Real-Time Operating Systems: The Next Stage in Embedded Systems

Samarth Shah

Sardar Patel Institute of Technology
Dept. of Electronics Engineering.

Abstract

A real-time operating system (RTOS) is an operating system (OS) intended to serve real time application requests. It must be able to process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter. Real-time operating systems have evolved over the years from being simple executives using cyclic scheduling to the current feature-rich operating environments. The standardization of POSIX 1003.1, ISO/IEC 9945-1 (real-time extensions to POSIX) has contributed significantly to this evolution, however, the specification leaves plenty of room for individual implementations to both interpret and specialize their RTOSs. Accordingly, there has been a proliferation of both commercial and free RTOSs, notably, the ITRON OS, the OSEK-VDX OS specification, commercial RTOSs like VxWorks, VRTX, LynxOS, OSE and QNX, and free RTOSs like RT-Linux (RTAI), and Windows CE. The goal of the work reported in this paper is to draw the real-time systems practitioner and researcher’s attention to these choices and bring out the similarities and differences among them.

1. Introduction

The primary role of an operating system (OS) is to manage resources so as to meet the demands of target applications. Traditional timesharing operating systems target application environments that demand fairness and high resource utilization. Real-time applications on the other hand demand timeliness and predictability, and the operating systems targeting these applications meet these demands by paying special attention to
a host of OS features like: (i) Multitasking (ii) Synchronization (iii) Interrupt and Event Handling (iv) Input/output (v) Inter-task Communication (vi) Timers and Clocks (vii) Memory Management.

The design of a real-time operating system (RTOS) is essentially a balance between providing a reasonably rich feature set for application development and deployment and, not sacrificing predictability and timeliness. In this paper, we attempt to demonstrate that the various RTOSs implementing these standards, differ in their implementation choices and strategies. This demonstration should allow a practitioner to choose the right RTOS for a particular application. The specific real-time operating systems that are considered in this paper are:

**LynxOS:** A UNIX-compatible, POSIX-conformant real-time operating system for embedded applications from Lynx Real-Time Systems Inc. It is scalable, fully re-entrant, preemptible, and ROMable.

**ITRON:** An open RTOS specification for embedded systems resulting from the TRON project. Participant companies that have implemented the specification include, Fujitsu, Hitachi, Mitsubishi, Miyazaki, Morson, Erg Co., Firmware Systems, NEC, Sony Corp., Three Ace Computer Corp., and Toshiba.

**Windows CE:** Microsoft’s embedded operating system for handheld PCs and small embedded processors. Though it’s current version (2.0) does not really qualify as an RTOS, both feature-wise and performance-wise, Microsoft promises to fix these shortcomings in version 3.0.

**OSE:** A commercial RTOS from Enea Data Systems that boasts to have bridged the gap between applications and the kernel by providing a rich set of features inside the kernel. Its message based architecture allows for efficient IPC and synchronization.

**RTAI:** It evolved from NMT RTLinux (New Mexico Institute of Technology’s Real-Time Linux), and takes a unique approach of running Linux as a task (lowest priority) that competes with other real-time tasks for the CPU.

**VRTX:** A highly reliable RTOS from Mentor Graphics that is the first to be certified under the US FAA’s stringent RTCA/DO-178B level A standard for mission-critical aerospace systems. It is based on a Nanokernel running on top of a Hardware Abstraction Layer to provide fast and predictable response.

**VxWorks:** The most popular (and complete) commercial RTOS (from Wind River Systems) in the embedded industry with ports for virtually all CPUs in the market.

**QNX:** A real-time, extensible POSIX compliant OS with a lean micro-kernel and a team of optional cooperating processes.

**OSEK-VDX:** The “Open Systems in Automotive Networks” RTOS specification that has been adopted by the following organizations in their embedded systems: Adam Opel AG, BMW AG, DaimlerChrysler AG, University of Karlsruhe - IIIT, PSA, Renault SA, Robert Bosch GmbH, Siemens AG, Volkswagen AG.

The above list is in no way exhaustive but is a reasonable subset of more than 50 commercial, academic (research-based) and free RTOSs currently available. Also, we
only picked one of several (Lineo, ecos, Lynx Bluecat etc.) new RTOS derivatives of the Linux operating system.

2. Important Terminology and Concepts

**Determinism:** An application (or critical piece of an application) that runs on a hard real-time operating system is referred to as deterministic if its timing can be guaranteed within a certain margin of error.

**Soft vs Hard Real-Time:** An OS that can absolutely guarantee a maximum time for the operations it performs is referred to as hard real-time. In contrast, an OS that can usually perform operations in a certain time is referred to as soft real-time.

**Jitter:** The amount of error in the timing of a task over subsequent iterations of a program or loop is referred to as jitter. Real-time operating systems are optimized to provide a low amount of jitter when programmed correctly; a task will take very close to the same amount of time to execute each time it is run.

