ISSN 0280-5316
ISRN LUTFD2/TFRT--5731--SE

Investigation of Real-Time Operating Systems : OSEK/VDX and Rubus

Pontus Evertsson

Department of Automatic Control
Lund Institute of Technology
December 2004

Department of Automatic Control **Lund Institute of Technology** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER THESIS *Date of issue* December 2004 *Document Number* ISRNLUTFD2/TFRT--5731--SE
Author(s) Pontus Evertsson	*Supervisor* Karl-Erik Årzén at LTH in Lund Clas Emanuelsson at Haldex Traction in Landskrona. *Sponsoring organization*

Title and subtitle

Investigation of Real-Time Operating Systems: OSEK/VDX and Rubus (Analys av realtidsoperativsystem: OSEK/VDX and Rubus)

Abstract

The aim of this work was to investigate the possibilities and consequences for Haldex Traction of starting to use the OSEK/VDX standard for realtime operating systems. This report contains a summary of the realtime operating system documents produced by OSEK/VDX. OSEK/VDX is a committee that produces standards for realtime operating systems in the European vehicle industry. The report also contains a market evaluation of different OSEK/VDX realtime operating systems. The main differences between OSEK/VDX OS and a realtime operating system named Rubus OS are also discussed. There is a design suggestion of how to change an application that runs under Rubus OS to make it work with an OSEK/VDX OS. Finally a test of changing a small test application's realtime operating system from Rubus OS to the OSEK OS osCAN is presented.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 48	*Recipient's notes*
Security classification		

Contents

Contents 1
Acknowledgement 4
1 Introduction 5
  1.1 Motivation for this work 5
  1.2 Haldex AB 5
    1.2.1 Haldex Traction 5
      The coupling 6
        Basic function of the coupling 6
        Electronic control unit 7
        Software architectural design 7
  1.3 Outline of the report 8
2 OSEK/VDX 9
  2.1 Goal with OSEK/VDX 9
  2.2 OSEK/VDX Operating System 9
    2.2.1 Architecture of the OSEK/VDX operating system 9
      Task management 10
        Extended tasks 10
        Basic tasks 10
      Conformance classes 10
    2.2.2 Scheduling 11
      Non preemptive scheduling 11
      Full preemptive scheduling 11
      Groups of tasks 11
      Mixed preemptive scheduling 11
    2.2.3 Application modes 12
    2.2.4 Interrupt processing 12
    2.2.5 Event mechanism 12
    2.2.6 Resource management 13
      Problems with synchronisation mechanisms 13
        Priority inversion 13
        Deadlocks 14
      OSEK Priority Ceiling Protocol 14
    2.2.7 Alarms 14
    2.2.8 Error handling, tracing and debugging 15
  2.3 OSEK/VDX Time-Triggered 16
    2.3.1 Task management 16
    2.3.2 Interrupt processing 16
    2.3.3 Synchronisation 17
    2.3.4 Inter-Task Communication 17
    2.3.5 Error handling 17
  2.4 OSEK/VDX Communication 17
    2.4.1 Requirements 17
    2.4.2 Communication concept 17
    2.4.3 Interaction Layer 18
      Overview 18
      Message reception 18
      Message transmission 18
      Notification 19

Error handling 19
2.4.4 Conformance classes 19
2.5 OSEK/VDX Network Management 20
2.6 OSEK/VDX Implementation Language 20
2.6.1 General concept 20
3 Some OSEK/VDX RTOS 22
3.1 Why OSEK/VDX 22
3.2 Important factors of OSEK OS for Haldex Traction 22
3.3 Eight OSEK/VDX RTOS 23
3.3.1 Eurosmot 23
3.3.2 Ercosek 23
3.3.3 OSEKturbo 24
3.3.4 RTA-OSEK 24
3.3.5 ProOSEK 25
3.3.6 Nucleus OSEK 25
3.3.7 OX-OSEK 26
3.3.8 osCAN 26
3.4 Evaluation of the OSEK/VDX candidates 27
4 Rubus OS, Arcticus Systems 29
4.1 Rubus OS architecture 29
4.1.1 The Basic Services 29
Clocks and Timers 29
Event Log Services 30
Basic Queue Services 30
4.1.2 The Red Kernel Services 30
Red Schedule 30
Red Error Handling 30
4.1.3 The Blue Kernel Basic Layer Services 30
Blue scheduling 30
Blue Kernel Threads 31
Blue Error Handling 31
Signals 31
Interrupt Control 31
4.1.4 The Blue Kernel Thread Co-operation Layer Services 31
Mutex 31
Message Passing 32
4.1.5 The Blue Personality Services 32
4.2 Some differences between OSEK/VDX OS and Rubus OS 32
4.2.1 Scheduling 32
4.2.2 Periodic activities 32
4.2.3 Resource management 33
4.2.4 Application modes 33
5 Conversion from Rubus OS to OSEK/VDX 34
5.1 Changes in the scheduling 34
5.2 Changes in the resource management 35
5.3 Different application modes 35
5.4 Translation of the operating system calls 36
6 Test application 40
6.1 Introduction 40
6.2 The test hardware and OSEK OS 40

6.3 The test application 40
6.4 Two different approaches 42
6.4.1 Using Haldex Traction's makefiles 42
6.4.2 Using osCAN's makefiles 43
Summary 44
References 45

Acknowledgement

The report is the master's thesis of the Computer Science and Engineering Programme at Lund Institute of Technology.

Several people have helped me with my report. I want to thank my supervisor Professor Karl-Erik Årzén at the Department of Automatic Control, LTH and my other supervisor Clas Emanuelsson at Haldex Traction.

I also want to thank the staff at the TTE division of Haldex Traction for their help with my work.

I want to thank all the RTOS vendors mentioned in this report. Without their information and help the work would have been impossible.

Finally I want to thank Håkan Larsson for the help with the proofread and I want to thank Malin Bengtsson for not getting tired of the technical monologue.

1 Introduction

1.1 Motivation for this work

Since it becomes more and more common in the car industry with networks of different control units made by different distributors in the cars, standards for interfaces and protocols become more and more important. One industry standard, which grows mostly in the European vehicle industry, is OSEK/VDX.

Haldex Traction is a distributor of a limited slip coupling. This product contains an electrical control unit that reads information from a data bus in the car it is installed in and therefore the car manufacturers may enforce Haldex Traction to join OSEK/VDX in the future.

Haldex Traction's intensions with this master thesis was to deepen their knowledge about OSEK/VDX, find the best suited realtime operating system with support for OSEK/VDX and to be better prepared if Haldex Traction later on wants to change their present realtime operating system to one that supports the OSEK/VDX standard.

1.2 Haldex AB

Haldex [15] is an innovator in the vehicle business area. They provide proprietary systems and components for cars, trucks and industrial vehicles on the world market. The Haldex Group consists of four different companies in four product areas. The companies are:

- **Haldex Brake Systems:** Produces subsystems and components for air brake and suspension systems for commercial vehicles. Some examples of products are ABS systems, automatic and manual brake adjusters and disc brakes.
- **Haldex Hydraulic Systems:** This division mainly produces hydraulic systems for steer and lift systems for forklifts, construction equipment, and trucks et cetera.
- **Haldex Garphyttan Wire:** Produces steel-alloyed spring wire products for applications with precise performance demands.
- **Haldex Traction:** Produces an All Wheel Drive (AWD) system for passenger cars.

The group has 4 100 employees and production in North America, South America, Europe, India and China.

1.2.1 Haldex Traction

Haldex Traction [16] was founded in 1998. Its headquarter is placed in Landskrona and it has one support office in Detroit, USA. Haldex Traction has 200 employees. The customers are the VW-group, Ford and General Motors. Currently Volkswagen, Audi, Skoda, Seat, Volvo and Bugatti use the AWD-system.

The coupling

Haldex Limited Slip Coupling (LSC) is installed in the car to distribute the torque between the front and the rear axis (Figure 1-1). It can be installed both in front driven and rear driven vehicles. The coupling is based on a Swedish patent acquired by the Haldex Group.

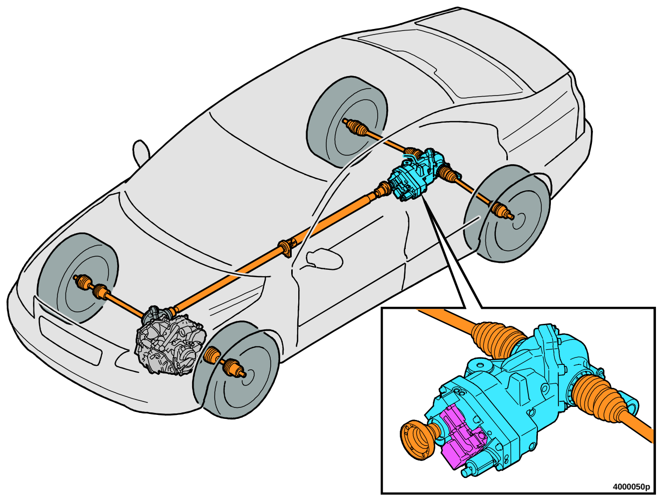

Figure 1-1 Haldex LSC in a front driven car.

Basic function of the coupling

Haldex LSC comprises three different functional parts:

- A hydraulic pump driven by the slip between the front and rear axis.
- A wet multi-plate clutch.
- A computer-controlled throttle valve.

A bit simplified, the unit is a hydraulic pump in which the housing and the annular piston are connected to one shaft and the piston actuator is connected to the other. When a speed difference occurs between the two shafts (the rear and the front axes) the pumping starts immediately and generates an oil flow (Figure 1-2). This oil flow compresses the wet multi plate clutch causing the speed difference between the axes to decrease. The oil returns to a reservoir via a controllable valve. The oil pressure, that decides the torque distribution between the axes, can be adjusted with the help of the controllable valve.

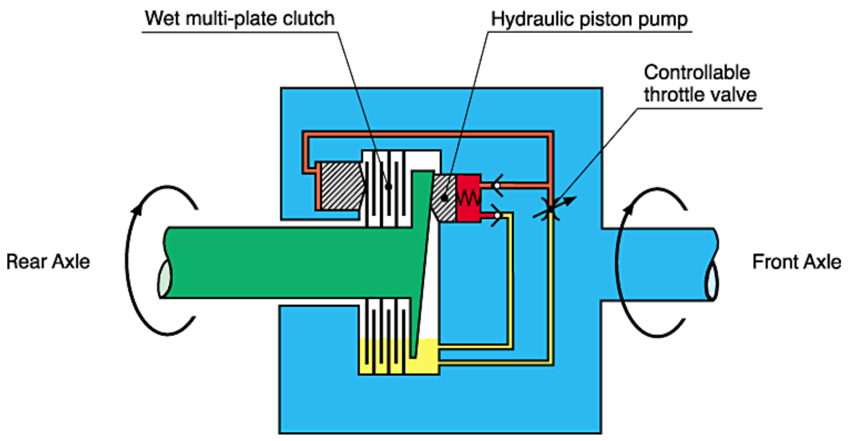

Figure 1-2 Simplified picture of Haldex LSC

Electronic control unit
To control the valve the electrical control unit (ECU) Infineon C167cs is used. It's a 16-bit RISC controller with 20 MHZ CPU clock and 256 kB flash memory.
Through the valve the software is able to control the torque transfer characteristics.

