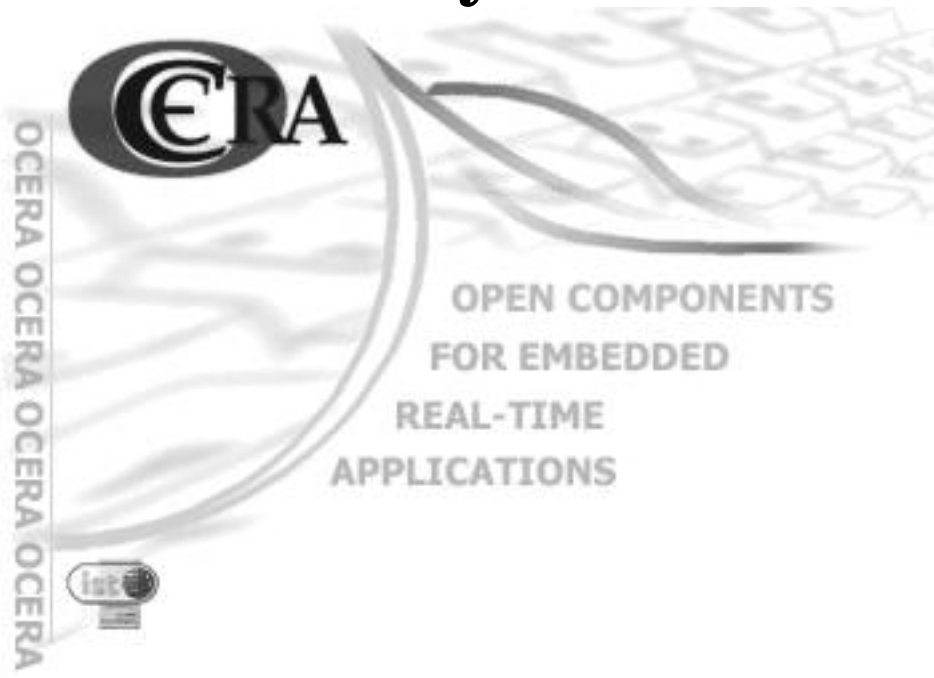
WP1 - RTOS State of the Art Analysis



Deliverable D1.1 - RTOS Analysis

WP1 - RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis

by Ismael Ripoll, Pavel Pisa, Luca Abeni, Paolo Gai, Agnes Lanusse, Sergio Saez, and Bruno Privat

Table of Contents

List of Tables

List of Figures

Document presentation

Table 1. Project Co-ordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14. CP: 46022, Valencia, Spain
Phone:	+34 9877576
Fax:	+34 9877579
E-mail:	alfons@disca.upv.es

Table 2. Participant List

Role	Id.	Name	Acronym	Country
CO	1	Universidad Politécnica de Valencia	UPVLC	E
CR	2	Scuola Superiore S. Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA	CEA	FR
CR	5	UNICONTROLS	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	VISUAL TOOLS S.A.	VT	E

Table 3. Document Version

Release	Date	Reason of change
1.0	November,	First Release

I. Introduction

Table of Contents

Chapter 1. Introduction

The objective of this workpackage is to make a study of the state of the art of real-time technology which is made available by the research community, and to determine what types of mechanisms actually turn out to be most useful for real-time applications. In concrete, this workpackage will analyse the real-time operating systems (RTOS) features and extract the main characteristics that will be included in the OCERA development.

The project is to produce and integrate prototypes of various innovative real-time techniques, research areas of particular interest include scheduling, resource management, fault-tolerance and communication with real-time constraints, which are identified as the key elements to provide predictable and high performance distributed real-time operating systems.

RTOS is a generic term for a set of operating systems that provide support for real-time applications. There is a wide range of RTOS, from the small and simple enough to fit in a few kilobytes of memory that can run on simple processors, to the high-end range RTOS that provides full graphical user interface that require several megabytes of RAM and powerful processors (MMU, protected mode, etc.).

Originally, Linux was designed to be used in a server or desktop environment. Since then, Linux has evolved and grow to be used in almost all the computer areas, among others, in embedded systems, parallel clusters, realtime systems, etc.

There is a large group of researchers, hackers and companies adding realtime capabilities: reducing the memory requirement, porting Linux to embedded processors, improving the response time, etc. Real Time Linux is just another piece of software developed in a Open-Source development methodology. There is not a single company or research group that concentrates all the development of realtime Linux, but a set of not connected, overlapped or even rival implementations are taking place simultaneously.

The two main contributions of this working package are:

1. A list of features that are available in commercial RTOS.
2. A detailed description of the real-time features already implemented in Linux. Features that are included in the Linux kernel by default or that are distributed separately.

To achieve these goals, we have focused our study in a small (compared with the large amount to existing) number of RTOS: VxWorks, QNX, RTEMPS and LynxOS. And also the main Linux kernel and all the extensions to improve the real-time capabilities of Linux like RTLinux, RTAI and the preemptable patch are analysed and compared in this paper.

The rest of this deliverable is organised as follows: In the first part a summary of the POSIX and OSEK standard followed by the list of main features we are interested and are taken into consideration in the RTOS analysis. Several summary tables are presented in the last section of the fist part. In the second part of the deliverable a study of each RTOS is presented.

The conclusions of this deliverable can be found in the deliverable D3.2 "Functionality Not Available in Open RTOS" of the Workpackage 3 "Market Analysis".

Chapter 2. POSIX Standard

2.1. Introduction

POSIX, that stands for Portable Operating System Interface, is a standard that is being jointly developed by the IEEE and The Open Group. It defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level. The current revision of POSIX is *The Open Group Base Specifications Issue 6* and also the *IEEE Std 1003.1-2001*.

This standard is composed by four major components:

- *Base Definitions*: This include general terms, concepts and interfaces common to entire standard.
- *System Interfaces*: This comprises the definitions for system service functions for the C programming language, function and portability issues, error handling and recovery.
- *Shell and Utilities*: It contains the definitions for a standard source code-level interface to command interpretation services.
- *Rationale*: It contains information that does not fit well into the rest of the document structure.

The IEEE Std 1003.1-2001 standard is a single common revision to IEEE Std 1003.1-1996, IEEE Std 1003.2-1992, and the Base Specifications of The Open Group Single UNIX Specification, Version 2. In order to develope the current revision several base documents has been used. The base documents that are involved in the definition of system interfaces are:

- IEEE Std 1003.1-1996 (POSIX-1) (incorporating IEEE Stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995)
- The following amendments to the POSIX.1-1990 standard:
  - IEEE P1003.1a draft standard (Additional System Services)
  - IEEE Std 1003.1d-1999 (Additional Realtime Extensions)
  - IEEE Std 1003.1g-2000 (Protocol-Independent Interfaces (PII))
  - IEEE Std 1003.1j-2000 (Advanced Realtime Extensions)
  - IEEE Std 1003.1q-2000 (Tracing)
- Open Group Technical Standard, February 1997, System Interface Definitions, Issue 5 (XBD5)
- Open Group Technical Standard, February 1997, System Interfaces and Headers, Issue 5 (XSH5)
- Open Group Technical Standard, January 2000, Networking Services, Issue 5.2 (XNS5.2)
- ISO/IEC 9899:1999, Programming Languages - C.

As it can be observed this standard includes support for source portability of applications with realtime requirements, but this support is maintly optional for POSIX-conforming implementations. The specific functional areas included for realtime support and their definitions[1] are basically the following:

- Semaphores.

  A minimum synchronization primitive to serve as a basis for more complex synchronization mechanisms to be defined by the application program.

- Process Memory Locking.

  A performance improvement facility to bind application programs into the high-performance random access memory of a computer system. This avoids potential latencies introduced by the operating system in storing parts of a program that were not recently referenced on secondary memory devices.

- Memory Mapped Files.

  A facility to allow applications to access files as part of the address space.

- Shared Memory Objects

  An object that represents memory that can be mapped concurrently into the address space of more than one process.

- Priority Scheduling.

  A performance and determinism improvement facility to allow applications to determine the order in which threads that are ready to run are granted access to processor resources.

- Realtime Signal Extension.

  A determinism improvement facility to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signal functions.

- Timers.

  A mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed.

- Interprocess Communication.

  A functionality enhancement to add a high-performance, deterministic interprocess communication facility for local communication.

- Synchronized Input and Output.

  A determinism and robustness improvement mechanism to enhance the data input and output mechanisms, so that an application can ensure that the data being manipulated is physically present on secondary mass storage devices.

- Asynchronous Input and Output.

  A functionality enhancement to allow an application process to queue data input and output commands with asynchronous notification of completion.

Another optional support that can be interesting for the developemnt of embedded applications is the *Threads* support. This extension to POSIX defines functionality to support multiple flows of control within a process. These *flows of control* are called threads and they share their address space and most of the resources and attributes defined in the operating system for the owner process.

The specific functional areas included in threads support are:

- Thread management: the creation, control, and termination of multiple flows of control that share a common address space.
- Synchronization primitives optimized for tightly coupled operation of multiple control flows in a common, shared address space.

Finally, the IEEE Std 1003.1-2001 standard also proposes a set of tracing facilities that can be quite useful at the development stage of a embedded real-time application. The tracing facilities defined in the standard allow a process to select a set of trace event types, to activate a trace stream of the selected trace events as they occur in the flow of execution, and to retrieve the recorded trace events. A trace event is a data object that represents an action executed by the system, and that is recorded in a trace stream. The trace events can be retrieved later from the trace stream, allowing the system behaviour analysis.

All these functionalities are not mandotory in a POSIX-conforming implementation, but defined as a set of options that may be supported by that system. In this line, the IEEE Std 1003.1-2001 standard defines several *XSI extensions* that groups together several of these options in so-called *XSI Option Groups*. The option groups that can be of interest for embedded real-time applications are described in the next section.

2.2. XSI System Interfaces and Extensions

The X/Open System Interface is the core application programming interface for systems conforming to the Single UNIX Specification. This is a superset of the mandatory requirements for conformance to IEEE Std 1003.1-2001.

A system that wants to be a XSI-conforming implementation shall meet the criteria for POSIX conformance and support all functions and headers defined in IEEE Std 1003.1-2001 as part of the XSI extension. Additionally, it shall support the following options defined in the standard: File Synchronization, Memory Mapped Files, Memory Protection, Threads, Thread Process-Shared Synchronization, Thread Stack Address Attribute and Thread Stack Address Size.

Despite of these options that are mandatory to be a XSI-conforming system, the system may also support one or more of the following *XSI Option Groups*: *Encryption*, *Realtime*, *Advanced Realtime*, *Realtime Threads*, *Advanced Realtime Threads*, *Tracing*, *XSI STREAMS* and *Legacy*. All the *realtime* option groups jointly with the tracing option group are clearly of great interest for developing embedded real-time applications. The options each option groups requires are detailed next:

Realtime

The options of IEEE Std 1003.1-2001 that are grouped together in this option group are: Asynchronous, Synchronized and Prioritized Input and Output, File Synchronization, Memory Mapped Files, Shared Memory Objects, Process and Range Memory Locking, Memory Protection, Semaphores, Timers, Realtime Signals Extension, Message Passing and Process Scheduling.

Advanced Realtime

The options of the standard that are grouped together in the *Advanced Realtime* option group are: Advisory Information, Clock Selection, Process CPU-Time Clocks, Monotonic Clock, Timeouts, Typed Memory Objects, Spawn and Process Sporadic Server.

Realtime Threads

This option group includes the following options: Thread Priority Inheritance and Protection, and Thread Execution Scheduling.

Advanced Realtime Threads

The *Advanced Realtime Threads* option group requires the following POSIX options: Thread CPU-Time Clocks, Thread Sporadic Server, Spin Locks and Barriers.

Tracing

This option group includes the following tracing facility options: Trace, Trace Event Filter, Trace Inherit and Trace Log.

Notes

1. This definitionas are extracted from the IEEE Std 1003.1-2001 standard.

Chapter 3. OSEK/VDX

3.1. Introduction

OSEK/VDX [1] is a joint project of the automotive industry that aims to the definition of an industry standard for an open-ended architecture for distributed control units in vehicles.

The objective of the standard is to describe an environment which supports efficient utilization of resources for automotive control unit application software. This standard can be viewed as a set of API for real-time operating system (OSEK) integrated on a network management system (VDX) that together describes the characteristics of a distributed environment that can be used for developing automotive applications.

The typical applications that have to be implemented have tight real-time constraints and an high criticality (for example, a power-train application). Moreover, these applications have to be made in a huge number of unit, therefore there is a need to reduce the memory footprint to a minimum enhancing as possible the OS performance.

Here are some keywords that helps to better characterize the philosophy that drove the main architectural choices of the OSEK Operating System:

Scalability.

The operating system is intended for use on a wide range control units (either system with minimal hardware resources like RAM, ROM, CPU time, i.e. 8 bit microcontrollers). To support a wide range of systems the standard defines four conformance classes that tightly specifies the main features of an OS. Note that memory protection is not supported at all.

Portability of software.

The standard specifies an ISO/ANSI-C interface between the application and the operating system that is identical in all the implementations of the OS. The aim of this interface is to give the ability to transfer an application software from one ECU to another ECU without bigger changes inside the application. Due to the wide variety of hardware where the OS has to work in, the standard does not specify any interface for the Input/Output subsystem. Note that this fact reduces (if not prohibits) the portability of the application source code, since the I/O system is one of the main software part that impacts on the architecture of the software. We can say that the prime focus is not to achieve 100% compatibility between the application modules, but to ease their direct portability between compliant operating systems.

Configurability.

Another prerequisite needed to adapt the OS to a wide range of hardware is a high degree of modularity and configurability. This configurability is reflected by the toolchain proposed by the OSEK standard, where some configuration tools help the designer in tuning the system services and the system footprint. Moreover, a language called OIL (OSEK Implementation Language) is proposed to help the definition of a standardized configuration information.

Statically allocated OS.

All the OS objects and features are statically allocated. This fact allow to simplify all the OS: the number of application tasks, resources and services requested are defined at compile time. Note that this approach ease the implementation of an OS capable of running on ROM, and moreover it is completely different from a dynamic approach followed in other OS standards like for example POSIX.

Support for time triggered architectures.

The OSEK Standard provides the specification of OSEKTime OS, a time triggered OS that can be fully integrated in the OSEK/VDX framework.

In the following sections the main features of the OSEK/VDX standard will be analyzed in detail.

3.2. Architecture of the OSEK/VDX operating system

The architecture on which an OSEK Operating System is based can be viewed as a traditional fixed priority approach.

Each task in the system can be a basic task (BT) or an extended task (ET) (extended tasks are basic tasks that can react to external asynchronous events).

Every task in the system has assigned a fixed priority (statically assigned at compile time), and the scheduler always selects the higher priority task from the ready task queue. Interrupt service routines typically preempt the running task (except in case the running task uses resources).

To provide support for different features in the Operating system, the various requirements of the application in terms of number of tasks, memory consumption and like are listed in four conformance classes. The compliance of an OSEK OS is always stated with respect to one conformance class. Basically, conformance classes exist to allow partial implementations of the standard along pre-defined lines, creating an upgrade path from classes of lesser functionality to classes of higher functionality with no change to the application tasks.

The conformance classes specifies different requirements for the following attributes:

- Multiple requesting task activations (only one activation or more than one)
- Task types (basic tasks only or basic and extended tasks)
- Number of tasks per priority (one or more than one)

The following conformance classes are defined by the standard :

BCC1

Only basic tasks limited to one activation request per task and one task per priority, while all tasks have different priorities.

BCC2

Like BCC1, plus more than one activation request per task and more than one task per priority.

ECC1

Like BCC1, plus extended tasks.

ECC2

Like ECC1, plus more than one task per priority and multiple requesting of task activation allowed for basic tasks.

3.3. Task Management

In the OSEK OS, a task provides the framework for the concurrent and asynchronous execution of functions. The Scheduler is then responsible for scheduling tasks following a well defined scheduling algorithm.

The OSEK operating system provides two kind of tasks: basic tasks and extended tasks. The only difference between the two concepts is that extended tasks are allowed to use the operating system call `WaitEvent()`. Basically that call allow an extended task to release the CPU waiting for an asynchronous event without terminating the current instance.

Each task in the system has assigned a fixed priority (statically assigned at compile time; the value 0 is defined as the lowest priority of a task), and it can be preemptive or non-preemptive. If the running task is preemptive the scheduler always made a preemption when needed, otherwise it reschedules the system at the end of the running task instance. A preemptive task can disable preemption for a while locking a resource called `RES_SCHEDULER`.

In any moment of its life a task is characterized by its state. The OSEK standard defines four task states:

running

In the running state, the CPU is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any point in time, while all the other states can be adopted simultaneously by several tasks.

ready

All functional prerequisites for a transition into the running state exist, and the task only waits for allocation of the processor. The scheduler decides which ready task is executed next.

waiting

A task cannot continue execution because it has to wait for at least one event. Only Extended tasks can jump into this state (because they are the only that can use events).

suspended

In the suspended state the task is passive and can be activated.

Note that basic tasks have no waiting state: a basic task can only represent a synchronization point at the beginning and at the end of the task. Application parts with internal synchronization points have to be implemented by more than one basic task. An advantage of extended tasks is that they can handle a coherent job in a single task, no matter which synchronization requests are active. Whenever current information for further processing is missing, the extended task switches over into the waiting state. It exits this state whenever corresponding events signal the receipt or the update of the desired data or events.

Depending on the conformance class a basic task can be activated once or multiple times. The latter means that an activation issued when a task is not in the suspended state will be recorded and then executed when the task will finish the current instance.

The termination of a task instance only happen when a task terminates itself (to simplify the OS, no explicit task kill primitives are provided).

3.4. Application modes and system startup

The OSEK Operating system gives a support for Application Modes. In real applications, an embedded system may execute different applications in a mutually exclusive way (for example, the normal operation, a factory test, and like). The application mode is a means to structure the software running in the system according to those different conditions and are a clean mechanism for development of totally separate systems. Once the operating system has been started, it is not allowed to change the application mode.

Typically each application mode uses its own subset of tasks, ISRs, alarms and timing conditions, although if some kind of sharing between modes is possible.

The start up performance is another safety critical issue for embedded system in automotive applications since reset conditions may occur during normal operation (for example, a power-train application should be capable of rebooting the whole system in a few microseconds, because the system must safely control the spark on the engine cylinders). The system startup is completely left to the particular implementation, although if some hint is given on how design the boot-up sequence. In any case the standard suggest the avoidance of lengthy or complicated starting procedures.

3.5. Interrupt processing

Since the standard must be suitable for different microcontrollers, the specification of interrupt handling routines only cover the general approach that a compliant OS should follow, without coping with any hardware related issues.

In particular, the standard provides two kind of ISR handlers:

ISR category 1

: The ISR does not use an operating system service. In practice, the OS does not handle these interrupts, and the designer is free to write his handler, with the only restriction that he can not call any OS service. Typically, these are the fastest highest priority interrupts.

