National Institute of Standards and Technology

Introduction to Linux for Real-Time Control

Introductory Guidelines and Reference for Control Engineers and Managers

Prepared for:

Intelligent Systems Division
National Institute of Standards & Technology
100 Bureau Drive, Mail Stop 8230
Gaithersburg, MD 20899-8230

frederick.proctor@nist.gov

+1 301 975 3425

Prepared by:

Aeolean Inc.
54 Sarah Bishop Road
Ridgefield, CT 06877-1215

simon@aeolean.com

+1 203 438 3118

This report describes a rapidly-evolving aspect of computing, real-time Linux. Since the report cannot capture every possible detail and facet of the real-time Linux computing world, it would be appreciated if any changes, corrections, new material, notifications of outdated material, and any other observations are sent to:-

frederick.proctor@nist.gov

This material will become the basis of future releases of this document.

Contents

List of Figures

List of Tables

Chapter 1

Executive Summary

Real-time versions of Linux have matured to the point that they can be considered as viable candidates for real-time applications. Successful real-time applications have been demonstrated in the past few years by research groups in universities, government laboratories and industry, the real-time modifications are stable and proven, and commercial enterprises are offering training and support services.

Open source software ensures future maintainability and extensibility of software systems. Real-time versions of Linux offer important advantages to control engineers in providing an open source operating system that rivals the performance of the proprietary real time kernels.

At the National Institute of Standards and Technology (NIST) the Control Systems Group in the Intelligent Systems Division has had considerable success in using real-time modifications to Linux in real-time control applications for machine tools and robots. A review of that work and summaries of other successful applications provide the reader with an overview of recent applications.

This report is directed to control engineers and managers. It is intended to provide an overview of real-time operating systems and particularly of the real-time modifications to the Linux operating system in enough detail that the reader can make an informed decision whether to commit the resources to evaluate Linux as a real-time operating system.

1.1 History of Linux

Unix was developed at Bell Labs around 1970. Since it was clean, efficient, and provided economical access to a multi-user environment, it soon gained a significant following in the technical community. Unix and Unix-like operating systems came to dominate the server and workstation markets when they emerged in the 1980s.

Linus Torvalds developed Linux as a free, open source, Unix-like operating system as a student project at the University of Helsinki in 1991. For many years Linux was something of an underground cult phenomenon among hackers and university researchers. In recent years, however, Linux has emerged as a mainstream technology, particularly stimulated by IBM's adoption of Linux for their server business.

1.2 Standards

The Institute of Electrical and Electronic Engineers (IEEE) has led the development of a standard, based on the Unix operating system, that defines the requirements of a Portable Operating System Interface (POSIX). The IEEE Std 1003.1-2001 is a unification of two of the previous standardization efforts, POSIX.1 (System Interfaces and Headers), and POSIX.2 (Shell and Utilities). Standardization efforts in parallel with POSIX.1 developed POSIX.4, to cover real-time extensions to POSIX.1. POSIX.4 has since been superseded by IEEE Std 1003.1d-1999.

Linux was designed from the start to be a POSIX-compliant operating system. The current standard Linux complies with POSIX.1, the system call interface, and POSIX.1c, the threads extension. It is almost compliant with POSIX.2, the shell and utilities standard.

Some of the real-time Linux variants have been designed to be compliant with the 1003.1d real-time extensions standard.

This report uses POSIX as the basic reference standard for discussing operating systems, Linux, and real-time extensions of Linux.

This report is also relevant to standards efforts in the Department of Defense since DOD is always interested in extensibility and maintainability of their weapon systems. In particular, the specification of a Common Operating Environment in the architecture developed by the Weapon Systems Technical Architecture Working Group (WSTAWG) references POSIX system calls for hard real-time embedded weapons system applications. Real-time Linux directly addresses this emerging standard.

1.3 Real-Time Versions of Linux

Current developer releases of standard Linux can be used for soft real-time applications where the sample time is relatively long, say tens of milliseconds, and the application can tolerate missed timing schedules. An example is video processing, where an occasional flicker is often acceptable.

Hard real-time applications require rigid determinism and fast timing. Real-time variants of Linux suitable for such applications have appeared since 1997. Research groups at New Mexico Tech, Politecnico di Milano, and Kansas University have produced widely-distributed open-source versions that have been successfully deployed.

There are two basic approaches to modifying the standard Linux kernel, the core of the operating system, to provide this real-time response. Each of these is distributed as patches to the standard kernel.

- Micro-Kernel: This approach (New Mexico, Milano) is to insert a new, highly efficient code layer between the hardware and the standard kernel. The additional code layer, called a micro-kernel, takes care of all of the real-time functionality including interrupts, scheduling, and high-resolution timing. This micro-kernel runs the standard kernel as a background task. An alternative code layer, called a nano-kernel (ADEOS), is one that only handles interrupt dispatching. This nano-kernel approach can support multiple operating systems, one being real-time, including non-Linux operating systems.
- IEEE 1003.1d Kernel: The second approach (Kansas, TimeSys) is to implement the real-time extensions to POSIX.1 within the structure of the standard Linux kernel. These extensions add timer, scheduling, and preemption logic directly into a single, monolithic, kernel.

The standard kernel in recent developer releases shows latency (the time delay from an interrupt to the start of processing that interrupt) and jitter (variations of timing of periodic events) of the order of one millisecond. The real-time versions of Linux have latency and jitter of the order of a few microseconds on processors running at several hundred megahertz. Table 1.1 demonstrates that the performance is comparable to the commercial real time kernels.

	Application	**Latency/Jitter**
Standard O/S	Non Real Time	100 microseconds to 100 milliseconds
Standard Linux	Soft Real Time	1 millisecond
IEEE 1003.1d Linux	Hard Real Time	10 to 100 microseconds
Micro-Kernel Linux	Hard Real Time	1 to 10 microseconds
RTOS Kernels	Hard Real Time	1 to 10 microseconds

Table 1.1: Linux Performance Comparison

With this level of performance, and with the advantages of free open-source distribution to enable maintainability and extensibility, these real-time versions of Linux are finding increasing popularity in government and industry as well as in universities.

1.4 Applications

The National Institute of Standards and Technology (NIST) has used real-time versions of Linux for machine tool control, stepper motor control and robotics. Those applications, and a sampling of other non-NIST applications, will be summarized to give the reader a feel for the areas where real-time Linux has been used. The Real Time Linux Foundation web site provides an expanded list of real-time Linux applications.

1.5 Sources of Real-Time Linux and Sources of Additional Information

Tables summarizing the characteristics of both open-source and commercial distribution of real-time versions of Linux are provided. Pointers are provided to further sources of information throughout the report and in the appendices. Links are embedded in the electronic version of the report that will connect the reader directly to those sources. These links are also listed in the bibliography as text for the readers of hard copy versions.

Real-time modifications of Linux are achieving growing acceptance in applications of control and automation, data acquisition, and audio and video processing. This report provides an introduction to that world and directs the reader to sources of further information.

Chapter 2

Introduction

Everyone that uses computers has undoubtedly heard of Linux, the free, open-source, Unix-like operating system introduced by Linus Torvalds in 1991. It has been continually refined and enhanced by Torvalds along with a world-wide group of contributors since then. Particularly since IBM adopted Linux for their server products, Linux has become a mainstream technology, gaining something like 25% of the market share for servers.

While organizations may adopt standard Linux for server applications, particularly with handholding and support from an IBM, relatively few have used Linux as a development platform for real-time applications. This is because real-time extensions of Linux have only appeared since 1997 and using those real-time extensions require modifying the standard Linux kernel and hence requires intimate familiarity with the Linux world.

Real-time versions of Linux have matured to the point that they can now be considered as viable candidates for real-time applications. Significant successes in using Linux for real-time applications have been demonstrated in the past few years by research groups in universities, industry, and government laboratories. The real-time modifications are stable and proven, and commercial enterprises are offering training and support services.

The Control Systems Group of the Intelligent Systems Division of NIST has had great success in using the real-time modifications to Linux for real-time control applications in robotics and machine tool control. Those applications, and a sampling of other non-NIST applications will be summarized to provide the reader a feel for the areas where real-time Linux has been used.

This report is directed to control engineers and managers. It is intended to provide an overview of real-time operating systems and particularly of the real-time modifications to the Linux operating system. Enough detail is provided so that the reader can make an informed decision whether to commit the resources to evaluate and consider adopting Linux as a real-time operating system.

Pointers are provided to further sources of information throughout the report and in more detail in the appendices. Links are embedded in the electronic version of the report that will connect the reader directly to those sources. These links are also listed in the bibliography as text for the readers of hard copy versions.

2.1 Trademarks and Service Marks

All and any trademarks and service marks that are reproduced in this report are the legal property of the registered trademark or service mark owner.

Absolutely no representation is made that any trademark is the property of the United States Government, nor the National Institute of Standards and Technology, nor Aeolean Inc.

2.2 NIST Disclaimer

Certain companies and products are identified in this report in order to accurately explain the concepts. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply a judgment that the products identified are necessarily the best available for the purpose. There is no representation that the report presents an exhaustive list of all companies and products that are available to the user of real-time operating systems. Omission of any company or product does not imply a judgment that the product is unsuited in any way whatsoever for the purposes described in this report.

Chapter 3

Operating Systems and Real-Time Kernels

An operating system is the interface between a user's program and the underlying computer hardware. It also manages the execution of user programs such that multiple programs can run simultaneously and access the same hardware. Everyone who uses a computer encounters an operating system, whether it is Windows, Mac, Linux, DOS, or Unix.

This chapter describes the architecture and functionality of an operating system and then summarizes the requirements of a real-time operating system. Discussion of standards and of specific details will be limited to Unix-like operating systems, since Linux is a Unix-like OS. The chapter concludes with an overview of real time kernels.

3.1 Architecture of an Operating System

An operating system, or more specifically the core or kernel of the operating system, is always resident in memory and provides the interfaces between user programs and the computer hardware. This is shown schematically in Figure 3.1. The name **kernel** follows from the analogy of a nut, where the kernel is the very heart of the nut and, in the computing domain, the kernel is the very heart of the operating system. Continuing the analogy, protective layers around the kernel that provide user authorization and interaction are called shells .

The physical memory in the computer is partitioned into user space and kernel space, with the kernel space reserved for the kernel code. The kernel of a multi-tasking operating system can manage multiple user programs running simultaneously in user space so that each program thinks it has complete use of all of the hardware resources of the computer and, other than for intentional messages sent between programs, each program thinks that it has it's own memory space and is the only program running.

Communication between user-space programs and the kernel code is achieved through system calls to the kernel code. These system calls typically are to access shared physical resources such as disk drives, serial/parallel ports, network interfaces, keyboards, mice, display screens,

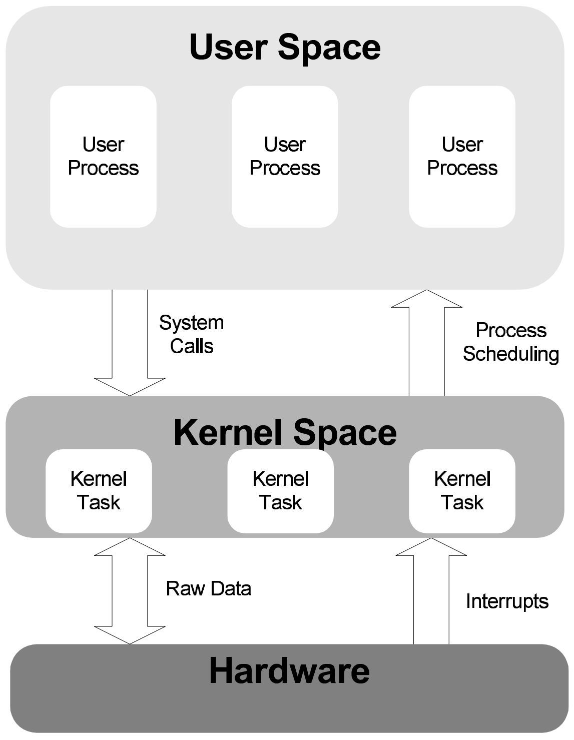

Figure 3.1: Operating System Architecture

and audio and video devices. One unifying aspect of Linux/Unix systems is that all the physical resources appear to the user programs as files and are controlled with the same system calls such as `open(), close(), read(), write()`.

All of the input/output activity is controlled by the kernel code so that the user-space programs do not have to be concerned with the details of sharing common physical resources. Device-specific drivers in the kernel manage those details. An operating system is thus tailored to run on specific computer hardware and it isolates user programs from the specifics of the hardware, allowing for portability of user-space application code.

The operating system is generally larger than just the kernel component resident in physical memory. Some code that extends the kernel functionality is also resident in physical memory in user space and includes functionality like kernel logging, process initiation, timed process execution, and remote procedure call support.

Some operating systems support a mechanism of loadable modules to extend the kernel functionality. Linux is one example of such an operating system where modules can be loaded and unloaded into the kernel space by user-space commands. One major benefit is that developers of kernel drivers do not have to repeatedly reboot the computer and reload the complete kernel to test changes to the kernel functionality. A minor benefit is to reduce the amount of memory used by the kernel by loading only the modules that are required to interface to the currently-configured hardware.

Operating systems usually provide a wide variety of common services beyond managing execution of the user programs and interfacing to the hardware. Function libraries and utility programs such as file management programs, browsers, editors, compilers, debuggers, e-mail user agents, diagnostics, and disk management programs are examples. These programs remain resident on disk until needed.

The architecture of an operating system is thus a core or kernel that remains in memory at all times, a set of processes in user-space that support the kernel, plus various modules and utility programs that remain stored on disk until needed. The kernel manages simultaneous execution of multiple user programs and isolates user programs from the details of managing the specific hardware of the computer.

3.2 Functionality of a general purpose operating system

The main functions of the kernel of the operating system are memory management, process scheduling, interfacing to the hardware, file management, and communication with external devices and networks. Each of these will be discussed briefly.

3.2.1 Memory Management

Memory management is the mechanism provided by the operating system to allocate memory requested by a process and deallocating memory when a process terminates. Another requirement is to ensure that memory, previously allocated and now no longer required by the process, is released and made available for allocation to other processes. This last requirement is known as garbage collection.

There are two fundamentally different approaches to memory addressing. One is a flat memory model using address registers that are wide enough to address any word in the largest-conceivable memory space such as in the Motorola Power PC Microprocessors. The second is a segmented address model that uses two address registers, one that holds addresses for a sub-set of the memory space, a second one that selects which of the sub-sets to address, such as in the Intel 80x86 microprocessors.

The memory space available to any user program is a combination of cache memory, physical memory and virtual memory. Cache memory is an integral part of the CPU silicon and stores a small subset of instructions and data from the currently executing program. Physical memory is semiconductor Random Access Memory (RAM) and stores instructions and data for the currently executing program. Virtual memory is a percentage of the available hard disk storage and typically stores instructions and data for programs that do not fit into the physical memory space. Cache memory can be accessed 10 to 100 times faster than RAM which in turn is 100 to 1000 times faster than disk access.

The kernel code utilizes the CPU and hardware memory-management features to transparently move the instructions and data between cache, physical, and virtual memory, trying to anticipate what data and instructions will be accessed and to have those available in cache or at least in RAM when needed.

3.2.2 Memory Protection

A basic requirement for a general-purpose operating system is that each user process operates in it's own, isolated, address space. No user process should be able to write into the memory allocated to another user process. This user-space memory protection is enforced by the CPU's Memory Management Unit (MMU). For most embedded applications, see Section 3.4.1, the CPU does not have a MMU and consequently, the user-space processes must be carefully managed so that memory controlled by other processes is not corrupted.

For the special case of the kernel code itself, all tasks within the kernel share the same address space. The underlying assumption is that the experienced programmers writing the kernel code exercise great care to not write into memory that they are not responsible for.

These memory-protection rules also apply to real-time operating systems. If the real-time operating system can support user-space real-time processes, and the CPU does have a MMU, then the processes execute in their own, protected memory space.

3.2.3 Scheduling and Task Management

The kernel code is responsible for scheduling the execution of the user-space programs. The scheduling is done in such a way that users are unaware that the program code will be stopped from executing and placed on hold for many milliseconds to allow other user program code to execute. The scheduling may be achieved by giving each user program a fixed percentage of CPU time, or user programs may be prioritized and higher priority programs given a higher percentage of CPU time.

When a higher priority program is executed and the lower priority program is paused, then the higher priority program is said to have **preempted** the lower priority one. Fast and deterministic preemption is the key to effective real-time operating systems, as will be discussed in later sections of this report.

The POSIX standard, described in Section 3.3, defines a **thread** to be a flow of control within a process. All threads share the same memory space within the process.

A **process** is defined by the POSIX standard to be a single memory space with one or more threads executing in that space. Each and every process running under the operating system has it's own memory space. The accepted understanding is that processes are executing in user space rather than kernel space.

There is no commonly agreed definition of what constitutes a task. In this document, the word **task** shall be used to describe the executable code components that are running in kernel space.

A fundamental concept of scheduling is that of dividing execution time into fixed slices. This time slicing can be implemented in user space by a process executing a `sleep()` system call when the process has finished the cyclic component of it's execution. The kernel returns control to the user process on, or after, the expiration of the time requested in the `sleep()` call. By returning control some time after the expiration of the `sleep()` call, an operating system can still claim POSIX compliance yet introduce jitter into the execution of repetitive tasks.

