April 2012

An Introduction to RTOS

Sanjay Singh

Department of Electronics design, (ED&T), Gorakhpur UP India, dcsanjaypm@gmail.com

Nishant Tripathi

Department of Electronics design, DOEACC Society, Gorakhpur UP India, nishanttri@gmail.com

Anil Kumar Chaudhary

Dept. of Electronics Design & Technology, DOEACC Society, Gorakhpur, UP, India, anil.viet@yahoo.com

Mahesh Kumar Singh

Dept. of Electronics Design & Technology, DOEACC Society, Gorakhpur, UP, India, maheshvns@radiffmail.com

Follow this and additional works at: <https://www.interscience.in/ijcsi>

Part of the [Computer Engineering Commons](#), [Information Security Commons](#), and the [Systems and Communications Commons](#)

Recommended Citation

Singh, Sanjay; Tripathi, Nishant; Chaudhary, Anil Kumar; and Singh, Mahesh Kumar (2012) "An Introduction to RTOS," *International Journal of Computer Science and Informatics*: Vol. 1 : Iss. 4 , Article 12.

DOI: 10.47893/IJCSI.2012.1054

Available at: <https://www.interscience.in/ijcsi/vol1/iss4/12>

This Article is brought to you for free and open access by the Interscience Journals at Interscience Research Network. It has been accepted for inclusion in International Journal of Computer Science and Informatics by an authorized editor of Interscience Research Network. For more information, please contact sritampatnaik@gmail.com.

An Introduction to RTOS

Sanjay Singh, Nishant Tripathi, Anil Kumar Chaudhary & Mahesh Kumar Singh

Dept. of Electronics Design & Technology, DOEACC Society, Gorakhpur, UP, India
E-mail : dcsanjaypm@gmail.com, nishanttri@gmail.com, anil.viet@yahoo.com, maheshvns@radiffmail.com

Abstract - RTOS (real time operating system) can be defined as “The ability of the operating system to provide a required level of service in bounded response time.” A real time system responds in a (timely) predictable way to unpredictable external stimuli arrivals. To build a predictable system, all its components (hardware & software) should enable this requirement to be fulfilled. Traffic on a bus for example should take place in a way allowing all events to be managed within the prescribe time limit. However it should not be forgotten that a good RTOS is only a building block. Using it in a wrongly designed system may lead to a malfunctioning of the RT system. A good RTOS can be defined as one that has a bounded (predictable) behavior under all system load scenarios (simultaneous interrupts and thread execution). In RT system, each individual deadline should be met. Real-time systems are designed to control and monitor their environment. Most of these systems are using sensors to collect environment state and use actuators to change something.

Key words - RTOS, kernel, OS memory, intertask communication.

I. INTRODUCTION

RTOS is a multitasking operating system intended for real-time application. Such applications include

- Embedded system (programmable thermostats, household appliance controllers)
- Industrial robots, spacecraft, industrial control (SCADA)
- Scientific research equipment etc.

A RTOS facilitates the creation of a real time system, but does not guarantee the final result will be real-time, this requires correct development of the software. An RTOS does not necessarily have high throughput, rather an RTOS provides facilities which, if use properly, guarantee deadline can be met generally or deterministically (known as soft or hard real-time, respectively). An RTOS will typically use specialized scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behavior in the final system. An RTOS is valued more for how quickly and/ or predictably it can respond to particular event than for the amount of work it can perform over a given period of time. Key factors in an RTOS are therefore minimal interrupt latency and minimal thread switching latency.

II. DIFFERENCE BETWEEN DESKTOP OS AND RTOS

A real-time operating system is an operating system that allows one to specify constraints on the rate of

processes, and that guarantees that these rate constraints will be met, where as an operating system is a low level program that runs on a processor responsible for scheduling processes, allocating storage and interfacing to peripherals among many other things. Basically an operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs. Fig. 1 shows difference between RTOS is related to OS.

Application Program	Application Program	Application Program
OS User Interface Shell		
File and Disk Support		
OS Kernel (RTOS=OS Kernel)		
Hardware		

Fig.1: Relation between RTOS and OS Kernel

The key difference between general computing operating system and real time operating systems is the need for “deterministic” timing behavior in the real-time operating systems. Formally, “deterministic” timing means that operating system services consume only known and expected amounts of time. In theory, these services times could be expressed as mathematical formulas. These formulas must be strictly algebraic and not include any random timing components. Random elements in service times could cause random delays in

application software and could then make the application randomly miss real-time deadline a scenario clearly unacceptable for a real-time embedded system. Many non-real-time operating systems also provide similar kernel services. General-computing non-real-time operating systems are often quite non-deterministic. Their services can inject random delays into application software and thus cause slow responsiveness of an application at unexpected times. If you ask the developer of a non-real-time operating system for the algebraic formula describing the timing behavior of one of its services (such as sending a message from task to task), you will invariably not get an algebraic formula. Instead the developer of the non-real-time operating system (such as windows, UNIX and Linux) will just give you a puzzled look. Deterministic timing behavior was simply not a design goal for these general-computing operating systems.

