Examensarbete

Implementing Memory Protection in a Minimal OS

av

Richard Eklycke
Per Fagrell

LIU-IDA/LITH-EX-A--09/006--SE

2009-03-12
Final thesis

Implementing Memory Protection in a Minimal OS

by

Per Fagrell
Richard Eklycke

LITH-IDA-EX--09/006--SE

Supervisor : Mathias Bergvall
Embedded Platforms
at ENEA

Examiner : Professor Simin Nadjm-Tehrani
Dept. of Computer and Information Science
at Linköpings universitet

This report was written in \LaTeX.
Abstract

The car industry has created a series of standards called AutoSAR as a response to the increasing number of processors in modern vehicles. Among these specifications is one for real-time operating systems (RTOS). This RTOS standard includes requirements for memory protection. This thesis outlines the work involved in introducing the memory protection outlined in this specification in the OSEck operating system. The work consisted of updating the operating system, implementing the AutoSAR OS API, and updating the suite of tools used to build the finished system.

The AutoSAR specifications were found to be very thorough and well thought out. The OS API was successfully implemented, and the data-structures needed to permit its functionality. The existing software tools were updated to conform with the new requirements from AutoSAR, and additional software was created to ease the configuration process.

Memory protection was successfully implemented in the OSEck operating system, including two implementations of the trap interface. The memory protection functionality adds yet another layer of user-configuration to the operating system. Also, additional overhead for system calls, context switches and message passing is expected. A general evaluation of how OSEck application performance is affected is beyond the scope of this thesis, but preliminary studies of additional instruction counts on certain system calls have been performed.

Keywords: MMU, AUTOSAR, OSEK, MEMORY PROTECTION, EMBEDDED
Acknowledgements

We would like to extend our sincerest gratitude to Mathias Bergvall and Andreas Tyrberg for their many hours of help and discussion.
Contents

List of Figures x
List of Tables xii

1 Introduction 1
1.1 Background ............................................. 1
1.2 Problem Statement ....................................... 2
1.2.1 Adding Memory Protection ............................ 2
1.2.2 Solidifying AutoSAR Requirements ................. 2
1.3 Objectives ............................................. 3
1.4 Approach .............................................. 4
1.5 Related Work .......................................... 4
1.6 Limitations ............................................. 4
1.7 Document Structure ...................................... 5
1.7.1 Assumed knowledge .................................. 5

2 Background 7
2.1 Operating System Concepts .............................. 7
2.1.1 Kernel-Mode and User-Mode .......................... 8
2.2 Memory Concepts ....................................... 9
2.2.1 Memory Map ....................................... 9
2.2.2 Development ...................................... 10
2.2.3 The Problem ...................................... 11
2.3 Memory Protection ...................................... 13
2.3.1 Stack Monitoring .................................. 13
2.3.2 Paging .......................................... 13
2.3.3 Segmentation ...................................... 14
2.3.4 MPU/MMU ......................................... 14

3 Background and Motivation for AutoSAR 15
3.1 OSEK – The Foundation for AutoSAR ................. 15
3.2 The OSEK Implementation Language ................. 16
3.3 OSEK on OSEck and OILTool .......................... 17
3.4 HIS – Application Protection for OSEK ............... 19
3.5 History And Background of AutoSAR ................. 19
3.5.1 Extensions to OSEK .............................. 21
3.6 Future Compatibility ................................... 23
3.7 The SWAP Consortium .................................. 23
4 The OSE\textsubscript{ck} Operating System 25
  4.1 Overview of OSE\textsubscript{ck} 25
  4.2 Subsystems of OSE\textsubscript{ck} 25
    4.2.1 Build System 26
    4.2.2 Process Management 27
    4.2.3 Pool Management 28
    4.2.4 Signal Management 29
  4.3 Linking and Bootstrapping the System 31
    4.3.1 Linking OSE\textsubscript{ck} 31
    4.3.2 Bootstrapping the OS 33

5 Updating OS Structures 35
  5.1 Introduction 35
  5.2 Overview 36
  5.3 User-Mode Processes 37
    5.3.1 Implementation Feasibility Study 38
    5.3.2 Implementation of the Trap Interface 41
  5.4 Adding MMU/MPU-support 45
    5.4.1 Design 45
    5.4.2 Implementation 47
  5.5 Updating the OSE\textsubscript{ck} Structures 48
    5.5.1 Process Management 48
    5.5.2 Pool Management 49
    5.5.3 Signal Management 49
    5.5.4 An Example of Signal Management 50

