M.Tech. credit seminar report,
Electronic Systems Group, EE Dept, IIT Bombay,
Submitted in November 2002.

Embedded Operating Systems for Real-Time Applications

Sagar P M (02307406)
Supervisor: Prof. Vivek Agarwal

Abstract **: The advent of microprocessors has opened up several product opportunities that simply did not exist earlier. These intelligent processors have invaded and embedded themselves into all fields of our lives be it the kitchen (food processors, microwave ovens), the living rooms (televisions, airconditioners) or the work places (fax machines, pagers, laser printer, credit card readers) …etc.**

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 such as semaphores, mailboxes, queues, time delays, time outs…etc.

This report looks at the basic concepts of embedded systems, operating systems and specifically at Real Time Operating Systems in order to identify the features one has to look for in an RTOS before it is used in a real-time embedded application. Some of the popular RTOS have been discussed in brief, giving their salient features, which make them suitable for different applications.

I. 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.

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 [2].

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.

II. 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 2.4 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.

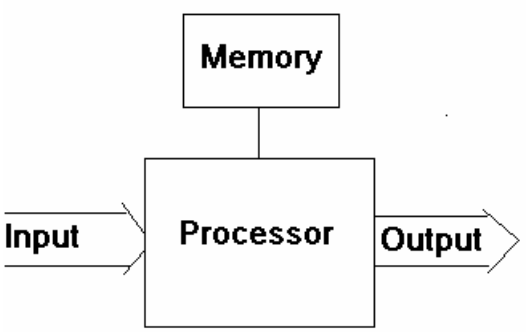

Fig. 1 Generic Embedded system

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

i. 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.
ii. Throughput: The system may need to handle a lot of data in a short period of time.
iii. Response: the system has to react to events quickly
iv. Memory: Hardware designer must make his best estimate of the memory requirement and must make provision for expansion.
v. Power consumption: Systems generally work on battery and design of both software and hardware must take care of power saving techniques.
vi. Number of units: the no. of units expected to be produced and sold will dictate the Trade-off between production cost and development cost
vii. Expected lifetime: Design decisions like selection of components to system development cost will depend on how long the system is expected to run.
viii. Program Installation: Installation of the software on to the embedded system needs special tools.
ix. Testability & Debugability: 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.

x. 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.

III. WHAT IS AN OPERATING SYSTEM?

The operating system organizes and controls the hardware and it is that piece of software that turns the collection of hardware blocks into a powerful computing tool. Broadly the tasks of the Operating system are:

Processor Management: The main tasks in processor management are ensuring that each process and application receives enough of the processor's time to function properly, using maximum processor cycles for real work as is possible and switch between processes in a multi-tasking environment.

Memory and Storage Management: The tasks include allotting enough memory required for each process to execute and efficiently use the different types of memory in the system.

Device Management: The operating system manages all hardware not on the computer's motherboard through driver programs. Drivers provide a way for applications to make use of hardware subsystems without having to know every detail of the hardware's operation. The driver's function is to be the translator between the electrical signals of the hardware subsystems and the high-level programming languages of the operating system and application programs. One reason that drivers are separate from the operating system is for upgradability of devices.

Providing Common Application Interface: Application program interfaces (APIs) let application programmers use functions of the computer and operating system without having to directly keep track of all the details in the CPU's operation. Once the programmer uses the APIs, the operating system, connected to drivers for the various hardware subsystems, deals with the changing details of the hardware.

Providing Common User Interface: A user interface (UI) brings a formal structure to the interaction between a user and the computer. Recently all developments in user interfaces have been in the area of the graphical user interface (GUI). Apple's Macintosh and Microsoft's Windows are the popular GUIs.

Four types of Operating systems, based on the kind of applications they support are:

i) *Single-user, single task* - This operating system is designed to manage the computer so that one user can effectively do one thing at a time. The Palm OS for Palm hand-held computers is a good example.

ii) *Single-user, multi-tasking* - This is the type of operating system most of us use on our desktop and laptop computers today. Windows 98 and the MacOS are examples of OS that let a single user have several programs in operation at the same time.

iii) *Multi-user* - A multi-user operating system allows many different users to take advantage of the computer's resources simultaneously. The operating system must make sure that the requirements of the various users are balanced, and that each of the programs they are using has sufficient and separate resources so that a problem with one user doesn't affect the other users. Unix is an example of multi-user operating system.

iv) *Real-time operating system (RTOS)* – The main task of a RTOS is to manage the resources of the computer such that a particular operation executes in precisely the same amount of time every time it occur. "In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not to move at all because the system is busy [8]."

