Micriμm, Inc.

The 10-Minute Guide to RTOS

Application Note

AN-1004

Jean J. Labrosse
Jean.Labrosse@Micrium.com
www.Micrium.com

Summary
A Real-Time Operating System (RTOS) is software that manages the time of a microprocessor or microcontroller. This application note is a reprint from an article I published in Electronic Design Magazine, June 1994 issue.

Introduction
Real-time systems are characterized by the fact that severe consequences can result if logical as well as timing correctness properties of the system are not met. A real-time multitasking application is a system in which several time critical activities must be processed simultaneously. A real-time multitasking kernel (also called a real-time operating system, RTOS) is software which ensures that time critical events are processed as efficiently as possible. The use of a RTOS generally simplifies the design process of a system by allowing the application to be divided into multiple independent elements called tasks.

Foreground/Background.
Systems which do not use a RTOS are generally designed as shown in figure 1. These systems are called foreground/background. An application consist of an infinite loop which calls application modules to perform the desired operations. The modules are executed sequentially (background) with interrupt service routines (ISRs) handling asynchronous events (foreground). Critical operations must be performed by the ISRs to ensure that they are dealt with in a timely fashion. Because of this, ISRs have a tendency to take longer than they should. Information for a background module made available by an ISR is not processed until the background routine gets its turn to execute. The latency in this case depends on how long the background loop takes to execute.

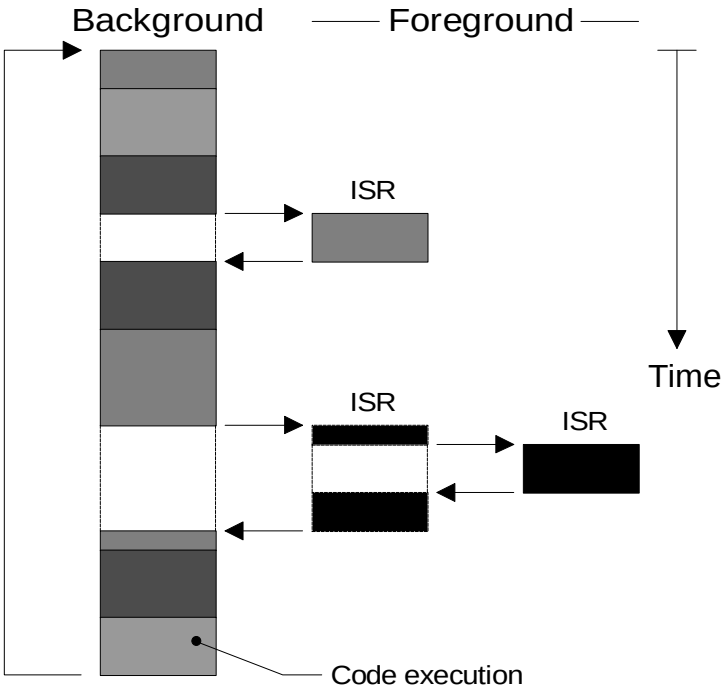

Figure 1. Foreground/Background application

REAL-TIME KERNELS

Multitasking.
Multitasking is the process of scheduling and switching the CPU between several tasks; a single CPU switches its attention between several sequential tasks. Multitasking provides the ability to structure your application into a set of small, dedicated tasks that share the processor. One of the most important aspects of multitasking is that it allows the application programmer to manage complexity which is inherent in real-time applications. Real-time kernels can make application programs easier to design and maintain. A task is a simple program which thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into tasks which are responsible for a portion of the problem.

Kernel.
The kernel is the part of the multitasking system responsible for the management of tasks and communication between tasks. When the kernel decides to run a different task, it simply saves the current task's context (CPU registers) onto the current task's stack; each task is assigned its own stack area in memory. Once this operation is performed, the new task's context is restored from its stack area and then, execution of the new task's code is resumed. This process is called a context switch or a task switch. The current top of stack for each task, along with other information, is stored in a data structure called the Task Control Block (TCB). A TCB is assigned to each task when the task is created and is managed by the RTOS.

