ISSN 0280-5316
ISRN LUTFD2/TFRT--5684--SE

Porting the Ericsson Bluetooth Stack
A Real-Time Analysis

Mats Attnäs
Ulrik Laurén

Department of Automatic Control
Lund Institute of Technology
April 2002

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
April 2002

Document Number
ISRN LUTFD2/TFRT--5684--SE

Author(s)
Mats Attnäs and Ulrik Laurén

Supervisor
Karl-Erik Årzén. LTH
Pär-Gunnar Hjälmdahl, Ericsson

Sponsoring organization

Title and subtitle
Porting the Ericsson Bluetooth Stack – A Real-Time Analysis.
(Flyttning av Ericssons Bluetooth stack – En realtidsanalys)

Abstract
This master's thesis discusses the real-time issues of the operating system used by the Ericsson Bluetooth stack and the effects of replacing this operating system.
The practical part consists of switching the operating system for the Ericsson Bluetooth stack and verifying if it still is operational and fulfils all timing requirements.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
104

Recipient's notes

Security classification

Porting the Ericsson Bluetooth stack

– A real-time analysis

Master's thesis by

Mats Attnäs and Ulrik Laurén

Lunds tekniska högskola

April 2002

1 Contents

2 Introduction

2.1 Goal and purpose

The main goal of this master thesis was to switch the current real-time operating system that powers the *Ericsson Bluetooth embedded platform* (BEP).

It is recommended that the reader of this master thesis have at least a basic understanding of real-time operating systems and embedded platforms to fully comprehend and profit from the contents.

The reason for this was to evaluate if and how it could be done, and what problems one may encounter. Also, it would show if the intention of making the platform portable had been successful or not. This could be useful for future applications in other hardware or software environments, allowing a greater flexibility and possibility to integrate the system with existing client products.

The thesis was done in co-operation with *Ericsson Technology Licensing*.

2.2 What we have done

The work has mainly consisted of three parts:

- Preparatory investigation of suitable operating systems for porting.
- Adaptation of the system to interface with the new operating system.
- Analysis of the performance differences of the two ports, and their implications.

2.3 What we have learned

- The principles of the Bluetooth protocol.
- How an embedded application is designed and what its limitations are.
- How real-time operating systems work, and differences between design principles of such.
- How to debug and examine an embedded application.

2.4 Requirements and constraints

There are quite a few constraints placed on the work performed in the master thesis since it deals with an already existing technology that should be modified, not rebuilt. The main requirement is that the system has to be able to function in such a tightly planned hard real-time environment that BEP offers. But since it is designed for an embedded environment there is also a strict limit for how much memory that could be used by the system.

Both of these requirements are dealt with by most real-time operating systems but there are other limitations placed on the system. It is necessary for the RTOS to function with the ARM7 processor architecture both in ARM and Thumb mode (see Appendix VII for details on the ARM architecture). The reason for this is that BEP has parts of it developed in assembly language that utilise both of these modes and the test environment use the ARM7 processor.

The final requirements were that no changes should be made to the current development environment and that the new operating system had to fit into the model of the system. The reason behind these requirements was that this master thesis should also try to discern any problems that could arise when working with a new real-time operating system. This meant that it was necessary to develop systems that support the existing functionality implemented in the *virtual operating system* (VOS) using the functions available in the new operating system. This requirement also prevented any changes to the programs used in the original development environment.

2.5 Investigated operating systems

To implement the port, a target operating system had to be found first. Three factors were especially important: Real-time performance, compactness and availability.

It was also necessary to gain basic knowledge of the existing operating system, OSE Epsilon, to know how to adapt the new operating system. The principle of the VOS is to require only very basic functionality from the operating system, thereby allowing many different operating systems.

Our choice of target operating system finally fell on µC/OS-II, because of its simplistic and efficient design and its royalty free license.

2.5.1 OSE Epsilon

Supported target processors	Siemens C166/167, ARM7/Thumb, NEC V85X, Atmel AVR, Mitsubishi M16, 8051, 68HC11, Z80, 64180
Supported compilers	Distributed as target specific assembly code
Supported standards	None
Supplied as	Assembly code
Guaranteed maximum interrupt latency	Hard real-time
Scheduling policies	Fixed priority preemptive scheduling (priority processes) + Round robin (background processes)
Priority inversion avoidance mechanism	None
Nested interrupts	Yes

OSE Epsilon is a real-time operating system developed by Enea OSE AB. They also have a couple of other versions of their operating system but Epsilon is the smallest version. It is the main real-time operating system used by Ericsson and the platform currently used for the Ericsson Bluetooth stack.

2.5.2 µC/OS-II

Supported target processors	x86, PowerPC, ARM, MIPS, StrongARM, NEC V850, Hitachi SH, and many more
Supported compilers	ANSI-C
Supported standards	None
Supplied as	Source
Guaranteed maximum interrupt latency	Hard real-time
Scheduling policies	Fixed priority preemptive scheduling
Priority inversion avoidance mechanism	None
Nested interrupts	No

µC/OS-II is a very small real-time operating system developed by Jean J. Labrosse of Micriµm Incorporated. Its source code is openly available, and the terms of use are very advantageous.

2.5.3 eCos

Supported target processors	x86, PowerPC, ARM, MIPS, StrongARM, NEC V850, Hitachi SH, Panasonic AM3x, SPARC
Supported compilers	GCC (GNU)
Supported standards	EL/IX, ISO C, POSIX.1, µITRON
Supplied as	Object, source
Guaranteed maximum interrupt latency	Soft real-time
Scheduling policies	Prioritised FIFO, bitmap
Priority inversion avoidance mechanism	Priority inheritance, priority ceilings
Nested interrupts	Yes

eCos is a highly configurable real-time operating system for deeply embedded applications, maintained by Red Hat. It is *not* an embedded Linux, though it offers POSIX.1[1] compatibility, as well as an optional µITRON[2] compatibility layer.

eCos is also an open source initiative, and therefore free of charge.

2.5.4 Nucleus PLUS

Supported target processors	x86, PowerPC, ARM, MIPS
Supported compilers	ANSI-C
Supported Standards	µITRON, OSEK
Supplied as	Source
Guaranteed maximum interrupt latency	Hard real-time
Scheduling policies	Prioritised FIFO
Priority inversion avoidance mechanism	Yes
Nested interrupts	Yes

Nucleus PLUS is a royalty free, small and powerful RTOS from Accelerated Technology, optionally supporting µITRON or OSEK[3].

[1] Set of standards designed to provide application portability between Unix variants
[2] Application interface for real-time systems, widely used in the Japanese embedded market
[3] An RTOS interface standard used mainly in the vehicle manufacturing industry

2.5.5 VxWorks

Supported target processors	x86, PowerPC, ARM, MIPS, 68K, i960, SH, SPARC, NEC V8xx, M32 R/D, RAD6000, ST 20
Supported compilers	ANSI-C/C++
Supported Standards	POSIX 1003.1b
Supplied as	Object
Guaranteed maximum interrupt latency	Hard real-time
Scheduling policies	Prioritised FIFO, round robin
Priority inversion avoidance mechanism	Yes
Nested interrupts	Yes

VxWorks from Wind River Systems is the most widely used RTOS in the world, and is both scalable and efficient.

2.6 Outline of the thesis

A short description of the information presented in the chapters:

In chapter 3 there is an introduction to the more important concepts necessary for understanding the contents of this report. The chapter is followed by an analysis of the Bluetooth specification and the Ericsson solution, focusing on either real-time or memory constraints. This is followed by a short introduction and analysis of the three operating systems that were examined thoroughly, namely OSE Epsilon, eCos and µC/OS-II. After this introduction and analysis, a detailed description of the modifications that were necessary for the Ericsson solution to function with its new operating system, µC/OS-II, follows. The performance tests and their implications are presented and discussed in the next chapter, followed by a short description of the programs used. This is followed by the conclusions of the master thesis and finally there are the acknowledgements and references followed by a short glossary of important abbreviations.

Following all these chapters are a number of appendices that are referenced from various parts of the report.

3 Background

3.1 Real-time operating systems

3.1.1 Introduction

An operating system is basically the software needed to run applications, programs, on a computer system. This software includes process handling, hardware drivers and memory handling, all those things that often are taken for granted when using modern computers.

When real-world interaction becomes an issue, so do real-world time constraints. These constraints are most often of a temporal nature, it is necessary to complete certain tasks within a certain time. When working with humans these interactions often have constraints in the range of a few seconds, but when interacting with other computer devices these timing constraints are reduced to very small units.

For this reason, real-time operating systems contain certain basic primitives:

- Prioritised processes/threads
- Multileveled interrupts
- Mutexes and semaphores

3.1.2 Classification of a real-time operating system

There are two basic types of real-time operating systems, *hard* and *soft* ones. The hard real-time systems are predictable; it is possible to guarantee maximum and minimum execution time for system calls, procedures, and interrupt latencies. Hard real-time operating systems therefore have very high demands in terms of scheduling and interrupt handling.

In soft real-time systems, the system only needs to behave in a consistent manner, being allowed to fail to fulfil the time constraints from time to time. This leeway allows the system to employ somewhat less strict interrupt handling, and the scheduling can be done in a not so strict and predictable manner.

Real-time operating systems can alternatively be grouped according to their interaction with their surroundings. They can be either time or event driven, or in rare cases both.

An event driven real-time operating system reacts to events that occur during execution, and can modify the order of execution, while a time driven real-time operating system has a fixed execution schedule that is calculated before execution starts.

3.1.3 Scheduling

The scheduler plans the execution to ensure that the processes can fulfil their deadlines. However, the running process may be interrupted by a more important process, in which case the operating system has to be able to do a context switch, letting the more important process run. This is allowed only if at least one of the following conditions are met:

- A clock tick interrupt has occurred. Clock tick interrupts occur when the system updates the time, and checks if any other task with a higher priority wants to run.
- A running task performs a system call.
- The interrupt handler orders a switch due to an external interrupt.

The system can be divided into four distinct layers, see Figure 1. The bottom layer is the hardware that contains CPU, physical memory, a clock, and communication hardware. On top of this layer is the hardware adaptation layer, containing functionality for registers and interrupt handling. The real-time operating system layer consists of functionality for scheduling, synchronisation and interprocess communication. The highest layer consists of applications that run on the platform.

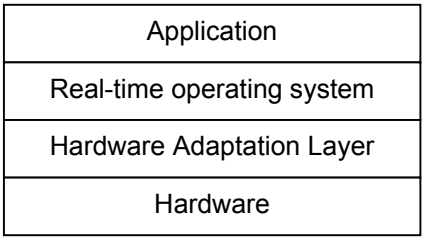

Figure 1. The structure of an embedded solution

3.1.4 Internal components

This section describes the internal components of a typical real-time operating system. Many of these components can also be found in standard operating systems.

3.1.4.1 Scheduling mechanism

The scheduling mechanism plans the order in which the active processes are to execute. This planning can be done according to a variety of principles. The basic concept for all of them is that processes have a deadline to uphold. The two most common algorithms are rate monotonic scheduling and earliest deadline scheduling. Both of them have pros and cons. Rate monotonic scheduling has a stabile, predictable behaviour, but assumes that all tasks are independent. Earliest deadline scheduling is easy to implement and uses the CPU very efficiently, but may result in unpredictable behaviour if a task fails to meet its deadline. Both algorithms are designed to be used in an event driven operating system, where new processes are activated due to external events, and not according to a predefined plan.

3.1.4.2 Memory handler

The memory handling system is responsible for the handling of dynamic memory, with the use of a central memory pool. When a process has finished using a memory portion, it returns it to the memory pool. To prevent memory fragmentation it is important to have a good algorithm for memory allocation, especially in small, embedded systems, where often the amount of available memory is very limited.

3.1.4.3 Interrupt handler

Interrupts can be activated from either hardware or software, and both have to be handled. This is taken care of by the interrupt handler, which can force context switches. If the priority of an interrupt is not high enough it will be discarded, and the running process will continue running. A discarded interrupt is dealt with either by ignoring it, hoping that if the problem remains a new interrupt will occur, or delaying it until the executing processes has finished. When an interrupt is called, there is a short delay before the interrupt can start executing. This delay is called interrupt latency, and during this period the executing process is stored away and the new process, activated by the interrupt, is placed into memory ready to start executing.

3.1.4.4 Clock

An external clock generates ticks through hardware interrupts to the operating system, allowing it to update the internal clock. Processes used to measure time intervals and to set timers to activate after a certain amount of ticks can then use the internal clock. This is necessary if the operating system is designed to support periodical and sleeping processes. The precision of the clock has to be very good to prevent drift over long periods.

3.1.4.5 Error Handler

The error handling consists of a central system that can handle exceptions from running processes and activate a suitable response. This response varies according to the severity of the error encountered. Serious errors can cause the error handler to terminate the executing process or even freeze the entire system.

3.1.4.6 Data access synchronisation

Access to shared resources, such as memory areas or devices, can be controlled using some kind of signalling device that prevents other processes from using that resource. This can be implemented in many different ways, but it is often implemented with semaphores or mutexes.

It is necessary for a real-time operating system to prevent simultaneous access to resources because this might result in corrupted data.

3.2 Embedded system

An embedded system is a computer system that is integrated into a device, performing a certain task. The size and complexity of both the computer system and its task may vary substantially, though generally it is a matter of small devices with very limited processing and memory capabilities. Furthermore, embedded systems often act in co-ordinance with external events and conditions, such as sensors and timers, rather than actual user interaction.

3.3 Bluetooth

Bluetooth is a wireless communications standard developed by the *Bluetooth Special Interest Group* (SIG). Its main focus is to provide wireless connectivity between any kind of electronic device, at a low price and easy access. To put it simple, Bluetooth reduces the need for cables and specific communication protocols between devices, allowing everything to communicate with anything.

Bluetooth powered units can communicate and send data or voice over the connections. The existing standard is designed to support a multitude of different devices including everything from headsets for mobile telephones to connecting computers to networks.

In Bluetooth, there are *masters* and *slaves*. A master can have many slaves, and may also be the slave of another master. A master with slaves is called a *piconet*. Several inter-connected piconets form a *scatternet*.

3.4 Bluetooth specification and the Ericsson solution

The Ericsson Bluetooth solution for embedded systems is a hardware and software platform, see Figure 2. The hardware contains the baseband controller block (EBC), a CPU, some RAM, some ROM, some flash memory and input/output ports (UART and USB). Apart from the application, the software mainly consists of two parts: the Bluetooth stack and the operating system. To make a port of the stack as easy as possible, an interface layer (VOS and templates) is put between the two. All operating system calls from the stack are consequently done through this layer.

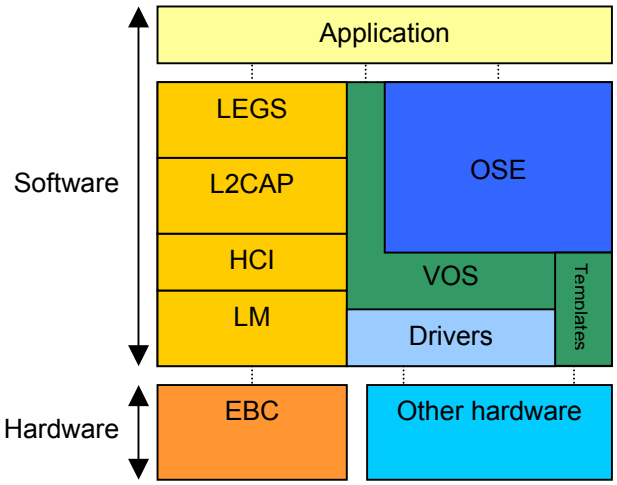

Figure 2. Schematics of the Ericsson Bluetooth platform, and its surrounding components, both hardware and software

In a host solution where the Bluetooth device is not planned to run everything on its own processor the layout of the components is a bit different as shown in Figure 3.