The software consists of two parts with different purposes, the base software and the application software.

The goal of the base software is to control the internal functions of the coupling, for example to compensate for differences in the viscosity of the oil caused by variations in the operating temperature. Another example of a task for the base software is to control a small electric pump that provides the coupling with a low oil pressure to ensure minimum activation time.

The application software's function is to make the coupling behave in a desired way in every different driving situation. In a roundabout one specific behaviour is desired and when parking another. The state of the car, or the driving situation, is determined with the help of information from other active systems in the car via a data bus.

Software architectural design
Haldex Traction's software system is decomposed into four different main subsystems: Diagnostics application, Control application, Strategic control and Base software.

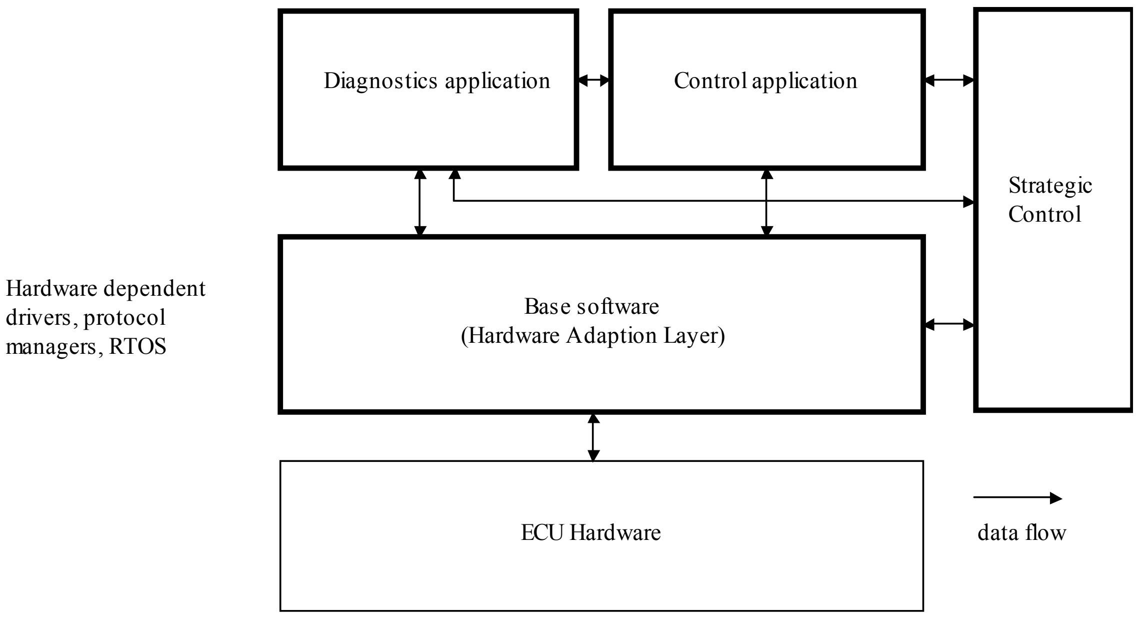
Diagnostics application
Control application
Strategic Control
Hardware dependent drivers, protocol managers, RTOS
Base software (Hardware Adaption Layer)
ECU Hardware
data flow

Figure 1-3 Haldex Traction's software architecture

- **Base software:** It contains all hardware dependant drivers. It also contains hardware adaptation and protocol managers.
- **Strategic Control:** This is the supervisory part of the system. It monitors the other parts of the software and reacts on errors. If a serious error is detected, the strategic control is able to stop other parts of the software from executing and puts the coupling in a safe state.
- **Control application:** It contains the control laws that controls the stiffness of the coupling, and therefore also the vehicle dynamics.

- **Diagnostics application:** It contains the code that communicates with tester equipment, and executes commands from such equipment. If the diagnostics application wishes to control the coupling, it makes a request to the strategic control, which decides if this is allowed. The diagnostics application also collects error information and stores errors to the error log.

All the subsystems in Haldex Traction's application use the realtime operating system Rubus OS.

1.3 Outline of the report

The OSEK/VDX standard is described in chapter 2. This includes a summary of the standard documents OSEK/VDX Operating System, OSEK/VDX Time-Triggered, OSEK/VDX Communication, OSEK/VDX Network Management and OSEK/VDX Implementation Language. In chapter 3 there are eight different OSEK/VDX realtime operating systems presented. The operating systems are also ranked according to the qualities of an OSEK/VDX realtime operating system that are most important for Haldex Traction. The main differences between an OSEK/VDX realtime operating system and Rubus OS are described in chapter 4. Chapter 5 provides a design suggestion for how to convert Haldex Traction's application if changing realtime operating system from Rubus OS to an OSEK OS. In chapter 6 is the operating system of a test application changed from Rubus OS to an OSEK OS.

2 OSEK/VDX

OSEK [1] is the abbreviation for the German term "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" (Eng: "Open Systems and the Corresponding Interfaces for Automotive Electronics"). It was founded in 1993 as a joint project of the German automotive industry aiming for an industry standard for distributed control units in vehicles. In 1994 some French automotive companies joined OSEK and introduced their VDX-approach (Vehicle Distributed eXecutive) which is a similar project started by French automotive companies. As a result OSEK/VDX was introduced.

The initial OSEK/VDX partners are Adam Opel AG, BMW AG, DaimlerChrysler AG, IIIT-University of Karlsruhe, GIE.RE. PSA, Renault, Robert Bosch GmbH, Siemens AG and Volkswagen AG. Presently there are 65 companies in the OSEK/VDX technical committee. Among these companies are both vehicle producers like FIAT, Ford, GM Europe and Porsche and producers of realtime operating systems and tools, like Accelerated Technology, ETAS, Greenhills and Metrowerks.

2.1 Goal with OSEK/VDX

The motivation for introducing the OSEK/VDX project is high recurring expenses in the development of non-application related aspects of control unit software and the incompatibility of control units made by different distributors caused by the use of different interfaces and protocols.

The goal is to increase portability and reusability of the application software. This is achieved by specifying interfaces that are abstract and as application independent as possible in the areas of real-time operating system, communication and network management. The user interfaces should also be independent of hardware and network.

2.2 OSEK/VDX Operating System

OSEK/VDX Operating System [2] describes a standard for real time operating systems, which support efficient utilisation of resources for automotive control unit application software. The operating system described is a single processor operating system meant for distributed embedded control units.

The interface between the operating system and the application software is standardised using a set of system services. The system services are identical for all implementations of the OSEK/VDX operating system.

ISO/ANSI-C syntax was used when defining the system services.

2.2.1 Architecture of the OSEK/VDX operating system

There are a few different processing levels defined for an OSEK operating system. In increasing priority order the processing levels are task level, level for operating system internal activities and interrupt level. The task level with the lowest priority is where the application software is executed. The second highest priority level is assigned to the operating systems internal activities. The interrupt level is assigned the highest priority and this is where the interrupt service routines are executed.

Task management

When developing complex control applications it is often a good idea to split up the application into smaller tasks according to their real-time requirements. The operating system controls which task should be running via the scheduler and it supports parallel and asynchronous execution of tasks.

Extended tasks

An extended task can be in one of the following four different states: running, ready, waiting and suspended.

Only one task can be in the running state at a given time. The CPU is assigned to the task in the running state.

All tasks that are ready to execute are in the ready state. In a context switch the scheduler decides which task to execute next.

When a task is waiting for an event to occur it is in the waiting state.

When a task is passive and don’t want to execute it is in the suspended state. A task in the suspended state can be activated and moved to the ready state.

Basic tasks

The state model for basic tasks is exactly the same as the state model for extended tasks except that it does not have a waiting state. The advantage of basic tasks compared to extended tasks is that they require less system resources (RAM). A short description of the different states follows:

Only one task can be in the running state at a given time. The CPU is assigned to the task in the running state.

All tasks that are ready to execute are in the ready state. In a context switch the scheduler decides which task to execute next.

When a task is passive and do not want to execute it is in the suspended state. A task in the suspended state can be activated and moved to the ready state.

Conformance classes

To overcome the problem that different application software have various requirement on the system and to be able to use the OSEK operating system on a wide range of hardware the conformance classes were introduced. There are four different conformance classes and they are upward compatible.

The conformance classes defined are:

- BCC1 only supports basic tasks and are limited to one request per task (meaning that the operating system cannot handle activations of a basic task already activated). It is also limited to one task per priority.
- BCC2 is like BCC1 but supports multiple requesting of task activation, which means that if a task that is already activated receives one or more activation requests these request are queued. The activation requests are queued per priority in activation order. BCC2 does also support more than one task per priority.

- ECC1 is like BCC1 but with support for extended tasks.
- ECC2 is like BCC2 but with support for extended tasks and multiple requesting of task activation are not allowed.

2.2.2 Scheduling

All tasks are assigned a priority. Zero is the lowest priority and larger numbers define higher priorities. Priorities are defined statically and cannot be changed at runtime. At a context switch the task with the highest priority gets to execute. Tasks of identical priority are activated in FIFO (First In First Out) order. OSEK/VDX supports the following scheduling policies: non preemptive scheduling, full preemptive scheduling, groups of tasks, mixed preemptive scheduling.

Non preemptive scheduling

When using a non preemptive scheduling policy task switching is only allowed at given points in the code. The points of rescheduling are the following:

- When a task terminates successfully.
- When a task terminates successfully and explicitly activates a successor task.
- When an explicit call of the scheduler is made.
- When a transition into the waiting state takes place.

When using non preemptive scheduling it is possible that a high priority task has to wait for a task with lower priority to finish its execution before the high priority task can start.

Full preemptive scheduling

When using full preemptive scheduling the currently running task may be preempted at any instruction by a task with a higher priority. This means that the latency period is independent of the run time of lower priority tasks. When the system is fully preemptive one must always expect to be pre-empted by another task. It is possible to block the scheduler if a critical region must not be preempted.

Groups of tasks

By defining groups of tasks it is possible to allow tasks to combine the aspects of preemtive and non preemptive scheduling. Tasks within a group behave a bit different. For a task that has the same or lower priority as the task with the highest priority within the group, the tasks within the group acts like non preemptive tasks. For a task that has higher priority than the task with the highest priority within the group, the tasks within the group acts like preemptive tasks.

Mixed preemptive scheduling

The mixed preemptive scheduling was introduced to gain the benefits of both preemptive and non preemptive scheduling. In a system with a few parallel threads with long execution times preemptive scheduling is to prefer while in a system with many very short tasks a non preemptive scheduling policy is preferable.