ISR category 2

: The ISR is handled by the system, so OS calls can be called from the handler.

Inside any ISR no rescheduling will take place. Rescheduling takes place on termination of the ISR category 2 if a preemptable task has been interrupted and if no other interrupt is active. At the end of the ISR category 1 no rescheduling takes place too, and this is the reason because ISR category 1 should have the highest priority in a correct design.

3.6. Events

The event mechanism is only provided for extended tasks and can be used to communicate binary information that synchronize these tasks on asynchronous events. Each extended task owns a set of events, that can be triggered by other (basic and extended) tasks or by ISR of category 2.

The typical behavior of an extended task is to wait for asynchronous events calling the OS service `WaitEvent()`. This service usually blocks the task until an event arrives. After servicing the event, the task calls again `WaitEvent()` to wait other events.

Events can be set only if the task is not in the suspended state. This seems to suggest that an extended task should never be in the suspended state.

3.7. Scheduling

The scheduler decides on the basis of the task priority which is the next of the ready tasks to be transferred into the running state (dynamic priority management is not supported). Tasks on the same priority level are started depending on their order of activation.

The OSEK standard provides four flavors of fixed priority scheduling, outlined below:

Full Preemptive Scheduling

Full preemptive scheduling means that the running task may be rescheduled at any instruction by the arrival of high priority tasks.

Non Preemptive Scheduling

Non preemptive scheduling means that task switching is only performed via one of a selection of explicitly defined system services (like task termination, explicit call to the scheduler and arrival of an event that wakes up an extended task).

Mixed Preemptive Scheduling

Since preemptiveness is a task attribute, preemptive and non-preemptive tasks can be mixed in the same application. The running task will influence the policy really used.

Task Grouping Using Internal Resources

This scheduling policy is very similar to the preemption threshold technology, where threshold values are implemented using the OSEK Priority Ceiling protocol together with *internal resources* locked and unlocked at the start and at the end of every task instance.

3.8. Resource Management

The standard provides support for binary resources that can be used to implement critical sections. Priority inversion and deadlock are avoided using a variant of the SRP called *OSEK Priority Ceiling*.

The protocol in fact is a version of SRP adapted to fixed priority:

- every resource as assigned a ceiling that is the maximum priority of the tasks (and ISRs) that use the resource;
- when a task requires a resource, its current priority is raised to the ceiling of the resource;
- when a task releases a resource, the priority of this task is reset to the priority which was dynamically assigned before requiring that resource.

The normal properties of SRP applies to the protocol. In particular, Priority inversion, chained blocking and deadlocks are avoided. Moreover, there is no need for waiting queues, since a task can be scheduled only when all the resources it needs are free.

Resources are typically used by task only. In the OSEK standard, resources can be used either by a task or by an ISR of category 2. An ISR that use a resource can be thought as an high priority task: its execution can be delayed due to lower ISRs or tasks accessing resources with ceiling greater or equal than the IRS priority. This is the natural behavior in those systems where tasks activations and priorities are mapped on interrupts, and the raising of task's priorities is done with a proper programming of the interrupt controller.

The OSEK standard also provides a support for Preemption Thresholds through the use of *internal resources*. An internal resource is simply a resource that is locked when a task instance starts, and is unlocked when the task instance ends. The ceiling of the internal resources can be thought as the Preemption Threshold of the tasks.

In the same way the standard provides a special resource called `RES_SCHEDULER` that can be used to disable preemption. In practice, the `RES_SCHEDULER` is a resource with ceiling equal to the maximum priority in the system. In the same way a non preemptive task can be thought as a task that use an internal resource with the same ceiling of `RES_SCHEDULER`.

Finally note that, although a technique similar to Preemption Threshold is used, stack sharing between tasks of a same Non Preemption Group can not be exploited due of the OS calls `WaitEvent()` and `Schedule()` [2]. In fact, these calls releases the internal resource taken by a task, letting execution to more than one task in the same Non Preemption Group.

3.9. Miscellaneous

The OSEK operating system provides services for processing recurring events (for example, timers that provide an interrupt at regular intervals, or encoders at axles that generate an interrupt in case of a constant change of an angle). These events are recorded into implementation dependent counters, then used by software alarms. When an alarm (that can be one-shot or periodic) fires, a task can be activated, or an event can be set, or finally an alarm-callback routine can be called. Alarms and counters are statically defined at compile time. The only dynamic parameters that can be set are when an alarm has to expire and the period of a cyclic alarm.

To ease the tracing and the debugging of the system the OSEK standard provides system specific *hook routines* to allow user-defined actions within the OS internal processing. These hook routines are called by the operating system and they are composed by user code that is executed into an OS primitive, usually with ISR of category 2 disabled. These routines are only allowed to use a subset of API functions (mainly they can use functions for get internal OS states, to ease the tracing of the application). They are called at system startup, at system shutdown, before and after a preemption, and in case of an error. In particular, two different kinds of errors are distinguished:

Application errors

: The operating system could not execute the requested service correctly, but assumes the correctness of its internal data.

Fatal errors

: The operating system can no longer assume correctness of its internal data. In this case the operating system calls the centralized system shutdown.

The standard gives two ways of handling errors: a centralized way (using an *Error Hook* that is called every time an error occurs in a system primitive), and a decentralized way (where the application code must check itself for the correctness of the return value of every primitive).

3.10. OSEK COM

The OSEK standard comprises also an agreement on interfaces and protocols for in-vehicle communication called *OSEK COM*. The term in-vehicle communication means both communication between nodes and internal communication in a node of the whole vehicle. The basic idea is to provide a standardized API for software communication that is independed from the particular communication media used in a way to ease porting of applications between different hardwares.

The OSEK COM standard is composed by:

- An Interaction layer which provides communication services for the transfer of application messages.
- A Network layer which provides services for the unacknowledged and segmented transfer of application messages. The network layer provides flow control mechanisms

to enable interfacing of communication peers featuring different level of performance and capabilities.

- A Data link layer interface which provides services for the unacknowledged transfer of individual data packets over a network to the layers above.

OSEK COM provides a rich set of communication facilities but it is likely that many applications will only require a subset of this functionality. For that reason, the standard defines a set of conformance classes to enable the integration of OSEK COM in systems featuring various levels of capabilities in a scalable way, enabling the car producer to integrate software parts produced by different suppliers.

OSEK COM defines these levels as Communication Conformance Classes (CCCs). The main purpose of the conformance classes is to ensure that applications which have been for a particular conformance class are portable across different OSEK implementations and ECUs featuring that same or higher level of communication functionality. OSEK COM defines five communication conformance classes to provide support from ECU internal communication only (CCCA) up to inter-ECU external communication (CCC2).

For an OSEK implementation to be compliant, message handling for intra processor communication has to be offered. The minimum functionality required is CCCA as described in the OSEK COM specification.

3.11. OSEK NM

The OSEK standard also cover a standardization of basic and non-competitive infrastructure between the various embedded systems that can be present in a vehicle. In fact, very often electronic control units made by different manufacturers are networked within vehicles by serial data communication links.

For that reason the standard propose a Network Management system (OSEK NM) that provides standardised features which ensure the functionality of inter-networking by standardised interfaces.

The essential task of NM is to ensure the safety and the reliability of a communication network. This is obtained implementing access restriction to each node (access must be restricted only from authorized entities), keeping the whole network tolerant to faults, and implementing diagnostic features capable of monitoring the status of the network in an indirect (monitoring application messages) or in a direct way (monitoring by dedicated NM communication using token principle).

Moreover, the network management also cover the initialization of network resources, network configuration the co-ordination of global operation modes (e.g. network wide sleep mode), and a support for diagnosis.

3.12. OSEKTime

The OSEK Standard produced a specification for a Time Triggered Operating System (OSEKtime OS) that aims to represent a uniform functioning environment for single processor distributed embedded control units with a fault-tolerant communication layer.

The OSEKtime operating system supports static scheduling and offers all basic services for real-time applications, i.e., interrupt handling, dispatching, system time and clock synchronization, local message handling, and error detection mechanisms.

The OSEKtime operating system serves as a basis for application programs which are independent of each other, and provides their environment on a processor. The are two types of entities: interrupt services routines managed by the operating system and time triggered tasks.

The Osek Operating system can coexist with a OSEKTime OS, for handling both time triggered and event-driven computations on the same Embedded Control Unit. Basically, the interface of the OSEK OS remains the same (apart from some small changes in the startup/shutdown procedures). The main concept is that the OSEKTime OS assigns its idle time to the OSEK OS.

The OSEK OS, its tasks and its interrupts have always a lower priority than the similar entities in the OSEKTime OS. Non-preemptive tasks remains non-preemptive only in the OSEK/VDX domain, and they can be preempted bt the Time Triggered dispatcher and by the Time Triggered Tasks.

Notes

1. The term OSEK means “Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug” (Open systems and the corresponding interfaces for automotive electronics); the term VDX means Vehicle Distributed eXecutive. This chapter shortly describe the specification of the Operating System Specification, release 2.2, and recalls some other OSEK documents.
2. `Schedule()` is a point of rescheduling that can be used in both preemptive and non-preemptive tasks.

Chapter 4. Analyzed features

One of the first conclusions achieved in the first project meeting was then importance of the standards, and in particular the realtime extensions defined in the POSIX standard. POSIX is a mature, well developed, independent set of standards then are followed by most of the UNIX industry. Also the use of already existing standards in the open source community is a must.

OSEK is a RTOS specification designed to fulfil the requirement of the automotive industry. It was designed for systems with small hardware resources like 8 bit processes with no MMU. Although there are some ports of Linux to small processors like MC68000 family, Linux and RTLinux are specially designed for mid-range to high range processors. This is why we decided to select POSIX as the reference standard used in this analysis.

POSIX stands for stands for Portable Operating System Interface, and is an IEEE standard designed to facilitate application portability. It is the codification and standardisation of the common core of UNIX™ APIs, and realtime OS. A detailed list and status of all the sub-standards that are part of POSIX can be found in the PACS web page. The subset of POSIX standards that are related to real time or embedded systems are listed below:

- 1003.1b: Realtime Extensions (IEEE Approved)
- 1003.1c: Threads (IEEE Approved)
- 1003.1d: Additional Realtime Extensions (IEEE Approved)
- 1003.1j: Advanced Realtime Extensions (IEEE Approved)
- 1003.1q: Tracing (IEEE Approved)
- 1003.5: Ada binding to 1003.1 (IEEE Approved)
- 1003.5a: Ada Update (IEEE Reaffirmed)
- 1003.5b: Ada Realtime (IEEE Reaffirmed)

Following is the list of features that has been studied in every analyzed RTOS:

- *General overview and architecture*. The internal architecture of the RTOS determines the overall performance and the type of environments where it can be applied. For example, a RTOS with protected address space for processes will have longer context switches times than that with a flat memory space.
- Hardware support
  - *Processors supported*.
  - *Multiprocessor support*.
- Process management
  - *Weighted and/or lightweight processes*. Lightweight processes (threads) is an efficient method to implement concurrent systems due to the efficient resource management and fast context switching. The POSIX thread standard define many realtime features.
  - *Scheduling policy*. POSIX define four scheduling policies: `SCHED_FIFO`, `SCHED_RR`, `SCHED_SPORADIC` and `SCHED_OTHER`. What kind of scheduling policies provide for hard and soft tasks?

  - *Periodic tasks*. Most of the scheduling theory was developed to provide support to periodic tasks. POSIX do not provide specific system calls to implement periodic activities; it has to be implemented by mean of timers and signals. Do the system provide any method to implement periodic tasks?
  - *Range of priorities*. POSIX require at least 32 priorities.
  - *Thread creation and cancellation*. POSIX standard provides a lot of facilities to deal the concurrency problems related to the termination of the running tasks. Do the RTOS provide a way to start and terminate threads in a safe way?
- Memory management
  - *Memory protection*. This is a mayor issue in realtime, since it is a key feature to provide fault tolerance.
  - *Dynamic memory allocation*. Fixed block size memory allocation? do provide the standard `malloc()` and `free()` functions?
- Inter-process communication
  - *Semaphores*. The most common and versatile synchronisation primitive.
  - *Mutex*. Mutexes are a special type of semaphore used to protect critical sections. The main difference with normal semaphores is that the thread that locks the mutex becomes the owner of the mutex, and this thread is the only one that can unlock it. Which type of mutex are available (spin-lock, recursive, etc.)?
  - *Priority inversion control*. Do provide some type of priority inversion control in concurrency control primitives?
  - *Message queues*. Real time extensions of POSIX include a message queue interface.
  - *Mailboxes*
  - *Shared memory* Since normal weight processes are executed in a protected memory space and threads (lightweight processes) are executed in a single common memory space. This functionality is only applicable to systems with separate memory spaces.
  - *Signals (POSIX signals)*
- Time and timers
  - *Time facilities*. How the time is handled is one of the most important issues in a RTOS: Absolute/relative delays, time resolution, how timers are managed, configurable timer resolution, types of clocks, etc.
  - *Facilities to add new hardware timers*. Embedded boards are usually equipped with special timer and watchdog hardware. Do the RTOS provide some utility to use it?
- Driver programming (low level programming)
  - *Interrupts*. How interrupts are handled? What facilities provide the RTOS to write interrupt handlers?
  - *System facilities to do IO*. Embedded systems usually have some custom devices. The RTOS has to provide some facilities to allow the engineer to write its own device drivers.
- Quality of Service
- Network
  - *Available protocols*.
- Filesystems
  - *Filesystem support*.

  - *Reservation features*.

- Trace and debug
  - *Program debugging*. Debugging and trace utilities are fundamental features that a RTOS must provide. There are two characteristics that stress the need of these type of tools: on one hand, embedded systems use to have limited communication capabilities (sometime only a serial line); on the other hand realtime systems should contain no bugs, this is achieved by means of a careful design and extensive debugging.
  - *Event/timing debugging*. Timing correctness is as important as logical correctness in realtime systems. The RTOS must provide timing information about the execution of the application. Recently, POSIX.1q added tracing facilities that allow the user to trace system and user events.

- Miscelanea
  - *Graphic support*.
  - *Development environment*. Most RTOS, commercial and open source, use GNU development utilities: gcc, binutils, gdb, etc.
  - *Programming languages*. The "C" language is available in all the RTOS, but more and more other languages like ADA or C++ are also used.
  - *Compatibility with other RTOS APIS*. Like pSOS, VxWorks, OSEK, self propietary API.

Chapter 5. Summary

	Hardware	**Multi-processor**	**Scheduling**	**Concurrency**
Linux	Alpha, ARM, i386, MIPS, PowerPC, Sparc, SuperH, Etrax, m68k, PA-RISC	Yes	SCHED_FIFO, SCHED_RR, SCHED_OTHER	UNIX-processes & Pthreads
RTLinux	i386, PPC, ARM	Yes	SCHED_FIFO	Pthreads
RTAI	i386, MIPS, PPC, ARM, m68k-nommu	Yes	Fixed priority	Lightweight processes
RTEMS 4.5+	M68k, ColdFire, SuperH, i386, MIPS, i960, PowerPC, SPARC, AMD A29K. PA-RISC	Static allocation	SCHED_FIFO, SCHED_RR, SCHED_OTHER	Pthreads
QNX	I386, ARM, MIPS, PowerPC, SuperH	Yes	SCHED_FIFO, SCHED_RR, SCHED_OTHER	UNIX-processes, Lightweight processes, Pthreads
VxWorks 5.x	m68k/CPU32/ColdFire/PowerPC, i386, ARM, SuperH, MIPS	Optional	Fixed priority, Rodun-robin	Lightweight processes
LynxOS	i386, m68k, PowerPC, ARM, SPARC, PA-RISC	Yes	FIFO, Priority Quantum, Round-Robin, Non-preemptive	UNIX-processes & Pthreads

	Priorities lower- higher	**Memory protection**	**Dynamic memory**	**Inter-process communication**
Linux	(0-100)	Yes	Yes	Semaphores, Mutexes, Condition-var., shared-mem, signals, pipes.
RTLinux	(0-1000000)	No	No	Semaphores, Mutexes, Condition-var., FIFO
RTAI	(0x3fffFfff-0)	No	Yes	Semaphores, Mutexes, Condition-var., FIFO, Mailbox, shared-mem, net_rpc, Pqueues.
RTEMS 4.5+	(255-1) (254-1) Posix	No	Yes	Semaphores, Mutexes, Condition-var., Pqueues, Events.
QNX	(0-31)	Yes	Yes	Semaphores, Mutexes, Condition-var., Barriers, Atomic operations, rd/wr-locks, Pqueues, shared-mem, FIFO.
VxWorks 5.x	(255-0)	No	Optional	Semaphores, Mutexes, Message, RTSignals,

	Priorities lower- higher	**Memory protection**	**Dynamic memory**	**Inter-process communication**
LynxOS	(0-255)	Yes	Yes	Semaphores, Mutexes, shared-mem, Pqueues, signals, pipes, Condition-Var.

	Priority inversion control	**Time resolution**	**Timers**	**Low level programming**
Linux	None	Configurable (HighResTimers)	POSIX timers	No interrupt programming.
RTLinux	Immediate ceiling	Hardware dependant	None	Full control HW
RTAI	Inheritance	Hardware dependant	None	Full control HW
RTEMS 4.5+	Inheritance & Immediate ceiling	Hardware dependant	POSIX timers	Full control HW
QNX	Immediate ceiling	Configurable	POSIX timers	Interrupts can be handled by user processes.
VxWorks 5.x	Inheritance	Configurable	Watchdog timers, POSIX timers	Full control HW
LynxOS	Inheritance	Configurable	POSIX timers	POSIX-style threads of execution within the kernel for interrupt handling.

	QoS	**Network**	**Filesystem**
Linux	FIFO, CBQ, CSZ, ATM, PRIO, RED, SFQ, TLE, TBF, GRED, Diffserv, Ingress, RSVP	IP, UDP, TCP, SLIP, PPP, ICMP, DHCP, RARP, RARP, TFTP, RPC, FTP, HTTP	ReiserFS, ext2, ext3, NFS, CIFS, ADFS, FAT, VFAT, NTFS, CRAMFS, ISO9660, MINIX, QNX4, ROM, JFS, XFS, Flash
RTLinux	None	None	None
RTAI	None	IP, UDP	None
RTEMS 4.5+	None	IP, UDP, TCP, SLIP, PPP, ICMP, DHCP, RARP, TFTP, RPC, FTP, HTTP, CORBA	IMFS, DOSFS/FAT
QNX	None	IP, UDP, TCP, ARP, ICMP, IGMP, QNET	RAMFS, QNX4, DOS, ISO9660, ext2, NFS, CIFS, Flash

	QoS	Network	Filesystem
VxWorks 5.x	None	TCP/IP IP, UDP, TCP, IGMP, ICMP, ARP RIP 1/2 SLIP, CSLIP, PPP BOOTP, DNS, DHCP, TFTP FTP, RLOGIN, RSH, TELNET	FAT, NFS, raw, TrueFFS
LynxOS	None	IP, UDP, TCP, ICMP, IGMP, ARP, RARP, DHCP, NAT, RPC, NTPv3, Raw, Zebra routing, TFTP	Lynx Fast File system, ISO9660, NFS, RAMFS.

