An Introduction to Real-time Operating Systems

Quadros Systems, Inc.

www.quadros.com

Table of Contents

1 RTOSes, Kernels and Executives 3
2 Reasons to Use an RTOS 3
3 RTOS Properties and Functions 3
4 System Resource Management 4
5 Hard and Soft Real-time 4
6 Multitasking 4
7 Priority and Preemption 5
8 Scheduling Models 5
9 Kernel Classes 6
10 Interrupt Handling 6
11 Responsiveness and Latency 6
12 More Information 6

Copyright © 2006-2009. All Rights Reserved

1 RTOSes, Kernels and Executives

A real-time operating system or RTOS (sometimes known as a real-time executive or kernel) is a library of functions that implements rules and policies concerning time-critical allocation of a computer system's resources. The intent of this document is to provide a top-level overview of real-time operating systems. If you would like information on how these concepts are implemented in a specific RTOS, you can find detailed information on the RTXC Quadros RTOS by visiting www.quadros.com/resources.

Definition

The RTOS determines which applications should run in what order and how much time should be allowed for each application before giving processor access to another process:

- manages the sharing of internal memory among multiple tasks.
- handles input and output to and from attached hardware devices, such as serial ports, buses, and I/O device controllers.
- sends messages about the status of operation and any errors that may have occurred

The hallmark of a well-designed RTOS is predictable performance. This is best achieved by the consistent application of *Policies* and *Rules*. *Policies* guide the design of an RTOS. *Rules* implement those policies and resolve policy conflicts. Neither can be violated without indeterminate or catastrophic results to the system's operation.

An example of a policy would be the requirement for the design to be deterministic (predictable). An example of a rule would be the requirement that threaded lists, permitting random order of node deletion, be implemented as doubly-linked lists. This rule is an implementation of the policy of deterministic design. The double links permit direct access to a node during the deletion process, making it a predictable procedure.

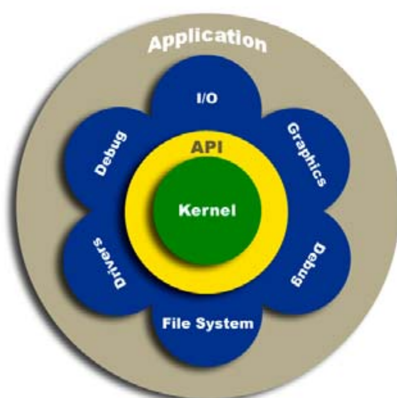

Figure 1. The RTOS handles input and output to and from attached hardware devices.

Rules permit software processes to operate and gain access to system resources in an orderly manner. Access to functions in the RTOS library may take several forms (usually calls to functions). The functions, called kernel services, embody and enforce these rules to ensure order in the application processes that use them.

2 Reasons to Use an RTOS

A well-designed RTOS provides a number of tangible benefits to the developer. It

- abstracts away the complexities of the processor,
- provides a solid infrastructure constructed of rules and policies that provide consistency and repeatability
- simplifies development and improves developer productivity by utilizing high level kernel objects to easily handle complex functions
- integrates and manages resources needed by communications stacks and middleware (TCP/IP, USB, SDIO, CAN, FAT and Flash file systems, etc.)
- optimizes use of system resources and improves product reliability, maintainability and quality

An RTOS can bring all those elements together into a platform that allows the application developer to begin development at a much higher point, enabling a shorter time-to-market with higher reliability and lower risk.

3 RTOS Properties and Functions

Any competent RTOS must do certain things in order to justify its use. It needs to:

- manage the processor and other system resources to meet the requirements of the application
- be able to respond to, and synchronize with, events
- be able to move data efficiently between processes
- be able to manage the demands of the process with respect to an independent variable such as time
- perform in a predictable manner with operations that take place within a predictable about of time

While the capabilities above are primary requirements, there are secondary requirements that are not necessarily mandatory, including the ability to:

- provide efficient management of RAM
- provide for exclusive access to resources

In order to meet these requirements, both primary and secondary, a kernel includes the following components:

- a *scheduler* that determines which programs get access to the CPU and in what order
- a *function library* serving as an interface between the application code and the kernel
- a *library of services* that operate on classes of data objects to cause desired program behavior
- a set of *user-defined data* objects, representing the needs of the application, on which the kernel services operate

Copyright © 2006. All Rights Reserved

4 System Resource Management

The primary function of the RTOS is to manage certain system resources, such as the CPU, memory, and time. Each resource must be shared among the competing processes to accomplish the overall function of the system:

- System memory is a finite resource and therefore must be shared.
- Because the CPU operates much faster than the physical process it is controlling or monitoring, the CPU can be shared to prevent delays in processing. Such delays would violate a basic system policy.
- Time is the most difficult and unforgiving resource managed by the kernel.