Interrupts.
An important issue in real-time systems is the time required to respond to an interrupt and to actually start executing user code to handle the interrupt. All RTOSs disable interrupts when manipulating critical sections of code. The longer a RTOS disables interrupts, the higher the interrupt latency. RTOSs generally disable interrupts for less than about 50 uS but obviously, the lower the better. The code that services an interrupt is called an *Interrupt Service Routine* (ISR).

Scheduler.

The scheduler, also called the dispatcher, is the part of the kernel which is responsible for determining which task will run next and when. Most real-time kernels are priority based; each task is assigned a priority based on its importance. Establishing the priority for each task is application specific. In a priority based kernel, control of the CPU will always be given to the highest priority task ready to run. When the highest priority task gets the CPU, however, depends on the type of scheduler used. There are two types of schedulers: *non-preemptive* and *preemptive*.

Non-preemptive scheduling.

Non-preemptive schedulers require that each task does something to explicitly give up control of the CPU. To maintain the illusion of concurrency, this process must be done frequently. Non-preemptive scheduling is also called *cooperative multitasking*; tasks cooperate with each other to gain control of the CPU. When a task relinquishes the CPU, the kernel executes the code of the next most important task that is ready to run. Asynchronous events are still handled by ISRs. An ISR can make a higher priority task ready to run but the ISR always return to the interrupted task. The new higher priority task will gain control of the CPU only when the current task voluntarily gives up the CPU. This is illustrated in figure 2. Latency of a non-preemptive scheduler is much lower than with Foreground/Background; latency is now given by the time of the longest task.

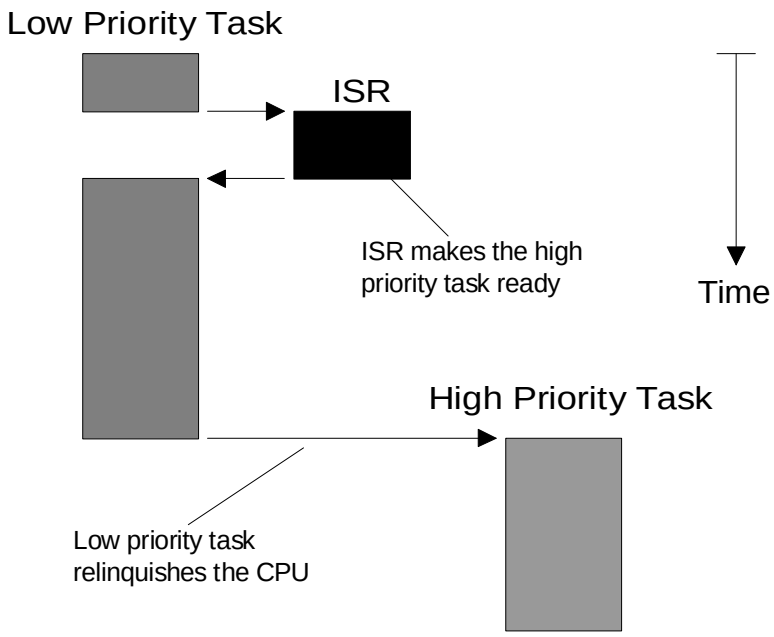

Figure 2. Non-preemptive scheduling

Preemptive scheduling.
In a preemptive kernel, when an event makes a higher priority task ready to run, the current task is immediately suspended and the higher priority task is given control of the CPU. If an ISR makes a higher priority task ready to run, the interrupted task is suspended and the new higher priority task is resumed. Most real-time systems employ preemptive schedulers because they are more responsive. Preemptive scheduling is illustrated in figure 3.

Reentrancy.
A *reentrant* function is a function which can be used by more than one task without fear of data corruption. Conversely, a *non-reentrant* function is a function which cannot be shared by more than one task unless mutual exclusion to the function is ensured by either using a semaphore (described later) or, by disabling interrupts during critical sections of code. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables (CPU registers or variables on the stack) or protect their data when global variables are used. Compilers specifically designed for embedded software will generally provide reentrant run-time libraries. Non-preemptive schedulers do not require reentrant functions unless a function is shared between a task and an ISR. Preemptive schedulers requires you to have reentrant functions if the functions are shared by more than one task.

