Real Time Operating System

Pornima Kakade

Department of MCA

G.H Raisoni College of Engineering

(An Autonomous Institute under UGC Act 1956)

*Hingna Road, Digdoh Hills, Nagpur-16*

kakade\_pornima.ghrcemca@raisoni.net

***Abstract–*** A real-time operating system (RTOS) supports real-time applications and embedded

systems. Real-time applications have the requirement to meet task deadlines in addition to the logical correctness of the results.

In this paper, will explain some Real Time Operating Systems (RTOS) basics. And briefly cover the RTOS most important features of a representative RTOS, and advantages of RTOS .

***Keywords–*** RTOS, Real time operating system, RTOS component, advantages of RTOS

**I. INTRODUCTION**

A real-time system is one whose correctness involves both the logical correctness of the

outputs and their timeliness [11]. A real-time system must satisfy bounded response-time

constraints; otherwise risk severe consequences, including failure. Real-time systems are

classified as hard, firm or soft systems. In hard real-time systems, failure to meet response-time

constraints leads to system failure. Firm real-time systems are those systems with hard deadlines,

but where a certain low probability of missing a deadline can be tolerated. Systems in which

performance is degraded but not destroyed by failure to meet response-time constraints are called

soft real-time systems. A real-time system is called an embedded system when the software

system is encapsulated by the hardware it controls. The microprocessor system used to control the

fuel/air mixture in the carburetor of many automobiles is an example of a real-time embedded

system. An RTOS differs from common OS, in that the user when using the former has the ability

to directly access the microprocessor and peripherals. Such an ability of the RTOS helps to meet

deadlines.

**How does an RTOS work?**

The core of an RTOS is known as the **kernel**. An API is provided to allow access to the kernel for the creation of threads, among other things. A **thread** is like a function that has its own stack, and a Thread Control Block (**TCB**). In addition to the stack, which is private to a thread, each thread control block holds information about the state of that thread.

The kernel also contains a **scheduler**. The scheduler is responsible for executing threads in accordance with a scheduling mechanism. The main difference among schedulers is how they distribute execution time among the various threads they are managing. Priority-based, preemptive scheduling is the most popular and prevalent thread scheduling algorithm for embedded RTOSes. Typically, threads of the same priority execute in a round-robin fashion.

Most kernels also utilize a system tick interrupt, with a typical frequency of 10ms. Without a system tick in the RTOS, basic scheduling is still available, but time-related services are not. Such time-related services include: software timers, thread sleep API calls, thread time-slicing, and timeouts for API calls.

The system tick interrupt can be implemented with one of the hardware timers in the embedded chip. Most RTOS have the ability or extension to reprogram the timer interrupt frequency dynamically such that the system can sleep until the next timer expiration or external event. For example, if you have an energy sensitive application you might not want to run the system tick handler every 10ms if not necessary. Suppose for example the application is idle and the next timer expiration is 1000ms away. In this case, the timer can be reprogrammed to 1000ms and the application can enter low-power mode. Once in this mode, the processor will sleep until either another external event occurs or the 1000ms timer expires. In either case, when the processor resumes execution the RTOS adjust the internal time according to how much time has elapsed and normal RTOS and application processing is resumed. This way, the processor only executes when the application has something to do. During idle periods the processor can sleep and save power.

Applications can be divided into different threads with or without the use of an RTOS. For example, one thread can be responsible for reading the keyboard, another for checking temperature, a third for printing messages on a LCD screen, and so on. With an RTOS you not only get the tool to create threads, but also tools to communicate between the threads, and tools to ensure that threads that are time critical are executed within their real-time constraints. Since the interface between the different threads becomes very clean when using an RTOS you will save time both in development and in maintenance of the application

The main point is that, if programmed correctly, an RTOS can guarantee that a program will run with very consistent timing. Real-time operating systems do this by providing programmers with a high degree of control over how tasks are prioritized, and typically also allow checking to make sure that important deadlines are met.

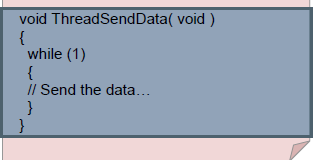
**II.RTOS COMPONANTS :**

Let’s see what features an RTOS has to offer and how each of these features might come in handy in different applications.