![Jitter Diagram](image)

Figure Jitter is a measure of how much the execution time of a task differs over subsequent iterations. Real-time operating systems are optimized to minimize jitter.

3. RTOS Feature Comparison

Figure 1 gives a functional diagram of an RTOS with its various components. The following discussion delves into these components, and their desirable functionality.

The main components in the functional diagram are the hardware and the kernel of the RTOS running on top of it and servicing tasks and interrupts that comprise the real-time application. The OS has to provide, (i) task management (scheduling, dispatching, creation and termination of tasks etc.), (ii) synchronization (for resource sharing) (iii) interrupt handling (manipulate and monitor the interrupt descriptor table- IDT) to service hardware interrupts (iv) memory management (virtual memory and dynamic memory allocation) (v) programmable clocks and timers, and (vi) inter-task communication (sockets, pipes, FIFO, shared memory etc.). The following sub-
sections will describe the desirable functionality from each of these components and how the various RTOSs compare.

4. Multitasking

In computing, **multitasking** is a method where multiple tasks, also known as processes, are performed during the same period of time. The tasks share common processing resources, such as a CPU and main memory.

In the case of a computer with a single CPU, only one task is said to be *running* at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved.

It is essential for an RTOS to clearly distinguish between *schedulable* and *non-schedulable* entities. Schedulable entities are typically characterized by a context (a control block) and can make explicit requests for resources (CPU, memory, I/O), further they are scheduled by a *scheduler*. The scheduler itself and such entities like interrupt handlers, and most system calls are non-schedulable by nature. Often they are characterized by the fact that they can execute continuously, periodically or in response to events. Further, their use of the CPU is implicit. Multi-tasking involves fast switching between tasks allowing multiple tasks to be in a state of execution yet only one task is executing at any instant. A RTOS must provide (at a minimum) a multi-tasking mechanism that is priority-based and preemptive in nature. It should provide sufficient number of priority levels to be of practical use. For example Windows CE provides only 8 priority levels making it rather impractical for use in a majority of real-time scenarios. All of the reviewed RTOSs support a priority-based preemptive scheduling mechanism with OSE and OSEK providing cyclic and non-preemptive scheduling in addition (see Table 1). A typical number of priority levels
sufficient for most real-time applications is 32, however VRTX and VxWorks have 256 levels and LynxOS has 256 priority levels with another 256 levels each for the RR, Quantum and FCFS schedulers. RTAI allows for potential priority levels with Linux operating at priority level. OSEK being a specification (with several implementations), requires that any implementation provides at least 8 priority levels (ITRON does not specify any such limit). Lastly, Windows CE as its major shortcoming provides only 8 priority levels.

5. Synchronization
Synchronization is necessary for real-time tasks to share mutually exclusive resources (devices, memory areas, buffers etc.), which is also needed for implementing task-dependence (execute statement in task after task statement in task). Traditional solutions using semaphores (and related constructs like monitors, critical regions) can result in unbounded priority inversion. Priority inversion is said to occur when a higher priority task is temporarily forced to wait for a lower priority task. Such inversion of priority can go unbounded when medium priority tasks preempt the lower priority task (due to lack of resource conflicts).

Classical solutions to the problem are the simple priority inheritance protocol (PIP) and the complex priority ceiling protocol (PCP) (popularly implemented as the highest locker protocol (HLP)). Both protocols prevent unbounded priority inversion where PCP provides a better (lower) bound at a higher cost (implementation-wise). It
is desirable therefore that an RTOS provide at least PIP. Table 2 gives the synchronization mechanisms provided by the various RTOSs. While a majority of the RTOSs support

6. Interrupt and Event Handling

Handling interrupts is at the heart of an embedded system. By managing the interaction with external systems through effective use of interrupts can dramatically improve system efficiency and the use of processing resources. The actual process of determining a good handling method can be complicated, challenging and fun. Numerous actions are occurring simultaneously at a single point and these actions have to be handled fast and efficiently.

Embedded systems have to handle real world events such as the detection of a key being pressed, synchronization of video output, or handle the transmission and reception of data packets for a communication device. These events have to be handled in real time, which means an action has to take place within a particular time period to process the event. For instance, when a key is pressed on an embedded system it has to respond quickly enough so that the user can see a character appearing on the screen, without any noticeable delay. If an inordinate delay occurs the user will perceive the system as being non-responsive.

Software interrupts like Signals (POSIX) are also desirable. OSE is the exception to the rule that interrupt handlers be non schedulable entities (see Table 3).

