An Overview of Real-Time Operating Systems

Walter Cedeño,[1,2]* *and Phillip A. Laplante*[1,2]
[1]*Johnson & Johnson Pharmaceuticals Research & Development, Exton, PA*
[2]*Penn State University, Great Valley School of Graduate Professional Studies, Malvern, PA*

Keywords:
real-time systems, laboratory automation, operating systems, smart devices

Over the past 25 years, advances in semiconductor manufacturing have led to smaller and faster computers, which in turn has stimulated the development of "smarter" laboratory devices that can control complex networks of devices and process large amounts of data fast and reliably. As more functionality is pushed down to laboratory devices and personal computers, the sophistication needed to manage external resources, events, and information grows. In some cases, only real-time operating systems (RTOSs) can meet the time and resource constraints of such systems. Whether you write your own software for lab automation, write middleware to help communicate between programs, or use off-the-shelf software, it is beneficial to know when a RTOS is an appropriate platform for your application. This paper provides an overview of RTOSs, the criteria needed for their evaluation, and examples of typical RTOSs. Our main purpose is to enable the reader to understand basic concepts of real-time systems and to stimulate further investigation into their unique properties in the context of laboratory automation. (JALA 2007;12:40–5)

*Correspondence: Walter Cedeño, Ph.D., Johnson & Johnson Pharmaceuticals Research & Development, 665 Stockton Drive, Exton, PA 19335, USA; Phone: +1.610.458.5264, ext. 6577; Fax: +1.610.458.8249; E-mail: wcedeno@prdus.jnj.com

1535-5535/$32.00

doi:10.1016/j.jala.2006.10.016

Why real-time operating systems for laboratory automation?

Real-time operating systems (RTOSs) are often used to develop applications for systems with complex time and resource constraints. This situation often typifies laboratory automation where one or more computers must synchronize the activities among one or more instruments involving time, process, or precedent constraints. Time constraints might include actions such as "mix for at least *x* seconds" or "heat at 100 °C for 1 min". Process constraints condition activities, for example, "pick x and place at y" or "rotate 30°". In addition, precedent constraints such as "before", "during", "after", and their complements add further complexity to process control. Fortunately, RTOSs provide the necessary features to handle the demanding time, process, and precedent constraints often associated with such systems.

RTOSs are deterministic by design, which allows them to meet deadlines associated with external events using a limited set of resources. Advances in modern development tools and frameworks have made RTOSs more accessible to developers of all levels. Developing applications for RTOSs used to be a job for the most skillful developers not only due to the complexity of the application, but also due to the need to use in-circuit emulators or very sophisticated cross-development platforms. Advances in tools, languages, and frameworks, however, have made the development of applications for real-time systems easier. In the next sections, we examine certain important aspects of RTOSs. Our intent is to provide information to allow you to understand the concepts in this area, to help

you determine when a RTOS will benefit an application, and to stimulate your interest to learn more.

This paper is organized as follows. The next section, Characteristics of RTOSs, provides an overview of RTOS concepts and important characteristics. The section Evaluating an RTOS provides information that can assist in the evaluation of RTOSs. The section A Survey of RTOSs provides a summary of available RTOSs and describes some of the features for RT-Linux and Windows CE. The last section, Conclusion, provides advice on next steps.

Characteristics of RTOSs

There are many characterizations for "real-time system" and the term is often used ambiguously because real-time systems have such differing time constraints. For example, some real-time systems only need the application to satisfy average time deadlines, with small variability, when processing external events. This might be the case when controlling mixing or heating processes or in the case of display processes. Critical real-time systems, on the other hand, have very strict time deadlines that must be met every time, for example, if a controlled dose of radiation must be delivered to some sample. The application for a critical real-time system must have sufficient time to process an external stimulus, called the response time, within a predetermined value under all possible circumstances. Laplante's definition of a real-time system captures the salient points. "A real-time system is one whose correctness involves both the logical correctness of the outputs and their timeliness."[1]

Let's consider an application to control a simple process involving the mixing, heating, and then mixing again of a sample in a food production laboratory. Probably for most food ingredients, we do not need to precisely time the duration of the mixing and heating, but it is very important that the order of the processes be correct. On the other hand, if our automation equipment is controlling a high-throughput screening process where samples degrade after a predetermined amount of time, we probably need a much higher level of precision.