IV. 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 [4]."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 [11].Real-time systems in which missing a deadline is catastrophic are called Hard Real time systems. If systems allow deadlines to be missed at times and still can be recovered they are called Soft Real-time systems.

V.WHY OPERATING SYSTEMS FOR REAL-TIME APPLICATIONS

Operating system is not a required component of any computer system. A simple microwave oven does not require an operating system. But as the complexity of applications expands beyond simple tasks the benefits of an operating system far outweighs the associated costs. Since embedded systems (PDAs, cell phones, VCRs, industrial robot control, or even the toaster) are becoming more complex hardware-wise with every generation, and more features are put into them in each iteration, applications they run require more and more to run on actual operating system code in order to meet the system response requirements and to keep the development time reasonable [2].

Real-time Operating System allows real-time applications to be designed and expanded easily. Functions can be added without major changes to the software. The use of an RTOS further simplifies the design process by splitting the application code into separate tasks. With a pre-emptive RTOS all time critical events are handled as quickly and efficiently as possible. An RTOS allows one to make better use of the system resources by providing valuable services such as semaphores, mailboxes, queues, time delays, time outs…etc.

The price we pay for these benefits is the extra cost of the RTOS, the royalties per unit, more RAM and ROM and around 2 to 4 % additional CPU overload.

VI. FEATURES YOU NEED IN A GOOD EMBEDDED RTOS

A closer look at some of the RTOS concepts is necessary to identify the features required for embedded real-time applications.

Operating System Architectures: *Monolithic Operating System*:

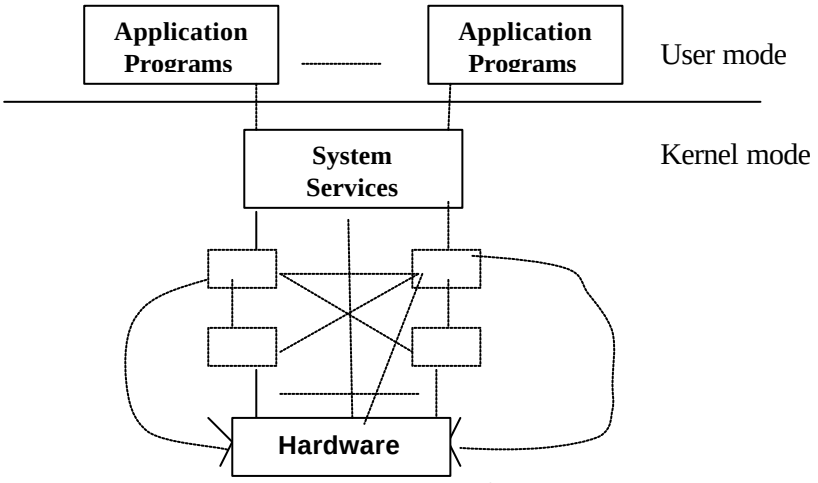

Fig. 2 Monolithic Operating System- from [4]

In this case the OS is just one piece of code composed of different modules. One module calls the other in one or more ways (Fig. 2). Here more the modules, more will be the interconnections and more complex the software becomes

Layered Operating System: this is a better approach compared to a monolithic OS. A system call goes directly to each individual layer. Here an application can access the BIOS or even the hardware (Fig. 3). In an RTOS even going directly to the hardware is desirable.

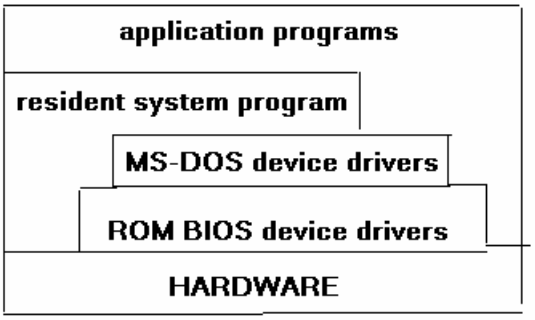

Fig. 3 Layered Operating System – from [4]

Client –server operating system: The basics of OS limited to a strict minimum (scheduler and synchronization primitive) and all other functionality is on another level implemented as system threads or tasks (Fig. 4). A lot of these server tasks are responsible for different functions or system calls. This structure allows making the OS scalable.