6 Turning AutoSAR OS Requirements into Design 53
  6.1 AutoSAR Requirements Analysis 54
  6.2 The OS API and AutoSAR Data Types 54
  6.3 New Data Structures 55
  6.4 Updating OIL and Its Tools 56
    6.4.1 Updating OILTool 56
    6.4.2 Updating OIL Implementation Definition 57
    6.4.3 Compilation of Scalability Class Information 57
    6.4.4 Implementing ShutdownHook 57
  6.5 Keeping Track of the Current Application 58
  6.6 Requirement OS208 58

7 Integrating OSE and AutoSAR 61
  7.1 Overview of integration 61
  7.2 Adapting AutoSAR to OSE\textsubscript{ck} 61
    7.2.1 Additional Information From OIL Files 61
    7.2.2 User-mode/Kernel-mode for AutoSAR 62
    7.2.3 Handling Protection Errors 62
    7.2.4 Interrupt Handling 62
    7.2.5 New Test System 62
  7.3 Implementing OSE\textsubscript{ck} features requested by AutoSAR 63
    7.3.1 Killing/restarting Applications 63
    7.3.2 Page Table Operations 64
List of Figures

1.1 The two problems as top-down vs. bottom-up. .................................. 2
2.1 Two applications in memory. ......................................................... 8
2.2 Transition from unprivileged to privileged mode. ................................ 9
2.3 An example of how the memory map may be laid out. .......................... 9
2.4 Overview of a typical build process. ............................................... 10
2.5 Example of faulty loop logic leading to memory corruption ................. 12
2.6 Example of faulty pointer dereferencing. ....................................... 12
2.7 An example of how stack monitoring works. ............................... 14
3.1 An overview of the OSEK interfaces. ............................................. 16
3.2 Process description from OIL-file to executable binary. ....................... 17
3.3 Example of an AST for a simple two statement program. ............... 19
3.4 An overview of the AutoSAR Architecture. ................................ 20
3.5 Sequence diagram over trusted functions. ........................................ 22
4.1 An overview of OSEck and its subsystems. .................................... 26
4.2 An overview of the build process. .................................................. 27
4.3 A detailed view of the build system. ............................................. 28
4.4 How process data structures are related. ...................................... 29
4.5 An example of a pool with two free blocks. .................................. 29
4.6 Memory layout of a signal. .......................................................... 29
4.7 Example of the signal system in action. ....................................... 30
4.8 Result of link script example. ...................................................... 32
5.1 The second problem area in context. .............................................. 35
5.2 Dependency graph for OSEck implementation. .................................. 36
5.3 Example of how a weak alias works. ............................................. 40
5.4 Standard stack frame for the E500ABI[1] ...................................... 41
5.5 System call stack frames. ............................................................. 42
5.6 Sequence diagram of the signal new signal system. ......................... 50
5.7 Sequence diagram of the signal new signal system continued. .......... 51
6.1 The second problem area in context. ............................................. 53
6.2 Example of the Application data-structure. .................................... 55
6.3 Example of the Accessing Applications data-structure. ..................... 56
6.4 The problem presented by requirement OS208. ................................ 59
7.1 A rough sketch of the new test system. ........................................ 63
8.1 The main view of OILGen. 66
9.1 An example of an alignment error. 73
List of Tables

1.1 Growth of electronics in cars in the last 14 years.[2] . . . . . . . . 1

3.1 OSEK API Summary . . . . . . . . . . . . . . . . . . . . . . . 16
3.2 Lexical tokens generated by “sum=3+2;”. . . . . . . . . . . . . 18

6.1 Data types directly specified by AutoSAR. . . . . . . . . . . . . 54
6.2 API services directly specified by AutoSAR. . . . . . . . . . . . . 55

8.1 Number of instructions used in the dynamic table implementation. 67
8.2 Number of instructions used in the function pointer implementa-
tion. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
Chapter 1

Introduction

This chapter will serve as an introduction to this Final Thesis in Computer Science and Engineering for two final year students. It therefore corresponds to 60 ECTS in fulfillment of the final project for both students (30 ECTS each). A description of the project background, related work and limitations will be presented.

