Embedded OS Benchmarking

Technical Document

Amir Hossein Payberah

Table of Content

Index of Figures

1 Scope

This document is intended to describe the features of some embedded operating system (such as RTLinux, eCos and RTEMS) and compare them.

2 RTLinux

RT-Linux is an operating system in which a small real-time kernel coexists with the Posix-like Linux kernel. The intention is to make use of the sophisticated services and highly optimized average case behavior of a standard time-shared computer system while still permitting real-time functions to operate in a predictable and low-latency environment. At one time, real-time operating systems were primitive, simple executives that did little more than offer a library of routines. But real-time applications now routinely require access to TCP/IP, graphical display and windowing systems, file and data base systems, and other services that are neither primitive nor simple. One solution is to add these non-real-time services to the basic real-time kernel, as has been done for the venerable VxWorks and, in a different way, for the QNX microkernel. A second solution is to modify a standard kernel to make it completely pre-emptable. This is the approach taken by the developers of RT-IX (Modcomp). RT-Linux is based on a third path in which a simple real-time executive runs a non-real-time kernel as its lowest priority task, using a virtual machine layer to make the standard kernel fully pre-emptable.

In RT-Linux, all interrupts are initially handled by the Real-Time kernel and are passed to the Linux task only when there are no real-time tasks to run. To minimize changes in the Linux kernel, it is provided with an emulation of the interrupt control hardware. Thus, when Linux has "disabled" interrupts, the emulation software will queue interrupts that have been passed on by the Real-Time kernel. Real-time and Linux user tasks communicate through lock-free queues and shared memory in the current system. From the application programmers point of view, the queues look very much like standard UNIX character devices, accessed via POSIX read/write/open/ioctl system calls. Shared memory is currently accessed via the POSIX mmap calls. RT-Linux relies on Linux for booting, most device drivers, networking, file-systems, Linux process control, and for the loadable kernel modules which are used to make the real-time system extensible and easily modifiable. Real-time applications consist of real-time tasks that are incorporated in loadable kernel modules and Linux/UNIX processes that take care of data-logging, display, network access, and any other functions that are not constrained by worst case behavior.

In practice, the RT-Linux approach has proven to be very successful. Worst case interrupt latency on a 486/33Mhz PC measures well under 30 microseconds, close to the hardware limit. Many applications appear to benefit from a synergy between the real-time system and the average case optimized standard operating system. For example, data-acquisition applications are usually composed a simple polling or interrupt driven real-time task that pipes data through a queue to a Linux process that takes care of logging and display. In such cases, the I/O buffering and aggregation performed by Linux provides a high level of average case performance while the real-time task meets strict worst-case limited deadlines.

RT-Linux is both spartan and extensible in accord with two, somewhat contradictory design premises. The first design premise is that the truly time constrained components of a real-time application are not compatible with dynamic resource allocation, complex synchronization, or anything else that introduces either hard to bound delays or significant overhead. The most widely used configuration of RT-Linux offers primitive tasks with only statically allocated memory, no address space protection, a simple fixed priority scheduler with no protection against impossible schedules, hard interrupt disabling and shared memory as the only synchronization primitives between real-time tasks, and a limited range of operations on the FIFO queues connecting real-time tasks to Linux processes. The environment is not really as austere as all that, however, because the rich

collection of services provided by the non-real-time kernel are easily accessed by Linux user tasks. Non-real-time components of applications migrate to Linux. One area where we hope to be able to make particular use of this paradigm is in QOS, where it seems reasonable to factor applications into hard real-time components that collect or distribute time sensitive data, and Linux processes or threads that monitor data rates, negotiate for process time, and adjust algorithms.
The second design premise is that little is known about how real-time systems should be organized and the operating system should allow for great flexibility in such things as the characteristics of real-time tasks, communication, and synchronization. The kernel has been designed with replaceable modules wherever practical and the spartan environment described in the previous paragraph is easily "improved" (or "cluttered", depending on one's point of view). There are alternative scheduling modules, some contributed by the user community, to allow for EDF and rate-monotonic scheduling of tasks. There is a "semaphore module" and there is active development of a richer set of system services. Linux makes it possible for these services to be offered by loadable kernel modules so that the fundamental operation of the real-time kernel is run-time (although not real-time) reconfigurable. It is possible to develop a set of tasks under RT-Linux, test a system using a EDF schedule, unload the EDF scheduling module, load a rate monotonic scheduling module, and continue the test. It should eventually be possible to use a memory protected process model, to test different implementations of IPCs, and to otherwise tinker with the system until the right mix of services is found.