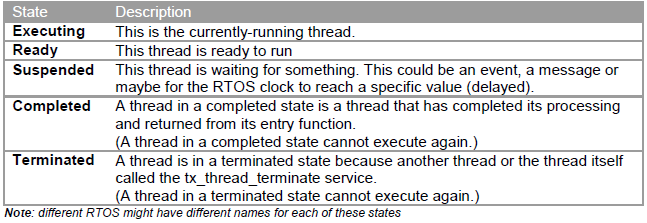
**1. Threads**

Threads are like functions, each with its own stack and thread control block (TCB). Unlike most functions, however, a thread is almost always an infinite loop. That is, once it has been created, it will never exit.

A thread is always in one of several **states**. A thread can be ready to be executed, that is, in the **READY** state. Or the thread may be suspended (pending), that is, the thread is waiting for something to happen before it goes into the READY state. This is called the **WAITING** state.



Here is a short description of the states we will use in ThreadX.



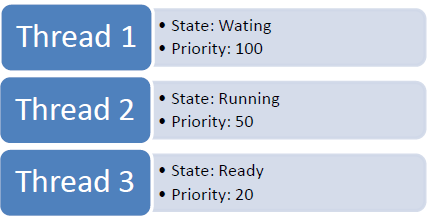
**2. Scheduler**

There are two major types of scheduler from which you can choose:

**2.1. Event-driven(Priority-Controlled Scheduling Algorithm)**

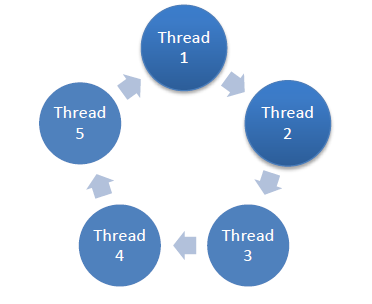
Usually different threads have differing response requirement. For example, in an application that controls a motor, a keyboard and a display, the motor usually requires faster reaction time than the keyboard and display. This makes an event-driven scheduler a must.

In event-driven systems, every thread is assigned a priority and the thread with the highest priority is executed. The order of execution depends on this priority. The rule is very simple: **The scheduler activates the thread that has the highest priority of all threads that are ready to run.**

****

**2.2 Time-sharing**

The most common time-sharing algorithm is called **Round-Robin**. With round-robin scheduling, the scheduler has a list of the threads that make up the system and it uses this list to check for the next thread that is ready to execute. If a thread is READY, that thread will execute. Associated with each thread is its ‘time-slice’. This time-slice is the maximum time a thread can execute for each round the scheduler makes .



**3. Assigning priorities**

It is very important to assign correct priorities to your different threads. A lot of papers have been written about how this can be done in order to get the most out of your RTOS based application. We will not dive deep into this subject but just mention a couple of rules that are helpful.

1. Use as few priority levels as possible. Only assign different priorities when preemption is absolutely necessary. This will reduce the amount of context switches done in the system. And the fewer context switches done, the more time can be spent on executing application code.

2. Make sure that all critical timing constraints are met in your application. Depending on what type of application you have, this can be a tough one. One way of solving this is to use RMA (Rate Monotonic Algorithm). The ThreadX RTOS also provides a unique technology called preemption-threshold. This can be used to reduce context switches as well as help guarantee the execution of application threads.

**4.Thread communications**

We also need to be able to communicate between the different threads in an RTOS. Communication can take the form of an event, a semaphore (flag), or it can be in the form of a message sent to another thread.

The most basic communication is via an **event**. An **event** is a way for a single thread to communicate with another thread. An interrupt service routine (ISR) can also send an event to a thread. Some RTOS can also send one (1) event to more than one thread.

**Semaphores** are usually used to protect shared resources, for example if there is more than one thread that needs to read/write to the same memory (variable). This is done so that a variable does not change through the actions of another thread while it is being addressed by the active thread. The principle is that you need to obtain a semaphore associated with a variable protected in this way before reading/writing to this memory. Once you have obtained the semaphore, no one else can read/write to this memory until you release the semaphore. This way you can ensure

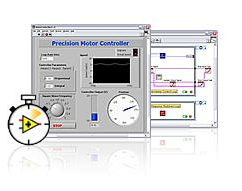
that only one thread at the time reads/writes to a memory location or variable.