The reason not to use preemptive scheduling is when a task switch consumes the same amount of time as the execution of the task or when one have limited amount of RAM or finally if a task must not be pre-empted.

When using the mixed preemptive scheduling the scheduling policy depends on the scheduling policy of the running task. If the running task is preemptable then preemptive scheduling is used and if the running task is non preemptable then non preemptable scheduling is used.

2.2.3 Application modes

Application modes are designed because many micro control units (MCUs) may run several completely independent applications, for example normal operation or factory test. The application mode has to be decided at start up. It is not allowed to change application mode at runtime.

Normally the different application modes have their own set of tasks, ISR's etc. but if the same functionality is needed again, in a different application mode, it is allowed and recommended to share between the modes.

At start up it is up to the user code, using no system services, to determine which application mode that should be started and pass it as a parameter to the API-service StartOS. It should be fast and easy to discover which system mode to start to avoid a long and complicated start up procedure.

2.2.4 Interrupt processing

OSEK/VDX describes two categories of interrupts, ISR category 1 and ISR category 2. The difference between category 1 and 2 is that it’s not possible to use operating system services within an interrupt service routine of category 1 and after the ISR has finished, the execution continues at the exact instruction where the interrupt has occurred. ISR's of category 2 are allowed to use some operating system services, for instance the Activate Task service routine and after termination of the ISR of category 2 a rescheduling will take place.

Inside the ISR no rescheduling takes place. The scheduling of the ISR's is hardware dependent and therefore it is not specified in OSEK/VDX. The number of interrupt priorities also depends on the hardware and on the implementation.

It is always possible to disable interrupts, both within an interrupt service routine and in an ordinary task.

2.2.5 Event mechanism

An event is always owned by an extended task. An extended task can be the owner of several events. The events are used for synchronization, message passing and so on.

All tasks, both basic and extended, can set any event but only the owner of the event can wait for and clear the event.

When an extended task is waiting for an event it is placed in the waiting state. When the event occurs it is moved to the ready state and the operating system reschedules. If a task is waiting for several events it is moved to the ready state when the first event occurs. If an extended task tries to wait for an event that has already occurred, the task remains in the running state.

2.2.6 Resource management

Resource management is used to control concurrent access to a resource when several tasks with different priorities try to gain access to it. The resource management is useful under the following circumstances:

- Using preemptable tasks.
- If tasks and interrupt service routines share resources.
- If interrupt service routines share resources.

The resources could for instance be memory, hardware areas or the scheduler. The resource management handles the following problems:

- Two tasks cannot occupy the same resource at the same time (mutual exclusion).
- It prevents priority inversion, see section 2.2.6.1.1.
- It prevents dead locks, see section 2.2.6.1.2.
- An attempt to access a resource never results in a waiting state because of the OSEK Priority Ceiling Protocol, described in chapter 2.2.6.2 OSEK Priority Ceiling Protocol.
- Two tasks or interrupt service routines cannot occupy the same resource at the same time (this is true if the resource management is extended to include interrupts.)

Problems with synchronisation mechanisms

Priority inversion

Priority inversion is a typical problem with synchronization mechanisms. It means that a lower-priority task delays the execution of a higher-priority task despite that they do not share any common resources.

The solution to this problem in OSEK is the OSEK Priority Ceiling Protocol.

Figure 2-1 shows the problem of priority inversion. Task T1 has the highest priority and task T3 the lowest. Task T3 occupies semaphore S1 and is then pre-empted by T1. Task T1 tries to access the semaphore S1 but is denied because it is already occupied by T3. Because of S1 is occupied T1 enters the waiting state. Now T2 is put in the running state. After T2 is finished T3 gets to run again and releases the semaphore S1. Now after the low priority threads are finished the high priority thread T1 is put in the running state again. The low priority threads delayed T1. Also the thread T2 that did not use the semaphore.

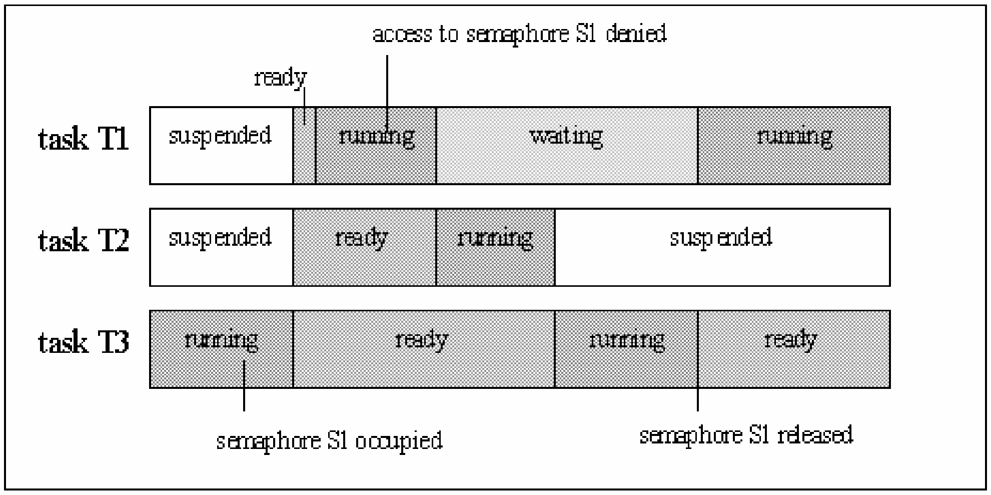

Figure 2-1 Priority inversion

Deadlocks
Deadlock is another problem of synchronisation mechanisms. Deadlock means that task execution is impossible because of infinite waiting for mutually locked resources.

Figure 2-2 shows the problem of deadlocks. Task T1 has the highest priority. When the task T1 is running it accesses the semaphore S1 and then stops, waiting for an event to occur. The task T2 is transferred to the running state and occupies the semaphore S2. An event happened and T1 is put in the running state. T1 tries to access the semaphore S2 but is denied because it’s already occupied by the task T2. Now T2 is back in the running state and tries to access the semaphore S1 but is denied because the semaphore is already occupied by the task T1. This is a deadlock.

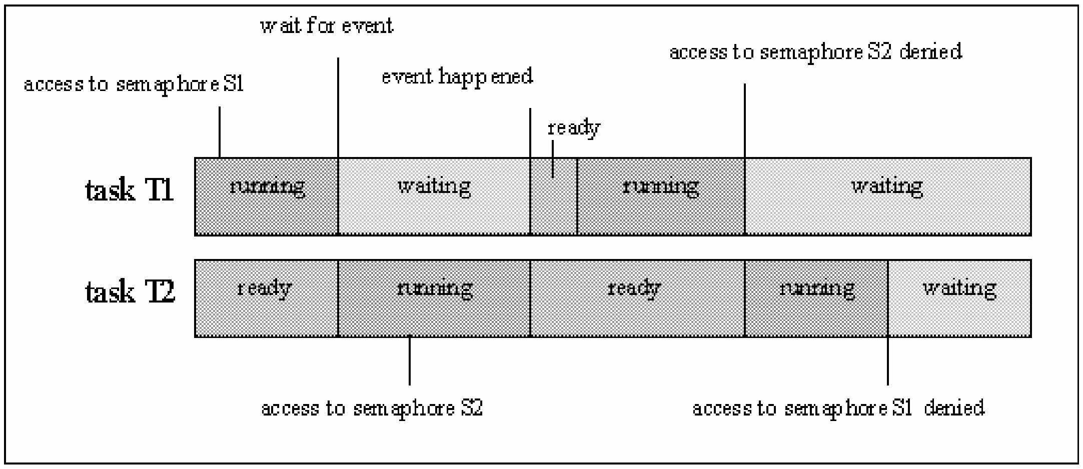

Figure 2-2 Deadlock

OSEK Priority Ceiling Protocol
The OSEK priority ceiling protocol exists to avoid priority inversion and deadlocks. It works as follows:
All resources are assigned a ceiling priority. The ceiling priority is equal to or higher than the priorities of all tasks that have access to the resource or any of the resources linked to this resource. The ceiling priority must be lower than the lowest priority of all the tasks that do not have access to the resource and which have priorities higher than the task with the highest priority of all the tasks that access the resource.

When a task tries to access a resource its priority is raised to the ceiling priority of that resource. When a task releases a resource the priority of that task is lowered to its normal priority.

The OSEK priority ceiling protocol with extensions for interrupt levels works exactly as described above but the ceiling priorities are set with regard to both the tasks and the interrupt service routines that have access to the resources.

2.2.7 Alarms
OSEK/VDX supports the handling of recurring events by alarms and counters. An alarm can either be activated at regular time intervals or for example by encoders at certain positions of an axis. An alarm can be either a single alarm or a cyclic alarm. Counters and alarms are defined statically.

An alarm must be connected to one counter and one task or one alarm-callback routine. The alarm will be activated when the counter it is related to reaches a predefined value. When the alarm is activated it can do one of the following three things depending on the configuration:

- Activate the alarm-callback routine (a short function that runs with category 2 interrupts disabled).
- Activate the task.
- Set an event for the task.

2.2.8 Error handling, tracing and debugging

To simplify error handling, tracing and debugging OSEK/VDX provides different hook routines. The hook routines allow user defined actions within the OS internal processing.

Features of hook routines:

- Hook routines are called by the operating system.
- They have higher priorities than all tasks.
- They cannot be interrupted by category 2 interrupt service routines.
- The hook routines are part of the operating system.
- They are implemented by the user and have user-defined functionality.
- Their use of API functions is restricted.

The following five hook routines exist in an OSEK/VDX OS:

- ErrorHook
- PreTaskHook
- PostTaskHook
- StartupHook
- ShutdownHook

The ErrorHook is activated if a system service returns a StatusType value that is not equal to E_OK. In the Error Hook the user is able to access some additional information to support a more effective error management.

The PreTaskHook and the PostTaskHook may be used for debugging or time measurement purposes. The PreTaskHook is called every time directly after a new task enters the running state. The PostTaskHook is called every time directly before the old task leaves the running state.

The StartupHook is called every time at system start up. The purpose of the StartupHook is to perform user defined start up functions. After a reset the user is first able to execute hardware specific code and then make the StartOS call. After this call the operating system runs its initialisation code and then calls the StartupHook. In the StartupHook the user has the possibility to execute user-defined initialisation code. When the StartupHook is finished the operating system enables user interrupts and starts the scheduler.

When the operating system call ShutdownOS is called by the application or because of a fatal error the ShutdownHook is started. The user is able to define any system behaviour in the ShutdownHook. It is even allowed not to return from the routine.