What distinguish real-time systems are their time constraints. Real-time systems are classified as hard, firm, or soft systems. In hard real-time systems, the most critical type, failure to meet its time constraints will lead to system failure (imagine a nuclear reactor control system). In firm real-time systems, the time constraints must be met in most cases, but can tolerate missing a low number of deadlines (consider a food processing plant control system). In soft real-time systems, the performance is degraded when time constraints are not met but the system will not catastrophically fail (this is often the case with real-time animation of visual displays). Usually an "ordinary" operating system (OS), with some real-time features, is suitable for firm and soft real-time applications but RTOSs are necessary for hard real-time systems. The choice of OS will play an important role in the application design and how any hard time constraints will be met.

An OS is the software that manages the hardware resources of a computer and provides an abstraction layer between the hardware and the applications that will be running in the system. The OS can be considered a resource manager as well because it manages access to all devices in the system. Last, but not least, the OS is a policy enforcer. That is, the OS defines the rules of engagement between the applications and resources. An OS is composed of multiple software subsystems, and the core components in the OS form its kernel. As depicted in Figure 1, the OS provides several subsystems to manage the central processing unit (CPU), main memory, and external devices. The OS also provides an application programming interface (API), which defines the rules and interfaces that enable applications to OS features and communicate with the hardware and other software applications. Most modern OSs support the creation of multiple threads[a] in a process.[b] This allows for development of complex applications that support concurrency and can handle multiple asynchronous events from external sources. A complete review of the design of modern OSs can be found in Refs. 2–4.

An RTOS is an OS that supports applications that must meet time constraints while providing logically correct results (e.g., so that mixing occurs before and after heating). RTOSs also provide the necessary features to support real-time applications. They provide a deterministic environment where we can calculate, a priori, the response time[c] for the worst-case scenario (so that we know, e.g., that the sample will eventually be screened). The IEEE Portable Operating System Interface for Computer Environments, POSIX 1003.1b,[5] provides a list of basic services an RTOS must support. We list many of these and discuss some in the context of laboratory automation.

- *Asynchronous Input/Output (I/O)*: The ability to overlap application processing and application initiated I/O operations. Improves application performance and increases CPU utilization. This might be the case, for example, when the RTOS initiates a request to a device to start mixing for 5 s, and then moves on to making a request to another device to start heating to 100 °C, without having to wait for the first request to be acknowledged or completed.
- *Synchronous I/O*: The ability to assure return of the interface procedure when the I/O operation is completed. Allows a thread to block and wait for the processing of an I/O operation. Synchronous I/O might be desirable when control of the device can only be granted to one processor

[a]A thread is generally considered the execution unit of an OS. Sometimes called a task, it contains a set of instructions executed by the CPU and other resources.

[b]A process provides the environment where one or more execution units (depends on OS) execute concurrently. Information in the process is shared by all of its execution units.

[c]The time it takes for the processing of a particular event.