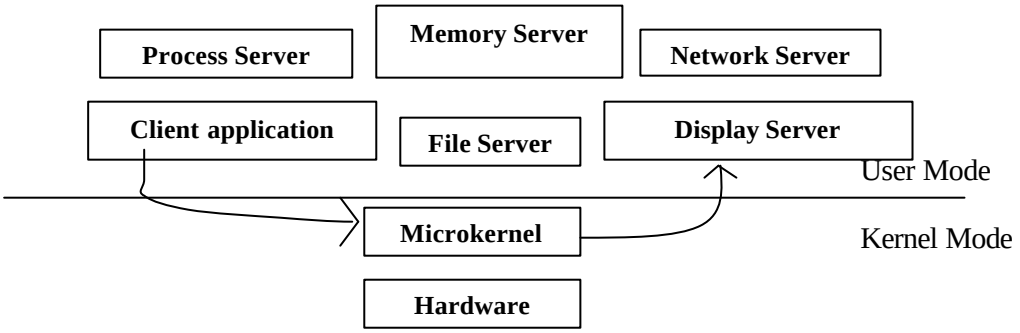

Fig. 4 Client-Server Operating System- from [4]

This feature is required in embedded systems where scalability is a prime requisite. This makes debugging easier and distribution over multiple processors simpler. One module crash does not necessarily crash the whole system resulting in more robustness. Implementing redundancy in OS is more achievable using this architecture. Dynamic loading and unloading of modules also becomes possible.

The major problem here is the overhead due to memory protection. Every time a service is requested the system has to switch from applications memory space to servers memory space. The switching time will increase when processes are protected from each other. On the other hand if the protection is removed, a bug in the application might affect the system processes compromising the system stability.

Process -Thread -Task model [4]

A multi-tasking concept is essential if one wants to develop a good real time application. Indeed an application has to be capable of responding in a predictable way to multiple simultaneous external events arriving in uncontrolled way. If only one processor is used we have to introduce pseudo parallelism called multitasking [4]. The application running on a system is subdivided into multiple tasks. In complex systems like UNIX, the system is

considered to be consisting of different processes. Here the context in each process is very heavy resulting in large switching times. This approach was changed due to:

i) Implementing multitasking approach, which is a requirement in complex distributed software, is too heavy in a process model.
ii) Bringing Real-Time and non-Real-Time world together, which the POSIX standards were aiming at, is not easy using a process concept.

Thus, the concept of thread was brought in, which is like a light-weight process. A thread inherits the context of the process but uses only a subset of it so that switching between threads can be done quickly [4]. Today in a Real-time environment a process is an application subdivided into tasks or threads.

Task & Task States:

Task is the basic building block of software written under an RTOS. Each task in RTOS is in one of the following three states (Fig. 5) [1].

1. Running: The microprocessor is executing the instruction that make up this task. In single processor systems only one task is running at a time.
2. Ready: means that some other task is running but this task has things that it could do if processor becomes available.
3. Blocked: this task has nothing to do right now even if microprocessor becomes available. Tasks get into this state because they are waiting for some external event. For example a task that handles data coming from a network will have nothing to do when there is no data.

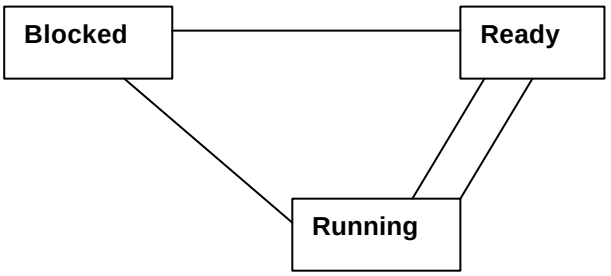

Fig. 5 Task States

Scheduler: The heart and soul of any operating system is its scheduler [2]. This piece of the operating system decides which of the ready tasks has the right to use the processor (go into running state) at a given time. Some of the common scheduling algorithms used in mainstream operating systems are first-in-first-out (FIFO), shortest job first and round robin. First-in-first-out [FIFO] scheduling is used in DOS, which is not a multitasking operating system. Here each task runs until it is finished and only after that next task is started. In shortest job first scheduler each time a running task completes or blocks itself, next task selected is one that will require the least amount of processor time to complete. Round robin is the only scheduling algorithm of the three in which the running task can be pre-empted, that is, interrupted while it is running. In this case, each task runs for some predetermined amount of time. After that interval has elapsed, the running task is preempted by the operating system and the next task in line gets its chance to run.