2.3 OSEK/VDX Time-Triggered

OSEK/VDX Time-Triggered (OSEKtime) [6] is another description of an OSEK/VDX realtime operating system with the main difference that it uses static scheduling. All important services, i.e. interrupt handling, dispatching, system time and clock synchronization, local message handling and error detection mechanisms are supported. If both static and dynamic scheduling is desired for the application it is possible to combine OSEK/VDX Time-Triggered with OSEK/VDX OS. Then OSEK/VDX OS gets the CPU when OSEK/VDX Time-Triggered is in the idle mode.

2.3.1 Task management

OSEK/VDX Time-Triggered uses time-triggered tasks. A time-triggered task has three different task states:

- **Running:** Only one task in the entire system can be in the running state at any point in time. The task in the running state is the task that is using the CPU at that point. Only the OSEKtime Dispatcher can set a task in the running state.
- **Preempted:** A task is placed in the pre-empted state if another time-triggered task is activated before the first task has finished its execution. It leaves the pre-empted state and is placed in the running state again when the pre-empting tasks changes from the running state to the suspended state.
- **Suspended:** All tasks in the suspended state are passive and can be activated by the Dispatcher.

The tasks are scheduled statically and the information on the task activation times is stored in the dispatcher table. The OSEKtime Dispatcher is responsible for starting the tasks at the right time. There is also functionality for monitoring the deadlines. This is also handled by the dispatcher but with the help of the Deadline Monitoring dispatcher table. If a task violates its deadline an error hook is started.

In OSEK/VDX Time-triggered there is a special task named ttIdleTask. This task is always the first task that is started by the OSEKtime dispatcher. That means that this task always gets the CPU if there are no other tasks activated. If one combines OSEK/VDX Time-triggered with OSEK/VDX OS, OSEK/VDX gets the CPU instead of the ttIdleTask when no tasks or interrupt service routines in OSEKtime wants to execute.

2.3.2 Interrupt processing

Interrupt service routines are also supported by OSEK/VDX Time-triggered but with one restriction. When configuring the operating system for the application one has to provide the operating system with information on how often a particular interrupt service routine is allowed to execute. One has to define an interval in time where each interrupt may occur at most once. This makes it possible for the static scheduler to calculate worst case execution times even when interrupts are allowed.

2.3.3 Synchronisation

OSEK/VDX Time-triggered has a synchronisation mechanism available to make it possible for several different electrical control units to synchronise their local time with a global time base. The synchronisation is done at start up and is adjusted to the global time every time after a dispatcher round.

2.3.4 Inter-Task Communication

OSEKtime also supports external message handling. The functionality that has to be supported by every OSEKtime implementation is described in the OSEKtime FTCom specification [21].

2.3.5 Error handling

OSEKtime also supports error handling. The error handling in OSEKtime is the same as in OSEK/VDX OS.

2.4 OSEK/VDX Communication

OSEK/VDX Communication [3] describes a standard for the communication between tasks and interrupt service routines within and between ECUs. There are different conformance classes with different functionalities. The conformance classes CCCA and CCCB only support internal communication, i.e. between tasks and ISR's within an ECU, while the conformance classes CCC0 and CCC1 also support external communication, i.e. between ECUs. Since Haldex Traction's application only needs internal communication this will be the focus.

2.4.1 Requirements

The OSEK COM specification fulfils the following requirements:

- General communication functionality.
- Portability, reusability and interoperability of application software.
- Scalability.
- Support for Network Management (NM).

2.4.2 Communication concept

OSEK COM provides an API with services for the transfer of messages using send and receive operations. OSEK COM specifies the Interaction Layer (IL).

The internal communication is handled entirely by the interaction layer but the external communication has to use the Network Layer and the Data Link Layer. OSEK COM does not specify the Network Layer or the Data Link Layer but it defines the minimum requirements to support all features of the Interaction Layer.

2.4.3 Interaction Layer

Overview

The communication in OSEK COM is based on messages. Messages are sent in the form of message objects and the content of the messages is user defined. All messages and message properties have to be defined statically via the OSEK Implementation Language (OIL).

The IL describes an API to handle messages. It has functionalities for initialisation, data transfer and communication management.

Message identifiers are used to identify message objects. Messages are sent by sending message objects and received by receiving message objects. The OSEK COM supports m-to-n communication. This means that zero or more senders are able to send messages to the same sending message object and sending message objects are able to store messages in zero or more receiving message objects. One receiving message object receives messages from exactly one sending message object.

There are two types of receiving message object. They can be either queued or unqueued. The queued receiving message object has a FIFO (first in first out) queue to store incoming message. The size of the queue must be set to a value different from zero. If a queue is full all the messages it receives are lost. A message can only be read once in a receiving queue since the read operation removes the oldest message from the queue. If the queue is empty the IL does not provide any message data to the application. In the case of m-to-n communication each receiver has its own message queue and the messages from these queues are consumed independently.

The unqueued receiving message object only has place for one message object. A message can be read more than once. The read operation always returns the latest message. If no message has been received since the start of the IL the application receives a message value set at initialisation.

Message reception

Message data are copied from the message object to the application message when the API services ReceiveMessage or ReceiveDynamicMessage are called.

It is possible to connect a reception filter to a message object. The message filter uses a predefined filtering algorithm to check if the message fulfils certain conditions. If it does not, the message is discarded. Different filtering algorithms can be defined for each message.

Message transmission

When sending an internal message the IL routes the message directly to the receiving part of the IL. A message can be stored in zero or more message objects when it’s transferred. The message is sent when the SendMessage or SendZeroMessage API service is called. When a zero-length message is sent no data transfer takes place.

Notification

OSEK COM describes a notification mechanism to be able to determine the final status of a previously called send or receive operation. The notification is only performed if the message is properly received. This means that there is no notification if the filtering algorithm discards the message or if it is lost because of a full message queue.

There are four different notification classes but only the first class is supported for internal communication. Notification Class 1 activates the configured notification mechanism directly after the message has been stored in the receiving message object.

The following four notification mechanisms exist:

- **Callback routine:** The IL calls a user defined callback routine.
- **Flag:** A flag is set that can be check by the application.
- **Task:** The IL activates a user-defined task.
- **Event:** The IL sets an event for a user-defined task.

A given sender or receiver is limited to use only one of the notification mechanisms.

Error handling

OSEK COM also provides an error service. There are two different kinds of errors defined:

- **Application error:** The requested API service was not executed correctly but the IL's internal data are correct. The centralised error treatment is called and after that the decentralised error treatment is called with error status information.
- **Fatal errors:** A fatal error means that the IL cannot assume correctness of its internal data. This leads to a call to the centralised system shut down.

Two levels of error checking are provided by the OSEK COM:

- **Extended error checking:** The extended error checking is used under the development of an application. It supports more accurate error checking but requires more execution time and more memory consumption.
- **Standard error checking:** The standard error checking is less demanding and is used in a fully debugged system.

2.4.4 Conformance classes

To support different application requirements and specific system capabilities the communication specification supports four different Communication Conformance Classes (CCCs). The following communication conformance classes are defined:

- **CCCA:** Defines the minimum requirements for supporting the OSEK/VDX Communication document. It only supports internal communication for unqueued messages. No message status information is supported.
- **CCCB:** This CCC does also only support internal communication and all features of CCCA plus queued messages and message status information.

- **CCC0:** Defines the minimum requirements for supporting both internal and external communication. All features of CCCA are supported plus some minimum features to support external communication.
- **CCC1:** Supports all features of OSEK/VDX COM.

2.5 OSEK/VDX Network Management

The purpose of the network management [4] is to make a standard for network management in a network of ECUs that all uses OSEK/VDX compliant realtime operating systems. It becomes more and more common in the vehicle industry with networks of ECUs that communicate with each other, where all the nodes are made by different manufacturers. To ensure safety and functionality in such networks network management becomes important.

Network management in OSEK/VDX means the possibility for all nodes in the network to know the status of all other nodes and thereby detect failures or missing nodes in the network. The network shall also support network related diagnostic features.

The uses of these functions are up to the system responsible but to make this possible every node in the network have to support OSEK/VDX Network Management.

2.6 OSEK/VDX Implementation Language

OSEK/VDX implementation language (OIL) [5] describes a standard for the configuration of an application using OSEK/VDX. The goal is to make the software more portable between different OSEK/VDX realtime operating systems. The OIL-file may be hand-written or generated by an OIL configuration tool. Most OSEK/VDX realtime operating system suppliers have OIL configuration tools that read and generate OIL-files according to a user defined specification of the system.

2.6.1 General concept

An OSEK/VDX application is described with the help of a set of OIL objects. The CPU is a container for all these OIL objects. An OIL object consists of a predefined set of attributes and references. OIL defines all standard attributes for each OIL object.

A specific OSEK/VDX implementation may have additional attributes in the OIL objects but it is not allowed to change a standard attribute or to add a new OIL object.

The following OIL objects shall be used to describe an OSEK/VDX application with OIL:

- **CPU:** A container for all other objects.
- **OS:** The OS object is used to define OSEK OS properties for the application.
- **APPMODE:** Used to define different application modes.
- **ISR:** This is where the interrupt service routines of the application are defined.

- **RESOURCE:** A resource is a code segment that can be occupied by a task.
- **TASK:** A task that is scheduled by the realtime operating system.
- **COUNTER:** A counter is used as hardware/software tick source for alarms.
- **EVENT:** An event that can be sent to a task.
- **ALARM:** An alarm is based on a counter. When it activates it can activate a task, set an event or activate an alarm-callback routine.
- **COM:** This is where the standard attributes for the communication subsystem are set.
- **MESSAGE:** The message attribute belongs to OSEK COM and defines the supplier-specific attributes to configure data exchange through messages between different tasks, interrupt service routines and CPUs.
- **NETWORKMESSAGE:** The message attribute belongs to OSEK COM and defines the OEM-specific attributes to configure data exchange through messages between different tasks, interrupt service routines and CPUs.
- **IPDU:** An IPDU is defined in OSEK COM. IPDU is used when transporting messages between different CPUs.
- **NM:** This is where the standard attributes for the network management subsystem are set.

3 Some OSEK/VDX RTOS

3.1 Why OSEK/VDX

The technical benefits of OSEK/VDX are described under the heading Section 2.1. In this chapter the market related benefits of OSEK/VDX presented.

Below a table of the leading global vehicle producers of 2002 is presented [20]. Many of the largest vehicle producers are either in the list of initial partners of OSEK/VDX (steering committee) or in the technical committee. Haldex Traction has three large customers, the VW-group, Ford and General Motors. They are all members of either the OSEK/VDX steering committee or the OSEK/VDX technical committee.