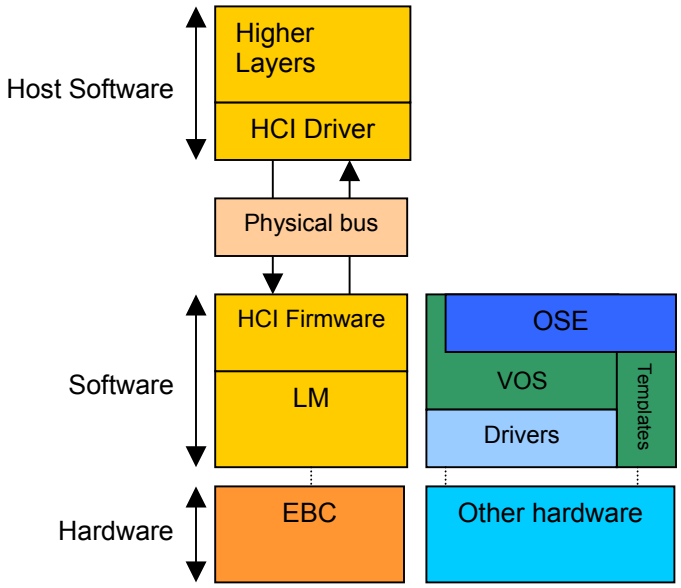

Figure 3. Shows a schematic describing the host version of the Ericsson Bluetooth platform.

3.4.1 Link Manager

The link manager is the lowest software layer in the stack. It controls the usage of the common channel, connections of new slaves and searches for available units. It also implements security for the data transferred through encryption and controls how to portion packets between the master and the slaves on the common channel.

The physical channel is divided into equally sized time frames of 1,250 µs each. Each frame is divided into two slots, which hence are 625 µs each. The master in a connection uses the first slot in each frame to send data and the second slot to receive data from the slaves.

This layer is very tightly coupled with the Ericsson baseband controller, EBC. It is the main product developed by Ericsson Technology Licensing, EBT.

3.4.2 Host Control Interface

The host control interface implements a uniform interface for accessing the lower layers of the Bluetooth stack. This layer is divided into two parts when implementing a host based solution. The commands can be transferred over some kind of physical medium e.g. USB, RS-232.

3.4.3 Higher layers

The function of the higher layers is to implement the services that are necessary for the different roles that the Bluetooth device can be used in.

In what is called the *embedded solution*, the higher layers are a part of the software running on the same CPU and using the same memory. This usually puts higher demands on the available resources and may result in changes to the operating system used in the solution.

3.5 Communication links

There are two defined types of links that can be established between master and slave unit:

- Synchronous Connection-Oriented link (SCO link)
- Asynchronous Connection-Less link (ACL link)

3.5.1 SCO link

This link is designed for time-bounded information, e.g. voice communication between a master and a specific slave in a piconet. The master can have a maximum of three different SCO links established at the same time to either one or several different slaves.

All the SCO links are point-to-point and maintained by using reserved slots at regular intervals, adherence to the protocol is therefore important so that a reserved slot is not used for another type of communication thereby losing the link. No data sent over the link is repeated. There is not even a retransmission system for lost data packets. Slots not reserved for the SCO link can be used for other types of communication between Bluetooth units.

In an SCO link is it always possible for a slave to send data to the master as a response to a message from the master, unless it was directed to another slave unit.

3.5.2 ACL link

The ACL link is used for point-to-multipoint communication in a piconet. It is possible to create an ACL link to a slave that is already involved in an SCO link by using the slots that are not already reserved. In contrast to the SCO link there may only exist one ACL link between a master and a slave.

Data transferred over an ACL link is most of the time protected by a retransmission system that assures data integrity. Communication from a slave to the master is only allowed if the slave was addressed directly by the master. Packets sent without being addressed to a specific slave is considered to be a broadcast and is read by all the slaves connected to the piconet.

4 Analysis of the Bluetooth specification

4.1 Introduction

Ideally, we would do a white box analysis of the existing system, determining timing requirements and limitations. Due to insufficient documentation on the high and mid level design, though, this was not feasible. We instead performed a black box analysis of the existing timing constraints in the Ericsson solution.

4.2 Timing requirements

4.2.1 Timing requirements in the general Bluetooth specification

The current specifications for Bluetooth do not contain any demands concerning software latencies in the system. The only existing constraints concern the hardware supporting the software system. The radio system has a ±10 µs long window for receiving data. This prevents minor clock mismatches from disturbing the system.

However, the time limits in the hardware propagate to the software in the Ericsson solution. Therefore, a short description of the different transmission time limits is given.

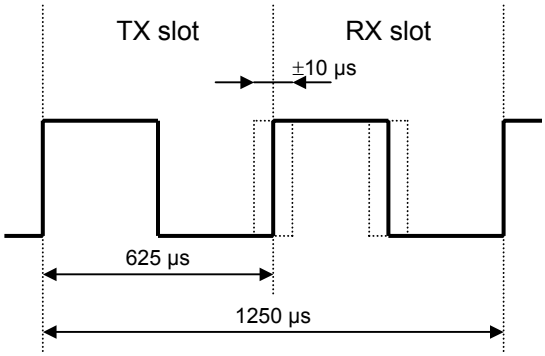

Figure 4. The time limits of the Bluetooth protocol

The Bluetooth protocol sends and receives information in frames. Each frame is divided into two slots, one TX slot and one RX slot. During the TX slot the unit can send data and during the RX slot it can receive data. Each slot is 625 µs long and a frame is therefore 1,250 µs long. For a master, the first slot is TX and the second RX. For slaves, the order is reversed.

4.2.2 Timing requirements in the Ericsson solution

In the Ericsson solution, the transmission system for the hardware consists of two small buffers, one for sending packets and one for packets to be sent next. The reception system is similar, but with one buffer for receiving packets and one buffer for previously received packets.

It is possible to decide packet type and create the specific packet during a period of 1,250 µs but it is much easier to plan and create packets for a longer time period. The reason for this is that voice transfers are made with SCO packets and these have a fixed interval. A SCO packet has to be sent every third frame if the system is not to lose quality. Voice communication usually continues during long periods of time, at least when time is measured in µs. But if the delay between incoming data and its outgoing response is too long, the system loses much flexibility. The ability to plan ahead also removes some sensitivity concerning interrupts but the system is still sensitive for interrupts when it comes to writing new data into the buffer.

4.3 Memory requirements

4.3.1 Memory requirements in the general Bluetooth specification

There are none.

4.3.2 Memory requirements in the Ericsson solution

The existing Ericsson solution that we used was developed on Lydia boards using the Irma C circuit. This development environment is equipped with 56 kB of RAM and another 512 kB of flash memory. The entire system with real-time operating system and processes for controlling the Bluetooth system has to function without any extra memory space. This place high demands on the memory footprint of a new operating system. The existing real-time operating system kernel, Enea's OSE Epsilon, fulfils the demands by implementing only the most important features. It has the following features:

- inter-process communication
- dynamic memory management
- error handling
- hardware and software interrupts
- timer interrupts (alarms)

- synchronisation mechanisms

- pre-emptive process scheduling

The Ericsson solution of the implementation of the Bluetooth stack should not be affected by the used real-time operating system. The solution to this problem is a layer around the real-time operating system called core.

4.4 Core

4.4.1 Introduction

The core layer, described in [6], contains all the functionality that is necessary to run the Bluetooth stack. It contains a couple of different modules each implemented to handle one specific function for the stack. This design was chosen to make the system portable because no changes have to be made to the stack when the operating system is changed. Nevertheless, a few of the modules in the core have to be updated and the templates modified to work with the new operating system.

The modules VOS, IRQ, handler control, power management, timer, clock and cache are contained in the core layer.

4.4.2 VOS

The virtual operating system acts as an interface between the operating system's services and the Bluetooth stack. This module and the operating system have to be configured for specific applications. This configuration is done with the template module, which is not a part of the core.

Drivers and applications use the functions offered by VOS in order to simplify porting of the implementation to another operating system, since operating system specific commands are only used within VOS, and not scattered all over the entire code. When porting the system the procedures implemented in VOS have to be rewritten to use the new functions presented by the new real-time operating system. This is based on the hypothesis that a new real-time operating system is very similar to the existing system and has comparable functions.

4.4.3 IRQ

The interrupt handling module handles interrupts from the hardware and supports nested interrupts with different levels of priority. By supporting nested interrupts, this module allows interrupts with a higher priority to execute immediately.

Procedures that are called at a certain interrupt are called *interrupt service routines*. They are registered in this module.

No changes should be done to the interrupt handler when porting to another operating system, because all modifications will be found in templates.

4.4.4 Handler Control

This implementation of core supports a group of handlers that are controlled from this module. Different processes register handler events, triggering the registered handler when the events occur. A couple of examples of different handlers are the following

- Error handler is called when an error occurs in another module in the core or directly from the RTOS through the VOS.
- Send handler is used to debug the system in runtime and triggers when modules send messages.
- Trace handler is used for output of debug text.
- Event handler is used to log important events.
- Idle handler is the process that runs when the system enters idle mode.

4.4.5 Power Management

This module contains methods for controlling the power consumption through activating and deactivating different parts of the hardware. The module depends on the VOS, IRQ and timer modules and must be modified if the necessary functions used by VOS are not available in the new operating system.

4.4.6 Timer

This module contains the functionality to measure time and create alarms that perform specified procedure calls after a set period. It also contains information about how long time has passed since the latest system reset.

4.4.7 Clock

The clock module handles the different hardware clocks available in the system and controls the frequency of the CPU. Through this module it is possible to access the different hardware clocks and set their frequency.

4.4.8 Cache

This module handles the cache memory available in the hardware through initialisation and configuration of the cache.

4.5 RTOS

This is not exactly a module in the same sense as the other parts of the core, it is the heart that powers the core. This module contains functionality for process scheduling, dynamic memory handling, inter-process communication and data synchronisation.

When exchanging the RTOS is it important that the new system is not too complex with many modules and that the drivers are independent from the kernel.

5 OSE Epsilon for ARM

This chapter will present and discuss the different results that have been investigated and presented in [3] and give an insight into the current operating system used by Ericsson Technology Licensing in their Bluetooth solution. This report also contains measurements of the different times that were considered critical to the performance of the existing solution based on OSE Epsilon.

5.1 Problems

The only major problem with OSE Epsilon is the fact that the code is not available for analysis since a program generates it and it does not contain any comments. All data about how the internal functions work in the operating system has to be mapped through experiments.

5.2 Interrupt handling

The interrupt handling implemented for OSE Epsilon handles nested interrupts with five different levels of priority, an interrupt with a higher priority can supersede a lower prioritised. If nested interrupts occur the system stores all registers used by the previous interrupt on a special stack. This stack has enough room to store one set of registers for each level of priority. If a task is interrupted the registers are saved in the task's control block making it easy to locate them afterwards.

The functionality for this is located in a few assembly macros that exist in the file `osarm.mac`. The macros are `INIT_INT_NESTED()` and `QUIT_INT_NESTED()`.

5.3 Memory handling

Memory handling in OSE Epsilon is based on lists containing memory chunks of different sizes. These memory chunks are not created at initialisation but created later, out of a large memory pool, when they are needed. The disadvantage of this is that it may result in a slightly longer time to allocate a new memory chunk since it has to be created. The advantage is that the setup time during system initialisation should be shorter and it also gives a greater flexibility since it is not necessary to know the exact number of the different memory chunks that will be used.

There is a hidden header in front of the allocated memory area that contains 14 bytes of data. Most of the fields are two bytes in length except the first field that is four bytes long and the last field that is only one byte long. This header contains the following fields:

- Next This field contains a 32-bit value that is used as a pointer to the next memory chunk.
- Owner Defines the owner of the memory chunk, a value of zero means that OSE Epsilon owns the chunk.
- Size This field contains the memory chunk size that was requested by `VOS_Alloc()`.
- Head A field that describes in which memory size group the chunk is registered.
- Sender Contains the identifier of the task that sent this memory chunk to another task. It is set to zero when the memory is allocated.
- Sig_no A number that relates to the memory chunk when it is used as a signal.
- Data The head of the memory chunk that can be used by the different tasks to store data in. This field is only one byte in length.

5.4 Timing

OSE Epsilon fulfils the demands that are placed on it regarding the time critical sections. This was proven in [3].

5.4.1 Interrupt service times

Presented in the report are details about the interrupt latencies where a value of 40 – 45 µs is presented for the total delay from the hardware as well as the operating system. OSE Epsilon uses 30 – 35 µs to load the interrupt registers and perform a context switch. Hardware and sections of the code that are locked from interrupts use the rest of the time. The longest time it takes for completing an interrupt service routine is 330 µs. This has been presented in [3] when describing a reception of a RD interrupt. A RD interrupt means that the system has received an ACL data packet. The most common interrupt is the timer interrupt and it takes a total time of 260 µs to execute this interrupt service routine in the setup used in [3].

5.4.2 Scheduling

The case presented for performing a pre-scheduling is two timer interrupts, one to start the scheduler and one to start the EBC transfer, followed by the maximal processing time necessary and the time necessary for transfer the results to the EBC. This amounts to 260 µs used for the timer interrupt and 1750 µs for the processing of the next sequence of packets sent, resulting in a total of 2010 µs used. This sequence of actions should be performed over a time period of eight

frames, each frame is 1250 µs, giving a total of 10 000 µs available. This leaves an amount of 7990 µs for interrupt handling and other actions that may disturb the process.

Based on the results in the report the worst type of interrupt to service is the interrupt for receiving ACL data, that takes 330 µs to process, and the interrupt for transmitting ACL data, that takes 250 µs to process. There may be one of each of these interrupts during a frame. This result in a maximum of eight interrupts of each type during the pre-scheduling. Handling these interrupts take a total of 8 * (330 µs + 250 µs), that is 4640 µs used for the interrupts, leaving a total of 3350 µs to use for other applications during these eight frames. During the last frame, that writes the scheduled data to the available hardware registers, 260 µs is used for the timer interrupt activating the write function and another 260 µs is used for writing the data. The remaining 730 µs are left for handling other interrupts and necessary applications. The same worst case of receiving and transmitting ACL data applies to this situation and results in another 330 µs + 250 µs used for interrupts handling, this leaves 150 µs for other uses in this frame.

When the entire system use a common CPU, as in an embedded solution, it is necessary to add all other interrupts that may occur. Interrupts from the UART or USB for data transfers are not considered in this and the system only has a total of 3350 µs left to use for these interrupts during scheduling and 150 µs during the frame when writing the scheduled data to hardware. This results in a very small amount of time left in the final frame for anything else than writing the data which means that writing the data to the hardware has to have a very high priority so that it is not delayed. If the writing of the next sequence of scheduled frames are delayed so that it does not happen during the planned frame this will result in difficulties since the system will transmit erroneous packets.

