Figure 4

The semaphore must be initialized to a value through a service provided by the kernel before it can be used. Because the kernel provides multitasking, a resource such as a printer must be protected from simultaneous access by two or more tasks. Because you only have one printer, you initialize the semaphore with a value of 1. A task desiring access to the printer performs a WAIT operation on the semaphore. If the semaphore value is 1, the semaphore value is decremented to 0 and the task continues execution. At this point, the task 'owns' the printer. If another task needs access to the printer, it must

also perform a WAIT operation. This time, however, the semaphore value is 0 which indicates that the printer is in use. In this case, the task is removed from the ready list and placed in the semaphore waiting list. When the first task is done with the printer, it performs a SIGNAL operation which frees up the printer. At this point, the task waiting for the printer it is placed in the ready list. If the task that was waiting for the printer is more important than the current task, the current task will be preempted and the task waiting for the printer will be given control of the CPU. The pseudo code for the semaphore management functions is shown below.

```
UBYTE OSSemWait(OS_EVENT *sem, UWORD to)
{
    Disable Interrupts;
    if (sem->Value > 0) {
        sem->Value--;
        Enable Interrupts;
    } else {
        Remove task's TCB from ready list;
        Place task's TCB in the waiting list for the semaphore;
        Wait for semaphore to be 'signaled' (execute next task);
        if (timeout occurred) {
            Enable Interrupts;
            return (timeout error code);
        } else {
            Enable Interrupts;
            return (no error);
        }
    }
}


void OSSemSignal(OS_EVENT *sem)
{
    Disable Interrupts;
    if (any tasks waiting for the semaphore) {
        Remove highest priority task waiting for the sem. from wait list;
        Place the task in the ready list;
        Enable Interrupts;
        Call the scheduler to run the highest priority task;
    } else {
        sem->Value++;
        Enable Interrupts;
    }
}
```

Kernels generally support other types of events such as message mailboxes, message queues, event flags, pipes, etc. The operation of these other types of events is very similar to semaphores. A message mailbox is a data structure containing a pointer to a user definable message along with a waiting list. Tasks are placed in the waiting list until messages are sent by either another task or an ISR. A message queue is a data structure containing a list of pointers to messages arranged as a FIFO (First-In-First-Out) along with a waiting list. Tasks are placed in the waiting list until another task or an ISR deposits one or more messages in the FIFO. Event flags are grouping of single-bit flags (on or off) and a waiting list. Tasks are placed in the waiting list until one or more bits

are set. Pipes are character size FIFOs with a waiting list. Again, tasks are placed in the pipe's waiting list until characters are received.

SYSTEM TICK

Every kernel provides a mechanism to keep track of time. This is handled by a hardware timer which interrupts the CPU periodically. The ISR for this timer invokes a service provided by the kernel which is responsible for updating internal time dependant variables. Specifically, a number of services rely on 'timeouts'. This ISR is generally called the *clock tick* ISR. The clock tick interrupt is usually set to occur between 10 and 100 times per second. A task can thus suspend its execution for an integral number of 'clock ticks'. A special list is used to keep track of tasks that are waiting for time to expire. This list is called the *delayed task list*. The delayed task list can be set up as a *delta list* which basically orders delayed tasks so that the first task in the list has the least amount of time remaining to expire. For example, if you had five tasks with values of 10, 14, 21, 32 and 39 tenths of a second remaining to timeout then, the list would contain 10, 4, 7, 11 and 7. The total time before the first task to expire is 10, the second is 10+4, the third is 10+4+7, the fourth is 10+4+7+11 and finally, the fifth task would be 10+4+7+11+7.

INTERRUPT PROCESSING

Kernels provide services that are accessible from ISRs to notify tasks about the occurrence of events. In order to use these services, however, your ISRs must save all CPU registers and notify (through a function) the kernel that an ISR has been entered. In this case, the kernel simply increments a nesting counter (generally an 8-bit variable). The nesting counter is used to determine the nesting level of interrupts (how many interrupts were interrupted). Upon completion of the ISR code, your ISR must invoke another service provided by the kernel to notify it that the ISR has completed. This service basically decrements the nesting counter. When all interrupts have returned to the first interrupt nesting level, the kernel determines if a higher priority task has been made ready-to-run by one of the ISRs. If the interrupted task is still the most important task to run, the CPU registers are restored and the interrupted task is resumed. If a higher priority task is ready-to-run, the kernel saves the stack pointer of the interrupted task into its TCB, obtains the stack pointer of the new task, restores the CPU registers for the new task and resumes execution of the new task. This process is shown in figure 5.