1.1 Background

Today’s modern car has more electronic equipment than ever. Everything from brake systems to cruise control are handled by computers. As the years have passed by, these embedded computers have evolved and the software in them has become more functionally complex. For example, many of the technologies used in green vehicles, such as hybrids, are managed through Micro-Controller Units (MCUs)[3]. Table 1.1 shows some approximate statistics on this increase in MCUs[2]. As more control is moved from the driver to the computer, it is even more important for the programs running on these computers to operate in a safe and reliable way.[4]

Automotive Open System Architecture (AutoSAR) is a series of standards from the automotive industry aiming at increased portability and reliability within automotive software[4]. Enea is an international company providing engineering services to global telecommunications industry, but are also involved in designing and developing their own embedded Operating System (OS)[5]. As of 2008 they have approximately 50% of the world market for platform software in new mobile phones and base stations[5]. Enea is part of a Swedish consortium known as Software Automotive Platform (SWAP)[6]. Among other

<table>
<thead>
<tr>
<th>Total in Vehicle</th>
<th>1994</th>
<th>2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCUs</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>MHz</td>
<td>85</td>
<td>2000</td>
</tr>
<tr>
<td>MCU memory (program + data)</td>
<td>1.1MB + 160kB</td>
<td>19MB + 1.25MB</td>
</tr>
<tr>
<td>Transistors</td>
<td>21 million</td>
<td>340 million</td>
</tr>
<tr>
<td>Bus Bandwidth</td>
<td>700kbit/sec</td>
<td>23Mbit/sec</td>
</tr>
</tbody>
</table>

Table 1.1: Growth of electronics in cars in the last 14 years.[2]
responsibilities within this collaboration, Enea is in charge of the development of a software environment which will facilitate building an experience knowledge base with the AutoSAR methodology. This software environment will be created around Enea’s existing Real-Time Operating System (RTOS) called OSE Compact Kernel (OSEck), which is an OS adapted for low-end embedded systems[7].

1.2 Problem Statement

This report aims to find answers to two main problem areas.

- “How is memory protection added to a minimal operating system not designed for it?”
- “What demands on the operating system do the AutoSAR requirements make?”

These were pursued in parallel, with each student being responsible for one problem area. The two problems can be seen as approaching an implementation of AutoSAR on OSEck from a bottom-up versus top-down approach, as shown in figure 1.1.

![Figure 1.1: The two problems as top-down vs. bottom-up.](image)

1.2.1 Adding Memory Protection

How is memory protection added to a minimal operating system not designed for it? Minimal in this case refers to an OS aimed at micro-controllers or a micro-kernel of similar complexity. An OS contains many data structures keeping track of numerous types of information. In a typical micro-kernel design, these structures are globally accessible. The applications run with the same privileges as the OS, and so it has no means of blocking access to sensitive data, or to set up any partitioning among the different programs.

The main goal in this problem area is to evaluate how one would add the necessary division of privilege, discover what inherent OS data structures need to be accessible to applications, and to ascertain the degree to which this access needs to be granted.

1.2.2 Solidifying AutoSAR Requirements

What demands on the operating system does the AutoSAR requirements make? The AutoSAR specification is written from an application developer's point of view and as free from implementation details as possible. Some requirements have a direct mapping to hardware or OS features. Most are open to at least
some amount of interpretation. Some requirements are restricted by functional requirements or general requirements placed on the system as a whole.

The main goal in this problem area is making a comprehensive list of what demands these requirements make on the OS and development environment. This list will then be the basis of a mapping between the AutoSAR OS standard on the one side and OSE<sub>ck</sub> on the other.

1.3 Objectives

The problems will be solved with the intention of:

1. Making a concrete implementation of the requested functionality for OSE<sub>ck</sub>.

2. Extending the current functionality of the programs associated with developing software for OSE<sub>ck</sub>.

3. Fulfilling a major portion of the requirements requested by the AutoSAR OS standard.

Requirements for Updating a Minimal OS

The following list outlines the work items to be done in order to successfully achieve the goals of the objective 1 above.

- Trap interface implemented for all system calls. Normal operation in user-mode, and OS and system operation in kernel-mode.

- Memory Management Unit (MMU) initialized correctly on start-up and working with the implemented memory protection features.

- OSE<sub>ck</sub> subsystems working correctly with the new logical division of software into applications.

Requirements From the AutoSAR OS Specification

The following list outlines the work items to be done in order to successfully achieve the goals of the objectives 2 and 3 above.

- OilTool fully updated to handle new OS Objects and parameters. Error checking and content validation shown to be working correctly. New demands from integration documented.

- AutoSAR OS API implemented.

- A list of new requirements and design decisions made during the course of the work.

---

1The specifications also outline timing protection features for hard real-time environments that we do not consider in this thesis.
1.4 Approach

As the Master Thesis is carried out by two people, the work is divided accordingly:

**Richard Eklycke** will examine how memory protection functionality can be introduced into the core parts of OSEck. This includes writing code to interface with the MMU, as well as introducing a separation between kernel-mode and user-mode.

**Per Fagrell** will examine how to use the more primitive memory protection functionality being implemented in OSEck to fulfill the requirements for the AutoSAR OS specification. Per will also examine the need for updating the development tools required to build programs using the new protection functionality.