Nr	Vehicle producers	Units 2002	OSEK/VDX partners	Haldex Traction's customers
1	General Motors	8 504 434	Technical committee	Customer
2	Ford Motor Co.	6 819 594	Technical committee	Customer
3	Toyota Motor Corp.	6 167 703	-	-
4	Volkswagen AG	4 989 030	Steering committee	Customer
5	Daimler Chrysler AG	4 540 900	Steering committee	-
6	PSA/Peugeot-Citroen SA	3 267 474	Steering committee	-
7	Hyundai Motor Co.	2 939 499	-	-
8	Honda Motor Co.	2 820 000	-	-
9	Nissan Motor Co.	2 735 530	-	-
10	Renault SA	2 403 975	Steering committee	-
11	Fiat S.p.A	2 079 336	Technical committee	-
12	Mitsubishi Motors Corp.	1 847 800	-	-
13	Suzuki Motor Corp.	1 707 392	-	-
14	BMW Group	1 057 344	Steering committee	-
15	Mazda Motor Corp.	964 800	-	-

Table 3-1 Leading global vehicle producers

3.2 Important factors of OSEK OS for Haldex Traction

Haldex Traction's currently used realtime operating system is Rubus OS from the Swedish company Arcticus. Since the AWD control application works very well with Rubus OS the main reason to switch operating system is to follow the OSEK/VDX standard. Here follows a list of Haldex Traction's most important requirements for the OSEK/VDX realtime operating systems and their suppliers:

- They have to support the hardware ST10F272 from ST Micro Electronics, which Haldex Traction will be using.
- The realtime operating system has to be certified for the OSEK/VDX OS standard.
- The realtime operating system should be well tested and preferably used by some well-known companies.
- A low price is of great importance.
- The supplier shall be a large and well-established company.

- It is good if they have support in Sweden.
- It is good if they support the compiler named Tasking.

3.3 Eight OSEK/VDX RTOS

An extensive market research based on information found on the web and a couple of reports [17][18][19] resulted in the following eight realtime operating systems for further investigations:

- Eurosmot from Euros
- Ercosek for Etas (Etas Group)
- OSEKturbo from Metrowerks
- RTA-OSEK from LiveDevices (Etas Group)
- ProOSEK from 3Soft
- Nucleus OSEK from Accelerated Technology
- OX-OSEK from Trialog
- osCAN from Vector Informatic

3.3.1 Eurosmot

The company Euros develops Eurosmot [7][20]. Euros has been in the OSEK/VDX business for a while. They do not support ST10F272 today but the will do in the future.

OSEK/VDX certified	Version 2.2.1
Conformance classes	BCC1, BCC2, ECC1, ECC2
OIL support	Yes
Minimum resource usage	See appendix A (confidential)
Compiler	Tasking
Number of offices in the company	1
Number of employees	?
When the company was founded	?
Included in the price	OS, OIL configurator
Price for 10 developers	See appendix A (confidential)
Price for the operating system	See appendix A (confidential)
Support in Sweden	No
Introduced on the market	1997 under the name OSEKplus
Major companies that use the RTOS	ContiTemic, Daimler Chrysler

3.3.2 Ercosek

This realtime operating system Ercosek [8][20] was developed by Etas but when LiveDevices joined the ETAS Group they decided to concentrate on LiveDevices OSEK/VDX OS implementation RTA-OSEK.

OSEK/VDX certified	The system was replaced by RTA-OSEK
Conformance classes	-
OIL support	-
Minimum resource usage	-
Compiler	-
Number of offices in the company	-
Number of employees	-
When the company was founded	-
Included in the price	-
Price for 10 developers	-
Price for the operating system	-
Support in Sweden	-
Introduced on the market	-
Major companies that use the RTOS	-

3.3.3 OSEKturbo

OSEKturbo [9][20] is developed by Metrowerks. OSEKturbo does not support the conformance classes BCC2 and ECC2 that Haldex Traction needs, unless they do not want to change the current implementation.

OSEK/VDX certified	Version 2.2
Conformance classes	BCC1, ECC1
OIL support	Yes
Minimum resource usage	See appendix A (confidential)
Compiler	Tasking 7.0
Number of offices in the company	15
Number of employees	550
The company was founded in	1985
Included in the price	OS, configuration tool
Price for 10 developers	See appendix A (confidential)
Price for the operating system	See appendix A (confidential)
Support in Sweden	No, onsite training € 3 000 for two days + travel costs.
Introduced on the market	1995
Major companies that use the RTOS	Siemens VDO, Daimler Chrysler

3.3.4 RTA-OSEK

RTA-OSEK [10][20] is developed by LiveDevices that is part of the Etas Group. The main features with this real time operating system are the development tools, Planner and Builder.

OSEK/VDX certified	Version 2.2
Conformance classes	BCC1, BCC2, ECC1, ECC2
OIL support	Yes
Minimum resource usage	See appendix A (confidential)
Compiler	Tasking 7.5

Number of offices in the company	1
Number of employees	35
When the company was founded	1994
Included in the price	OS, Planner, Builder
Price for 10 developers	See appendix A (confidential)
Price for the operating system	See appendix A (confidential)
Support in Sweden	No, they are willing to go here from UK if needed.
Introduced on the market	1999
Major companies that use the RTOS	Tier1, Volvo OEMs

3.3.5 ProOSEK

ProOSEK [11][20] is developed by the German company 3Soft. For Swedish customers it is sold and supported by the Swedish company ENEA with an office in Malmö.

OSEK/VDX certified	Version 2.2
Conformance classes	BCC1, BCC2, ECC1, ECC2
OIL support	Yes
Minimum resource usage	See appendix A (confidential)
Compiler	Tasking 8.0
Number of offices in the company	2
Number of employees	201
When the company was founded	1988
Included in the price	OS, OIL configurator
Price for 10 developers	See appendix A (confidential)
Price for the operating system	See appendix A (confidential)
Support in Sweden	Yes, ENEA has sale and support in Sweden
Introduced on the market	?
Major companies that use the RTOS	BMW, Audi

3.3.6 Nucleus OSEK

Nucleus OSEK [12][20] is developed by the company Accelerated Technology.

OSEK/VDX certified	Version 2.2
Conformance classes	BCC1, BCC2, ECC1, ECC2
OIL support	Yes
Minimum resource usage	See appendix A (confidential)
Compiler	Tasking 7.5
Number of offices in the company	7
Number of employees	3700
When the company was founded	1990
Included in the price	OS, OIL configurator
Price for 10 developers	See appendix A (confidential)
Price for the operating system	See appendix A (confidential)
Support in Sweden	One person in Sweden, offices in Finland and

	UK.
Introduced on the market	2001
Major companies that use the RTOS	?

3.3.7 OX-OSEK

OX-OSEK [13][20] is developed by TRIALOG, but it is not certified for OSEK/VDX. It supports OSEK/VDX conformance classes BCC1 and ECC1 but since OSEK/VDX is a protected trademark of Siemens AG, TRIALOG is not allowed to claim that OX-OSEK supports OSEK/VDX because the OSEK/VDX committee has not certified it.

OSEK/VDX certified	No, support OSEK/VDX
Conformance classes	BCC1, ECC1
OIL support	Yes
Minimum resource usage	See appendix A (confidential)
Compiler	Tasking
Number of offices in the company	1
Number of employees	30
When the company was founded	1987
Included in the price	OS
Price for 10 developers	See appendix A (confidential)
Price for the operating system	See appendix A (confidential)
Support in Sweden	No
Introduced on the market	1998
Major companies that use the RTOS	Valeo

3.3.8 osCAN

Vector Informatic that is a part of Vector develops osCAN [14][20]. The product is interesting since it has a good price, support in Sweden, is developed by a large company with 15 years in the market and it supports all conformance classes and implementations that Haldex Traction needs.

OSEK/VDX certified	Version 2.2
Conformance classes	BCC1, BCC2, ECC1, ECC2
OIL support	Yes
Minimum resource usage	See appendix A (confidential)
Compiler	Tasking
Number of offices in the company	6
Number of employees	500
When the company was founded	1988
Included in the price	OS, OIL configurator
Price for 10 developers	See appendix A (confidential)
Price for the operating system	See appendix A (confidential)
Support in Sweden	Yes
Introduced on the market	1998
Major companies that use the RTOS	GM, Ford, VW, Saab, Volvo Car

3.4 Evaluation of the OSEK/VDX candidates

The conclusion after the evaluation of the different realtime operating systems is that Vector Informatics' OSEK/VDX implementation osCAN is the operating system best suited for Haldex Traction.

The first RTOS that was excluded from the evaluation was Etas' Ercosek. The reason was that when LiveDevices joined the ETAS Group, ETAS decided to concentrate on LiveDevices OSEK/VDX OS implementation RTA-OSEK, so Ercosek was replaced by RTA-OSEK.

The most important factor in the evaluation is that the realtime operating system supports Haldex Traction's new hardware, ST10F272 from ST Micro Electronics. This demand did not eliminate any of the alternatives since all the examined operating systems support the hardware ST10F272.

The second most important factor is that the implementation of the operating system is certified for some version of OSEK/VDX. OX-OSEK is not certified and was therefore excluded from the evaluation.

To be able to send messages between tasks in Haldex Traction's application an implementation of OSEK/VDX that also supports the communication specification, OSEK COM, is needed. Among the remaining seven RTOS everyone except two do support OSEK COM. The implementations that are missing the OSEK COM are Euros' Eurosmot and Metrowerks' OSEKturbo. Left after the exclusion are RTA-OSEK, Nucleus OSEK, ProOSEK and osCAN.

RTA-OSEK developed by LiveDevices and Nucleus OSEK developed by Accelerated Technology are the newest realtime operating systems in the evaluation. RTA-OSEK has been on the market approximately five years under various names and it is also an expensive product. There are two main reasons why Nucleus OSEK is not suited for Haldex Traction. The major disadvantage is that Accelerated Technology cannot guarantee that their realtime operating system works together with Vector's CAN-devices, which is a demand from Haldex Traction. The other disadvantage is the fact that Nucleus OSEK is the newest system in the evaluation.

Both of the two final systems seem to be good choices with regard to Haldex Traction's demands. One advantage with 3Softs ProOSEK is that it supports OSEK/VDX Network Management but this was not considered necessary.

The system that was chosen was Vector's osCAN. It supports the necessary hardware, is certified for OSEK OS, OIL and COM, it has been on the market since 1998 and is well tested. For Haldex Traction's volume of production and number of developers osCAN is also the least expensive system of all realtime operating systems in the evaluation. Vector Informatics is a large company with about 500 employees and one support office in Gothenburg, Sweden. osCAN is also the operating system with shortest response time among the four systems RTA-OSEK, Nucleus OSEK, ProOSEK and osCAN.

Table 3-2 is a simplified illustration of how the choice of an OSEK OS was made.

OSEK OS	Hardware support	OSEK/VDX certified	OSEK COM	Long time on the market	Best price
osCAN	X	X	X	X	X
ProOSEK	X	X	X	X	-
Nucleus OSEK	X	X	X	-	-
RTA-OSEK	X	X	X	-	-
OSEKturbo	X	X	-	-	-
Eurosmot	X	X	-	-	-
OX-OSEK	X	-	-	-	-
Ercosek	-	-	-	-	-

Table 3-2 Illustration of the elimination of OSEK OS. A (X) indicates that the OSEK OS fulfils the criteria. A (-) indicates that the OSEK OS is eliminated.