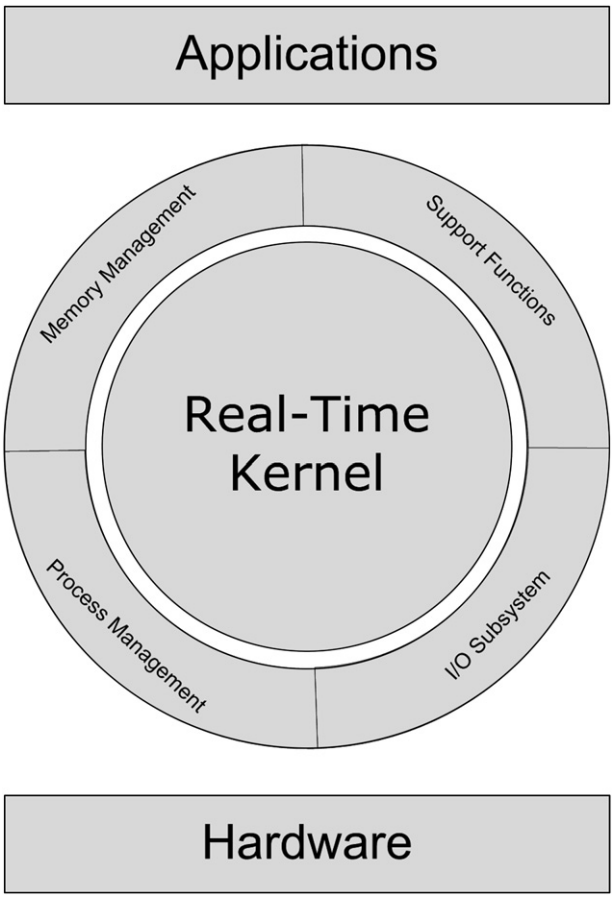

Figure 1. OSs provide a layer of abstraction between the hardware and other applications.

at a time, for example, if a robot arm will be shared by more than one thread.

- *Memory locking*: The ability to guarantee memory residence by storing sections of a process that were not recently referenced on secondary memory devices. Allows fine grain of control of which part of the application must stay in physical memory to reduce the overhead associated with transferring memory to disk. Memory locking might be used to keep in memory a thread that monitors a critical process that requires immediate attention.
- *Semaphores*: An OS primitive that provides the ability to synchronize resource access by multiple processes or threads. Synchronization mechanisms are very important in RTOSs to ensure that two threads do not try to use the same resource simultaneously.
- *Shared memory*: The ability to map common physical space into independent process-specific virtual space. Commonly used to share information between different processes or threads.
- *Execution scheduling*: Ability to schedule multiple threads. Common scheduling methods include round-robin[d] and priority-based preemptive[e] scheduling. Round-robin scheduling might be used when there is a set of low criticality tasks that occur in a regular order (e.g., fill, mix, heat, mix, pour). Preemptive priority scheduling is used in mission critical systems where one or more events will have a high level of urgency (e.g., the temperature of a process has exceeded safe limits).
- *Timers*: Timers improve the functionality and determinism of the system. A system should have at least one clock device (system clock) to provide good real-time services. Timers allow applications to set up events at predefined intervals or time.
- *Interprocess communication (IPC)*: IPC is a mechanism in which threads share information needed for a particular application. Common RTOS communication methods include mailboxes, shared memory, and queues.
- *Real-time files*: The ability to create and access files with deterministic performance.
- *Real-time threads*: Real-time threads are schedulable entities of a real-time application that have individual timeliness constraints and may have collective timeliness constraints when belonging to a runnable set of threads.

In addition to the POSIX, other basic requirements described on a recent survey of RTOS[6] are the following:

- *Low overhead*: The context-switch[f] times for threads and the OS overhead should be minimal.
- *Preemptive*: The RTOS must be able to preempt the currently executing thread to give the CPU to a higher-priority thread. Again, this feature is needed to allow an urgent event (e.g., safe temperature exceeded) to preempt an activity of lower criticality.
- *Deterministic synchronization*: Ability for multiple threads to communicate among themselves within a predictable time.
- *Priority levels*: The RTOS must provide sufficient priority levels to allow for effective application implementation. This feature is important in enabling flexible preemptive priority scheduling.
- *Predefined latencies*: The timing of API calls must provide expected latencies. This feature is helpful when process steps, such as adding, mixing, or heating, need to be commenced within a certain amount of time of a preceding process step.