In general, the idea is to be more broad in our approach than necessary to make any findings as applicable as possible to a variety of readers.

1.5 Related Work

The basics of memory protection have been studied and implemented since at least 1961[8]. Memory protection of some sort can be found in modern Unix, Linux and Windows OSs. The principles involved are part of any introductory course in OS programming.

The AutoSAR standard has been implemented by several other companies. Freescale², primarily a micro-controller manufacturer, has released an AutoSAR compatible OS and tool-set. This is particularly tuned to the Freescale processors in the S12X, MPC556x and MPC551x MCU families[9]. Vector³ has a software bundle[10] for the development of AutoSAR software and a concrete implementation of most of the Basic Software Modules[11]. Elektrobit⁴ was the first company to release a full AutoSAR core, and offers several tools to help in the development of Electronic Control Unit (ECU) software, as well as a full core and suite of Basic Software Modules (BSW)[12].

The main difference between all these commercial solutions and the one developed in the course of this project, is the fact that the end user – members of the SWAP consortium – will have access to all the code. Since the whole project is educationally oriented[6], documentation of the efforts required to implement AutoSAR and the possibility of working hands on with the code weigh very high.

1.6 Limitations

The final product is only required to work on one specific microcontroller, the MPC5567[13]. It will most likely be compatible with most if not all Book-E e200z6 PowerPC processors. If there is enough time, the produced code will be

---
²http://www.freescale.com
³http://www.vector-worldwide.com
⁴http://www.elektrobit.com
revised and refactored to make it easier to port to other target processors. The project uses version 2.0.1 of the AutoSAR specification and not the 3.1 version because newer versions were not publicly available when the project started, and the differences between the releases – as pertains to memory protection at least – were minimal.

1.7 Document Structure

The thesis has the following organization:

Chapter 2: Background information. Contains a brief review of OS and memory concepts. Introduction to basic memory protection mechanisms.

Chapter 3: A background of AutoSAR and the motivations that led to its creation.

Chapter 4: An overview of the OSEck OS.

Chapter 5: A summary of the work done to update OSEck with memory protection.

Chapter 6: A summary of the work done to solidify the AutoSAR requirements into design and the problems encountered during this work.

Chapter 7: A summary of the work done to integrate the two halves of the project.

Chapter 8: An overview of the projects results, including basic measurements of performance and size impacts of the extensions.

Chapter 9: A discussion of the project and our thoughts on future work within this area.

1.7.1 Assumed knowledge

This thesis is written with the assumption that the reader has at least taken a course in basic computer hardware. Knowledge about assembler programming in general is assumed, as well as an understanding of basic computer components such as Central Processing Unit (CPU) and memory.
Chapter 2

Background

This chapter will introduce background information needed to understand the rest of the report and the decisions made as part of the work. It begins with a brief introduction to operating systems and how they work, then goes through the concepts of memory and building an embedded system. The chapter is concluded with a discussion about the need for memory protection and an outline of some basic memory protection mechanisms.

2.1 Operating System Concepts

An operating system (OS) is software that manages hardware and creates an environment for applications. It provides access to resources, divides use of the processor to allow multiple programs concurrently and deals with interrupts in a uniform manner. The exact services provided by the OS vary but this simplification of hardware usage and allocation of resources is universal. The one program running at all times in a system is called the kernel.[14]

Code that is loaded into memory and executing is called a process. An application, for example a text-editor, can be made up of several processes. If two instances of the same application are running, only the data will be different. In this case, the two instances can share code (see figure 2.1). In small systems this type of code-sharing can be very important due to memory restrictions.[14]

A stack is a segment of memory which contains data about the active subroutines for a process. Such data typically include function parameters, return addresses, and local variables[14]. For example, for every function call data related to the call is pushed onto the stack. This data is preferably organized according to the stack frame defined in the application binary interface (ABI) for the target architecture. In OSEck, there is a stack for each process. Some OSs have a separate stack for the kernel[14].

---

1Interrupts are signals from hardware or software that cause execution to be temporarily halted while specialized interrupt code is run.

2It is of course possible to use one’s own set of conventions. However, interfacing with e.g. C-programs will require much more work unless the compiler supports the chosen set of conventions.

3A set of low-level conventions for a particular architecture. It is used to make the interface between applications and the OS behave in a consistent way.
2.1. Operating System Concepts

(a) Two applications in memory.
(b) Two applications. Program B and C share a code segment.

Figure 2.1: Two applications in memory.