Kernel services must be designed and coded to require minimal execution time yet remain predictable. Execution speed of the kernel services determines the responsiveness of the system to changes in the physical process. However, it is equally important that each service be as deterministic as possible (predictable) with respect to time. Without predictable timing, a system designer has no assurance that the time constraints of the physical process will be met. The key to sharing system resources is the kernel's use of various classes of kernel objects.

5 Hard and Soft Real-time

A task that has operational time constraints that must be met in order to avoid a catastrophic failure is called a *hard real-time* task. A system can have several such tasks and the key to their correct operation lies in scheduling them so that they meet their time constraints. That necessarily involves *a priori* setting of priorities to them and then analyzing each one with respect to the others to determine if a feasible schedule exists. A feasible schedule in a hard real-time system is one in which all tasks meet their known time constraints. In short, the basic property of the hard real-time elements of a system is that they are *predictable*.

In applications involving soft real-time, timing constraints of those elements are looser than those of hard real-time to the extent that even a failure of a task to meet its time requirements still provides some value to the application. In essence, the soft real-time task does not offer a guarantee to meet its time constraint, but only that it will make a "best effort" attempt to do so.

6 Multitasking

Multitasking is one of the major policies implemented in a modern RTOS. Most modern real-time kernels make use of the work done in the mid-1960s by Dr. Edsger Dijkstra. Although multitasking was an acknowledged concept before that time, Dr. Dijkstra's work had the greatest impact on the industry because it formulated a set of constructs and rules for implementing the concept.

Given that a computer is usually faster than the processes that cause the events to which it responds, it is still impossible for all but the most specialized CPUs to simultaneously execute different code entities. However, by decomposing the functions of the system into different program code sets and rapidly switching the CPU between them, the designer can achieve the effect of concurrency. The computer achieves and maintains its highest efficiency by using idle system time in one code entity (while, for example, waiting for some particular event to occur) to run another code entity. This rapid switching produces the appearance of simultaneous execution of multiple program code sets.

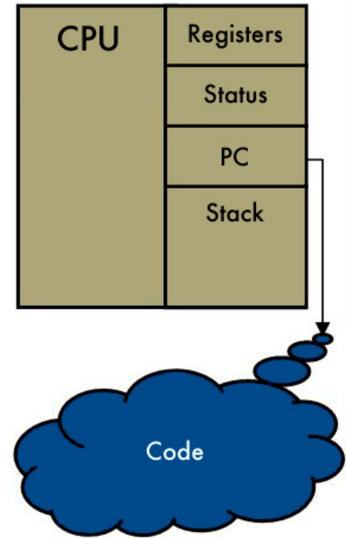

Figure 2a.
Physical Processor

Switching from one code entity (Task) to another forms the basis of multitasking. The multitasking policy of the kernel's scheduler determines the method of switching. In a multi-tasking configuration there is one CPU that must be shared among a set of tasks. Figure 2a shows the typical processor model, consisting of a CPU, some registers, a processor status, a program counter (PC), and a stack.

To share the physical processor, each task needs to have the same properties as the physical processor. Thus a task must have a set of registers, a status, a PC to point to the next executable instruction of the task and a stack for its local variables. Of course, each task will have its set of code that it is executing. The task, therefore, can be considered a virtual processor, awaiting its opportunity to have its properties switched into the physical processor.

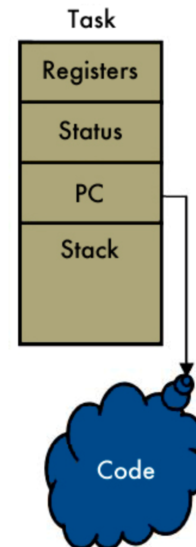

Figure 2b. Each task is a virtual processor with the same properties as the physical processor

Figure 2b depicts the virtual processor properties model. It should be noted that while in the virtual processor state, the task consumes no resources of the physical processor

Copyright © 2006-2009. All Rights Reserved

except for the memory it occupies.