4 Rubus OS, Arcticus Systems

Rubus OS [20][23] does not support OSEK/VDX but it is the realtime operating system that Haldex Traction is currently using. The Swedish company Arcticus Systems develops it.

OSEK/VDX certified	No
Conformance classes	-
OIL support	-
Minimum resource usage	See appendix A (confidential)
Compiler	Tasking 7.5
Number of offices in the company	1
When the company was founded	1985
Included in the price	OS
Price for 10 developers	See appendix A (confidential)
Price for the operating system	See appendix A (confidential)
Support in Sweden	Yes, office in Stockholm.
Introduced on the market	?
Big companies that uses the RTOS	Haldex Traction, Volvo

4.1 Rubus OS architecture

Rubus OS is a realtime operating system designed for safety critical applications. The system is scalable why it could be used even in small micro controllers. The scheduling supports both statically and dynamically scheduled threads.

4.1.1 The Basic Services

The basic services in Rubus contains the services common for both Red and Blue threads. The services are basic clocks and timers, event log services and basic queue services.

Clocks and Timers

There are two types of clocks in Rubus OS, the basic clock and the blue clock. The basic clock is the system clock representing the real-time clock for the system. The resolution of the basic clock is the smallest resolution possible for the Red and Blue timers. The blue clock is based on the basic clock. At regular intervals the value of the basic clock is copied to the blue clock. The blue clock has a resolution that is equal to or less than the resolution of the basic clock. The benefits of having two clocks are simplified time handling in the Blue Kernel.

Rubus OS supports two types of timers, one shot and periodic. The initial time expiration can be defined either relative or absolute for both types of timers. The one shot timer is activated once while the periodic timer is activated time-periodically.

Event Log Services
With this service it is possible to log Red and Blue run-time events with a time stamp added to each event. It is up to the user to specify which events to log. The log is a FIFO queue and can be used to analyse the execution behaviour with the Rubus Execution Analyser tool.

Basic Queue Services
This service makes it possible to store information in a FIFO queue. The queue may be used for communication between all different threads in the system.

4.1.2 The Red Kernel Services
The Red Kernel manages the execution of Red Threads. A Red Thread in Rubus OS is an object with the following characteristics. A Red Thread consists of program code. A Red Thread is part of a Red Schedule, which is statically allocated. A Red Thread executes according to a Red Schedule. The execution of a Red Thread cannot be blocked.

A Red Schedule consists of a number of Red Threads. A Red Schedule can represent an application or a Red Execution Mode in an application. The application modes can be switched at runtime.

Red Schedule
A Red Schedule consists of sets of Red Threads. Each set of Red Threads has its own release time. The threads in a set are executed in sequence. The running behaviour of a Red Thread is defined in the Red Schedule. The set of Threads that a Red Thread belongs to decides the release time of the thread, deadline for the thread is also defined in the Red Schedule and the memory area for the Red Schedule. The Red Schedule is executed periodically with the period specified by the user.

Red Error Handling
All Red OS calls returns the error status of the operation. If the Red Kernel detects an error the user defined function redError is called.

4.1.3 The Blue Kernel Basic Layer Services
The Blue Threads executes in the left over time when no Red Threads wants to execute. The Blue Kernel is responsible for run-time services for management of the Blue Threads, the execution of the Blue Threads, services for co-operation between threads and memory management.

A Blue thread consists of program code and it can be created and terminated. A Blue Thread executes according to the scheduling policy and the priority of the thread. A Blue Thread object is statically allocated. When executing a Blue Thread can be blocked waiting for a signal.

Blue scheduling
A Blue Thread is always in one of the following states:

- **Dormant:** The thread is not alive, not created or terminated.
- **Ready:** The thread is ready to execute.
- **Blocked:** The thread is blocked, waiting for a signal.
- **Running:** The thread is executing.

The Blue Scheduling is performed at runtime and the scheduling policy used is FIFO scheduling. Every priority have one FIFO queue. When a Blue Thread becomes ready it is placed in the last place in its priority queue. The first thread in the queue with the highest priority gets to execute when no other thread is in the running state.

Blue Kernel Threads

The Blue Kernel has two internal Blue Threads for its operation, the Blue Kernel Thread and the Blue Idle Thread.

The Blue Kernel Thread handles time supervision. It has the highest priority among the blue threads and executes at every Blue timer tick.

The other Blue Kernel Thread is the Blue Idle Thread. The Blue Idle Thread executes when no other threads want to execute and has the lowest priority.

Blue Error Handling

All the Blue OS calls returns the error status of the operation. If the Blue Kernel detects an error the user defined function redError is called.

Signals

Rubus OS also has support for signals. With the signal mechanism it is possible to pause a Blue Threads execution and wait for a selected set of signals. The first signal in the set that arrives activates the thread. The signals can also be used for synchronization.

Interrupt Control

The Blue Kernel Basic Layer also has services for interrupt control. These services make it possible for a Blue Thread to capture an interrupt and to protect critical sections of code.

4.1.4 The Blue Kernel Thread Co-operation Layer Services

Rubus OS is a scalable RTOS and the Blue Kernel Thread Co-operation Layer is optional. The layer contains functionality for the management of synchronization of resources and communication between Blue Threads.

Mutex

Mutexes are a synchronization mechanism between threads. A mutex in Rubus OS has the following characteristics:

- A mutex is a type of binary semaphore.
- The owner of a mutex is the Blue thread performing the lock operation.

- A mutex can be locked and unlocked but it is only the owner that can perform unlocking.
- Mutexes in Rubus OS uses the priority inheritance protocol [22] to avoid priority inversion.
- A mutex is statically allocated.

Message Passing

Message passing allows threads to communicate via message FIFO queues. A message queue handles messages of a fixed size. It has services for copying messages to and from the queue. A message queue has no owner that means different threads can open, close and copy messages to and from the queue.

4.1.5 The Blue Personality Services

The Blue Personality Services contains services for input output management. Services for the management of I/O devices, file operations and simple and standardised interface to I/O devices.

4.2 Some differences between OSEK/VDX OS and Rubus OS

4.2.1 Scheduling

OSEK/VDX

OSEK/VDX uses dynamic scheduling of tasks. All tasks are assigned a fixed priority at system generation where zero is the lowest and every number higher than zero describes a higher priority.

Rubus

Rubus OS have two types of threads. They have red threads and blue threads.
The red threads are time periodic threads and use static scheduling.
The blue threads are scheduled dynamically according to their priority. All red threads have the same priority, a priority that is higher than the priority of all blue threads.

4.2.2 Periodic activities

OSEK/VDX

Time periodic activities are handled by the use of counters and alarms. A counter is set up. An alarm is connected to the counter and the activation interval has to be decided. Each time the alarm activates it can start a task, send an event to a task or start the alarm-callbackroutine.

Rubus

Time periodic threads are preferably defined as Red threads. All Red threads are time periodic and are scheduled statically. This makes the execution of critical time periodic threads fast and reliable.

4.2.3 Resource management

OSEK/VDX

OSEK/VDX uses the OSEK priority ceiling protocol to avoid priority inversion and deadlocks. All resources are configured off-line and all tasks that may want to use the resource have to be connected to the resource during configuration. All resources are statically assigned a ceiling priority at system generation.

When a task wants to acquire a resource it has to use the operating system call getResource and when it's done with the resource it has to make the system call releaseResource to release it.

Rubus

Rubus uses the priority inheritance protocol to avoid priority inversion. For most systems the priority inheritance protocol has better overall performance than the priority ceiling protocol but the worst case performance is usually better with the priority ceiling protocol.

Rubus uses binary semaphores to encapsulate critical code sections. It is up to the developer to avoid deadlocks.

4.2.4 Application modes

OSEK/VDX

OSEK/VDX supports the use of application modes. Different application modes may run completely independent applications. It is not allowed to change application mode at runtime.

Rubus

A number of Red threads are grouped into a Red schedule. Each Red schedule represents an application or an application mode. The application mode can be switched during runtime by an operating system call.

5 Conversion from Rubus OS to OSEK/VDX

Due to the differences between Rubus OS and the OSEK/VDX specifications there is some work to be done when changing the application from using Rubus OS to an operating system that is implemented according to the OSEK/VDX specification.

Since Rubus OS and OSEK/VDX use different scheduling policies the scheduling has to be considered if an application shall be adapted from using Rubus OS to OSEK/VDX. One way to do this is to translate the schedule in Rubus OS to the use of timers and alarms in OSEK/VDX using the rate monotonic scheduling policy [22].

Also the resource management has to be considered, as the resource management does not work the same way. The solution is straightforward. A segment in the code protected by semaphores will instead be set up as a resource in OSEK/VDX.

Rubus OS does have the advantage that it is possible to change application mode at runtime. There is no standard solution to this in OSEK/VDX.

All Rubus OS operating system calls that are used in Haldex Traction's application have to be translated and mapped to OSEK/VDX operating system calls.

5.1 Changes in the scheduling

Rubus OS supports both static and dynamic scheduling while OSEK/VDX only supports dynamic scheduling. In the slip coupling application all threads, except for two low priority threads, are Rubus Red threads, which means that they are time periodic and scheduled statically.

Since there only exists one semaphore in the entire application and the part of the code that uses the semaphore is only called from threads with the same priority it means that there will be no changes in the priorities even though the OSEK Priority Ceiling Protocol is used.

This means that Haldex Traction's present schedule in Rubus OS can be translated to OSEK/VDX using timers and alarms according to rate monotonic scheduling. With the help of alarms and timers in OSEK/VDX it is possible to obtain the same behaviour as with Rubus OS with the difference that the scheduling will take more time than before because of the change from dynamic to static scheduling. This extra time needed is very small compared to the execution time of the application and should not affect the behaviour.

In Rubus OS the Red (static) threads have the highest priority. In Haldex Traction's application there are three groups of Red threads. One group that starts every second ms and have to be finished within two ms. The other two groups starts every fifth ms and have to be finished within five ms.

The schedule will be translated according to the following design plan:

- The first group of Red threads that have the shortest period will be given the highest priority in the system and will be set to non-preemptive. All threads in this group will be converted to one thread to minimise the loss of time caused by task switching. A timer and a cyclic alarm will be connected to the thread to achieve the cyclic time period of two ms.
- There are two groups of Red threads with a period of 5 ms and they will be given the second highest priority in the system. These groups have to be preemptive because

they will execute in the leftover time between the executions of the group with two ms period time.

- Depending on the project there are three or more Blue (dynamic) Rubus threads in Haldex Traction's application. These threads have the same priority, which means that they have the lowest priority in the system since all Blue threads have lower priority than Red threads.

  The priority and scheduling of these threads will not change. All Blue threads will be given the same priority also in the new operating system. They will continue to be scheduled dynamically, preemptively and with the lowest priority in the system.