6 eCos

6.1 Introduction

eCos (Embedded Configurable Operating System) was the first contestant to be evaluated for the position of replacing OSE Epsilon in the embedded Bluetooth software in this project. It has a number of advantages and characteristics. It is

- Free of charge for both development and commercial use
- Designed to be highly modular and easily configurable, making it possible to include only the parts that you need
- Compatible with many interface standards, such as µITRON, EL/IX, and of course ANSI-C
- Open source, developed and maintained by a large number of people around the world. This may be considered both an advantage and a disadvantage:

  - It has a key role in the pricing policy, people work for free
  - It is constantly and quickly updated if any problems arise

  But,

  - It may end up becoming inconsistent
  - It tends to be poorly documented

6.1.1 Configuration layout

To make configuration as easy as possible, a special configuration tool is provided. This tool presents the internal elements of the eCos real-time operating system in a graphical way too improve the feeling of control for the user. It is easy to find and understand the variables that need changes with this GUI (shown in Figure 5), but it also hides a lot of information from the user.

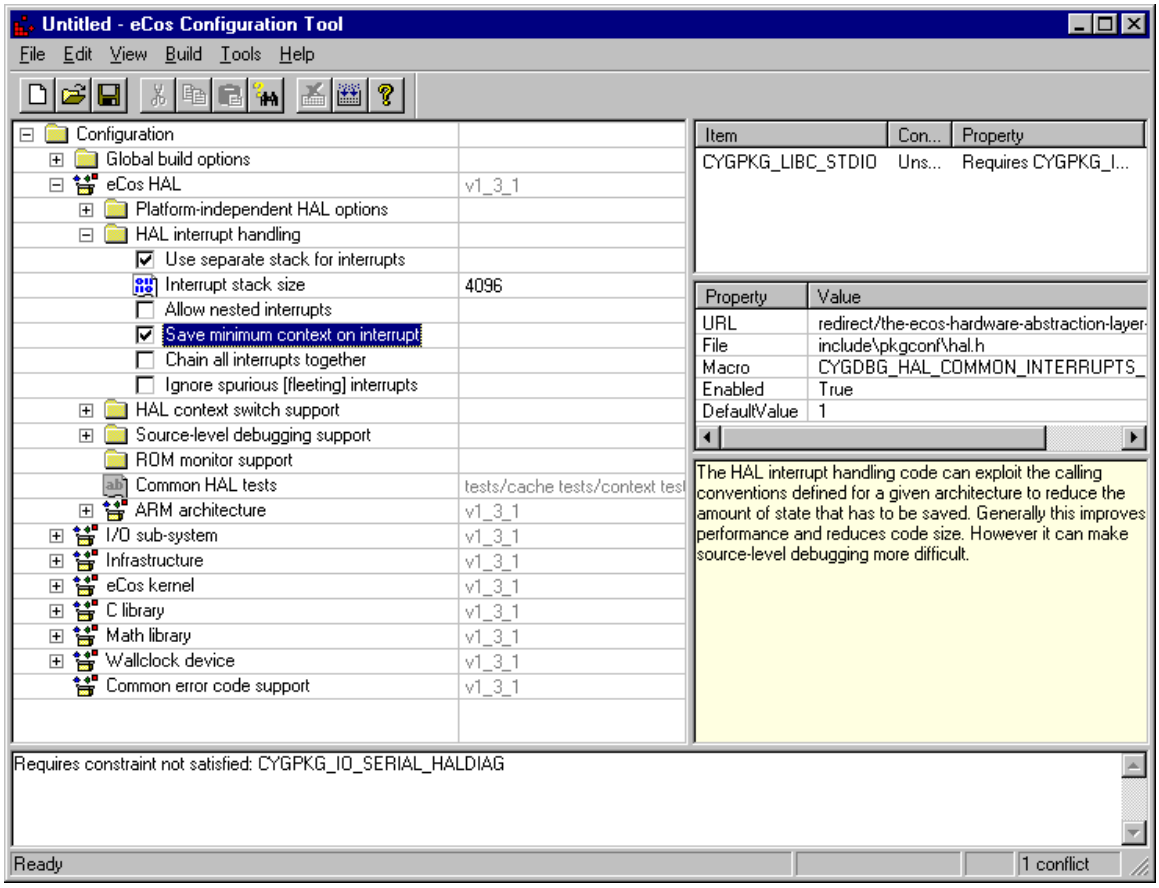

Figure 5 The configuration tool used by eCos.

Each configuration is based on a number of packages. Some base packages are required, depending on the target platform. The only platform dependent packages are the *Hardware Abstraction Layer* (HAL) packages. For convenience, packages are grouped together in pre-defined template setups. Once a platform and a template have been chosen, the configuration tool allows for modifications of defined parameters.

The packages are defined in a number of text files, written in a TCL-based script language called *Component Definition Language* (CDL).

6.2 Problems encountered

6.2.1 Compilation

Although eCos is supposed to be very open and portable, its ties with the GNU compiler and utilities remain strong. eCos relies on many non-ANSI features of the GNU C++ compiler, making compilation with ARM's tools difficult. A few of the language extensions that are provided by the GNU C++ compiler were added specifically to support eCos. The first of these extensions is *constructor priority ordering* which gives the possibility to decide in which order static objects are

created through the use of the `__attribute__` mechanism. This feature makes it possible to e.g. create a scheduler object before the system creates any thread objects. Another extension is *selective linking* that allows the user to only import specific sections and not an entire library when a part of the application needs to use an external function or variable.

This is a problem when working with the ARM Development Suite since it does not support these features that are necessary to compile eCos correctly. There are two different ways of handling this problem. The first solution is to restructure and rewrite the code in eCos so that the use of these special extensions is unnecessary. Even if it would be possible to modify the eCos code in this way, it would be out of scope for this project, and would make integration of new versions of the operating system difficult.

The second solution to this problem is that the ARM Development Suite allows import of compiled object files in various formats, including the ones preferred by the GNU tools; ELF and XCOFF. This makes it possible to compile the eCos code with the GNU compiler into object files, and by including header files for the Bluetooth core code, making it available in the ARM environment. The core code could then be compiled with ARM's compiler, and linked together using its linker into a complete flashable executable.

This solution has other problems that were encountered after a few short tests. As it turns out, even though object file formats are compatible, there are differences when it comes to the basic routines such as division and modulo, which are called differently in the GNU object files. It may be possible to remedy this by altering the files, but for now, it remains an unsolved puzzle.

It would most likely be possible to compile the Ericsson Bluetooth stack using the GNU tools, and thereby permitting eCos to be used as the operating system. But since all development is done using ARM's tools, and will continue to be done using them, this option is somewhat uninteresting from Ericsson's point of view.

6.2.2 Configuration

In its specifications, eCos defines clearly the difference between the hardware abstraction layer and the operating system modules. In reality, though, separating them is hard work. Since very little peripheral interaction is required in the application, most of the hardware abstraction layer is unnecessary.

7 µC/OS-II

7.1 Introduction

The second real-time operating system that was tested was µC/OS-II, version 2.04, from Micriµm Incorporated, owned by Jean J. Labrosse. This is not exactly a real-time operating system, it is more of a real-time kernel and therefore contains less unnecessary functions for the project.

The advantages with µC/OS-II are many but a short list of the most important ones are:

- *Available code.* All code for the real-time kernel is available and modifiable so that it works on the intended system. A lot of different ports are available from the homepage.

- *FAA (Federal Aviation Authority) certification.* The real-time kernel has been certified for use by safety critical systems in aviation and medical products. This proves that µC/OS-II is a very robust real-time operating system.

- *Small footprint.* Depending on the processor it is possible to reduce the footprint of µC/OS-II to around 2KB of code and 200 bytes of data, excluding the stacks.

- *Execution time.* Most functions in µC/OS-II have a constant and deterministic execution time. The execution time does not depend on the number of running processes.

- *Available functions.* Supports pre-emptive real-time scheduling and has a multitasking kernel. It also contains the functions for semaphores, messaging, task management, time management and fixed size memory management.

7.2 Problems

All systems contain a few problematic features that have to be handled and µC/OS-II was not an exception to this. A few of the limitations may result in problems later during development of the Bluetooth system. This chapter will give a short description of the areas where µC/OS-II differs from OSE Epsilon and what may cause problems that will affect the porting of the virtual operating system layer.

7.2.1 Priorities

There is a maximum of 63 tasks in µC/OS-II excluding the idle task. This limit is the result of how priorities are implemented in µC/OS-II. Another problem with the current implementation of priorities in µC/OS-II is that it does not allow multiple tasks to share a priority level. In the OSE Epsilon implementation of the system this feature of shared priorities levels is used by the different tasks. The reason for these limitations is that the priorities are implemented with a table with 64 places available and the lowest priority is used by the operating system for the idle task.

7.2.2 Messaging

In µC/OS-II, there is no record of which process sent a certain message. This is required by VOS, so it is necessary to add such a reference in the VOS layer itself. Consequently, this causes some raised complexity of the VOS layer.

7.2.3 Memory allocation

Memory handling is based on fixed size memory chunks so that the system has a certain amount of memory chunks of each size. This requires a good understanding of how many chunks the different tasks need and how large these chunks need to be. When the tasks free these allocated chunks it is important to return them to the correct memory handler. This requires that either VOS or the task remember from which memory handler each chunk is allocated.

7.2.4 Stack handling

In OSE Epsilon, the configuration file handles creation and initialisation of the necessary stacks. This is not the case in µC/OS-II, since this is a much more simplistic operating system to work with and therefore requires more work during initialisation. It is necessary to declare and initialise all stacks used by the system e.g. the main thread stack and interrupt stacks. To reduce the complexity of stack creation the initialisation for all the interrupt stacks are placed in the same file, `int_hdlr.s`, a part of the core system and the rest of the stacks are initialised by the macro `DECLARE_PROCESSES()` found in `config_macros.h`. This macro is used by the function `VOS_Init()` found in `vos.c`.

7.2.5 Interrupt handling

There is no interrupt handler delivered with the µC/OS-II source since its implementation will differ too much depending on which platform that is used. There is instead a second layer, a hardware abstraction layer, which contains the interface to the hardware, a so-called port. The available interrupt handler from Lee Dunbar's porting to the ARM7Thumb processor did not support nested interrupts, which resulted in a need to design and implement a new interrupt handler, to use with µC/OS-II, that can handle nested interrupts.

8 Porting VOS to µC/OS-II

8.1 Introduction

The porting of VOS consisted of making it rely on µC/OS-II rather than OSE Epsilon. During this porting, a couple of problems were encountered and solved. They are summarised in the following sections.

8.1.1 Code organisation

The current implementation of the system contains the VOS interface that should be the only part of the system that utilises the variables and functions in the real-time operating system that powers the system. Unfortunately this is not completely true as both `usb.c` and `core.c` contains references to specific OSE Epsilon commands and variables. In the new implementation that uses µC/OS-II instead of OSE Epsilon this has been changed so that `core.c` only uses variables and functions available in `vos.c` and `vos.h`. The problem with `usb.c` is that it uses semaphores and that there is no support in `vos.c` for semaphores and therefore either has to be added to the virtual operating system or rewritten so that `usb.c` uses the messaging system available in µC/OS-II. Therefore, USB support has been left out for the moment.

Function names that begin with OS, such as `OSStart()` or `OSMemCreate()` are always µC/OS-II functions. Functions that are a part of the Bluetooth system are named in the style of `CORE_Start()`, `VOS_Alloc()` and so on depending on from which module they are.

8.1.2 Memory Allocation

In µC/OS-II, the memory allocation is semi-dynamic, in a way that prevents fragmentation and assures a deterministic allocation time. A fixed memory area is defined, and a special "memory pool control block" handles how much of this memory area is used. The memory area is divided in a number of equally sized and indivisible memory chunks that are returned to the caller upon allocation requests to the controlling memory pool control block. The way the adaptation to the VOS allocation and de-allocation is done, a number of memory pool control blocks are created, each with a certain memory chunk size. Currently, there are eight such memory pool control blocks, set according to Table 1.

When returning an allocated memory chunk to the µC/OS-II memory pool control block (MPCB), it is necessary to provide a reference to the memory pool control block from which the chunk was retrieved. In order to make this work with the `VOS_Free()` function, which only takes a pointer to the allocated memory chunk,

the index number for the memory pool control block is placed in a hidden header, in front of the memory chunk, as shown in Figure 6.

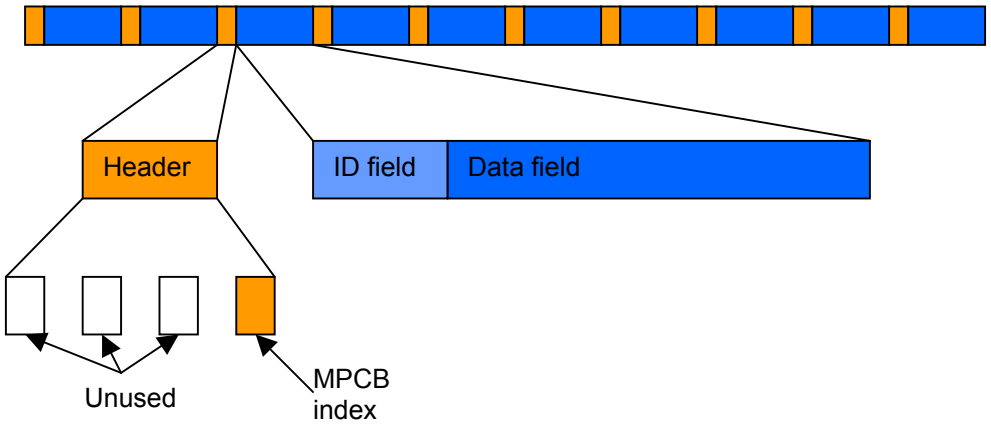

Figure 6. Description of a memory chunk.

	Number of chunks	Size of chunk + header	Memory usage
Block Handler 1	10	4 + 4	80
Block Handler 2	5	8 + 4	60
Block Handler 3	5	12 + 4	80
Block Handler 4	10	20 + 4	240
Block Handler 5	10	56 + 4	600
Block Handler 6	2	84 + 4	176
Block Handler 7	2	124 + 4	256
Block Handler 8	2	272 + 4	552
		Total amount of memory used	2044

Table 1. Specification of the available memory chunks.

Tests have been done to verify that this is indeed the memory distribution needed when running the stack with a reasonable use case. There are some extra chunks for each size, to ensure that the system works in somewhat more extreme cases. The test results are presented in 9.3.3.

The overhead of four bytes for each memory chunk is a bit on the high side but it is unfortunately necessary since the memory chunks have to stay word aligned. In the current implementation only two of the four bytes will ever be used. OSE Epsilon, on the other hand, uses 16 bytes, so four bytes is actually quite slim.

8.1.3 Configuration

In order to allow simple configuration of memory pool size distributions, stack sizes and process setup, VOS and core use a couple of header files, created with a small configuration program. This program reads the file `system.con` and creates two header files named `config_mempool.h` and `config_macros.h`, which are used by VOS to initialise the memory pool and the tasks, and an additional file called `config_stacks.s`. The last file `config_stacks.s` is used by `int_hdlr.s` to set the correct sizes for the interrupt stacks.