Unfortunately embedded operating systems cannot use any of these simple scheduling algorithms. 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 [2]. We also need that interrupt handling in case of different simultaneous interrupts should be handled in a pre-emptive 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.

Tasks and Data

Each task has its own private data [includes register values, Program Counter and stack]. All other data like global, static, initialized, un-initialised…etc is shared among the tasks. A situation like this can lead to many of shared data problems. If task1 calls a function ReadX for reading a shared data that is being modified by task2, there is a chance that data read by task1 is erroneous (Fig. 6).

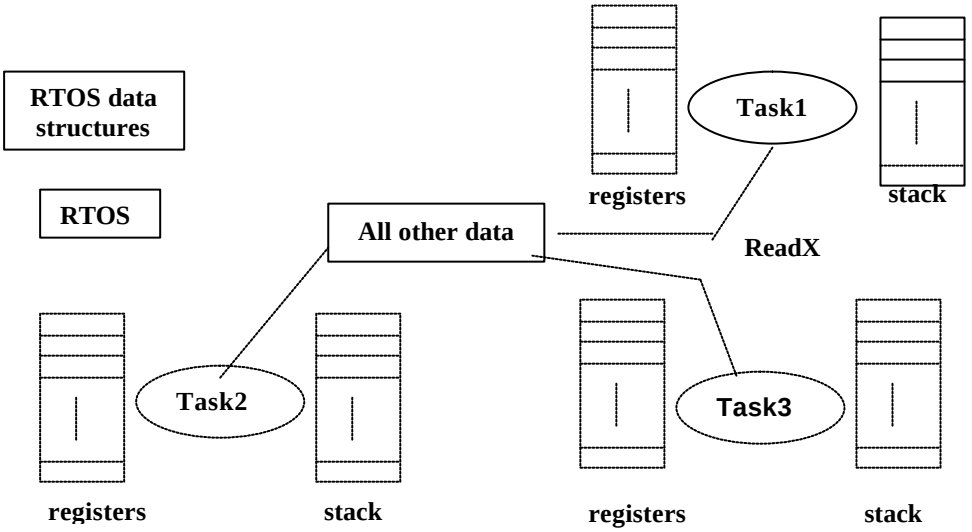

Fig. 6 Tasks and data- adapted from [1]

Task Synchronisation & Intertask Communication:

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

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. Here, in the function where task2 is modifying shared data X (Fig. 6), 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. Two types of semaphore namely binary semaphore and counting semaphore exist. A counting semaphore is used when more than one task uses the same resource like in the case of a buffer pool management. Using a different semaphore for highest priority tasks ensures better response. Multiple semaphores can be used to protect different shared resources.

Semaphore can also act as a signaling device for synchronisation. 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 [1]. When using Semaphores, one should ensure that it does not lead to Priority inversion or Deadly embrace [refer appendix A]. Some RTOS have a method called priority inheritance to tackle this problem.

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 [3].

Message Queues: 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 [3].

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 synchronise with occurrence of multiple events. A task can be synchronized when any of the events have occurred as in disjunctive synchronisation (logical OR) or may be synchronized when all the events have occurred as in conjunctive synchronisation (logical AND) [3]. 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 [1].

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 incase of a non-pre-emptive kernel
c) The highest priority task that is ready to run incase 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 [1].

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.

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 of dynamic memory allocation is preferred [4].

VII. 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.

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

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].

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 [10].

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 [10].

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 multi-

tasking, 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 [10].

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 [10]:

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.

VIII. CONCLUSIONS

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[11]. 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

i) Open Source
ii) Portable
iii) ROM able
iv) Scalable
v) Pre-emptive
vi) Multi-tasking
vii) Deterministic
viii) Efficient Memory Management
ix) Rich in Services
x) Good Interrupt Management
xi) 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 [7]. 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 [6].

REFERENCES

[1] David E Simon, *An Embedded Software Primer.* Reading, MA: Addison-Wesley, 1999.

[2] Michael Barr, *Programming Embedded systems in C and C++*. CA : O'Reilly & Associates,1999.

[3] Jean J Labrosse, *MicroC/OS-II The Real-Time Kernel*. 2nd ed. Gilroy, CA: CMP Books, 2002.