3 eCos

In this section the eCos OS features are shown.

3.1 Introduction

eCos is an open source, configurable, portable, and royalty-free embedded real-time operating system. eCos is provided as an open source runtime system supported by the GNU open source development tools.
One of the key technological innovations in eCos is the configuration system. The configuration system allows the application writer to impose their requirements on the run-time components, both in terms of their functionality and implementation, whereas traditionally the operating system has constrained the application's own implementation. Essentially, this enables eCos developers to create their own application-specific operating system and makes eCos suitable for a wide range of embedded uses. Configuration also ensures that the resource footprint of eCos is minimized as all unnecessary functionality and features are removed. The configuration system also presents eCos as a component architecture. This provides a standardized mechanism for component suppliers to extend the functionality of eCos and allows applications to be built from a wide set of optional configurable run-time components. Components can be provided from a variety of sources including: the standard eCos release; commercial third party developers or open source contributors.
eCos is designed to be portable to a wide range of target architectures and target platforms including 16, 32, and 64 bit architectures, MPUs, MCUs and DSPs. The eCos kernel, libraries and runtime components are layered on the Hardware Abstraction Layer (HAL), and thus will run on any target once the HAL and relevant device drivers have been ported to the target's processor architecture and board. Currently eCos supports a large range of different target architectures:

- ARM, Intel StrongARM and XScale
- Fujitsu FR-V
- Hitachi SH2/3/4
- Hitachi H8/300H

- Intel x86
- MIPS
- Matsushita AM3x
- Motorola PowerPC
- Motorola 68k/Coldfire
- NEC V850
- Sun SPARC

including many of the popular variants of these architectures and evaluation boards.
eCos has been designed to support applications with real-time requirements, providing features such as full preemptability, minimal interrupt latencies, and all the necessary synchronization primitives, scheduling policies, and interrupt handling mechanisms needed for these type of applications. eCos also provides all the functionality required for general embedded application support including device drivers, memory management, exception handling, C, math libraries, etc. In addition to runtime support, the eCos system includes all the tools necessary to develop embedded applications, including eCos software configuration and build tools, and GNU based compilers, assemblers, linkers, debuggers, and simulators.

3.2 Feature set

The key features of eCos are as follows:

- eCos is distributed under the GPL license with an exception which permits proprietary application code to be linked with eCos without itself being forced to be released under the GPL.
- Powerful GUI-based configuration system allowing both large and fine grained configuration of eCos. This allows the functionality of eCos to be customized to the exact requirements of the application.
- Full-featured, flexible, configurable, real time embedded kernel. The kernel provides thread scheduling, synchronization, timer, and communication primitives. It handles hardware resources such as interrupts, exceptions, memory and caches.
- The Hardware Abstraction Layer (HAL) hides the specific features of each supported CPU and platform, so that the kernel and other run-time components can be implemented in a portable fashion.
- Support for µITRON and POSIX Application Programmer Interfaces (APIs). It also includes a fully featured, thread-safe ISO standard C library and math library.
- Support for a wide variety of devices including many serial devices, ethernet controllers and FLASH memories. There is also support for PCMCIA, USB and PCI interconnects.
- A fully featured TCP/IP stack implementing IP, IPv6, ICMP, UDP and TCP over ethernet. Support for SNMP, HTTP, TFTP and FTP are also present.
- The RedBoot ROM monitor is an application that uses the eCos HAL for portability. It provides serial and ethernet based booting and debug services during development.
- Many components include test programs that validate the components behaviour. These can be used both to check that hardware is functioning correctly, and as examples of eCos usage.
- eCos documentation included this User Guide, the Reference Manual and the Components Writer's Guide. These are being continually updated as the system develops.