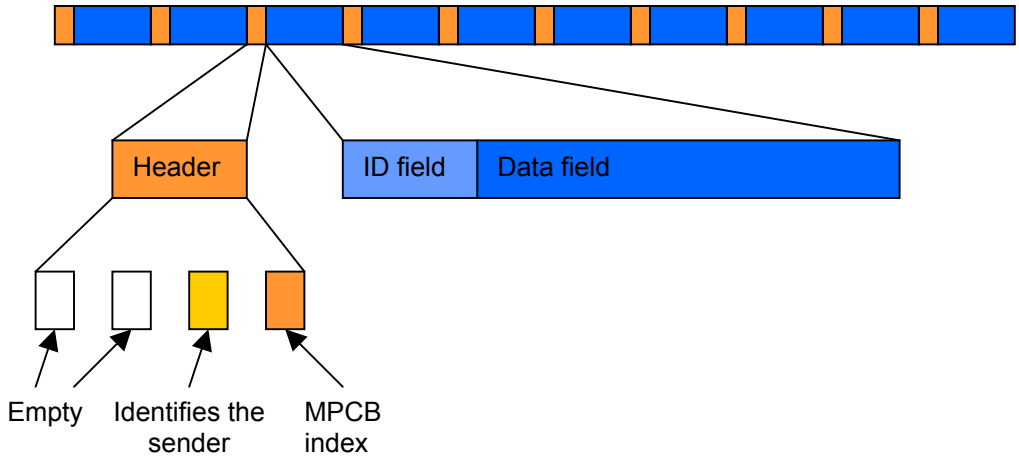

Figure 7. Description of a memory chunk, including the sender identifier.

8.1.4 Message handling

It is important for VOS to be able to identify from which process a message was sent. This presented a problem since there is no support from µC/OS-II when dealing with this problem. The problem is solved by using one of the bytes available in the header (there are three unused bytes left since only one byte is used for describing the MPCB) to store the priority of the sending task, see Figure 7. Since there may not exist more than one task at each priority level the priority can be used to identify a single task. With a maximum of 64 different priorities one byte is enough to store this information.

In OSE Epsilon, each task is associated with a message queue, which is reflected in the way `VOS_Send()` only takes the receiver task id as destination parameter. In µC/OS-II, there is no direct connection between tasks and message queues. Tasks can therefore receive messages from any queue, and any number of queues can be created. In the VOS adaptation though, an array consisting of pointers to the queue control blocks is created, linking each task with exactly one unique queue.

8.1.5 Synchronisation

There are neither semaphores nor mutexes in VOS. Synchronisation between tasks is created through signals sent between them and not through blocking access to shared resources. This solution removes the need for time slicing between tasks since they cannot be allowed to compete for resources. There is instead a form of co-operative multitasking based on signals.

The only exception to this is the interrupts. Interrupts can happen at any time and it is not possible to guarantee that a task is not using a shared resource at the time of an interrupt. It is therefore possible to disable all interrupts to prevent them from disturbing the system when it accesses shared resources.

8.1.6 Limitations

During implementation a few limitations in µC/OS-II were found. They were not serious and could easily be solved, in the worst case through some extra coding.

The following limitations were solved during implementation:

- During allocation of memory it is necessary for VOS to know which MPCB to access for receiving a memory chunk of the correct size.

- During deallocation of memory chunks it is necessary for VOS to know from which MPCB the chunk was allocated.

- It is not necessary for OSE Epsilon to know the exact number of memory chunks for each size since they are dynamically created during memory allocation. In µC/OS-II it is necessary to know how many memory chunks that are needed of the different sizes since it has to be set during start-up.

- The amount of space for signals has to be set before the system is started. Each task has a limited amount of space for received signals. The amount has been set by tests.

- There is no support for tasks sharing the same priority. The previous solution with OSE Epsilon allowed several tasks to share a priority but in the µC/OS-II implementation the priorities were rearranged so that a priority only is used by one task.

The following limitations are still left in the final system:

- There cannot be more than 63 tasks. This is due to the fact that the table that handles priorities only contains 64 positions and the lowest priority is reserved for the idle task.

- It is necessary to know the maximum amount of memory chunks of each size that the system uses at the same time. This information is used to create the configuration files and prevents crashes due to insufficient amount of memory chunks.

8.2 Modifications made to VOS

This section contains an in depth description of the changes that were made to the virtual operating system. It is divided into subsections describing the different services that VOS provides.

8.2.1 Memory allocation and deallocation

8.2.1.1 VOS_Alloc

This function receives a parameter to know how much memory space the calling process needs to allocate. The most important steps are shown in Code section 1. It returns a pointer to a memory chunk of the requested size, or larger if the most efficient size was not available, by making a call to the corresponding memory pool control block. Any errors that are encountered by the call to the µC/OS-II function `OSMemGet()` are handled by a call to `VOS_Error()`. The final action stores a reference in the hidden header describing from which memory pool control block the memory chunk was taken.

```
/* Choose memory block from the requested size */
if        (ulSize <= MEMPOOL_BLOCK0_SIZE &&
           aptMemPool[0]->OSMemNFree > 0) {
  memCtlBlockIndex = 0;
} else if (ulSize <= MEMPOOL_BLOCK1_SIZE &&
           aptMemPool[1]->OSMemNFree > 0) {
  memCtlBlockIndex = 1;

  ...

} else if (ulSize <= MEMPOOL_BLOCK7_SIZE &&
           aptMemPool[7]->OSMemNFree > 0) {
  memCtlBlockIndex = 7;
}

/* Allocate a memory chunk */
ptPointer = (void *) OSMemGet(aptMemPool[memCtlBlockIndex], &err);
```

```
  ... Error handling

/* Put a four byte padding before the allocated memory */
*((uint32 *) ptPointer) =
                    (uint32) ((0xFF << 8) | memCtlBlockIndex);

/* Move the pointer four bytes */
ptPointer = (void *) ((uint32 *) ptPointer + 1);

return ptPointer;
```

Code section 1. The code used for memory allocation.

The actual selection of memory pool control block is done in the macro `CHOOSE_MEMPOOL_INDEX()`, found in `config_mempool.h`, but is displayed in its pre-processed form for readability.

8.2.1.2 VOS_Free

This function receives a pointer to the used memory chunk and returns it to the pool of available memory. The code is shown in Code section 2. Information about which memory pool control block it should be returned to is stored in the header that is placed in front of the memory chunk. Without the information stored in this header it would not be possible to make a correct function call to the µC/OS-II function `OSMemPut()`.

```
/* Get a pointer to the padding */
ptPad = (uint32 *) (((uint32 *) *ppMem) - 1);

/* Get the memory block index from the padding */
blockIndex = (uint8) (*ptPad & 0xFF);

/* Free the allocated memory */
OSMemPut((OS_MEM *) aptMemPool[blockIndex], (void *) ptPad);
```

Code section 2. The code used for deallocation.

8.2.2 Time management

This is not used by VOS and therefore should not be implemented.

In µC/OS-II time is monitored by the use of time ticks. These ticks are used for the functionality of delaying a task a set amount of time. This functionality will not be used by the existing implementation of the system since it utilises the functionality of the timer component when the need for time management is encountered.

8.2.3 Process information

8.2.3.1 VOS_CurrentProcess

This function returns an identifier for the running task. The identifier is the same as the priority of the task since there cannot be more than one task for each priority. The only code, outside of `vos.c`, that utilises this function is located in `core.c` and it is used when handling logging and error information.

If the system is handling an interrupt and is currently inside an IRS when there is a call to the function it returns a value of 0xFF.

8.2.4 Message transfer

8.2.4.1 VOS_Send

This function has two parameters, a pointer to the message and a task identifier (the priority of the receiving task) describing the receiving task. The essential code of this function is presented in Code section 3. It stores the task identifier inside the hidden header placed in front of the memory chunk used for the signal. This identifier is also used for locating the correct message queue to place the signal in. After the message is posted in the message queue the function checks for any errors and if found makes a call to `VOS_Error()`.

```
if (OSIntNesting == 0) {
    /* Set sender thread id in the padding word (see figure in VOS_Alloc) */
    *((uint32 *) ((uint32) *ppMsg - 4)) &= (OSPrioCur << 8) | 0xFFFF00FF;
  }

  /* Will work since tProcess is the priority */
  pEvCtlBlk = aptECBMap[tProcess];
  if (pEvCtlBlk != NULL)
  {
    /* Send the message */
    err = OSQPost(pEvCtlBlk, *ppMsg);

    /* Handle any returned error codes*/
    if (err != OS_NO_ERR)
    {
      VOS_Error(err);
    }
  }
```

Code section 3. Sending a message.

8.2.4.2 VOS_Receive

This function searches for the oldest received message to the calling task. When a call is made to the function `VOS_ReceiveList()` it stores the ignored messages in a special list.

The message returned by this function could either be from the list of previously ignored messages or, if that list is empty, from the message queue connected to the task. If no message is found the function will wait forever for a message to arrive but such a situation cannot occur since this is a message driven system. As shown in Code section 4 the priority of the running process is used as an index to find the correct message queue to check.

```
/* Retrieve the id (priority) of the running process. */
tCurrentProcess = (VOS_TProcess)OSPrioCur;

…

/* Wait for a message, may change running process */
/* MAX_INT used for timeout and this may result in a long wait*/
ptTempMsg =
  (VOS_TMessageHeader*) OSQPend(aptECBMap[tCurrentProcess],
                                0, &err);
```

Code section 4. Receiving a message.

8.2.4.3 VOS_ReceiveList

This function searches for messages sent to the calling task that fulfils the parameters sent to this function. The parameters consist of a list of the message identifiers that are acceptable and the number of identifiers sent in the list. It searches through the previously ignored messages and if none of these match the list it starts checking the message queue connected to the task until it locates a correct message. Any message that does not fulfil the parameters is ignored and added to the list of ignored messages. The first message that matches one of the identifiers in the list is returned to the caller of this function. If the function cannot locate a matching message it will continue to wait until such a message is received.

8.2.4.4 VOS_Sender

This function returns the identifier of the sending task for the message that is sent as a parameter to the function. The identifier for the sender is located in the header and is the same as the priority of the sending task.

```
/* Get the full value of the word size padding */
pad = *((uint32 *) ((int32) pMsg - 1));

/* Return the second least significant byte. */
/* This is the sender priority */
return (VOS_TProcess) (pad >> 8) & 0x000000FF;
```

Code section 5. How to access information about the sender of the message.

8.2.5 Critical sections

8.2.5.1 VOS_EnterCritical

There is no support added in VOS for any advanced handling of resource allocation and deallocation with e.g. mutexes or semaphores. The system implemented is instead based on only one available critical resource. Taking this resource locks the system so that it does not send any interrupts to the interrupt handler. This is done through the use of the IRQ mask described in Appendix I.

8.2.5.2 VOS_ExitCritical

Reactivates the system so that it can once again reach the interrupt handler with interrupts from the system, restoring the IRQ mask as described in Appendix I.

8.2.6 Error handling

8.2.6.1 VOS_Error

This function receives errors from the entire operating system and relays them to the error handler implemented in core. The error structure that is sent to the error handler contains information about the error code, which process that created the error, and a reference to the stack pointer of the process. The error codes are listed in either `core_error.h` or `driver_error.h` depending on where they originate. Errors that originate from within µC/OS-II have their error codes listed in `ucos_ii.h` and they have a value between 0-130 depending on which function in µC/OS-II that created the error.

All errors sent to `CORE_ErrorHandler()` will be transferred to the `SIGLOG_ErrorHandler()` that sends the data about the error to the communication port used for testing the system. Both functions prevent the system from continue to run so that the system has to go through a hardware reset before it is fully functional again.

All the possible error codes returned by this function are presented in Appendix VI.

8.2.7 Start-up and activation

8.2.7.1 VOS_Init

This function initialises all the necessary components used by the different tasks. Each available priority has its message queue and message pool, for storing ignored messages, created. Both of these message-handling devices are then initialised. The memory pool control blocks and the tasks are also created and initialised with the macros found in the file `config_macros.h`.

Everything that was previously done in the OSE Epsilon for ARM file `osarmcon.s`, generated from the configuration file `osarm.con`, is now done in VOS. More specifically, it is done in `VOS_Init()`.

8.3 Other CORE modifications

8.3.1 Interrupt handling

The new system contains two different stacks used by the interrupt handler. The information stored in the registers when an interrupt occurs is stored in the *interrupt register stack*. During an interrupt it is necessary to have a stack that can be used by the interrupt to store temporary data and this is done on the *interrupt stack*. The reason for the two stacks is that the system supports nested interrupts[4] and has to switch mode for this to function and this result in problems with the stack pointer. The simplest way of handling the stack pointer is to save the registers in one stack and temporary variables on another stack.

The two stacks have different needs regarding size since the interrupt register stack needs a total of 280 bytes, 56 bytes for storing all the data from the registers and it need to be able to store five complete sets of registers. The reason for five sets of registers is that one set is from the interrupted task and four sets from the maximum times an interrupt can be suspended by an interrupt with a higher priority. The interrupt stack also need quite a lot of space and is not as easy to calculate since it needs to be able to store the maximum use of the stack for each interrupt priority level.

8.3.2 Boot sequence

The boot sequence, starting from the label `BOOT_Entry` has had some minor changes. Generally, it does the same thing as the OSE port:

---

[4] Nested interrupts means that interrupts can suspend other interrupts of a lower priority. Therefore allowing a nested structure where there may exist many different interrupts that need to run before the system can leave the interrupt phase and continue with the running task.

- *Platform dependent memory initialisation* by setting the wait states[5] and the type of memory used.

- *Code copying from flash to RAM*. If the code resides in flash, it needs to be copied to RAM in order to run.

- *Variable initialisation*. Some variables may have set initial values. These need to be copied to the right location in the RAM. All other variables are set to zero in the same fashion.

8.3.2.1 Stack initialisation

During the boot sequence, the memory area designated as the interrupt stack is used as stack. Once the tasks has started, after calling `OSStart()` in `os_core.c`, this stack is overwritten, since it is no longer needed.

Each task has its own stack, defined in C code upon creation. Task creation is done by `VOS_Init()` found in `vos.c` and the memory areas for the stacks are also declared there. The actual declarations are done with the help of a macro called `DECLARE_PROCESSES()`, from a generated h-file based on the desired number of tasks and stack sizes for each of them. Initialisation is done the same way with the macro `INIT_PROCESSES()`. The configuration of these macros is covered in Appendix V.

[5] A wait state is a delay of one or more clock cycles added to the processor's instruction execution time to allow it to communicate with slow external devices

9 Performance tests

9.1 Execution time

9.1.1 Introduction

In order to assess the performance of the µC/OS-II port, measurements of crucial parts of the system were made. How do the differences in implementation affect the time needed to perform certain tasks, and how does that in turn affect the fulfilment of the real-time demands?

9.1.2 Method

To measure the time needed for the different operations, a digital oscilloscope in conjunction with binary values to a couple of LEDs on the development board were used. Writing to a certain memory address sets these LEDs. The way this can be used to measure the time is as follows: When the region that is to be measured is entered, the value of an LED is set, and when the region is exited, the value is unset. The oscilloscope, with probes connected to the LEDs, can then measure the time passed quite accurately.

Using this method of measurement does however cause some additional time for setting and unsetting the values of the pins. In order to get an accurate value, this additional time must be measured separately and subtracted.

9.1.3 Results

9.1.3.1 Critical regions

These are basically the same implementation, since the original handling of critical regions in the ARM/THUMB port of µC/OS-II was replaced with the one used in the OSE port, in order to allow nested interrupts to work the way it is assumed in the system. See Appendix III and Appendix IV for more on this.