In OSE, interrupts are associated with interrupt processes that are assigned high priority. OSEK has elaborate support for interrupts (its primary target being automotive environments with numerous remote sensors and actuators). LynxOS, QNX, VRTX and Vx-Works provide preemptible ISRs. RTAI allows very primitive interrupt handling which involves programming the 8259 interrupt controller through the RT Hardware Abstraction Layer. Lastly, though Windows CE supports writing ISRs(non-schedulable) the technical documentation by Microsoft describes the
preferred method to be the use interrupt service threads (IST) that run at priority level 0 (highest). Nesting of interrupts is allowed in all but RTAI and Windows CE.

7. Communication

Inter process communication (IPC) in RTOSs is primarily to exchange data on the same processor, however with an increasing number of real-time systems taking a more distributed (networked) form of operation some RTOSs allow process communication between processes resident on different processors. Popular forms of IPC include, shared memory, message queues, pipes, FIFOs (file in file out) and sockets. Desirable properties of IPC mechanisms in the context of an RTOS include, provision for non-blocking communication, bounded operation (r/w) latency and asynchronous communication. All the RTOSs that provide an IPC mechanism provide the above properties. While shared memory (physical address) is often an obvious mechanism for IPC it can be cumbersome and unsafe unless the RTOS provides an API for it, which is the case with all the studied RTOSs (see Table 4).

<table>
<thead>
<tr>
<th>Table 4. Inter-Process Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechanism</td>
</tr>
<tr>
<td>Shared Memory</td>
</tr>
<tr>
<td>Message Queue</td>
</tr>
<tr>
<td>Pipe</td>
</tr>
<tr>
<td>FIFO</td>
</tr>
<tr>
<td>Socket</td>
</tr>
</tbody>
</table>

Popular IPC mechanisms in traditional OSs like sockets while being supported have been totally rewritten to provide real-time response. For example, QNX provides a mechanism called Socket that is a less memory intensive version of a traditional socket. Though Linux itself has a rich IPC set, RTAI provides only FIFOs which are used to communicate between real-time tasks and also between real-time tasks and Linux tasks. VxWorks supports RPC (remote procedure calls) for distributed system implementation.

Timers and Clocks

All the RTOSs reviewed in this paper provide good support for timers, time-triggered tasks and clocks. All of them at low access of clocks at nanosecond resolutions when supported by the hardware.

Memory Management

Most older RTOSs did not see the need for supporting virtual memory, due to the lack of an MMU (memory management unit) on the processor and, due to the non-determinism introduced by it. However, most modern processors (with the exception of small embedded processors) come with a programmable MMU. Dynamic memory allocation allows programming flexibility but introduces the overhead of garbage
Samarth Shah

Therefore, calls to malloc can block due to unavailability of memory. Several of the RTOSs allow restricted use of dynamic memory allocation, for example (see Table 5) almost all of them disallow dynamic memory allocation calls in interrupt service routines.

<table>
<thead>
<tr>
<th>Table 5: Memory Management</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Providing support for virtual memory is often a very difficult choice to make if the processor has an MMU because, not supporting VM would amount to a waste of the MMU, while supporting it would have the downside of non-determinism. VxWorks has dealt with this issue by providing virtual memory as an optional add-on to the core RTOS.

8. Performance Comparison

Several researchers and practitioners have passionately argued that the oft-quoted performance metrics of context switch latency and interrupt latency are not the sole measures of merit of an RTOS. However, almost all RTOS publish these metrics as primary figures of merit.

The POSIX 1003.1 (optional Annex G) standard specification goes to great lengths by elaborating a list of performance metrics of interest in an RTOS, however, being an optional component leaves no incentive for RTOS manufacturers to adhere to it.

In addition to the context switch time and the interrupt latency it is desirable to know the maximum time taken by every system call. Such measures should be predictable and independent of the current state of the operating system. Context switch time is the delay incurred in saving the context of the current running process and restoring the context of the next process chosen by the dispatcher to run. Interrupt latency is the time elapsed between the occurrence of an interrupt and the execution of the first instruction of the corresponding interrupt handler.

VRTX is a clear choice (in terms of performance) in that it clearly outperforms OSE, LynxOS and VxWorks in its context switch time and comes close to the best in its interrupt latency. The comparison between QNX and Windows CE shows that Windows CE needs improvement which may be expected from the next version (3.0).
9. Conclusions and Future Work
In conclusion we have shown in this study that the world of RTOSs is less chaotic than it appears on the surface. Though only a few RTOSs were covered in this study they are a good representative set giving sufficient breadth to the observations made. The performance comparison reported gives some insight into the relative quality of these RTOSs.

The domain of real-time operating system has a very active area of research in recent years. The field has seen many RTOS’s being built with many different ideas, principles and paradigms. However, additional practical experience with such RTOS’s is desirable.

10. Acknowledgment
We would like to thank Thinklabs Inc. Their course on Real time Operating Systems made us aware of the application features of Real time operating systems. Without their invaluable support, this would have been difficult to achieve. Most of all, our deepest gratitude goes to our family.

References
[4] Course content of real time operating systems by Thinklabs.