A Study of Real-Time Embedded Software Systems and Real-time Operating Systems

M. Tech Seminar Report

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

by

Shonil Vijay

Roll No: 05329001

under the guidance of

Dr. Kavi Arya

Kanwal Rekhi School of Information Technology

Indian Institute of Technology, Bombay

Mumbai

Acknowledgemnts

I would like to thank Dr. Kavi Arya and Prof. Krithi Ramamritham for their invaluable support and guidance.

Shonil Vijay

Abstract

Embedded systems are the computing devices hidden inside a vast array of everyday products and appliances such as cell phones, toys, handheld PDAs, cameras, etc. An embedded system is various type of computer system or computing device that performs a dedicated function and/or is designed for use with a specific embedded software application. Embedded systems may use a combination of 'Read-only' as well as with 'Read-Write' based operating system. But an embedded system is not usable as a commercially viable substitute for general-purpose computers or devices. As applications grow increasingly complex, so do the complexities of the embedded computing devices.

An embedded real-time operating system is the software program that manages all the programs in an embedded device after initial load of programs by a boot loader. It normally guarantees a certain capability within a specified storage size and time constraint as well as with application programs. It also normally has small foot print including initial boot loader, OS kernel, required device drivers, file systems for the user data and so forth. It has very-likely structure of a normal operating system however mainly differentiated by some factors such as type of pre-installed device, functional limits, taking designed job only.

This paper attempts to throw some light on the technologies behind the embedded systems design and concludes by the survey of some of the available real-time operating systems.

Contents

1. Introduction

Last few decades have seen the rise of computers to a position of prevalence in human affairs. It has made its mark in every field ranging personal home affairs, business, process automation in industries, communications, entertainment, defense etc.

An embedded system is a combination of hardware and software and perhaps other mechanical parts designed to perform a specific function. Microwave oven is a good example of one such system. This is in direct contrast to a personal computer. Though it is also comprised of hardware and software and mechanical components it is not designed for a specific purpose. Personal computer is general purpose and is able to do many different things.

2. What is a real-time System

A real-time system is one whose correctness involves both the logical correctness of outputs and their timeliness. It must satisfy response-time constraints or risk severe consequences including failure. As defined by Donald Gillies "A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time in which the result is produced. If the timing constraints are not met, system failure is said to have occurred."

These systems respond to a series of external inputs, which arrive in an unpredictable fashion. The real-time systems process these inputs, take appropriate decisions and also generate output necessary to control the peripherals connected to them. The design of a real-time system must specify the timing requirements of the system and ensure that the system performance is both correct and timely. There are three types of time constraints:

- Hard: A late response is incorrect and implies a system failure.

- Soft: Timeliness requirements are defined by using an average response time. If a single computation is late, it is not usually significant, although repeated late computation can result in system failures.

- Firm: Firm real-time systems have hard deadlines, but where a certain low probability of missing a deadline can be tolerated.

Most real-time systems interface with and control hardware directly. The software for such systems is mostly custom-developed. Real-time Applications can be either embedded applications or non-embedded (desktop) applications. Real-time systems often do not have standard peripherals associated with a desktop computer, namely the keyboard, mouse or conventional display monitors. In most instances, real-time systems have a customized version of these devices.

3. What are Embedded Systems

An embedded system is generally a system within a larger system. Modern cars and trucks contain many embedded systems. One embedded system controls anti-lock brakes, another monitors and controls vehicle's emission and a third displays information on the dashboard. Even the general-purpose personal computer itself is made up of numerous embedded systems. Keyboard, mouse, video card, modem, hard drive, floppy drive and sound card are each an embedded system.

Tracing back the history, the birth of microprocessor in 1971 marked the booming of digital era. Early embedded applications included unmanned space probes, computerized traffic lights and aircraft flight control systems.

In the 1980s, embedded systems brought microprocessors into every part of our personal and professional lives. Presently there are numerous gadgets coming out to make our life easier and comfortable because of advances in embedded systems. Mobile phones, personal digital assistants and digital cameras are only a small segment of this emerging field.

Embedded systems do not provide standard computing services and normally exist as part of a bigger system. Embedded systems are usually constructed with the least powerful computers that can meet the functional and performance requirements. This is essential to lower the manufacturing cost of the equipment.