	Debug	Languages	API compatibilitiy
Linux	GDB, DDD, Insight, System debugg, LTT	C, C++, ADA, Java,	POSIX 1003.1, VxWorks, pSOS
RTLinux	Simple trace, GDB	C, C++	POSIX 1003.1c
RTAI	KGDB	C	Custom, POSIX 1003.1b
RTEMS 4.5+	GDB, DDD, Debug over: ethernet, serial, BDM	C, C++, ADA	RTEID/ORKID,uITRON 3.0, POSIX 1003.1b
QNX	GDB, memory overrruns.	C, C++, Java	POSIX 1003.1, POSIX 1003.1b
VxWorks 5.x	GDB, Debug over: ethernet, serial, WindView, Trace	C, C++	Propietary (VxWorks), POSIX 1003.1, POSIX 1003.1b
LynxOS	GDB, Insight, Debug over: ethernet, serial	C, C++, ADA	Propietary, POSIX.1/.1b/.1c, Unix BSD 4.3. ABI compatibility with Linux 2.4

II. RTOS Analysis

Table of Contents

Chapter 6. Realtime Support in Linux

Linux is a full-featured UNIX® implementation. The main design criteria of the Linux kernel is the throughput while realtime and predictability is not an issue. The main handicap to consider Linux as a realtime system is that the kernel is not preemptable; that is, while the processor executes kernel code, no other process or event can preempt kernel execution.

Although Linux is not a realtime system, it has some features, already included in the mainstream source code or distributed as patch files, designed to provide realtime to Linux. These are the features described in this section.

From the programmer point of view, there are two main programming paradigms to build a realtime application: weight-processes (normal UNIX processes) and lightweight-processes (known as threads or LWP). Linux provides support for both execution environments, mostly based on POSIX standards 1003.b and 1003.c respectively.

Most of the realtime extensions are not included into the standard "C" library "`libc`". These system calls are implemented and distributed in in the GNU "C" library (glibc) but are located in a separate library file called "`librt`". Therefore, to compile a program that makes use of realtime features, the compiler must be invoked flag the "`-lrt`".

6.1. Scheduling

From the very first versions of Linux, the scheduler was realtime POSIX compatible. It supports, among others, the fixed priority (`SCHED_FIFO`) policy, which is the base feature to build a realtime systems. It also provides the POSIX required: `SCHED_RR` and `SCHED_OTHER`. The range of priorities is [0..99].

A lot of work has been done to improve the performance of the scheduler through a careful design which yield to a new scheduler code and structure. Next stable Linux kernel (2.4.19), as well as unstable kernel development (2.5.x), will replace the old scheduler code with an improved “O(1) scheduler” developed by Ingo Molnar . This new scheduler is able to manage a large number of processes with no overhead degradation.

6.2. Virtual Memory

It is not possible to build realtime applications on a system with virtual memory. The random and long delays introduced when RAM is exhausted and swapping is required is intolerable in a realtime system. Linux provides the `mlock()` and `mlockall()` functions that disables paging for the specified range of memory, or for the whole process respectively. Therefore, all the “locked” memory will stay in RAM until the process exits or unlocks the memory.

`mlock()` and `mlockall()` are included in the POSIX realtime extensions.

6.3. Shared Memory

One of the main communications paradigms used in realtime applications is shared memory.

Linux processes can share memory with each other and with drivers the POSIX.1b call `mmap()`. This function can be used both, to map into main memory a regular file and to map shared memory objects. When multiple processes map the same memory object (or

file), they share access to the underlying data, which is a efficient way to communicate large amounts of data between processed.

Since version 2.4.x and glibc 2.2 (GNU "C" library), Linux provides open shared memory objects, which is part of the POSIX realtime extensions. This API has the following functions: `shm_open()` and `shm_unlink()`.

6.4. High Resolution POSIX Timers (HRT)

Current POSIX API defines two different timer facilities:

- BSD timers: `setitimer()` and `getitimer()` functions.
- IEEE 1003.1b REALTIME timers: `timer_create()`, `timer_settime()`, `timer_getoverrun()`, etc.

Linux provides the BSD POSIX timers with a timing resolution around 10ms, which is clearly not suitable of realtime applications. The Hight Resolution Timers (HRT) project, sponsored by MontaVista, provides microsecond resolution with lower overhead following the IEEE 1003.b POSIX API. It is distributed as a patch file which can be downloaded from Sourceforge. Current distribution works with Linux kernel 2.4.18.

This patch can provide high resolution timers with very low overhead because of two main design issues: the use of several timing and interrupt hardware sources (the old 8254, the Pentiun internal instruction TSC, and the ACPI[1] timers when available), and a clever data structure to maintain the timers.

The patch provides high resolution clocks: `CLOCK_REALTIME_HR` and `CLOCK_MONOTONIC_HR`. And the accompanying functions: `clock_settime()`, `clock_gettime()`, `clock_getres()`. Also the POSIX timers functions are implemented: `timer_create()`, `timer_delete()`, `timer_settime()`, `timer_gettime()` and `timer_getoverrun()`.

6.5. Realtime Signals

POSIX extended the signals generation and delivery to improve the realtime capabilities. Signals take an important role in realtime as the way to inform the processes of the occurrence of asynchronous events like high-resolution timer expiration, fast interprocess message arrival, asynchronous I/O completion and explicit signal delivery.

The main characteristics of this type of signals are:

- The range of realtime signals supported by Linux is form 32 (`SIGRTMIN`) to 63 (`SIGRTMAX`).
- Signals can deliver a small piece of data (an integer or a pointer) to the signal handler (signal-catching function).
- Signals that carry information are delivered in chronological FIFO order.
- It is possible to automatically create a thread in response to a signal.

Linux fully supports the POSIX realtime signals standard.

6.6. POSIX Asynchronous I/O

Asynchronous I/O (AIO) is the POSIX interface to provide high efficiency asynchronous I/O access. The standard way to access I/O devices (files, drivers, sockets, fifos, etc.)

defined by UNIX the `read()` `write()` blocking sequence, where a next file access is performed only when the previous request has been completed. AIO mechanism provides the ability to overlap application processing and I/O operations initiated by the application.

A process can start one or more IO requests to a single file or multiples files and continue its execution. Also, a single system call can start a sequence of I/O operation on one or several files, which reduces the overhead due to context switches.

There are two implementations available in Linux: the one provided at the library level by using non-aio system calls (included in the `glibc/librt` since version 2.1); and the kernel implementation first developed by SGI™ called KAIO (till linux kernel 2.4.0), and now the Linux-AIO which provides this functionality to newer kernels.

6.7. POSIX Threads

Current Linux implementation of POSIX threads (POSIX 1003.1c) is based on the work done by Xavier Leroy, known as LinuxThreads. LinuxThreads is now integrated in the `glibc`, and distributed as a part of the it. It provides kernel-level threads: threads are created with the `clone()` system call and all scheduling is done in the kernel. This kind of threading is defined as 1:1, i.e. each thread is mapped to a Linux process.

LinuxThreads implements most of the POSIX API: mutex, condition variables, cancellation, signals, timed calls, etc. The library also provides POSIX semaphores. Mutex do not implement any protocol to prevent priority inversion. Since threads are scheduled by the Linux scheduler, the scheduling policies are the same as in Linux: `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`.

6.8. Quality of Service

Nowadays, linux offers a sophisticated component for bandwidth management called *Traffic Control*. This component supports method for classifying, prioritising, and limiting both incoming and out-coming traffic. Therefore, linux can do the following list of things: limit bandwidth for certain computers, help to fairly share bandwidth, protect the Internet from abuses, restrict access, do routing based on user id, MAC address, source IP address ... and so on.

For working with this subsystem, the kernel versions 2.2.x has to be patched, but the versions 2.4.x and uppers implement directly this functioning.

Network subsystem overview

The following figure shows the network subsystem:

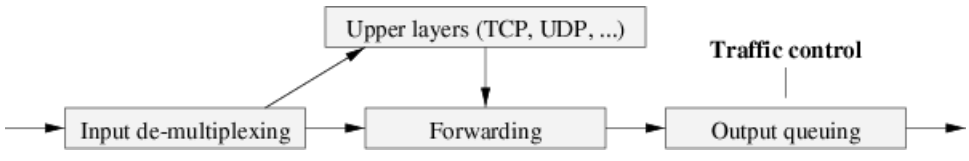


Figure 6-1. Network Subsystem

There are four components:

- *Input demultiplexing*: Decides if a incoming packets are passed to higher layers or are directly forwarded to the network.
- *Upper Layers*: Processes packets and may also generate new traffic and pass it to the lower layers.

- *Forwarding*: This layer performs the selection of the output interface, the selection of the next hop, encapsulation, etc.
- *Output Queueing or Traffic Control*: This is the most important component and decides if packets are queued or dropped, decides in which order packets are sent, etc.

Once the traffic control releases a packet for sending, the network device driver sends it to the network.

Traffic Control overview

The traffic control component consist of the following elements: *queueing disciplines* (qdisc), *classes* (within a queueing discipline), *filters* and *policing*

In this way, queueing discipline provides a method to enqueue a packet. A class is the place where packets are stored and processed in a specific way, afterwards, the qdisc selects the following packet for sending from classes. Filters are used by a qdisc to assign incoming packets to one of its classes. And finally, policing is used to ensure that incoming traffic does not exceed certain bounds.

The following picture illustrates an example of traffic control configuration:

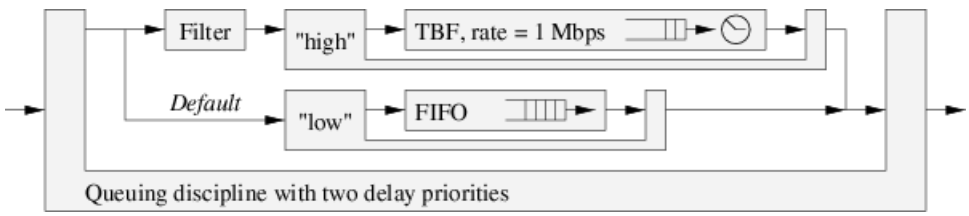


Figure 6-2. Traffic Control configuration

This configuration consists of a queuing discipline with two delay priorities, as well as, two classes: the higher class contains a token bucket filter discipline that limits the traffic, while the lower class contains a FIFO qdisc. Therefore, while the higher class has packets for sending (rate < 1Mbps), the priority qdisc selects a packets from this class. The filter decides which packets are sent to the higher class. Once a priority qdisc selects the following packet for sending, the network driver sends it on the network.

In conclusion, the traffic control layer decides whether the packets are queued or dropped, in which order the packets are sent, and finally it may delay packet transmission. Moreover, the traffic control elements can be combined in a modular way to support *Differentiated Service* (DS), *Integrated Service* (RSVP), *ATM* and so on.

The following four sections describe the traffic control elements.

Queueing discipline

Each network interface has a queue discipline attached with it, which controls how packets are enqueued and treated.

A qdisc is a black box, which is able to enqueue packets and dequeue them using its own algorithm, for example, a CBQ qdisc uses a WRR (Weight Round Robin) scheduling to select the following packet for sending on the network.

Moreover, qdisc are divided into two categories:

- *Classfull*: Qdiscs that may have child qdiscs attached to them.

- *Leafs*: Qdiscs that have not child's.

The available classfull qdiscs are:

- *PRIO* a n-band strict priority scheduler,
- *CBQ* Class Based Queue,
- *CSZ* Clak-Scott-Zhang,
- *ATM* Asynchronous Transfer Model,
- *DSMARK - DSCP* a Diff-Serv Code Point marker and
- *INGRESS*

The available leafs qdiscs are:

- *FIFO* a simple FIFO (it is the default qdisc),
- *TBF* Token Bucket Filter,
- *RED* Random Early Detection,
- *GRED* Generalised Random Early Detection,
- *TEQL* Traffic Equaliser and
- *SFQ* Stochastic Fair Queue.

Classes

A class is attached to a qdisc. However, queueing disciplines and classes are intimately tied together; the presence of classes and their semantics are fundamental properties of the queueing discipline. There is only one available class. This is the CBQ class. Note that a CBQ may work as queue discipline or class.

Filters

Filters are used to classify packets based on certain properties of them (address IP ...). The supported filters are:

- *rsvp* (use RSVP protocol for classification),
- *u32* (anything in the header may be used for classification),
- *fw* (use the firewall rules for classification),
- *route* (use routing table for classification decisions) and
- *tcindex* (use the DS field for classification).

Note that the u32 filter is the most advanced filter available and the tcindex filter is used in DiffServ (differentiation services).

Policing

The goal of policing is to ensure that traffic does not exceed certain bounds. There are four types of policing mechanisms: policing decisions by filter, refusal to en-queue a packet, dropping a packet from an inner queueing discipline and dropping a packet when en-queuing a new one.

6.9. Compatibility

Linux was developed around UNIX and POSIX standards. Therefore, its native API is mostly POSIX compatible.

Also MontaVista™ developed Wind River pSOS® and Wind River VxWorks® two emulation environments. These emulators are designed to ease the port of legacy RTOS code Linux. The emulation libraries are available at Sourceforge.

Notes

1. Advanced Configuration and Power Interface

Chapter 7. Low-Latency Patches for Linux

7.1. Kernel Latency

When scheduling a real-time user-level Linux process, the real-time performance can be affected by bottom halves (BHs) execution, kernel non-preemptable sections, and so on. The *kernel latency* is a quantity used to measure the difference between the theoretical schedule and the actual one.

The kernel latency is defined as follows: Let T be a real-time process that requires execution at time t, and let t' be the time at which T is actually scheduled; we define the kernel latency experienced by T as L = t' - t.

The biggest source of kernel latency are kernel non-preemptable sections (including BHs and Interrupt Service Routines - ISRs). In fact, non-preemptable sections in the kernel can prevent a high priority task from being scheduled for a very long time (up to 100ms). For example, if interrupts are disabled at time t, task T can only enter the ready queue later when interrupts are re-enabled. In addition, even if T enters the ready queue at the correct time t and has the highest real-time priority in the system, it may still not be scheduled if another task is running in the kernel in a non-preemptable section. In this case, T will be scheduled when the kernel exits the non-preemptable section at time t'.

The length of a kernel non-preemptable section depends on the strategy that the kernel uses to guarantee the consistency of its internal structures, and on the internal organization of the kernel. The standard Linux kernel is based on the classical *monolithic* structure, in which the consistency of kernel structures is enforced by allowing at most one execution flow in the kernel at any given time. This is achieved by disabling preemption when an execution flow enters the kernel, i.e., when a system call is invoked or when an interrupt fires. When a system call is invoked, the thread that invokes it enters in the kernel and becomes non-preemptable, returning preemptable when the execution exits from the kernel. When an interrupt fires, a short ISR is invoked with interrupts disabled: the ISR acknowledges the hardware interrupt and schedules a BH for execution. The BH will be executed in a non-preemptive way immediately before returning to user mode; hence, if the ISR interrupts a system call, the BH will be executed only *after* that the system call is completed (system calls can synchronize with ISRs by explicitly disabling interrupts). Summing up, in a standard Linux kernel, the maximum latency is equal to the maximum length of a system call plus the processing time of all the interrupts that fire before returning to user mode. Unfortunately, this value can be as large as 100ms.

7.2. Possible Solutions

Two different approaches can be used to reduce the size of kernel non-preemptable sections: the one used by the Low-Latency patches (Ingo Molnar and Andrew Morton)[LowLat], and the one used by the kernel preemptability patch (MontaVista, TimeSys)[kpreem, TimeSys].

7.2.1. Low-Latency Linux

This approach "corrects" the monolithic structure by inserting explicit rescheduling points (that effectively are preemption points) inside the kernel. In this approach, when a thread is executing inside the kernel it can explicitly decide to yield the CPU to some

other thread. In this way, the size of non-preemptable sections is reduced, thus decreasing the latency. In a low-latency kernel, the consistency of kernel data is enforced by using cooperative scheduling (instead of non-preemptive scheduling) when the execution flow enters the kernel.

This approach is also used by some real-time versions of Linux, such as RED Linux. In a low-latency kernel, the maximum latency decreases to the maximum time between two rescheduling points. Since the low-latency patch has been carefully hand-tuned for quite a long time, it performs surprisingly well.

7.2.2. Preemptable Linux

The preemptable approach, used in most real-time systems, removes the constraint of a single execution flow inside the kernel. Thus it is not necessary to disable preemption when an execution flow enters the kernel. To support full kernel preemptability, kernel data must be explicitly protected using mutexes or spinlocks. The Linux preemptable kernel patch, sponsored by MontaVista, uses this approach and makes the kernel fully preemptable. Kernel preemption is disabled only when a spinlock is held.

A similar approach is used by TimeSys, that uses mutexes instead of spinlocks, and provide priority inheritance. While the MontaVista patch disables preemption when a spinlock is acquired, the TimeSys one is based on blocking synchronization.

In a preemptable kernel, the maximum latency is determined by the maximum amount of time for which a spinlock is held inside the kernel. Again, it is important to note that BHs are serialized using a spinlock, thus they contribute to the latency.

An additional patch (lock-breaking) merges some of the low-latency rescheduling points into the preemptable kernel, for decreasing the amount of time for which spinlocks are held.