The features of an RTOS are necessary, but not sufficient, to implement a real-time system. Whether or not an RTOS provides the necessary features for your system is worthless if the underlying hardware does not provide the necessary horsepower. The CPU speed, the memory access speed, and the device access speed define the potential speed of the underlying

[d]In round-robin scheduling, threads with equal priority are assigned the CPU for a fixed period of time in order.

[e]A preemptive scheduler is one that can take away the CPU from the currently executing thread to allow a higher-priority thread to run.

[f]A context switch is the process of storing and restoring the state of the CPU to allow multiple threads to share a single CPU.

hardware. However, industrial and laboratory automation software can present a substantial processing load. Therefore, you need to ensure that the hardware is capable of supporting the hard real-time constraints and worst-case scenario in your fully loaded system regardless of the underlying RTOS. Once the hardware is proven to be suitable for your real-time application, then you can evaluate the features of RTOSs.

Evaluating an RTOS

Whether you write your own software for lab automation, write middleware to help communicate between programs, or use off-the-shelf software, it is beneficial to know how to select the right RTOS for your environment. Of course, the first question you should always consider is if an off-the-shelf RTOS will be compatible with the automation software that you need to host. Many commercial RTOSs and automation software developers have such compatibility lists.

A real-time application for an embedded system[g] will often require an RTOS with a very small footprint[h] and little overhead. A real-time application running on a PC or Mac can support more complex RTOSs that provide additional features and sophisticated user interfaces. Both hardware and software choices should be considered together when designing a real-time system. The system cost, the development time, and the chance of success will depend on the choices you made. One of the most important criteria, often overlooked, when evaluating a RTOS is the availability of development tools such as real-time symbolic debuggers and appropriate programming language support. The use of sophisticated IDE is common today. IDEs provide developers with tools and examples to quickly develop your software, test it, and manage different revisions. Once developers become familiar with the IDE, their productivity increases and the ability to work in a team environment improves.

Does the development host have to be the same as the target host? Not always. Some development environments allow the user to develop on one platform and deploy to another. Obviously, developing in the same platform helps catch problems faster. On the other hand, developing on a bigger platform will allow the use of sophisticated tools that will increase productivity.

If you are going to write some of your own code for any reason, a question arises. Which programming language should you use for real-time systems? The selection of programming language depends in part on the skills of the developers involved and whether the language is supported by the RTOS' development environment. Training will be necessary when using a new language, but this additional burden may be worth it if the language is part of a framework that simplifies development and provides features often necessary for real-time applications.

There are many nontechnical criteria that must be considered as well. Your decision will dictate the cost of ownership for your company. For example, you may choose a commercial RTOS or one from the open source community. In certain extreme cases (such as for low-production volume high-reliability systems), you may even need to develop an RTOS from scratch. For any of these cases, you must evaluate what is going to be the total cost to the company. Commercial RTOS products may have a high up-front cost and may charge royalty fees for every deployment of your system. You may also consider obtaining the source code, which will add to the cost. The open source RTOS might offer free access to source and binaries with no royalties, but often require the purchase of support and long-term commitment from the user community. Open source RTOS may lead to higher long-term costs due to reliance on low-level development tools and highly technical individuals. In summary, the total cost of ownership depends on both up-front and long-term costs of royalties, support, maintainability, training, and consulting services.

There are also many technical decision criteria to be considered, details of which can be found in Refs. 7–10. And a discussion of matching RTOS features to specific project criteria can be found in Ref. 11. In any case, the implementation of the POSIX features provided in Characteristics of RTOSs would be a good place to start, because the first thing you need to do is to choose between a thread- or process-based RTOS. In general, context-switch overhead between threads in the same process is lower than when the threads are in different processes. You need to consider the overhead associated with the different features provided and questions such as "what is the overhead associated with interrupt handlers[i] and associated interrupt processing threads?" Other considerations are the mutual exclusion primitives and data exchange mechanisms used for synchronization between threads and/or processes. Finally, the memory management and scheduler design affect the performance of applications in an RTOS and must also be considered during the evaluation of RTOSs.