	µC/OS-II	**OSE Epsilon**
VOS_EnterCritical()	7.44 µs	8.84 µs
VOS_ExitCritical()	8.04 µs	12.24 µs

Table 2. Measured times for entering and exiting critical regions

The improvements in handling of critical regions in the µC/OS-II port are due to two minor modifications:

- The variable `uiVosCriticalCounter` (defined in `vos.c`) and the variable `IRQ_LOCK_COUNTER` were joined into one. They both have the same meaning, and by only increasing one counter it was possible to reduce the amount of code. The reduction in code has given a slight decrease in execution time.

- In the assembly function `arm_unlock()`, a lock was made before the unlock as a precaution. This lock was removed, since such precautions are unnecessary when the lock states have been verified by asserts.

9.1.3.2 Memory allocation

The allocation and de-allocation (freeing) of dynamic memory is also an important time consuming part of the operating system. It is something that is closely coupled with message passing, since the messages need to be dynamically declared.

	µC/OS-II	**OSE Epsilon**
VOS_Alloc(272)	53.4 µs	50.8 µs
VOS_Alloc(4)	48.2 µs	46.8 µs
VOS_Free(272)	28.4 µs	49.8 µs
VOS_Free(4)	28.4 µs	48.6 µs

Table 3. Measured times for allocating and de-allocating dynamic memory. The values in parentheses indicate the number of bytes requested

OSE is slightly faster when allocating, but significantly slower when de-allocating. This may be due to differences in the memory pool data structures, OSE may need to step through a list to put back a memory chunk.

The differences when allocating a large block in contrast to a small one are more or less equal. The time consumed by the µC/OS-II function `OSMemGet()` is in itself always constant, but the selection of suitable memory pool control block in VOS is not (see 8.1.2).

9.1.3.3 Message passing

The message passing is one of the central parts of the system, and a quick response time is very important for the overall performance.

The performance tests were made in the following way. First, the two VOS functions `VOS_Send()` and `VOS_Receive()` were measured isolated, so that these times could be compared to those in the more complex scenarios below. Isolated means that there is no task switch involved when sending, and no waiting done when receiving. This is done by first doing a `VOS_Send()` to the current process, and after that a `VOS_Receive()`. The sending does not cause a task switch, since the current task evidently is the highest priority task ready, and no one is ready to receive the message since it was sent to itself. The receive function does not wait, since there most certainly is a message waiting.

In order to include the time for the task switches, two scenarios were devised.

- *Sending a message to higher priority task* causes the sending task to be preempted, and the higher priority task to resume running, given that it was waiting for a message. The two tasks in question were `hci_commander` (priority 15 in the µC/OS-II port, priority 11 in the OSE Epsilon port) and `hci_transport` (priority 4 and 3).

- *Sending a message to a lower priority task* does not result in a task switch, even if the receiver task is waiting for a message. Eventually, the lower priority task will get to run, and handle the message. This will of course take a while, and that duration is included in the measured time. The two tasks in question in were `lm_connection` (priority 11 in the µC/OS-II port, priority 7 in the OSE port) and `lm_supervision` (priority 13 and 8).

Both scenarios were defined as the first occurrence of the situation after a connection to a slave has been made, in order to make the results comparable.

	µC/OS-II	OSE Epsilon
VOS_Send(), no switch	43.8 µs	111.8 µs
VOS_Receive(), no wait	35.8 µs	55.8 µs
Send+receive, to higher priority process	140.0 µs	180.0 µs
Send+receive, to lower priority process	705.0 µs	720.0 µs

Table 4. Measured times for message passing

The reason why the `VOS_Send()` takes so much more time in the OSE port than in the µC/OS-II port may be that they have different kinds of data structures handling the states of the tasks (waiting, ready, running, dormant). In µC/OS-II, task states are handled using the thread priority as identifier in a number of arrays and matrices, making the determination of which task is the highest ready and which tasks are waiting for messages in certain queues a quick and efficient operation. OSE may do this differently, but in turn allows several tasks to share the same priority, and thereby allows time sliced background tasks, which is not used.

9.1.3.4 Interrupt handling

Many time critical operations in a real-time system are initiated by an interrupt, either because of a timer or external events. Therefore, a quick response time is important when handling interrupts.

Also, it is important that the handling of an interrupt service routine does not take too long, since this will delay all other execution. This is, however, more a concern for the designer of the interrupt service routine than of the routines that handle them.

Only the preamble[6] of the interrupt handling is listed, since that is what causes most the interrupt latency, and therefore is of most interest. Comparing the interrupt handling that is done after the interrupt service routine has finished would be interesting too, but was left out because of its complexity and lack of time.

[6] The preamble is the part of the interrupt that saves information from the registers and prepares the system. It does not contain the time spent deciding which interrupt to start since this time is the same for both versions.

	µC/OS-II	OSE Epsilon
Preamble, non nested	14.2 µs	4.3 µs
Preamble, nested	9.4 µs	2.7 µs

Table 5. Measured times for preparing for a interrupt service routine to run

As seen in Table 5, the OSE port is considerably faster at handling interrupts. This is due to differences in the way the task information is stored at switches. The solution in the ARM/THUMB port of µC/OS-II uses the task stacks to store the saved registers and status. OSE Epsilon stores this in a special area allocated for each task, which makes switching simpler but requires a bit more memory. See Appendix II for details on task switching and stack handling in the µC/OS-II port.

9.1.4 Conclusion

All in all, the µC/OS-II port and the OSE port perform equally well. This is mostly because they are very similar in concept, at least in the parts used by the VOS.

The µC/OS-II port is in fact generally faster, though the lengthy preamble in the interrupt handling may be a cause for concern. Improvements can possibly be made by simplifying the access to the saved task registers and status in the ARM/THUMB port of µC/OS-II.

The implications of these differences do not seem to affect the ability to meet the real-time demands.

9.2 Pre-scheduling performance

The performance of pre-scheduling is crucial when it comes to fulfilling the real-time constraints of the system. Tests were therefore conducted to examine how the change of operating system affected the time consumed handling the pre-scheduling.

9.2.1 Method

The tests were conducted in the following way. Upon entry in the pre-scheduling function a LED is turned on, and upon exit it is turned off again. The time used by interrupts is also included in the measurement so the results show the real execution time for the pre-scheduler.

It is possible to see how the execution time differs by using the average function on the oscilloscope. It measures the latest 128 time periods where the LED was active and summarises this data into a diagram. See Figure 9 and Figure 10 for such images displaying the execution time distribution for OSE Epsilon and µC/OS-II respectively.

9.2.2 Results

The system was measured when it was transmitting DM1 packets over an ACL link, other types of data packets were tested but discarded since there was an obvious trend showing that DM1 packets placed a maximal strain on the system.

The images shown in Figure 9 and Figure 10 may need a short explanation. The horizontal axis represents the execution time for the pre-scheduling and the vertical axis displays the percentage of the executions that were running for as long as the horizontal axis indicates at that position. Both of the plots presented share the same scale for the vertical and horizontal axes to ease comparisons. The sudden edges in the diagrams shows that many executions took about the same time to complete, most probably due to a fix number of execution paths depending on the pre-scheduling needs. The slopes after the sudden edges are caused by delays of one sort or another, including interrupt handling.

Displayed in Figure 9 is the time distribution for pre-scheduling using OSE Epsilon. There are three important execution times that are shown in the diagram, the fastest execution time is 400 µs. This is true for approximately 18% of the times that pre-scheduling is performed. Almost a third of all executions finished after 500 µs and a total of 60% of the executions have finished after 1680 µs. The longest execution in this diagram took 2240 µs, but longer samples have been observed.

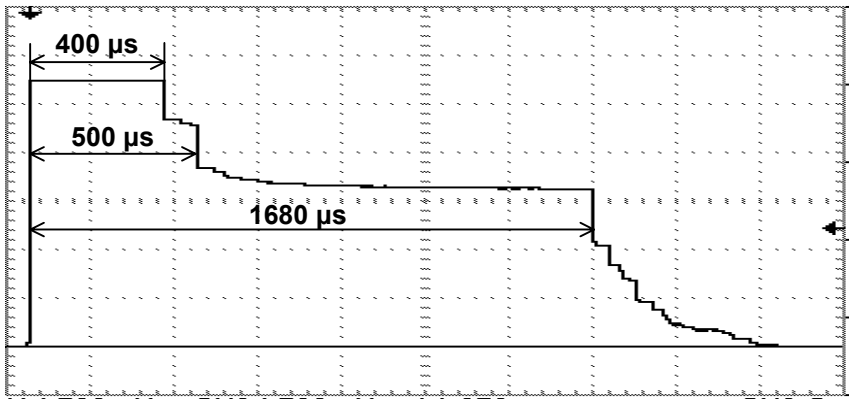

Figure 9. The distribution of time spent pre-scheduling when running OSE Epsilon during high-load data transfer.

Showed in Figure 10 are the results from using µC/OS-II when running the pre-scheduling. The similarities between the diagrams are remarkable and show almost the same pattern. The only significant difference between them is that there is a small delay for a small percentage of the samples at the 1680 µs time limit.

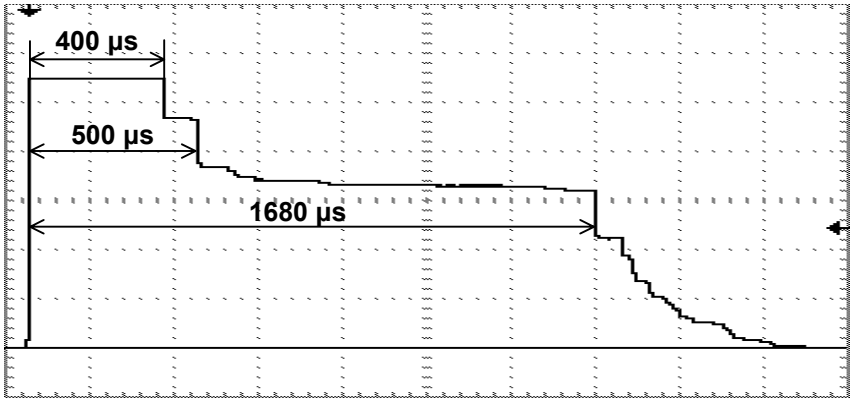

Figure 10. The distribution of time spent pre-scheduling when running µC/OS-II during high-load data transfer.

9.2.3 Conclusions

The scenario studied in both cases was chosen in order to press the system hard, pumping a continuous stream of data. However, it is not a worst case scenario since it was not possible to guarantee constant interrupts. However, it shows the performance during a high-load usage scenario.

As seen in the figures, there are no significant differences in the performance, the µC/OS-II port is insignificantly slower at times. The differences discussed in the previous section are apparently not affecting the overall performance.

9.3 Memory usage

9.3.1 Introduction

Another area of concern was the usage of available memory. It is important to keep the footprint of the operating system as small as possible. A sequence of tests was performed to see how many memory chunks of the different sizes that were necessary.

9.3.2 Method

The use of the memory chunks is measured by a small modification to the `VOS_Alloc()` and `VOS_Free()` functions. They use a small two-dimensional matrix to keep both the current number of allocated memory chunks, from all the available MPCBs, and the maximum number of chunks used for each MPCB.

The system then sends data between two Bluetooth units over an ACL link with different packet types. After the system has been transmitting data at the maximum speed for a short moment the values of the matrix are checked.

This test is not performed with the system running in flash which will affect the speed with which it can perform different actions but this should not affect the use of memory chunks since they are not time dependent.

9.3.3 Results

The memory usage was tested with both data medium rate packets (DM) that contain CRC for the data payload and data high rate packets (DH) that does not contain any CRC for the data payload. The number following the type of packet is the number of time slots a packet may cover. The values in Table 6 display the maximum number of memory chunks of different sizes that are used simultaneously by the system.

As shown in Table 2 there are not very large differences between the different types of packets. It is very interesting, though, to see that not a single memory chunk of with a size of 84 bytes is ever allocated during ACL data transfer.

The system's worst case is when DM1 packets transfer data, at that point it uses the maximum amount of memory chunks. The theoretical worst case for the amount of memory used is then 1112 bytes and the best case is for either DM5 or DH1 that both use in their worst case scenarios 988 bytes.

Available	Size	DM1	DM3	DM5	DH1	DH3	DH5
10	4	5	5	5	5	5	5
5	8	3	3	3	3	3	3
5	12	5	5	4	4	5	4
10	20	3	3	3	3	3	3
10	56	5	4	3	3	4	4
2	84	0	0	0	0	0	0
2	124	1	1	1	1	1	1
2	272	2	2	2	2	2	2

Table 6. Results of the memory usage test with different data packets.

9.3.4 Conclusion

Since the system will use a larger memory chunk if the most efficient size is not available are there no problems with the distribution of the smaller memory chunks. But there may be problems with the largest memory chunks since there may arise an unexpected need from the system to allocate a lot of these large chunks.

However, at the present there is no reason to be alarmed by the use of memory chunks in the system. Rather the opposite is true, there may be a possibility to reduce the amount of available memory chunks but there might be unforeseen situations that demand a higher memory usage so the overhead should be left untouched since it does not affect the system in a negative way.

10 Tools used

Here follows a short presentation of the tools that were used during the work of this porting of the BEP to µC/OS-II.

10.1 Trace32

Trace32 is a program developed by Lauterbach Datentechnik GmbH. The program, when used together with an in-circuit-debugger, allows the user to see how the code behaves on the test board. It is possible to look at the registers in the CPU as well as the code in the memory. This tool was very helpful during debugging and testing due to the insight it gave to the inner workings of the code.

The program supports special features and can retrieve internal statistical information from µC/OS-II, but only when used with the 68HC08 processor, and OSE Epsilon, when used on ARM7.

10.2 ARM Software Development Tool

The *ARM Software Development Tool* (SDT) is a program used to handle a software project. It deals with compilation, assembling and linking, as well as sets parameters organised in different *build variants*. For example, a build variant can be configured to include debug code or have different output formats, allowing execution in either Trace32 or directly on the board.

10.3 Config

The configuration program was written in C and compiled with GCC. It is a very simple program that translates a configuration file into several macros and save these in a couple of h-files, used at compilation. For further information see Appendix V.

11 Conclusion

11.1 The choice of operating system

The operating system chosen for the port, µC/OS-II, is conceptually very similar to OSE Epsilon, the operating system that it replaced.

In a way, this may have made the results of the thesis less significant. Replacing it with a fundamentally different operating system would have proved the level of flexibility of the stack and how it would have to be changed in order to function under this new environment. But then again, the thesis had to be finished within reasonable time.

Other interesting operating systems that were not selected are Nucleus and VxWorks. The main reason behind not selecting any of them was that they were introduced quite late in the project, but we still did a quick examination of how they relate to µC/OS-II.

11.2 The portability of the Ericsson Bluetooth stack

The version of the stack that was used when porting has proven to be well designed for portability.

However, one issue was discovered during the project. It concerns how the system prevents an involuntary rescheduling by blocking all the interrupts. This may result in problems with operating systems that contain support for running parallel processes. It presents no problem with the current system since it is designed for a host solution. Possible operating systems for porting are primarily those that are used by the system as a library of functions, incorporated into the executable, thereby allowing only the stack to run on the system. These types of operating systems are commonly known as micro-kernels.

If another type of operating system is selected this will modify the basis of the system in such a way that a redesign might be necessary to introduce time slicing so that the operating system's functions do not starve. This is outside the scope of this master thesis but might be considered for further analysis by Ericsson Technology Licensing.

11.3 VOS requirements and limitations

- *No resource synchronisation*. There is no explicit resource protection mechanism incorporated in VOS since there are no semaphores or mutexes. These mechanisms are not necessary if there are no situations where two tasks si-

multaneously wants to access a resource, but it requires a more careful system design.