7.3. Evaluation

We measured the latency for the standard, Low-Latency, preemptable and preemptable with lock-breaking kernels while running different loads in background. All the experiments were performed using the Latency Benchmark tool downloadable from [FT]. The

results are shown in the following figures:

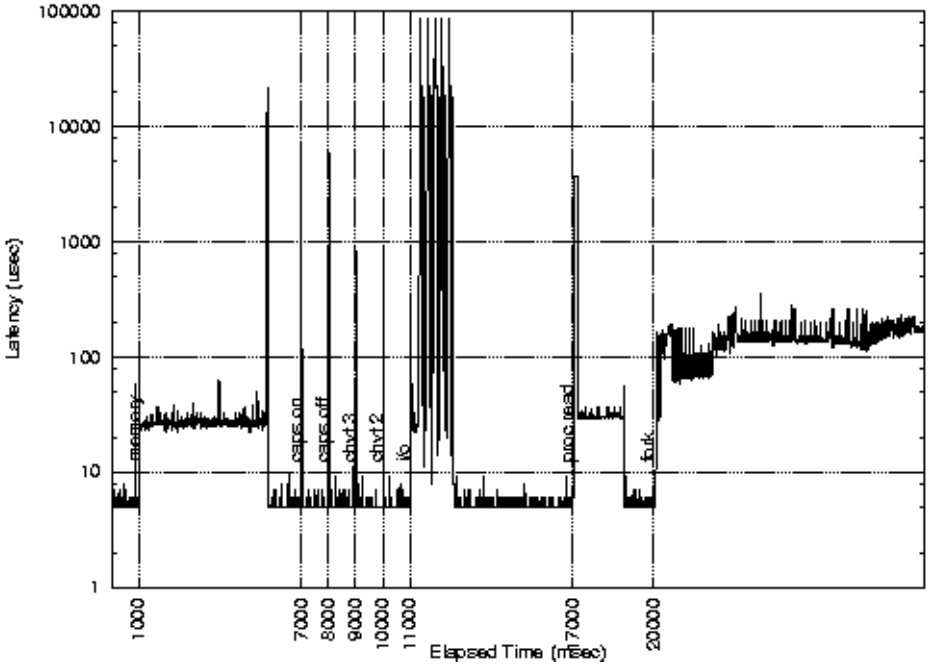


Figure 7-1. Latency in the Standard Kernel

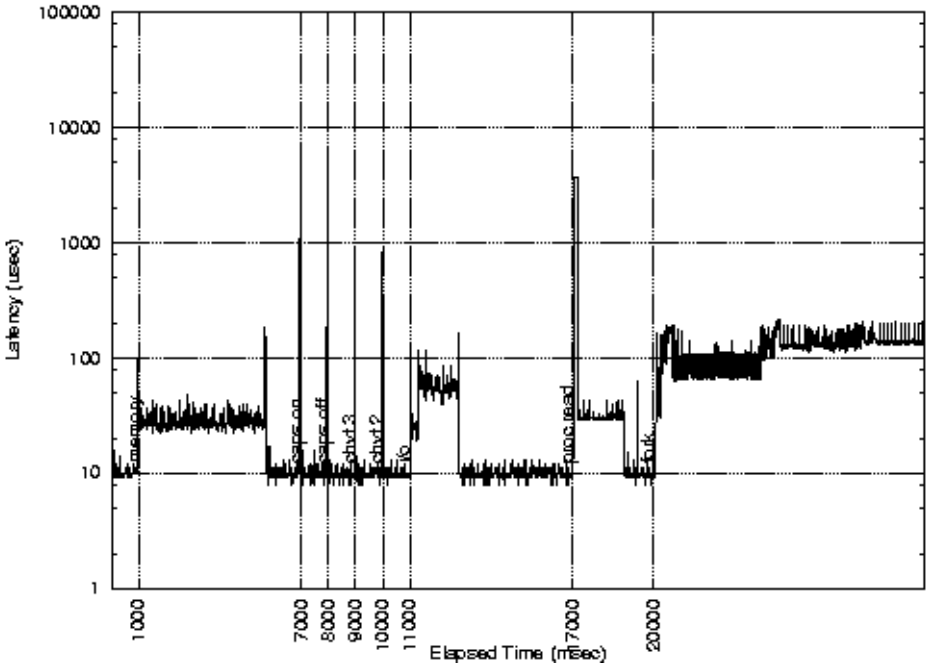


Figure 7-2. Latency in the Low Latency Kernel

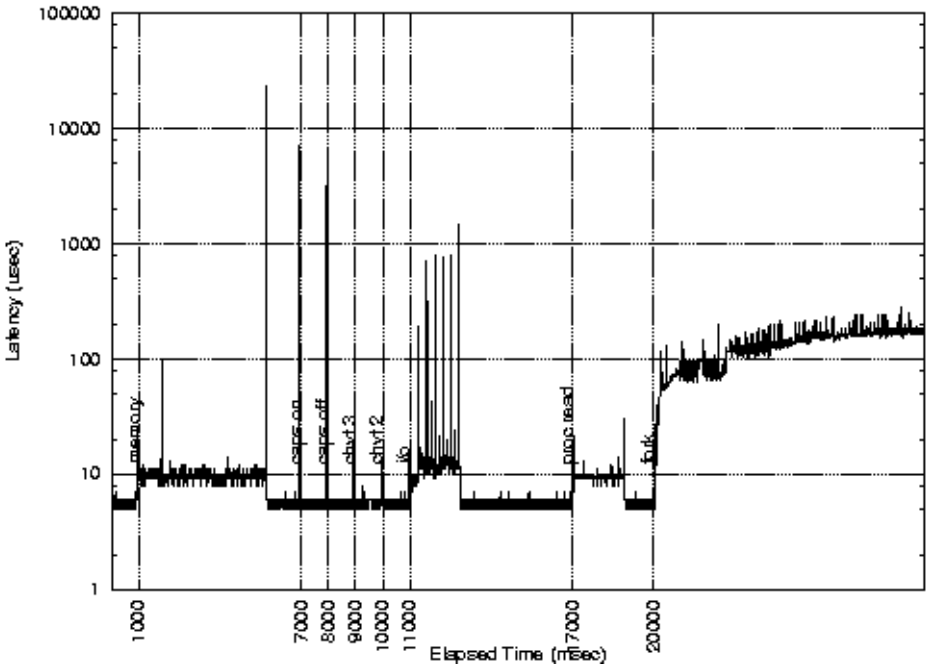


Figure 7-3. Latency in the Preemptable Kernel

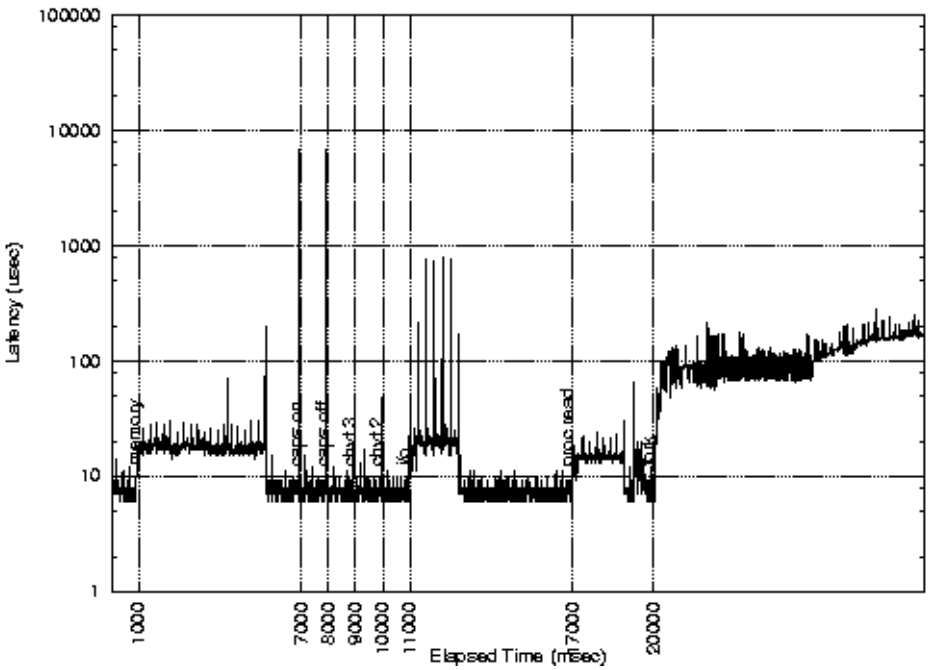


Figure 7-4. Latency in the Preemptable Lock-Breaking Kernel

From the first figure, it is possible to see that standard Linux exhibits high latencies at the end of the memory stress (a program that reads and writes a large array in memory), during the I/O stress (a program that reads and writes large amount of data on a file), when accessing the /proc file system, and when switching the caps/lock led. The large latency at the end of the memory stress is due to the munmap() system call. Comparing the figures it is possible to see that the low latency kernel solves all the problems except the /proc file system access and the caps/lock switch. On the other hand, the preemptable kernel eliminates the large latency in the /proc fs access, but does not solve the problem

with the memory stress, and is not as effective as the low latency kernel in reducing the latency during the I/O stress. Finally, the lock-breaking preemptable kernel seems to provide good real-time performance, but still has some problem during the I/O stress.

By repeating similar experiments for longer amounts of time, we verified that the low-latency kernel is characterized by larger *average latencies* respect to the preemptable and lock-breaking preemptable kernels, but reduces the *worst case* latencies. In other words, the tail of the probability distribution function is shorter for a low-latency kernel. As an example, the PDFs of the latency during the I/O stress are reported in the following figures (note that this new experiments were performed on a computer that is faster than the one used for the previous experiments).

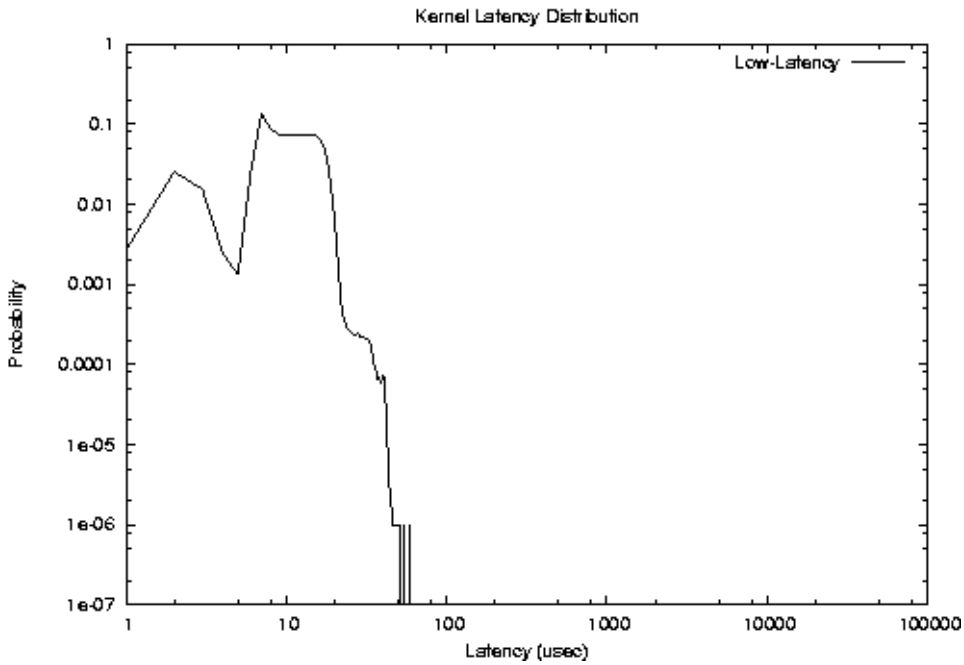


Figure 7-5. PDF of the Latency in the Low-Latency Kernel

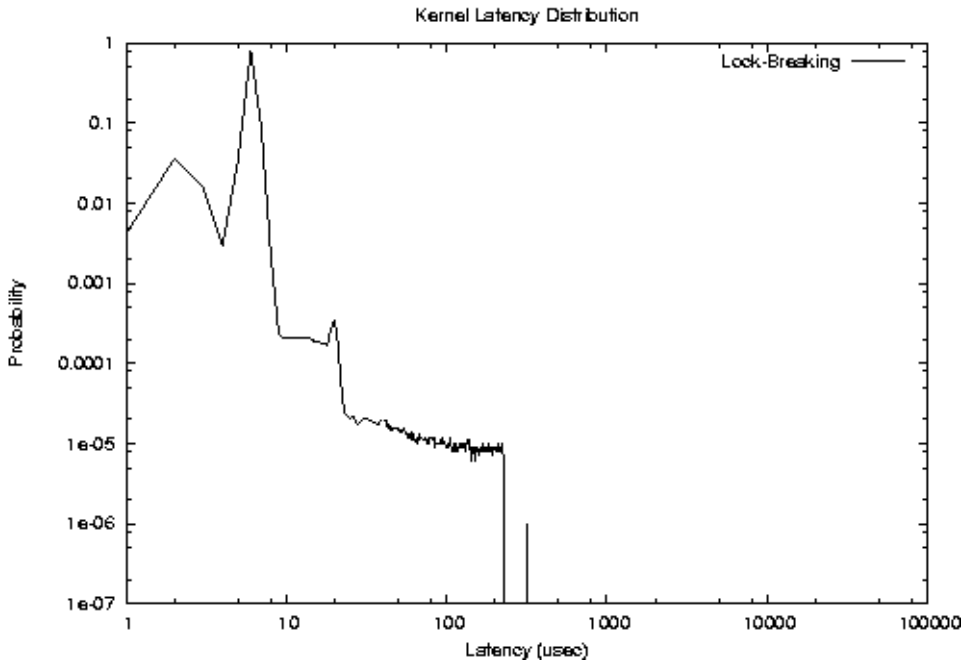


Figure 7-6. PDF of the Latency in the Lock-Breaking Kernel

According to the previous figures, it seems that the low-latency patch provides better real-time performance. This is probably due to the fact that the Linux kernel still contains “big spinlocks” that are held for long amounts of time, and that the lock-breaking patch does not “break” properly. A classical example is the spinlock used to serialize BHs. However, in the future Linux developers will likely decrease the size of kernel critical sections (to improve SMP performance), hence it seems to be reasonable to think that the latency of the preemptable and lock-breaking preemptable kernel will decrease.

Chapter 8. RTLinux/GPL

There are two different versions of RTLinux: RTLinux/Pro and RTLinux/Open.

RTLinux/Open is available on the web. The main (only) developer of this version was FSMLabs and stops its development in 2Q/2001. RTLinux/Pro is available for a free from FSMLabs and the license is non-GPL.

This paper is focused of the RTLinux/Open version.

8.1. Architecture overview

There are two approaches to provide real-time performance in a Linux system:

1. Improving the Linux kernel preemption.
2. Adding a new software layer beneath Linux kernel with full control of interrupts and processor key features.

These two approaches are known as "*preemption improvement*" and "*interrupt abstraction*" respectively. This second approach is the one used by RTLinux.

RTLinux is a small, fast operating system, following the POSIX 1003.13 "minimal real-time operating system" standard.

RTLinux adds a layer of virtual hardware between the standard Linux kernel and the computer hardware (see Figure 8-1, *RTLinux layer architecture*). As far as the standard Linux kernel is concerned, this new layer appears to be the actual hardware. RTLinux implements a complete and predictable RTOS with no interference from the non-real-time Linux. The RTLinux threads are executed directly by a fixed-priority scheduler. The whole Linux kernel, and all the normal Linux processes, are managed by the RTLinux scheduler as the background task. This way, it is possible to have a complete general purpose operating system running on top of a small predictable RTOS.

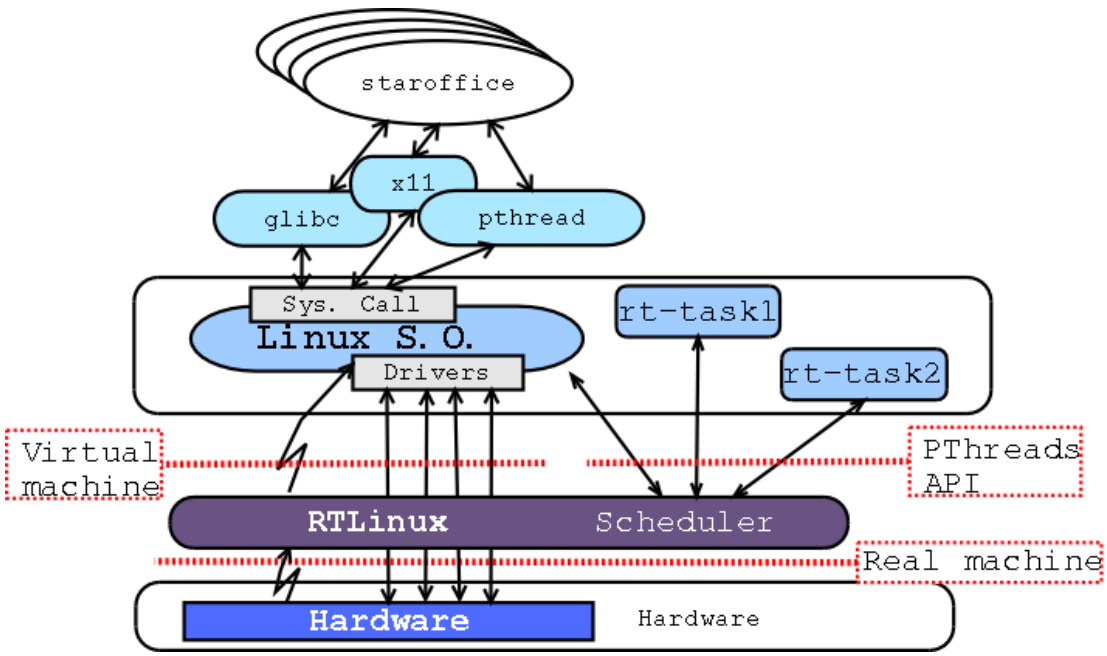


Figure 8-1. RTLinux layer architecture

There are three main modifications done to the Linux kernel in order to virtualize the hardware so that RTLinux can take full control of the machine: The RTLinux layer takes direct control of all the hardware interrupts, interrupts that are not controlled by real-time threads are forwarded to the Linux upper level; RTLinux also takes the control of the timer hardware (8254 and APIC when available) and implements a virtual timer to Linux; and the last modification done to Linux to remove the basic control of the hardware from Linux is to replace all the `cli` and `sti` (disable and enable interrupt flag) functions calls from the Linux kernel so that Linux can no make a real disable but a virtual interrupt disable. These modifications are quiet complex and tricky, but do not require large code (Linux) modifications.

RTLinux provides an execution environment "below" the Linux kernel. One consequence of this, is that realtime threads can not use Linux services because deadlock or system inconsistencies may happen. To overcome this problem, the realtime system has to be divided into two separated layers: the hard realtime layer, executed on top of RTLinux, and the soft realtime, executed as normal Linux processes. Several mechanisms (FIFO, shared memory) can be used to communicate threads in both layers.

The two layer approach is a useful method to provide hard realtime while having all the features of a desktop operating system. It decouples the mechanism of the realtime kernel from the mechanism of the general purpose Linux kernel so that each system can be optimized independently.

8.2. Hardware characteristics

Supported processors:

i386, PPC, ARM (StrongARM/iPAQ).

Supported multi processor:

Yes. It is available for i386 architectures.

8.3. Process management

Scheduling policy:

There are three scheduling policies available: `SCHED_FIFO`, `SCHED_SPORADIC` `SCHED_EDF`. `SCHED_FIFO` is a fixed priority scheduling and threads with the same priority are scheduled in FIFO order. `SCHED_SPORADIC` an implementaion of the sporadic server used to run aperiodic activities. `SCHED_EDF` implements a dynamic priority scheduling policy the EDF (Earliest Deadline First). Each thread has a fixed priority and a deadline. Threads are sorted by priority, but same priority threads are scheduled according to the EDF policy.