A survey of RTOSs

There are over 30 RTOSs,[12,13] which can be classified as open source versus commercial. Here we provide a summary of the features of a few of the top 10 RTOSs listed in a 2005 survey[14] by Embedded System Design.

The VxWorks commercial RTOS from Wind River is the most widely adopted in the embedded industry (e.g., it is used on the International Space Station). VxWorks was first released in the early 1980s and provides a flexible API with more than 1800 methods. The development host can be Red Hat Linux, Solaris, SuSE Linux, Windows 2000 Professional, or

[g]An embedded system is a special-purpose system in which the computer and its application are completely encapsulated by the device it controls.

[h]Memory requirements.

[i]An interrupt handler is the software module that processes software or hardware interrupts. It is activated by the system whenever the interrupt it services occurs. The handler usually processes the interrupt and passes control to the associated interrupt thread to continue processing.

Windows XP. VxWorks is available for all popular CPU platforms: x86, PowerPC, ARM, MIPS, 68K, CPU 32, ColdFire, MCORE, Pentium, i960, SH, SPARC, NEC V8xx, M32 R/D, RAD6000, ST 20, and TriCore. The kernel supports preemptive priority scheduling with 256 priority levels and round-robin scheduling. VxWorks is a multithreading RTOS that provides deterministic context switching and supports semaphores and mutual exclusion with inheritance. This RTOS also provides message queues and Open-standard Transparent IPC for high-speed communications between threads.

The Windows CE RTOS is a commercial RTOS developed in the late 1990s by Microsoft. Windows CE has a small footprint and can run in under a megabyte of memory. There exist three main development platforms (Windows Mobile, SmartPhone, and Portable Media Center) that allow developers to use feature-rich tools to develop applications for x86 and other architectures. Windows CE can have up to 32 processes active with multiple threads in each process. The scheduler supports round-robin or priority-based preemptive scheduling with 256 priority levels, and uses the priority inheritance protocol[j] for dealing with priority inversion.[k] Large parts of Windows CE are available in source form. Windows CE supports OS synchronization primitives such as critical sections, mutexes, semaphores, events, and message queues to allow thread to control access to share resources. A unique feature of Windows CE is the concept of fibers. A fiber is a unit of execution that must be manually scheduled by the application. A fiber is an execution unit that runs within the context of the thread that schedules it. A thread can schedule multiple fibers but they are not preemptively scheduled. The thread schedules a fiber by switching to it from another fiber. The running fiber assumes the identity of the thread. Fibers are useful in situations where the application needs to schedule its own threads.

The most widely adopted free, open source RTOS, eCos (embedded Configurable operating system) was released in 1986. eCos provides a graphical-configuration tool and a command line-configuration tool to customize and adapt the RTOS to meet application-specific requirements. This feature allows the user to set the OS to the desired memory footprint and performance requirements. Development hosts are Windows and Linux and the supported target processors are x86, PowerPC, ARM, MIPS, Altera NIOS II, Calmrisc16/32, Freescale 68k ColdFire, Fujitsu FR-V, Hitachi H8, Hitachi SuperH, Matsushita AM3x, NEC V850, and SPARC. The eCos kernel can be configured with the bitmap scheduler or the multilevel queue (MLQ) scheduler. Both schedulers support priority-based scheduling with up to 32 priority levels. The bitmap scheduler is somewhat more efficient and only allows one thread per priority level. The MLQ scheduler allows multiple threads to run at the same priority. First in, first out (FIFO) or round-robin is used to schedule threads with the same priority. The eCos RTOS supports OS primitives such as mutexes, semaphores, mailboxes, and events for synchronization and communication between threads.

