A survey of Real Time Operating Systems for Embedded Systems Development in Automobiles

M. Tech Seminar Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

R. Vamshi Krishna
Roll No: 04305015

under the guidance of

Prof. Krithi Ramamritham

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgments

I would like to thank **Prof. Krithi Ramamritham** and **Dr. Kavi Arya** for their invaluable support and guidance.

R. Vamshi Krishna

Abstract

Choosing a Real-Time Operating System(RTOS) for an embedded application is a difficult task. There a numerous products which claim to be RTOS. Also for the RTOS to be useful in automobiles it must possess the necessary characteristics.

The purpose of this report is to list out the various aspects in choosing a commercial RTOS for use in automobiles. To that end this report gives a brief description of the OSEK open standard and looks at two real-time operating systems with a view towards use in automobiles.

Contents

1 Introduction to Real-Time Systems **1**
1.1 Introduction 1
1.2 Issues in real-time Computing 1
1.2.1 Task Management and Interrupts 2
1.2.2 Scheduling and Fault-Tolerance 2
1.2.3 Memory Management 3
1.2.4 Communication and synchronization 3
1.3 User Interface 4
1.3.1 API Richness 4

2 OSEK/VDX : An Open-Ended Architecture for automobiles **5**
2.1 Introduction to the Operating System Specification 6
2.1.1 Task Management 6
2.1.2 Processing Levels and priorities 6
2.1.3 Interrupts and events 7

3 A Report published by CSA **8**
3.1 Results of Evaluation 9

4 QNX Neutrino from QNX Software Systems **10**
4.1 Introduction to QNX 10
4.2 Microkernel Architecture 10
4.3 Communication and synchronization 10
4.4 Task Processing and Interrupts 11
4.5 Memory Management Method 11
4.6 User Interface 11
4.7 API Richness 11
4.8 Documentation and Support 11

5 Embedded Cygnus Operating System(eCos) 12
5.1 Introduction 12
5.2 Overview of Features of eCos 12
5.2.1 Open Source and Royalty-Free 12
5.2.2 eCos Kernel 13
5.2.3 Schedulers 13
5.2.4 Synchronization Primitives 14
5.2.5 Threads and Interrupt Handling 14
5.2.6 Application Programming Interface 14
5.2.7 GUI/IDE 14
5.2.8 Documentation and support 15

6 Comparison of QNX and eCos 16

7 Conclusion 17

Chapter 1

Introduction to Real-Time Systems

1.1 Introduction

A real-time system is defined as a system in which the correctness of its output(s) depends not only upon the logical computations carried out but also upon the time at which the results were delivered to the external interface. A good RTOS can be called as one that has bounded (predictable or timely) behavior under all system input scenarios.

In general, a real-time task falls into one of the 3 categories, namely:

- Hard real-time: missing a deadline has catastrophic results for the system.
- Firm real-time: missing a deadline entails an unacceptable quality reduction as a consequence.
- Soft real-time: deadlines may be missed and can be recovered from. The reduction in system quality in soft real-time systems is acceptable.

We will look at the different issues in real-time systems with emphasis on hard real-time systems.

1.2 Issues in real-time Computing

A complex real-time system has, in general, both hard and soft deadlines. Since hard deadlines are critical to the functioning of the system, they must be met even under the worst conditions. As a result the system's performance under worst-case conditions is fundamental to many, if not all, real-time systems.

[10, 9] describe the features of an RTOS that are desirable. We present a few of them along with the features that make them suitable for embedded control systems in automobiles.

1.2.1 Task Management and Interrupts

From the definition of real-time, an application has to be capable of responding in a predictable or timely way to multiple simultaneous external events. This is the basis of multitasking. The level of multitasking depends on the whether the processes are run as a whole or are divided into task or threads.

Threads are preferable over tasks as they have smaller context switching times. This results in faster and timely response to an event. Each thread is assigned a priority by the scheduler or the user (in case of user threads).

Interrupt latency must be minimum, but more essentially time bounded. To that end the routine that service interrupts must be small. Also care must be taken of the situations where interrupts occur when another interrupt is being serviced. There are different ways of doing it and are implementation specific.

Sometimes during the execution of an interrupt it is not preempted(by way of another interrupt). For that reason sometimes interrupts are disabled at the start of servicing an interrupt and re-enabled at the end of servicing. This disabling time must be ideally very small.