3.2.4 Hardware Interfaces

Operating systems provide the low level programs, called device drivers , that control the computer hardware. Drivers can be written to be called directly from within the kernel code or they can operate independently and interrupt the kernel code when something happens that needs attention. The first mode is called polling and the second is called interrupt driven.

Most operating systems work with interrupt-driven drivers. Data originating from sources outside the computer system can arrive at any time. The interface device (keyboard, mouse, serial port, parallel port, network interface) indicates the arrival of the data by signaling the CPU at the hardware level. This signaling is through the interrupt pins provided on the CPU. Operating systems generally acknowledge an interrupt by stopping the CPU from executing the current process and running the interrupt-service routine (ISR) that reads the incoming data from the interface device. When the ISR finishes, control is returned to the original, interrupted, process. These interrupt processes are an integral part of the kernel and are designed to perform a minimal amount of processing, typically just reading the incoming data and writing it to kernel space storage, before returning.

Interrupts can be assigned priorities so that processes that need rapid response, such as disk reads, can interrupt lower priority processes such as serial port reads.

The time between an interface device requesting service, by raising an interrupt flag, and the time that the CPU starts processing the interrupt process is called the interrupt latency.

Another significant role for the kernel code is to synchronize user-space and kernel-space access to the shared hardware resources. The traditional method is to enable some kind of software lock in the kernel when one user-space program is using a physical resource, and to disable that lock when the user-space program is finished with the resource. All other user-space programs are prohibited (by the kernel code) from accessing the physical resource when it is locked. One specific type of lock is a spin lock, where the program sits in a tight loop until the necessary hardware device is available. This maximizes the response for the user program.

3.2.5 File Management

One of the significant functions of the operating systems is to provide persistent storage of information that is created in memory by the execution of a process. This information is typically stored as files on the file system supported by the operating system.

The operating system keeps track of which cylinders, tracks, and sectors on a disk are used to store each and every block of information that comprises a file. The user is unaware that the file is stored in non-contiguous groups of sectors on the disk drive. Separate directories, or folders, can be created to hold related files. Files are managed hierarchically, such that folders can contain other folders which, in turn, contain folders and files.

File systems can exist on many different types of media, including hard disks, CD ROM, floppy disks, flash memory, even read/write memory.

Real-time applications may have file management requirements. One such example is data acquisition systems. In such applications, the file management tasks would be run at a lower priority while the real-time applications would run at a higher priority, preempting the file management tasks as required.

3.2.6 Communication

Data communication, or message passing, between processes in Linux/Unix is commonly achieved with the Berkeley Software Distribution (BSD) socket system calls. These were introduced in release 4.2 of BSD Unix in 1983. The idea of a **socket** is to "plug-in" to the other process and send the message, with the operating system taking care of delivering it. The socket layer supports communication between processes that are running on the same machine or even different machines. Sockets can be opened using a number of different protocols including the Unix socket for intra-machine communication and TCP sockets for inter-machine communication. One disadvantage of the original socket implementation is that information is transmitted as plain text between a pair of sockets. Netscape addressed this issue in 1994 by proposing the Secure Socket Layer which provides for encryption of the information and identification of both machines involved in the communication. The SSL protocol, now known as the Transport Layer Security protocol, is at the heart of financial transactions on the Internet.

Sockets allow programmers to specify communications at the application level. The operating system then notes where the recipient process is and implements lower level communication protocols to deliver the message. Those lower level mechanisms can be distinguished as external or internal communications.

3.2.6.1 External Communication

The operating system provides a set of primitives that enables the computer to communicate with other machines using standard protocols such as Internet Protocol (IP), Novell IPX, AppleTalk, and Microsoft SMB. These higher-level protocols are, in turn, built upon Ethernet (IEEE 802.3) or other Media Access Protocols. The operating-system primitives include device drivers for the communication hardware, packet routing and filtering functions, and implementation of the protocol stacks such as Transmission Control Protocol/Internet Protocol (TCP/IP).

3.2.6.2 Internal Communication

Internal communication is the class of communication between two processes, between two tasks, or between a process and a task when both are running on the same computer. This is generally referred to as inter-process communication.

Inter-process communication is supported by the operating system through primitives such as shared memory, binary signals, and pipes. Since two or more processes could be accessing the shared memory segment at the same time, there is a need to indicate when one process is writing to the segment so that the other process(es) wait until the write is complete before performing their own read/write action. This indication is achieved by semaphores whose functionality includes the action of restarting a process that has been placed on hold waiting for the shared memory.

Pipes are an alternative to shared-memory communication and use different system calls to the shared-memory interface. Shared-memory and pipes allow finite-size messages to be passed between processes while binary signals convey only one bit of information.

Signals are another communication mechanism between processes. Only one bit of information is involved, but the operating system may support about 100 such signals with each conveying their own explicit meaning to the receiving process.

All these inter-process communication mechanisms involve a timing overhead. Signals are faster than pipes which are faster, in turn, than shared memory; sockets are the slowest of all.

3.3 Standards

The Institute of Electrical and Electronic Engineers (IEEE) has led the development of a standard, based on the UNIX operating system, that defines the requirements of a Portable Operating System Interface (POSIX). The IEEE Std 1003.1-2001 [46, 94] is a unification of two of the previous standardization efforts, POSIX.1 (System Interfaces and Headers), and POSIX.2 (Shell and Utilities).

Standardization efforts in parallel with POSIX.1 developed IEEE Std 1003.1b-1993 to cover real-time extensions and IEEE Std 1003.1c-1995/96 to detail the thread extensions. The original work on real-time extensions was known as POSIX.4 [38]. Since 1993 there has been further work on the real-time extensions standard and, consequently, IEEE 1003.1b has been superseded by IEEE Std 1003.1d-1999 [46].

A group of four profiles for real-time POSIX characterizing single-task and multi-task operating systems with and without file systems was finalized in 1998. This group of standards is known as 1003.13-1998 PSE51 to PSE54 (Posix Standard Environment).

The International Organization for Standardization (ISO) has adopted the POSIX.1 standard as ISO 9945-1 and the POSIX.2 standard as ISO 9945-2 [65] and has superseded IEEE 1003.13 with ISO 15287-2:2000.

POSIX Standard	**IEEE Standard**	**Description**
POSIX.1	1003.1-2001	System Interfaces and Headers
POSIX.2	1003.1-2001	Shell and Utilities
POSIX.4	1003.1b-1993	Real-time extensions
	1003.1d-1999	Supersedes 1003.1b-1993
POSIX.4a	1003.1c-1994	Threads Extensions

Table 3.1: POSIX/IEEE Standards

Linux was designed from the start to be a POSIX-compliant operating system. The current standard Linux complies with POSIX.1, the system call interface, and POSIX.1c the threads extension. It is almost compliant with POSIX.2, the shell and utilities standard.

This report uses POSIX as the basic reference standard for discussing operating systems, Linux, and real-time extensions of Linux.

3.3.1 Department of Defense Standards

The Department of Defense and the Army in particular are working toward standards for interoperability of weapons systems software. The Weapon Systems Technical Architecture Working Group (WSTAWG) has specified a Joint Technical Architecture-Army (JTA-A) that features a Weapons Systems Common Operating Environment (COE). The COE calls for open systems with standard interfaces, and cites POSIX specifically. Rather that trying to specify or standardize at the operating system level, the COE calls for standard Application Programming Interfaces using standard system calls such as those specified in the POSIX standards.

WSTAWG is directly targeted at hard real-time and embedded systems developers. The open source and POSIX conformance features of Linux are directly relevant to the WSTAWG philosophy. Further, the open source nature of Linux matches the Government's interests in acquiring full data rights in order to maximize maintainability and extensibility.

3.4 Required Functionality for Real-Time

Real-Time Operating Systems (RTOS) are those able to provide a required level of service in a bounded response time [1]. They can deliver a response in a time less than a designated timing interval. The timing interval may be long in computing terms i.e. orders of seconds, or it may be short i.e. orders of microseconds. For example, a real-time process control system for a chemical or food plant may only sample a sensor and calculate a control command once a second. On the other hand, for smooth response, a stepper motor must be serviced every few microseconds. A so-called hard real-time system is one that misses no timing deadlines, a soft real-time system can tolerate missing some timing deadlines.

3.4.1 Embedded vs Real-Time Operating Systems

Embedded programs are those that are a fixed and integral part of a device. For example, a hand-held computer, a telephone answering system, and the control computer for the engine of a car all have fixed programs that start up whenever power is turned on. These are called embedded applications. Depending on the required response time, the operating systems for embedded applications may or may not be considered real-time operating systems. Servicing human interactions does not in general require real-time performance, but controlling a machine tool, scientific experiment, or weapon system does.

3.4.2 Measures of Performance for Real-Time Operating Systems

The most vital characteristic of a real-time operating system is how responsive the operating system is in servicing internal and external events. These events include external hardware interrupts, internal software signals, and internal timer interrupts. One measure of responsiveness is **latency**, the time between the occurrence of an event and the execution of the first instruction in the interrupt code. A second measure is **jitter**, the variation in the period of nominally constant-period events. To be able to offer low latency and low jitter, the operating system must ensure that any kernel task will be preempted by the real-time task. Section 4.3 details these measures and Section 4.4.1 provides a summary of latency and jitter for standard and real-time Linux.

3.4.3 POSIX Extensions for Real-Time Applications

The POSIX real-time extensions [1] provide an insight into the additional functionality that is required for a real-time operating system. These real time extensions add message queues (for communication between tasks), shared memory, counting semaphores (needed to synchronize accesses to shared memory), priority-based execution scheduling, real-time signal extensions, and higher resolution timers. The timers can generate time intervals with at least one microsecond resolution.

3.5 Overview of Available Real-Time O/S

A 1999 market survey by Real-Time Magazine [122] predicted a market for real-time operating systems in 2001 that would have VxWorks, Windows CE .NET, QNX, and pSOSystem as the top four RTOS. These four operating systems are discussed here in some detail. Latency numbers provided in the corresponding performance sections are derived from a report generated by the Dedicated Systems Encyclopedia[33]. A more exhaustive list of real-time operating systems is given in Table A.1.

3.5.1 VxWorks

Wind River's **VxWorks** [119] has been widely used in robotics, process control, avionics, CNC equipment, and flight simulation control applications. VxWorks has also been deployed in applications in the telecommunications, consumer electronics, data networking, and medical system areas.

The *wind* micro-kernel core of VxWorks provides multitasking, interrupt support and both preemptive and round robin scheduling. The intertask communications mechanisms supported by VxWorks includes shared memory, message queues, semaphores (binary, counting, mutual exclusion with priority inheritance), events and pipes, sockets and remote procedure calls, and signals. VxWorks is POSIX compliant, implements the POSIX pThreads and adheres to the POSIX Real Time Extension, including asynchronous I/O, counting semaphores, message queues, signals, memory management (page locking), and scheduling control.

The Tornado embedded development platform, of which VxWorks is the run-time component, provides a full suite of cross development tools and utilities and communication facilities for host connections to the target. The Tornado autoscaling feature analyses the application code and incorporates optional subsystems such as network services (TCP/UDP sockets).

The 200 odd Board Support Packages (BSPs) support popular commercial off-the-shelf (COTS) boards as well as the RISC and CISC architectures. Examples of targets supported are the Motorola/IBM PowerPC, Motorola 68K/CPU32, Intel Pentium Architecture family, Intel StrongArm and ARM.

3.5.1.1 Performance

Average latency 1.7 microseconds, maximum latency 6.8 microseconds on a 200MHz Pentium machine [33].

3.5.2 Windows CE .NET

Microsoft's **Windows CE .NET** [88] is a real-time operating system designed for mobile applications and applications requiring a small footprint. It supports wireless technologies such as Bluetooth, 802.1x, IPv6, Object Exchange Protocol (OBEX), MediaSense, and Real Time Communication (RTC)/Session Initiation Protocol (SIP). Windows CE .NET supports the Kerberos Security Protocol and the Secure Sockets Layer (SSL). It provides 256 levels of thread priority, nested interrupts, priority inversion and can be configured to be 200 Kb with limited kernel functions.

Windows CE .NET supports the ARM720T, ARM920T, ARM1020T, StrongARM, XScale, MIPS II/32 with FP, MIPS II/32 without FP, MIPS16, SH-3, SH-3 DSP, SH-4, 486, 586, Geode, and Pentium I-IV processors. The New Platform Wizard helps in starting designs for devices such as Cellphones/Smartphones, Digital imaging devices, industrial automation devices, Internet/media appliances and Personal digital assistant (PDA)/Mobile handhelds.

Applications developers on Windows CE .NET can use about 10% of the Win32API's available on the Windows 2000 O/S. The POSIX API is not supported by Windows CE .NET. .NET applications may be created with Microsoft Visual Studio .NET and native code created with Microsoft eMbedded Visual C++ 4.0. Using emulation technology, Windows CE .NET developers can build and test the design on their Windows 2000 or Windows XP Professional workstations.

Windows CE .NET, as a real-time version of Windows, is not POSIX compliant since Windows is not a Unix-like operating system.

3.5.2.1 Performance

Average latency 2.4 microseconds, maximum latency 5.6 microseconds on a 200MHz Pentium machine [33].

3.5.3 QNX Neutrino RTOS

QNX Software Systems Ltd markets the QNX **Neutrino** RTOS [83]. This RTOS has been deployed in embedded systems for over 20 years in mission and life-critical systems, medical instruments, aviation and space systems, process-control systems, and in-car devices. CPU's supported are the x86/Pentium, PowerPC, ARM, StrongARM, XScale, MIPS, and SH-4. QNX Neutrino complies with the IEEE 1003.1b real-time standard.

The QNX micro-kernel provides services such as thread scheduling, inter-process communication, and uses synchronous message passing to communicate with other OS modules. All OS modules run in their own memory protected address space. QNX Neutrino also provides conventional synchronization services like semaphores, conditional variables, and spin locks. Qnet, the QNX transparent distributed processing mechanism facilitates individual nodes in the network to access and use resources from any other node on the network.

The QNX Neutrino RTOS architecture provides several modular components, including the embeddable QNX Photon microGUI to develop integrated Internet capabilities, international fonts, and sophisticated user interfaces. File system support is provided for QNX, Linux, DOS, Flash, CD-ROM/DVD, CIFS and NFS. QNX Neutrino provides networking support with QNX Neutrino distributed processing (Qnet), Tiny TCP/IP stack, and NetBSD 1.5 IPv4 or IPv6 stack. Devices supported include USB, audio, PCI, serial, parallel, and IDE and SCSI disks. QNX Neutrino also provides a range of board support packages for popular boards and processors.

3.5.3.1 Performance

Average latency 1.6 microseconds, maximum latency 4.1 microseconds on a 200MHz Pentium machine [33].

3.5.4 pSOSystem 3

Wind River's **pSOSystem 3** [118] is a multitasking environment designed for use in embedded system development for both simple and complex networked applications. Recent versions of the pSOSystem provides the application developer with memory management and protection, resource monitor, and an exception manager to prevent system resets on application exceptions. The pRISM+ 3 IDE (Integrated Development Environment) and the pROBE+ target/debug agent provide tools and technologies to develop and debug firmware and mixed platform applications, profile and optimize system performance and enable and manage a team-based development process. pRISM+ supported debuggers allow full viewing of pSOS+ objects such as queues, semaphores or memory regions. pMONT+ provides real time analysis tools. The pREPC+ reentrant ANSI-compliant library, pHILE+ file system manager, and pLM+ library manager facilitate plug-in software components and libraries.

The pSOSystem 3 multitasking device independent kernel manages/allocates resources and coordinates multiple asynchronous activities. It's priority-based task scheduler supports task-specific, time-based preemptive scheduling and handles interrupts externally. pSOS+ services include task management, semaphores, events, timers, fixed/variable length queues, and asynchronous signals. Other features include User and Supervisor modes, and an Event logger.

POSIX style signals (timers, semaphores and messages) and extensions for pThreads are supported. Event driven operations are facilitated by allowing tasks to simultaneously wait for events, timers, queues and semaphores. Mutual exclusion semaphores with the priority inheritance and priority ceiling protocol are implemented. pSOSystem 3 provides QBIND (quick bindings) mode which result in speed improvements for selected system calls.

Networking support provided by the pSOSystem 3 includes TCP/IP stacks, LAN/WAN protocol, RPC, NFS client/server protocols, STREAMS, SNMP and HTTP.

The BSPs support commercially available target platforms including the Motorola/IBM PowerPC architecture family, Motorola/IBM PPC, Motorola MPC8xx, and the MIPS architecture family. The hosts supported are Windows NT 4/2000 and UNIX-Solaris workstations.

3.5.4.1 Performance

Average latency 1.9 microseconds, maximum latency 3.8 microseconds on a 200MHz Pentium machine [33].

3.5.5 Summary

This quick review demonstrates that commercial real-time operating systems are full-featured operating systems that provide a host of services which emphasize time and memory efficiency and rapid and deterministic event response. The major commercial products outlined above have been implemented on a wide variety of processors and are solid, well-supported industrial-grade software products. A summary of other RTOS products is provided in Appendix A.

Chapter 4

An Introduction to Real-Time Linux

4.1 History and Functionality of Linux

Linux is a Unix-like operating system. The first version of Unix, called UNICS (Uniplexed Operating and Computing System) was developed by Ken Thompson for the then new Digital Equipment Corporation PDP 11 family of minicomputers. UNICS (later shortened to Unix) was designed to be a simpler alternative to the MULTICS (Multiplexed Operating and Computing System) project. The MULTICS project was a joint effort by General Electric, Massachusetts Institute of Technology and Bell Laboratories and was begun in the mid 1960's. The MULTICS operating system was in turn inspired by MIT's Project MAC.