2.1.1 Kernel-Mode and User-Mode

It is often desirable in an OS to have protection facilities that prevent user programs from directly interfacing with the state of the CPU or the underlying hardware. An efficient way of enforcing this is to divide the processor instruction set into privileged and non-privileged instructions. The user processes are only allowed to execute non-privileged instructions, while the kernel and kernel services are allowed all types of instructions. If the user programs are not allowed to execute privileged instructions, but still want to interface with the underlying hardware, they have to request these services in a controlled manner from the kernel. The user processes do this by issuing a so-called system call. A system call is just like a normal function, except that it is executed by the OS in a privileged mode. By having the kernel execute the privileged instructions, the state of the system and its hardware can be maintained in a controlled way. When the system call is finished, the process returns to the normal unprivileged execution mode and the user program continues from where the system call was invoked.\cite{14}

For a system call to be able to enter privileged mode, there must exist some mechanism for this transition. Typically, this mechanism is provided for by a special machine instruction that issues a software interrupt which forces the processor to enter privileged mode. More specifically, when the system call instruction is executed, the program counter jumps from the user’s code to an exception handler and continues the execution of code from there. The exception handler, which runs in privileged mode, can then dispatch the selected system call. The set of rules for passing and processing the parameters sent to the exception handler is, in this thesis, called the trap interface. For the exception handler to be able to know what system call to dispatch, the user program must provide this information in some way. This is typically done by either passing relevant data in registers or as an argument to the system call instruction. The transition from unprivileged mode to privileged mode is depicted in figure 2.2.\cite{14}

Privileged mode is often called kernel-mode or supervisor-mode, while unprivileged mode is often referred to as user-mode. These terms are synonymous, but the pick of choice is different for each hardware vendor. In this thesis, the
2.2 Memory Concepts

This Section will explain some background information on memory, development for embedded systems and give an introduction to memory protection. For the purposes of this report, memory will always mean immediate access memory, such as RAM or ROM, and not secondary storage, such as flash drives or hard-drives.

2.2.1 Memory Map

In any computer system, there are two types of memory. There is non-volatile memory for the long-term storage of program code and predefined variable values and there is volatile memory for run-time variables and stack usage. Accessing these different regions – located on different physical chips – in a consistent manner is made possible by the memory map. An example is shown in figure 2.3. The different memories are given separate non-overlapping address ranges. This way there is a single address space to access all the different memory chips.[15]
2.2.2 Development

This section outlines the basics of software development needed to understand the problems faced when implementing the memory protection scheme. It also explains the difficulties that arise from programming for an embedded platform.

Programming for a Desktop Environment

In a general purpose operating system (GPOS), such as Microsoft Windows or Linux, software development is a relatively simple process. Code is written in one or more source files, compiled and executed. The GPOS will load the program into memory and hand over the CPU to the program entry-point. This process is made possible by the wealth of information about the system the compiler has and the dynamic nature of program loading in a GPOS. An overview of the build process can be seen in figure 2.4(a).[16]

Programming for an Embedded Environment

In an embedded environment, the compilation process contains much less automation as assumptions about the target platform cannot be made by the software. There are also a number of other features that are present in a GPOS that an embedded programmer may have to provide himself.[16]

The process begins with writing source code. The code is then compiled into object files. The object files are linked together with bootstrap – or start-up – code which initializes the hardware environment. If the system uses a high level language like C, language run-time routines also need to be linked in. This file does things like prepare the stack and handle function calls. If the system uses an OS then this is usually linked in as well. The next step is to locate the file. This means that the system is told how much memory is available and where in memory each part of the system should end up, see figure 2.4(b). This information is written in a link script. In a GPOS, locating is done for you at
run-time. Once this information is processed there is a binary file that is ready to be transferred to the embedded system in question.\cite{16}

\subsection{The Problem}

The problem with a system running without memory protection is that no non-trivial program is free from programming errors\cite{17}. As mentioned in \cite{18}, even in a system with 99\% bug-free code, the remaining 1\% can corrupt the rest of the system causing a crash.

There are essentially five scenarios that require memory protection to catch programming errors. Suppose there is a system running two applications, Task1 and Task2. If Task1 contains a programming error that makes it save data into a memory address that does not belong to it, four different outcomes may be observed. The address may belong to the OS itself, potentially leading to a total system failure as some key part of the system may be corrupted, for example code related to process scheduling. Another outcome may be that an existing data structure is corrupted, resulting in unpredictable behavior as decisions are made based on wrongful information. If the address belongs to Task2, then Task2 might crash. This is seldom recoverable, and the OS will at best kill Task2 and restart it, and at worst crash along with Task2. The most difficult situation to deal with is if the address belongs to Task2, and the corruption does not cause Task2 to crash. This means that the programming error may never be noticed, or that Task2 behaves oddly due to an incorrect variable, resulting in many hours work trying to find a fault in Task2 which in this case is not causing the problem. Similarly, if Task1 instead reads data from a wrongful address, it may behave strangely, despite its data seeming to be correct.