3.3 Architecture

As application complexity and project costs continue to rise, software portability and reuse are two prime concerns of both engineers and managers. RTOSes typically address these issues with a layered software architecture that abstracts the details of the target hardware from the application, enhancing both application port-

ability and reuse. eCos follows this paradigm with a well-defined interface between application and target-specific components.

The dashed line in Figure 1 divides the layers of software components. The first layer above the dashed line, the kernel, networking stack and file system are independent of the processing hardware or board product. These components interface with the upper compatibility and library layers to present a consistent platform for the application layer. Below the dashed line is the RedBoot ROM monitor, the HAL or Hardware Abstraction Layer and the device drivers. These components are written, configured and optimized for the specific target hardware and should be supplied by the hardware vendor.

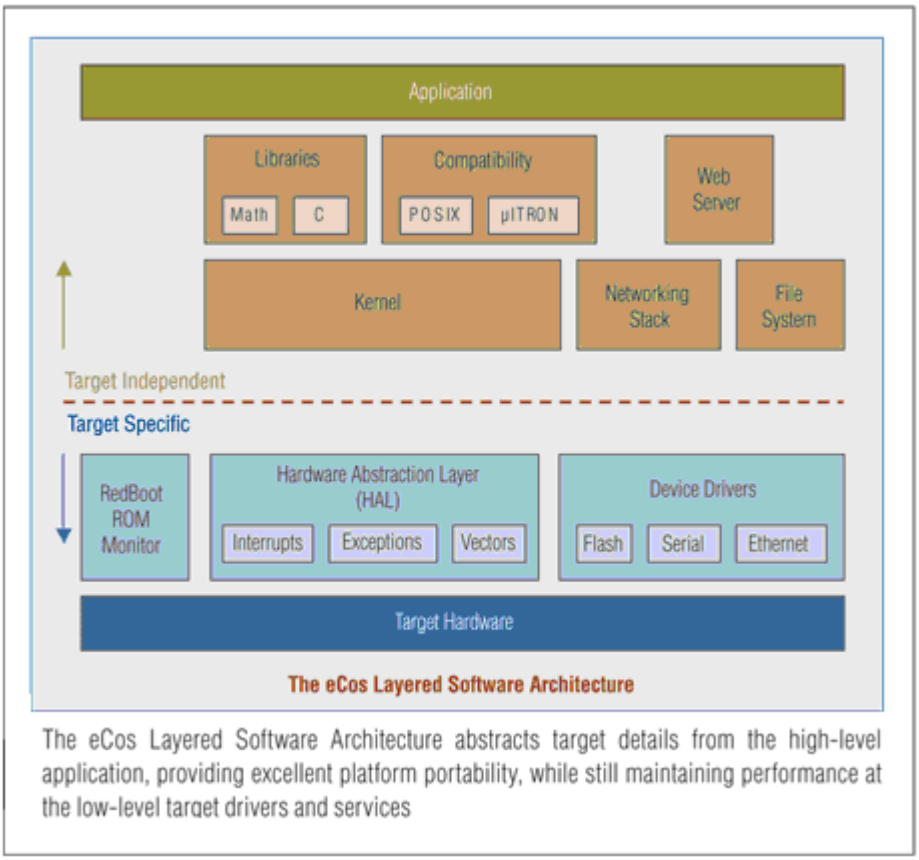

Figure 1- eCos Architecture

3.3.1 Portability and Performance

The HAL is a key component in eCos portability. It presents a consistent API to the upper OS layer, allowing the same application code to run on any platform that is supported with a HAL. This means that each hardware platform requires its own HAL to support the specific processor and peripheral set of the target.

As a complete package, the HAL often includes components for platform-specific resources, loading and booting, interrupt handling, context switch support, cache startup support, source level debugging, ROM monitor support, as well as other features. In keeping with the eCos goal of configurability, only HAL components that are actually required for a specific platform or application are built into the kernel. Selecting which components are built is simplified through the eCos Configuration Tool. This Graphical User Interface (GUI) displays details of each component and allows the user to choose which components will be built.