Contemporary OSs such as Linux and Windows XP, called XP Embedded, also have extensions that enable them to support real-time applications. But these OS are only suitable for large real-time systems due to footprint required. On the other hand, there is no need to use specialized tools and there are a large number of developers who can quickly learn how to make use of the real-time features.

Conclusion

For years, the use of RTOSs has been mostly limited to embedded systems. But more recently, applications with real-time requirements are being developed for common platforms. As the need for real-time systems increases, you should understand the benefits and limitations of RTOSs so that you can make the best choice should the need arise. The information provided here will provide background information to help evaluate RTOS' technical features and other nontechnical criteria.

When selecting a real-time system, one of the first things that a user must do is to identify the real-time constraints and categorize them as hard, firm, or soft. If the system does not have any hard time constraints, then a contemporary OS might be sufficient. But for a hard or firm real-time system you will need to consider the trade-offs between the different RTOSs and decide which one fits your needs and your budget. Then, take into account the cost of ownership of the RTOS, the hardware platform, the development tools, and most importantly, if the RTOS can meet the deadline constraints. Not all the criteria used for evaluation should have the same weight, therefore, prioritize the most important features for the system. If you consider all the relevant criteria appropriately, the RTOS selected will be the right one at that moment. This investment can then pay for itself for years to come.

References

1. Laplante, P. A. *Real-Time Systems Design and Analysis*. 3rd edition-Hoboken, NJ: Wiley; 2004; xxi, 505 p.
2. Tanenbaum, A. S.; Woodhull, A. S. *Operating Systems: Design and Implementation*. 3rd editionUpper Saddle River, NJ: Pearson Prentice Hall; 2006; xvii, 1054 p.
3. Stallings, W. *Operating Systems: Internals and Design Principles*. 5th editionUpper Saddle River, NJ: Pearson Prentice Hall; 2005; xiv, 818 p.
4. Deitel, H.; Deitel, P.; Choffnes, D. *Operating Systems*. 3rd editionPrentice Hall; 2004.
5. IEEE. Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language]. 1996, ANSI/IEEE Std 1003.1.

[j]The priority inheritance protocol (PIP) is a method to resolve priority inversion problems. In PIP, a lower-priority thread is temporarily assigned the priority of a higher-priority thread when both threads are using the same OS resource.

[k]Priority inversion occurs when an OS resource held by a lower-priority thread delays the execution of a higher-priority thread because both are contending for the same resource.

6. Baskiyar, S. A survey of contemporary real-time operating systems. *Informatica* **2005,** *29*, 233−240.
7. Anderson, M. E. *Selecting the right RTOS, a comparative study*. COSPA Knowledge Base, 2002.
8. Straumann, T. *Open source real time operating systems overview*. 8th International Conference on Accelerator & Large Experimental Physics Control Systems, San Jose, California, 2001.
9. Stankovic, J. A.; Rajkumar, R. Real-time operating systems. *Real-Time Syst*. **2004,** *28*, 237−253.
10. Lu, S.; Halang, W. A.; Gumzej, R. *Towards platform independent models of real time operating systems*. 2nd IEEE International Conference on Industrial Informatics, 2004.
11. Laplante, P. A. Criteria and an objective approach to selecting commercial real-time operating systems based on published information. *Int. J. Comput. Appl.* **2005,** *27(2)*, 82−96.
12. Mhatre, P. N. Real time operating systems (Rtos), embedded systems pocket Pc, embedded systems development VxWorks, QNX, Windows CE, Palm OS OneSmartClick.Com. http://www.onesmartclick.com/rtos/rtos.html. Accessed November 26, 2006.
13. Real time operating systems Dedicated Systems Encyclopedia. http://www.realtime-info.be/encyc/buyersguide/rtos/rtosmenu.htm.
14. Turley, J. Embedded systems survey: operating systems up for grabs. *Embedded Syst. Design* **2005** CMP. http://www.embedded.com/showArticle.jhtml?articleID=163700590.