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

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

[6] John A. Stankovic, Krithi Ramamritham, "The Design of the Spring Kernel," *Proc. IEEE- Real-Time Systems Symposium*, Dec.1987, pp.146-57.

[7] Robert G Arnold, "A Modular Approach to Real-Time Super systems," *IEEE Transactions on Computers* 31(5): May 1982, pp.358-98.

[8] www.HowStuffWorks.com/operating-system.htm, How Operating Systems Work, HowStuffWorks, accessed November 14, 2002.

[9] http://www.cs.arizona.edu/people/bridges/oses.html, Operating Systems Project Information, Patrick Bridges, accessed November 12, 2002.

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

[11] Brian Santo, "Embedded Battle Royale," *IEEE Spectrum*, Dec. 2001, pp.36-41.

APPENDIX A

A.1. Priority Inversion:

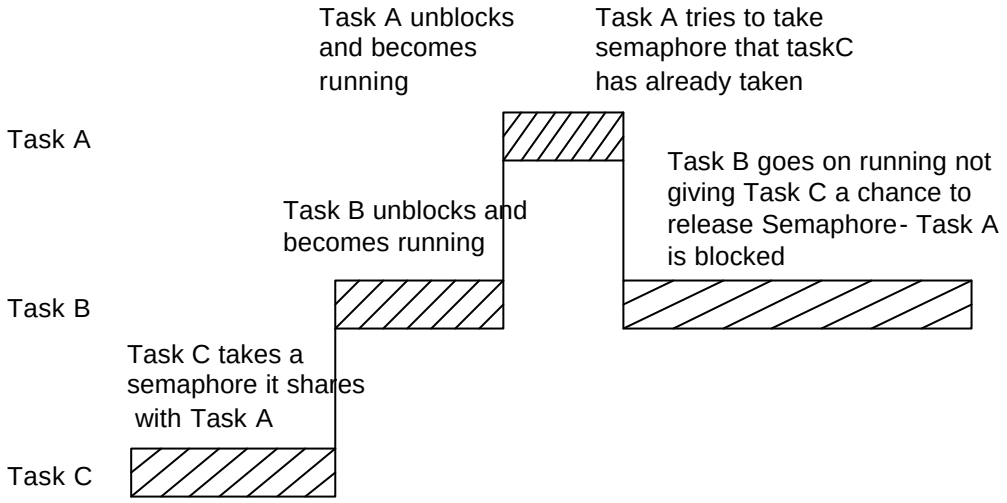

Fig. 7 Priority Inversion

Priority inversion occurs when a lower priority task shares a semaphore with a higher priority task. The figure illustrates how the higher priority task A is blocked and a lower priority task B is running not giving a chance for the lowest priority task C to release the semaphore required by task A.

A.2. Deadly Embrace: This is a situation in which two tasks are unknowingly waiting for resources held by the other. Let us look at this by an example.Task1 and Task2 operate on variables x and y after permission to use the by getting SemaphoreX and SemaphoreY. If Task1 calls TakeSemaphoreX but before it calls TakeSemaphoreY RTOS stops task1 and runs task2.Task2 gets SemaphoreY but when it tries to take SemaphoreX it blocks since SemaphoreX is with task1.RTOS will switch back to task1 which now calls TakeSemaphoreY but gets blocked. So now both task1 and task2 are blocked waiting for resource held by the other.

A.3. Other Real Time Operating Systems [9]

i) C EXECUTIVE : is an operating system kernel for embedded applications. It provides a small, efficient, real-time software environment for programs written in C. C EXECUTIVE is available as small as 5 KB in ROM space, on 8-, 16- and 32-bit CISC and RISC processors, providing the foundation for a common, portable software strategy. PSX provides a single-user, single-group, subset of POSIX.1, with up to 32,000 preconfigured processes. PSX adds a substantial subset of the POSIX.1 system calls to the basic C EXECUTIVE kernel. Using these calls allows applications to migrate from POSIX-conformant UNIX platforms to board-level systems, or vice versa.

ii) Chimera :Developed by the Advanced Manipulators Laboratory, at Carnegie Mellon University, the Chimera Real-Time Operating System, is a next generation multiprocessor real-time operating system (RTOS) designed especially to support the development of dynamically reconfigurable software for robotic and automation systems.

iii) Harmony(National Research Council of Canada)
Harmony is a multitasking, multiprocessing operating system for real-time control. It is developed at the National Research Council to serve the need for a flexible system for real-time control of robotics experiments and for other applications of embedded systems where predictable temporal performance is a requirement. Harmony is scalable, configurable and portable, both across different target computers, and across different development hosts.