Other components of the embedded system are similarly chosen, so as to lower the manufacturing cost. In conventional operating systems, a

programmer needing to store a large data structure can allocate big chunks of memory without having to think of the consequences. These systems have enough main memory and a large pool of virtual memory (in the form of disk space) to support such allocations. The embedded systems' developers do not enjoy such luxuries and have to manage with complex algorithms to manage resources in the most optimized manner.

One major subclass of embedded systems is real-time embedded systems. A real-time system is one that has timing constraints. Real-time system's performance is specified in terms of ability to make calculations or decisions in a timely manner. These important calculations have deadlines for completion. A missed deadline is just as bad as a wrong answer. The damage caused by this miss will depend on the application. For example if the real-time system is a part of an airplane's flight control system, single missed deadline is sufficient to endanger the lives of the passengers and crew.

In most of the real-life applications, real-time systems often work in an embedded scenario and most of the embedded systems have real-time processing needs. Such software is called Real-time Embedded Software systems.

3.1 Inside an Embedded System

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 2.4GHz). 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.

Within the exception of these few common features, rest of the embedded hardware is usually unique and varies from application to application. Each system must meet a completely different set of requirements.

The common critical features and design requirements of an embedded hardware include:

1. Processing power: Selection of the processor is based on the amount of processing power to get the job done and also on the basis of register width required.
2. Throughput: The system may need to handle a lot of data in a short period of time.
3. Response: the system has to react to events quickly
4. Memory: Hardware designer must make his best estimate of the memory requirement and must make provision for expansion.
5. Power consumption: Systems generally work on battery and design of both software and hardware must take care of power saving techniques.
6. Number of units: the no. of units expected to be produced and sold will dictate the Trade-off between production cost and development cost
7. Expected lifetime: Design decisions like selection of components to system development cost will depend on how long the system is expected to run.
8. Program Installation: Installation of the software on to the embedded system needs special tools.
9. Test & Debug ability: setting up test conditions and equipment will be difficult and finding out what is wrong with the software will become a difficult task without a keyboard and the usual display screen.
10. Reliability: is critical if it is a space shuttle or a car but in case of a toy it doesn't always have to work right.

4. Real-time Operating systems

Real-time computing is where system correctness not only depends on the correctness of logical result but also on the result delivery time. So the operating system should have features to support this critical requirement to render it to be termed a Real-time operating System (RTOS).

The RTOS should have predictable behavior to unpredictable external events. “A good RTOS is one that has a bounded (predictable) behavior under all system load scenario i.e. even under simultaneous interrupts and thread execution.” A true RTOS will be deterministic under all conditions.

These operating systems occupy little space from 10 KB to 100KB as compared to the General Operating systems which take hundreds of megabytes.

We observe that the choice of an operating system is important in designing a real-time system. Designing a real-time system involves choice of a proper language, task partitioning and merging, and assigning priorities to manage response times. Depending upon scheduling objectives, parallelism and communication may be balanced. Merging highly cohesive parallel tasks for sequential execution may reduce overheads of context switches and inter-task communications.

The designer must determine critical tasks and assign them high priorities. However, care must be taken to avoid starvation, which occurs when higher priority tasks are always ready to run, resulting in insufficient processor time for lower priority tasks. Non-prioritized interrupts should be avoided if there is a task that cannot be preempted without causing system failure. Ideally, the interrupt handler should save the context, create a task that will service the interrupt, and return control to the operating system.

Using a task to perform bulk of the interrupt service allows the service to be performed based on a priority chosen by the designer and helps preserve the priority system of the RTOS. Furthermore, good response times may require small memory footprints in resource-impoverished systems. Clearly the choice of an RTOS in the design process is important for support of priorities, interrupts, timers, inter-task communication, synchronization, multiprocessing and memory management.

4.1 Basic Requirements of an RTOS:

The following are the basic requirements for an RTOS:

(i) *Multi-tasking and preemptable*: To support multiple tasks in real-time applications, an RTOS must be multi-tasking and preemptable. The scheduler should be able to preempt any task in the system and give the resource to the task that needs it most. An RTOS should also handle multiple levels of interrupts to handle multiple priority levels.

(ii) *Dynamic deadline identification:* In order to achieve preemption, an RTOS should be able to dynamically identify the task with the earliest deadline. To handle deadlines, deadline information may be converted to priority levels that are used for resource allocation. Although such an approach is error prone, nonetheless it is employed for lack of a better solution.

(iii) *Predictable synchronization:* For multiple threads to communicate among themselves in a timely fashion, predictable inter-task communication and synchronization mechanisms are required. Semantic integrity as well as timeliness constitutes predictability. Predictable synchronization requires compromises. Ability to lock/unlock resources is one of the ways to achieve data integrity.