On the other hand, real-time operating systems often go a step beyond basic determinism. For most kernel services, these operating systems offer constant load-independent timing. In other words, the algebraic formula is as simple as: $T(\text{message send}) = \text{constant}$, irrespective of the length of the message to be sent, or other factors such as the numbers of takes and queues and messages being managed by the RTOS.

Many RTOS proponents argue that a real-time operating system must not use virtual memory concept, because paging mechanics prevent a deterministic response. While this is a frequently supported argument, it should be noted that the term “real-time operating system” and determinism in this context cover a very wide meaning, and vendors of many different operating systems apply these terms with varied meaning. When selecting an operating system for a specific task, the real-time attribute alone is an insufficient criterion, therefore. Deterministic behavior and deterministic latencies have value only if the response lies within the boundaries of the physics of the process that is to be controlled. For example, controlling a combustion engine in a racing car has different real-time requirements to the problem of filling a 1000000 liter water tank through a 2” pipe.

Real-time operating systems are often used in embedded solutions, that is computing platforms that are within other device. Examples for embedded systems include combustion engine controllers or washing machine controllers and many others. Desktop PC and other general-purpose computer are not embedded systems. While real-time operating systems are typically designed for and use with embedded systems, the two aspects are essentially distinct, and have different requirements. A real-time operating system for embedded system addresses both set of requirements.

III. RTOS ARCHITECTURE

The heart of any RTOS is the kernel. Inside the kernel is the scheduler. It is basically a set of algorithms which manage the task running order. Multitasking definition comes from the ability of the kernel to control multiple tasks that must run within time deadlines. Multitasking may give the sensation that multiple threads are running concurrently, as a matter of fact the processor runs task by task, following scheduled order. RTOS structure is shown in fig.2.

Fig. 2 : RTOS Architecture

IV. PROGRAMMING WITH RTOS

4.1 Semaphores

Multitasking systems need inter processing communication. The communication depends on synchronization, which is realized with the semaphores, depending on the application there are three kinds of semaphores.

- Binary – this is most used.
- Mutual Exclusion (Mutex) – it’s similar to binary, but it is very useful to handle priority inheritance, deletion or recursion.
- Counting – it is used when many processor access the same semaphore.

4.2 Message Queues

Two tasks can communicate using a queue. It seems that task A sends directly a message to task B, but as matter of fact, they are sharing a storage area, and a third process (the operating system) takes care of transforming the data from one task to other. Using only a queue the communication can only be mono directional (eg. Task A-> task B). When a bidirectional (eg. Task A <-> task B) communication is needed, and then two queues must be used. Mailbox is, as well, a type of buffer managed by the operating system. Can be used by multiple tasks and is very similar to a FIFO structure.

4.3 Task Scheduling

The task scheduler establishes time slots. Time slots width and activation depend on available resources and priorities. Round Robin Method is one such example. This method handles tasks with same priority, using the “time slice” concept. Each task is allowed to run for a fixed time, at the end of which it stops. The operative system then activates the next ready task.

4.4 Signals

Task can receive notifications when an event is available from external environment within interrupt lines (hardware notification) or from another task within signals (software notification). A task that was already generated can have two states: delivered or pending, depending by the destination.

V. INTERTASK COMMUNICATION AND RESOURCES SHARING

Multitasking systems must manage sharing data and hardware resources among multiple tasks. It is usually “unsafe” for two tasks to access the same specific data or hardware resource simultaneously. (“Unsafe” means the results are inconsistent or unpredictable, particularly when one task is in the midst of changing a data collection. The views by another task are best done either before any change beings, or after changes are completely finished.) There are three common approaches to resolve this problem:

- Temporarily masking/ disabling interrupts
- Binary semaphores
- Message passing

General purpose operating systems usually do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wished. Modern CPUs make the interrupt disable control bit (or instruction) inaccessible in user mode to allow operating systems to prevent user tasks from doing this. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater system cell efficiency and also to permit the application to have greater control of the operating environment without requiring OS intervention.

On single processor systems, if the application runs in kernel mode and can mask interrupts, often that is the best (lowest overhead) solution to preventing simultaneous access to a shared resource. While interrupts are masked, the current task has exclusive use of the CPU, no other task or interrupt can take control, so the critical section is effectively protected. When the task exits its critical section, it must unmask interrupts,

pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the largest path through the critical section is shorter than the desired maximum interrupt latency, or else this method will increase the system’s maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few source code lines long and contains no loops. This method is ideal for protecting hardware bitmapped registers when the bits are controlled by different tasks.

When the critical section is longer than a few source code lines or involves lengthy looping, an embedded/ real-time programmer must resort to using mechanisms identical or similar to those available on general-purpose operating systems, such as semaphores and OS-supervised interprocess messaging. Such mechanisms involve system calls, and usually invoke the OS’s dispatcher code on exit, so they can take many hundreds of CPU instruction to execute, while masking interrupts may take as few as three instructions on some processors. But for longer critical sections, there may be no choice; interrupts cannot be masked for long periods without increasing the system’s interrupt latency.