The HAL also provides some important board-specific, real-time facilities including an exception handler, an interrupt handler and virtual vectors. The exception handler allows an embedded system to recover from hard-

ware exceptions like overflow, a bad memory access or a divide-by-zero operation. An unrecoverable situation in a military system might prove disastrous. Virtual vectors support the ROM-based monitor, allowing application debugging over an Ethernet or serial communication channel.
Handling external interrupts is a fundamental requirement of most embedded real-time systems and how they are handled is crucial to overall performance. The HAL supports two types of interrupts. Interrupt Service Routines (ISR) are used for simple tasks that can be dispatched quickly. Deferred Service Routines (DSR) are used for complex tasks that can be implemented as soon as possible, but with interrupts re-enabled to maintain low latency for the ISRs. Having both types of interrupt handling available lets a user prioritize interrupts according to system needs and still guarantee low latency.

3.3.2 The Kernel

The eCos kernel was designed to satisfy four main objectives:

1) Low interrupt latency, the time it takes to respond to an interrupt and begin executing an ISR.
2) Low task switching latency, the time it takes from when a thread becomes available to when actual execution begins.
3) Small memory footprint, memory resources for both program and data are kept to a minimum by allowing all components to configure memory as needed.
4) Deterministic behavior, throughout all aspects of execution, the kernel's performance must be predictable and bounded to meet real-time application requirements.

Figure 2 lists these key metrics.

Four Key Kernel Metrics for RTOS Performance

Metric	Description	Requirement
Interrupt Latency	The time it takes to respond to an external interrupt and begin executing an ISR (Interrupt Service Routine).	Many embedded real-time systems must respond to external events through hardware interrupts and must service them appropriately. Low interrupt latency is crucial in many applications.
Task Switching Latency	The time it takes from when a thread becomes available to when actual execution begins.	Overall system performance is dependent on low task switching latency.
Memory Footprint	Memory resources needed for both program and data.	Embedded systems typically have limited memory resources due to size, power and cost constraints. The RTOS must maintain a small footprint to match the limited available memory.
Deterministic Behavior	The kernel's ability to consistently and predictably perform tasks.	In many real-time processing applications, data is introduced to the system and expected out of the system at a constant rate. Throughout all aspects of execution, the kernel's performance must be predictable and bounded to meet the real-time processing requirements.

Figure 2 - eCos Kernel Objective

eCos offers an interesting feature to further improve application performance: the option to build with or without an actual kernel. In simple applications that don't require scheduling or multi-tasking, eCos functions to set up and run the hardware can be built without the kernel, improving execution speed and reducing the memory footprint. In many traditional DSP applications this type of processing is common.
Moving in the other direction, eCos can be a full-featured OS with a complete set of kernel and core components including: scheduling and synchronization, interrupt and exception handling, counters, clocks, alarms, timers, POSIX and µITRON compatibility layers, ROM monitors, RAM and ROM file systems, PCI support, TCP/IP networking support, as well as features continuously being added and contributed by third parties. As mentioned earlier, the eCos Configuration Tools allow easy configuration and building of the kernel.

4 RTEMS

In this section RTEMS as real time OS is introduced.

4.1 Introduction

RTEMS (Real-Time Executive for Multiprocessor Systems) is a commercial grade real-time operating system designed for deeply embedded systems. It is a free open source solution that supports multi-processor systems. RTEMS is designed to support applications with the most stringent real-time requirements while being compatible with open standards. Development hosts include both MS-Windows and Unix (GNU/Linux, FreeBSD, Solaris, MacOS X, etc.) platforms. The RTEMS project is managed by OAR Corporation with invaluable development input from the vibrant and talented members of the RTEMS community

4.2 Feature set

RTEMS features are as follows:

- POSIX 1003.1b API including threads
- RTEID/ORKID based Classic API (similar to pSOS+)
- TCP/IP including BSD Sockets
- uITRON 3.0 API
- GNU Toolset Supports Multiple Language Standards
    - ISO/ANSI C
    - ISO/ANSI C++ including Standard Template Library
    - Ada95 with GNAT/RTEMS
- Multitasking capabilities
- Homogeneous and heterogeneous multiprocessor systems
- Event-driven, priority-based pre-emptive scheduling
- Optional rate-monotonic scheduling
- Intertask communication and synchronization
- Priority inheritance
- Responsive interrupt management
- Dynamic memory allocation
- High level of user configurability
- Portable to many target environment.
- High performance port of FreeBSD TCP/IP stack
- UDP, TCP
- ICMP, DHCP, RARP, BOOTP, PPPD
- Client Services
    - Domain Name Service (DNS) client
    - Trivial FTP (TFTP) client
    - Network Filesystem System (NFS) client
- Servers
    - FTP server (FTPD)
    - Web Server (HTTPD)
    - Telnet Server (Telnetd)
- Sun Remote Procedure Call (RPC)
- Sun eXternal Data Representation (XDR)
- CORBA