(iv) *Sufficient Priority Levels:* When using prioritized task scheduling, the RTOS must have a sufficient number of priority levels, for effective implementation. Priority inversion occurs when a higher priority task must wait on a lower priority task to release a resource and in turn the lower priority task is waiting upon a medium priority task. Two workarounds in dealing with priority inversion, namely priority inheritance and priority ceiling protocols (PCP), need sufficient priority levels. In a priority inheritance mechanism, a task blocking a higher priority task inherits the higher priority for the duration of the blocked task. In PCP, a priority is associated with each resource which is one more than the priority of its highest priority user. The scheduler makes the priority of the accessing task equal to that of the resource. After a task releases a resource, its priority is returned to its original value. However, when a task's priority is increased to access a resource it should not have been waiting on another resource.

(v) *Predefined latencies:* The timing of system calls must be defined using the following specifications:

- *Task switching latency* or the time to save the context of a currently executing task and switch to another.
- *Interrupt latency* or the time elapsed between the execution of the last instruction of the interrupted task and the first instruction of the *interrupt handler*.
- *Interrupt dispatch latency* or the time to switch from the last instruction in the *interrupt handler* to the next task scheduled to run.

4.2 Memory Management

An RTOS uses small memory size by including only the necessary functionality for an application while discarding the rest. Below we discuss static and dynamic memory management in RTOSs. Static memory management provides tasks with temporary data space. The system's free memory is divided into a pool of fixed sized memory blocks, which can be requested by tasks. When a task finishes using a memory block it must return it to the pool. Another way to provide temporary space for tasks is via priorities. A pool of memory is dedicated to high priority tasks and another to low priority tasks. The high-priority pool is sized to have the worst-case memory demand of the system. The low priority pool is given the remaining free memory. If the low priority tasks exhaust the low priority memory pool, they must wait for memory to be returned to the pool before further execution.

Dynamic memory management employs memory swapping, overlays, multiprogramming with a fixed number of tasks (MFT), multiprogramming with a variable number of tasks (MVT) and demand paging. Overlays allow programs larger than the available memory to be executed by partitioning the code and swapping them from disk to memory.

In MFT, a fixed number of equalized code parts are in memory at the same time. As needed, the parts are overlaid from disk. MVT is similar to MFT except that the size of the partition depends on the needs of the program in MVT. Demand paging systems have fixed-size pages that reside in non-contiguous memory, unlike those in MFT and MVT. In many embedded systems, the kernel and application programs execute in the same space i.e., there is no memory protection.

4.4 Task Scheduling

This section discusses scheduling algorithms employed in real-time operating systems. We note that predictability requires bounded operating system primitives. A feasibility analysis of the schedule may be possible in some instances.

Task scheduling can be either performed preemptively or non-preemptively and either statically or dynamically. For small applications, task execution times can be estimated prior to execution and the preliminary task schedules statically determined. Two common constraints in scheduling are the resource requirements and the precedence of execution of the tasks.

Typical parameters associated with tasks are:

- Average execution time
- Worst case execution time
- Dispatch costs
- Arrival time
- Period (for periodic tasks).

The objective of scheduling is to minimize or maximize certain objectives. Typical objectives minimized are: schedule-length and average tardiness or laxity. Alternatively, maximizing average earliness and number of arrivals that meet deadlines can be objectives. Scheduling approaches have been classified into: static table driven approach, static priority driven preemptive approach, dynamic planning based approach, dynamic best effort approach, scheduling with fault tolerance and resource reclaiming. These are briefly discussed below:

(i) *Static table driven*: The feasibility and schedule are determined statically. A common example is the cyclic executive, which is also used in many large-scale dynamic real-time systems. It assigns tasks to periodic time slots. Within each period, tasks are dispatched according to a table that lists the order to execute tasks. For periodic tasks, there exists a feasible schedule if and only if there is a feasible schedule within the least common multiple of the periods. A disadvantage of this approach is that a-priori knowledge of the maximum requirements of tasks in each cycle is necessary.

(ii) *Static priority driven preemptive*: The feasibility analysis is conducted statically. Tasks are dispatched dynamically based upon priorities. The most commonly used static priority driven preemptive scheduling algorithm for periodic tasks is the Rate Monotonic (RM) scheduling algorithm.