iv) Helios (Perihelion Distributed Software)
Helios is a micro kernel operating system for embedded and multiprocessor systems. The operating system is modular in design and can scale from an embedded runtime executive up to a fully distributed operating system.

v) Lynx(Lynx Real-time Systems)
LynxOS is a proprietary UNIX-like real-time operating system. LynxOS looks and feels like UNIX from the user/programmer point of view. It was developed from the ground-up with high performance, deterministic hard real-time response in mind. The OS is in effect a complete re-implementation of UNIX from a real-time perspective.

vi) Maruti (University of Maryland)
Maruti is a time-based operating system research project at the University of Maryland. Maruti 3.0, the current version is an operating system suitable for field use by a wider range of users. The integration of the time-based, hard real-time technology with industry standards and more traditional event-based soft- and non-real-time systems is on.

vii) OS9 (Microware Systems Corporation)
OS-9 is a real-time, multi-user, multitasking operating system developed by Microware Systems Corporation. It is modular, allowing new devices to be added to the system simply by writing new device drivers, or if a similar device already exists, by simply creating a new device descriptor. All I/O devices can be treated as files, which unifies the I/O system. In addition, the kernel and all user programs are ROMable.

viii) OSE
OSE is a full-featured family of high quality, reliable and high performance real-time operating systems from Enea OSE Systems, Sweden. There is an OSE kernel for every need, from OSE Basic (for Z80, i8051 and others) up to OSE Delta (for M68k, PPC and others). OSE Delta is also the first RTOS to be certified according to the software quality standard IEC 1508. OSE Delta supports runtime configuration, runtime program loading, multi-CPU systems and TCP/IP.

ix) Roadrunner
In traditional operating systems, input/output (I/O) subsystems implement a push-pull environment that provides system calls to allow user applications to pull data from or push data to a device. An important set of applications make combined use of push-pull to implement simple streaming, i. e. data is moved from one device to another with no transformations. Using push-pull I/O to implement these applications does not provide maximum performance. Roadrunner is aiming at a kernel design optimized for simple streaming applications. The Roadrunner operating system is being developed specifically to implement multiple, concurrent, high-speed speed data streams with Quality-of-Service (QOS) parameters.

x) Real-Time Mach Project (Carnegie Mellon University)
Real-Time Mach is a research prototype real-time operating system intended for use as a platform for doing real-time systems research. The system is being developed by the ART Project in the School of Computer Science, Carnegie Mellon University.

xi) RTEMS (Redstone Military Arsenal)
RTEMS is a real-time operating system for embedded computer systems with the following features:
a) event-driven, priority-based, preemptive scheduling
b) homogeneous and heterogeneous multiprocessor systems support
c) optional rate monotonic scheduling
d) intertask communication and synchronization
e) responsive interrupt management
f) dynamic memory allocation

xii) RTMX O/S
RTMX is a commercial, BSD 4.4-derived, real-time system that offers POSIX 1003.4 real-time programming support with user tunability along with the standard UNIX functionality of BSD networking, X windows, and a full C development environment.

xiii) RTX
RTX is a very small, very fast real time executive. It utilizes signals and queuing as a basis for managing and scheduling tasks. Here it becomes very easy to support multiple processors, communication channels, and to synchronize processes. RTX is completely free, but it is not public-domain software. If you decide to use the software, you may receive an automatic license to do so even in commercial products, if you provide adequate, reasonable credit to its developer.

xiv) Spring Real-Time Project (University of Massachsetts, Amherst)
The Spring kernel has been designed and implemented to support/provide predictability, on-line dynamic guarantees, atomic guarantees, end-to-end scheduling and resource reservations. It utilizes a micro-kernel design for multiprocessor architectures and provides an interface to remote processes, support for distributed shared memory, and predictable low level communication. The kernel exists as a component of Spring's integrated environment. This environment extracts significant semantic information and this information is used at runtime to support flexibility.

xv) Sumo (Lancaster University)
Past few years members of the SUMO team have been designing and implementing a microkernel based system with facilities to support distributed real-time and multimedia applications and ODP based multimedia distributed application platforms. It is aiming at both communications and processing support for distributed real-time/ multimedia applications in end systems, and such applications require thread-to-thread real-time support according to user supplied quality of service (QoS) parameters.