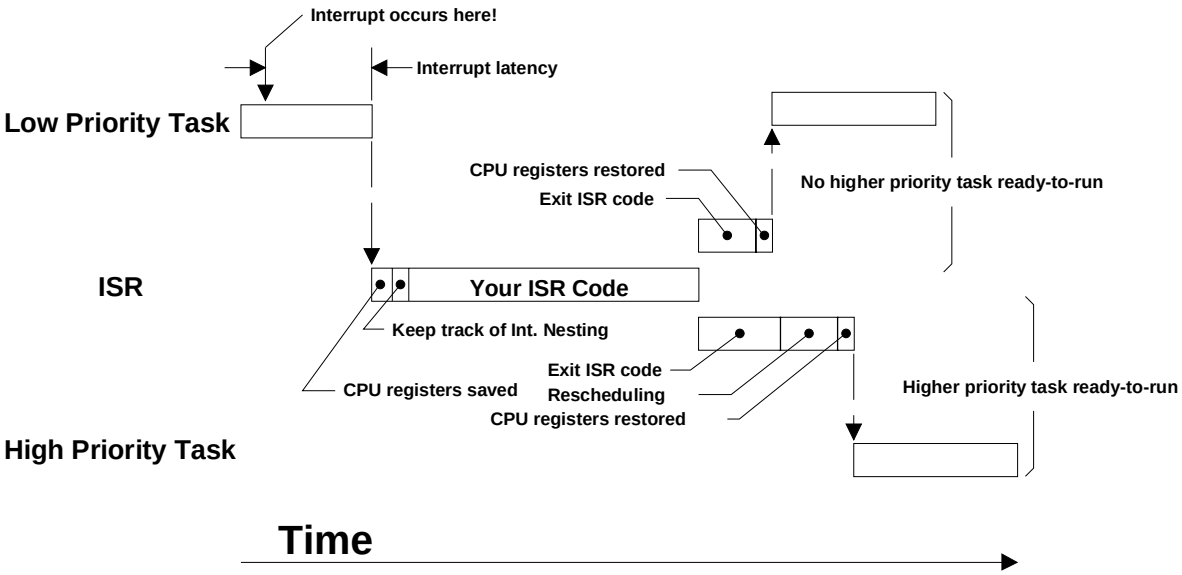

Figure 5

CONCLUSION

A kernel allows real-time applications to be easily designed and expanded; other functions could be added without requiring major changes to the software. A large number of applications can benefit from the use of a kernel. Kernels can ensure that all time critical events are handled as quickly and efficiently as possible. A minimum kernel requires only about 1 to 3 KBytes of ROM and a few hundred bytes of RAM. Some kernels even allow you to specify the size of each task's stack on a task-by-task basis. This feature helps reduce the amount of RAM needed in your application. A common misconception about a kernel is that it adds an unacceptable amount of overhead on your CPU. In fact, a kernel will only require between 1 and 4% of your CPU's time in exchange for valuable services. There are currently about 50 kernel vendors. Products are available for 8, 16 and 32 bit microprocessors. Some of these packages are complete operating systems and include a real-time kernel, an input/output manager, windowing systems, a file system, networking, language interface libraries, debuggers and cross-platform compilers. The cost of a kernel varies between $100 to well over $10,000. By buying a kernel you get the vendor’s expertise in the business, the support, the documentation and the piece of mind. You should, however, ensure that the kernel vendor guaranties that his product doesn’t contain any bugs. Furthermore, if you do find a bug you should hope that the vendor will not charge you to have it fixed! If you have a choice, don’t design your own kernel! Many people (including myself) have under estimated the intricacies associated with a kernel.

BIBLIOGRAPHY

Comer, Douglas
Operating System Design, The XINU Approach
Englewood Cliffs, New Jersey
Prentice Hall, Inc. 1984
ISBN 0-13-637539-1

Labrosse, Jean J.
µC/OS-II, The Real-Time Kernel, 2nd Edition
Lawrence, Kansas
R&D Technical Books, 2002
ISBN 1-57820-103-9

Labrosse, Jean J.
Embedded Systems Building Blocks, Complete and Ready-to-Use modules in C
Lawrence, Kansas
R&D Technical Books, 2000
ISBN 0-87930-604-1