Periodic threads:

The system provides special system calls to implement periodic threads. `pthread_make_periodic_np()` and `pthread_wait_np()`

Range of priorities and maximum number of threads:

Minimum and maximum priority is 0 and 1000000 respectively. There is no limit in the number of running threads, but the scheduling cost is proportional to the number of threads. Current scheduler code is designed to handle efficiently a low number of threads (around 10).

Thread creation and deletion:

It provides all the POSIX Thread termination facilities and also some extensions to remove termitate threads easier.

A thread can terminate itself by calling `pthread_exit()` function. `pthread_join()` suspend execution the execution of the calling thread until the target thread terminates. To implement this behavior, when a thread exits, the system do not delete the supporting data until other thread has *joined*. The system also provides the `pthread_detach()` function to indicate that the target thread will not be joined and the system support data can be reclaimed as soon as the thread exits.

A thread can request the cancellation (termination) of other thread: `pthread_cancel()`. The thread that receives the cancellation request, depending on the *cancelability state*, can do one o the following action:

- `PTHREAD_CANCEL_DISABLE`: The cancel request is ignored.
- `PTHREAD_CANCEL_DEFERRED`: The thread will be canceled but only at some safe points.
- `PTHREAD_CANCEL_ASYNCHRONOUS`:The thread is canceled immediately.

A thread can install cancellation cleanup handlers: `pthread_cleanup_push()` and `pthread_cleanup_pop()`. The cleanup handlers are called when the thread exits, is canceled or the handler is removed.

`pthread_delete_np()` function can be used to termintate inmediately a thread and can be used instead of the pair: `pthread_cancel()/pthread_join()`

8.4. Memory management

Protected address spaces

Although RTLinux is designed to run in processors with MMU, all the application threads and the RTLinux kernel run in the same address space. There is no memory protection between threads and the kernel and also between threads themselves.

From the point of view of memory management, RTLinux is the *guest* operating system of the Linux Kernel. The Linux kernel has the whole control of the memory.

Dynamic memory allocation:

RTLinux do not provide dynamic memory allocation nor use it internally. The main argument is that dynamic memory allocation is not predictable if implemented ef-

ficiently. The Real-Time goal of predictability is usually achieved by preallocating most of the resources the threads will use at run time.

It is possible to allocate all the memory that each thread will require before the threads are created.

8.5. Inter-Process communication

Semaphores

POSIX REALTIME semaphores are fully implemented: `sem_init()`, `sem_destroy()`, `sem_getvalue()`, `sem_post()`, `sem_trywait()` and `sem_wait()`. These are counting semaphores.

Mutex

RTLinux supports the POSIX pthread_mutex_ family of functions: `pthread_mutex_init()`, `pthread_mutex_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()` and `pthread_mutex_unlock()`. As well as all the supporting, `pthread_mutexattr_...` like, functions to handle mutex creation attributes.

The supported mutex types are:

- `PTHREAD_MUTEX_NORMAL`: The default POSIX mutex.
- `PTHREAD_MUTEX_SPINLOCK_NP`: Provides a interface to spin-locks used to synchronize the execution in multiprocessor systems.

The supported mutex protocols are:

- `PTHREAD_PRIO_NONE`:No priority control is performed on locking and unlocking.
- `PTHREAD_PRIO_PROTECT`: Immediate priority inheritance. The thread that locks the mutex inherits the priority ceiling of the semaphore, and returns to the original priority when the unlocks the mutex.

Priority inversion control

Mutex provide immediate priority inheritance.

Messages queues

There is no message queues available.

Mailboxes

There is no mailboxes.

Shared memory

shared memory is provided with a non-POSIX interface called *mbuff*. Since all threads are executed in the kernel address space (rtlinux threads share by default all the memory), this sharing memory mechanism is used to communicate rtlinux threads and normal linux processes.

Both execution environments, RTLinux and Linux, have the same mbuff API.

Following the idea that not dynamic memory allocation can be requested during normal system execution, allocation and releasing functions can not be used from RTLinux threads, only at module loading or by linux processes.

FIFOs

RTLinux provide a single IPC called FIFO. It is First-In-First-Out queues that can be read from and written to by Linux processes and RTLinux threads. FIFOs are uni-directional - you can use a pair of FIFOs for bi-directional data exchange.

8.6. Time and timers

Time resolution

Time is measured in nano-seconds. Two different data structures are used to measure time: POSIX structure:

```
struct timespec {
   time_t tv_sec; /* seconds */
   long tv_nsec; /* nanoseconds */
};
```

and RTLinux specific:

```
typedef long long hrtime_t; /* Nano seconds */
```

RTLinux use the highest resolution that the underlying hardware provides.

Currently supported clocks are:

- `CLOCK_MONOTONIC`: This POSIX clock runs at a steady rate, and is never adjusted or reset.
- `CLOCK_REALTIME`: This is the standard POSIX realtime clock. Currently, it is the same as `CLOCK_MONOTONIC`.
- `CLOCK_RTL_SCHED`: The clock that the scheduler uses for task scheduling, it is the best hardware clock.

The following clocks are architecture-dependent. They are not normally found in user programs.

- `CLOCK_8254`: Used on non-APIC x86 machines for scheduling. Its frequency is 1193180Hz.
- `CLOCK_APIC`: Used on SMP x86 machines and single processor equipped with local APIC. This clock ticks at the same frequency than the internal processor clock. If the processor is clocked at 1GHz or higher, then the this clock has a resolution smaller than 1 nano-second.

User timers

There is no user timers. There is only one timer handler per hardware timer, which is the associated with the interrupt handler. When the scheduler module is loaded, the scheduler takes the control of the timer and no other thread can use the timer.

The only timing facility that a thread can use being a periodic thread.

Facilities to add new hardware timers

The internal structure of the source code is prepared to add new timer drivers easily. There is a `struct rtl_clock` similar to the device driver structure used to register UNIX device drivers.

8.7. Driver programming

Interrupts

RTLinux defines two types of interrupts: Soft and Hard Interrupts.

Hard Interrupts the one originated directly by the real hardware. There is very little system interference in the service of these interrupts, therefore the interrupt latency is almost only limited by the performance of the underlying hardware. Since these handlers are executed at the RTLinux level, it is not possible to call linux services. Interrupt handlers are installed and uninstalled with `rtl_request_irq()` and `rtl_free_irq()` functions.

Soft interrupts are normal Linux kernel interrupts, in the sense that the interrupt handlers are executed by the Linux kernel thread. The latency of this interrupts are the same than that of a normal linux, i.e., no realtime. On the other hand, is is possible to use all the facilities of the linux kernel. The following functions are used to manage this kind of interrupts: `rtl_get_soft_irq()` and `rtl_free_soft_irq()`.

It is possible to generate interrupts from RTLinux to Linux with the function: `rtl_global_pend_irq`. Also, by using the POSIX signal mechanism is possible to send signals (which are received like soft interrupts) to the linux thread.

Low level prograaming

Since RTLinux threads are executed in kernel address space and processor mode, it is possible to use all the reserved processor instructions directly. To direct access io ports in ISA bus: `rtl_inb`, `rtl_inb_p`, and the outb counterparts. To access

physical memory (devices located at the PCI bus) it is possible to use `ioremap()`[1] function at thread initialization and then access the device through the returned pointer.

8.8. Quality of Service

There is no QoS utilities.

8.9. Network

A separate module called *RTnet* provides the drivers (only two network drivers are supported) and the protocols (IP, ARP, UDP and ICMP) over Ethernet. The API is quiet similar to the sockets API: `rt_socket()`, `rt_connect()`, `rt_sendmsg()`, etc. Current version of RTnet only works with Linux kernel version 2.2.x.

Also several CAN drivers are available.

8.10. Filesystems

The module `rtl_posixio.o` implements the `/dev` filesystem. It provides unix like (`open()`, `read()`, `write()`, ...) functions to access device driver services, and drivers can be registered with the function: `rtl_register_rtldev()`.

Current implementation provides the following devices: `/dev/rtf`, `/dev/mem` and `/dev/ttyS`

RTLinux do not provide any block device nor regular filesystem implementation. Since the background RTLinux process is the Linux kernel, RTLinux designers choose not to include non-realtime features that are already available and usable.

8.11. Trace and debug

Logic debug

Source level debugging using GDB directly from the same machine that is running the system. The debugging target system (RTLinux) and the host system (where GDB is running) are communicated with FIFOS.

It is possible to debug periodic threads (gdb command **info threads**) in the usual way: step by step, break points, inspect variable values, etc. Multiprocessor systems can be debugged. Also is possible to use graphical front-ends like: DDD, gvd, xxgdb, insight.

Timing debug

An optional tracer module can be included in the system. The tracer will register all the relevant system events and user-defined ones.

The builtin set of events includes RTLinux interrupt handlers entry and exit, scheduler entry and exit, spin-lock acquiring and freeing, context switches, interrupts disabling and enabling. It is possible to attach an arbitrary 32-bit value to each event record. The current value of instruction pointer is also logged with each event.

Events are grouped into classes. It is possible to select logging of events of any combination of different classes during runtime.

Events are logged in circular buffers located in shared memory, where a user-space program can collect and store in disk. This logging mechanism causes a minimal interference.

8.12. Miscelanea

Development tools

The development tools are the same than the used to develop the Linux kernel: GCC, binutils, make, etc.

It is possible to do cross development (host-target) and stand alone development, that is, use the same computer to compile and run/test the embedded system.

Programming languages

The main programming language is "C". But it is also possible to use C++.

Ted Baker reported a preliminary porting of the ADA language to the RTLinux system.

API compatibility

POSIX 1003.13/PSE51-compatible.

Notes

1. ioremap: Maps the given physical address range into the virtual address space with no caching, suitable for being accessed from a driver.

Chapter 9. RTAI

RTAI is the acronym of *Realtime Application Interface*. It was first started at the Dipartimento di Ingeneria Aerospaziale, Politecnico di Milano by Professor Paolo Mantegazza.

RTAI started as a variant of RTLinux in 1999 when Paolo Mantegazza tried to collaborate with RTLinux including new features and rearrange the RTLinux code but Victor did not accepted his contributions. Since then, the relations between RTAI and RTLinux developers are not good.

Although RTAI still contains code from the original RTLinux code, the API differences between them evolved in opposite directions. RTLinux was rewritten to be POSIX compatible and to remain small and simple; RTAI developers added new features and system calls following a personal and non always coherent style. One of the main development guideline of RTLinux was to keep the system as simple as possible and only add new features when strictly necessary. On the other hand, RTAI developers accepted and integrated into the system all code contributions which resulted in a full featured, but some times redundant and incompatible, system.

Currently, RTAI developers have started two major tasks:

- Replace the underlying hardware control RTHAL, based on the RTLinux original code and patented by Victor Yoddaiken, with a new technology called ADEOS "Adaptive Domain Environment for Operating Systems".
- Redesign the API.

9.1. Architecture overview

The basic structure of RTAI is the same than RTLinux. A new software layer is beneath Linux kernel with full control of interrupts and processor key features.

RTAI scheduler treats the Linux operating system kernel as the idle task. Linux only executes when there are no real time tasks to run, and the real time kernel is inactive. The Linux task can never block interrupts or prevent itself from being preempted.

9.2. Hardware characteristics

Supported processors:

i386, MIPS, PPC, ARM, m68k-nommu.

Supported multi processor:

Yes. It is available for i386 architectures.

9.3. Process management

Scheduling policy:

There are two scheduling policies available: `RT_SCHED_FIFO` and `RT_SCHED_RR`.

Function `rt_set_sched_policy(RT_TASK *task, int policy, int rr_quantum_ns)` is used to set the priority and the round robin quantum.

Periodic threads:

The system provides special system calls to implement periodic threads: `rt_task_make_periodic()` and `rt_task_wait_period()`

Range of priorities and maximum number of threads:

Minimum and maximum priority is 0x3fffFfff and 0 respectively. Which is the opposite way as how POSIX define priorities, that is, in POSIX the higher the priority, the higher the priority.

The number of active tasks is only limited by the amount of memory available.

Thread creation and deletion:

`rt_task_init()` and `rt_task_delete` functions are used to create and destroy tasks. If task task was waiting on a queue, i.e. semaphore, mailbox, etc, it is removed from such a queue and messaging tasks pending on its message queue are unblocked with an error return.

9.4. Memory management

Protected address spaces

Although RTAI is designed to run in processors with MMU, all the application tasks and the RTAI kernel run in the same address space. There is no memory protection between tasks and the RTAI kernel and also between threads themselves.

From the point of view of memory management, RTAI is the *guest* operating system of the Linux Kernel. The Linux kernel has the whole control of the memory.

Dynamic memory allocation:

The module `rt_mem_mgr` provides the functions `rt_malloc()` `rt_free()`.

Some amount of memory is pre-allocated then the module in inserted, and real time tasks can allocate and free memory from this pool while executing in RTAI environment.

9.5. Inter-Process communication

Semaphores

Two different and incompatible semaphore facilities are implemented in RTAI: RTAI-SEMAPHORES, FIFO-SEMAPHORES.

- RTAI-SEMAPHORES: The following functions are available: `rt_typed_sem_init()`, `rt_sem_init()`, `rt_sem_delete()`, `rt_sem_signal()`, `rt_sem_wait()`, `rt_sem_wait_if()`, `rt_sem_wait_until()`, `rt_sem_wait_timed()`.

  A semaphore of the kind can operate in one of the three following modes (types): `CNT_SEM` for counting semaphores, `BIN_SEM` for binary semaphores, `RES_SEM` for resource semaphores. Counting semaphores can register up to 0xFFFE events. Binary semaphores do not count signalled events, their count will never exceed 1 whatever number of events is signalled to them.Resource semaphores are special binary semaphores suitable for managing resources. The task that acquires a resource semaphore becomes its owner. The owner has its priority increased to that of any task blocking on a wait to the semaphore. Resource semaphores can be recursed.

- FIFO-SEMAPHORES: FIFO semaphore's API: `rtf_sem_init()`, `rtf_sem_destroy()`, `rtf_sem_post()`, `rtf_sem_trywait()`, `rtf_sem_timed_wait()`.

  One of the first RTLinux and RTAI communication mechanism has an intrinsic synchronization capability, and that feature is the one used to implement this kind of semaphores. Also, current implementation of FIFO's is based of mailboxes.

Mutex

RTAI supports the POSIX `pthread_mutex_` family of functions in the POSIX compatibility module: `pthread_mutex_init()`, `pthread_mutex_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()` and `pthread_mutex_unlock()`. As well as all the supporting, `pthread_mutexattr_...` like, functions to handle mutex creation attributes.

The supported mutex types are:

- `PTHREAD_MUTEX_FAST_NP`: The default POSIX mutex.
- `PTHREAD_MUTEX_RECURSIVE_NP`: This type of semaphore can be locked more then one time without causing a self task deadlock.
- `PTHREAD_MUTEX_ERRORCHECK_NP`: Detects and reports simple usage errors.

Jointly with the mutex the module also implements POSIX Conditional Variables.

Priority inversion control

Mutex provide immediate priority inheritance.

Messages queues

Provides four different (incompatible) intertask messages facilities:

- Message queues compatible with POSIX 1003.b queues. The functionality is provided by the POSIX compatibility module. The API is: `mq_open()`, `mq_send()`, etc.
- Small (4 bytes) synchronous messages. Messages can be priority ordered (determined by the compile time option MSG_PRIORD). Also there are timed sending and receiving primitives. The API is: `rt_send()`, `rt_send_until()`, `rt_receive()`, etc.
- Extended intertask messaging. Allows to use intertaks messages of any size. The API is: `rt_sendx()`, `rt_sendx_until()`, `rt_receivex()`, etc.
- Remote Procedure Calls (these RPC has no has nothing in common then the well know inter-host RPC, like Sun Microsystems™ RPC) do the same thing that synchronous messages but the tasks are coupled awaiting a reply from the receiver. RPCs operate like complementary, send and receive message pairs. The API is: `rt_rpc()`, `rt_rpc_if()`, `rt_return()`, etc.

Mailboxes

RTAI provides mailboxes. Messages are ordered in FIFO order. Different size messages of are allowed. Multiple senders and receivers can read and write messages to the same mailbox.

There are several sending an receiving functions that provides a lot of flexibility (receiving functions are omitted of the following list for the sake of clarity):

- `rt_mbx_send()`: Blocking send of the whole message.
- `rt_mbx_send_wp()`: Send as many bytes as possible, without blocking the calling task.
- `rt_mbx_send_if()`: Send a message, only if the whole message can be passed without blocking the calling task.
- `rt_mbx_send_until()`, `rt_mbx_send_timed`: Send a message with timeout.

Shared memory

Shared memory is provided with a non-POSIX interface called *shmem*. Since all threads are executed in the kernel address space (rtlinux threads share by default all the memory), this sharing memory mechanism is used to communicate RTAI threads and normal linux processes.

Although the API is no the same, it is quiet similar to the SYSTEM V shared memory. The first time an area is requested, the system allocates a new chunk of memory; subsequent requests return the pointer to the already allocated memory.

FIFOs

RTAI fifos maintain full compatibility with those available in NMT_RTLinux. They are implemented on top of mailboxes.

Also provides a mechanism to create fifos by name.

9.6. Time and timers

Time resolution

Time is stored in a RTIME which is defined as:

```
struct timespec {
typedef long long RTIME;
};
```

The main function to get time is `rt_get_time()`, which return the time measured in internal count units since `start_rt_timer()` function was called. Function `rt_get_time_ns()` is the same as rt_get_time but the returned time is converted to nanoseconds.

User timers

There is no user timers. There is only one timer handler per hardware timer, which is the associated with the interrupt handler. When the scheduler module is loaded, the scheduler takes the control of the timer and no other thread can use the timer.

The only timing facility that a thread can use being a periodic thread. And the delay functions: `rt_busy_sleep()`, `rt_sleep()` and `rt_sleep_until()`. Some functions use nanoseconds while other work with hardware ticks.

How time has to be used by the programmer is not well documented. The use of hardware ticks is very efficient, but produces different results when the same program runs on different hardware (for example, a i486 computer use the old 8254 clock while a Pentium has a builtin timer which provides a highest resolution clock).

Watchdog

The Watchdog module provide services to protect tasks (and the host Linux OS) against programming errors in RTAI applications. The watchdog can be programmed to perform several action on the occurrence of a task overrun.

9.7. Driver programming

Interrupts

RTAI defines two types of interrupts: Soft and Hard Interrupts.

Hard Interrupts the one originated directly by the real hardware. There is very little system interference in the service of these interrupts, therefore the interrupt latency is almost only limited by the performance of the underlying hardware. Since these handlers are executed at the RTAI level, it is not possible to call linux services. Interrupt handlers are installed and uninstalled with `rt_request_global_irq()` and `rt_free_global_irq()` functions.