A new system implementation language, called C, was written by Ken Thompson and Dennis Ritchie. They rewrote the Unix kernel using the C compiler in 1973. C has evolved into a standard in its own right and has become a general purpose programming language commonly used for both low-level device drivers and high-level application programs.

Unix was clean and efficient, and since it provided economical access to a multi-user environment it soon gained great popularity in the technical community. AT&T gave universities royalty free licenses, which created a growing pool of advocates as students graduated and went into industry. The workstation and server markets have been dominated by Unix and Unix-like operating systems since they were created in the 1980's.

Despite the fact that licenses were royalty free, Unix was not an ideal teaching tool since licenses restricted source code distribution and modification. Several simplified Unix-like operating systems were created for teaching. One of the more popular in the late 1980's was Andrew Tannenbaum's operating system MINIX [120].

In the early 1980's, Richard Stallman initiated a project to develop freely-available software. Freely-available software is characterized by not having any restrictions, nor requiring any royalty payments, on the distribution of the software source code. One of the first projects for the Free Software Foundation (FSF) was to develop a C compiler which was essential to the major goal of developing an operating system. Since contributors to the FSF had come from a Unix background, the project was titled GNU (Gnu is Not Unix, which is read as New, is Not

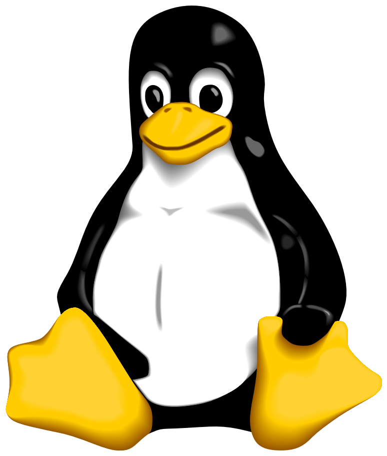

Figure 4.1: Linux Mascot

Unix) [115]. The C compiler was named **gcc** (GNU C compiler) to differentiate it from the proprietary **cc** compiler that was supplied with Unix systems.

The FSF wanted to ensure that any changes to the gcc compiler, and any other FSF software, were made available with the same license as the original work. So the GNU Public Licence (GPL) came into being [37]. The GPL specifies that source code for any changes to any FSF project must be made available with the same conditions as the original contribution.

Quickly following the release of an operational C compiler, there was a concerted effort to develop the user-space command-line tools that would be required for users to interact with the planned operating system. Since the command-line tools were developed from the ground up, these tools do not contain any of AT&T's proprietary Unix code.

Linus Torvalds started the development of Linux as a student project at the University of Helsinki [123]. The first version of Linux, released in September 1991, was distributed as source code and could only be compiled under MINIX using gcc. The early versions made extensive use of the paging architecture of the Intel 386 processor, seemingly limiting Linux to that CPU family. These limitations have been progressively removed over the last decade, and now standard Linux can execute on fifteen different processor families ranging from IBM's zSeries mainframes to the ARM processor commonly used in handheld devices.

A Linux logo competition evolved around the time of the release of version 2.0. At Linus Torvalds request, the focus of all the entries was the penguin and Linus finally chose the image in Figure 4.1 of a penguin "with a stomach full of herring" as embodying the contented yet strong philosophy of Linux [81].

For many years Linux was an underground phenomenon, creating something of a cult among hackers and university computer science students and faculties. More recently, Linux has become widely popular as a main stream technology since it is available with open source code with no requirement to pay royalties or license fees. Freely available, open-source software is not necessarily free of cost for the life of the product. Companies such as Red Hat have business models based on selling documentation, technical support and maintenance for Linux. IBM's adoption of Linux for their server business has given credibility for corporate decision makers and helped fuel the rapid rise of Linux in recent years.

Linux version numbering, as given in Table 4.1 provides that even-numbered versions e.g. 1.0.x, 2.2.x are stable releases. Odd-numbered versions are developer releases that are continually having features added to the base kernel.

Version Range	**Period**
0.02 – 0.99	October 1991 – March 1994
1.0.0 – 1.3.100	March 1994 – May 1996
2.0.0 – 2.0.39	June 1996 – January 2001
2.1.0 – 2.1.129	September 1996 – November 1998
2.2.0 – 2.2.22	January 1999 – September 2002
2.3.0 – 2.3.99	May 1999 – May 2000
2.4.0 – 2.4.19	January 2001 – August 2002
2.5.0 – 2.5.50	November 2001 – December 2002
2.6.0 –	Release scheduled March 2003

Table 4.1: Linux Version History

Contributions to the kernel code are made by many developers from all parts of the world. Ultimately, the code contributions that are incorporated into the official releases are personally approved by Linus Torvalds. These official releases are made available for download from kernel.org [124]. There are also un-official source-code patches to the kernel that various developers make available so that people who want to incorporate the offered functionality can do so by modifying their own copy of the kernel source code using the **patch** program and the source-code patches. One example of patches that made their way into the kernel.org distribution is code contributed by Robert Love that reduced the interrupt latency to less than 1 millisecond [82]. These modifications were first made available in December 2001 as patches against kernel version 2.4.17. Then in February 2002 they were incorporated into the kernel.org distribution with version 2.5.4.

The motivating goal for the development of Linux was to have a POSIX-compliant operating system. This target has been maintained over the years and Linux complies with POSIX.1, system interface and headers, and, for the major part, compliant with POSIX.2 shell, tools, and user utilities.

Code developers for Linux typically use the GNU tool chain along with their favorite editor to create and test code [115]. Compilers exist for essentially all programming languages ranging from Ada, through C/C++, to Java. There are also graphical Integrated Design Environments (IDEs) for code-development and debugging [31].

4.1.1 Nomenclature

Throughout this report the phrase "standard Linux" or "standard kernel" shall refer to the kernel.org distribution, or non-real-time Linux kernel. The phrase "real-time Linux" or "real-time kernel" shall refer to all implementations, discussed in Section 4.2.2, that bring low-latency and low-jitter real-time performance to the Linux kernel.

4.1.2 Standard Linux for Soft Real-Time Applications

It is possible to use standard Linux for soft real-time control applications where the sample time is relatively long, say 100s of milliseconds, and the application can tolerate missed timing schedules. One example is displaying video, where missed schedules mean that a small number of frames are lost but the video continues. Another example is for streaming audio where missed schedules mean loss of audio generally heard as clicks or pops. A third example is data acquisition and display where a missed schedule means that a small set of data points never get displayed. If loss of data can be tolerated, then standard Linux, especially the latest kernels (version $>$ 2.5.4) that include the bottom-half preemption logic, described in Section 4.2, can be used. If that is not the case, then the hard real-time implementations are required.

4.1.3 Embedded Applications

Real-time systems may also be embedded systems. Embedded applications typically run in small physical and computing footprints, often without keyboards, mice or monitors, and usually without rotating media such as hard drives or CD-ROMs that could not withstand harsh environments. Programmers can configure Linux to run without any of these devices, and there are several popular embedded Linux distributions that come pre-configured and include useful tools for customization. Common customizations include adding flash memory storage to replace rotating media hard disks, and replacing the memory- and disk-consuming X Windows graphics component with stripped-down versions. Indeed, many commercial products run Linux internally and do not resemble desktop computers at all; examples include set-top boxes, television video recorders, personal digital assistants (PDAs), and game consoles. Many Linux distributions streamline the system considerably, both reducing the time to boot and resource consumption; a web list is maintained at [80].

In embedded applications the multi-minute boot time for standard Linux would generally be a problem. For example, a consumer would reasonably expect a set-top cable-TV box to be operational as soon as it is turned on. Sub-ten-second boot times are possible with a memory-based file-system and, in the case of an IA32 PC, replacing the BIOS with LinuxBIOS that contains only the minimal functionality that Linux requires at boot time [78].

The usually-assumed platform for standard Linux is an Intel-based desktop or laptop machine. What is becoming more commonplace is for manufacturers of handheld devices, set-top devices, or dedicated appliances to incorporate standard Linux as the operating system for these embedded applications. These applications may not need real-time performance and the latency and jitter of standard Linux is not a limitation for these applications. Alternatively, there can be embedded hardware that requires deterministic real-time performance and in these cases, one of the real-time Linux distributions will be used in the embedded application. So, embedded Linux can run on hardware ranging from desktop machines to video wrist watches, but embedded Linux is not necessarily real-time Linux.

4.2 Modifications that make Linux Real Time

Interrupt processing in the standard kernel is divided into two halves. The two sections are referred to as top-half and bottom-half tasks. The bottom-half task is the interrupt handler and reads data from the physical device into a memory buffer. The top-half task reads from the memory buffer and passes the data to a kernel accessible buffer. In the standard kernel, without the preemption patches, all interrupts are disabled when the bottom-half task is running. This means that there can be an arbitrary delay, the latency, before a second interrupt can be serviced.

The standard Linux kernel provides a default timer resolution of ten milliseconds. The timer resolution can be improved by recompiling the standard kernel using a different timer divisor, but even with the improved latency of one millisecond in version 2.5.4 there is not much point in much higher timer resolution.

All variants of real-time Linux have introduced modifications at the kernel level. The result of these modifications is to reduce both the interrupt latency and jitter between periodic interrupts to the microsecond range, allowing for faster response to external events and higher resolution timing.

4.2.1 Standard-Kernel Preemption

This methodology is to modify the standard Linux kernel to ensure that higher priority kernel processes can preempt lower priority processes and gain access to needed resources. This involves changing the standard device-driver interrupt-handler so that higher-priority interrupts are not blocked for arbitrary amounts of time while a lower-priority interrupt handler completes its task.

Starting with Linux version 2.5.4, the standard kernel incorporates preemption logic as shown in Figure 4.2. The resultant latency and jitter is around 1 millisecond on Pentium processors with CPU clocks running at 100s of MHz. This performance improvement is satisfactory for supporting user-space audio and video applications, but is clearly inadequate for running periodic interrupts at sub-millisecond sample times. These sub-millisecond sample times are possible with the implementations discussed in Section 4.2.2.

Note that in Figure 4.2 the real-time task is one of many tasks being controlled by the kernel scheduler. This task is the kernel's reference to the real-time user-space process. The scheduler is able to run this higher-priority real-time task by preempting any lower priority non-real-time

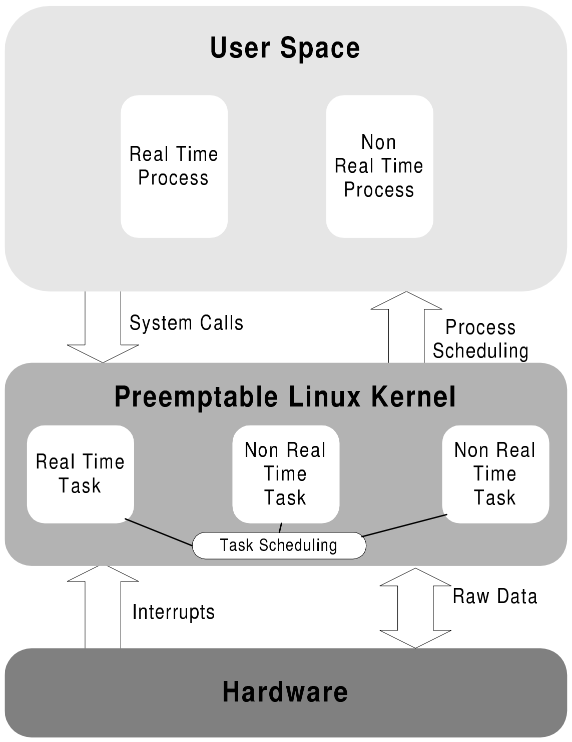

Figure 4.2: Preemptable Kernel Architecture

tasks in kernel space and thus the user-space process can achieve the one millisecond scale latency that version 2.5.4 made possible.

4.2.2 Real-Time Patches to the Standard Kernel

Four strategies for modifying standard Linux to provide real-time capability are discussed in this Section. Three of the four involve adding a second, highly efficient, kernel to handle real-time tasks and the fourth involves making changes directly to the standard kernel. Example implementations are briefly mentioned here and discussed in more detail in Chapter 7. Each of these variants of real-time Linux is distributed in the form of kernel patches. That is, none of these variants are a part of the official kernel source code distributed at kernel.org.

4.2.2.1 Micro Kernel

This implementation style provides a second kernel that is an interface layer between the standard kernel and the hardware layer as shown in Figure 4.3. This second kernel is a compact code module, a micro kernel, that controls the execution of real-time tasks and runs the standard Linux kernel as a background task. The micro kernel intercepts the hardware interrupts and ensures that the standard kernel cannot preempt any interrupt processing in the micro kernel. It also schedules the real-time tasks with the highest possible priority to minimize the task latency.

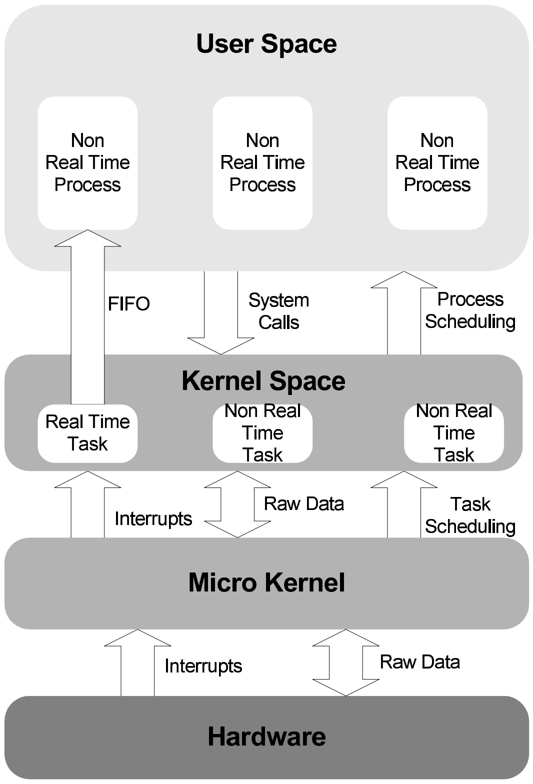

Figure 4.3: Micro Kernel Real-Time Architecture

The primary example of the micro-kernel implementation is RTLinux, developed at New Mexico Tech. The originator, Victor Yodaiken, has been awarded a patent that covers the logic of this implementation [139]. The latest major release of RTLinux, Version 3, was made available in February 2001. The main change for this version is that it implements the POSIX 1003.13 PSE51 minimal real-time environment [71].

RTAI, developed at Politecnico di Milano, is the second distribution of real-time Linux to use the micro-kernel strategy [29]. This distribution started with an early release of RTLinux, in April 2000, and enhanced the higher-level interfaces to the real-time services. The most recent release, version 24.1.10, was made in September 2002.

4.2.2.2 POSIX Real-Time Extensions

The second real-time modification strategy is to directly modify the standard kernel and to provide libraries that together implement the POSIX real-time extensions, thereby creating a system that conforms to the IEEE 1003.1d standard. In contrast to the micro-kernel strategy there is no second kernel in this approach. The patches are made directly to the standard kernel to implement the timers, signals, semaphores, process memory locking, shared memory, priority scheduling, synchronized and asynchronized I/O specified in IEEE 1003.1d.

Two different approaches are currently available:-

KURT The Kansas University Real-Time Linux (KURT) distribution started out as a project to implement the timer enhancements specified in IEEE 1003.1d. While the current KURT release contains the high-resolution timers and real-time scheduling improvements, it does not claim adherence to IEEE 1003.1d.

TimeSys Linux TimeSys was the distributor of the CMU Resource Kernel approach described in Section 4.2.2.4. The Resource Kernel approach provided many useful capabilities but did not reach latency levels needed for many hard real-time applications.

In May 2002, TimeSys Corp released their own open-source preemption patches to the standard kernel [23]. In addition to making the standard kernel preemptable, the modifications include a kernel scheduler that provides scheduling for the bottom-half kernel interrupt processing and for interrupts themselves. This architecture is depicted in Figure 4.2 and brings the kernel latency and jitter down below 100 microseconds for CPUs running at 100s of MHz.

The preemption and scheduling patches, released under a GPL licence, do not provide the high-resolution timers that are needed for repetitive real time tasks. The code modules that implement the timers along with modules for CPU and network resource reservation are part of the TimeSys proprietary development environment called TimeSys Linux.

4.2.2.3 Nano Kernel

The Adaptive Domain Environment for Operating Systems (ADEOS), released as an open-source project in June 2002, provides an interface between the standard kernel and the hardware [101]. This is similar to the micro kernel approach described above, but is implemented in such a way as to avoid the claims of the Yodaiken patent. The background history and design for ADEOS is described in a white paper published in 2001 [136]. The ADEOS project is releasing all code under the the provisions of the GNU GPL to provide a real-time framework that is not encumbered by any patented technology.

One significant characteristic of the nano-kernel approach is that many operating systems can be run in parallel on top of the nano-kernel layer. As shown in Figure 4.4 one of the operating systems can be real time, the others can be non-real-time operating systems.

ADEOS developers have called their hardware-to-kernel interface a nano-kernel rather than a micro-kernel to further distinguish their contribution from the patented work of Yodaiken.