- In-memory filesystem (IMFS)
- mini-IMFS (reduced services and footprint)
- MS-DOS FAT32, FAT16, and FAT12
- TFTP client filesystem
- NFS client

4.3 Architecture

We show two side of RTEMS architecture here: Application architecture and internal architecture.

4.3.1 RTEMS Application Architecture

One important design goal of RTEMS was to provide a bridge between two critical layers of typical real-time systems. As shown in the following figure, RTEMS serves as a buffer between the project dependent application code and the target hardware. Most hardware dependencies for real-time applications can be localized to the low level device drivers.

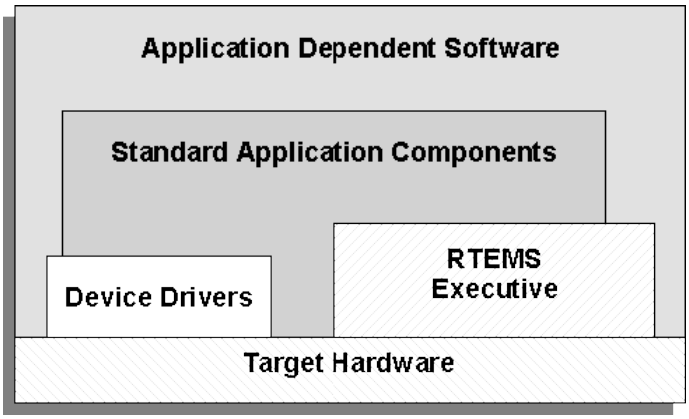

Figure 3 – RTEMS Application Architectre

The RTEMS I/O interface manager provides an efficient tool for incorporating these hardware dependencies into the system while simultaneously providing a general mechanism to the application code that accesses them. A well designed real-time system can benefit from this architecture by building a rich library of standard application components which can be used repeatedly in other real-time projects.

4.3.2 RTEMS Internal Architecture

RTEMS can be viewed as a set of layered components that work in harmony to provide a set of services to a real-time application system. The executive interface presented to the application is formed by grouping directives into logical sets called resource managers. Functions utilized by multiple managers such as scheduling, dispatching, and object management are provided in the executive core. The executive core depends on a small set of CPU dependent routines. Together these components provide a powerful run time environment that promotes the development of efficient real-time application systems. The following figure illustrates this organization:

