ISSN: 2277 – 9043
International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE)
Volume 2, Issue 2, February 2013

The Design of Real Time Operating Systems for Embedded Systems

Mr. Mahesh P.Gaikwad

Ashokrao Mane Group of Institutions,
Vathar-416 112

Abstract—

Embedded market place is booming, due to less expensive electronic components and new technologies. The prices of processors, memories, and other devices have been dropping, while their performance has been on the rise. This has made the implementation of many applications possible, when only a few years ago they were not.

The increased complexity of embedded applications and the intensified market pressure to rapidly develop cheaper products have caused the industry to streamline software development. Software development has been further streamlined with the advent of purchasing third party software modules, or intellectual property (IP), to perform independent functions required of the application, thereby shortening time-to-market. Finally, software development has been made simpler, quicker, and even cheaper with the incorporation of embedded operating systems.

1. INTRODUCTION

A real-time operating system (RTOS) is an operating system that guarantees a certain capability within a specified time constraint. For example, an operating system might be designed to ensure that a certain object was available for a robot on an assembly line. In what is usually called a "hard" real-time operating system, if the calculation could not be performed for making the object available at the designated time, the operating system would terminate with a failure. In a "soft" real-time operating system, the assembly line would continue to function but the production output might be lower as objects failed to appear at their designated time, causing the robot to be temporarily unproductive [2].

1.1 Architecture of RTOS

For simpler applications, RTOS is usually a kernel but as complexity increases, various modules like networking protocol stacks debugging facilities, device I/Os are includes in addition to the kernel.

The general architecture of RTOS is shown in the figure below.

ISSN: 2277 – 9043
International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE)
Volume 2, Issue 2, February 2013

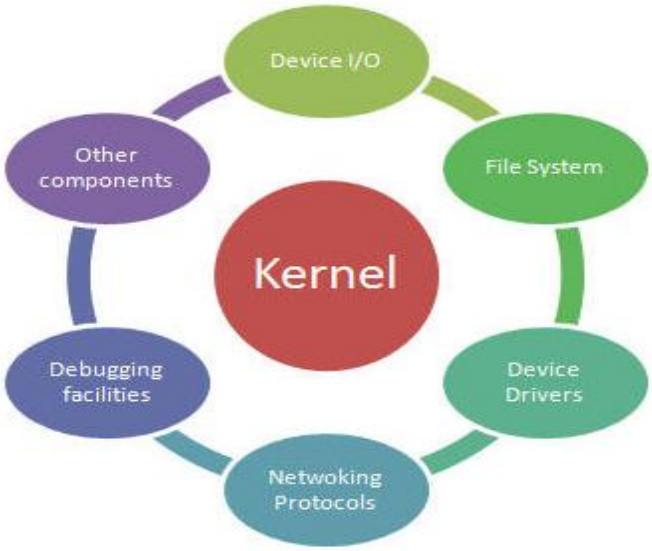

Fig.1.1 Architecture of RTOS

1.2 Development

RTOSes affect the real-time system development process in numerous ways. Some of the effects include hardware abstraction, multitasking, code size, learning curves, and the initial investment in the RTOS. None of these factors should be taken lightly.

Hardware abstraction or the mapping of processor dependent interfaces to processor independent interfaces is one advantage of RTOSes. For example, most processors include hardware timers. Each processor may have a completely different interface for communicating with their timers. Hardware abstraction will include functions in the RTOS that interface with the hardware timers and provide a standard API for the application-level code. This reduces the need to learn many of the details of how to interface with the various peripherals attached to a processor, thereby reducing development time. Hardware abstraction makes application code more portable.

Multitasking is an extremely useful aspect of RTOSes [8]. This is the ability for several threads of execution to run in pseudo-parallel. On most processors, only one task can be executing on a processor at a time. Multitasking is achieved by having a processor execute a task for a certain small interval of time and then execute another, and so forth, as seen in Figure 1.2.

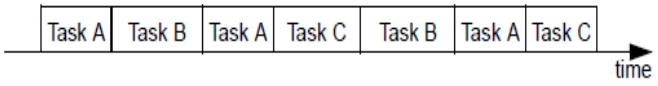

Figure 1.2: Multitasking. Tasks share the processor by using time-division multiplexing.