4.2.2.4 Resource Kernel Extension

As shown in Figure 4.5 the resource kernel is a compact code module that provides a gateway to resources such as file systems, serial and parallel ports, and network interfaces for both the normal kernel and also for user processes. Not only does the resource kernel control the interrupt preemption issues for the standard kernel, it provides mechanisms where user-space

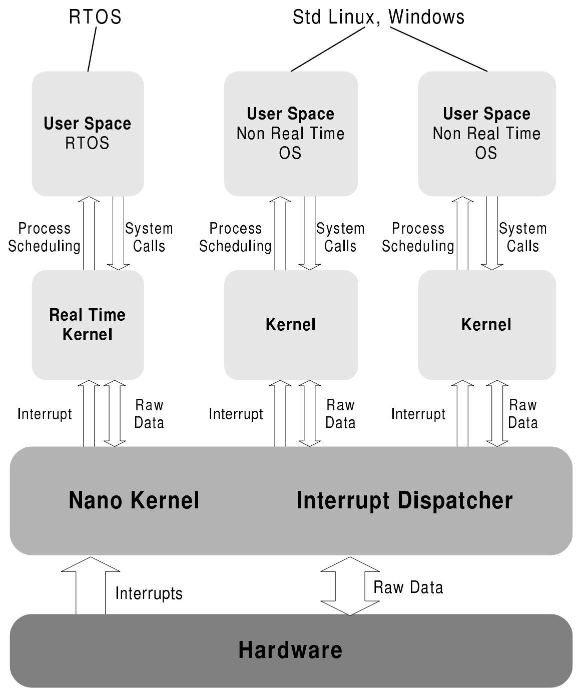

Figure 4.4: Nano Kernel Architecture

programs can request, reserve, and be guaranteed a finite percentage of machine resources such as memory, CPU cycles, network and file-system-transfer bandwidth.

The resource kernel architecture was developed at Carnegie Mellon University starting in 1996 [125] and was distributed for several years by TimeSys. The resource kernel as originally developed never reached latencies required for hard real-time applications, but it provided the base for the TimeSys Linux May 2002 release described in Section 4.2.2.2.

4.3 Performance Characteristics

The fundamental criteria for evaluating performance of a real-time operating system are event latency and periodic jitter. An event can be an interrupt generated by hardware external to the CPU or it can be a signal internal to the operating system such as a notification to start the next task on a ready-to-run queue.

4.3.1 Event Latency

Events can be either hardware-generated interrupts or operating-system generated software signals. Event latency is shown in Figure 4.6.

In the case of a hardware interrupt, the latency is the time from when a hardware interrupt requests service to the time when the first instruction of the interrupt-service routine is exe-

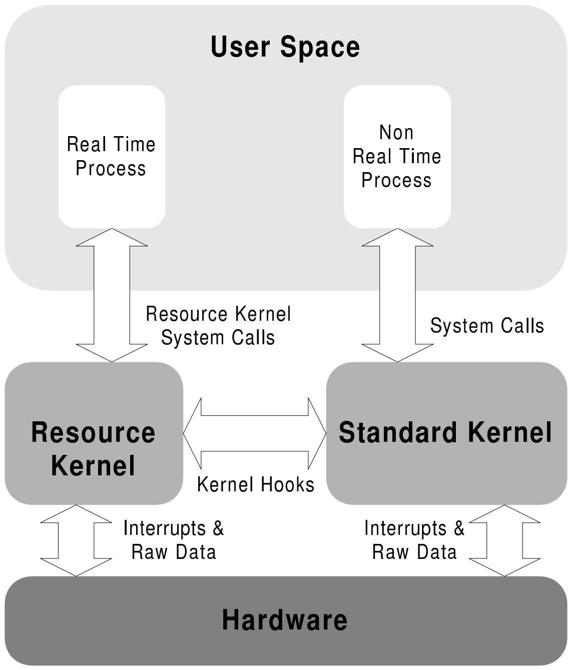

Figure 4.5: Resource Kernel Real-Time Architecture

cuted. This latency can be measured with instrumentation external to the computer system e.g. oscilloscopes, timer/counters, etc. One consideration in measuring interrupt latency is that the measurement will include any delays introduced by the computer-system peripheral bus. Such delays could be due to other peripheral devices transferring data over the bus and preempting CPU bus access.

For an operating-system generated event, the latency is the time from when the signal is generated to when the first instruction of the task is executed. This latency can be measured using the CPU's clock counter register. This 64-bit register is incremented at the clock rate, every 2 nanoseconds for a 500MHz CPU clock, and a sequence of its values can be stored in memory for later analysis. This measurement technique is not affected by any delays due to peripheral device bus accesses, because the counter register is internal to the CPU and is always incremented every clock cycle.

4.3.2 Periodic Jitter

Periodic jitter as shown in Figure 4.7 refers to the variations in time that a repetitive task experience as it executes.

The repetitive task is at the heart of sampled-data control of mechanical devices. The models of the device to be controlled are calculated on the assumption that the sample time is known and fixed. The control algorithms are, in turn, calculated from the device model, reinforcing the dependence on a known and stable sample time. Any jitter in the sample time leads to imprecision in the control-system performance.

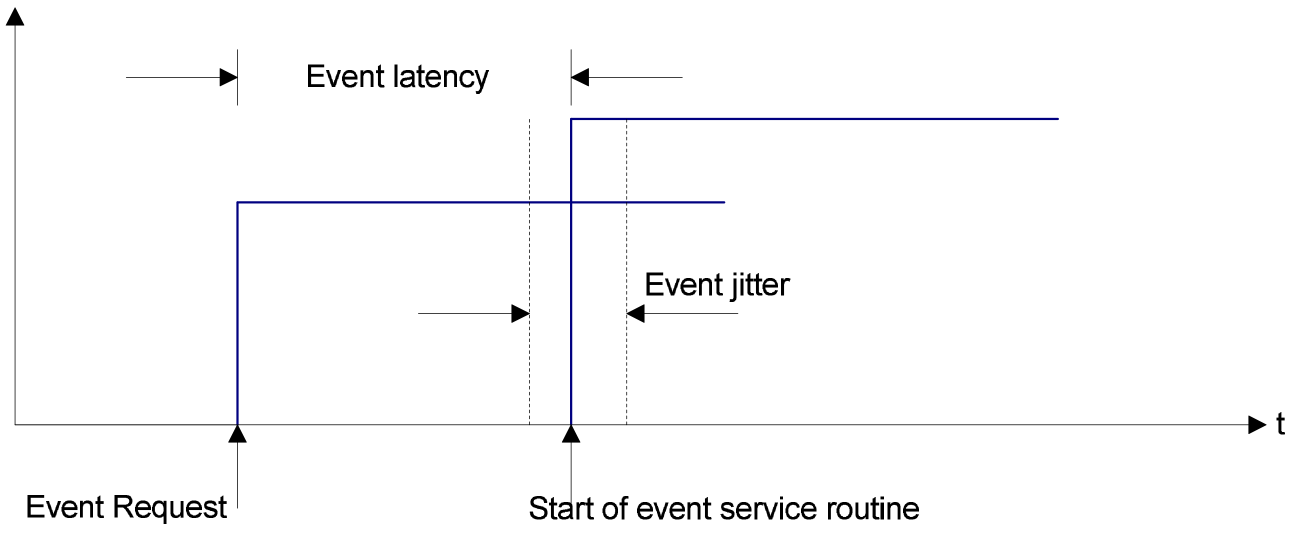

Figure 4.6: Event Latency

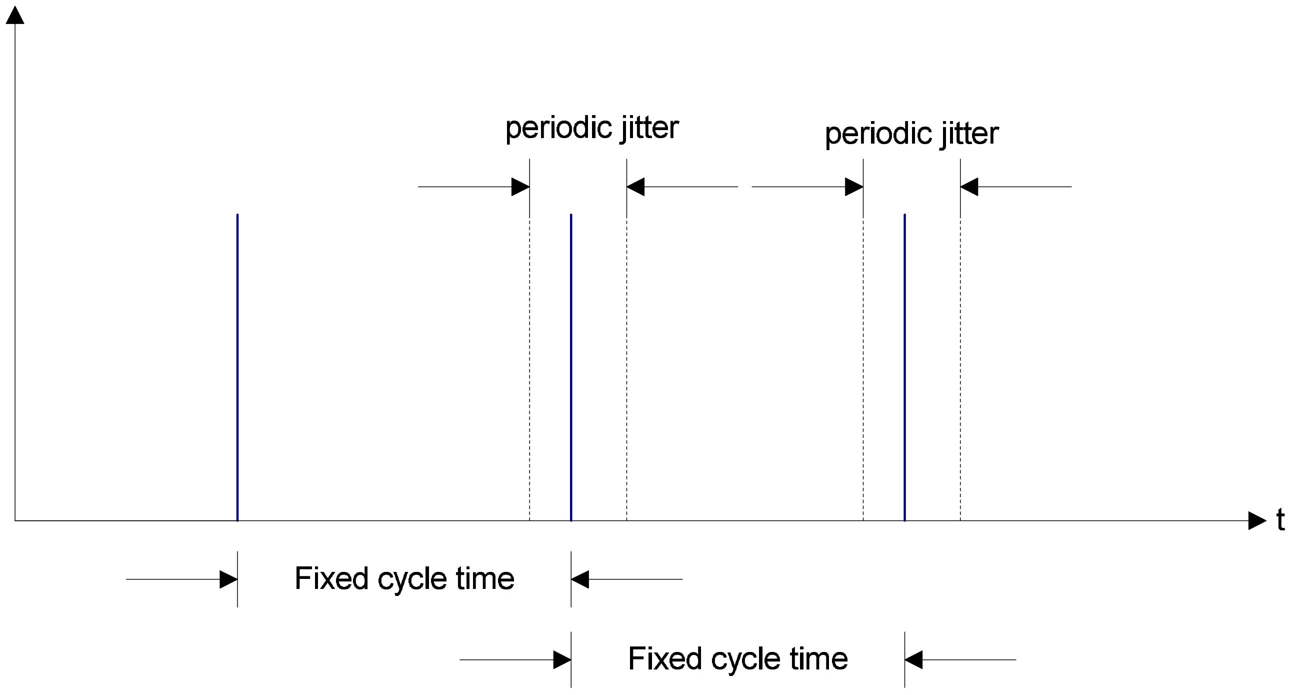

Figure 4.7: Periodic Jitter

4.4 System-Performance Measurement

Seminal papers discussing the latency and jitter issues have been presented at the Second and Third Real-Time Linux Workshops. [96] [130] Both these papers discuss, and present results, for periodic jitter that have been obtained using the time-stamp-counter technique discussed in Section 4.3.1. The first paper, [96], reports a periodic jitter for both RTLinux and RTAI of less than 6 microseconds for a 500 microsecond periodic event running on a 400MHz Pentium. Periodic jitter for a 50 millisecond periodic process is reported as being less than 10 microseconds for both RTLinux and RTAI in the second paper [130].

Three articles at linuxdevices.com [25] first present results that show standard Linux is not capable of real-time performance because of the significant variation in interrupt latency as the machine load changes. The articles then proceed to look at the options available that will make Linux real-time, and then conclude that a periodic jitter test program running on a 465MHz Celeron IA32 CPU can produce a worst-case jitter less than 30 microseconds for both RTLinux and RTAI.

The members of the Linux community who are interested in multi-media (audio/video) applications running on standard Linux are concerned with the periodic jitter performance in standard Linux. The large variations in periodic jitter prompted the development of the kernel preemption code that is now part of the 2.5 series standard kernels. Results presented in [39] are for a 750MHz Pentium running standard kernel version 2.4.6 that has been patched with the pre-emption code. The 5.8 millisecond periodic task is reading 768 bytes of audio data. The average jitter reported is less than 1 millisecond with the system running a variety of stress tests. A tool was developed [111] to ease the process of plotting the jitter in either a standard or a real-time kernel. The tool plots the data in a graphical form onto the X11 display.

4.4.1 Comparing Real-Time and Standard Linux

The Electronics Laboratory in the Physics Department at the University of La Plata, Argentina, published a comparison of standard and RTLinux [36] in 1997. The hardware was a 120MHz Pentium machine running standard Linux and then RTLinux.

One test compares the periodic jitter in generating a 40 millisecond square wave, the second test compares the interrupt latency for standard and RTLinux.

The square wave task was executing when the machine was subject to a number of different loading conditions. The average periodic jitter was 185 microseconds for standard Linux and reduced to 1.75 microseconds for RTLinux

The interrupt latency task was also run under the same varying load conditions. The latency varied from 20 microseconds to 4800 microseconds using standard Linux and was reduced to a range of 2.4 microseconds to 4.4 microseconds when the machine was running RTLinux.

These results were obtained by measuring the transitions of pins on the parallel port and the transitions of the interrupt-request line with period-measuring timer/counter hardware. One compounding issue for this technique is that the timing data is transferred over the CPU data bus to the peripheral chip that is the parallel port, and this data bus can be held by other peripheral devices performing Direct Memory Access (DMA) data transfers. The reported results include the I/O timing uncertainties.

4.4.2 Comparison with other Real-Time Kernels

Commercial RTOS offer a wider range of board support packages than those provided by the flavors of real-time Linux. It is in the commercial interest of CPU board manufacturers to be able to offer at least one real-time operating system that will operate on their hardware straight out of the box. Equally, the commercial interests of RTOS producers are well served by being able to offer their product on a wide range of CPUs and CPU/board combinations. Real-time Linux is becoming more attractive to board manufacturers, so much so that several offer real-time Linux as one of the RTOS that can ship with their boards.

There are commercial companies supporting some versions of real-time Linux, listed in Chapter 7, and support agreements may be entered into directly with those companies. In general, technical support for open-source real-time Linux varieties is more limited than for commercial RTOS products.

An advantage of the open-source model is that it is possible for the user to implement strategies for further improving performance by modifying the code in the interrupt handler, like polling the time-stamp counter reported in [96]. Techniques such as this bring timing uncertainties into the nanosecond realm.

4.4.2.1 Performance Comparison

Real-time Linux, such as RTLinux and RTAI, exhibit comparable latency and periodic jitter to commercial real-time operating systems, that is, latency and jitter in the order of microseconds for processors running at a clock rate of 100s of MHz.

	Application	**Latency/Jitter**	**Dev. Sys.**	**Royalty**
Standard O/S	Non Real Time	100 micro to 100 milliseconds	< $500	< $100
Standard Linux	Soft Real Time	1 millisecond	$0	$0
IEEE 1003.1d Linux	Hard Real Time	10 to 100 microseconds	$0 to $20,000	$0
Real-Time Linux	Hard Real Time	1 to 10 microseconds	$0	$0
RTOS Kernels	Hard Real Time	1 to 10 microseconds	$10,000	$100

Table 4.2: Linux Performance Comparison

Table 4.2 summarizes the relative performance and costs of a cross-section of operating systems. The example *Standard O/S* could be standard Linux prior to version 2.5.4 or Windows. The *Standard Linux* row refers to standard Linux released after version 2.5.4. Approximate costs are presented in the last two columns. The *Dev. Sys.* column itemizes the cost per developer of a license for the development system tools while the *Royalty* column lists the cost per deployed unit of the final product.

The cost of development systems and royalties is only a small part of the total life cycle cost of a real-time or embedded system. Thus, Table 4.2 should not by itself be used to make decisions on which type of operating system to deploy. However, the attractiveness of free, open-source software for long term extensibility and maintainability of software systems is clear. Real-time versions of Linux now provide those attractions with performance that is comparable to proprietary commercial RTOS kernels.

Chapter 5

Application Case Studies

This chapter reviews NIST success stories in using RTLinux and RTAI in machine tool and robot control. This is followed by a sampling of applications using real-time Linux in other areas. The intention of this chapter is to convince the reader that real-time Linux is gaining widespread acceptance not only in universities but also in government and industry. The Real Time Linux Foundation provides an extensive list of applications where real-time Linux is being used at www.realtimelinuxfoundation.org.

5.1 NIST Sampled Data Control

5.1.1 EMC

The Enhanced Machine Controller (EMC) is an open-source implementation of a machine tool controller originally written by NIST, [2, 14, 112], and is now a collaborative effort involving engineers and enthusiasts from around the world. The project was started at NIST to provide a reference implementation of a complete Computer Numerical Controller (CNC) and to help validate open architecture controller standards such as the Open Modular Architecture Controller (OMAC) and STEP-NC.

EMC, as shown in Figure 5.1, implements real-time control of servo or stepper motor machines such as three-axis knee mills popular in small job shops. EMC is configurable and has been used to drive other types of machines, such as hexapods and robot cranes. Real-time implementations of EMC have been ported to RTLinux and RTAI. Non-real-time implementations have been ported to Linux, SunOS and Microsoft Windows.

EMC consists of four main components. The motion controller **emcmot** is the real-time component, and is responsible for planning trajectories, computing the machine kinematics, and driving the motors. These functions are commonly performed by CNCs in industrial automation.