5.2 Changes in the resource management

When using binary semaphores to protect critical sections in the code the semaphore has to be locked when entering the critical section and unlocked when leaving the section. When the semaphore is locked, if another thread tries to lock the semaphore it has to wait until the thread that first locked the semaphore unlocks it. Because of this only one thread at a time can execute the critical section.

With an OSEK/VDX realtime operating system all critical code sections have to be set up before generating the system. At runtime when a thread wants to enter a critical section it has to make the operating system call getResource. If another thread also tries to enter the same critical section it will be blocked until the first thread makes the operating system call releaseResource.

The things that have to be done concerning resource management when changing the application from Rubus OS to OSEK/VDX are first of all to find all resources in the code that are protected by semaphores and define all the found resources as resources in the OIL-configurator. The next thing to do is to change all the calls that lock a semaphore with calls to getResource and all the calls that unlock a semaphore with calls to releaseResource, with the correct resource as a parameter.

5.3 Different application modes

In Rubus OS there is a possibility to have different application modes. This opportunity is used in Haldex Tractions application. They have for example one init mode at start up, one operating mode for normal execution and one safety mode if things go wrong.

A certain Red Schedule defines an application mode in Rubus OS. Different application modes have different Red Schedules. A Red Schedule is a set of tasks that are scheduled statically. This means that different application modes may have different tasks and therefore completely different behaviour. Haldex uses different application modes but the differences between the modes are not very large. Except for in the init mode there are just one or two tasks that separate the different application modes.

In an OSEK operating system one has the possibility to run a start up hook as the first thing after the operating system has been started. This works like the init schedule Haldex uses after Rubus OS has started so there is no need to do anything but look which tasks are currently placed in the init schedule and place them in the start up hook.

As the differences between the application modes in Haldex Traction's application are very small it is still possible to have them when using an OSEK/VDX operating system.

OSEK/VDX also supports different application modes but it is not possible to change mode during runtime, as is presently done with Rubus OS. This means that the application mode functionality has to be constructed.

In Haldex Traction's application, when changing to OSEK/VDX, there are at least two different ways to create different application modes. One way is to simply have a global variable that keeps track of the present application mode. Every time the application mode needs to be changed this is done by changing the value of the global application mode variable. The parts that differ in the different modes just have to check the application mode variable to know what to do.

Another solution is to have a message box linked to all tasks. This will work in exactly the same way as the global variable does. If a task wants to change the mode it sends a message with the new mode to the common message box and before an application mode dependent task start its execution it has to check the message box.

One advantage with the global variable is that this solution probably is faster. No operating system call is needed in this solution but an ordinary global variable has to be checked. On the other hand global variables are normally avoided so at this point of view the message box solution may be preferable.

5.4 Translation of the operating system calls

All Rubus OS operating system calls that are used in Haldex Traction's application need to be mapped to OSEK/VDX. For some Rubus operating system calls there is a similar operating system call in OSEK/VDX. In this case the only thing that has to be done is a change to the name of the OS call in OSEK/VDX.

Many OS calls in Rubus do not have a matching OS call in OSEK/VDX. These OS calls are a bit more complicated to map. The OS calls in OSEK/VDX, C-functions and assembly commands have to be used to make the application behave in the same way it did when calling the original Rubus OS call.

The following Rubus OS operating system calls are the ones that are used in Haldex Traction's application. Below is a design suggestion of how to translate each one of these OS calls:

blueFind:
The threads are always reached by name. Therefore this function is not needed in an OSEK/VDX operating system.

blueIntrSend:
This Rubus OS call sends an event to a task. If the task that receives the event was waiting for an event it becomes ready to execute. The OSEK OS call SetEvent does the same thing.

blueMutexFind:
This is used to find a Mutex specified by name. In OSEK resources are used instead of Mutex and it is always possible to reach a resource directly by name.

blueMutexInit:
OSEK does not use Mutex, but resources are instantiated in the OIL-configurator.

blueMutexTimedLock:
This OS call tries to lock a Mutex. If the Mutex is already locked it waits for the time interval specified by the in parameter timeout for another thread to unlock the Mutex. In OSEK/VDX resources are used instead of Mutexes. The OSEK OS call GetResource locks a resource but there is no possibility to set a timeout because of the OSEK Priority Ceiling Protocol (Immediate Inheritance Protocol) that OSEK OS uses. This protocol makes that a task never has to wait to lock a resource.

blueMutexUnlock:
The Mutex is unlocked and if other threads are waiting for the Mutex the scheduling policy is used to determine which thread shall acquire the Mutex. The same thing happens when using the OSEK OS call releaseResource.

blueName:
The threads are always reached by name and therefore this function is not needed in an OSEK/VDX operating system.

bluePreemptionLock:
This Rubus OS call makes that the calling thread cannot be pre-empted.
It is possible to disable all interrupts in an OSEK OS with the call DisableInterrupt. Before this is done it is important to use the call GetInterruptDescriptor to be able to save the current state of interrupts. The result of disabling all the interrupts is that the executing task or interrupt service routine cannot be pre-empted.

bluePreemptionUnlock:
If all interrupts were disabled to achieve the Rubus OS call bluePreemptionLock then pre-emption is enabled by enabling all the interrupts with the OSEK OS call EnableInterrupt.

blueSigEmptySet:
Clears the pending signals for a specific thread. A pending signal is a signal that has been sent but not yet received by another thread. Signals are used to send events.

In OSEK OS there is an OS call ClearEvent but the difference between Rubus OS and OSEK OS is that ClearEvent can only remove the pending events that the calling task owns.

blueSigSend:
This Rubus OS call sends a signal to a specific thread and if this thread was waiting for the signal, the thread is unblocked. The OSEK OS call SetEvent does the same thing.

blueSigTimedWait:
This function makes the blue thread wait the time timeout for the set of signals defined by set. It only waits if there is no signal pending at the time of the call.

There is an OSEK OS call WaitEvent that works the same way. The waiting thread is in the waiting state until at least one of the events specified has been set. The difference is that it is not possible to define a timeout in WaitEvent in OSEK OS.

blueSleep:
The blueSleep call causes the calling thread to be suspended for a specified amount of time.

There is no sleep function in OSEK OS.

blueStart:
Since there are no red or blue threads in OSEK OS there is no need for a blueStart call. The OSEK OS call StartOS starts the entire realtime operating system.

bsInit:
This Rubus OS call is used to initialise the Basic Services. This call does not need to be replaced with anything because an OSEK OS does not initialise through an operating system call.

bsLogPut:
This function puts a log object in a log queue. There does not exist a corresponding call in OSEK OS.

bsQueueGet:
This Rubus OS call takes the oldest message from a queue. The same functionality exists in OSEK COM. It is possible to have m-to-n communication with the unread messages stored in a FIFO queue. The OSEK COM call to get a message from a message queue is ReceiveMessage.

bsQueueInit:
This Rubus OS call is used to initiate a queue in Rubus. It is a FIFO queue that all tasks and interrupt service routines are able to write to and read from.

In OSEK the OSEK COM has to be started with the call StartCOM. The message queue and which tasks and interrupt service routines that are allowed to write to and read from the queue has to be configured statically before system generation in the OIL-file.

bsQueuePut:
This call puts a message into the message queue. The corresponding call in OSEK COM is SendMessage.

bsStatus:
bsStatus is used to check if the running thread is a Rubus Red or a Rubus Blue thread and to check the preemption status.

If it is important for the application's functionality to know if the task running is a former Rubus Red or Rubus Blue thread it is possible to solve this with a global status variable. The first thing every task has to do is to read and save the old status in a local variable and set the global status variable to the status of the task. When a task has finished its execution it uses the local status variable to change the global status variable back. This has to be done to support pre-emption.

The pre-emption status check can be solved in the same way.

bsTvToJiffies:
This OS call is used to calculate the number of timer ticks in a time interval.
The OSEK OS call GetAlarmBase returns among other things the number of processor ticks needed to increase the system counter. If this is known and the frequency of the CPU is known then it is possible to calculate the number of timer ticks in a time interval.

halBlueIntrContext:
This Rubus OS function is used to enter an interrupt service routine. There is no corresponding OS call in an OSEK OS. A call to the desired function is possible to perform.

redGetAllThreads:
This Rubus OS call returns the next Red thread that is going to execute. This is impossible in OSEK OS because OSEK OS uses dynamic scheduling. The dynamic scheduling means that which task is going to execute next is decided at runtime just before the new task starts.

redInit:
This is used to initiate the Red Kernel. This is not needed in OSEK/VDX.

redMaxTimeGet:
This Rubus OS call returns the longest execution time registered for the task. There is no similar call in OSEK but it should not be hard to implement it. Read the system clock in the pre task hook and store the value in a temporary variable. Read the system clock again in the post task hook, calculate the execution time and if it is longer than the earlier stored execution time value, store this value instead. The longest execution time is stored in the execution time variable.

redSetSchedule:
This function is used to choose the appropriate red schedule. In OSEK/VDX it is not possible to change execution state while running but it is possible to make an implementation that solves this problem according to the solution in Chapter 5.3.

6 Test application

6.1 Introduction

Haldex Traction's application is mostly made up of Rubus Red threads. These threads are statically scheduled and time periodic. An important thing for Haldex Traction to ensure when changing realtime operating system from Rubus OS to an OSEK OS is that the time-periodic activities work as they do today.

Time periodic activities in OSEK OS are solved with the help of alarms and timers. With a cyclic alarm connected to a timer it is possible to set a time-periodic event. With the event it is possible to activate a task.

Rubus OS uses static scheduling and this is not possible in OSEK OS. Haldex Traction's application only contains one protected resource and this resource is only called from threads with the same priority so there will be no changes in priorities because of resources. If the tasks are given the correct priorities they will execute in the desired order. The only difference that will be noticed between Rubus OS and OSEK OS in Haldex Traction's application is that the scheduling that is performed at runtime in OSEK will take a small amount of time, but this time is so small that it will not affect the functionality of the application.

6.2 The test hardware and OSEK OS

Haldex Traction is currently using the micro-control unit Infineon C167cs. For the next generation of the control unit, generation 3, they have changed the micro control unit from C167cs to ST10F272 from ST Micro Electronics. If Haldex Traction decides to change to an OSEK realtime operating system, this will not be introduced in an earlier generation than generation 3.

For the test application there were several advantages to test OSEK OS with Haldex Traction's present micro-control unit Infineon C167cs. The first advantage is that most of the distributors of OSEK OS already have a configuration for this micro-control unit on the shelf. If the ST10F272 should be used, most of the OSEK distributors would have to do some configurations and tests to make it work and this both takes time and costs money.

Secondly, the staff at the TTE division at Haldex Traction are experts on the C167cs micro-control unit.

The last reason for using the C167cs for the test application is that Haldex Traction has more available debuggers for this processor.