This is known as time-division multiplexing. The effect is that each task shares the processor and uses a fraction of the total processing power. If an RTOS supports *preemption*, then it will be able to stop or preempt a task in the middle of its execution and begin executing another. Whether or not an RTOS is preemptive has a large influence on the behavior of the real-time system. However, in some systems, preemption may cause problems known as race-conditions, which can lead to data corruption and deadlock. Fortunately, these problems can be solved with careful software development, so they are not a focus of this study. Whether preemption is supported or not, multitasking allows for the application to be divided into multiple tasks at logical boundaries, resulting in a system model as shown in Figure 1.3. This vastly simplifies the complexity of programming the application.

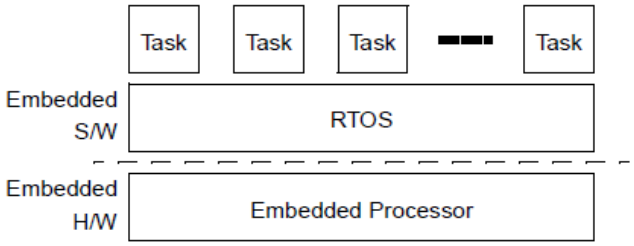

Figure 1.3: Model of a Real-Time System With an RTOS. A number of tasks are managed by an RTOS; all of which run on a microprocessor.

The code size must be taken into account when developing real-time systems. RTOSes often introduce quite a bit of code overhead to the system. Fortunately there are

several different RTOSes available with many different footprint sizes. Also, there are many RTOSes that are scalable, creating a variable sized footprint, depending on the amount of functionality desired.

Unfortunately, there are other overheads associated with RTOSes. There are several different operating systems, each supporting a limited number of processors and each with its own API to learn. The learning curve will increase development time whenever an RTOS is used that the developers are unfamiliar with. Also, if a proprietary one is used, it must be initially developed. If it is developed by a third party, it must be paid for, either on a one-time or per-unit basis.

These factors must each be taken into consideration when choosing an appropriate RTOS for a given design. Any one of them can have an extremely significant effect on the development process.

1.3 Performance

The use of RTOSes has several drastic effects on the performance of real time systems. Namely, they have great influence upon processor utilization, response time, and real-time jitter. All of these issues must be taken into consideration, before an RTOS is chosen. The processor utilization refers to the fraction of available processing power that an application is able to make use of. RTOSes often increase this fraction by taking advantage of the otherwise wasted time while a task is waiting for an external event and running other tasks. However, in order to provide the services available to a particular RTOS, including multitasking, preemption, and numerous others, a processing overhead is introduced. It is important to note that although this processing overhead may be significant, the services provided by the RTOS will simplify the application-level code and reduce the processing power required by the application. This will make up for some of the RTOS overhead. Also, many RTOSes are scalable, but they cannot be perfectly optimized for every application without devoting a significant amount of development time to them. In other words, since most RTOSes are designed to be general-purpose to at least some degree, they will introduce a processing overhead due to the functions they perform that are suboptimal or unnecessary for the given application. This is an unavoidable performance hit.

The response time is defined as the time it takes for the real-time system to response to external stimuli. As with an a periodic server model, this is defined as the time between the occurrence of an interrupt signal and the completion of a user-level response task, as illustrated in Figure below

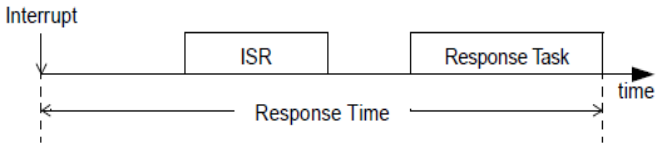

Figure 1.4: Response Time. The interrupt is serviced by its ISR as soon as interrupts are enabled. The ISR then triggers the response task, which the RTOS will execute as soon as it can.

The response task is generally triggered by an interrupt service routine (ISR). This delay is highly dependent on several factors, including whether or not the RTOS is preemptive and whether polling or interrupts are used to sense the stimulus. With preemption and interrupts, the system will have a much shorter average response time, because the current task does not have to complete before the response occurs. Without preemption, the system will have a widely distributed response time, but a smaller minimum response time, because there is less task state to maintain. The exact effects of RTOSes on response time are widely varying, but, in general, RTOSes increase response time to at least some degree. This is due to the additional processing the RTOS performs that is required to maintain the precise state of the system, including the status of each task. However, the magnitude of this effect can be reduced if the RTOSes have been optimized for response time.