- *No time slicing*. There is no support for automatic time slicing between processes in VOS. This complicates the selection of new operating systems because there should not be any need for simultaneous tasks. A solution to this is to use micro-kernels that can be incorporated into the system and that do not need running tasks.

- *Preemptive scheduling*. It is necessary to check each time a signal is sent if the receiving task is waiting for a signal and if the receiving task has a higher priority if so a task switch takes place. This is due to the fact that the system only can have one processes running at a time and the signal works like a token where the task holding the token may execute.

- *Minimal functional requirements*. There are very few operating systems that do not support the very basic functionality necessary for implementing VOS. It only needs the following functions:

  - Signalling service with queues. The tasks need some basic support for sending signals and a possibility to store several signals before they are processed.

  - Memory handling. The basic functionality of allocating and freeing memory is of course necessary.

  - Unique addressing of tasks. All tasks need a unique address so that they can be identified and selected.

  - Prioritised task scheduling. The tasks need priorities and functionality in the operating system to decide which task that should be allowed to run.

  - Interrupt handling for nested interrupts. Support for at least five levels of nested interrupts is also necessary to handle the current implementation.

11.4 Time requirements

The real time demand placed on the current system requires that all time limits be kept. This limits the possible operating system candidates to only hard real-time operating systems. As shown previously in Chapter 9 there are some minor differences between the two implementations but they both work without any problems.

The most time critical part of the system is the scheduling of how to handle the next couple of slots, i.e. when to send or receive data. We were not able to detect any differences between the system running OSE Epsilon and µC/OS-II when measuring these values. The differences in interrupt latency between the two operating systems (see 9.1.3.4) have had almost no effect on the system. The reason for this is that the µC/OS-II version of VOS saves a lot of time when handling signals. Sending and receiving signals is somewhat faster compared to the solution based on OSE Epsilon (see 9.1.3.3). The same is also true for allocating and freeing the memory used by the signals (see 9.1.3.2).

The conclusion when looking back at the test results is that there are no problems for VOS to handle the scheduling independently of whether it is OSE Epsilon or µC/OS-II that powers it. The behaviour of the execution time of the scheduling task has not given us any reasons for concern.

11.5 General implementation experiences

After having adapted the system to µC/OS-II, our main experiences regarding porting of the Ericsson Bluetooth stack have been these:

- The adaptation of VOS is fairly easy, once you are aware of its requirements and how these differ from the behaviour of the target operating system. Most operating systems have mechanisms similar to those in OSE Epsilon that are used by VOS, so at this level adaptation to most operating systems should be possible. The risk, though, is that too much of what should be part of the operating system is put in the VOS, making it less virtual than it ought to be.

- It is important to know the underlying limitations of the operating system. If some part requires modification, the adaptation may take much longer than anticipated, especially if your knowledge of the platform is limited. This was our problem when adapting the interrupt handler and context switch. In retrospect, the work needed to be done there would have discouraged us from using µC/OS-II if we had known it all from the start. Having to modify the operating system should not be a part of the porting.

12 Acknowledgements

First of all would we like to thank the personnel at Ericsson Technology Licensing at the Department of Research and Development – Software that gave us the opportunity and support for this master thesis. Especially the manager Leif Ekman for allowing us to take this project and Ivan Fulöp for the idea that was the base for this project. We would also like to thank our supervisor at Ericsson, Pär-Gunnar Hjälmdahl, for his never-ending patience when guiding us through the work.

Another important person for this master thesis that we would like to thank is Professor Karl-Erik Årzén at the Department of Automatic Control at Lund Institute of Technology for his support with the work and his opinions on how to improve the report.

13 References

1 *Specification of the Bluetooth System*, version 1.1, 2001, Bluetooth Special Interest Group

2 *MicroC/OS-II The Real-Time Kernel*, ISBN 0-87930-543-6, Jean J. Labrosse

3 *LM and HCI CPU Performance*, EN/LZT 108 5277/12 R1, Ericsson Technology Licensing (Confidential)

4 *The homepage of µC/OS-II,* http://www.micrium.com/

5 *ARM Architecture Reference Manual*, ISBN 0-13-736299-4, Dave Jaggar

6 *Functional Description: Bluetooth Embedded Platform, Core, OSE Epsilon 3.5.1 for ARM,* 155 16-cnh 202 05/2 Uen (Confidential)

14 Glossary

14.1 Abbreviations

ACL	Asynchronous Connection-Less
ANSI	American National Standards Institute
BEP	Bluetooth Embedded Platform
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
EBC	Ericsson Bluetooth Core
EBT	Ericsson Technology Licensing AB
ECB	Event Control Block
eCos	Embedded Configurable Operating System
FS	Frame Scheduler
GCC	GNU (GNU's Not Unix) C Compiler
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HCI	Host Control Interface
L2CAP	Logical Link Control and Adaption Protocol
LM	Link Manager
MPCB	Memory Pool Control Block
RAM	Random Access Memory

ROM	Read-Only Memory
RTOS	Real Time Operating System
RX	Receive
SCO	Synchronous Connection-Oriented
SIG	Special Interest Group
TX	Transmit
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VOS	Virtual Operating System
µITRON	Micro Industrial TRON (The Real-time Operating system Nucleus)

Appendix I The IRQ mask

This appendix describes how the problem with prioritised interrupts is solved with the help of a combination of hardware and software. The original architecture did not support prioritised interrupts because it only had one flag that could activate or deactivate interrupts.

In order to stop certain interrupts from coming, there is a hardware IRQ mask implemented. There is also the I flag, which is part of the Program Status Register (PSR), disabling all interrupts if set to one.

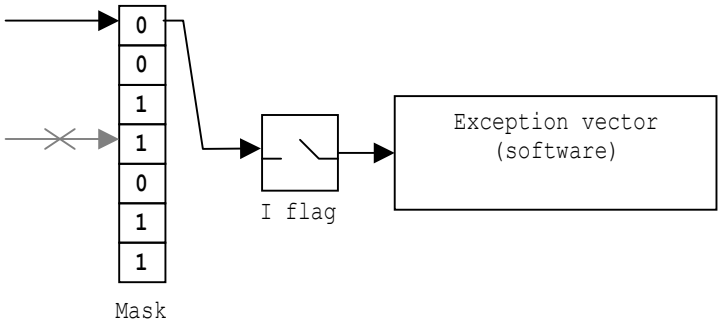

Figure 11. The passage of an IRQ

To retrieve the current mask, the content of rIRQ_MASK is read. The bit positions in the mask correspond to the different interrupts. Position zero, the least significant bit, corresponds to interrupt number zero, position one to interrupt one, and so on. A bit set to zero in a position means that the interrupt is passed on, a bit set to one means that it gets blocked, as shown in Figure 11.

To set bits in the mask, you write to `rIRQ_SETMASK`. Only the bits set to one have any effect on the mask, the bits already set to one in the mask are kept even if the new bit value in the data written to `rIRQ_SETMASK` is a zero. Theoretically, this correspond to `rIRQ_MASK |= rIRQ_SETMASK`.

To turn off bits in the mask, you write to `rIRQ_CLRMASK`. Only the bits set to one have any effect, though in this case, they turn off bits in the mask. This corresponds theoretically to `rIRQ_MASK &= ~rIRQ_CLRMASK`.

To allow a temporary change of the mask, normally to disallow all interrupts, the current mask is always stored in the variable `IRQ_SHADOW_REGISTER`.

Similarly, to simplify shutting off all interrupts, the variable `IRQ_SHADOW_MASK_REG` contains the value needed to pass to `rIRQ_SETMASK` in order to turn off all interrupts.

```
  1000010000001
| 1000
  1000010001001
```

Figure 12. An example of how to set the IRQ mask

Example: The mask is 0x1081, which is 1000010000001 binary. This means that all interrupts except number zero, number seven and number twelve pass through. If you want to disallow number three too, you write 0x8 to `rIRQ_SETMASK`, making the new mask become 1000010001001 (see Figure 12).

Appendix II General information about task and interrupt handling

One of the most important parts of an operating system is the handling of tasks and interrupts. They have many aspects in common, so they are best described together.

These things are part of the platform specific parts of µC/OS-II, which understandably is not covered very thoroughly in Labrosse's book [2]. This port is based on the µC/OS-II ARM-Thumb port done by Lee Dunbar [4].

1 Task switches

The task switching scheme in µC/OS-II is a co-operative one. The term co-operative refers to the fact that the tasks voluntarily enter a waiting state allowing another ready task to run. Therefore, µC/OS-II would not be a good choice of operating system if you were to implement a system that has several "competing" tasks. In the Bluetooth stack, however, all tasks work together for one common goal, and all task switching is based on passing of signals. This makes µC/OS-II a perfect candidate, being the slim and efficient operating system that it is.

Most task switches are caused by a call to `OSSched()`, which in the µC/OS-II VOS implementation is in turn always called from `OSQPost()` or `OSQPend()`. A detailed view on `OSSched()` can be found in Appendix III and Code section 7.

It seems that the way Labrosse describes a task switch does not correspond with the way the implementation of the task switch is done in the ARM port of µC/OS-II. Labrosse describes how the call to `OS_TASK_SW()` in `OSSched()` should only set a flag, and let the actual task switch be handled by a periodic ISR. In the ARM port though, the task switch is done directly in `OS_TASK_SW()`. The latter is probably more efficient, since it would switch directly, and not when the next periodic interrupt occurs.

2 Interrupt Service Routines

The handling of which ISR is to run, and the setting of the IRQ mask is done in the core module. ISR's are registered and unregistered with `IRQ_RegisterIsr()` and `IRQ_UnregisterIsr()` respectively.

The code of all ports contains sections that are called critical regions, this means that the entire region has to be finished before the task is allowed to be postponed by the system. This is implemented by deactivating the interrupts during

these regions. A postponed task is a task that is going to be switched, and a task switch that is not called by the running task is the result of an interrupt. Unfortunately, this complicates time slicing since it is the result of a periodic timer interrupt service routine that interrupts a running task to determine which task that is the most suited to run in that particular time slice.

However, time slicing is not at all part of the task model in µC/OS-II, and is not used by the Bluetooth stack either. The µC/OS-II task model does however support letting tasks sleep for a certain period of time (a certain number of ticks), but this is not implemented in our port, since it is not used in the Bluetooth stack and would only require excessive processing. Implementing this functionality would consist of assigning an ISR to a periodic interrupt (timer interrupt), and letting this ISR call the µC/OS-II function `OSTimeTick()`. Also, the constant `OS_TICKS_PER_SEC` in `os_cfg.h` would have to be adjusted to match the interval of the timer interrupt. The actual task switch would then be done after having finished the ISR, which is described in more detail below.

3 Idle task

There is an idle task defined by µC/OS-II, but it will never get to run, since there is another idle task in the Bluetooth stack (which does in fact handle important functionality). This hinders the µC/OS-II statistics functions to work properly, since they rely on the amount of time the µC/OS-II idle task gets to run.

Since the extra idle task defined in µC/OS-II has no other purpose than statistics it is not important for the system. The effects of that it will not be allowed to execute are marginal and will only affect measuring systems in µC/OS-II that are not used.

4 The stack

When a task stops running, either voluntarily (e.g. when waiting for a signal) or by an interrupt, its register values and program status are put on the task stack, as shown in Figure 13. For a more detailed description of the registers, see Appendix VII.

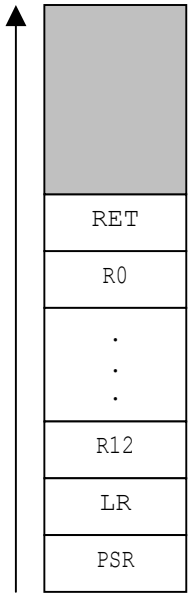

Figure 13. The stack setup used when saving processor registers and program status at a task switch.

The return address, marked RET in Figure 13, is the point where the task will continue running once it is its turn to resume running. The link register, marked LR in Figure 13, is the point where the task will return to after a call to a subroutine has finished. In the case of an ordinary task switch, i.e. a call to `OSSched()`, LR and RET will both be the same address, the instruction after the call to `OS_TASK_SW()`. In the case of a task having been interrupted by an interrupt, it is important to keep the LR of the task and at the same time know the return point, since the task may very well be in a position where it is about to use its LR.

5 Scenario: A normal task switch

A task sends a message to another task:

- `VOS_Send()` is called. It takes a pointer to the message to send, and the task id of the receiver task (which in µC/OS-II is equal to its priority).

- With the help of an array, containing pointers to the message queue control blocks of each process (indexed by the task id), a call to `OSQPost()` is made. `OSQPost()` needs to know to which queue the message is to be sent to, so it needs a reference to the corresponding message queue control block.

- It is determined in `OSQPost()` which task that is the highest ready. This situation may have changed if a higher priority task was waiting for a message from the queue to which a message was just sent.

- After having calculated which task that has the highest priority and is ready to run, a call to `OSSched()` is made.

- `OSSched()` is called, meaning that the task is in a position where it has changed the condition of which task is the highest ready. This is either a result of a task entering a waiting state by making a call to the function `VOS_Receive()` or `VOS_ReceiveList()`, or a task sending a message to a task with a higher priority.

- `OS_TASK_SW()` is called. It stores the values of the program status and the other registers on the stack of current task (see Figure 13). The value of the current stack pointer is put in the task control block, so that it can be retrieved once this task gets to run again. The variables `OSPrioHighRdy` and `OSTCBHighRdy`, calculated by `OSSched()`, now contain priority and pointer to the task control block of the new task to run. The new stack pointer is retrieved from the task control block that belongs to the task with the highest priority, and the program status and registers are popped from this stack. Finally, a jump to the address where the task was interrupted is done, upon which its old program status is also restored.

- There are at this point three possible situations:

  - The task that is started have been running earlier but reached a position where it made a call to `OSSched()`. The task will now continue to run from this same point.

  - It is the first time the task gets to run. It will then start running at the address of the function that is declared to be the main task function.

  - The task was interrupted by an interrupt. It will then continue running wherever it was interrupted.

6 Scenario: Interrupt handling

- Something, either hardware or software causes an interrupt to be generated. In the ARM architecture, an interrupt always leads to the following course of action:

  - IRQ mode is entered. This means that anything you do to LR, SP, or SPSR will not affect the interrupted mode. At start-up, the SP in IRQ mode is set to point to the correct memory area.

  - The address of the point where the program was running when the interrupt occurred is stored in LR.

- The program counter position is set to address 0x18. This is in the area known as the *exception vector*. On this address there is a jump instruction leading to `CORE_IrqHandler()`.

- In `CORE_IrqHandler()`, the first step is to determine if it is a nested interrupt. Looking at the value of the variable INT_LEVEL does this. At the same time, the value of this variable is increased.

- There is one major difference between handling a nested interrupt and a non-nested one: When not nested, the program status register and processor registers are stored on the stack of the interrupted task. This allows for an easy switch to take place after the interrupt has been handled. When nested, the processor registers and program status are instead stored on the IRQ mode stack. If they were to be stored on the task stack, each task stack would have to be big enough to have room for the registers of the worst-case nesting situation.

- After having stored the necessary information, the µC/OS-II variable `OSIntNesting` is increased with one. `OSIntNesting` is used in several places to tell whether an ISR is currently being serviced. It is essentially the same as `INT_LEVEL`, but to avoid having to change the µC/OS-II code, they were both left as they were.