The I/O controller **emcio** is responsible for handling input and output associated with spindle control, the emergency stop relays, part coolant, machine lubricators and other auxiliary

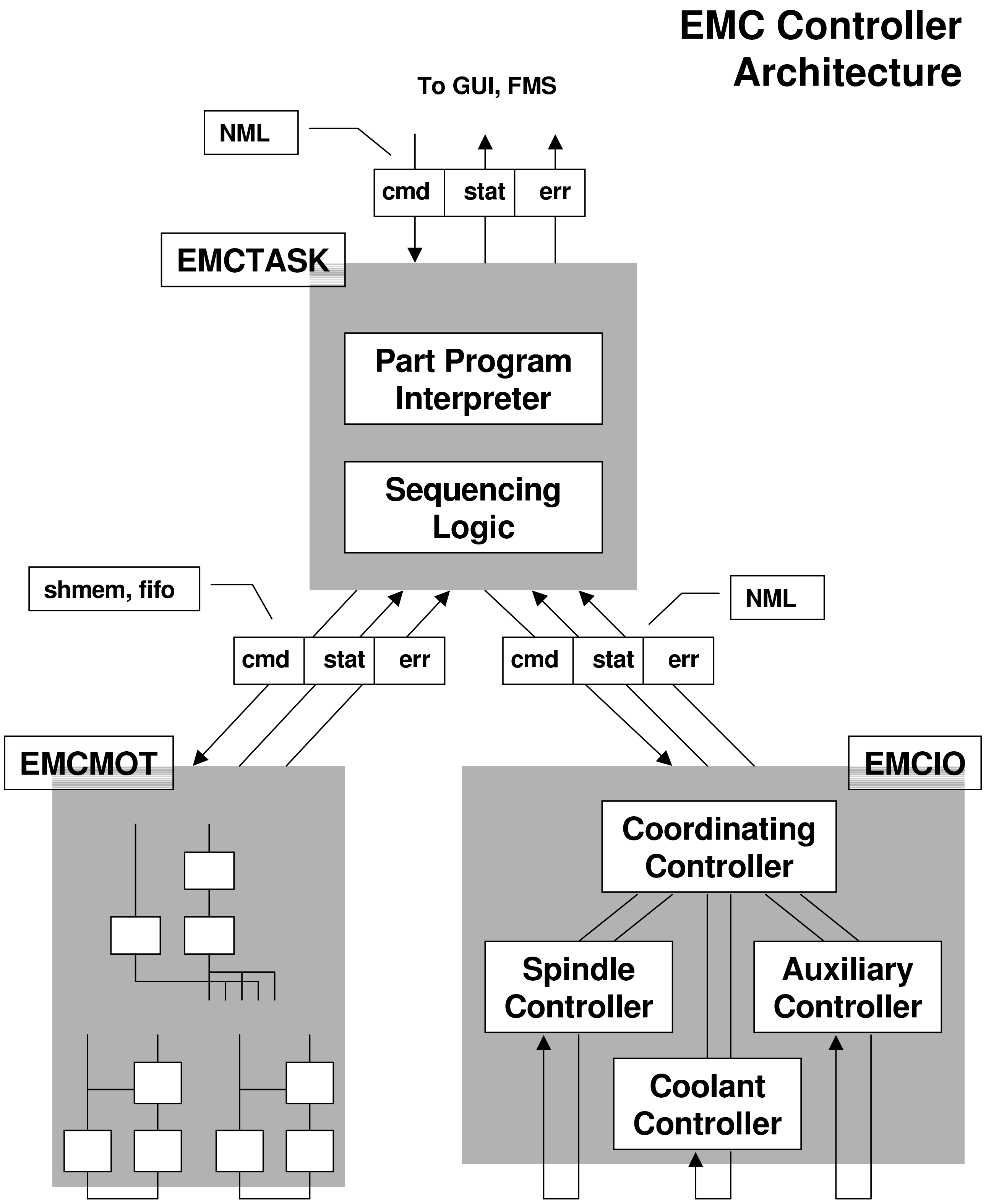

Figure 5.1: EMC Controller Architecture

equipment. These functions are commonly performed by programmable logic controllers (PLCs) in industrial automation. **emcio** does not run in real time, unlike a conventional PLC, although it is possible to do so. In fact, several open source implementations of real-time PLCs exist, such as MatPLC [133] and ClassicLadder [30] and EMC developers are actively pursuing them as replacements to **emcio**.

The task controller **emctask** is responsible for handling user requests such as to jog axes or run numerical control (NC) part programs, and sequences the interoperation of **emcmot** and **emcio**. **emctask** contains an interpreter for the RS-274 "G code" NC programming language and is compatible with computer-aided manufacturing (CAM) software, such as Mastercam and GibbsCAM.

The graphical user interface **emcgui** provides a modern point-and-click window into the EMC. Several **emcgui** implementations have been written, in Java, C++/X Toolkit and Tcl/Tk.

These components communicate using the Neutral Messaging Language, a mailbox model in which commands, status and other data structures are defined as C++ structures and read or written to buffers. NML is scalable, allowing processes running on one machine and processor type to communicate transparently with processes on other machines and processor architectures as easily as they communicate natively. It supports various communication mechanisms, such as operating system shared memory, backplane bus global memory, TCP/IP and UDP/IP networking. NML is part of the NIST Real-time Control Systems (RCS) library [40]. RCS provides platform-independent application programming interfaces (APIs) to common operating system functions such as shared memory, semaphores and timers. RCS has been ported to a wide variety of real-time and non-real-time operating systems including Linux, VxWorks, SunOS and Microsoft Windows.

emcmot communicates with **emctask** using shared memory buffers, one for commands from **emcmot** and one for status back. Although Linux processes like **emctask** cannot interrupt a real-time process like **emcmot**, care must be taken to ensure that data read by either process is consistent. For example, **emctask** may be in the middle of writing the command buffer when it is interrupted by emcmot. In this case, emcmot will read a command that is partially new and partially old. Likewise, **emctask** may be in the middle of reading the status buffer when it is interrupted, reading status that is partially old and partially new. To prevent this from happening, or rather detect that it has and flag that it should be ignored, head and tail counts are used. Head and tail counts are each a byte that bracket the data to be protected. The writer first increments the head count, changes the data, and sets the tail count to match the head. The reader copies the data out and then compares the head and tail count. If they do not match, the data is ignored. This has the potential of indefinitely postponing a successful read operation in cases where a match never occurs. In practice this arises only when the time it takes to write the data approaches the cycle time of the processes, saturating the communication bandwidth. Well-designed systems should not approach this limit, and should always handle occasional unsuccessful read attempts, usually by "coasting" on the last successful read result.

emcmot executes on the built-in timer interrupt, not on external interrupts that may be generated by hardware timers on the I/O boards that interface to motors or other electronics. Two types of scheduling are available in both RTLinux and RTAI. With one-shot scheduling, the timer chip is reprogrammed to expire at the next scheduling task occurrence. This is useful when the a task period varies depending on its current activity. The disadvantage is that reprogramming the timer takes additional time. With pure periodic scheduling, the timer chip is set up once at the beginning with a fundamental period, and all real-time tasks must execute at that period or integer multiples of it. The advantage is that reprogramming time is saved. The disadvantage is the inflexibility in task periods. In **emcmot**, pure periodic scheduling is used due to the short periods that must be supported (10 microseconds if stepper motors are controlled) and the small number of tasks, one or two, that will run.

5.1.2 Stepper Motor Control

EMC works with either servo motors or stepper motors. With servo motors, the EMC samples the current position of the motor, computes where it should be based on the desired trajectory, and generates a driving signal based on the error between the two. This is known as closed-loop control and is the subject of an extensive literature. Of primary importance in these systems is stability. A controller that does not sample and compensate often enough will cause the system to oscillate. As a rule of thumb, the more responsive the system, the faster the controller must sample. Small, light and quick systems require faster controllers than slow, heavy systems. Machines typically controlled by EMC, such as machine tools, are relatively slow and heavy; sample time of a few hundred microseconds are usually sufficient. This is below the threshold that can be handled by a general purpose operating system, but well within the range handled by real-time operating systems.

With stepper motors, the EMC must generate both the output points for the motors and the individual pulses that drive the motors themselves. These individual pulses occur much more frequently than the sampling time required in a comparable closed-loop system, on the order of tens of microseconds. It is desirable to reduce the fundamental period at which stepper motor pulses can be generated as much as possible, since the smaller the base period, the finer the control over motor speed.

Actual EMC installations on stepper motor systems have run with base periods of 10 microseconds. During this time, the stepper motor task executes, computing the up or down levels of pairs or quadruples of bits for each stepper motor, depending on the type of motor wiring. The task then writes the outputs, accumulates the steps and directions taken by each motor as a measure of their position, and finishes. Although the theoretical speed attainable is 50KHz (corresponding to 20 microsecond total for an up count and the following down count), the speed attainable in practice is about one fifth of this due to the large differences in adjacent allowable frequencies at the high end. That is, if a complete up count/down count is completed in 20 microseconds that is a frequency of 50 KHz, but if the up count/down count cycle takes 3 periods instead of 2, that is a frequency of 33 KHz, and 4 periods is 25 KHz. This quantization effect of allowable frequencies results in large command changes that tend to excite resonances in the stepper motor. The practical limit is 100 microsecond periods or 10 KHz, where the adjacent allowable frequencies are only about 10% different and resonances are not excited by sudden extreme command changes.

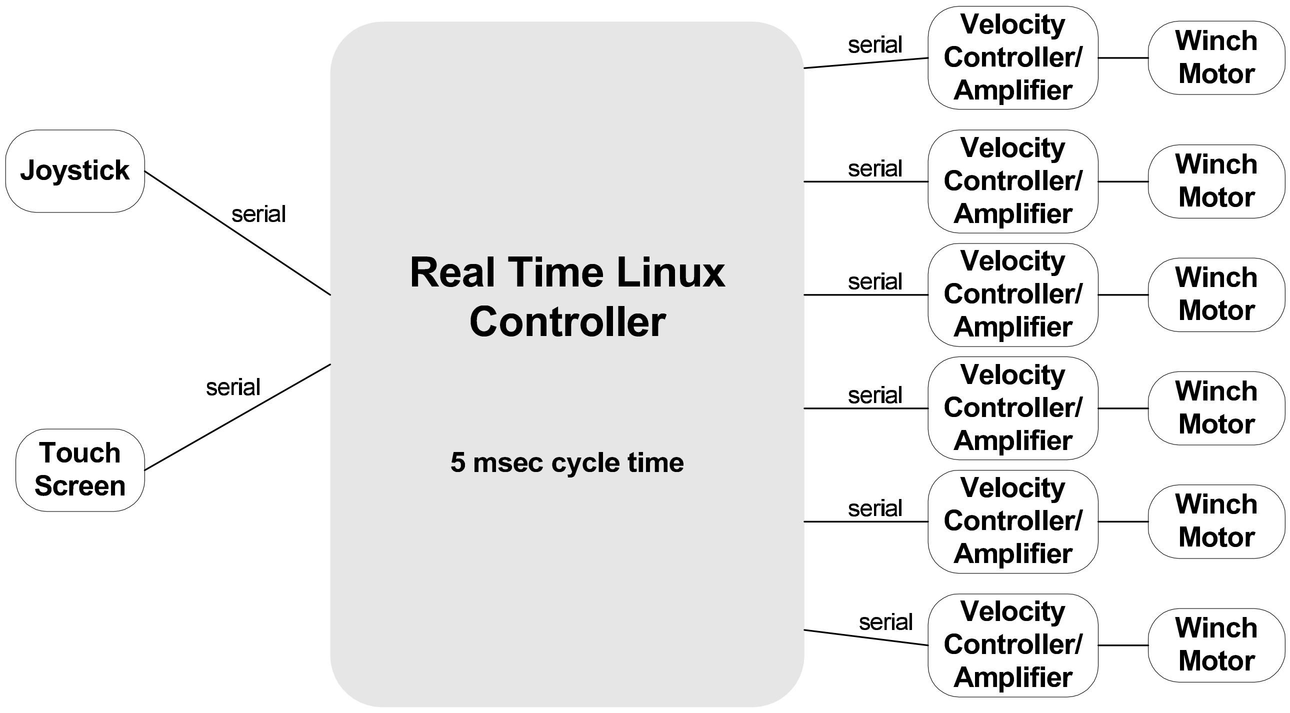

Figure 5.2: Cable Robot Architecture

5.1.3 Cable Robots

NIST built an embedded real-time Linux controller for a cable robot system applied to painting, cleaning, and descaling operations in a hazardous environment [99]. The cable robot system, depicted in Figure 5.2, consists of six servo-controlled winches attached to a moving platform that is suspended from fixed support points by cables. Individually, any one cable drives the platform in a non-intuitive way. When coordinated properly, all six cables can be commanded to effect straight-line motion of the platform. The intent of the platform is to carry construction and maintenance equipment, such as paint sprayers or removers, throughout the large work volume typical of ships or aircraft.

Amplifiers with built-in velocity servos power the winch motors. The amplifiers provide a serial interface over which velocity commands and position feedback are sent. The controller implements resolved-rate control teleoperation, in which a joystick generates the desired velocity of the moving platform in Cartesian space (V_x, V_y, V_z, and V_{yaw}), with roll and pitch always commanded to zero velocity to maintain a stable horizontal orientation of the platform. This desired Cartesian velocity vector is transformed into cable speeds through the inverse Jacobian function. The cable speed commands to the amplifiers occupy about 30 bytes.

In one configuration, a single serial link connects to a single amplifier serving a single motor. This utilizes the RS-232 protocol running at 56 Kbps and results in a control cycle of about 5 milliseconds. A second configuration uses the RS-485 protocol at 9.6 Kbps and supports two amplifiers each of which commands three motors. The cycle time for this configuration is about 100 millisecond.

Note that the longer the period, the longer the time interval over which the relationships between Cartesian and cable velocities will deviate from their exact values at the start of the interval. This will induce more drift in the roll and pitch directions (as well as the others), and cause the platform to go out of level to a greater degree. So, although the control period does not affect stability (velocity servo control is done in the amplifiers), it does affect kinematic fidelity.

The system was based on the PC/104 form factor with a low-power Geode Pentium-compatible processor, running BusyBox Linux, kernel 2.4.1, with the RTLinux Version 3.0 real-time patch. Mass storage is a 96 MByte DiskOnChip rather than rotating mass storage. The graphics system consisted of a serial touch screen on an 800x600-pixel display, running the Qt/Embedded windowing interface. Additional PC/104 I/O modules provided up to eight serial connections, and some digital and analog I/O.

The system also demonstrated the use of real-time Linux drivers for serial I/O. These drivers are required due to the restriction against calling standard Linux device drivers in real-time code, as discussed in Section 6.1.2.

5.2 Other Examples of Real-Time Linux Applications

5.2.1 Data Acquisition

5.2.1.1 Airborne Scanning Radar Altimeter

The Linux-based data system for the Airborne Scanning Radar Altimeter (SRA) project [131] was begun in 1998 at the NASA Goddard Space Flight Center. The SRA uses RTLinux to drive the waveform digitizer, compute the centroid-based range measurement between the transmit and return pulses, manage 96 automatic gain control loops, correct for aircraft attitude and off-nadir angle and deposit formatted data in a shared memory block which is extracted by a normal Linux application for recording to a disk file.

The RTLinux code, running on a 200 MHz Pentium machine, processes 256-point waveforms from the radar altimeter every 700 microseconds and integrates that data with GPS position information and aircraft attitude data obtained from the inertial navigation system. The actual control of the radar is achieving using custom programmed PIC microprocessor chips. Non real-time code writes the computed results to disk or makes the data available to other processing nodes for display in real time.

5.2.1.2 GPS Receiver

The Department of Electrical and Computer Engineering at Cornell University has developed Cascade, an RTLinux based Global Positioning System [70]. Work was started in 2000 and was made possible by an Office of Naval Research Grant. It is expected that Cascade will be used in ionospheric monitoring stations in equatorial regions of the world.

The receiver is based on the Zarlink Semiconductor GP2021 12-channel GPS correlator. The monolithic DOS version of the GPS architect code structure was redesigned and ported to RTLinux running on a 400 MHz Pentium processor. The GPS architect code was partitioned into two sections, the application code in kernel space and the user interface code in user space. The application code consists of five periodic threads to calculate the navigation solution, an ISR and a FIFO handler. The ISR runs every 800 microseconds and processes raw data from the GPS receiver to maintain the signal-tracking loops. The additional five threads, running with periods from 100 milliseconds to one second, cooperate to compute the navigation solution. FIFOs are used to communicate with the user-interface process.

5.2.1.3 Electronic Excitation Spectroscopy

Work at Dusseldorf University utilized RTLinux for the data acquisition in a system for very-high-resolution electronic-excitation spectroscopy of aromatic molecules [66]. RTLinux provides the round robin scheduler and interrupt handling mechanisms. The GUI of the data acquisition software runs as a process in user space. Queues and shared memory are used for communication between the user and kernel space processes. The complete system runs on a 233 MHz Pentium machine with the real-time code executing every 200 microseconds.

5.2.1.4 Solar Spectral Flux Radiometer

The Solar Spectral Flux Radiometer (SSFR), developed at the Atmospheric Physics Branch of NASA's Ames Research Center, is a PC/104-based system that measures solar irradiance to infer the physical properties of aerosols and clouds and to determine the radiative effects of clouds, aerosols and gases on climate [11].

The computing hardware is centered on a 266 MHz Pentium processor with 32 MByte of RAM along with a 40 MByte flash file system and a PCMCIA slot for removable flash storage. The hardware includes an interface to an IRIG-B timecode generator.

RTLinux provides a low-latency, constant sample-time environment to open and close the shutters on the spectrometers at repeatable, configurable sample times around 100 milliseconds. It also acquires aircraft navigation data which is time labeled by the common IRIG-B timecode generator. The nanosecond resolution RTLinux timers are used as a backup if the IRIG-B timestamp information is unavailable for any reason.

5.2.1.5 Telescope Camera

Ultracam is a three-channel CCD telescope camera designed for the observation of rapidly changing astronomical phenomena. It will image a target at rates of up to 500 Hz and will time stamp those images to within 1 millisecond of Universal Time (UT) [7].