A periodic system must respond with an output before the next input. Therefore, the system's response time should be shorter than the minimum time between successive inputs. RM assigns priorities proportional to the frequency of tasks. It can schedule any set of tasks to meet deadlines if the total resource utilization less than *ln* 2. If it cannot find a schedule, no other fixed-priority scheduling scheme will. But it provides no support for dynamically changing task periods/priorities and priority inversion. Also, priority-inversion may occur when to enforce rate-monotonicity, a noncritical task of higher frequency of execution is assigned a higher priority than a critical task of lower frequency of execution.

(iii) *Dynamic planning based*: The feasibility analysis is conducted dynamically—an arriving task is accepted for execution only when feasible. The feasibility analysis is also a source for schedules. The execution of a task is guaranteed by knowing its worst-case execution time and faults in the system. Tasks are dispatched to sites by brokering resources in a centralized fashion or via bids. A technique using both centralized and bidding-approach performs marginally better than any one of them but is more complex.

(iv) *Dynamic best effort approach*: Here no feasibility check is performed. A best effort is made to meet deadlines and tasks may be aborted. However, the approaches of Earliest Deadline First (EDF) and Minimum Laxity First (MLF) are often optimal when there are no overloads. Research into overloaded conditions is still in its infancy. Earliest deadline first (EDF) scheduling can schedule both static and dynamic real-time systems. Feasibility analysis for EDF can be performed in $O(n^2)$ time, where n is the number of tasks. Unlike EDF, MLF accounts for task execution times.

(v) *Scheduling with fault tolerance*: A primary schedule will run by the deadline if there is no failure and a secondary schedule will run by the deadline on failure. Such a technique allows graceful degradation but incurs cost of running another schedule. In hard real-time systems, worst-case blocking must be minimized for fault tolerance.

(vi) *Scheduling with resource reclaiming*: The actual task execution time may be shorter than the one determined a-priori because of conditionals or worst-case execution assumptions. The task dispatcher may try to reclaim such slacks, to the benefit of non real-time tasks or improved timeliness guarantees.

5. Case Studies

Some of the popular RTOSs are reviewed here to identify their salient features which make them suitable for different embedded real-time applications. One of the General Purpose Operating Systems is also discussed here to highlight why a General Purpose Operating System is not suitable for real-time applications.

5.1 QNX RTOS v6.1

The QNX RTOS v6.1 has a client-server based architecture. QNX adopts the approach of implementing an OS with a 10 Kbytes micro-kernel surrounded by a team of optional processes that provide higher-level OS services .Every process including the device driver has its own virtual memory space. The system can be distributed over several nodes, and is network transparent. The system performance is fast and predictable and is robust.

It supports Intel x86family of processors, MIPS, PowerPC, and StrongARM. Documentation is extensive except for the details on the APIs [10]. QNX has successfully been used in tiny ROM-based embedded systems and in several-hundred node distributed systems

5.2 VRTX

VRTX has multitasking facility to solve the real-time performance requirements found in embedded systems. Pre-emptive scheduling is followed ensuring the best response for critical applications. Inter-task communication is by use of mailboxes and queues. Mailbox is equivalent to an event signal and events can pass data along with the event. Queues can hold multiple messages and this buffering facility is useful when sending task produces messages faster than the receiving task can handle them. Dynamic memory allocation is supported and allocation and release is in fixed size blocs to ensure predictable response times.

VRTX has been designed for development and target system independence as well as real-time clock independence. VRTX provides core services which every microprocessor can use to its advantage.

5.3 Windows CE 3.0

Windows CE 3.0 is an Operating system rich in features and is available for a variety of hardware platforms. It exhibits true real-time behavior most of the times. But the thread creation and deletion has periodic delays of more than 1 millisecond occurring every second. The system is complex and highly configurable. The configuration of CE 3.0 is a complicated process. The documentation does not give in depth knowledge about inner workings of the system though the APIs are well documented. The system is robust and no memory leak occurs even under stressed conditions. CE 3.0 uses virtual memory protection to protect itself against faulty applications.

5.4 pSOSystem/x86 2.2.6

pSOS+ is a small kernel suitable for embedded applications. This uses the software bus to communicate between different modules. The choice of module to be used can be done at compile time making it suitable for embedded applications. System has a flat memory space. All threads share the same memory space and also share all objects such as semaphores. So it has more chances of crashing. Around 239 usable thread priority levels available making it suitable for Rate monotonic scheduling. pSOS has a multiprocessor version pSOS+m which can have one node as master and a number of nodes as slaves. Failure in master will however lead to system crash. The Integrated Development Environment is comprehensive and is available for both Windows and UNIX systems. The drawback of this RTOS is that it is available only for selected processors and that lack of mutexes in some versions leads to priority inversion.