Given these models, the concept of multitasking is simply a procedure that constantly makes judgments as to which virtual processor (Task) needs to get control of the physical processor at any given time. When it is necessary to stop the current task in control of the physical processor and give it to a new task, the kernel swaps the properties of the physical processor with those of the selected virtual processor (Task). This procedure is the famous *context switch*. Figure 2c. depicts the physical processor being shared by a set of virtual processors and governed by a Scheduler.

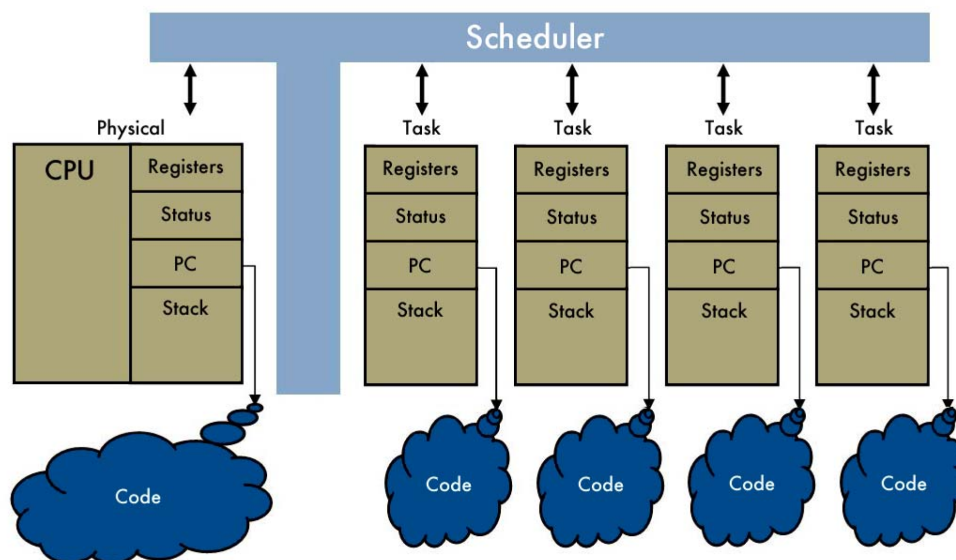

Figure 2c. Multitasking is best thought of as the physical processor being shared by any number of virtual processors, governed by a Scheduler.

7 Priority and Preemption

To achieve efficient computer usage, a multi-tasking real-time executive uses an orderly transfer of control from one code entity to another. To achieve orderly transfers, the executive must monitor system resources and the execution state of each code entity and it must ensure that each entity receives control of the CPU in a timely manner.

The key word here is *timely.* A real-time system that does not perform a required operation at the correct time has failed. That failure can have consequences that range from benign to catastrophic. Response time for a request for executive services and the execution time of these services must be fast and predictable. With such an executive, application program code can be designed to ensure that all needs are detected and processed.

Real-time applications usually consist of several processes (tasks and threads) that require control of system resources at varying times due to external or internal events. Each such code entity must compete with all other code entities for control of system resources such as memory, execution time, or peripheral devices. Program code can be compute-bound (heavily dependent on CPU resources) or I/O-bound (heavily dependent on access to external devices).

Program code that is I/O bound or compute bound cannot be allowed to monopolize a system resource if a more important task requires the same resource. There must be a way of interrupting the operation of the lesser entity and granting the resource to the more important one.

One method for achieving timeliness is to assign a priority to each task (or code entity). The executive uses this priority to determine a task's place within the sequence of execution of other code entities. For example, tasks of low priority may have their execution preempted by a task of higher priority to permit a high priority task to perform a time-critical function.

How the kernel determines which virtual processor needs to get control of the physical processor is the province of the system Scheduler.

8 Scheduling Models

Round-Robin Scheduling

Tasks at the same priority level execute in a round-robin manner. Round-Robin scheduling activates tasks in some order within the current priority level at each point of scheduling.

Tick-Sliced Scheduling

Tick-sliced scheduling is a variant of round-robin scheduling. Both methods are similar in every respect except that a tick-sliced task can only run for a predefined number of ticks (a tick quantum) from an associated counter. The tick may represent time or some other unit particular to the application. A task remains in control of the CPU until the tick quantum expires or until the task yields control or blocks. If the tick quantum expires, the Scheduler forces the task to yield if there is another task at the same priority is waiting to run.