- Next, a switch is done to SYS mode, in which the procedure `IRQ_ISR()` is called. In SYS mode, the ISR has its own stack, specially designated to be used by ISR handling only. In both the IRQ mode and the SYS mode, the I flag of the PSR is set to one, disabling interrupts. `IRQ_ISR()` will decide whether to unset the I flag when calling the assigned ISR procedure, depending on the priority of the IRQ.

- The ISR may have done some changes to the signal queue, i.e. done a `VOS_Send()`. For this reason, a check of the tasks signal queues are done right after the ISR has finished, by doing a call to `OSIntExit()`. `OSIntExit()` will update `OSPrioHighRdy` and `OSTCBHighRdy` to reflect this new situation.

- At this point, there are three possible branches of execution:

  - If the interrupt is nested, the processor registers and program status are simply retrieved from the IRQ stack, and execution is resumed inside the interrupted ISR.

  - If the interrupt was not nested, and the ISR did not cause any change in the task signal queues, the processor registers and program status are retrieved from the stack of the interrupted task, and execution is resumed where it was interrupted.

- If the interrupt was not nested, but the ISR caused a change in the task signalling state, the highest priority task ready will become the new current task. This means in fact just that the processor registers and program status are retrieved from the stack of this new current task, and that execution will resume wherever that task was running when it was interrupted. Also, the variables `OSPrioCur` and `OSTCBCur` will be updated to match the new current task.

Appendix III Implementation issues concerning task and interrupt handling

1 Nested interrupts

Dunbar's port does not allow nested interrupts, and is in many aspects therefore fundamentally different from the original OSE Epsilon solution. Most importantly, because of this, the critical sections use the I flag as interrupt prevention in Dunbar's solution. This has had to be changed, to avoid the following situation:

If a critical section were to be exited while in an ISR, it would allow new interrupts to occur, even though this may not have been what was intended.

The OSE Epsilon version of `IRQ_ISR()` is currently based on disabling the interrupts by setting the IRQ mask to disallow any interrupts. The current interrupt enabling mask is stored in software, as `IRQ_SHADOW_REGISTER`, which enables the function to restore the mask as it was before entering the critical region.

One way to implement Dunbar's solution is to use the µC/OS-II variable `OSIntNesting` that contains the current IRQ nesting level. A zero means that the system currently is not handling an interrupt. If one desperately would like to use the I flag as critical region lock, it would be necessary not to unset it while in an ISR that should not be interrupted.

The solution implemented in the µC/OS-II port does not use Dunbar's solution using the I flag it uses instead the OSE Epsilon solution with the IRQ mask. This has meant that some small changes, to the upper level µC/OS-II code, had to be made to the port Lee Dunbar has created for running on ARM/Thumb architecture.

2 Problems with `OSSched()`

There are no problems when using the `OSSched()` in conjunction with the I flag as a controller of the critical regions. The I flag is stored in the PSR and all tasks have a separate PSR, entering a critical region in one task will not affect another task. The original code, see Code section 6, performs an `OS_ENTER_CRITICAL()` when the task starts a task switch and ends with an `OS_EXIT_CRITICAL()` when the task is later switched in again.

However, when using the IRQ mask, whether to allow an IRQ or not is not a part of the program status and will affect all tasks. Therefore, if this scheme is used, no interrupts will be allowed between the points where `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()` are called. This solution will not work since it prevents interrupts to happen after a task switch unless the new task makes a call to `OS_EXIT_CRITICAL()` when it starts running.

```
void OSSched (void)
{
    extern void OS_TASK_SW(void);
    INT8U y;

    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {
        y             = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OS_TASK_SW();
        }
    }
    OS_EXIT_CRITICAL();
}
```

Code section 6. The original contents of `OSSched()`

To solve this, a call to `OS_EXIT_CRITICAL()` is done in `OS_TASK_SW()` if one is in a critical region, as determined by the value of `IRQ_LOCK_COUNTER`.

This way, the critical region is always exited before switching to a task. Also, worth adding, a task could not possibly be inside a critical region at the moment of task switch, since that is part of the definition of a critical region. The stretch of code between the call to `OS_EXIT_CRITICAL()` and the actual switch to the program counter position of the new task is protected from interrupts by having the I flag set. It will however be reset when switching to the new task, since its PSR is restored from the SPSR at the jump. SPSR is set when fetching the processor registers and program status from the task stack.

Consequently, `OSSched()` is altered so that `OS_EXIT_CRITICAL()` is only called if no task switch was performed, see Code section 7.

```
void OSSched (void)
{
    extern void OS_TASK_SW(void);
    INT8U y;
```

```
    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {
        y             = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OS_TASK_SW();
        } else {
            OS_EXIT_CRITICAL();
        }
    } else {
        OS_EXIT_CRITICAL();
    }
    OS_EXIT_CRITICAL();
}
```

Code section 7. The modified contents of `OSSched()`

Appendix IV How `CORE_IrqHandler()` works

The assembly function `CORE_IrqHandler()` handles IRQ's, and is called directly from the exception vector. It can be divided into three sections:

- Context saving of current task
- Call to ISR handler
- Context loading for the task about to continue running

The context saving and loading is described in Appendix II, in the section describing the stack.

In the first section, there are two possible branches of execution. The first one handles non-nested interrupts, i.e. interrupts that have occurred during normal task execution. The second one handles nested interrupts, where the execution of another ISR has been interrupted by an interrupt. Interrupts can interrupt one another according to their priorities, a higher priority ISR can interrupt a lower priority ISR.

In the third section, where the return to the interrupted context is done, there are three branches of execution. One handles the most common case, where task execution has been interrupted by an interrupt, and that same task is resumed. The other handles the case where the ISR that has just finished caused another task to be the highest priority task ready to run. This results in the task with the highest priority to be resumed. The third case is where an interrupt was interrupted by another interrupt that is resumed.

```
CORE_IrqHandler
; --------- INIT_INT_NESTED - BEGIN ------
        SUB     LR,LR,#4            ; LR-4 gives point of interruption
        STMFD   SP!,{R0-R3}         ; Push R0-R3, so we can use them

        LDR     R2,=OSIntNesting    ; OSIntNesting++
        LDRB    R3,[R2]
        ADD     R3,R3,#1
        STRB    R3,[R2]

        CMP     R3,#1
        BGT     CORE_IrqHandler_L1

        ; OSIntNesting == 1
        MRS     R3,CPSR
        MRS     R1,SPSR
        ORR     R1,R1,#0x80         ; No IRQs
```

```
        BIC     R1,R1,#0x20          ; ARM mode
        MSR     CPSR_cxsf,R1
        MOV     R2,LR
        MOV     R0,SP
        MSR     CPSR_cxsf,R3

        STMFD   R0!,{LR}             ; Put return address on process stack
        STMFD   R0!,{R2}             ; Put old mode LR on process stack
        MOV     LR,R0                ; Use LR instead of R0 as i-stack pointer
        LDMFD   SP!,{R0-R3}          ; Restore R0-R3

        STMFD   LR!,{R0-R12}         ; Save old mode registers
        MRS     R0,SPSR              ; Save old state
        STMFD   LR!,{R0}
        LDR     R0,=OSTCBCur         ; Update process stack pointer
        LDR     R0,[R0]
        STR     LR,[R0]

        B CORE_IrqHandler_L2
```

Code section 8. Describing the first part of the `CORE_IrqHandler()`.

The tricky thing here, as seen in the first conditional part of `CORE_IrqHandler()`, see Code section 8, is how to get hold of the stack pointer of the current task. `CORE_IrqHandler()` is called when an interrupt occurs, and is automatically in IRQ mode, with the address of the interruption point in LR, and the program status (CPSR) at that time in SPSR. Since LR and SP are banked, there is no direct way of getting hold of a SP or LR in another mode. The ^ operator can only retrieve USR/SYS registers, and in this setup, only SVC mode gets interrupted at the lowest nesting level. The solution is a quick switch to the mode that was interrupted, store its LR and SP on R0 and R2, and return to IRQ mode. Once the stack pointer is retrieved, the registers can be placed on the task stack.

```
CORE_IrqHandler_L1
        ; INT_LEVEL > 1
        LDMFD   SP!,{R0-R3}          ; Restore R0-R3
        STMFD   SP!,{LR}             ; Save return address
        SUB     SP,SP,#(14*4)
        STMIA   SP,{R0-R12,LR}^      ; Save old mode registers
        MRS     R0,SPSR              ; Save old state
        STMFD   SP!,{R0}

CORE_IrqHandler_L2
        MSR     cpsr_c,#0x9F         ; enter SYS mode, no IRQs
```

Code section 9. Storing the registers due to a nested interrupt.

When handling a nested interrupt, see Code section 9, the same data needs to be stored as when handling a non-nested interrupt. Since the ISRs always run in SYS mode, they are accessible through the ^ operator, and putting them on the

stack is very simple. The stack used, in this case, is the IRQ mode stack. The IRQ stack is only used to store these register and program status values (apart from the temporary use in the beginning of the interrupt handling routine).

Once these values are stored, a switch to SYS mode is done, and the ISR gets to use the SYS mode stack. All ISRs share the same stack, but since they use it in a "linear" fashion, that is not a problem. By linear it is meant that each ISR that use the stack, i.e. push data onto it, will pop all data from the stack before it ends. This guarantee that the position of the stack pointer will be the same when the ISR is finished as it was when it started. So even if an ISR get interrupted by another interrupt, when it gets turn to run again, the stack is just as it was when the ISR was interrupted.

```
;--------- INIT_INT_NESTED - END ------

        BL      IRQ_ISR             ; Call the ISR handler (c code)

        BL      OSIntExit           ; Notify uC/OS
;--------- QUIT_INT_NESTED - BEGIN ------

        MSR     cpsr_c,#0x92        ; IRQ mode, no IRQs
```

Code section 10. The call to `IRQ_ISR()` and notifications sent to µC/OS-II.

```
        LDR     R0,=OSIntNesting
        LDRB    R1,[R0]
        CMP     R1,#0
        BNE     CORE_IrqHandler_L3

        ; OSIntNesting == 0

        LDR     R0,=OSCtxSw
        LDRB    R1,[R0]
        CMP     R1,#0
        BEQ     CORE_IrqHandler_L4

        ; OSCtxSw == 1
        MOV     R1,#0
        STRB    R1,[R0]
```

Code section 11. Check for nested interrupts and if a context switch is necessary.

After the ISR has been handled a check is made to decide which of the three execution branches that is to be followed. The first check is made to the variable `OSIntNesting` after it has been decreased by one. There are nested interrupts if

the variable `OSIntNesting` does not equal zero and a branch to Code section 13 is done, where the system restores the previous interrupt.

If there are no nested interrupts a check is made to see if something changed during the ISR, concerning which ready task that has the highest priority. There is a branch to Code section 14, where the old task is restored, if the variable `OSCtxSw` is set to zero otherwise the system continues with Code section 12 and replaces the running task with the new higher prioritised task.

```
; Swap in next process
MSR     cpsr_c,#0x93         ; SVC mode, no IRQs

; Change the id of the current task
; and save previous id
LDR     R4,=OSPrioCur
LDR     R6,=OSPrioPrev
LDRB    R5,[R4]
STRB    R5,[R6]
LDR     R5,=OSPrioHighRdy
LDRB    R5,[R5]
STRB    R5,[R4]

; Get highest priority task TCB address
LDR     R4,=OSTCBCur
LDR     R6,=OSTCBHighRdy
LDR     R6,[R6]
LDR     SP,[R6]              ; get new task's stack pointer

; OSTCBCur = OSTCBHighRdy
STR     R6,[R4]              ; set new current task TCB address

LDMFD   SP!,{R4}
MSR     SPSR_cxsf,R4

MOV     R0,SP
ADD     SP,SP,#(15*4)

LDMFD   R0,{R0-R12,LR,PC}^   ; Does not change mode
NOP
```

Code section 12. Switching in a new task after the interrupt.

Why were not the registers just put the on the IRQ mode stack? The reason is that putting the interrupted registers and program status directly on the task stack allows for a quick task switch after having serviced the interrupt. This is required if the ISR caused a change in which task wants to run. When inside the ISR, the task switch is not done directly, where it would normally be, but is delayed until after the ISR has finished, see Code section 12. Then, the id of the task that wants to run can be found in `OSPrioHighRdy`, and the switch is just a matter of loading the stored registers and program status from the stack of the new task. Execution will resume wherever that task was running when it was interrupted.

Also, the variables `OSPrioCur` and `OSTCBCur` will be updated to match the new current task.

```
CORE_IrqHandler_L3
        ; OSIntNesting > 0
        LDMFD   SP!,{R4}             ; Get ISR mode from interrupt stack
        MSR     SPSR_cxsf,R4         ; Move this mode to SPSR

        LDMFD   SP,{R0-R12,LR}^      ; Load registers from interrupt stack
                                     ; to SYS registers
        NOP
        ADD     SP,SP,#(14*4)
        LDMFD   SP!,{PC}^            ; Load PC from interrupt stack to SYS
                                     ; register and switch SPSR to CPSR
        NOP
```

Code section 13. Returning to a previous interrupt.

When dealing with nested interrupts the system executes the code presented in Code section 13. The first action is to restore the previous program status register from the stack and store it in the SPSR. Followed by loading the SYS mode registers with their stored values from the interrupt stack through the use of the ^ operator, this operator allows the system to write directly to the SYS registers. The final actions before the switch are setting the stack pointer to a correct value and load the previous program counter with the use of the ^ operator resulting in that the CPSR is replaced with SPSR. The interrupted ISR can now continue to execute.

```
CORE_IrqHandler_L4
        ; OSIntNesting == 0 && OSCtxSw == 0
        LDR     R0,=OSTCBCur         ; Get a ref to the stack pointer
        LDR     R0,[R0]              ; Read the address of the SP
        LDR     R1,[R0]              ; Read the SP
        ADD     R2,R1,#(16*4)        ; Update process stack pointer
        STR     R2,[R0]              ; Store the new SP on the process stack

        LDMFD   R1!,{R4}             ; Load process mode from top of stack
        ORR     R5,R4,#0x80          ; No IRQs
        BIC     R5,R5,#0x20          ; ARM mode
        MSR     CPSR_cxsf,R5         ; Switch to process mode without IRQs
        MSR     SPSR_cxsf,R4         ; Move process mode to SPSR

        LDMFD   R1,{R0-R12,LR,PC}^   ; Load all registers from process stack
                                     ; and switch SPSR to CPSR
        NOP
;--------- QUIT_INT_NESTED - END ------
```

Code section 14. Switching in the previous task.

The final execution path when exiting an interrupt is to continue running the previous task that was running before the interrupt occurred. The code used to restart the previous task is presented in Code section 14. The task's registers are saved on the task's own stack and the stack pointer is accessed through the task control block.

Appendix V Configuring the µC/OS-II VOS port

In order to simplify configuring the parts of the system that handle memory allocation, task declaration and stack sizes, a special configuration program was made. This program, simply called `config`, takes a configuration file, and outputs three files that are included in the µC/OS-II VOS port.

The first file, `config_mempool.h`, contains header information about the number of memory pools, and their respective sizes, `config_macros.h` and `core_handlers.c` use it. In `core_handlers.c`, this information is needed for debug output of the status of the memory pool.