There are many ways a program may end up attempting to write to an address that does not belong to it. Consider the following example. The code below was intended to create an array of ten integers, and then place the first ten squares in it. Due to programming error, however, the loop will try to write to 100 places within the array.

\begin{verbatim}
int arr[10];
int ctr;

for(ctr=0; ctr<100; ctr++) { //error
    arr[ctr] = ctr*ctr;
}
\end{verbatim}

This scenario will obviously lead to unintended consequences, e.g as shown in figure 2.5. If the array is at the end of the current program’s memory area, then it may never notice the error. The first ten places in the array work just as expected, and the area erroneously written to is never referenced. This error is known as a buffer overflow\cite{19}, since the array that is involved is often a character buffer.

Another common scenario is pointer\textsuperscript{4} dereferencing errors. Consider the situation that a section of code has a pointer to an integer, which should hold

\textsuperscript{4}A pointer is a programming variable that holds an address instead of any actual content. By dereferencing the pointer it is possible to access the content at the address the pointer points to.
2.2. Memory Concepts

(a) Before erroneous loop.
(b) After erroneous loop.

Figure 2.5: Example of faulty loop logic leading to memory corruption

When the program tries to use `procs_left`, a value that can be seen as essentially random will be used instead of the intended value. Writing to the
address is also an error, with potentially disastrous results. An example of this is depicted in figure 2.6.

In both of these cases, the programming error consisted of one single character. Clearly there is a large probability that a system with several applications running many hundreds or thousands of lines of code will have at least a few of these types of errors. Since some of these may only make themselves known under a very specific set of circumstances, and even then may not cause an immediately noticeable fault, even very thorough testing may miss a critical programming error. Memory protection helps not only find the bulk of the errors in a system during development, it also protects other processes at run-time from these hard to find bugs.

2.3 Memory Protection

This section will cover the various ways in which hardware or the OS may enforce memory protection. Segmentation and paging are discussed, followed by a description of how an MMU or MPU can be used to help restrict access to a process’ memory.

2.3.1 Stack Monitoring

When a function is called several pieces of information are placed on the stack. If the function calls itself, and ends up in a loop where each new call calls the function, the stack will eventually fill with this information. This is called a stack overflow. There are other ways a stack can overflow, but recursion – when a function calls itself – is a very common one. Another common way is a buffer over-run. This is when a series of characters are saved to a memory region which is not big enough to contain them all. The characters at the end are written to memory which does not belong to the buffer, and if there are enough of them may corrupt the stack’s memory.[14]

Since the stack is needed for the correct functioning of the system this is a very serious problem. Overflowing the stack will write to a memory region that does not belong to the stack, which means its contents may be written over by whatever process owns the memory. Alternatively, that process may crash as a result of the overflow.[14]

Stack monitoring is used to check for stack overflows. This means that this is not a preventative mechanism. For an example of how stack monitoring works, see figure 2.7. The monitoring is done by writing a special character sequence – commonly called a canary – to the top of the stack. At some point the system checks the top of the stack to make sure that the special characters are still there. If not, they have been written over by – for example – a buffer over-run, and the system can be halted before any major fault occurs.

2.3.2 Paging

Paging is a way to use a very large virtual address space to address a smaller physical address space. This way a very large program may think it has its entire contents in memory, but actually have parts of it stored in secondary storage, to be read into memory when that region is referenced for the first time. The
address space is divided into even sized areas known as \textit{pages}. These pages can be marked with access rights, such as read-only or read-write.\cite{19}\cite{14}

\subsection{Segmentation}

Segmentation is the dividing of memory into contiguous pieces of variable size. A program then uses addresses made of two parts, a segment identifier and an offset into that segment. The OS then translates this into a physical address. By using different segments for different functions and data, it is possible to apply access rights to increase security. For example, data can be marked as read-write but not executable or a function marked as execute only.\cite{19}

\subsection{MPU/MMU}

A Memory Management Unit (MMU) is a piece of hardware, often part of the processor core in modern computers, that manages address translation for paging, and access rights for memory protection. The MMU is comprised of translation hardware, which converts a virtual address into a physical address, and a Translation Look-aside Buffer (TLB). The TLB keeps a cache of recent page look-ups to speed up the fetching of physical addresses\cite{20}\footnote{For instance, if a 4kB section of code executes in order then there would only be one page lookup.}. Since the translation of an address produces a delay, it is important to reduce the number of TLB look-up misses\cite{19}. Normally this is helped by a page-replacement algorithm, which replaces the page it finds least used. A smart algorithm needs hardware support to tell if a page has been accessed or not. As mentioned in \cite{20}, TLBs generally have high hit ratios\footnote{A hit ratio is how likely it is that a given address is in the TLB.}, in excess of 90 percent. \cite{14}

A Memory Protection Unit (MPU) has all the functionality of an MMU except for virtual address translation. An MPU simply applies access rights for a specified memory area.
Chapter 3

Background and Motivation for AutoSAR

This chapter will give background information about AutoSAR and SWAP. A brief history of OSEK (German: “Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen”) and the AutoSAR architecture will be given, followed by a brief motivation for the SWAP project. Finally we will place our work in the context of these.