Preemptive Scheduling

Preemptive scheduling is the policy that leads to the concept of preemptive scheduling in which the Scheduler gives control of the physical processor to the task that has the highest priority and is also ready to accept control of the physical processor.

Copyright © 2006. All Rights Reserved

As seen in Figure 3, the task at the lower priority (Task 2) is preempted at the occurrence of an event that activates or releases Task 1.

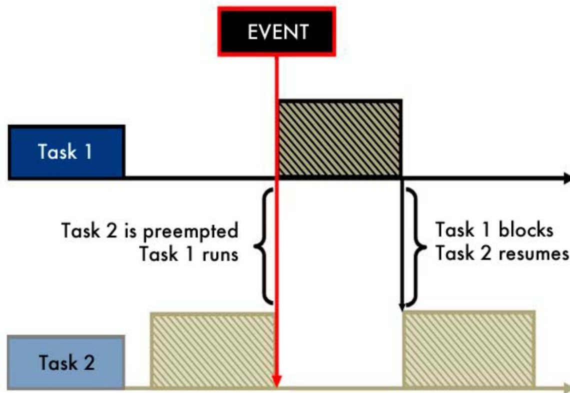

Figure 3. In a prioritized, preemptive system the task with the highest priority will preempt a task of a lower priority. Often, a high priority task is associated with an external event.

9 Kernel Classes

An RTOS operates on a set of structures commonly called classes. Each class supports a set of operators commonly called kernel services that are invoked by application processes to achieve an expected behavior. These classes include the following:

Tasks manage execution of program code; each task is independent of other tasks but can establish relationships with other tasks in many forms, including data structures, input, output, or other constructs.

Intertask communications are mechanisms to pass information from one task to another following. Commonly used classes for intertask communications include *Semaphores*, *Messages* and *Mailboxes*, *Queues, Pipes* and *Event Flags*.

Semaphores provide a means of synchronizing tasks with events.

Mutexes permit a task to gain exclusive access to an entity or resource.

Timers and Alarms count ticks and generate signals.

Memory Partitions manage RAM to prevent fragmentation.

Queues permit passing of fixed amounts of data from a producer to a consumer in FIFO order.

Messages and Mailboxes are useful in managing variable size data transmissions from a sender to a receiver.

Kernel Services are routines that are executed to achieve certain behavior of the system. When an application code entity requires a function provided by the kernel, it initiates a kernel service request for that function.

ISR (Interrupt Service Routine) is a software routine that is activated to respond to an interrupt. (See interrupt handling below.)

10 Interrupt Handling

Interrupts are generally external events from other elements of the system that demand immediate attention. When an interrupt occurs the processor finishes the current instruction and then enters an ISR, where

- the address where the interrupted process is to continue following treatment of the interrupt is saved, along with the state of the processor (registers, etc.);
- the processor begins executing a routine to service the device that caused the interrupt, and
- upon completion of the device handling, the ISR restores the interrupted state of the processor and returns to the code at the saved continuation address.

11 Responsiveness and Latency

Prioritization is not a guarantee that a task will execute on time. Other factors must be considered, including the time the RTOS needs to release the processor and schedule the next task. This is often referred to as latency. Other scheduling variables must also be taken into account. Here is a concise list:

Arrival Time (also called *Release Time*) is the time at which the task becomes ready to run but not necessarily in control of the processor (i.e., executing)

Start Time is the time at which the task gains control of the processor and starts its execution.

Finish Time is the time at which the task completes its execution.

Computation Time is the time the processor needs to execute the task without interruption. The maximum value for a given task is referred to as its *Worst Case Execution Time* (WCET)

Deadline is the time before which the task should complete its execution in order to be correct

Period is the minimum time period between successive points of release.

Lateness is the amount of time a task exceeds its deadline. If the lateness value is <0, the task has completed before its deadline.

12 More Information

For more information on the RTXC Quadros RTOS, supported processors and other QSI products, including FAT and flash file systems, TCP/IP stack, UPnP, USB, SDIO, CAN, GUI tools and more, please contact Quadros Systems at sales@quadros.com or call your local Quadros sales representative.

Copyright © 2006-2009. All Rights Reserved