The second file, `config_macros.h`, contains five macros used exclusively in `vos.c`, named `DECLARE_MEMPOOL()`, `DECLARE_PROCESSES()`, `INIT_MEMPOOL()`, `INIT_PROCESSES()` and `CHOOSE_MEMPOOL_INDEX()`. They handle the declaration and initialisation of the tasks and the memory pool control blocks. The last macro, `CHOOSE_MEMPOOL_INDEX()`, is used to choose the appropriate memory pool control block based on a requested memory chunk size. This way of using macros may not be considered good programming style, but it makes it possible to make changes to the configuration easily and in one place. The number of tasks and memory pool control blocks is flexible, which would have been difficult to attain if it were to be hard-coded in `vos.c`.

The third file, `config_stack.s`, is included by `int_hdlr.s` in the declaration of the stacks used by the interrupt handling routines.

The configuration file, `system.con`, presently looks like this (with the exception of task function names, which have been replaced by generic names):

```
#################################################################
#
#  Memory pool
#  ===========
#
#  Max nbr: 256
#  Order is unimportant.
#
#       Chunk size  Block count
#       (bytes)
#-------------------------------
mem     4           10
mem     8           5
mem     12          5
mem     20          10
```

```
mem     56          10
mem     84          2
mem     124         2
mem     272         2


##################################################################
#
#  Processes
#  =========
#
#  Max nbr: 63
#  Order is unimportant, though prio must be unique.
#
#       Name of process func    Prio    Stack size  Queue size
#                               (0-62)  (bytes)     (elements)
#------------------------------------------------------------------
proc    task_1                  15      1600        200
proc    task_2                  4       400         80
proc    task_3                  3       460         20
proc    task_4                  5       300         20
proc    task_5                  13      400         20
proc    task_6                  11      600         20
proc    task_7                  10      800         20
proc    task_8                  9       500         150
proc    task_9                  8       500         20
proc    task_10                 7       600         20
proc    task_11                 2       500         20
proc    task_12                 0       400         20
proc    task_13                 6       340         20
proc    task_14                 12      240         20
proc    task_15                 14      280         250
proc    task_16                 16      1160        30
proc    CORE_IdleHandler        17      200         1


##################################################################
#
#  Static stacks
#  =============
#
#  These are the stacks used in the interrupt handling.
#
#       Mode        Stack size
#       (irq/sys)   (bytes)
#------------------------------
stk     irq         280
stk     sys         1500
```

Appendix VI Error codes from µC/OS-II

1 Introduction

When the old real-time kernel was removed from the system a lot of the old error messages were replaced with new messages that cannot be interpreted by the existing version of HCI Toolbox. We have assembled all the new error messages that can be sent from the real-time kernel in this appendix. This appendix contains information gathered from the chapters four, six and seven in [2].

2 µC/OS-II functions

Since this porting does not use all the available µC/OS-II functions is it not necessary to describe all the different error codes that are available in µC/OS-II, this document has focused on the error codes that can be expected from the functions used in the port. The error messages are described by their defined name, as presented in `ucos_ii.h`, and their respective numerical value with a short description of what the reasons are for this error.

2.1 `OSMemCreate()`

All the memory chunks are created and added to their respective memory pool control blocks (MPCB) depending of their size during the start-up of µC/OS-II. This function is performed with the `OSMemCreate()` function in µC/OS-II. The MPCB that is created for each size of memory chunks will later be used too handle all actions that are performed upon the memory chunks. But it is possible to make flawed calls to this function by giving it parameters that cannot be used and these calls may result in some of the following responses.

`OS_MEM_INVALID_BLKS` (111) - This error signal is sent if there is less than two memory chunks declared for use by the MPCB. A MPCB must contain at least two memory chunks otherwise is it only a variable.

`OS_MEM_INVALID_SIZE` (112) - The size of a memory chunk has to be able to contain a memory pointer when it is not used to contain data. These memory pointers are used to keep track of the empty chunks, they will point to the next empty chunk in the MPCB. The size of a memory pointer is four bytes in the existing system.

`OS_MEM_INVALID_PART` (110) – This signal is sent if there are no free memory pool control blocks left to control the memory chunks that are being created. There has to be a memory pool control block for each size of the memory chunks.

2.2 OSMemGet()

The function VOS_Alloc() uses this function to get a pointer to a memory chunk. The function needs to know from which MPCB to get a memory chunk, but that is handled internally in VOS_Alloc() and will not demand anything from the user of VOS. There has been a small change to the original behaviour of VOS_Alloc() so that if a certain size of memory chunks are depleted the next, larger, MPCB is called to get a memory chunk.

OS_MEM_NO_FREE_BLKS (113) – This signal informs the user that there are no available chunks left in any MPCB that handles memory chunks of the wanted size or larger.

2.3 OSMemPut()

When a memory chunk is no longer used it should be returned to the pool of free memory chunks in its MPCB and that is done by VOS_Free() using the µC/OS-II function OSMemPut().

OS_MEM_FULL (114) - This error message happens if someone tries to free more memory chunks to a MPCB than has been allocated for it by OSMemGet(). The most common cause for this signal is if all memory chunks of this MPCB has already been deallocated.

2.4 OSQCreate()

This function is used by the macros created when initialising the system during start-up, in VOS_Init(). But this function does not return any error messages neither to the user nor to the system.

2.5 OSQPend()

This function is used in both VOS_Receive() and VOS_ReceiveList(). The call to this function contains a maximum time that the function can wait for a response and information about from which queue the message should be retrieved.

OS_ERR_EVENT_TYPE (1) - The call has not been made to a message queue. It has been sent to some other type of event handler. This error should not be able to occur since there are no other types of event handlers implemented in this porting.

OS_ERR_PEND_ISR (2) - If an ISR tries to retrieve a message and there is no message for it to get it will result in this error since an ISR should not stop and wait for other actions in the system.

OS_TIMEOUT (10) – This signal is sent if there did not come a message within the given time frame specified in the call of the function. This error code should never be encountered since the timeout functionality is disabled.

2.6 OSQPost()

When a task wants to send a signal to another task the function VOS_Send() is used. This function uses the OSQPost() function from µC/OS-II with extra information, that is retrieved and declared by VOS_Send(), about which message queue that is used by a given task.

OS_ERR_EVENT_TYPE (1) - This error is the result of trying to place it in an event handler that is not a message queue. This error code should not be able to occur since there are no other type of event handlers implemented in the existing porting.

OS_Q_FULL (30) – This signal is used if there is no room for more messages in the recipient's queue. The maximum amount of messages in the queue to a task is set in the configuration file that is used to create the DECLARE_PROCESS() macro.

2.7 OSTaskCreate()

This function is used by VOS_Init() during the initiation of the system. It is used by the macro INIT_PROCESSES(). The following errors can be the result of flawed parameters sent to this function, but a user should never encounter these errors since they are neither handled nor presented by the macro.

OS_PRIO_INVALID (42) - This error code is returned if the priority of the task is lower than that of the lowest eligible priority. The lowest priority for any given task is set in the file os_cfg.h with the variable OS_LOWEST_PRIO.

OS_PRIO_EXIST (40) - This error signal informs the users that the used priority has already been selected for another task.

OS_NO_MORE_TCB (70) - This error code is returned to OSTaskCreate() by the function OSTCBInit() if there are no empty task control blocks left to use for a task initialisation.

Appendix VII The ARM architecture

1 Introduction

This appendix will summarise the information presented in the first three chapters in [5]. This information facilitates understanding of the problems encountered when dealing with interrupts and task switches, and how certain architectural issues affect the way the system is implemented.

2 Registers

R0-R12 The R0 to R12 are general purpose registers that are intended for just about anything. Certain call conventions apply to the use of these registers when dealing with compiled code though.

Link Register (LR) The link register stores the return address in a branch-with-link call (BL), allowing execution of procedures.

Stack Pointer (SP) The stack pointer is intended to hold the address of the current stack top or bottom, depending on stack growth direction. The THUMB calls PUSH and POP use the SP implicitly, other instructions need to have it specified explicitly.

Program Counter (PC) The program counter contains the current execution address. Jumps can be made by changing its value.

Current Program Status Register (CPSR) The current program status register contains the program status information. See section 4 below for more information.

Saved Program Status Register (SPSR) The saved program status register contains the last program status information, used to facilitate switching modes or flags.

3 ARM and THUMB execution

There are two subsets of the ARM architecture instruction set. The so called ARM instructions use the full 32-bit values while THUMB instructions only use 16-bit values.

The advantages of using only 16-bit instructions are evident when working with a 16-bit data bus, since it only needs one data transfer to move an entire command and not two transfers. It also reduces the amount of memory used since commands only take half as much space when reduced to 16-bit values. The THUMB instruction set is however a subset of the full set of ARM instructions and is not as powerful.

Another difference between the two different instruction sets is the fact that the THUMB instruction set only has access to the eight lowest hardware registers, R0 to R7. Another limitation to the thumb instructions is that they cannot directly modify or access the current program status register or the saved program status register.

A switch from ARM execution mode to THUMB execution mode is made with the BX command. If the least significant bit in the branch target address is set to one, a switch will be performed, and the system will assume all following instructions to be THUMB instructions. If the least bit in the branch target address it is set to zero there will be no change and the system will continue to interpret the instructions as 32-bit instructions.

A switch in the other direction, from THUMB to ARM, happens when an exception occurs before the system starts to execute the exception handler. Information about from which mode the jump to the exception handler came is stored in the saved program status register. This information is saved so that when the return is performed the instructions will be interpreted correctly.

4 Program status register

There are two different hardware registers called current program status register and saved program status register. These registers contain the following fields:

Flags field – this 8-bit field contains the four flags N, Z, C and V (Negative, Zero, Carry and oVerflow). These flags are changed by logical and arithmetic operations and can later be tested to see if a following instruction should be performed.

Status field – this field is not used in this version of the ARM architecture.

Extension field – this field is not used in this version of the ARM architecture.

Control field – this 8-bit field contains three control bits that only can be changed in privileged mode. These bits can disable interrupts and indicates what type of instructions that the system is running. There are also five bits used for determine which mode that the processor operates in. The effects and differences between these modes will be described later.

When the processor switches mode the old program status register is saved in SPSR.

There are three different reasons for modifying the CPSR, setting the conditional flags to a known value, enable or disable interrupts and changing processor mode.

5 Processor modes

The ARM architecture contains seven different processor modes. All but the user mode are so called privileged modes, where the possibility to change mode or processor state exists. The intention of the architecture design is that normal program execution should use the user mode, and operating system functions should use the privileged modes.

All modes have their own LR (Link Register), SP (Stack Pointer), CPSR (Current Program Status Register) and SPSR (Saved Program Status Register). The exception is user mode and system mode, who share all registers. Also, there is no SPSR in USR mode, since mode changes are not allowed.

5.1 User mode (USR)

This is the default mode, which does not allow changing the mode nor processor state.

In the current system, user mode is not used. The mode used for normal execution is instead supervisor mode (SVC).

5.2 Interrupt request mode (IRQ)

When an interrupt occurs, address 0x18 is called, and IRQ mode is automatically entered. From 0x18, a call to the interrupt handler is done.

In IRQ mode, and all modes that can occur as a result of an exception, the program counter at the point of interruption is put in the LR of the new mode, and the program status is put in SPSR of the new mode. This way, both the position and the state of the execution can be resumed once the exception has been handled.

Interrupts can be disabled using the I flag in the program status register, or by setting the IRQ mask. See for Appendix I more on this subject.

5.3 Fast interrupt request mode (FIQ)

FIQ mode is a high priority interrupt request mode, intended for small and independent functions. FIQ mode has a separate set of register for R8 to R12. This makes it possible to do fast processing without having to save registers on the stack. It is also possible to save values between interrupt calls without being forced to save it in memory and later restore the data.

Fast interrupt requests can be disabled using the F flag in the program status register.

5.4 Supervisor mode (SVC)

The supervisor mode is a general purpose privileged processor mode. In the current system, it is used to for general task execution.

5.5 Abort mode (ABT)

Abort mode can be used to implement virtual memory, which is not used in the current system.

5.6 Undefined mode (UND)

If an instruction cannot be interpreted, or is out of context, a call to address 0x4 is done, and the processor mode is automatically changed to undefined mode. This processor mode is very rarely used.

5.7 System mode (SYS)

System mode is in fact the same mode as user mode, except that it is privileged.

In the current system, system mode is used when handling interrupt service routines.

Appendix VIII Files

These are the files that contain µC/OS-II, the parts of core that are port specific, and the low-level implementations of task and interrupt handling.

```
includes.h                - Common µC/OS-II header file.
os_arm.c                  - ARM/THUMB specific µC/OS-II code.
os_arm.h                  - ARM/THUMB specific µC/OS-II definitions.
os_armaux.h               - ARM/THUMB specific µC/OS-II definitions.
os_cfg.h                  - µC/OS-II configuration constants.
os_core.c                 - Base µC/OS-II functions for intialisation
                            and task scheduling.
os_cpu.h                  - ARM/THUMB specific µC/OS-II definitions.
os_mbox.c                 - µC/OS-II mailbox functions.
os_mem.c                  - µC/OS-II memory pool functions.
os_q.c                    - µC/OS-II queue functions.
os_sem.c                  - µC/OS-II semaphore functions
os_task.c                 - µC/OS-II task initialisation and
                            configuration functions.
os_time.c                 - µC/OS-II tick and delay functions.
ucos_ii.c                 - Main µC/OS-II file that includes all non
                            platform specific code files (os_core.c,
                            os_mbox.c, os_mem.c, os_q.c, os_sem.c,
                            os_task.c, os_time.c).
ucos_ii.h                 - µC/OS-II configuration constants and
                            structs.

core\
  core\exp\
    core.h                - Core module header file.
    core_cache.h          - Core module header file.
    core_clock.h          - Core module header file.
    core_debug.h          - Core module header file.
    core_error.h          - Core module header file.
    core_fiq.h            - Core module header file.
    core_irq.h            - Core module header file.
    core_msg.h            - Core module header file.
    core_power.h          - Core module header file.
    core_support.h        - Core module header file.
    core_timer.h          - Core module header file.
    core_types.h          - Core module header file.
    core_version.h        - Core module header file.
    vos.h                 - Macro and type definitions needed when
                            using the VOS.
core\src\
```

```
  core.c                  - Core handlers, including the core idle task
                            function.
  critical.s              - Critical region functions.
  int_hdlr.s              - Low level interrupt handling and task
                            switching functions. Also defines the
                            stacks used in th interrupt handling.
  vos.c                   - VOS implementation that uses µC/OS-II.
  core\src\config\
    config_macros.h       - Defines macros needed to declare and
                            initialise tasks and mempool in vos.c.
                            Automatically generated by the config
                            program.
    config_mempool.h      - Defines constants and declarations needed
                            to use access the memory pool.
                            Automatically generated by the config
                            program.
    config_stacks.s       - Defines the sizes for the stacks used by
                            the interrupt handling, and is included in
                            int_hdlr.s. Automatically generated by the
                            config program.
core\system\
  boot.s                  - Contains the boot entry point, from where
                            everything is started. Calls CORE_Start()
                            in start.c.
  core_config.c           - Declares structures needed in vos.c, based
                            on core_config.h.
  core_config.h           - Defines system constants.
  core_handlers.c         - Contains core handler functions for SIGLOG
                            output.
  except.s                - Contains the exception vector.
  fiq_hdlr.s              - Contains the FIQ handler, included in
                            except.s.
  lm_install.c            - Configures the LM, called at start-up.
  start.c                 - Contains CORE_Start(), which intialises
                            core and µC/OS-II.
  start_hdlr.c            - Contains the core start handler.
  swi_hdlr.s              - Contains the SWI handler.
```