The rack-mounted control PC running RTLinux provides the interface between the GPS receiver, that provides the precision time source, the camera controller itself, via a fiber-optic link, and the laboratory LAN. Ultracam is the result of a joint effort by the UK Astronomy Technology Centre in Edinburgh, University of Sheffield, Sheffield, UK, and University of Southampton, Southampton, UK. This instrument was commissioned, in September 2002, on the William Herschel Telescope on La Palma.

5.2.2 Control Applications

5.2.2.1 Machine Tool Control

iArte is a real time software environment for the development of industrial process control and machine control applications using RTAI Linux [5]. It was developed as a graduate thesis in 2000 at the Engineering Department of Brescia University. The industrial process control functions are realized using the familiar Programmable Logic Control (PLC) structure. The machine control applications can be realized with a sample time as low as 1 millisecond. Examples of machine controls developed using iArte are: a cutting machine for granite and marble slabs, a ball bearing correction and finishing machine, and a brake systems test bench.

5.2.2.2 Controlling Creatures with Linux

The Jim Henson Company's latest Performance Control System (HPCS) culminates 15 years and many generations of animatronic control-system development [107]. RTLinux is at the heart of the current system, developed in 1998/99, synchronizing video-frame traffic with actuator control commands sent to the multiple electro-mechanical control loops in the puppets themselves. This enables the Henson puppeteers to orchestrate puppet motion and blend that motion with graphical video imagery. The sample time for this system is 16.67 milliseconds (to match 60 frames per second video). The Henson developers commissioned FsmLabs to refine RTLinux by implementing a read-wait synchronization mechanism to ease communication from the real-time micro kernel to the user-space task.

5.2.2.3 Laser Cutting Machine

Prima Industrie of Torino Italy designs, develops, manufactures and sells laser machines for the cutting, welding and surface treatment of two- and three-dimensional parts. One of their products is the PLATINO [62], a laser cutting machine which uses the RTAI extensions to Linux for numerical control on a Pentium III processor. The cutting machine features speeds up to 2.5 m/sec, and accelerations up to 12 m/sec^2

5.2.2.4 Telescope Control System

The SOuthern Astrophysical Research (SOAR) Telescope Control System was prototyped and developed with the aid of LabVIEW under RTLinux over the period 1998–1999 [4]. Since the developers were working with version 1 of RTLinux, the communication between user-space and real-time processes is via the, then, RTLinux FIFO mechanism. The user-space code ran three control loops, one at 10 seconds, one at 1 second, and the third at 50 milliseconds. These loops were synchronized to the real-time activation task through the FIFO communication buffers.

5.2.3 Robotics

5.2.3.1 Humanoid Robot H7

The humanoid robot H7 [68] is a research platform for perception-action coupling in intelligent behavior of humanoid type robots. The work was undertaken by the Digital Human Lab at the National Institute of Advanced Industrial Science and Technology, Japan, in 2001.

The 1.1 GHz Pentium on-board computer runs RTLinux interleaving hard real-time tasks ranging from smooth low-level motion control to high-level vision/sensor based behaviors. The online stability control as well as the walking trajectory generation are implemented as real time tasks in the RTLinux kernel module. The on-body stereo, voice input/output, obstacle avoidance, footprint planning, and collision detection are implemented as processes in user space. The servo-loop control runs at a 1 millisecond cycle, balance feedback at a 10 millisecond cycle, and the trajectory generation at a 1 second cycle.

5.2.4 Distributed Real-Time Computing

5.2.4.1 A Real-Time Linux Cluster

In 2000, researchers at the Mississippi State University prototyped an eight node, Linux-based processing cluster [3]. The real-time component of this project was a customized version of one of the earliest RTLinux micro kernels. On each of the computing nodes, RTLinux provides bounded-jitter time-based scheduling of computing and communication processes. The RTLinux processes also enforce the inter-node time base to synchronize the time-slice computational scheme utilized by the cluster.

Chapter 6

Real-Time Linux Implementation Considerations

The decision to use an open-source operating system, and the choice of a particular one, involves consideration of many factors. Specifically, the following issues must be considered:

- Staff expertise and comfort with Unix and Linux
- Support for specific hardware to be used
- Development environments
- Availability of technical support and consulting if needed
- Total life-cycle costs
- The GNU Public Licence

This chapter will touch on several of these issues.

6.1 Development Environment

While Integrated Development Environments for standard Linux, such as the multi-language Eclipse [31] and JBuilder for the Java world [128] are becoming more common, the usual development environment is the X11 windowing system in which the user develops code with the GNU tool chain featuring the gcc compiler [69].

As for standard Linux, the common mode for real-time Linux code developers is to use the GNU tool chain along with their favorite editor to create and test code. The implementation language will be C to fit with the philosophy of the standard and real-time kernels.

The real-time tasks are written as loadable kernel modules that can be inserted or removed from kernel space by user-space commands. Thus the real-time code can be developed and tested without rebooting the machine hosting the development and execution.

6.1.1 Control System Design Tools

Control-system design is a field of study in itself. It suffices to say that the design of the controller requires knowledge of the time and frequency response of the system to be controlled. Once that is known it is possible to apply design techniques to create the discrete-time controller. This controller and system can be simulated in a graphical environment such as Simulink [85] or LabView [19, 74] to validate the basic operation of the controller. Ultimately, the controller developed in the graphical environment must be implemented on the target hardware. The following material discusses tools to achieve that goal.

United Electronic Industries offers a PowerDAQ LabVIEW for Linux [63] which allows design of real-time Linux processes from LabVIEW and works with distributions of Linux Kernel 2.2.x and 2.4.x. The driver also runs under RTLinux and RTAI realtime extensions [64].

Quality Real Time Systems offers Real Time Linux Target, a Simulink based graphical design environment for design of real-time control systems [103]. The design platform must be running Simulink. The output from RTLT is compiled C code that is executable on a RTLinux system.

One open-source project, RTiC-Lab [42], is developing a graphical environment for designing real-time controllers. Those controllers can be implemented in the RTLinux or the RTAI flavors of real-time Linux.

Scicos [110] is a graphical, dynamic-systems modeler and simulator that is built on top of the matrix computation package SciLab. Output from the open-source Scicos/SciLab system can be postprocessed by SynDex to generate, amongst other formats, code for RTLinux [114].

6.1.2 Development Considerations

6.1.2.1 Kernel Space Tasks

Real-time tasks that are created to explicitly run in kernel space, typically as a loadable kernel module, suffer some restrictions on what software resources they may access. RTLinux and RTAI support this style of real-time execution.

Standard Linux device drivers may not be called, so access to resources such as serial ports, Ethernet cards and other devices with entries in the **/dev** directory cannot be made with the Unix `open()` call. There are replacement drivers specifically written to work with real-time Linux available for most resources, including the serial and Ethernet devices just mentioned, as well as many devices common to industrial automation or scientific computing such as CAN Bus and data acquisition boards.

System kernel resources also may not be used directly. These include disk files normally accessed with `fopen()`, heap memory allocation and deallocation with `malloc()/free()` and `new/delete`, console printing via `printf()` and traditional Unix interprocess communication mechanisms such as pipes, shared memory, semaphores and message queues. However, real-time Linux provides substitutes for many of these. `printf()` is implemented using simple replacements like `rtl_printf()` (RTLinux) or `rt_printk()` (RTAI) that direct output to system log files such as **/var/log/messages**. Shared memory and first-in-first-out (FIFO) queues are supported through similar replacements, and allow real-time processes to communicate directly with non-real-time Linux processes. Other IPC features, such as semaphores and message queues, are supported for synchronization and communication between real-time tasks, but not between real-time tasks and non-real-time tasks.

More problematic are the lack of file access and heap memory operations. Access to files known *a priori* can be accomplished by Linux-level initialization programs that read the files and write them into shared memory buffers. Access to any files by name can be accomplished similarly, with the addition of a protocol by which real-time processes can specify which files are to be loaded into or cleared from shared memory. Heap memory counterparts are available in real-time Linux, called **memory pools**, which require a set-aside of a fixed-size pool of memory at the beginning of the real-time application.

A design overview of the problems associated with `malloc()` is given in [47].

6.1.2.2 User Space Tasks

An alternative to explicitly running the real-time task as a kernel module is to run the real-time task as a user-space process. These processes can be debugged using standard debugging tools, such as gdb and can access a far greater range of system resources than the kernel-space real-time tasks.

Implementations that support user-space real time include Process Space Development Domain (PSDD) from fsmlabs [56], New LinuX Real Time (NEWLXRT) from RTAI [29], and TimeSys Linux [23].

6.1.3 Debugging Real-Time Tasks

The GNU Debugger, gdb, is the tool of choice to debug real-time tasks running in user space, as is the case in the preemptable kernel modifications. There is at least one graphical front end to gdb that provides the comfort of an integrated design environment [26].

Gdb can also be used to debug real-time tasks running in kernel space, as evidenced in RTLinux, and RTAI. Gdb does require debugging support from within the real-time task, and that support is part of the RTLinux distribution. Gdb behaves as if it is debugging a process on another machine, even though the real-time task is running on the same hardware.

There is a tracing toolkit [138] that can be configured to trace actions within RTAI tasks. This is not debugging in the sense of setting breakpoints and examining data, but it does provide a graphical interface into the execution path taken through a RTAI task. This Linux Trace Toolkit, LTT, has been chosen as the preferred tool for tracing within the standard Linux kernel.

6.2 Implementation Considerations

6.2.1 Supporting Hardware Interfaces for Machine Control

Several choices of hardware interface cards (plug compatible with the PCI or ISA buses) are available to facilitate the implementation of a real-time Linux system on a commodity IA32 architecture PC. Such cards include, for example, analog-digital converters, digital-analog converters, shaft-encoder quadrature counters, and pulse-width modulation amplifiers. Several of these card manufacturers explicitly support Linux and real-time Linux by providing drivers for their products [107]. Nonetheless, the nature of real-time control is such that the implementor of the control system may well need to develop the lowest-level drivers to access the interface at the register level.

The Control and Measurement Device Interface, Comedi [116], is an open source project to unify the user and kernel space interfaces to data acquisition hardware. It is a collection of drivers, each implemented as a core Linux kernel module, and libraries (both user space and real time) that provide a common user interface to the Comedi device drivers [108]. It supports both RTAI and RTLinux extensions.

6.2.2 Total Cost of Systems Developed with Real-Time Linux

Several factors affect the total cost of ownership. The cost of acquiring the real-time operating-system software will need to be taken into consideration. Acquisition costs include development licenses and royalty payments or recurring costs per deployed system. A major consideration should be the availability and cost of the technical engineering and application level assistance provided with the operating system. Issues of staff training, application software design, implementation, test and debug are significant. Ultimately, the deployed application must be maintainable when defects surface and extensible when the requirements change and enhancements are made necessary under tight schedules. [57].

6.2.3 Support Considerations

Staff training is a major cost and time component in developing expertise in new technologies. A number of commercial training courses are available that focus on Real-Time Linux design with an emphasis on RTAI [77] and also RTLinux [12]. Also available is an open-source hands-on course with real-time Linux [10].

6.2.4 Networking and File System Access

Commercial real-time operating system companies offer networking and file system access as additional-cost packages on top of the basic real-time functionality. For real-time Linux, TCP/IP, IPX, Novell, Appletalk, and SMB Protocol stacks are freely available for the standard

Linux kernel, and are therefore accessible to the real-time application developer. A similar situation exists for file systems. Since Linux journaling and non-journaling file systems, Apple, Microsoft, and RAM-based file systems are also freely available for the standard Linux kernel, these too are available for use by the real-time application developer.

6.2.5 Working with the GPL

The GNU Public License was outlined in Section 4.1. GPL provides that any software derived from a work covered by the GPL must be distributed under the GPL. The operative word is “derived”. So any software that is created from scratch without using any GPL code can be distributed under the provisions of any license. Such distribution may be accompanied by GPL code[59]. Many companies are adding value and code to the GPL code base while creating their own proprietary code from scratch and distributing that code as binary libraries along with source distribution of GPL code. Examples of companies that pursue this business model include FSMLabs Inc, TimeSys Corp, M-Systems Inc, and MontaVista Software Inc.

6.2.6 Conclusions

There are several areas in which real-time Linux comes up short in functionality and support when compared with standard Linux and with commercial RTOS offerings. The potential user must be aware that they not only have to compile the kernel with patches from another source, but also have to grapple directly with low level issues in interfacing to hardware and managing memory. On the other hand, training and consulting services for real-time Linux are now available commercially, and the suite of development and support tools is considered substantial and continuously growing. The question ultimately is whether having a free, open-source operating system for a real-time application is worth the extra effort to implement that application. The enthusiasm of real-time Linux converts is one form of evidence that many find this an agreeable trade-off.

Chapter 7

Sources of Real-Time Linux

Distributions of real-time Linux are listed and discussed in this Chapter. They are separated into two sections, open source and commercial, and alphabetically organized within these sections. The characteristics and functionality of each distribution is described using a tabular format. Every effort has been made to ensure the accuracy of this information at the time of writing. However, aspects will change over time. Inaccuracies should be reported as per Page 2 of this document.

Open-source distributions provide the source code and build scripts so the user can understand, modify, and build the code if desired. There are no royalty fees for use of the code. Since open-source projects are developed by a loose knit community of developers whose focus is generally on code development, documentation may be sparse and technical support nonexistent. On the other hand, these systems have been the basis of many successful applications, as discussed above, and the ability to work with source code offers substantial advantages for sophisticated users. Intellectual property issues discussed in Sections 4.1 and 6.2.5 should be considered.

Commercial distributions are by companies that must make a profit to stay in business. Some of these companies are spin-offs of universities that provide free open-source distribution and then charge for application consulting, technical support and maintenance, along the lines of the Red Hat business model. Others sell enhanced versions of the software along with support and documentation.

7.1 Open-Source Distributions of Real-Time Modifications to Linux

Entries are ordered alphabetically by the name of the distributing organization.

7.1.1 ADEOS

ADEOS provides a hardware abstraction layer allowing a real-time kernel and a general purpose kernel to co-exist. The role of this layer is to channel hardware interrupts to the operating system kernels at the next higher layer in the architecture. These operating systems could be real-time or non-real time. The ADEOS system does not implement a real-time operating system, but provides the mechanism by which interrupts can be passed to a real-time operating system at the highest possible priority. The same interrupts may be passed on to a non-real time operating systems once the real-time task has been completed.

The current distribution implements ADEOS as a layer between the hardware on an IA32 PC system and the version 2.4.18 standard Linux kernel.

The RTAI developers are working towards replacing their micro-kernel architecture with the ADEOS approach. This involves designing and implementing a RTOS that will sit on top of the ADEOS nano kernel.

Item	**Criteria**	**Comments**
1.	Product Name	ADEOS
2.	Standard Kernel	2.4.18
3.	Real-Time Kernel Version	ADEOS m2
4.	Supported H/W	IA32
5.	Applications	
6.	Years in Use	2002–
7.	Multiprocessor Support	Yes
8.	Real-Time Mechanism	nano-kernel
9.	Inter-Kernel Communication	Not needed.
10.	POSIX Compliance	No real-time POSIX compliance.
11.	Development Environment	Gnu Tool Chain
12.	Performance Evaluation	
13.	Distributor Website	Download ADEOS [100].
14.	Product Support	Web site FAQs, user groups, HOWTOs.

Table 7.1: ADEOS Real-Time Linux

7.1.2 KURT

KURT [127] is Kansas University's Real-Time Linux project in the Information and Telecommunications Technology Center. KURT was started in 1997 [126] to implement the POSIX real-time extensions to Linux. The development has continued and the latest release is a patch to standard Linux version 2.4.18.

As was discussed in Section 4.2.2.2, KURT uses a single kernel which is the standard kernel modified with patches to implement real-time functionality. The patches implement the queued real-time signals and the high-resolution timers which are specified in the IEEE 1003.1d.

Despite the implementation of some of the IEEE 1003.1d specification, KURT does not claim compliance with the 1003.1d standard.

Item	Criteria	Comments
1.	Product Name	KURT
2.	Standard Kernel	Linux: 2.4.18
3.	Real-Time Kernel Version	KURT v2.3 Beta
4.	Supported H/W	i386 (any IA32 with TSC)
5.	Applications	ATM network simulations [44].
6.	Years in Use	1997–
7.	Multiprocessor Support	No
8.	Real-Time Mechanism	POSIX timers and scheduling.
9.	Inter-Kernel Communication	Only one kernel.
10.	POSIX Compliance	No
11.	Development Environment	Gnu tool chain.
12.	Performance Evaluation	10s of microseconds of latency and jitter
13.	Distributor Website	Kansas University [127]
14.	Product Support	Web site FAQs, user groups.

Table 7.2: KURT Real-Time Linux

7.1.3 RTAI

RTAI, or Linux-RT [29] is an open-source project initiated by staff at Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano, Italy. It was originally based on RTLinux but evolved as a unique flavor of real-time Linux. The list of applications on their web site [28] demonstrates the following that RTAI has in Europe.

Item	Criteria	Comments
1.	Product Name	Real Time Application Interface
2.	Standard Kernel	2.4.x upto 2.4.19
3.	Real-Time Kernel Version	24.1.10
4.	Supported H/W	x86 (with and without FPU and TSC), PowerPC, ARM (StrongARM; ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712), MIPS
5.	Applications	Laser Cutting Machine[62] and Industrial control development environment[5]
6.	Years in Use	2000–
7.	Multiprocessor Support	Yes
8.	Real-Time Mechanism	micro kernel
9.	Inter-Kernel Communication	FIFOs, Shared Memory
10.	POSIX Compliance	No

Item	**Criteria**	**Comments**
11.	Development Environment	Gnu tool chain
12.	Performance Evaluation	< 10 microseconds latency and jitter
13.	Distributor Website	Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano [29]
14.	Product Support	Web Site FAQs, user groups

Table 7.3: RTAI: Real-Time Linux

7.1.4 RTLinux

Work, led by Victor Yodaiken, at the New Mexico Institute of Technology produced the first real-time Linux [49] in 1996. Now fsmlabs.com [54] is providing access to the open-source project and offering maintenance and development consulting services.