It is important to have different priority levels. Different priority levels ensure timeliness. This is done by assigning highest priority to threads (or tasks) that have the earliest deadline. Also threads that ensure safety in safety-critical systems need the highest priority. Generally there should be at least 128 priority levels.

1.2.2 Scheduling and Fault-Tolerance

The scheduler one of the basic parts of the OS. It has the function of switching from one task, thread or process to another. As this activity constitutes overhead it must be done as quickly as possible.

There are many algorithms for scheduling in real-time systems. [9] describes the some of the different paradigms and their related algorithms. The different paradigms are :

- Static table-driven approaches
- Static priority driven pre-emptive approaches
- Dynamic planning based approaches
- Dynamic best effort approaches

One of the metrics used to quantify a scheduling algorithm is predictability offered by it. Predictability is the ability of the scheduler to predict that the tasks will meet their deadlines.

Fault tolerance is an important issue in any RTOS because by definition if a deadline is missed, the results would be catastrophic. Thus a good RTOS must be able to recognize in advance when a deadline is going to be missed and take appropriate counter-measures [9].

One of the approaches to fault-tolerance is to have a backup schedule in case of failure. If there is no failure, the primary task gets executed, the alternative task is not executed and the time allocated for the alternative task is reused. Otherwise in case of a failure, a somewhat degraded version of the task is run. But this approach results in increased processing in terms of computing the alternative schedule and having scheduling them well beforehand such that in case the primary task fails the alternative task still has enough time before the deadline.

1.2.3 Memory Management

Each program needs to be held in memory to be executed. Generally the program may be placed in RAM. Also threads, tasks, queues(mailboxes) and many other operating system constructs require memory.Data kept in memory can be accessed directly or using memory management techniques. The problem with memory management techniques is the time constraint.

The fundamental requirement for memory in a real-time system is that memory access must be bound(i.e. predictable).Hence use of demand-paging in real-time systems is prohibited. Also care must be taken to load the associative maps in virtual memory systems onto the processor and lock them so that swapping of the pages does not take place. Generally systems do not use memory management in the final applications but use them to debug until the application is ready for deployment.

1.2.4 Communication and synchronization

Different threads communicate when they execute. They synchronize to make sure that all the exchanges occur at the right moment, under the right conditions and they do not get into each other's ways. Also care must be taken so that messages reach the target in the required time.

Communication between different processes can be typically using:

- Message Queues
- FIFO's and Pipes
- Mailboxes
- Shared Memory

Also different types of issues like multiple senders and multiple receivers must be taken into consideration.

1.3 User Interface

This feature does not directly affect the functioning of the real-time system. But it is essential to develop programs that have good real-time characteristics. In general an RTOS must provide a good GUI so that users/developers can use it to develop good applications.

1.3.1 API Richness

Simple applications would need a very few system calls. Complex applications might require many system calls. Also the question of implementing a part of the software as part of the OS or application is an important one.

The API provided by an RTOS, like the GUI, will not affect the execution time of a program but will enhance the ease of usage of the RTOS. Good API results in enhancement of the ability on the user's/developer's part to write good programs. Hence RTOS must provide a rich set of system calls and also user-level functions. Also it must provide support for API standards like POSIX which help in porting of code.

Chapter 2

OSEK/VDX : An Open-Ended Architecture for automobiles

OSEK/VDX is a standard for open-ended architecture for distributed control units in vehicles. It is defines a single processor operating system meant for distributed embedded control units. As a complete description of the specification is not possible, it's outline is given briefly.

The OSEK/VDX specification is divided into four areas called Conformance Classes (CC) and described by four documents. [2] describes the overall relationships or dependencies between the classes. Each class has its own specification.

The classes are:

- Operating system [6]
- Communication [3]
- Network management [5]
- OSEK/VDX Implementation Language [4]

The Operating system document describes the features that an operating system must possess to be suitable for automobiles. These include the areas of task processing, memory management, interprocess communication.

The Communication document describes the services to be offered by the operating system for communication between tasks and interrupt service routines(ISRs). These tasks or ISRs can be in the same control unit or different unit. Communication is via messages and this communication is possible only through the specified Application Program Interface (API).

The Network document describes the services by which different control units can be networked together. For this the operating system must provide standardized interfaces.

The OSEK/VDX Implementation Language document describes the mechanisms to configure an OSEK application inside a particular CPU.