It is possible to install interrupt handlers that will be executed as a normal Linux interrupt service routines. This is a powerful mechanism to execute code in Linux space, that can be triggered just sending a software interrupt from RTAI with the function `rt_pend_linux_irq`.

Low level programming

Since RTAI threads are executed in kernel address space and processor mode, it is possible to use all the reserved processor instructions directly. To direct access io ports in ISA bus: `rtl_inb`, `rtl_inb_p`, and the outb counterparts. To access physical memory (devices located at the PCI bus) it is possible to use `ioremap()`[1] function at thread initialization and then access the device through the returned pointer.

9.8. Quality of Service

There is no QoS utilities.

9.9. Network

A separate module called *RTnet* provides the drivers (only two network drivers are supported) and the protocols (IP, ARP, UDP and ICMP) over Ethernet. The API is quiet similar to the sockets API: `rt_socket()`, `rt_connect()`, `rt_sendmsg()`, etc. Current version of RTnet only works with Linux kernel version 2.2.x.

RTAI has extended its API to allow true remote (other host) procedure call RPC. New API functions has been added with the following syntax: replace the first two letters of the function name (for example: given the `rt_mbx_send()`, the new function `RT_mbx_send()` has been added); and the new function has two new parameters: node and port. This feature do not comply with any communication standard.

9.10. Filesystems

There is no filesystem support except the use of the Linux `/proc` directory which provides status and debug information on the current operating conditions.

9.11. Trace and debug

Logic debug

Source level debugging using Kgdb+Kmod which is a modified version of kgdb/gdbstubs which provides improved kernel module debug support including real-time modules. The target is debugged from a host by mean of serial cable which connects both machines.

Timing debug

An optional tracer module (Linux Trace Toolkit, LTT) can be included in the system. The tracer will register all the relevant system events and user-defined ones. LTT has also a graphical tool to visualize the logged events.

LTT provides developers with all of the information necessary to reconstruct a system's behavior over a certain period of time. Using LTT, one can graphically view the exact the dynamics of a system and find logical and temporal bugs of the system or the application.

9.12. Miscelanea

Development tools

The development tools are the same than the used to develop the Linux kernel: GCC, binutils, make, etc.

It is possible to do cross development (host-target) and stand alone development, that is, use the same computer to compile and run/test the embedded system.

Programming languages

The main programming language is "C". But it is also possible to use C++.

API compatibility

The original API do not adhere to any standard, but it has module which provides some compatibility with POSIX: mutex, conditional variables, message queues and POSIX threads.

Software watchdog

The higest priority periodic task that protects the systems against tasks errors. Available policies:

- Nothing.
- Resynchronise the task's frame.
- Debug.
- Stretch the period of the offending task.
- Slip the offending task by forcibly suspending it for a percentage of its period.
- Suspend the offending task.
- Kill the offending task and remove all trace of it.

Generate code for RTAI from Simulink (RTW)

The Real-Time Workshop (RTW) is a tool that produces C-code or target specific code directly from a Simulink model for a variety of environnements, including real- time systems.

Notes

1. ioremap: Maps the given physical address range into the virtual address space with no caching, suitable for being accessed from a driver.

Chapter 10. RTEMS 4.5+

This document contains a list of features of RTEMS executive for RTOS comparison. The current snapshot of version 4.5+ was studied.

RTEMS is a real-time executive which provides a high performance environment for embedded critical and military applications including many features. RTEMS executive implements more API interface. One of them strictly adheres to POSIX® 1003.1b standard. Kernel creates multithread, multitasking environment for application threads. Each group of system functions is implemented by corresponding manager located in independent library module. Resulting application images are build by linking kernel libraries with application code. Set of used and linked manager modules can be defined independently for each application. There is support for TCP/IP networking, network and local filesystems. Debugging is possible over serial line or Ethernet.

There are enumerated most important POSIX API functions for each studied executive functionality. Other mentioned APIs have similar sets of functions.

10.1. Hardware characteristics

Supported processors:

Motorola MC68xxx, MC683xx and ColdFire; Hitachi SH, Intel i386 and i960; MIPS; PowerPC; SPARC; AMD A29K; Hewlett-Packard PA-RISC .

Supported multi processor:

Most of processors targets can be used in multiprocessor environment, but tasks do not migrate between CPUs. They are strictly bind to predefined CPU.

In addition, there is a port to UNIX which can be used as a prototyping and simulation environment.

10.2. Process management

RTEMS executive does not implement multiprocess environment with separated application address spaces. As a result, next functions supporting independent process creation and deletion are not implemented: `fork()`, `execl()`, `execv()`, `execle()`, `execve()`, `execlp()`, `execvp()`, `pthread_atfork()` and `wait()`.

RTEMS executive is focused on multithread applications and its Task Manager supports full set of functions in classic and POSIX API. Cancellation functions are implemented by Cancellation Manager.

Table 10-1. Thread Creation and Deletion calls

POSIX API	Classic API	Description
`pthread_create()`	`rtems_task_create()`	Create a new thread of execution.
`pthread_exit()`	`rtems_task_delete()`	Destroy a thread.

POSIX API	Classic API	Description
`pthread_cancel()`		Cancel a thread at the next cancellation point.
`pthread_detach()`		Detach a thread so it doesn't need to be joined.
`pthread_join()`		Join a thread waiting for its exit status.

All other POSIX thread control related functions are implemented: `pthread_self()`, `pthread_equal()`, `pthread_once()`, `pthread_setschedparam()`, `pthread_getschedparam()` and group of functions `pthread_attr_getxxx()`/`pthread_attr_setxxx()`.

Scheduling policy:

- Event-driven, priority-based, preemptive scheduling
- Optional rate monotonic scheduling

Periodic threads:

RTEMS do not provide any special API call to schedule peritodic threads.

Range of priorities and maximum number of threads:

- 255 executive kernel priority levels (1 the highest till 255 the lowest)

  POSIX API defines its own priority range from 1 to 254 (1 the lowest and 254 the highest)

- Maximal number of threads can be defined for each application and configurable maximum depends on amount of available memory.

Thread creation and deletion:

Threads can be created dynamically.

Controlled thread deletion?. Yes

10.3. Memory management

Protected address spaces

Not supported.

Dynamic memory allocation:

`malloc()` and `free()` functions - yes

Variable size memory chunks allocation? yes

Based on Region Manager concept.

10.4. Inter-Process communication

RTEMS implements all standard POSIX 1003.1b IPC mechanisms for concurrent threads synchronization and communication.

Semaphores

POSIX binary, counting, with/without timeout, named/unnamed

Creation and destruction of unnamed semaphores: `sem_init()` and `sem_destroy()`

Opening/creation and closing of named semaphores: `sem_open()` and `sem_close()`. An unused named semaphore can be deleted by `sem_unlink()`.

Function `sem_wait()` attempts to lock a semaphore, If the semaphore is unavailable (value is zero) blocks until the semaphore becomes available. There is non-blocking version `sem_trywait()` and version with timeout `sem_timedwait()`.

Function `sem_post()` unlocks the semaphore and function `sem_getvalue()` enables to retrieve actual value of the semaphore.

Mutex

Similar set of basic POSIX API functions exists as for semaphores. There are classic API functions as well.

Basic functions: `pthread_mutex_init()`, `pthread_mutex_destroy()`, `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_timedlock()` and `pthread_mutex_unlock()`.

Next functions are used to manipulate with mutex attributes: `pthread_mutexattr_init()`, `pthread_mutexattr_destroy()`, `pthread_mutexattr_setprotocol()`, `pthread_mutexattr_getprotocol()`, `pthread_mutexattr_setprioceiling()`, `pthread_mutexattr_getprioceiling()`, `pthread_mutexattr_setpshared()` and `pthread_mutexattr_getpshared()`.

The following ordering protocols for waiting threads can be specified:

- `PTHREAD_PRIO_NONE`: blocking order FIFO.
- `PTHREAD_PRIO_INHERIT`: blocking order priority with the priority inheritance protocol in effect.
- `PTHREAD_PRIO_PROTECT`: blocking order priority with the priority ceiling protocol in effect.

Priority inversion control:

Possible to activate priority inheritance or priority ceiling protocol.

Conditional variables

RTEMS implements POSIX conditional variables in Condition Variable Manager.

The following functions are responsible of conditional variables creation, destruction and manipulation: `pthread_cond_init()`, `pthread_cond_destroy()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`, `pthread_cond_wait()` and `pthread_cond_timedwait()`.

The shared attribute can be defined and read by corresponding attribute functions.

Messages queues

FIFO or task priority wakeup order, LIFO urgent messages support, with timeout, fixed depth and max message size.

Message Passing Manager implements next POSIX functions: `mq_open()`, `mq_close()`, `mq_unlink()`, `mq_send()`, `mq_receive()`, `mq_notify()`, `mq_setattr()` and `mq_getattr()`.

Unix style FIFO special files

In development.

Mailboxes

Implemented only for ITRON API.

Shared memory

No special directives.

Signals

Standard and POSIX signals with timeout. Signals are numbered from 1 to 32.

There are standard functions for sending of signal to process (whole application) `kill()` and `sigqueue()` with value and queuing capability. The thread specific version exists as well `pthread_kill()`. Signals can be masked on process `sigprocmask()` or thread basis `pthread_sigmask()`.

There are defined standard functions for signal mask manipulation `sigaddset()`, `sigdelset()`, `sigfillset()`, `sigismember()` and `sigemptyset()`. Required reaction on unmasked pending signal or received signal is defined by function `sigaction()`.

More functions for waiting, examination of pending signals and combined mask manipulation and waiting are defined `pause()`, `sigpending()`, `sigsuspend()`, `sigwait()`, `sigwaitinfo()` and `sigtimedwait()`.

Alarm signal after specified number of seconds can be scheduled by `alarm()` function.

Other mechanisms

Native RTEMS API for 32 events.

An event flag is used by a task/thread (or ISR) to inform another task of the occurrence of a significant situation. Thirty-two event flags are associated with each task. A collection of one or more event flags is referred to as an event set. The data type rtems_event_set is used to manage event sets.

10.5. Time and timers

Time resolution

Tick timer resolution depends and can be set on BSP level.

Clock functions

Clock Manager provides functions for examining clock resolution `clock_getres()`, setting and getting real time `clock_gettime()` and `clock_settime()`. Execution delay functions `sleep()` and `nanosleep()`. Functions `gettimeofday()` and `time()` reads actual calendar time.

Application timers

Next POSIX API functions are provided by Timer Manager. Per-process (application) timers can be created and deleted by functions `timer_create()` and `timer_delete()`. Timer expiration time can be set and examined by `timer_settime()` and `timer_gettime()`. Number of overrun counts can be check by function `timer_getoverrun()`.

There are many other time related functions provided through different manages as well: `alarm()`, `sleep()`, `nanosleep()`, `sigtimedwait()`, `pthread_cond_timedwait()`, etc.

Facilities to add new hardware timers

N/A

10.6. Driver programming

Interrupts

Interrupts are not converted to signals or other asynchronous events. They are handled by any C routine which was connected to the given interrupt vector. An interrupt handler can be established by any normal thread, kernel driver is not needed.

Functions to manage interrupt handlers are system specific and that is why only classic API is available.

`rtems_interrupt_catch()`

This directive establishes new specified interrupt service routine (ISR) for the specified interrupt vector number. The previous ISR for the specified vector is returned by call.

`rtems_interrupt_disable()`

This directive disables all maskable interrupts and returns the previous level. A later invocation of the `rtems_interrupt_enable()` directive should be used to restore the interrupt level.

`rtems_interrupt_enable()`

This directive enables maskable interrupts to the level which was returned by a previous call to `rtems_interrupt_disable()`. Immediately prior to invoking this directive, maskable interrupts should be disabled by a call to rtems_interrupt_disable and will be enabled when this directive returns to the caller.

`rtems_interrupt_flash()`

This directive temporarily enables disabled maskable interrupts to the level which was returned by a previous call to `rtems_interrupt_disable()`.

`rtems_interrupt_is_in_progress()`

This directive returns `TRUE` if the processor is currently servicing an interrupt and `FALSE` otherwise. A return value of `TRUE` indicates that the caller is an interrupt service routine, not a task. The directives available to an interrupt service routine are restricted.

Kernel facilities

Drivers are written as regular POSIX or RTEMS threads. Only ISR notification function is special case, which should transfer/invoke processing to regular threads.

10.7. Quality of Service

none

10.8. Network

TCP/IP Stack

- High performance port of FreeBSD TCP/IP stack
- UDP, TCP
- ICMP, DHCP, RARP
- TFTP
- RPC
- FTPD
- HTTPD

- CORBA

10.9. Filesystems

In-Memory FileSystem (IMFS). The IMFS is a full featured POSIX filesystem that keeps all information in memory.

DOSFS/FAT filesystem in development.

10.10. Trace and debug

GNU debugger (gdb) - thread aware

DDD GUI interface to GDB

Debug over Ethernet, Serial Port or BDM

Timing debugger - probably no

System trace and event record - probably no

10.11. Miscellaneous

Graphic support depends on BSP package

There are configuration and known applications using graphics frame-buffer and there exists MicroWindows and Nano-X port for RTEMS executive.

Development environment: GCC and other GNU tools. Remote debugging via serial line or Ethernet.

Fail signal and recovery: setjmp/longjmp

Supported programming languages: C, C++, ADA

Provided RTOS APIs:

- RTEID/ORKID based Classic API.
- uITRON 3.0 API in development.
- ADA language API for RT systems.
- POSIX 1003.1b API including threads.

  The implementation status can be sketched by one section from reported compliance summary which shows next numbers for 362 POSIX 1003.1b functions defined by standard.

  - Implemented : 301
  - Unimplemented : 21
  - Unimplementable : 16
  - Partial : 2
  - Dummy : 19
  - Untested : 1

Following is a list of the most important lacking functionalities of current version of RTEMS executive.

- The current implementation of `dup()` is insufficient.
- FIFOs `mkfifo()` are not currently implemented.
- Asynchronous IO is not implemented.
- The `flockfile()` family is not implemented
- getc/putc unlocked family is not implemented
- Shared Memory is not implemented

Chapter 11. QNX

QNX is a realtime operating system that provides multitasking, multiuser, networking, message passing, preemtive scheduling, fast context-switching and so on services. Moreover, QNX achieves these capabilities with a POSIX standar API. Other important feature is modularity allowing QNX be scaled to very small sizes for embedded system or scaled for large systems (workstations). So, this chapter describes the QNX v6 realtime operating system features.

11.1. Architecture overview

The following figure shows the QNX architecture:

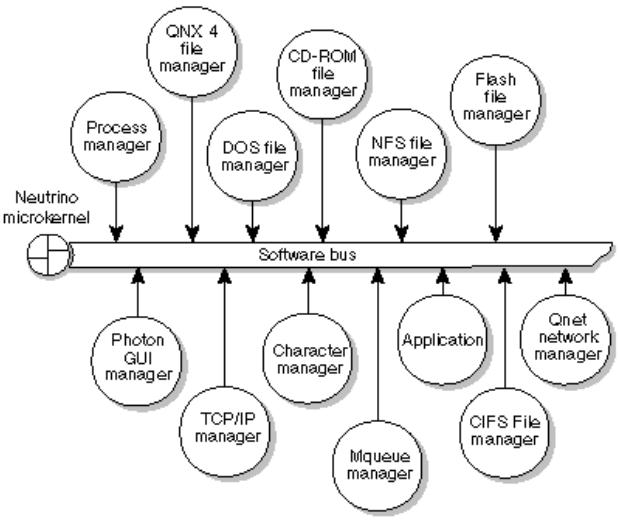


Figure 11-1. QNX architecture

As shown in figure, the QNX architecture consists of the small *Neutrino microkernel* managing a group of cooperating processes. The most important one is *process manager* (explained after) and the other processes are, for example, GUI manager, nerwork manager, devices manager, shared libraries and so on. Moreover, it is important to note that there is a module called *procnto* that consists of the Neutrino microkernel and process manager and that, this module, is required for all runtime systems.

Now, the most importants modules are presented:

Neutrino Microkernel

The QNX microkernel, known as *Neutrino*, is an implementation of the core *POSIX* together with the fundamental message passing services. The *POSIX* features that are not implemented in the microkernel are provided by optional processes and shared libraries.

Neutrino provides a few fundamental services:

- *Thread services:* Neutrino provides the *POSIX* thread creation primitives.
- *Signal services:* Neutrino provides the *POSIX* signal primitives.
- *Message passing services:* Neutrino handles the routing of all messages between all threads through the whole system.

- *Synchronization services:* Neutrino provides the *POSIX* thread synchronization primitives.
- *Scheduling services:* Neutrino schedules threads using the various *POSIX* real-time scheduling algorithms.
- *Timers services:* Neutrino provides the set of *POSIX* timer.

Process Manager

The *process manager* is capable of creating multiple POSIX processes (each of which may contain multiples POSIX threads). Its main areas of responsability include:

- *Process management:* It manages process creation, destruction, and process attributes such us user ID and group ID.
- *Memory management:* It manages memory protection, shared libraries, and POSIX shared memory primitives.
- *Pathname management:* It manages the pathname space (mountpoints).

Message-based Interprocess Comunication

When several threads run concurrently, as in typical realtime multitasking environments, the operating system must provide mechanisms to allow them to communicate with each other. This mechanism, called *Interprocess Comunication*, uses a message passing as its fundamental principle.

Message passing not only allows processes to pass data to each other, but also provides a mechanism to sinchronize the execution of several processes.

11.2. Hardware characteristics

Supported processors

The processors families x86, ARM, MIPS, PowerPC and SH-4 are supported by QNX.

Supported multi processor

QNX is available for x86 (any Intel multi processor specification) and PowerPC (600 or 700 series) multi processor architectures.

11.3. Process management

QNX provides both POSIX processes and POSIX threads. So, the process management is the responsible for the process creation and destruction as well as the management of process attributes.

POSIX processes management:

QNX realizes the POSIX processes creation and destruction, basically, using these POSIX functions: `spawn()`, `fork()` and `exec()`.

Moreover, QNX provides other non-POSIX functions. These primitives are: `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnp()`, `spawnv()`, `spawnve()`, `spawnvp()`, `spawnvpe()`, `execlpe()` and `execvpe()`.

Scheduling policy:

The QNX microkernel provides the following scheduling algorithms:

- *FIFO scheduling:*
- *Round-Robin scheduling:*
- *Adaptive scheduling:* A thread behaves as follows:
  - If the thread consumes its timeslice, its priority is reduced by 1. Note that it will only drop one level below its original priority.
  - If the threads blocks, it inmediately comes back to its original priority.
- *Sporadic scheduling:* Essentially, this algorithm allows a thread to service aperiodic events without exposing the hard deadlines of other threads or processes in the system.

Periodic threads:

QNX does not provide functions to implement directly periodic threads. Therefore, it is necessary to use timers calls to perform this functioning. Moreover, using timers, the following notification types may be used: signal, pulses and thread creation.