A binary semaphore is either locked or unlocked. When it is locked, a queue of tasks can wait for the semaphore. Typically a task can set a timeout on its wait for a semaphore. Problems with semaphore based designs are well known, priority inversion and deadlocks.

In priority inversion, a high priority task waits because a low priority task has a semaphore. A typical solution is to have the task that has a semaphore run at (inherit) the priority of the highest waiting task. But this simplistic approach fails when there are multiple level of waiting (A waits for a binary semaphore locked by B, which waits for a binary semaphore locked by C). Handling multiple levels of inheritance without introducing instability in cycles is not straight forward.

In a deadlock, two or more tasks lock a number of binary semaphores and then wait forever (no timeout) for other binary semaphores, creating a cyclic dependency graph. The simplest deadlock scenario occurs when two tasks lock two semaphores in lockstep, but in the opposite order. Deadlock is usually prevented by careful design, or by having floored semaphores (which pass control of a semaphore to the higher priority task on defined conditions.)

The other approach to resource sharing is for tasks to send messages. In this paradigm, the resources is managed directly by only one task, when another task wants to interrogate or manipulate the resource, it sends a message to the managing task. This paradigm suffers from similar problems as binary semaphores. Priority

inversion occurs when a task working on a low priority message, and ignores a higher priority message (or a message originating indirectly from a high priority task) in its in-box. Protocol deadlock occur when two or more tasks wait for each other to send response messages. Although their real-time behavior is less crisp than semaphore systems, simple message based systems usually do not have protocol deadlock hazards, and are generally better behaved than semaphore systems.

VI. CHOOSING RIGHT RTOS

Selection an RTOS for an embedded application can be a complex process. Therefore, it is incumbent upon the software developer to exercise extra caution during the selection process. The four key areas to consider are

- Safety- Does the RTOS contribute to the device's safety or compromise it? Is it prone to user error?
- Performance- Can the RTOS facilitate the development of application code? Does the code perform within required parameters?
- Reliability- Does the RTOS impact the device's reliability?
- Functionality- Does the RTOS have the facilities required to do the job?

However, careful RTOS selection can simplify be the certification process by vendors and reduce cost. The source code must be available, and the more readable and well-documented it is, the better. Availability means the code can be easily changed if modifications are necessary. Having the code readily available can directly affect cost. Because the total cost of certification is significantly related to the size of the code, a small OS memory footprint provides a definite cost advantage. Size or memory footprint is another aspect of RTOS performance. Using less memory has clear benefits, but like speed, the measurement of size is not completely obvious. Most OS products are scalable, which means that only the code required for specific functionality is included in the final memory footprint.

Looking for granular scalability in an OS is worthwhile endeavor, as it minimizes memory usage. Data security is now a key issue for RTOS selection that (maintaining confidentiality) requires designers to applicant the latest protocol components and see which protocol is supported by which RTOS.

An RTOS is much better than normal operating systems because it is time sensitive, fault tolerable and allow multiprocessing, that are not supported by all operating systems. A real-time operating system is intended for systems that need to tightly control

responsiveness and performance. With an RTOS you can prioritize operations such that the most critical areas of the application get control of the processor exactly when they need it. All other lower priority operations are paused. An RTOS also gives you all of the other advantages of an OS.

- Consistent coding platform
- Rich set of APIs to save coding time and errors
- Processor housekeeping functions
- Hardware abstraction
- Priority based primitive scheduling
- Multitasking
- Multithreading
- Better response time
- Valuable services like semaphore, mailbox, queues etc.
- Priority based primitive scheduling

VII. CONCLUSION

It is said that the combination of hardware design along with the software embedded into memory is said to be known as the embedded system design. Embedded system design is related to RTOS in the sense that it take control of all interrupts that would be handled at time of execution of the task thus the user to free to do another jobs simultaneously.

Another advantage with the RTOS and the hardware is that it is replacing the ASIC based design because the cost of the ASIC is very high and its hardware cannot be altered, on the other hand the embedded system design when added with RTOS gives us the concept of reconfigurable computing.

REFERENCES

- [1] RTOS for FPGA-A while paper by Colin Wall, Accelerated Technology, Embedded Systems Division, Mentor Graphics Corporation.
- [2] Victor Yodaiken and Michael Barabanov "design document about RTLinux in SMLabs 1997.
- [3] Koza, John R., Genitic Programming: On the Programming of Computers by Means of Natural Selection. Combridge, MA: The MIT Press, 1992.
- [4] Cramer, Michael Lynn: "A Representation for the Adaptive Generation of Simple Sequential Programs" Proceedings, International Conference on Genetic Algo, July 1985 [CMU], pp183-187.

- [5] Wayne Wolf, Computers as Components: Principle of Embedded Computing System Design, Morgan Kaufmann Publishers, 2001.
- [6] Frank Vahid and Tony Givargis, Embedded System Design: A Unified hardware/ Software introduction, John Wiley & Sons, 2002.
- [7] Scott Rosenthal, Selecting an embedded processor involves both simple and non-technical criteria, Lume, 1997.