2.1 Introduction to the Operating System Specification

Automotive applications are characterized by stringent real-time constraints. Hence the OSEK operating system specification offers the necessary functionality to support event driven control systems.

One of the goals of OSEK is to support the portability and re-usability of application software. Hence the interface between application programs and operating system is defined by standardized system services with well defined functionality.

2.1.1 Task Management

As the operating system is intended for use in any type of control units, it has to support time-critical applications on a wide range of hardware. Hence dynamic generation of objects in time-critical applications has been avoided. Instead the generation of system objects is assigned to system generation phase.

Basically OSEK defines two types of tasks.

- Basic tasks
- Extended tasks

Basic Tasks

Basic tasks are those that release the processor only if

- they terminate,
- the operating system switches to a higher priority task or,
- an interrupt occurs causing the processor to switch to an ISR.

Extended Tasks

Extended tasks are different from basic tasks in that they are allowed to use the system call *WaitEvent*, which may result in a waiting state.

2.1.2 Processing Levels and priorities

In the OSEK operating system there are basically two types of entities that compete for CPU. They are :

- Interrupt Service Routines(ISR) managed by the operating system.
- Tasks

The hardware resources of a control unit can be managed by operating system services. These services are called by a unique interface, either by the user program or internally by the operating system.

There are three processing levels in OSEK :

- Interrupt Level
- Logical level for Scheduler
- Task (Basic and Extended Task) Level

Within each task level, the tasks are scheduled (non or full or mixed preemptive scheduling) according to their user assigned priority. The priorities of tasks are statically assigned by the user.

For efficiency reasons OSEK does not implement dynamic priority management. All priorities of all tasks are defined statically i.e. the user cannot change it at the time of execution. OSEK prescribes the OSEK Priority Ceiling Protocol for avoid priority inversion.

2.1.3 Interrupts and events

Interrupts

The functions to process the interrupts are subdivided into two categories :

- **ISR Category 1** : The ISR does not use operating system service. Thus they have least overhead.
- **ISR Category 2** : The user can provide his own routine to be executed when this interrupt occurs.

Also the operating system provides system functions like *EnableAllInterrupts*, *DisableAllInterrupts*, *ResumeAllInterrupts*, *SuspendAllInterrupts* etc. These system calls have small latencies.

Event mechanism

Event mechanism is a means of synchronization and is provided only for extended tasks. Events are objects managed by the operating system.

Chapter 3

A Report published by CSA

In June 2003, the Canadian Space Agency(CSA) published a report on the process of selecting an RTOS for embedded systems development in satellite applications[8]. In the report a total of 48 Real-Time Operating Systems were evaluated and 20 of those were selected for ranking and others rejected.

The evaluation was done on the following categories :

- Kernel
- Scheduling
- Process/Thread/Task Model
- Memory
- Interrupt and Exception Handling
- Application Program Interface
- Development Information
- Commercial Information

The evaluation was performed with a view towards space-applications, but are equally applicable to automotives where hard real-time deadlines exist.

3.1 Results of Evaluation

The results showed that four RTOS are in the forefront. They are

- QNX Neutrino from QNX Software Systems
- OS-9 ^{TM}
- Precise/MQX ^{TM}
- OSE ^{TM}

Also VxWorks from Wind River, which is the most popular and well known RTOS, is placed 3 rd behind 9 others. eCos is the only open source software to make it to the top 20.

In the following chapters we will be looking at two RTOS namely QNX Neutrino from QNX software systems and eCos from Free Software Foundation(FSF). Also many of their features will also be discussed.

Chapter 4

QNX Neutrino from QNX Software Systems

4.1 Introduction to QNX

QNX Neutrino from QNX Software Systems is a micro-kernel based RTOS.

As it is micro-kernel based, only the most fundamental OS primitives are handled in the kernel. All other components drivers, file systems, protocol stacks, user applications run outside the kernel as separate, memory-protected processes. As a result any component failure does not affect the kernel or other processes. It is highly fault tolerant.

4.2 Microkernel Architecture

The micro-kernel implements only the core services like threads, signals, message passing, synchronization, scheduling and timer services [7]. The micro-kernel itself is never scheduled. Its code is executed as a result of a system call, the occurrence of an interrupt or a processor exception.

Being a microkernel the size of applications can be very small. Thus it is suitable for small footprint applications.