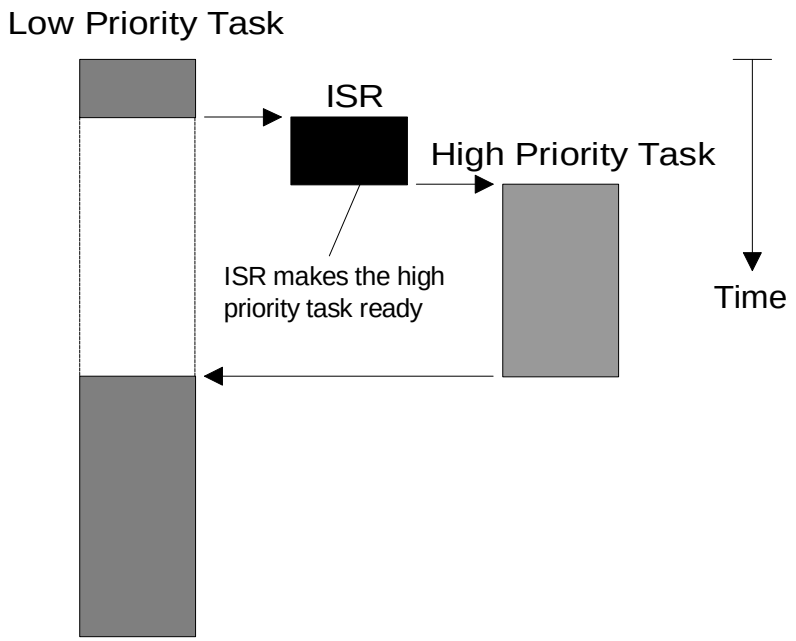

Figure 3. Preemptive scheduling

Kernel services.
Real-time kernels provide services to your application. One of the most common services provided by a kernel is the management of semaphores. A semaphore is a protocol mechanism used to control access to a shared resource (mutual exclusion), signal the occurrence of an event or allow two tasks to synchronize their activities. A semaphore is basically a key that your code acquires in order to continue execution. If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner. A suspended task consumes little or no CPU time.

Kernels also provide time related services allowing any task to delay itself for an integral number of system clock ticks. A clock tick typically occurs every 10 to 200 mS depending on the application requirements.

It is sometimes desirable for a task or an ISR to communicate information (i.e. send messages) to another task. This is called intertask communications and services for sending and receiving messages are generally provided by most kernels. The two most common kernel services for sending messages are the message mailbox and message queue. A message mailbox, also called a message exchange is typically a pointer size variable. Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox. Both the sending and receiving task will agree as to what the pointer is actually pointing to; they will agree on the message content. A message queue is used to send more than one message to a task. A message queue is basically an array of mailboxes.

Commercial RTOSs.
There are currently about 100 RTOS 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 RTOS varies between $100 to well over $10,000. With so many vendors, the difficulty is in the selection process.

Small embedded systems.
Many small embedded systems such as engine controls, intelligent instruments, robots, computer peripherals and telecommunications equipment can benefit from the use of a RTOS. Such systems are generally designed around an 8 bit microprocessor. With a 64 KBytes address space, most 8 bit microprocessors cannot afford a memory hungry RTOS. Commercial kernels are available that require only about 1 to 3 KBytes of ROM. 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 RTOS is that it adds an unacceptable amount of overhead on your CPU. In fact, a RTOS will only require between 1 and 4% of your CPU's time in exchange for valuable services. Other features for a small RTOS are:

- Low cost.
- Have minimum interrupt latency.
- Deterministic execution time for all kernel services.
- Be able to manage at least 20 tasks.
- Provide at least the following services.
- Allow tasks to be dynamically created and deleted.
- Provide semaphore management services.
- Allow time delays and timeouts on kernel services.

Conclusion.

A RTOS 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 RTOS. RTOSs can ensure that all time critical events are handled as quickly and efficiently as possible. Once you use a RTOS for an application, you will not want to do without one!

Contact Information

Micriμm, Inc.
949 Crestview Circle
Weston, FL 33327
954-217-2036
954-217-2037 (FAX)
e-mail: Jean.Labrosse@Micrium.com
WEB: www.Micrium.com