The OSEK realtime operating system that was used in the test application is Vector Informatics OSEK implementation osCAN. This is presently the best suited OSEK realtime operating system for Haldex Traction.

6.3 The test application

The test application should be as simple as possible but still test the use of time-periodic activities in an OSEK OS. Haldex Traction's limited slip coupling uses a small electrical pump controlled by the control software. This pump is used to obtain a base pressure of the oil in the coupling. The good thing about the pump is that it is easy to control and it is possible to verify the functionality of the pump just by listening to the sound from it.

The test application consists of one time-periodic thread that first increases the duty cycle of the pump from zero to one thousand and then decreases the duty cycle form one thousand back to zero and then starts from the beginning. The duty cycle is changed in steps of one every second millisecond. The duty cycle determines the speed of the pump where zero is no pump activity and one thousand is maximum speed.

This small test application that makes calls to Haldex Traction's code to steer the pump is a short and simple C-application. osCAN was configured to have one time periodic thread that starts every second millisecond in the OIL-configurator, which was included with the realtime operating system. The configurations needed in the OIL-configurator were very fast and simple.

A Red Rubus schedule consisting of one thread, the test thread, was configured to start every second millisecond to be able to test the test application with the old realtime operating system before changing to a new one. The application worked as expected and there were no problems.

The software architecture of the test application is illustrated in Figure 6-1 below.

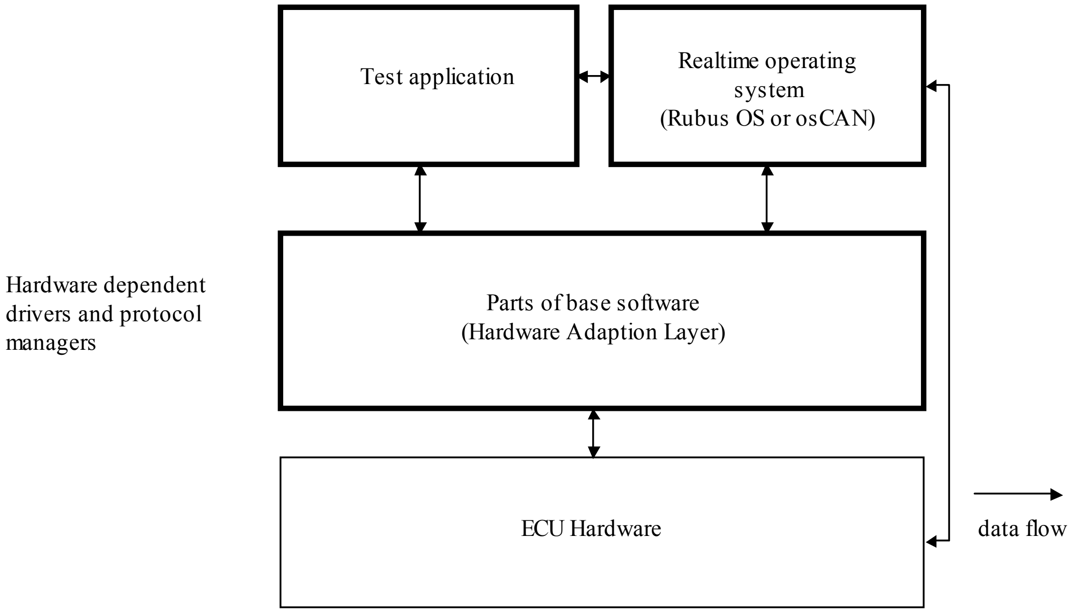
Test application
Realtime operating system (Rubus OS or osCAN)
Hardware dependent drivers and protocol managers
Parts of base software (Hardware Adaption Layer)
ECU Hardware
data flow

Figure 6-1 Software architecture of the test application

- **Test application:** The test application contains the test function that increases or decreases the duty cycle of the pump at each call.
- **Realtime operating system:** The test application is tested both with Rubus OS and with the OSEK OS osCAN. In the test application the realtime operating system is used to perform the scheduling.
- **Parts of the base software:** Parts of Haldex Traction's base software is used to be able to control the pump. The necessary initialisation methods are used as well as the methods to steer the pump.

6.4 Two different approaches

Haldex Traction already has a well working application that uses Rubus OS as its realtime operating system. When integrating this large already well-working application with a new realtime operating system there is another problem except for the functional differences between the two realtime operating systems. The memories in the processor, C167cs, are set up differently in Haldex Traction's application compared with how it is set up in osCAN. The interrupt vector table is also generated in different ways in the two systems. Haldex Traction's application and osCAN use completely different compiler and linker flags and the structure of Haldex Traction's makefiles compared with osCAN's makefiles are organized completely different.

There are two different approaches when integrating the two systems. One approach is to use Haldex Traction's makefiles, do everything in the same way as they do today and just replace Rubus OS with osCAN.

The second approach is to use the makefiles included with the osCAN product, use the way Vector Informatic sets up the system and just add Haldex Traction's application.

Both the approaches where tested in this master thesis.

The exact same test application was used in both approaches and with the same OIL-file made with the help of the OIL-configurator that was included with osCAN.

6.4.1 Using Haldex Traction's makefiles

This approach was tried as a first attempt to minimise the work of converting the system to work with OSEK OS. The work of replacing Rubus OS with OSEK OS was done in the following steps:

The first thing done was to compile the OSEK OS files and the test application in Haldex Traction's makefiles. There were no major problems with this. The second thing was to remove all Rubus OS calls in the parts of Haldex Traction's code that was not used by the test application. The part of Haldex Traction's code that is used to steer the pump does not contain any Rubus OS calls, so no translation of OS calls had to be done in the test application. Also the start up procedure in Haldex Traction's application which initiates everything and eventually starts Rubus OS, had to be replaced with a new start up procedure that initiates the things necessary to control the pump and starts osCAN instead of Rubus OS. When the Rubus OS calls were removed from the code it was possible to remove Rubus OS from the linking and compilation in Haldex Traction's makefiles. Also this went well.

The next and last thing to do was to include osCAN in the linking procedure in Haldex Traction's makefiles. This is when the problems started. The processor's memory was configured differently in the two systems so this had to be changed in osCAN which succeeded after a while. The real problems had to do with the interrupt vector table that is needed to start the realtime operating system. osCAN needs the linker flag NOVT (no vector table) to work but if this linker flag is introduced in Haldex Traction's make and linker structure it does not work. After trying to get around this for about four weeks without succeeding the next approach was tested instead.

6.4.2 Using osCAN's makefiles

If starting with the correct OIL-configuration but with the single thread empty it is no problem to compile and link osCAN and the empty application with the make structure included with osCAN.

When this was done the code from Haldex Traction's application necessary to control the pump was compiled with osCAN in osCAN's makefiles. Next step was to add the necessary initialisation of Haldex Traction's pump control software in the osCAN StartUPHook which is executed right after the operating system has started but before the scheduler has started. A call to the test application was added in the empty body of the time periodic task. This was easily done and when it did compile and link the format of the output-file was changed in the makefiles to the *.sre format that is used in the processor C167cs.

The test application behaved in the desired way with the pump spinning up and down with a period of four seconds.

To convert Haldex Traction's entire application, continue with this approach and port one module or task at a time. Use the design suggestions in chapter 5 to make it work.

Summary

OSEK/VDX is an industry standard developed as a joint project by the German and the French automotive industry. The goal is to minimise the expenses caused by recurring non-application related work and to reduce incompability between control units made by different distributors caused by the use of different interfaces and protocols.

The main goal behind the OSEK/VDX standard is very good. The negative aspects of the project is if companies, like for example Haldex Traction, is forced to change their good working application not because they need the standard but because they are forced by customers. Haldex Traction's realtime operating system Rubus OS uses static scheduling, which is faster and safer than dynamic scheduling but OSEK/VDX does not have a good answer to this yet. OSEKtime which uses static scheduling exists but has not been fully accepted by the realtime operating system distributors.

There are some different alternatives on the market when looking for an OSEK/VDX realtime operating system. The API of the realtime operating systems is specified by OSEK/VDX so the main differences are in how many and how much of the OSEK/VDX documents that are implemented, the price and the performance (amount of memory required and response times). The OSEK/VDX realtime operating system best suited for a specific company depends on that company's demands from the realtime operating system. The OSEK/VDX realtime operating system found to be best suited for Haldex Traction and their application is Vector Informatic's realtime operating system osCAN.

There are many differences between Rubus OS and osCAN that are described in this report. It should not be any problem to make Haldex Traction's application work like it does today with an OSEK OS instead of Rubus OS. If the design suggestions in this report are followed the work of changing realtime operating system should not be very large. A rough estimation is that this work will take one year for one person including testing or better one half calendar year for two persons.

References

[1] What is OSEK/VDX? http://www.osek-vdx.org/whats_osekvdx.html 2004-08-25

[2] OSEK (2004): OSEK/VDX Operating System Version 2.2.2

[3] OSEK (2004): OSEK/VDX Communication Version 3.0.3

[4] OSEK (2004): OSEK/VDX Network Management Version 2.5.3

[5] OSEK (2004): OSEK/VDX OSEK Implementation Language Version 2.5

[6] OSEK (2001): OSEK/VDX Time-Triggered Operating System Version 1.0

[7] EUROSmot http://www.euros-embedded.com/e/osek.htm 2004-08-25

[8] ERCOSEK http://www.spacetools.com/tools4/space/404.htm 2004-08-25

[9] Metrowerks OSEKturbo
http://www.metrowerks.com/MW/Develop/Embedded/OSEK.htm 2004-08-25

[10] RTA Software Products http://en.etasgroup.com/catalog/pdf_04/3_6.pdf 2004-08-25

[11] 3SOFT http://www.dreisoft.de/index_e.htm 2004-08-25

[12] Nucleus RTOS http://www.acceleratedtechnology.com/embedded/osek.php 2004-08-25

[13] OSEK Compliant Technology http://www.trialog.com/RealTimeAuto/osek.html 2004-08-25

[14] VECTOR INFORMATIK http://www.realtime-os.info/index.html??osekvdx.html 2004-08-25

[15] Haldex http://www.haldex.com 2004-08-25

[16] Haldex Traction http://www.haldex-traction.com 2004-08-25

[17] RTOS Partners http://www.hitex.com/pa4prod.html 2004-08-25

[18] Embedded Controller Hardware and Operating System Selection for MoBIES Powertrain Testbed
http://vehicle.me.berkeley.edu/mobies/powertrain/reports/Controller_Hardware_and_Operating_System.doc 2004-08-25

[19] Embedded Real-Time Operating System (RTOS) Vendors
http://www.dspconsulting.com/rtos.html 2004-08-25

[20] Personal contact information.

[21] OSEK (2001): OSEKtime FTCom specification

[22] Årzén K. (2003): Real-Time Control Systems

[23] Arcticus Systems (1999): Rubus OS Reference Manual 2.0