4.3 Communication and synchronization

QNX Neutrino RTOS is a message-based operating system. The message passing service is based on client-server model: the client sends a message to the server who replies with the result. Queues and mailboxes are the communication objects available.

Counting Semaphores, Binary Semaphores, Mutexes, Condition variables, Event flags are, as in most systems, available for synchronization. In addition to these there are spinlocks, joined threads, reader/writer locks, sleepons, barriers etc..

4.4 Task Processing and Interrupts

There are 63 different thread priority levels available to applications.The scheduler schedules the threads according to a prioritized FIFO, round-robin algorithm and a new method called “Sporadic Scheduling”. This new method of scheduling is a sort of adaptive scheduling.

The interrupt redirector in the micro-kernel handles the interrupts in their initial stages. This redirector saves the context of the currently running thread and sets the processor context such that the ISR has access to the code and data that are part of the thread the ISR is contained within(i.e this is the thread that attached the ISR).

QNX Neutrino also supports interrupt sharing. When an interrupt occurs, every interrupt handler attached to the hardware interrupt is executed in turn. Interrupts are not disabled during the execution of an interrupt handler. Thus interrupts can be nested.

4.5 Memory Management Method

In QNX Neutrino every process has its own virtual memory space. The virtual memory is supported by the paging mechanism of the processor. Virtual memory protects processes (both user and system processes) from each other, enhancing the overall robustness of the system. It does not implement demand paging.

4.6 User Interface

QNX Neutrino has an excellent GUI called photon microGUI that makes it easy to build, debug and maintain applications. Modules can be inserted and deleted using the GUI.

4.7 API Richness

The QNX Neutrino provides both a POSIX-compliant and proprietary API. The API is geared towards message-based systems, which is a natural match for the system architecture.

4.8 Documentation and Support

QNX Neutrino comes with excellent documentation and has wonderful support. The documentation keeps improving with the release of newer versions.

Chapter 5

Embedded Cygnus Operating System(eCos)

5.1 Introduction

eCos is an open source, configurable, portable, royalty-free embedded real-time operating system. Developers have full and unfettered access to all aspects of the runtime system. As no part is hidden or proprietary, users are at liberty to examine, add to, and modify the code as deemed necessary.

One of the key technological innovations of eCos is the configuration system. The configuration system allows the application writer to impose their requirements on the run-time components, both in terms of functionality and implementation, whereas traditionally the operating system has application writers own implementation. This enables eCos developers to create their own application-specific operating system and makes eCos suitable for a wide range of embedded uses.

Configuration ensures that the resource footprint of eCos is minimized as all unnecessary functionality and features are removed. The configuration system also presents eCos as a component architecture. This provides a standardized mechanism for component suppliers to extend the functionality of eCos and allows applications to be built from a wide set of optional configurable run-time components.

5.2 Overview of Features of eCos

5.2.1 Open Source and Royalty-Free

The royalty-free nature of eCos means that users can develop and deploy their application using the standard eCos release without incurring any royalty charges. In addition, there are no up-front license charges for the eCos runtime source code and associated tools.

5.2.2 eCos Kernel

The kernel is one of the key packages in all of eCos. It provides the core functionality needed for developing multi-threaded applications:

1. The ability to create new threads in the system, either during the startup or when the system is already running.
2. Control over the various threads in the system.
3. A choice of schedulers, determining which thread should concurrently be running.
4. A range of synchronization primitives, allowing threads to interact and share data safely.
5. Integration with the system's support for interrupts and exceptions.

Unlike some of the other kernels, memory allocation in eCos is handled my a separate package. Similarly each device driver would be a separate package.The various packages are combined and configured by using the eCos configuration technology to meet the needs of the application.

The eCos kernel package is optional. It is possible to write single-threaded applications which do not use any kernel functionality. Such applications are based around a central polling loop, continually checking all devices and taking appropriate action. It is also possible to write multi-threaded applications that use a full-fledged kernel and provide multitasking.

5.2.3 Schedulers

When a system involves multiple threads, a scheduler is needed to determine which thread should currently be running. The eCos kernel can be configured with one of two schedulers, the bitmap scheduler and the multi-level queue (MLQ) scheduler.