Range of priorities and maximum number of threads:

Each thread can have a range of priorities from 0 to 31 (the highest priority), *independently of the scheduling policy*. The special *idle* thread (in the process manager) has priority 0 and is always ready to run. Moreover, a thread have a *real priority* and a *effective priority*, and it is scheduled in accordance with its effective priority.

There is no limit thread creation in the same process, except the memory space, of course.

Thread creation and deletion:

QNX provides a full-POSIX API for managing the thread creation and deletion. The following table shows these functions as well as the microkernel primitives used by these routines:

Table 11-1. Thread creation and deletion calls

POSIX call	Microkernel call	Description
pthread_create()	ThreadCreate()	Create a new thread of execution.
pthread_exit()	ThreadDestroy()	Destroy a thread.

POSIX call	Microkernel call	Description
pthread_cancel()	ThreadCancel()	Cancel a thread at the next cancellation point.
pthread_detach()	ThreadDetach()	Detach a thread so it doesn't need to be joined.
pthread_join()	ThreadJoin()	Join a thread waiting for its exit status.

11.4. Memory management

Protected adress space:

Neutrino offers complete memory protection for user applications and for operating system components (device drivers, filesystems, etc.).

Moreover, QNX reaches the complex POSIX process model in a protected environment using a MMU (Memory Management Unit) mechanism. This protection is useful both for development and for the runtime system.

In conclusion, QNX provides a microkernel architecture with full memory protection between each operating system components such as filesystem, TCP/IP, Qnet and so on. Therefore each process is a address space separated and protected against the rest of components.

Dynamic memory allocation:

QNX provides dynamically request memory allocation using the `malloc()`, `realloc()` or `calloc()` functions, and provides the `free()` function to release memory allocation.

11.5. Inter-Process communication

QNX provides the POSIX standard process/thread synchronization and communication services. The following tables show a resume of all these mechanisms:

Table 11-2. Synchronization Services

Sync Mechanism	Implemented in	Used by	POSIX
Semaphore	Kernel	Process/Threads	YES
Mutexes	Kernel	Threads	YES
Recursive Mutexes	Kernel	Threads	YES
Condition variable	External Process	Threads	YES
Readers/Writers locks	External Process	Threads	YES
Barriers	External Process	Threads	YES
FIFO scheduling	Kernel	Process/Threads	NO

Sync Mechanism	Implemented in	Used by	POSIX
Atomic operations	Kernel	Process/Threads	NO

Table 11-3. Communication Services

Communication Mechanism	Implemented in	Used by	POSIX
Message Passing	kernel	Processes	NO
Signals	--	Processes/Threads	YES
PIPES	external process	Processes/Threads	YES
FIFOs	external process	Processes/Threads	YES
Message Queues	external process	Threads	YES
Shared Memory	process manager	Processes/Threads	YES

Semaphores:

QNX provides the full-POSIX semaphores calls. Therefore, both named and unamed semaphores are provided.

The following table list the primitives available on QNX for managing the semaphores, as well as, the microkernel calls used by these routines:

Table 11-4. Semaphore management primitives

POSIX Call	Kernel Call	Description
`sem_init()`	`SyncTypeCreate()`	Creates a semaphore
`sem_destroy()`	`SyncDestroy()`	Destroy synchronization object
`sem_wait()`	`SyncSemWait()`	Wait on a semaphore
`sem_trywait()`	`SyncSemWait()`	Wait on a semaphore
`sem_post()`	`SyncSemPost()`	Post a semaphore

Mutexes:

QNX provides the full POSIX mutexes calls where each of them, are associated with a microkernel call. The following table shows these routines:

Table 11-5. Mutexes management primitives

POSIX Call	Kernel Call	Description
`pthread_mutex_init()`	`SyncTypeCreate()`	Create a mutex.
`pthread_mutex_destroy()`	`SyncDesctroy()`	Destroy a mutex.
`pthread_mutex_lock()`	`SyncMutexLock()`	Lock a mutex.
`pthread_mutex_unlock()`	`SyncMutexUnlock()`	Unlock a mutex.
`pthread_mutex_trylock()`	`SyncMutexLock()`	Used to test whether the mutex is currently locked or not.

The following mutex types are supported by QNX:

PTHREAD_MUTEX_NORMAL,PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_RECURSIVE and PTHREAD_MUTEX_DEFAULT. Note that QNX supports the full POSIX mutex types.

Currently, there is only one POSIX mutex protocol available in QNX: the PTHREAD_PRIO_INHERIT protocol. Therefore, PTHREAD_PRIO_PROTECT protocol is not supported.

Finally, it is important to note how the classic priority inversion problem is solved: If a thread with a higher priority than the mutex owner tries to lock a mutex, then the effective priority of the current owner will be increased to that of the higher priority blocked thread waiting for mutex. The owner will returns to its real priority when it unlocks the mutex. This scheme is called *priority inheritance*.

Recursive mutexes

This service consists of that the attribute of the mutex can be modified (using PTHREAD_MUTEX_RECURSIVE type) to allow a mutex to be recursively locked by the same thread. So, QNX provides this POSIX functioning.

Priority inversion control:

As described earlier, the priority inversion control problem is solved using inmediate priority inheritance.

Condition variable:

In the same way as semaphores and mutexes, QNX provides full POSIX condition variable functions as well as the microkernel calls that are used by these routines. These functions are: pthread_cond_init(), pthread_cond_destroy(), pthread_cond_wait(), pthread_cond_signal() and pthread_cond_broadcast().

Reader/Writer Locks:

QNX provides full POSIX reader/writers functions. Now these functions are listed: pthread_rwlock_rdlock(), pthread_rwlock_wrlock(), pthread_rwlock_unlock(), pthread_rwlock_tryrdlock(), pthread_rwlock_trywrlock().

It is important to note that Reader/writer locks are not implemented directly within the kernel and have not associated a microkernel call either, but are built from the mutex and condvar services provided by the kernel.

Barriers:

QNX provides full POSIX barriers functions. Those functions are: pthread_barrier_init(), pthread_barrier_wait(), pthread_barrier_destroy() and pthread_barrierattr_* family.

FIFO scheduling:

This type of synchronization consist of selecting the POSIX FIFO scheduling algorithm. In this way, we can guarantee that two threads of the same priority don't execute the critical section concurrently.

Note that this feature is not accomplished on SMP systems.

Atomic Operations:

This synchronization mechanism allows to perform a short operation with the guarantee that the operation will performing atomically. The most important atomic operations that Neutrino provides are:

Table 11-6. Atomic Operations

Function	Description
`atomic_add()`	Add a value.
`atomic_sub()`	Substract a value.
`atomic_clr()`	Clear bits.
`atomic_set()`	Set bits.
`atomic_toggle()`	Complementing bits.

Message Passing:

QNX provides the message passing mechanism as the main form of IPC in QNX and Neutrino. Although other forms are available, those options are built over its native IPC.

The following illustration shows the state changes in a send-receive transaction.

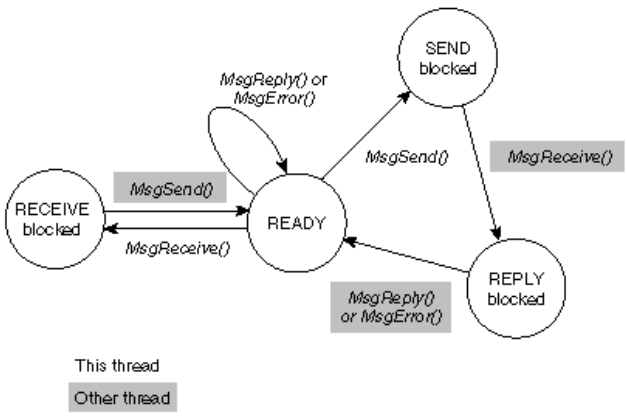


Figure 11-2. QNX message passing model

As shown in figure, a thread that does a `MsgSend()` to another thread or process will be blocked until the receiving thread does a `MsgReceive()`, processes the message, and executes a `MsgReply()`. If a thread executes a `MsgReceive()` without a previously sent message pending, it will block until another thread executes a `MsgSend()`. Therefore, while the send and receive operations are blocked and synchronous, `MsgReply()` and `MsgError()` don't block and furthermore, unlock the client from `MsgSend()`.

The following table lists the message passing API provided by QNX:

Table 11-7. Message Passing API

Function	Description
`MsgSend()`	Send a message and block until reply.
`MsgReceive()`	Wait for a message.
`MsgReceivePulse()`	Wait for a tiny, nonblocking message (pulse).
`MsgReply()`	Reply to a message.
`MsgError()`	Reply only with an error status. No message bytes are transferred.
`MsgRead()`	Read additional data from a received message.
`MsgWrite()`	Write additional data to a reply message.
`MsgInfo()`	Obtain info on a received message.
`MsgSendPulse()`	Send a tiny, nonblocking message (pulse).
`MsgDeliverEvent()`	Deliver an event to a client.
`MsgKeyData()`	Key a message to allow security checks.

Moreover, QNX provides other message passing functions and mechanism that are based on the API defined above. This mechanism are: *Multipart Transfers*, *Channels* and *Pulses*.

Signals:

QNX support POSIX signals, realtime signals and traditional UNIX signals. Moreover, Neutrino provides eight special signals. Therefore, a total of 64 signals have been defined. The following table resumes these signals range:

Table 11-8. Signals range

Signal range	Description
1...57	57 POSIX signals, including traditional UNIX signals.
41...56	16 POSIX realtime signals.
57...64	8 special Neutrino signals.

PIPEs:

To uses pipes in QNX, the pipe resource manager, called *pipe* must be loaded.

This POSIX comunication mechanism are available on QNX using the symbol `|` to create a pipe from shell and the `pipe()` or `popen()` functions to create a pipe from programs.

FIFOs:

As well as pipes, for using FIFOs in QNX, the resource manager *pipe* must be loaded.

To manage FIFOs, QNX provides the following utilities: the **mkfifo** and **rm** command to create/delete FIFOs from shell, and the `mkfifo()`, `remove()` and `unlink()` functions to create/delete FIFOs from programs.

Message Queues:

QNX provides a full POSIX message queue API. To use this synchronous mechanism in Neutrino, the message queue resouce manager called *mqueue* must be loaded.

There is a fundamental different between QNX messages and POSIX messages queue: while QNX messages block and copy data directly from the address spaces of the process sending to the address space of process receiving, POSIX message queue, on the other hand, do not block and may have pending messages queued.

In Neutrino microkernel, all mesages queues created will appear in the filename space under the directory */dev/mqueue*.

Shared Memory:

Full POSIX shared memory is implemented in Neutrino via the process manager called *procnto*. The following functions are implemented as messages to *procnto*: `shm_open()`, `close()`, `mmap()`, `mummap()`, `mprotect()`, `msync()`, `shm_ctl()` and `shm_unlink()`.

11.6. Time and timers

Timer resolution

Neutrino provides the full set of POSIX clock functionality. Therefore, time may be measured in nanosecond, as well as, in second.

Moreover, QNX also provides several non POSIX clock functions: `ClockAdjust()` and `ClockCycle()`.

The posible clock types are:

- `CLOCK_MONOTONIC`: This POSIX clock always increase at a constant rate and can not be adjusted.
- `CLOCK_SOFTTIME`: The same as `CLOCK_REALTIME`, but if the CPU is in power-down mode, the clock stop running.
- `CLOCK_REALTIME`: A clock that mantains a ystem time.

It is importan to know that valid dates on a QNX system range from January 1970 to January 2554, and note that POSIX may be limited to the year 2038.

11.7. Driver programming

Interrupts

Neutrino provides the following interrupt handling API:

Table 11-9. Interrupt Handling API

Function	Description
`InterruptAttach()`	Attach a local function to an interrupt vector.
`InterruptAttachEvent()`	Generate an event on an interrupt, which will ready a thread
`InterruptDetach()`	Detach from an interrupt using the ID returned by `InterruptAttach()` or `InterruptAttachEvent()`.
`InterruptWait()`	Wait for interrupt.
`InterruptEnable()`	Enable hardware interrupts.
`InterruptDisable()`	Disable hardware interrupts.
`InterruptMask()`	Mask a hardware interrupt.
`InterruptUnmask()`	Unmask a hardware interrupt.
`InterruptLock()`	Guard a critical section of code between an interrupt handler and a thread. It is necessary to make this code SMP safe.
`InterruptUnlock()`	Remove an SMP-safe lock on a critical section of code.

Using this API, a privileged thread (in user level) can call `InterruptAttach()` or `InterruptAttachEvent()`, passing a hardware interrupt number and the address of a function in the thread's address space to be called when the interrupt occurs. QNX allows multiple ISRs to be attached to each hardware interrupt number.

Low level programming

To access a PCI devices, QNX provides a well defined API. Some of these functions are: `pci_map_irq()` for mapping a interrupt pin to an IRQ, `pci_find_device()` to find the PCI device with a given ID and vendor ID, and so on.

In conclusion, QNX provides a lot of APIs to manage low level hardware features.

11.8. Quality of Service

Quality of Service are provided in QNX using Qnet. This protocol supports policies to ensure reliable transactions.

In QNX, the default QoS policy, called *sequential* works like this. The first network link is used until it fails, then the next link is used, and so on. The user specifies the preferred

order of the links: the most desired link is specified first, followed by the second choice, and so on.

11.9. Network

Basically, QNX provides two protocols: TCP/IP and QNet.

TCP/IP

QNX provides an implementation of the TCP/IP stack (npm-tcpip) relatively light, modular and using the common BSD Socket API (this API is the standar API for TCP/IP programing in the UNIX world). This implemenattion provides an NFS server/client, multicasting, support for IPsec security, IPv6, and so on.

QNet

Qnet is the QNX native networking protocol that extends the operating system message passing interprocess comunication (IPC) transparently through a networks of microkernels.

To understand how this protocol works, consider the case of a simple network with two nodes; one of them contains the client process, and the other one contains the server process. The client process creates a connection to the server, using the `ConnectAttach()` function call and then sends its messages using the `MsgSend()` function call (this network case works in the same way that in the single node case).

It is important to note that the POSIX calls such as `open()`, `read()`, `write()`, etc can be used too.

11.10. Filesystems

The following list shows the filesystem supported by QNX:

- *RAM filesystem:* Every QNX system provides a simple RAM-based "filesystem" that allows read/write files to be placed under */dev/shmen*.
- *QNX4 filesystem:* This is the native neutrino filesystem.
- *DOS filesystem:* The DOS Filesystem provides transparent access to DOS disks.
- *CD-ROM filesystem:* The CD-ROM filesystem shared library, called *fs-cd.so*, implements the ISO 9660 standard as well as a number of extensions, including Rock Ridge (RRIP), Joliet (Microsoft), and multisession (Kodak Photo CD, enhanced audio).
- *Flash filesystem:* Some of the flash filesystems supported by QNX are shown: generic flas filesystem (def-generic), SH$7750 Aspen eval board (devf-aspen), PowerPaq eval board (devf-ppag) and so on.
- *NFS filesystem*
- *CIFS filesystem:* This filesystem is known typically as *SMB*.
- *Linux Ext2 filesystem:* The Ext2 filesystem (fs-ext2.so) provides transparent access to Linux disk partitions.
- *Virtual filesystem:* QNX provides two types of virtual filesystem:

- *Package filesystem:* It is a virtual filesystem that presents a customized view of a set of files and directories to a client.
- *Inflator:* It is a resource manager that sits in front of other filesystems and inflates files that were previously deflated (using the deflate utility).

In conclusion, most of these filesystems are *resource managers* that adopt a portion of the pathname space (called *mountpoint*) and provide filesystem services through the standar POSIX API (`open()`, `close()`, `read()`, etc.).

QNX provides a process, called *pathname management* that manages the pathname space and allows to support the POSIX semantics for device and file access, while making the use of those services optional for small embedded systems.

Some of these devices filesystems or mountpoints are: `/dev/hd0`, `/dev/fd0`, `/dev/shmem`, `/dev/mem`, `/dev/zero`, `/dev/ser*` (serial device), `/dev/con*` (simple console device), `/dev/par*` (parallel printer), `/dev/pty*` (pseudo console device) and so on.

In conclusion, when a process opens a file using the POSIX `open()` call, the library routine sends the pathname to procnto, where the pathname is compared against the prefix tree to determine which resource managers should be sent this message (This work is realized by pathname management thread).

11.11. Trace and debug

Logic debug

The utility GBD can be used for debugging C and C++ programs.

Timing debug

When a program uses dynamic memory allocation, it is posible introduce *memory leaks*. Therefore, to determine whether the heap was corrupted, the malloc debug library is available in QNX. This library provides several checking by default: *allocation memory*, *reallocating memory* and *releasing memory*. Moreover, the `mallopt()` function allows optional checks: `MALLOC_CKACCESS` to detect buffer overruns and underruns as a result of string operations, `MALLOC_FILLAREA` to detect overruns as a result of user request size, `MALLOC_CKCHAIN` and `MALLOC_VERIFY`.

Furthermore, is important to note that when the library detects a problem, a message error is generated and the program is aborted. This behavior can be change specifying a handler that determines what is done when a warning o faltal error is detected.

Finally, this library provides the `malloc_dump_unreferenced()` function to trace and giving results. This function suspends all threads, performs the trace operation and prints messages of all memory leaks detected.

Other tracing mechanism utility called *procnto-instr* is provided by QNX. This module allows monitor the system execution in real time. Moreover, the `TraceEvent()` call can be used to generate custom events into the trace stream.

11.12. Miscelanea

Graphic support

The propietary environment built by QNX Software Systems is called *Photon MicroGUI*.

Development tools

QNX provides the Qcc and qcc compilers. They are based on the POSIX c86 utility. By default Qcc considers a C++ program while qcc considers a C program.

It is important to know that QNX provides a command that allows to built a scaled embedded systems. This utility is named **mkifs**.

It is possible to do cross development (host-target) and stand alone development, that is, use the same computer to compile and run/test the embedded system.

Programming languages

The main programming languages are C and C++.

Moreover, QNX provides the Visual Age Micro-Edition environment to develop programs written using the java languaje.

Finally, QNX provides a tool called the Photon Application Builder (abbreviated as PhAB or AppBuilder). It's a visual tool to develop user's interface. The code generated by this tool is C and C++.

API compatibility

POSIX 1003.1.

Chapter 12. VxWorks 5.x

12.1. Hardware characteristics

Supported processors:

Motorola 68k/CPU32/ColdFire/PowerPC, Intel x86, Intel ARM/StrongARM, Hitachi SuperH, MIPS

Supported multi processor:

Multiprocessor systems are supported with optional package VxMP.

12.2. Process management

Scheduling policy:

There are only two scheduling policies available. Default is preemptive priority scheduling, it can be changed to Round-Robin scheduling. User can switch between these two polities during runtime.

Periodic threads:

There are no means to handle periodic threads. It is neccessary for an application to use standard timers.