3.1 OSEK – The Foundation for AutoSAR

OSEK/VDX\(^1\) is a standards body which has released several specifications and standards for automotive ECUs and their operating systems. Formed in 1993, the idea was to provide an open-ended architecture for the many micro-controllers and other processors in a car. The first version of the specifications was released in 1995. Its design took into account the strict real-time demands of the automotive industry, and is fully event driven. The goal of the design was not 100% compatibility between different ECUs, but direct portability\(^{21,22}\).

The OSEK system is characterized by modularity and the potential for flexible configuration. The time-critical nature of OSEK systems means that no dynamic generation of processes or memory allocation is allowed. Everything is specified at configuration time, and generated by a system generation tool. This tool needs to be created by the implementer and is not a part of the specification. The standard allows for two levels of error handling, different conformance classes requiring more or less from the ECU hardware, and specifies a standardized interface. For a logical view of the OSEK interfaces, see figure 3.1\(^{22}\).

OSEK introduces several new concepts that sit on top of the existing OS. First the introduction of *Task* and *Interrupt Service Routine*\(^2\) (ISR) object to encapsulate processes. The use of *Events* to signal task state changes, and *Alarms* to trigger actions based on the system ticks\(^3\), and the configuration of the complete system through the use of a file or series of files in the OSEK

\(^1\)http://www.osek-vdx.org

\(^2\)An ISR is a process that runs from start to finish and is triggered by either a software or hardware interrupt

\(^3\)A tick is one complete rise-fall cycle of the processor's clock. A system tick is usually triggered on some multiple of the clock tick, for example once every millisecond.
3.2. The OSEK Implementation Language

The OSEK system is configured from an OIL file, which a tool can then use to tie the system together and generate any data that is needed for the API services to work. The idea is to make the application software more portable through a standardized configuration[21]. By having the hardware specific settings in a separate file, no changes need to be made to the system configuration when moving it to a new type of MCU.[24]

OIL files are divided into two main sections. The sections are called the implementation section and the application section. The implementation section describes what objects and attributes there can be in the application section and what if any default values they have. The application section then describes the current system and its components. The language supports include directives so

<table>
<thead>
<tr>
<th>ActivateTask</th>
<th>TerminateTask</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChainTask</td>
<td>Schedule</td>
</tr>
<tr>
<td>GetTaskID</td>
<td>GetTaskState</td>
</tr>
<tr>
<td>DisableAllInterrupts</td>
<td>EnableAllInterrupts</td>
</tr>
<tr>
<td>SuspendAllInterrupts</td>
<td>ResumeAllInterrupts</td>
</tr>
<tr>
<td>SuspendOSInterrupts</td>
<td>ResumeOSInterrupts</td>
</tr>
<tr>
<td>GetResource</td>
<td>ReleaseResource</td>
</tr>
<tr>
<td>SetEvent</td>
<td>ClearEvent</td>
</tr>
<tr>
<td>GetEvent</td>
<td>WaitEvent</td>
</tr>
<tr>
<td>GetAlarmBase</td>
<td>GetAlarm</td>
</tr>
<tr>
<td>SetRelAlarm</td>
<td>SetAbsAlarm</td>
</tr>
<tr>
<td>CancelAlarm</td>
<td>GetActiveApplicationMode</td>
</tr>
<tr>
<td>StartOS</td>
<td>ShutdownOS</td>
</tr>
</tbody>
</table>

Table 3.1: OSEK API Summary

Implementation Language(OIL) format. Resource management in the form of a semaphore object called Resource and the use of Priority Inheritance Protocol ensures that there can be no deadlock between two tasks[23]. For an overview of the OSEK API, and thereby a gist of the types of services the OSEK layer provides in a complete OSEK OS, see table 3.1.[22]
the two parts can be in separate files, allowing for a common hardware-specific implementation section which is kept separate from the system configuration.[24]

The program that processes the OIL file and generates system specific code and configurations is created by the implementer as the output will be implementation specific. The original OSEK OIL specifications make no requirements on the outputs nor on the validation of inputs, apart from the grammatical correctness required to parse the file(s).