Several definitions of real-time jitter exist, most of which are based upon the variation in the execution times of a given periodic task. This variation is caused by interference with interrupts and other tasks, as shown in Figure 1.5. A

ISSN: 2277 – 9043
International Journal of Advanced Research in Computer Science and Electronics Engineering (IJARCSEE)
Volume 2, Issue 2, February 2013

---

great deal of this jitter is caused by the nature of the application level code and is unavoidable. However, the RTOS can significantly increase the amount of jitter and reduce the predictability of the real-time system's behavior. This lack of predictability is due to portions of the RTOS code that are called frequently and with execution times that are quite large or that change with respect to variable system parameters. This reduced predictability may prohibit an application from meeting its design constraints. When an RTOS is being evaluated for a design, its performance effects must be carefully considered. A real-time system's behavior can almost completely depend on the design of the RTOS.

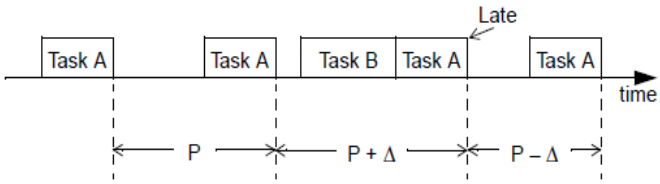

Figure 1.5: Real-Time Jitter. The execution ofTask B pushes back the 3rd execution ofTask A, causing the task completion times to deviate from their ideally periodic nature.

2. EMBEDDED SYSTEMS

All embedded systems contain a processor and software. The processor may be 8051 micro-controller or a Pentium-IV processor (having a clock speed of 2.4 GHz). Certainly, in order to have software there must be a place to store the executable code and temporary storage for run-time data manipulations. These take the form of ROM and RAM respectively. If memory requirement is small, it may be contained in the same chip as the processor. Otherwise one or both types of memory will reside in external memory chips. All embedded systems also contain some type of inputs and outputs (Fig. 1). For example in a microwave oven the inputs are the buttons on the front panel and a temperature probe and the outputs are the human readable display and the microwave radiation. Inputs to the system generally take the form of sensors and probes, communication signals, or control knobs and buttons. Outputs are generally displays, communication signals, or changes to the physical world.

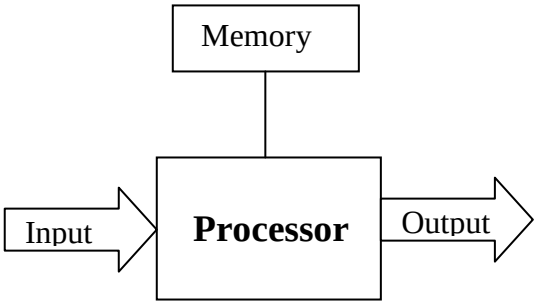

Fig. 2.1 Generic Embedded System

Today's embedded market place is booming, due to less expensive electronic components and new technologies. The prices of processors, memories, and other devices have been dropping, while their performance has been on the rise. This has made the implementation of many applications possible, when only a few years ago they were not. Several key technologies— the Internet, MPEG audio and video, GPS, DVD, DSL, and many more—have further expanded the realm of possibility and created new market sectors. These lucrative new opportunities have caught the attention of countless corporations and entrepreneurs, creating competition and innovation. This is good for the consumer, because the industry is under a great deal of pressure to develop products with quick time-to-market turnaround and to sell them as inexpensively as possible.

The increased complexity of embedded applications and the intensified market pressure to rapidly develop cheaper products have caused the industry to streamline software development. Logically, embedded software engineers have looked at how this problem has already been addressed in other areas of software development. One obvious solution has been the increased use of high-level languages, such as C, C++, and Java. Surprisingly, low-level assembly is still heavily used today, mostly because of simplistic applications, compiler inefficiency, and poor compiler target ability, due to complicated memory models and application specific instructions, such as the multiply-and-accumulate (MAC) instruction. However, these factors are no longer holding back

high-level languages as applications become more complex, compiler technology evolves, and processor architects eliminate poor compiler target ability. The emergence of powerful integrated development environments (IDEs) for embedded software has significantly contributed to making software development faster, simpler, and more efficient [19]. Software development has been further streamlined with the advent of purchasing third party software modules, or intellectual property (IP), to perform independent functions required of the application, thereby shortening time-to-market. Finally, software development has been made simpler, quicker, and even cheaper with the incorporation of embedded operating systems. Unfortunately, operating systems do introduce several forms of overhead that must be minimized.