Yodaiken was granted a patent in 1999 that covers the logic of running standard Linux as a background thread to the real-time tasks [139]. The license conditions in the RTLinux source code provide for royalty-free use of the patented process when it is used with software licensed under the Free Software Foundation GPL, or when it is used with software executing within an Open RTLinux Execution Environment.

The article “Open Source Real-Time Operating Systems” [106] provides some insights into the design tradeoff and considerations in developing applications with RTLinux.

Item	**Criteria**	**Comments**
1.	Product Name	RTLinux
2.	Standard Kernel	2.2.19 & 2.4.4.
3.	Real-Time Kernel Version	V3.1
4.	Supported H/W	IA32 486 and up, PowerPC, MIPS, AMD Elan NetSC520, Alpha
5.	Applications	Machine Tool Control [112] Cable Robot Control [99] Animatronics Control [107]
6.	Years in Use	1997–
7.	Multiprocessor Support	IA32 SMP
8.	Real-Time Mechanism	micro kernel
9.	Inter-Kernel Communication	FIFOs, Shared Memory
10.	POSIX Compliance	1001.13 PSE51
11.	Development Environment	Gnu tool chain
12.	Performance Evaluation	< 10 microseconds latency and jitter
13.	Distributor Website	FSMLabs Inc. [49]
14.	Product Support	Web Site FAQs, fsmlabs.com [54], user groups

Table 7.4: RTLinux: Real-Time Linux

7.1.5 TimeSys Linux GPL

TimeSys has released their standard-Linux preemption patches as TimeSys Linux GPL. This code is the heart of their commercial offering, see Section 4.2.2.2, and provides a preemptable kernel and an improved task scheduler. The proprietary components that add timers, periodic tasks, CPU and network reservation mechanisms are licensed for single-user development of real-time code.

Item	Criteria	Comments
1.	Product Name	TimeSys Linux GPL
2.	Standard Kernel	2.4.7
3.	Real-Time Kernel Version	3-1
4.	Supported H/W	ARM, MIPS, Pentium, PowerPC, SPARC, Super H, XScale
5.	Applications	
6.	Years in Use	1998–
7.	Multiprocessor Support	Yes
8.	Real-Time Mechanism	Preemptable Kernel
9.	Inter-Kernel Communication	Transparent to real-time application
10.	POSIX Compliance	Real-time extensions, IEEE 1003.1d-1999
11.	Development Environment	GUI IDE
12.	Performance Evaluation	$<$ 100 microseconds of latency and jitter
13.	Distributor Website	TimeSys Corporation. [23]
14.	Product Support	Web Site FAQs, user groups, Subscription with TimeSys Corp.

Table 7.5: TimeSys Linux GPL: Real-Time Linux

7.2 Commercial Distributions of Real-Time Modifications to Linux

The following descriptions of commercial distributions of real-time modifications to Linux are presented by company name in alphabetical order.

7.2.1 Concurrent Computer Corporation

Concurrent Computer Corporations offers packaged hardware/software solutions of which real-time Linux is one. The standard Linux kernel is replaced with their licensed RedHawk Linux [17] kernel that is IEEE 1003.1d compliant. They are targeting the real time [16] data-acquisition, video-on-demand, and simulation markets. They have a licensing agreement [15] to resell MontaVista's HardHat Linux as their own RedHawk kernel.

Item	Criteria	Comments
1.	Product Name	RedHawk Linux
2.	Supported H/W	Concurrent iHawk 860 Intel-based multiprocessors
3.	Real-Time Mechanism	Kernel preemption enhancements
4.	Development Environment	GUI development tools, Source-level Debugger, Event Analyzer, GUI interface to the RedHawk Linux frequency-based scheduler, Data Monitor tools [17].
5.	Distributor Website	Concurrent Computer Corporation [17] [18].

Table 7.6: RedHawk Linux: Real-Time Linux

7.2.2 FSM Labs Inc.

Finite State Machine Labs (FSMLabs) was founded by Victor Yodaiken and Cort Dougan in 1995 as VJY Associates. Cort was the prime author of PowerPC Linux which he ported and optimized for his masters thesis. They were joined by Michael Barabanov, who had implemented RTLinux for his Master's work. Cort and Michael both were graduate students at New Mexico Tech when Victor Yodaiken was a professor there.

RTLinux version 3.0 offers developers the option of using either Linux or netBSD as their general purpose OS, using the identical POSIX threads API [55].

Item	Criteria	Comments
1.	Product Name	FSMLabs RTLinux [51]
2.	Supported H/W	x86, PowerPC, Alpha
3.	Real-Time Mechanism	Micro kernel
4.	Development Environment	GNU tool chain using IEEE 1003.13 real-time services [52].
5.	Distributor Website	FSM Labs Inc [54]. Free Download [53]. Product Support & Prepackaged or Custom designed support packages [50].

Table 7.7: FSMLabs RTLinux: Real-Time Linux

7.2.3 MontaVista Software Inc.

MontaVista Software Inc. [60] [58] was founded in 1999 by James F. Ready, the co-founder of the world's first proprietary RTOS company (Hunter & Ready; VRTX). The MontaVista Linux operating system is an operating system tailored to the needs of embedded software developers. Jim Ready's goal with MontaVista is "to take Linux where VRTX went, generating income from support and subscriptions, not from royalties" [104].

Item	Criteria	Comments
1.	Product Name	MontaVista Linux [61].
2.	Supported H/W	x86/IA-32, PowerPC, StrongARM, XScale, MIPS, SH, ARM
3.	Real-Time Mechanism	Kernel pre-emption enhancements
4.	Development Environment	GUI IDE for Gnu Tool Chain
5.	Distributor Website	MontaVista Software Inc. [60].

Table 7.8: Monta Vista Linux: Real-Time Linux

7.2.4 QRTS

Quality Real Time Systems develops applications for real-time control and simulation. Their software can use the system calls of either RTLinux or QNX.

Item	Criteria	Comments
1.	Product Name	Custom Applications
2.	Supported H/W	IA32
3.	Real-Time Mechanism	RTLinux micro kernel
4.	Development Environment	GNU tool chain
5.	Distributor Website	Quality Real Time [102]

Table 7.9: QRTS: Real-Time Linux

7.2.5 REDSonic Inc.

REDICE-Linux [105] offers a revised preemptable kernel by implementing selected and limited preemption points within the kernel. REDSonic integrates the dual kernel approach and the preemptable kernel to provide a hybrid solution.

Item	Criteria	Comments
1.	Product Name	REDICE-Linux [105]
2.	Supported H/W	Intel Pentium II, 233MHz and higher, x86 and PowerPC architectures, StrongARM, ARM, and MIPS architectures.
3.	Real-Time Mechanism	Micro kernel and Kernel Preemption.
4.	Development Environment	REDICE-Linux SDK
5.	Distributor Website	REDICE-Linux [105]

Table 7.10: REDICE Linux: Real-Time Linux

7.2.6 Synergy Microsystems Inc.

Synergy Microsystems [89] primarily manufactures single board computers with both general-purpose CPUs and DSP chip sets. They support RTLinux as the operating system on their PowerPC single-board computers.

Item	Criteria	Comments
1.	Product Name	Linux SMP
2.	Supported H/W	PowerPC
3.	Real-Time Mechanism	RTLinux micro kernel
4.	Development Environment	GNU Tool Chain
5.	Distributor Website	Synergy Micro Systems [90]

Table 7.11: Synergy Micro RTLinux: Real-Time Linux

7.2.7 TimeSys Corporation

The original development of the current TimeSys product offering was performed at Carnegie Mellon University [125].

The product builds on an open-source preemptable kernel component and adds three proprietary components. These components connect to the kernel via callback hooks and augment the normal operating system with high-resolution timers and priority inheritance, with guaranteed access to CPU resources, and with guaranteed access to network resources.

Item	Criteria	Comments
1.	Product Name	TimeSys Linux/RealTime, TimeSys Linux/Net, TimeSys Linux/CPU
2.	Supported H/W	PPC, x86, MIPS, Xscale, ARM, StrongARM, Ultra-SPARC, Super H.
3.	Real-Time Mechanism	Preemptable Kernel.
4.	Development Environment	Tools [22] and SDK's [21].
5.	Distributor Website	TimeSys Corporation [20]. Downloads [23].

Table 7.12: TimeSys Linux/RealTime: Real-Time Linux

Appendix A

Currently-Available Real-Time O/S

A few real time operating systems were discussed as examples of commercial product offerings in Chapter 3. This appendix provides pointers to additional RTOS products. The list of real-time operating systems given in Table A.1 is presented in alphabetical order.

Operating system name	Manufacturer
Arx RTOS	Seoul National University
AvSYS	Avocet Systems
CMX RTOS	CMX Systems
Chorus	Jaluna S.A.
EROS	U Penn
Harmony	National Research Council
Integrity	Green Hills Software
iRMX	TenAsys
LynxOS	Lynuxworks
Maruti	University of Maryland
Nucleus	Mentor Graphics
OS-9	Microware
OnCore OS	OnCore Systems Corporation
On Time RTOS-32	On Time
OSE RTOS	ENEA
PDOS	EYRING corporation
pSOSystem 3	Wind River
QNX Neutrino RTOS	QNX Software Systems Ltd
Precise/MQX RTOS	ARC International
Real Time Mach	CMU
REAL/IX PX	Modcomp
RTMX O/S	RTMX Incorporated
SMX RTOS	Micro Digital Inc.
Spring Project	UMass, Amherst
SUMO	Lancaster University
ThreadX	Green Hills Software
TSX-32	S & H Computer Systems
VRTX	Mentor Graphics
VxWorks	Wind River
Windows CE .NET	Microsoft

Table A.1: Real Time Operating Systems

Appendix B

Sources of Further Information

This Chapter presents a selection of links (into the Bibliography) that provide more detailed information on the topic of real-time Linux. These links are listed in order of value and utility in the creation of this report.

B.1 Real-Time Linux Distributions

- The authoritative source for the standard Linux kernel code [124].
- Download site for RTLinux [53]
- Download site for RTAI [29]
- Download site for TimeSys Linux [23]
- Download site for KURT [127]

B.2 Real-Time Linux Support Organizations

- The Real Time Linux Foundation organizes an annual workshop on real-time Linux [76] and maintains a list of projects that utilize real-time Linux [134].
- LinuxDevices.com covers both embedded and real-time Linux issues [79]. The site presents product announcements and reviews along with tutorial material.
- LinuxLinks.com [80]maintains a list of web links to embedded Linux distributions.
- EmbeddedLinux.com home page lists a variety of sources of embedded Linux [32].
- An Austrian RTLinux site maintains a collection of links to RTLinux documentation [86] and a page of open-source RTLinux projects and applications [87].

B.3 Publications on Real-Time Linux

- Linux Journal is a print media and web monthly publication that includes articles on real-time Linux [67].
- Dedicated Systems Encyclopedia publishes a buyers guide for real-time operating systems including real-time Linux [34].
- DSP Consulting published a comparative survey including pricing information from 1999 [13].

B.4 Frequently Asked Questions: FAQs

- FAQs from FSMLabs provide an overview of RTLinux [48].
- RTAI does not have an FAQ page as such, but this link is the closest to an FAQ page [27].
- FAQs on real-time systems in general [8].

B.5 Real-Time Linux Tutorials

- This tutorial takes the reader through the process of developing a small code example using RTLinux [113].
- A significant, detailed description of the architecture and design process for a real-time Linux system [75]. The examples are based on RTAI.
- A hands-on, self study course on real-time Linux is found at [10]. The examples are based on RTAI.

B.6 Real-Time Linux Comparison Studies

- The NASA Glenn Research Center has published an evaluation of Real-Time Linux [9].

Bibliography

[1] IEEE Std 1003.1b-1993 IEEE Standard for Information Technology. Portable operating system interface (POSIX) part 1: System application programming interface, amendment 1: Realtime extensions. Technical report, IEEE, New York, 1994.

[2] James S Albus and R Lumia. The enhanced machine controller (emc): An open architecture controller for machine tools. *Journal of Manufacturing Review*, 7(3):278–280, September 1994.

[3] Manoj Apte, Srigurunath Chakravarthi, Jothi Padmanabhan, and Anthony Skjellum. A synchronized real-time Linux based myrinet cluster for deterministic high performance computing and mpi/rt. Technical report, Mississippi State University, 2001.
http://www.mpi-softtech.com/company/publications/files/WPDRTS_2001_Final.pdf.

[4] Michael C. Ashe and German Schumacher. SOAR control systems operation: OCS and TCS. In *Proceedings of SPIE*, Boston, MA, August 27–28 2002. SPIE.
http://www.soartelescope.org/Documents/Documents/4009-05_SOAR_Controls_Paper.pdf.

[5] RT Automation. iarte: Industrial automation real time environment, 2000.
http://www.rtautomation.it/inglese/main.html.

[6] Michael Barabanov and Victor Yodaiken. Introducing real-time Linux. *Linux Journal*, 34:1–1, February 1997.
http://noframes.linuxjournal.com/lj-issues/0232.html.

[7] Steven Beard, Andrew Vick, David Atkinson, Vik Dhillon, Tom Marsh, Stewart McLay, Mark Stevenson, and Chris Tierney. The Ultracam camera control and data acquisition system. In *Proceedings of SPIE*, Boston, MA, August 27–28 2002. SPIE.
http://www.shef.ac.uk/ phys/people/vdhillon/ultracam/spiepaper/spie_paper.html.

[8] B Van Beneden. Realtime computing FAQ, 2002.
http://www.faqs.org/faqs/realtime-computing/faq.

[9] Kalynnda Berens. Real-time Linux evaluation. Technical report, Glenn Research Center, NASA, 2002.
http://www.ivv.nasa.gov/business/research/osmasas/conclusion2002/Kalynnda_Berens_Real-time_Linux_Evaluation.ppt.

[10] Herman Bruyninckx. An open source hands-on course with real-time Linux. In *Real Time Linux Workshop*, Milan, Italy, 2001.
http://people.mech.kuleuven.ac.be/ bruyninc/rthowto.

[11] Sam Clanton. Embedded real-time Linux for instrument control and data logging, 2001. http://www.linuxjournal.com/article.php?sid=5667.

[12] K Computing. Embedded and real-time Linux development, 2002. http://www.kcomputing.com/datasheet_rtlinux.html.

[13] DSP Consulting. Embedded real-time operating system (RTOS) vendors, 1999. http://www.dspconsulting.com/rtos.html.

[14] Enhanced Machine Control. EMC Sourceforge Home Page, 2002. http://sourceforge.net/projects/emc.

[15] Concurrent Computer Corporation. Monta Vista Licensing agreement, 2001. http://www.linuxdevices.com/news/NS2534065714.html.

[16] Concurrent Computer Corporation. About Real-Time, 2002. http://www.ccur.com/realtime/rt_about.htm.

[17] Concurrent Computer Corporation. RedHawk Linux, 2002. http://www.ccur.com/realtime/sys_rdhwklnx.html.

[18] Concurrent Computer Corporation. Service and Support, 2002. http://www.ccur.com/realtime/rt_service_support.htm.

[19] National Instruments Corporation. What is LabVIEW?, 2002. http://amp.ni.com/niwc/labview/what.jsp?node=1381.

[20] TimeSys Corporation. Home Page, 2002. http://www.timesys.com.

[21] TimeSys Corporation. Linux Resource Kernel SDK, 2002. http://www.timesys.com/index.cfm?hdr=sdk_header.cfm&bdy=sdk_bdy.cfm.

[22] TimeSys Corporation. Linux Resource Kernel Tools, 2002. http://www.timesys.com/index.cfm?hdr=tools_header.cfm&bdy=tools_bdy.cfm.

[23] TimeSys Corporation. TimeSys Linux downloads, 2002. http://timesys.com/index.cfm?hdr=linux_header.cfm&bdy=linux_bdy_downloads.cfm.

[24] Charles Curley. Open source software for real-time solutions. *Linux Journal*, 66:1–1, October 1999. http://noframes.linuxjournal.com/lj-issues/issue66/3576.html.

[25] Kevin Dankwardt. What is real time? and benchmarks on real time Linux - Part 1,2,3, February 2002. http://www.linuxdevices.com/articles/AT5997007602.html.

[26] GNU DDD. Ddd data display debugger v 3.3, 2002. http://www.gnu.org/software/ddd.

[27] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. About RTAI, 2002. http://www.aero.polimi.it/ rtai/about/index.html.

[28] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. RTAI Applications, 2002.
http://www.aero.polimi.it/ rtai/applications/index.html.

[29] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. RTAI Home Page, 2002.
http://www.aero.polimi.it/ rtai/news/index.html.

[30] Marc Le Douarain. Classic Ladder, 2002.
http://membres.lycos.fr/mavati/classicladder.

[31] eclipse.org. Eclipse: a universal tool platform, 2001–2002.
http://www.eclipse.org.

[32] EmbeddedLinux.com. Embedded Linux projects and sources, 2002.
http://www.embeddedlinux.com.