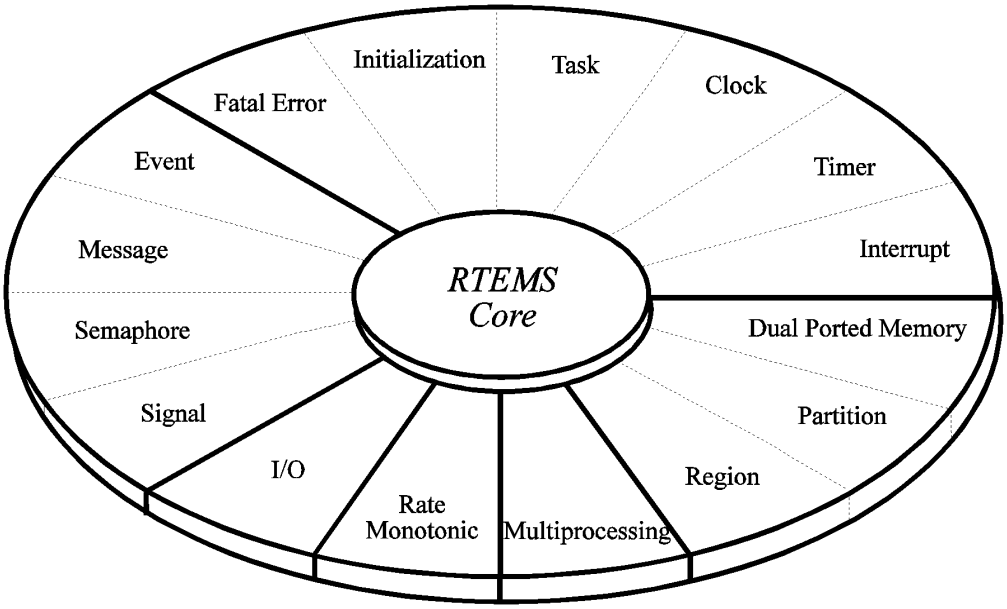

Figure 4 - RTEMS Internal Architecture

Subsequent chapters present a detailed description of the capabilities provided by each of the following RTEMS managers:

- initialization
- task
- interrupt
- clock
- timer
- semaphore
- message
- event
- signal
- partition
- region
- dual ported memory
- I/O
- fatal error
- rate monotonic
- user extensions
- multiprocessing

5 Benchmarking and Conclusion

This section gathers some comparing between these three OS.

The first compare is between RTLinux and RTEMS. This benchmarking shows Interrupt latency and context switching of these two OSes. The test was performed on the same hardware under the RTL, RTEMS and Vx-Works systems. 2'000'000 timer interrupts were generated at a rate of 4kHz and the maximal and average latencies were recorded. Measurements were made under both, idle and loaded conditions. Figure 5 shows the result:

	Interrupt Latency		Context Switching	
	max	avg±σ	max	avg±σ
		Idle System		
RTL	13.5	(1.7±0.2)	33.1	(8.7± 0.5)
RTEMS[1]	14.9	(1.3±0.1)	16.9	(2.3± 0.1)
RTEMS	15.1	(1.3±0.1)	16.4	(2.2± 0.1)
vxWorks	13.1	(2.0±0.2)	19.0	(3.1± 0.3)
		Loaded System		
RTL	196.8	(2.1±3.3)	193.9	(11.2± 4.5)
RTEMS[1]	19.2	(2.4±1.7)	213.0	(10.4±12.7)
RTEMS	20.5	(2.9±1.8)	51.3	(3.7± 2.0)
vxWorks	25.2	(2.9±1.5)	38.8	(9.5± 3.2)

[1] using pthreads

Figure 5 - RTLinux and RTEMS benchmark

eCos also has a kernel benchmark as follow. For example when eCos was run on the Pentek Model 4205 600 MHz, MPC7455 G4 PowerPC with 2 Mbytes of external L3 cache, thread switch time was recorded at 0.98 µsec. Switch times between 1 and 10 µsec are typically considered very good for commercially available RTOSes or for optimized versions of real-time Linux. Figure 6 shows execution times recorded for some of the other key scheduling and synchronization functions.

eCos Kernel Benchmarks	
Function	eCos Average Time (µsec)
Thread Switch	0.98
Put or Get Mailbox	0.47
Mailbox Put/Get	1.27
Post Semaphore	0.22
Wait Semaphore	0.26
Post/Wait Semaphore	1.33
Wait for Flag [AND]	0.35

Figure 6 - eCos Kernel Benchmarks

A side effect of a fully configurable system is that it is just about impossible to answer questions like "What is the memory footprint of the kernel?" or "What is the interrupt latency?" These figures depend in large part on

the configuration options selected by the user. For example if a simple application does not require the kernel at all, directly or indirectly, then the kernel package can be disabled completely and its memory footprint is 0 bytes.

These features show that eCos and RTEMS have better performance than Linux, but there are some important things that we should consider. The first one is we want to use the applications as a built in modules in kernel so the context switching and interrupt latency are not important. The other thing is security. In Linux kernel there is a Netfilter structure to implement security but this structure is not available in RTEMS kernel and eCos. We can use this structure to implement some features such as NAT, packet filtering and …. And the final note is we have a better knowledge in Linux than the other ones.

6 Reference

[1] http://ecos.sourcware.org
[2] http://www.cotsjournalonline.com/home/article.php?id=100164
[3] http://www.rtems.com
[4] T. Straumann, “OPEN SOURCE REAL TIME OPERATING SYSTEMS OVERVIEW”, Menlo Park, USA, 2001