Introduction to Real-Time Operating Systems

According to the 2012 UBM Embedded Market study, 70% of embedded systems have a real-time requirement.

The purpose of this paper is to provide a top-level overview of RTOS systems and concepts. If you need information on how these concepts are implemented in a specific real-time operating system, detailed information on the RTXC Quadros RTOS is available here:

www.quadros.com/rtos

Embedded Systems are generally understood to mean devices in which intelligence, and often communications capability, is 'embedded' inside the product. These are single purpose, pre-programmed systems as opposed to personal computers which are multi-purpose computing platforms.

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 to manage time-critical allocation of a computer system's resources. RTOSes are commonly used in embedded systems.

The RTOS provides a software abstraction layer to the underlying microcontroller or processor (CPU). The developer's application code uses the inherent capabilities of the RTOS to manage access to and scheduling of CPU resources. In short, the RTOS

- Determines which execution entities in the application should control the CPU, in what order, and how much time is allowed for each before giving up processor control.
- 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

Reasons to Use an RTOS

A well-designed RTOS provides a number of tangible benefits.

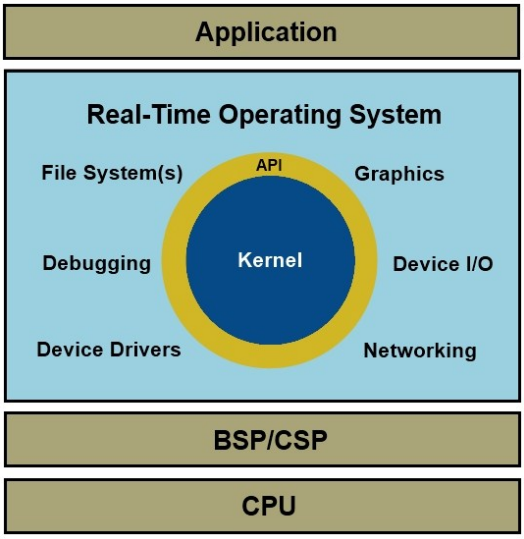
- Provides a solid foundation for your project with rules and policies to ensure consistency and repeatability
- Simplifies development and improves productivity
  - A rich set of kernel services (API calls) that save you from writing extensive code and using too much system overhead to achieve the same result.
  - High level RTOS objects can easily handle complex functions. Significant time savings from having to do everything yourself.
- Presents an abstraction to the processor which means you don't have to focus on many of the HW details
- Implements a reliable scheduling system to successfully manage multiple operations sharing the same processor
  - allows you to schedule when those operations run so that they do not conflict with each other
  - allows you to manage access to various processor or peripheral resources such that operations can run successfully without interference.
- Efficiently handles housekeeping functions – saving and restoring register sets, managing memory buffers
- Integrates and manages resources needed by communications stacks and middleware (TCP/IP, USB, 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.

Components of an RTOS

Scheduler

The scheduler, the central element in an RTOS, determines which application code entities get access to the CPU and in what order. In most commercial RTOSes there are three scheduling models: preemptive, cooperative (also called round robin), and time-sliced.



Function Library

The function library of the RTOS serves as the interface between the application code and the RTOS. Often known as Application Program Interfaces (APIs), these functions encapsulate the operational requirements of the RTOS into its various services. Application code entities make requests to the kernel through these APIs in order to cause the desired programmatic behavior of the application.

Classes and User-defined Data Objects

RTOSes use data structures generally organized into groups or classes by operation type. The user defines the set of objects in each class that the RTOS will use in controlling the application. The names may be somewhat different, depending on the RTOS

Hard vs. Soft real-time

In hard real-time, certain operational time constraints (deadlines) must be met to avoid a catastrophic failure. If all elements meet their deadlines, the system is predictable and considered schedulable. Such a system is also referred to as deterministic.

With soft real-time, timing constraints are less restrictive. Even a failure of an element to meet its deadline still provides some value to the application. 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.

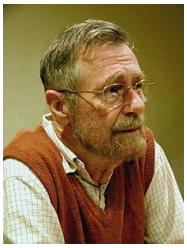


Photo by: Hamilton Richards

Most modern real-time kernels make use of the work done in the mid-1960s by Dr. Edsger Dijkstra and the concept of a 'society of sequential processes.' Although this concept of '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.

RTOS Properties and Functions

Primary Requirements

- Manage the processor and other system resources to meet the requirements of the application
- Respond to, and synchronize with, events
- Move data efficiently between processes
- 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 period of time

While the capabilities above are primary requirements, there are secondary requirements including the ability to provide:

- Efficient management of RAM
- Exclusive access to resources

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 RTOS.

RTOS services must be designed and coded to require minimal execution time yet remain predictable. Execution speed of the RTOS 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.

Multitasking

Without a CPU dedicated to each code entity it is impossible to implement simultaneous operations. However, CPU access time can be shared and achieve the appearance of simultaneous operation. By decomposing the functions of the system into different program code elements (tasks or threads) and rapidly switching the CPU between them, the designer can achieve the effect of concurrency. The orderly switching between many tasks is the basis of the concept of multitasking.