Real-time systems are embedded systems in which the correctness of an application implementation is not only dependent upon the logical accuracy of its computations, but its ability to meet its timing constraints as well. Simply put, a system that produces a late result is just as bad as a system that produces an incorrect result. Because of this need to meet timing requirements, implementations of real-time systems must behave as predictably as possible. Thus, their supporting software must be written to take this into account. The operating systems used in real-time systems—

real time operating systems (RTOSes)—are no exception. Therefore, in addition to their need to minimize overhead, RTOSes also have the goal of maximizing their predictability. Whether or not an RTOS can be used for a particular application depends upon its ability to optimize these constraints to within specified tolerance levels. This can prove to be quite difficult with modern embedded processor and RTOS designs.

So real time operating systems are an integral part of real time systems [7].Not surprisingly, four main functional areas that they support are scheduling, process management and synchronization, interrupts and memory management.

3. Scheduling

This part of the operating system decides which of the ready tasks has the right to use the processor at a given time [1]. Embedded operating systems cannot use any of the simple scheduling algorithms like FCFS, SJF, and RR etc. Embedded systems, particularly real-time systems, almost always require a way to share the processor that allows the most important tasks to grab the control of processor as soon as they need it. A deadline driven scheduling mechanism is the ideal one. However, the current state of technology does not allow this. Therefore most embedded operating systems utilize a priority based scheduling algorithm that supports pre-emption. We also need that interrupt handling in case of different simultaneous interrupts should be handled in a preemptive way.

A good embedded RTOS should have provision for lot of priority levels. A number of high priority levels have to be dedicated to the system processes and threads. And in a complex application with large number of threads, it is essential to be able to place all the real-time threads on a different priority level above the non real-time threads.

There is also necessary to have a backup scheduling policy. This is the scheduling algorithm to be used in the event that several ready tasks have same priority. The most common backup algorithm used is the *round robin*. If there are no tasks in ‘ready state’ when a scheduler is called, the idle task will be executed which is basically an infinite loop that does nothing. Idle task will have the lowest priority and will always be in ready state.

The actual process of changing from one task to another is called a context switch. Since the contexts are processor-specific, the code that implements this is also processor-specific. So it is always written in assembly language. For real-time systems the context switch should take only the bare minimum of time because this determines the response.

4. Process synchronization and Communication

There are several tools available in RTOS to enable inter task communication and task synchronization.

4.1 Semaphores: Semaphores are intertask communication tools used to protect shared data resources. Tasks can call Take-Semaphore and Release-Semaphore functions. If one task has called Take-Semaphore and has not called the Release-Semaphore to release it, then any other task that calls Take-Semaphore will block until first task calls Release-Semaphore. For example we can protect the shared data by taking the semaphore before modifying and releasing it only after that. Whenever task takes a semaphore it is potentially slowing the response of any other task that needs the same semaphore.

Semaphore can also act as a signaling device for synchronization. For example, a task that formats printed reports builds those reports into a fixed memory buffer. After formatting one report into the buffer the task must wait until interrupt routine has finished printing. Here the task can wait for a semaphore after it has formatted a report. The interrupt routine on feeding the report to printer can release the semaphore. The task on receiving the semaphore formats the next report. When using Semaphores, one should ensure that it does not lead to Priority inversion or deadly embrace. Some RTOS have a method called priority inheritance to tackle this problem.

4.2 Message Mailboxes: Messages are sent to a task using kernel services called message mailbox. Mailbox is basically a pointer size variable. Tasks or ISRs can deposit and receive messages (the pointer) through the mailbox. A task looking for a message from an empty mailbox is blocked and placed on waiting list for a time(time out specified by the task) or until a message is received. When a message is sent to the mail box, the highest priority task waiting for the message is given the message in priority-based mailbox or the first task to request the message is given the message in FIFO based mailbox.