Bitmap Scheduler

The bitmap scheduler allows only one thread per priority level, limiting the total no. of threads available to the no. of priority levels. A simple bitmap can be used to determine which threads are currently runnable. Bitmaps also can be used to keep track of threads waiting on some synchronization primitive like mutexes. Bitmaps are fast and totally deterministic.

Multi-level Queue(MLQ)

MLQ scheduler allows multiple threads at the same priority. Also some kernel functionality like SMP support and mutex priority ceiling or priority inversion is available only with MLQ scheduler.

5.2.4 Synchronization Primitives

eCos provides a number of different synchronization primitives like mutexes, condition variables, semaphores, mail boxes and event flags.

Mail boxes are used to indicate that a particular event has occurred and allows for one item of data to be exchanged per event. Typically this item of data would be a pointer to some data structure. Because of the need to store this extra data, mail boxes have finite capacity.

5.2.5 Threads and Interrupt Handling

During normal operation the processor will be running one of the threads in the system. This may be an application thread, a system thread running inside say the TCP/IP stack, or the idle thread. From time to time a hardware interrupt will occur, causing control to be transferred briefly to an interrupt handler. When the interrupt has been completed the system's scheduler will decide whether to return control to the interrupted thread or to some other runnable thread.

Instead the kernel uses a two-level approach to interrupt handling. Associated with every interrupt vector is an Interrupt Service Routine or ISR, which will run as quickly as possible so that it can service the hardware. However an ISR can make only a small number of kernel calls, mostly related to the interrupt subsystem, and it cannot make any call that would cause a thread to wake up. If an ISR detects that an I/O operation has completed and hence that a thread should be woken up, it can cause the associated Deferred Service Routine or DSR to run. A DSR is allowed to make more kernel calls, for example it can signal a condition variable or post to a semaphore.

5.2.6 Application Programming Interface

eCos kernel can be programmed in two ways. The kernel has its own C API, with functions like *cyg_thread_create* and *cyg_mutex_lock*. These can be directly called from the application code or from the additional packages. Alternatively there are a number of packages which provide compatibility with existing API's, for example POSIX threads or μITRON. These allow application code to call standard function such as *pthread_create*, and those functions are implemented using the basic functionality provided by the eCos kernel. This aids in code reuse and sharing of code.

5.2.7 GUI/IDE

eCos comes with an IDE that is easy to handle and eases the task of configuring the system. We can choose a build time using the GUI, various parameters like choice of scheduler etc..

5.2.8 Documentation and support

eCos being open source does not have much in terms of support except through mailing lists. But it has excellent documentation.

Chapter 6

Comparison of QNX and eCos

We present a comparison of QNX and eCos operating system based on their applicability to embedded systems development.

Feature	**QNX Neutrino**	**eCos**
Installation	easy	not easy
Small size	yes	yes
Fast context switches	yes	yes
Response to External interrupts quickly	yes	yes
virtual memory	Yes	No
Interrupt latencies	small	small
Scheduling mechanism	Sporadic Scheduling	Bitmap or MLC
Support	Very Good	Only via mailing lists

Chapter 7

Conclusion

QNX Neutrino is an excellent real-time operating system for use in developing embedded applications. It is robust, fault tolerant and has excellent technical support making it ideal for use in commercial applications.

eCos is a very fast growing real-time system. Due to its open source nature it can be modified to suit the needs of an application. Also it is in constant development making it a good prospect for designing embedded applications.

Hence from the discussion it can be concluded that if a proprietary real-time operating system can be afforded or deemed necessary then QNX Neutrino is a very good option. If the requirement is of an open source RTOS then eCos is the best choice.

Bibliography

[1] *eCos Documentation*. http://ecos.sourceware.org.

[2] *OSEK/VDX Binding Document v1.4*.

[3] *OSEK/VDX Communication(COM) Specification v2.2.2*.

[4] *OSEK/VDX Implementation Language(OIL) Specification v2.5*.

[5] *OSEK/VDX Network Management(NM) Specification v2.5.3*.

[6] *OSEK/VDX Operating System Specification v2.2.2*.

[7] QNX Neutrino RTOS v6.2 Evaluation Report. Technical report, August 2002.

[8] Philip Melanson and Siamak Tafazoli. A Selection Methedology for the RTOS Market. Technical report, Canadian Space Agency, June 2003.

[9] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of the IEEE*, pages 55–67, Jan 1994.

[10] Dedicated Systems. What makes a good RTOS?, June 2001.