Figure 2 shows how multitasking is implemented. The left side 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

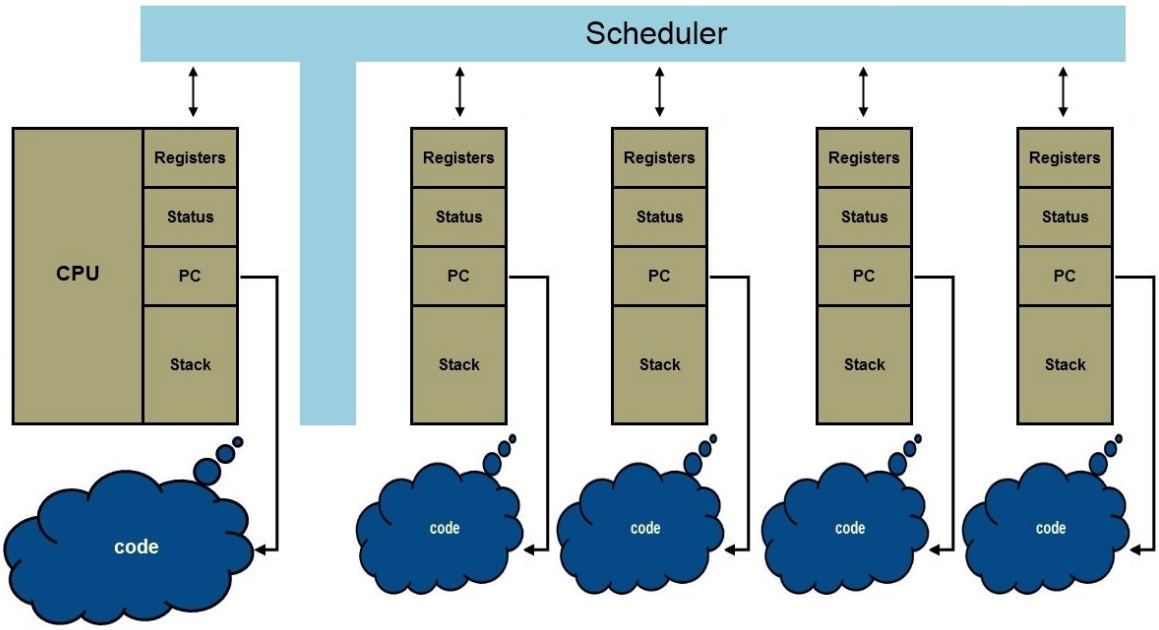


Figure 2
Multi-tasking is best thought of as the physical processor being shared by any number of virtual processors, governed by a scheduler

Priority Inversion

A fundamental maxim in a real-time system is that the highest priority task that is ready to run must be given control of the processor. If this does not happen deadlines can be missed and the system can become unstable or worse.

Priority inversion simply means that the priority you established for a task in your system is turned upside down by the unintended result of how priorities are managed and how system resources are assigned. It is an artifact of the resource sharing systems that are fundamental to RTOS operation.

So how do you avoid these situations or once in them, how does the system get out of the situation?

*We have an extensive blog on the topic: **RTOS Explained**. Here you will find more detailed explanations of RTOS principles including priority inversion.*

have the same properties as the physical processor: 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 also have its set of code that it is executing.

The right side of Figure 2 depicts several virtual processors (tasks). Each task awaits its opportunity to have its properties switched into the physical processor. Until a task gains control of the CPU it consumes no resources of the physical processor except for the memory it occupies.

The scheduler constantly makes judgments about which task needs to control the CPU at any given time. When it is necessary to stop the currently running task and give CPU control to a new task, the scheduler swaps the properties (or operating context) of the running task with those of the new task. This procedure is called a 'context switch.'

Priority and Preemption

To achieve the goal of efficient, shared CPU usage, a multi-tasking real-time operating system uses an orderly transfer of control from one task to another. To make this possible the scheduler must monitor system resources and the execution state of each task (running, ready, or blocked) to 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. This means that response time for a request for kernel services and the execution time of these services must be fast and predictable. The inherent predictability of the RTOS allows application program code to 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. It is generally considered to be bad design to allow a single task or thread to monopolize a system resource if a more important task requires the same resource. There must be a method of interrupting the operation of the lesser entity and granting the resource to the more important one.

Scheduling Models

There are several common scheduling models for allowing tasks to receive processor time. The utility of each of these depends on the requirements of the application.

Round-Robin (Cooperative) Scheduling

All tasks are of equal importance and are at the same priority. Task execute sequentially. Once a task receives processor control it will run until it completes or reaches a point of re-scheduling. It then yields CPU control to allow the Scheduler to determine the next task to get CPU control. During its execution cycle, a round-robin-scheduled task cannot block, wait, pause, or do anything that would allow another task to gain CPU control, however it can be prempted by an Interrupt (Exception). In some cases it can yield.

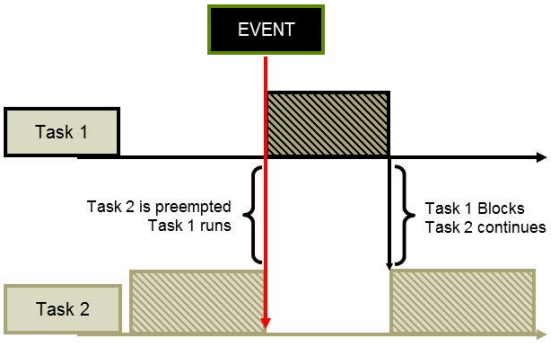


Figure 3
Task preemption

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.

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.

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 ap-plication 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.

RTXC RTOS

The RTXC Quadros family of RTOSes was developed to give developers the flexibility and scalability they need to efficiently program with any or all of the major processing models: high speed dataflow processing, control processing, convergent processing, and multiprocessing.

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 enti-ty 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 hand-ling below.)

Interrupt Handling

Interrupts are generally external events from other ele-ments of the system that demand immediate attention. When an interrupt occurs the processor finishes the cur-rent 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.

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 considered. These include:

References

Dijkstra, Edsger W. The structure of the 'THE'-multiprogramming system (EWD-196). E.W. Dijkstra Archive. Center for American History, http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD196.html

Barrett, Andrew T. "On Schedule", Micro Technology Europe, August 2012