4.3 Message Queues: It is used to send one or more messages to a task. Basically Queue is an array of mailboxes. Tasks and ISRs can send and receive messages to the Queue through services provided by the kernel. Extraction of messages from a queue may follow FIFO or LIFO fashion. When a message is delivered to the queue either the highest priority task (Priority based) or the first task that requested the message (FIFO based) is given the message.

4.4 Event Flags: Basically these are Boolean flags which tasks can set or reset that other tasks can wait for. Event flags are used in cases where a task has to synchronize with occurrence of multiple events. A task can be synchronized when any of the events have occurred as in disjunctive synchronization (logical OR) or may be synchronized when all the events have occurred as in conjunctive synchronization (logical AND).More than one task can wait for same event. RTOS can form groups of events and tasks can wait for any subset of events in a group.

5. Interrupts

"An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred[2]".When CPU recognizes an interrupt, it saves its context and jumps to a subroutine known as Interrupt Service routine (ISR).Upon completion of the ISR the program returns to

a) The background in the case of foreground/background system.

b) Interrupted task in case of a non-pre-emptive kernel.

c) The highest priority task that is ready to run in case of a preemptive kernel.

Each OS needs to disable interrupts from time to time to execute critical code that should not be interrupted. The number of lines of this code should be minimum and bound under all circumstances. ISR must not call any RTOS function that might get blocked. An ISR must not call any RTOS function that might cause RTOS to switch task states unless RTOS knows that an ISR and not a task is running. A good

RTOS should have shortest Interrupt latencies, interrupt responses and interrupt recovery times. The ISR processing time also must be kept to the minimum for the best real time response.

6. Memory management

Each programs need to held in a memory generally in a ROM to be executed. The task data (stack and registers) and all variables must be stored in RAM. In a real-time system the main requirement is that the access time should be bound or predictable. The use of demand paging is not allowed since the systems providing virtual memory mechanisms use memory swapping which is not predictable. RTOS have fast and predictable functions to allocate and free fixed size buffers. RTOS allows to setup pools each of which consist of same number of memory buffers. In any given pool all buffers are of same size. In many circumstances it is not acceptable for hardware failure to corrupt data in memory. In such instance hardware protection mechanism should be used. In Hard Real-time systems static memory allocation is used. In a Soft Real-time system dynamic memory allocation is preferred.

7. Conclusion

A real time operating system is an operating system for embedded computer systems. These operating systems are designed to be compact, efficient at resource usage, and reliable, forsaking many functions that non-embedded computer operating systems provide, and which may not be used by the specialized applications they run.

As the complexities in the embedded applications increase, use of an operating system brings in lot of advantages. Most embedded systems also have real-time requirements demanding the use of Real time Operating Systems (RTOS) capable of meeting the embedded system requirements. Real-time Operating System allows real time applications to be designed and expanded easily. The use of an RTOS simplifies the design process by splitting the application code into separate tasks. An RTOS allows one to make better use of the system recourses by providing with valuable services.

8. References

[1] K. Altisen, G. Goessler, and J. Sifakis. "Scheduler modeling based on the controller synthesis paradigm," Journal of Real- Time Systems, special issue on Control Approaches to Real- Time Computing, 23:55–84, 2002.

[2] P. Binns and S. Vestal. "Formalizing software architectures for embedded systems," Proceedings of the First Int. Conference on Embedded Software (EMSOFT 2001), Tahoe City, California, Springer, LNCS 2211, October 2001.

[3] D. L. Levine, D. C. Schmidt, and S. Flores-Gaitan, "An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers," in Proceedings of Multimedia Computing and Networking 2000 (MMCN00), (San Jose, CA), ACM, Jan. 2000.

[4] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet, vol. 21, no. 2, 2001.

[6] L. Abeni and G. Buttazzo, "Resource Reservation in Dynamic Real-Time Systems," *Real-Time Systems*, Vol. 27, No.2, pp. 123-167, July 2004.

[7] G. Buttazzo, "Achieving Scalability in Real-Time Systems," IEEE Computer, Vol. 39, No. 5, pp. 54–59, May 2006.

[8] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment," Journal of the ACM, No. 1, Vol. 20, pp. 40-61, 1973.