Range of priorities and maximum number of threads:

There are 256 priority levels in range 0 - 255, 0 is the highest priority. Number of threads limited only by amount of available memory etc.

Thread creation and deletion:

Threads can be created dynamically by calling `TaskSpawn()` function.

12.3. Memory management

Protected address spaces

There is no any protection between user and system space and among different processes in basic VxWorks OS. All processes have unlimited access into whole memory space. Memory protection is available as optional package (VxVMI).

Dynamic memory allocation:

Full support using standard functions `malloc()`,`calloc()`,`realloc()` and `free()`.

12.4. Inter-Process communication

VxWorks implement two APIs for IPC - native `wind` API and POSIX 1003.1b compliant API.

Semaphores

Binary, counting, mutex, with timeout.

Semphore creation and deletion is performed by calling `semBCreate()` (binary), `semCCreate()` (counting), `semMCreate()` (mutex) and `semClose()` functions.

Function semTake() attempts to take a semaphore. If the semaphore is not available, `semTake()` blocks until semaphore becomes available.Duration of blocking depends on parameter *`timeout`* which can be WAIT_FOREVER, WAIT_NOWAIT or any nonzero value which specifies timeout.When timeout is WAIT_NOWAIT, `semTake()` becomes nonblocking.

Mutex

Mutex semaphore implements priority inheritance algorithm and provides means disabling task deletion while this task owns such semaphore. These features can be switched on during semaphore creation by parameters SEM_INVERSION_SAFE and SEM_DELETE_SAFE.

Priority inversion control

Possible with mutex semaphores.

Messages queues

Synchronous, prioritized messages, with timeout, fixed depth and max message size. Depth and maximal message length is specified durinq queue creation and it can't be changed later. Wind API provides functions `msgQCreate()` for queue creation, `msgQSend()` for sending a message and `msgQRcvd()` for receiving a message.

Mailboxes

none

Shared memory

No special means because there are no memory protection between tasks. For multiprocessor systems optional package VxMP.

(Queued) signals

The wind kernel supports two types of signal interface: UNIX BSD-style signals and POSIX-compatible signals. The POSIX-compatible signal interface, in turn, includes both the fundamental signaling interface specified in the POSIX standard 1003.1, and the queued-signals extension from POSIX 1003.1b.

The signal facility provides a set of 31 distinct signals. A signal can be raised by calling `kill()`. A signal handler is bound to a particular signal with `sigaction()`. Signals are blocked for the duration of the signal handler. Tasks can block the occurrence of certain signals with `sigprocmask()`. If a signal is blocked when it is raised, its handler routine will be called when the signal becomes unblocked.

12.5. Time and timers

Time resolution

Configurable tick timer resolution, based on system clock. Limited by HW capabilities and kernel configuration, default system clock is 60Hz. As an example, on standard PC machine with Celeron 566MHz processor and kernel configured with TCP/IP stack, full debug support and asynchronous I/O support it is capable of up to cca 8kHz.

Facilities to add new hardware timers

N/A

12.6. Driver programming

Interrupts

Interrupts are not converted to signals or other asynchronous events. They are handled by any C routine which was connected to given interrupt vector by function `IntConnect`. An interrupt can be handled by application thread, kernel driver is not needed. All interrupt handlers use one common special stack which is prepared by the system during the start-up.

When a task causes a hardware exception such as illegal instruction or bus error, the task is suspended and the rest of the system continues uninterrupted. However, when an ISR causes such an exception, there is no safe recourse for the system to handle the exception. The ISR has no context that can be suspended. Instead, VxWorks stores the description of the exception in a special location in low memory and executes a system restart.

12.7. Quality of Service

none

12.8. Network

VxWorks currently supports loosely coupled network connections over serial lines (using SLIP, CSLIP, or PPP) or Ethernet networks (IEEE 802.3). It also supports tightly coupled connections over a backplane bus using shared memory. The standard VxWorks network stack uses the Internet protocols, based on the 4.4 BSD TCP/IP release, for all network communications.

In addition to the remote access provided by Tornado, VxWorks supports remote command execution, remote login, and remote source-level debugging. VxWorks also supports standard BSD socket calls, remote procedure calls, SNMP, remote file access, boot parameter access from a host, and proxy ARP networks.

Supported protocols: BSD 4.4 TCP/IP IP, UDP, TCP, IGMP, ICMP, ARP RIP 1/2 SLIP, CSLIP, PPP BOOTP, DNS, DHCP, TFTP FTP, RLOGIN, RSH, TELNET

In addition to the standard BSD socket interface, VxWorks also supports zbuf sockets, an alternative set of socket calls based on a data abstraction called the zbuf (the zero-copy buffer). Although this interface is WRS-specific, the interface can communicate with standard BSD sockets. Thus, the other end of the socket connection can use the standard BSD interface even if you chose to use the zbuf interface on the VxWorks side of the connection.

12.9. Filesystems

VxWorks support these filesystems: FAT, NFS, raw, TrueFFS, SCSI support

There is also another file system - The Target Server File System (TSFS). FSFS is a full-featured VxWorks file system, but the files operated on by using the file system are actually located on the host. TSFS uses a WDB driver to transfer requests from the I/O system to the target server. The target server reads the request and executes it using the host file system. Thus when you open a file with TSFS, the file being opened is actually on the host. Future `read()` and `write()` calls on the file descriptor obtained from the `open()` call actually read from and write to the opened host file.

12.10. Trace and debug

Command line (based on gdb) with graphic frontend. Remote debugging is possible via Ethernet or serial line.

WindView is timing debugger which displays detailed information for each event (such as the action that occurred, the context in which the action occurred, and the object associated with the action). In addition, WindView tags certain events with either high-resolution timestamps or event sequence numbers.

At the default logging level WindView shows only the context switches. You can configure WindView to show all task state transitions so that, for example, when a task goes from pended to active state, that event is logged and displayed. Or you can configure WindView to show details of selected objects in instrumented libraries. Instrumented objects include semaphore gives and takes, message queue sends and receives, timer expirations, and signals, as well as task and memory activities.

System trace and event record - yes, using VxView

12.11. Miscelaneous

Development environment: propietary compiler or gcc, graphic IDE (for win32 or Solaris). Remote debugging via serial line or Ethernet. Dynamic module loading/unloading

Fail signal and recovery - setjmp/longjmp

Supported programming languages: C, C++

Compatibility with other RTOS is limited. There are two APIs - VxWorks native specific API and POSIX 1003.1b compliant API, but POSIX API is not fully implemented.

Chapter 13. LynxOS

This document describes the main features of LynxOS®

LynxOS® is used in aeronautics and safety critical systems since it is presented as an higly dependable and fault-tolerant system.

LynxOS® is already available for a large number of processors and does its best to optimize their use, it envolves in particular very good memory management.

LynxOS® is a UNIX®/POSIX conforming RTOS. *It is fully conformant with POSIX interfaces for core services (1003.1), real-time extensions(.1b), and thread extensions(.1c)*

13.1. Hardware characteristics

Supported processors:

A large variety of processors are supported. Among them the most important are :

- Motorola: PowerPC (PPC 601, 603, 604), PowerPC G3 (PPC 75x), PowerPC G4 (PPC 7400,7410,74xx) with Altivec Support, PowerPC IBM 405,440.
- Intel: x86(IA-32) Architecture (and compatible).
- MIPS
- Xscale
- ARM9

13.2. Process management

LynxOS® is a hard-real-Time OS, it is fully preemptible and reentrant. It uses a RT Global Scheduler and implements priority inheritance and priority tracking so that the highest priority thread runs regardless of which process it is in or if it's a kernel thread. Moreover, it provides deterministic response-time for tasks even in the presence of multiple interrupts, the highest priority task will only be interrupted once for each device thanks to the priority tracking policy.

LynxOS® applies a uniform global priorities management:

- Kernel and application threads exist within the same priority space
- Same scheduler queues for kernel and application threads
- High priority application threads are scheduled before lower priority kernel threads

Kernel threads may utilize 1/2 priority steps and priority tracking so they run before the user task they serve, but after higher priority user tasks

Scheduling policy:

Three scheduling policies are available :

`SCHED_FIFO` (first-in,first-out)

Standard POSIX FIFO policy. A preemptable fixed priority scheduler.

`SCHED_RR` (round robin)

`SCHED_OTHER` (Proprietary Lynx scheduling policy named "Priority based quantum")

This is similar to round robin policy, excepted the fact that a configurable time quantum is defined for each level of priority. This time quantum can be set using the `setquantum()` call.

The default value for the time quantum is 640ms. It can be modified, it is the system constant named QUANTUM defined in the configuration file `/usr/include/param.h`

The scheduling policy is modifiable

3 functions `rinsert()`,`redesert()`,`rsched()` are used to manipul ate the different scheduler queues (Ready-Queue,Fast-Ready-Queue,...) and the function `newcontext()` achieves context switching between processes

Periodic threads:

Barriers as defined in POSIX1.d can be used to implement periodic threads

Range of priorities and maximum number of threads:

There is up to 256 levels of priority for the application (user level) and 256 for the kernel. Moreover, half priorities are used for the priority tracking mechanism specific to LynxOS® (see below).

Thread creation and deletion:

Every thread has its own stack, register set, priority, and scheduling algorithm.

The priority is inherited from the caller of `pthread_create()`.The schedpolicy attibute is `SCHED_FIFO`.

Supports thread control and cancellation model as defined by POSIX 1003.1c threads.

13.3. Memory management

Protected address spaces

Conventional UNIX® protections exist between application threads of different processes. Threads of a process share the virtual address space of that process (excellent for IPC and I/O). Application threads execute in the address space of a conventional process, kernel threads execute in the kernel's address space.

LynxOS® exploits very well hardware MMU from the processor MMU, so that each process has it's own virtual memory space perfectly protected. This is important to guaranty QoS and to built robust systems. Moreover paged style MMU eliminates system wide memory fragmentation.

Kernel data structures are protected thanks to user/kernel mode as in Linux. User processes "trap" into the kernel to execute system calls. Kernel/user mode is supported directly by microprocessor privilege levels. User processes are limited in memory regions they can access and instructions they can execute. Kernel can acces all memory regions.

Kernel threads minimize the time spent in H/W context

Dynamic memory allocation:

`malloc()` and `free()` functions are available.

13.4. Inter-Process communication

Semaphores

Provides POSIX semaphores (counting semaphores).

Mutex

Standard POSIX mutexes and condition variables.

Priority inversion control

No priority inversion but specific inheritance mechanism called priority tracking (see driver section).

Simple inheritance, immediate ceiling, etc.

Priority inheritance is implemented via binary semaphores. When a high priority task wants a resource held by a lower priority task, the lower priority task's priority is boosted to that of the higher priority task until the resource is released. This mechanism is used in the LynxOS® kernel and is available to applications and drivers.

The default protocol attribute of `pthread_mutextattrr_t` is `PTHREAD_PRIO_INHERIT` so the prioceiling attribute is irrelevant.

Messages queues

yes

Synchronous or/and Asynchronous, prioritized messages, etc.

Mailboxes

yes.

Shared memory

Them `mmap()` support for regular files and shared memory.

LynxOS provides a thread safe version of the standard 'C' library with extensions for higher performance.

13.5. Time and timers

Time resolution

Users can configure ticks per second for real-time clocks. The define `TICKSPERSEC` in the `/usr/include/conf.h` file defaults to 100 ticks per second (which leads to 10ms between ticks).

Recommended minimum and maximum ticks per second are 20 (50 ms between ticks) and 500 (2ms between ticks).These numbers can vary depending on a systems hardware limitations.

Facilities to add new hardware timers

It is possible to add hardware timers, they are handled by specific drivers interfaced with the POSIX timer.

13.6. Driver programming

Drivers can use POSIX-style threads of execution within the kernel for interrupt handling. LynxOS® treats these threads like normal users threads with software priorities not interrupt priorities. The driver interrupt handler does a minimum of work and signals the kernel thread that interrupt-related data is available.

LynxOS® implements priority tracking. Kernel threads begin their existence with a very low priority as created by a driver. When a user thread opens the device, the kernel thread promotes its own priority and inherits the priority of the user thread opening the device. If another user thread of higher priority opens the device, the kernel thread bumps its priority up to match the other thread; when I/O is complete the kernel thread returns to the next pending threads's priority level, or to its starting level. Kernel threads may use 1/2 priority steps and priority tracking so they execute before the user task they serve, but after higher priority user tasks.

13.7. Quality of Service

Every OS component is designed for absolute determinism. This means that they absolutely must respond within a known period of time. This predictable response is ensured even in the presence of heavy I/O due to the kernel's threading model enabling interrupt routines to be extremely short and fast.

13.8. Network

TCP/IP Technology has been available for many years. Based upon FreeBSD 4.2 network stack, it includes high level features (IPSec,IPv6, Integrated firewall, NAT (Network Address Translation).

Zebra routing protocols are supported.

SNMP support is available.

Other network facilities supported are : DHCP,NTP,XNTP,OpenSSL,NFS,Samba.

13.9. Filesystems

List of block filesystems supported (if any): ROMFS, RAMFS, Flash, NFS, etc.

Access from and to DOS possible using `ntools`.

13.10. Trace and debug

A LynxOS® integrated version of `gdb` called *Total/db* is available to help debugging embedded/real-time applications. It is included in the LynuxWorks Open DEvelopment Environment (ODE) described in the next section.

LynuxWorks has extended gdb capabilities to include :

- Multi-threaded applications debugging:
  - LynxOS® thread ID display
  - Thread context switching
  - Thread-specific breakpoints
- System and device driver debug (with LynusWorks skdb Simple Kernel Debugger)
- Remote network and serial target connections
- Cross hosted and LynxOS® native debugging
- Optional INSIGHT graphical debugger user interface

13.11. Miscelanea

The graphic environment is compatible with the X11 and Motif® standards.

A large offer of development and debugging tools are available (see http://www.lynusworks.com) The LynuxWorks Open Development Environment (ODE) includes a variety of open-source tools and utilities, including many derived from the Free Software Foundations's GNU family.The *GNU Toolchain*. A complete suite of open-source GNU solutions

GNU compilation and debugging tools including standard gcc g++ ANSI C and C++ compilers as well as the `gas` assembler for the PowerPc family. Versions of GNAT are available from Ada Core Technologies and g77 from LynuxWorks ftp site. Other Ada compilers from Rational, Aonix,Irvine Compiler and DDCI are available.

Exceptions handling are managed by the kernel. Specific High Availability Packages can be purchased from LynusWorks, they provide enhanced capabilities for this purpose.

LynxOS® provides API and ABI compatibility especially with Linux® Kernel v2.4.x.

LynxOS® is a UNIX® / POSIX conforming RTOS. It is fully conformant with POSIX interfaces for core services (1003.1), real-time extensions(.1b), and thread extensions(.1c). Moreover, it provides ABI compatibility with Linux®. Linux® application binaries can run unchanged in the LynxOS® environment without necessitating source code recompilation thanks to enhanced API compatibility and specific Dynamic Linked Libraries.

It has over 150 system calls that are similar with the Linux® system call API. In order to enhance compatibility, some system calls have been added to LynxOS® API (e.g. fchdir, setresuid, setresgid, wait4). Signal numbers were changed to match with the Linux® numbering scheme, constants used for IOCTL were changed to match Linux®, the error numbers returned by systm calls (errno) were changed to match Linux®.

LynxOS® v4.0 uses a modified version of the Linux® GLIBC dynamic library to resolve application calls at run-time. This modified GLIBC library provides the translation between Linux® and LynxOS® system calls. This permits the use of a broad spectrum of Linux® system calls including Pthread interfaces,Asynchronous IO, Networking and other POSIX interfaces.

LynxOS®-Linux® ABI compatibility is currently dependent on a Linux® kernel and GLIBC pair. LynxOS® v4.0 supports applications based on Linux® kernel v2.4.x and GLIBC v2.2.2/v2.2.4

New facilities have been added to facilitate dynamic library linkage. Build on relocation capabilities of ELF it may save considerable memory if many applications reuse a same code base. It also helps maintenance upgrades because applications need not be recompiled when the library is updated (providing API doesn't change as well).

LynxOS® doesn't support lazy linking. A spawn application is always linked completely at initialization. Default in LynxOS® is statically linked applications.

`dlopen()` and other standard interfaces are also supported.

13.12. Modularity

The ancient LynxOS® monolithic kernel has become a real micro_kernel on which KPIs (Kernel Plug-Ins) can be added. By default the micro-kernel offers the start-up and shutdown services, the low-level memory management,the interrupts management, synchronisation.

Bibliography

POSIX

[POSIX4] Bill Gallmeister, 1st Edition September 1994, O'Reilly Associates, Inc., *POSIX.4 Programming for the Real World*, 1-56592-074-0.

[POSIX] 2001, *The Open Group Base Specifications Issue 6. IEEE Std 1003.1-2001*.

OSEK/VDX

[osek] 2001, *OSEK/VDX Operating System Specification*, 2.2.

Linux

[ULK] Daniel P. Bovet and Marco Cesati, 2001, O'Reilly Associates, Inc., *Understanding the LINUX Kernel*, 0-596-00002-2.

[ToLinux] Bill Weinberg and Marco Cesati, MontaVista Software, Inc., 2001, *Moving from a Proprietary RTOS To Embedded Linux®*.

[POSIX4] Bill Gallmeister, 1st Edition September 1994, O'Reilly Associates, Inc., *POSIX.4 Programming for the Real World*, 1-56592-074-0.

[LowLat] Andrew Morton, *Linux Scheduling Latency (http://www.zip.com.au/~akpm/linux/schedlat.html)* .

[kpreem] Robert Love, *The Linux Kernel Preemption Project (http://kpreempt.sourceforge.net)* .

[TimeSys] TimeSys, *TimeSys Linux*.

[FT] Systems Software Lab --- Oregon Graduate Institute , *The Firm Timers Home Page (http://www.cse.ogi.edu/~luca/firm.html)* .

RTLinux

[RTLinux1] Michael Barabanov, 2001, FSM Labs, Inc., *Getting Started with RTLinux*.

[RTLtutorial] Ismael Ripoll, 2000, *Tutorial del API de RTLinux*.

RTAI

[RTAIGuide] Paolo Mantegazza, E. Bianchi, L. Dozio, Mike Angelo, and David Beal, September, 2000, Lineo, Inc, *DIAPM. RTAI Programming Guide 1.0*.

[Pierre00] Pierre Cloutier, Paolo Mantegazza, Steve Papacharalambous, Ian Soanes, Stuart Hughes, and Karim Yaghmour, November, 2000, Real Time Operating Systems Workshop, *DIAPM-RTAI POSITION PAPER*.

QNX

[WebQNX] *QNX Documentation www.qnx.com*.

Appendix A. GNU Free Documentation License

Version 1.1, March 2000

A.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added mate-

rial, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified

version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.