Messages allow you to send data to one or more threads. These messages can be of almost any size and are usually implemented as a mailbox or a queue. The behavior for mailboxes and message queues varies from different RTOS venders.

**III. Advantages Of**  **RTOS:**

Developing a real-time system with reliable performance and precise timing typically means integrating your application code, a real-time operating system, and underlying computer hardware – tasks that can add time and cost to your development cycle. With National Instruments real-time hardware and software, you can create real-time programs, prototype them on real world hardware, and deploy them in a fraction of the time it takes with traditional tools. Below are just some of the reasons why engineers and scientists are choosing National Instruments real-time products for their projects

**1. Accelerate Development with Graphical Programming**



[NI LabVIEW](http://www.ni.com/labview/whatis/) is a graphical programming language that you can use to create applications quickly and intuitively. When you combine it with the LabVIEW Real-Time Module, LabVIEW software helps you create hard real-time systems with precise timing, deterministic I/O, and reliable communication much faster than with traditional development tools. LabVIEW also has [hundreds of built-in signal processing, analysis, and control functions](http://zone.ni.com/devzone/cda/tut/p/id/3582) for you to reuse.

**2. Seamlessly Deploy to a Wide Variety of Targets**



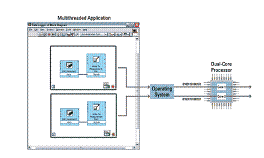
At any time during the development process, simply click on the LabVIEW run arrow to automatically download and test your application on any NI real-time hardware target. You can choose from rugged automation controllers, high-performance computing systems, single-board computers, or even generic PCs. All NI real-time targets feature a fully integrated software stack that includes the real-time operating system, I/O drivers, and network communication.

**3**. **Connect with Real-World I/O in Minute:**



Choose from dozens of I/O modules, including A/D and high-speed digital, to customize your real-time measurement or control system. Need to connect with remote devices on a bus? NI hardware can connect to virtually any industry-standard device from CAN to EtherCAT

**4. Take Advantage of Multicore Technology**



LabVIEW automatically analyzes your real-time program and runs parallel sections of code on different processor cores. In addition, you can optimize your processing by targeting important routines to specific processors, and perform low-level debugging using the NI Real-Time Execution Trace Toolkit.

**IV. CONCLUSIONS**:

Building a real-time system can be a good idea if you need to make sure that certain parts of your program run in a certain amount of time, or if you need to run your program reliably for long periods of time. If you are working on a mission-critical or safety-related project, then the need for building a real-time system is clear.

Even in the case where precise timing and long term reliability are not absolute requirements for your project, building a real-time system can provide added peace of mind that your program will continue to run without interrupting your measurement or control process. If the system that you are creating could result in maintenance costs in the event it is interrupted, the hardware and software costs required to create a real-time system may be well worth the investment.

Note that a real-time system does not necessarily make sense for every measurement or control project. Real-time operating systems typically only run one program at a time, and most real-time systems do not feature a user interface; in this case a separate computer must be used to provide graphics or user controls. Still, thousands of real-time systems are in operation today and will continue to be a good solution for projects that need precise timing and high reliability.

**REFERENCES**

[1] The Open Group, http://www.opengroup.org.

[2] Basic Concepts for Real Time Operating Systems **By Mats Pettersson, IAR Systems**

[3]wikipedia.org/wiki/**Realtime** **operating**\_**system**

[4] www.arm.com/community/.../rtos-**real**-**time**-**operating**-**system**.php

[5] www.ask.com/**Study**+**Materials**

[6] M.Stokely and N.Clayton, *FreeBSD Handbook*, Second edition, Wind River Systems,

2001.

[7] **Real Time** **Operating Systems**. Tutorial at SBCCI 2001. **Prof. Dr. Franz** J. Rammig.

[8]www.ece.stevenstech.edu/~ymeng/courses/CPE555/papers/rtos\_paper.**pdf** A Survey of **Real**-**time Operating Systems**.

[9]ModelBased Design **...** roducts/development\_tools/ide/ tornado 2/tornado\_2. \_ds.**pdf** **...** A **real**-**time operating system**