5.5 VxWorks (Wind River Systems)

VxWorks is the premier development and execution environment for complex real-time and embedded applications on a wide variety of target processors. Three highly integrated components are included with VxWorks: a high performance scalable real-time operating system which executes on a target processor; a set of powerful cross-development tools; and a full range of communications software options such as Ethernet or serial line for the target connection to the host. The heart of the OS is the Wind microkernel which supports multitasking, scheduling, intertask management and memory management. All other functionalities are through processes. There is no privilege protection between system and application and also the support for communication between processes on different processors is poor.

5.6 Windows NT

The overall architecture is good and may be a suitable RTOS for control systems that need a good user interface and can tolerate the heavy recourse requirements demanded for installation. It needs hard disk and a powerful processor. Configuration and user interaction requires a dedicated screen and keyboard. The choice of selecting components for installation is limited and it is not possible to load and unload major components dynamically. Because of all these limitations Windows NT not suitable for embedded applications. It is neither suitable for other real time applications because of the following factors:

a) There are only 7 priority levels & there is no mechanism to avoid priority inversion

b) The Queue of threads waiting on a semaphore is held in a FIFO order. Here there is no regard for priority, hampering the response times of highest priority tasks.

c) Though ISR responses are fast, the Deferred Procedure Calls (DPC) handling is a problem since they are managed in a FIFO order.

d) The thread switch latency is high (~ 1.2 ms), which is not acceptable in many real-time applications.

6. Conclusion

Real time Operating systems play a major role in the field of embedded systems especially for mission critical applications are involved. Selection of a particular RTOS for an application can be made only after a thorough study of the features provided by the RTOS.

Since IC memories are getting denser scaled down versions of general operating systems are able to compete with traditional Real Time Operating Systems for the embedded product market. The choice of Operating System generally comes after the selection of the processor and development tools.

Every RTOS is associated with a finite set of microprocessors and a suite of development tools. Hence the first step in choosing an RTOS must be to make the processor, real-time performance and the budget requirements clear. Then look at the available RTOS to identify the one which suits our application.

Generally an RTOS for embedded application should have the following features

Open Source, Portable, ROM able, Scalable, Pre-emptive, Multi-tasking, Deterministic, Efficient Memory Management, Rich in Services, Good Interrupt Management, Robust and Reliable

Within the class of real-time embedded systems, the general feature is that system and its application are fixed for the life of a product or the system. Thus there is a real need for a general purpose architecture which would be flexible enough to meet the varied requirements of these systems (wide range of sensors, threats, and scenarios), but which would still be dedicated and matched to an application through the use of special configurations of general modules. Even though most of the current kernels (RTOS) are successfully used in todays real-time embedded systems, but they increase the cost and reduce flexibility. Next generation real-time operating systems would demand new operating systems and task designs to support predictability, and high degree of adaptability.

7. References

[1] Dedicated Systems Experts, "*What makes a good RTOS*." Brussels, Belgium: Dedicated Systems Experts, 2001.

[2] James F. Ready, *"VRTX: A Real-Time Operating System for embedded Microprocessor Applications," IEEE Micro*. 6(4), Aug.1986, pp.8-17.

[3] Dedicated Systems Experts, *RTOS Evaluation Project*. Brussels, Belgium: Dedicated Systems Experts, 2001.

[4] S. Baskiyar ad N. Meghanathan, "A survey of contemporary Real-Time Operating Systems," Informatica 29(2005), pp. 233-240.

[5] P.A. Laplante, "*Real-Time Systems Design and Analysis*: *An Engineer's Handbook,*" Second edition, IEEE Press, 1997.

[6] C. Walls, *"RTOS for Microcontroller Applications,"* Electronic Engineering, vol. 68, no. 831, pp. 57-61, 1996.

[7] C.L. Liu and J.W. Layland, *"Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment,"* Journal of the ACM, v. 20, no. 1, pp. 46-61, 1973.

[8] K. Ramamritham and J. A. Stancovic, *"Scheduling Algorithms and Operating Systems Support for Real-time Systems,"* Proceedings of the IEEE, pp. 55-67, Jan 1994.