[33] Dedicated Systems Encyclopaedia. Free downloadable reports, 2002.
http://www.dedicated-systems.com/encyc/buyersguide/rtos/evaluations.

[34] Dedicated Systems Encyclopaedia. The RTOS buyer's guide, 2002.
http://www.dedicated-systems.com/encyc/buyersguide/rtos/Dir228.html.

[35] Jerry Epplin. Inside Real-Time Linux. *Dr. Dobb's Journal of Software Tools*, 25(3):72–78, March 2000.
http://www.ddj.com/ftp/2000/2000_03/rtlinux.txt.

[36] FISICA. Testing Real-Time Linux. Technical report, University of La Plata, Physics Department, Electronics Laboratory, 1997.
http://www.fisica.unlp.edu.ar/rt.

[37] Free Software Foundation. What Is Copyleft, 2002.
http://www.gnu.org/copyleft/copyleft.html.

[38] Bill O. Gallmeister. *POSIX.4 – Programming for the Real World*. O'Reilly & Associates, 1995. POSIX.4 was the original nomenclature for the current POSIX.1b.

[39] Nigel Gamble. The Linux kernel preemption project, 2001.
http://kpreempt.sourceforge.net.

[40] Veysel Gazi, Mathew L Moore, Kevin M Passino, William P Shackleford, Frederick M Proctor, and James S Albus. *The RCS Handbook: Tools for Real-Time Control Systems Software Development*. John Wiley & Sons, Inc., New York, 2001.

[41] Marcus Goncalves. Is real-time Linux for real? Technical report, Real-Time Magazine, 1999.
http://www.realtime-info.be/magazine/99q4/ar2.pdf.

[42] Edgar F Hilton. Real Time Controls Laboratory, 2001.
http://sourceforge.net/projects/rtic-lab.

[43] Edgar F. Hilton and Victor Yodaiken. Real-time applications with RTLinux. *Embedded Linux Journal*, 1:18–20, 22–25, January/February 2001.
http://embedded.linuxjournal.com/magazine/issue01.

[44] S. House and D. Niehaus. KURT-Linux support for synchronous fine-grain distributed computations. In *Proceedings of the Sixth IEEE Real-Time Technology and Applications Symposium (RTAS '00)*, pages 78–89, Washington - Brussels - Tokyo, June 2000. IEEE.
http://hegel.ittc.ukans.edu/projects/proteus/docs/RTAS-2000.pdf.

[45] M. Humphrey, E. Hilton, and P. Allaire. Experiences using RT-Linux to implement a controller for a high speed magnetic bearing system. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium (RTAS '99)*, pages 121–131, Washington - Brussels - Tokyo, June 1999. IEEE.

[46] IEEE Std 1003.1-2001 (Open Group Technical Standard, Issue 6). Standard for Information Technology—Portable Operating System Interface (POSIX). Technical report, IEEE Press, New York, 2001. Superceeds 1003.1-1996, 1003.2-1992, and include 1003.1d-1999.

[47] FSM Labs Inc. RTLinux Design White paper, 2001.
http://www.fsmlabs.com/developers/white_papers/design.htm.

[48] FSM Labs Inc. FAQ, 2002.
http://www.fsmlabs.com/products/faq.htm.

[49] FSM Labs Inc. First real-time linux, 2002.
http://www.fsmlabs.com/community.

[50] FSM Labs Inc. FSMLabs Product Support, 2002.
http://www.fsmlabs.com/products/support.htm.

[51] FSM Labs Inc. FSMLabs RTLinux, 2002.
http://www.fsmlabs.com/products/software.htm.

[52] FSM Labs Inc. FSMLabs RTLinux Development, 2002.
http://www.fsmlabs.com/developers.

[53] FSM Labs Inc. FSMLabs RTLinux Free Download, 2002.
http://www.fsmlabs.com/products/download.htm.

[54] FSM Labs Inc. Home page, 2002.
http://www.fsmlabs.com.

[55] FSM Labs Inc. RTCore/BSD Released, 2002.
http://www.fsmlabs.com.

[56] Fsmlabs Inc. Process Space Development Domain, 2002.
http://www.linuxdevices.com/news/NS3304386747.html.

[57] Lineo Inc. Embedded Linux: Issues, questions, and answers, 2002.
http://www.lineo.com/products/embedix_sdk/embedix/Embedding_Linux_v1-3.pdf.

[58] MontaVista Software Inc. Company Overview, 2002.
http://www.mvista.com/company/overview.html.

[59] MontaVista Software Inc. Frequently Asked Questions, 2002.
http://www.mvista.com/products/faq.html.

[60] MontaVista Software Inc. Home Page, 2002.
http://www.mvista.com.

[61] MontaVista Software Inc. MontaVista Linux Professional and Carrier Grade Editions, 2002.
http://www.mvista.com/products/index.html.

[62] Prima Industrie. Platino product information page, 2002.
http://www.primaindustrie.com/pr_platino.html.

[63] United Electronic Industries. Power DAQ LabVIEW for Linux, 2002.
http://www.ueidaq.com/products/products.asp?pn=LABVIEW-LINUX.

[64] United Electronic Industries. Power DAQ LabVIEW for Linux - Press release, 2002.
http://www.ueidaq.com/press/press.asp?MM_RecordId=33.

[65] ISO. Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language], 1996.
http://www.iso.org.

[66] Jochen Küpper. Data acquisition with real-time Linux. Master's thesis, Duesseldorf University, 2000.
http://www-public.rz.uni-duesseldorf.de/ jochen/computer/RT-Linux/RT-Linux.html.

[67] Linux Journal. The Monthly Magazine of the Linux Community, 2002.
http://www.linuxjournal.com.

[68] Satoshi Kagami. Humanoid robot h7 for autonomous and intelligent software research. In *Real Time Linux Workshop*, Milan, Italy, 2001.
ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-2001/proc/k02-kagami.pdf.

[69] Jason Kroll. Developing software with Linux, 1999.
http://linuxsolutions.hypermart.net/linux/linux_main.htm.

[70] Brent Ledvina, F. Mota, and P.M. Kintner. A coming of age for gps: A rtlinux based gps receiver. In *Proc. of the Workshop on Real Time Operating Systems and Applications and Second Real Time Linux Workshop (in conjunction with IEEE RTSS 2000)*, Orlando, Florida, 2000.
http://gps.ece.cornell.edu.

[71] Rick Lehrbaum. An interview with Victor Yodaiken. *Embedded Linux Journal*, 0:42–46, November 2000.

[72] Rick Lehrbaum. Using Linux in embedded and real-time systems. *Linux Journal*, 75, July 2000.

[73] Rick Lehrbaum. Adeos – a resource sharing multi-os environment, 2001.
http://www.linuxdevices.com/articles/AT7788559732.html.

[74] Thomas Leibner. Graphical Programming of Realtime Applications with LabVIEW for Linux RTAI. Technical report, Real Time Linux Foundation, 2002.
ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-2002/proc/c04_leibner.pdf.

[75] Arnold Leucht. Hard real-time capability under Linux – development within the micro-kernel real-time architecture, 2002.
http://www.electronicengineering.com/features/embedded/OEG20020220S0009.

[76] Real Time Linux Foundation Inc. Real time Linux foundation inc., 2002.
http://www.realtimelinuxfoundation.org.

[77] ASCENSIT Linux Solutions. Real-Time Linux Development, 2002.
http://www.ascensit.com/education/aa303/?Lang=us.

[78] LinuxBIOS.org. The LinuxBIOS Home Page, 2002.
http://www.linuxbios.org.

[79] LinuxDevices.com. Embedded and real-time Linux products, projects, and sources, 2002.
http://www.linuxdevices.com.

[80] LiNUXLiNKS.com. The Linux portal: Embedded distributions, 2002.
http://www.linuxlinks.com/Embedded/Distributions.

[81] Linux.org. Linux Logos and Mascots, 1996.
http://www.linux.org/info/logos.html.

[82] Robert Love. My Linux kernel patches, 2001.
http://www.tech9.net/rml/linux.

[83] QNX Software Systems Ltd. QNX Neutrino RTOS, 2002.
http://www.qnx.com/resource.

[84] P. Mantegazza, E. Bianchi, L. Dozio, and S. Papacharalambous. Rtai: Real time application interface. *Linux Journal*, 72:1–1, April 2000.
http://noframes.linuxjournal.com/lj-issues/issue72/3838.html.

[85] The MathWorks. Simulink 5.0.1, 2002.
http://www.mathworks.com/products/simulink.

[86] Nicholas McGuire. Primary user documents, 2002.
http://www.rtlinux.at/documents.html.

[87] Nicholas McGuire. Rtlinux applications and enhancements, 2002.
http://www.rtlinux.at/applications.html.

[88] Microsoft. Windows CE .NET Home Page, 2002.
http://www.microsoft.com/windows/embedded/ce.net/default.asp.

[89] Synergy Microsystems. Home page, 2002.
http://www.synergymicro.com.

[90] Synergy Microsystems. Linux SMP overview page, 2002.
http://www.synergymicro.com/Software/Linux.html.

[91] Charles J Murray. Shifting opinion shakes footing of embedded Linux. *EE Times*, May 12, 2002.

[92] Intelligent Machine Division NIST: Manufacturing Engineering Laboratory. Publication List, 2002.
http://www.isd.mel.nist.gov/documents/publist.htm.

[93] James Norton and Clark Roundy. Real-time Linux - where is it now? Technical report, Real-Time Magazine, 1999.
http://www.realtime-info.be/magazine/99q3/altatech.pdf.

[94] Institute of Electrical and Electronic Engineers. IEEE Std 1003.1-2001, 2001.
http://www.opengroup.org/onlinepubs/007904975/xrat/contents.html.

[95] Frederick M. Proctor. Linux, Real-Time Linux, and IPC. *Dr. Dobb's Journal of Software Tools*, 24(11):32–39, November 1999. code as ZIP file; code as tgz file.

[96] Frederick M. Proctor. Measuring performance in real time Linux. In *Real Time Linux Workshop*, Milan, Italy, 2001. ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-2001/proc/k03-proctor.pdf.

[97] Frederick M Proctor, B Damazo, C Yang, and S Frechette. Open architectures for machine control. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, December 1993.

[98] Frederick M Proctor and William P Shackleford. Timing studies of real-time Linux for control. In *Proceedings of DETC 01 ASME 2001 Design Engineering Technical Conferences & Information in Engineering Conference*, Pittsburgh, PA, September 9–12 2001. ASME.
http://www.isd.mel.nist.gov/documents/publist.htm.

[99] Frederick M Proctor and William P Shackleford. Embedded real-time Linux for cable robot control. In *Proceedings of DETC'02 ASME 2002 Design Engineering Technical Conf. & Computers & Information in Engineering Conference*, Montreal, Canada, September 29–October 2 2002.
http://www.isd.mel.nist.gov/documents/publist.htm.

[100] ADEOS Project. Download page, 2002.
http://savannah.gnu.org/download/adeos.

[101] ADEOS Project. Home page, 2002.
http://www.opersys.com/adeos.

[102] QRTS. Quality Real Time Systems, 2002.
http://qrts.com.

[103] QRTS. Real time Linux target: A simulink-based real-time control-design environment, 2002.
http://qrts.com/products/rtlt_linux/index.shtml.

[104] Jim Ready. Embedded systems conference: Keynote, 2000.
http://www.linuxdevices.com/articles/AT2306163492.html.

[105] REDSonic. Home page, 2002.
http://www.redsonic.com.

[106] Rafeeq Ur Rehman. Open source real-time operating systems. *Sys Admin: The Journal for UNIX Systems Administrators*, 10(1):91–95, January 2001.
http://www.samag.com/documents/s=1155/sam0101k/0101k.htm.

[107] Steve Rosenbluth, Michael Babcock, and David Barrington Holt. Controlling creatures with Linux. *Linux Journal*, 103:28–35, November 2002.
http://www.linuxdevices.com/articles/AT6435065918.html.

[108] David Schleef. Writing a Real-Time Compatible Device Driver for Linux: Comedi. Technical report, Real Time Linux Foundation, 1999.
ftp://ftp.realtimelinuxfoundation.org/pub/events/rtlws-1999/p16-schleef.pdf.

[109] Paul G. Schreier. Interfacing DA Hardware To Linux. Technical report, United Electronic Industries, 2001.
http://www.evaluationengineering.com/archive/articles/0102data.htm.

[110] Doctor Scilab. Scicos Home Page, 2002.
http://www-rocq.inria.fr/scilab/doc/scicos_html.

[111] Benno Senoner. Periodic Jitter, 2000.
http://www.linuxdj.com/latency-graph.

[112] William P Shackleford and Frederick M Proctor. Use of open source distribution for a machine tool controller. In *Proceedings of the International Society for Optical Engineering (SPIE) Photonics East Conference*, Boston, MA, November 5–8 2000. SPIE.
http://www.isd.mel.nist.gov/documents/publist.htm.

[113] Matt Sherer. Rtlinux application development tutorial, 2001.
http://www.linuxjournal.com/article.php?sid=5694.

[114] Yves Sorel. The AAAMethodology and SynDEx, 2002.
http://www-rocq.inria.fr/syndex.

[115] Richard Stallman. GNU's Not Unix!, 2002.
http://www.fsf.org.

[116] stm.lbl.gov. comedi: linux control and measurement device interface, 2000.
http://stm.lbl.gov/comedi/intro.html.

[117] Oncore Systems. Home page, 2002.
http://www.oncoresystems.com.

[118] Wind River Systems. pSOSystem 3 Home Page, 2002.
http://www.windriver.com/products/psosystem_3/index.html.

[119] Wind River Systems. VxWorks 5.x, 2002.
http://www.windriver.com/products/vxworks5/index.html.

[120] Andrew Tannenbaum. MINIX Information Sheet, 1987–1996.
http://www.cs.vu.nl/ ast/minix.html.

[121] A. Terrasa, A. Espinosa, and A. García-Fornes. Extending RT-Linux to support flexible hard real-time systems with optional components. *Lecture Notes in Computer Science*, 1474:41–48, 1998.
http://link.springer-ny.com/link/service/series/0558/papers/1474/14740041.pdf.

[122] Martin Timmeman. RTOS Market Survey Preliminary Results, 1999.
http://www.realtime-info.be/magazine/99q1/1999q1_p006.pdf.

[123] Linus Torvalds. Linux history, 1991.
http://www.li.org/linuxhistory.php.

[124] Linus Torvalds. The Linux kernel archives, 2002.
http://www.kernel.org.

[125] Carnegie Mellon University. Linux/RK, Original Development, 1997.
http://www-2.cs.cmu.edu/ rajkumar/linux-rk.html.

[126] Kansas University. KURT: POSIX project, 1997.
http://hegel.ittc.ukans.edu/projects/posix.

[127] Kansas University. KURT: Real Time Linux, 1997.
http://www.ittc.ku.edu/kurt.

[128] Borland USA. JBuilder download page, 2002.
http://www.borland.com/products/downloads/download_jbuilder.html.

[129] Bill Weinberg and Claes Lundholm. Embedded Linux - ready for real-time. Technical report, MontaVista Software, Inc., 2001.
http://www.mvista.com/dswp/RTReady.pdf.

[130] Phil Wilshire. Real time Linux: Testing and evaluation. In *Real Time Linux Workshop*, Orlando, FL, 2000. ftp://ftp.thinkingnerds.com/pub/projects/p-a03_wilshire.pdf.

[131] C. Wayne Wright and Edward J. Walsh. Scanning radar altimeter. *Linux Journal*, 58:1–1, February 1999.
http://lidar.wff.nasa.gov/sra/lj98.

[132] WSTAWG. Weapons Systems Technical Architecture Working Group, 2002.
http://wstawg.army.mil.

[133] Curt Wuollet. MatPLC, 2002.
http://mat.sourceforge.net.

[134] Peter Wurmsdobler. Real-time Linux based projects, 2002.
http://www.realtimelinuxfoundation.org/projects/projects.html.

[135] Peter Wurmsdobler and Nicholas McGuire. MiniRTL: A minimal real-time Linux. *Dr. Dobb's Journal of Software Tools*, 25(12):48, 50, 52, 54, December 2000.
http://www.ddj.com/ftp/2000/2000_12/minirtl.txt.

[136] Karim Yaghmour. Adaptive Domain Environment for Operating Systems. Technical report, OperSys.com, 2001.
http://opersys.com/ftp/pub/Adeos/adeos.pdf.

[137] Karim Yaghmour. Tracing real-time application interface using the Linux Trace Toolkit. *Embedded Linux Journal*, 2:39–40, 42, March/April 2001.
http://embedded.linuxjournal.com/magazine/issue02.

[138] Karim Yaghmour. The Linux Trace Toolkit, 2002.
http://opersys.com/LTT.

[139] Victor Yodaiken. US5995745: Adding real-time support to general purpose operating systems, 1999.
http://www.delphion.com/details?pn=US05995745__.

[140] Victor Yodaiken. FSMLabs Lean POSIX for RTLinux. Technical report, FSMLabs Inc, 2001.
http://www.fsmlabs.com/developers/white_papers/posix.htm.

[141] Victor Yodaiken, Pierre Cloutier, David Schleef, Philip N. Daly, Ragunathan Rajkumar, and Bernhard Kuhn. Development of RTOSes and the position of Linux in the RTOS and embedded market. In *Proceedings of the 21stSymposium on Real-Time Systems (RSS-00)*, pages 8–8, Los Alamitos, CA, November 27–30 2000. IEEE Computer Society.

Index