![Figure 3.2: Process description from OIL-file to executable binary.](image)

### 3.3 OSEK on OSEck and OILTool

OSEck has a working OSEK layer that was produced as part of an earlier thesis. Jenny Palmberg and Lili Ren produced an OSEK layer, an OIL parser called OILTool and a test system to run regression tests on the new features in the OSEK layer. An overview of how these components work together to produce a whole system can be seen in figure 3.2. The configuration file is given as input to the OIL Compiler, in this case OIL Tool, which generates source code containing system configuration information and functions that are registered with OSEck through the appcon.com configuration file which is also generated. This is fed to the mkconfig program which comes with OSEck which generates further source code to be compiled and linked with the rest of the system.

The OILTool parser is generated by ANThr⁴, an open-source language tool that can generate a parser for a given grammar in one of several target languages.

The ANThr grammar file is composed of two parts. The first is the lexer rules. A lexer – short for lexical analyzer – scans a source file and converts a series of characters into a series of tokens with associated data[25]. For example, the string “sum=3+2;” can be tokenized as in tab. 3.2. An example of how the lexer’s rules can look is presented below. It shows a series of rules needed for the lexer to identify a valid object name from a series of characters.

---

⁴http://www.antlr.org
### 3.3. OSEK on OSECK and OILTool

<table>
<thead>
<tr>
<th>Token</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>Identifier</td>
</tr>
<tr>
<td>=</td>
<td>AssignmentOp</td>
</tr>
<tr>
<td>3</td>
<td>Number</td>
</tr>
<tr>
<td>+</td>
<td>AdditionOp</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
</tbody>
</table>

Table 3.2: Lexical tokens generated by “sum=3+2;”.

The second part is the language parser. The language parser is used to create a syntax tree of the tokens from the lexer and then do syntactic and semantic analysis on this intermediate representation to check that the program is valid\[25\]. For example in any C-like language (such as C, C++ or Java), the following statement is grammatically incorrect: “=sum2+3;”. If this was fed to a parser for that particular language, an error would be generated as the parser would fail to identify the line from the rules it has that apply to making a valid statement. An example from the OIL grammar is presented below. The example shows the two accepted ways of how an object definition should be written. The last rule, name, also shows how the parser and lexer work together, as the parser rule name matches the lexer rule NAME.

```plaintext
DEC_DIGIT
   :   '0'..'9' ;

LETTER
   :   'a'..'z'|'A'..'Z' ;

NAME
   :   (LETTER|'_')(DEC_DIGIT|LETTER|'_')* ;

object_definition
   :   object_name description ';
   |   object_name '{' parameter_list '}' description ';

object_name
   :   object name ;

object
   :   'OS' | 'APPLICATION' | 'TASK'
   |   'COUNTER' | 'ALARM' | 'RESOURCE'
   |   'EVENT' | 'ISR'

name
   :   NAME ;
```

The parser generated by ANTLR in turns creates an Abstract Syntax Tree (AST), the manipulation of which makes up the bulk of OILTool. For an exam-
ple of an AST, see figure 3.3, which represents the following program:

\[
\begin{align*}
A &= 1 + 3; \\
B &= 2 - 1;
\end{align*}
\]

![Figure 3.3: Example of an AST for a simple two statement program.](image)

### 3.4 HIS – Application Protection for OSEK

The need for an application concept has already been identified and partially applied by Hersteller Initiative Software\(^5\), or HIS. HIS proposed a specification called HIS Protected OSEK OS, which was an extension of OSEK OS and OIL to provide an application concept and assigning system objects to these applications\([26]\). AutoSAR builds in part on the HIS standard, but deemed the standard not mature enough\([11]\). HIS is only an interest group, not a real consortium\([27]\), and due to the emergence of AutoSAR and the extension of the HIS Protected OSEK OS therein the group in charge of the specifications is currently inactive, with members of HIS contributing directly to AutoSAR\([28]\).

### 3.5 History And Background of AutoSAR

At the end of the last millennium the automotive industry started to implement more and more innovative features in cars as embedded computers. Driver assistance features such as GPS-navigation and proximity warnings became popular. Also, dynamic drive systems that could for example adjust motor timing depending on driving conditions started being introduced. These new and innovative uses of computer systems brought with them a increased complexity. As more of these functions started being hosted on the same networks or nodes, the complexity, and inherent software bugs started to become a problem. A breakthrough was needed to handle this growing issue.\([29]\)

Several Original Equipment Manufacturers(OEMs) and Tier 1 suppliers\(^6\) within the German automotive industry came together in 2002 over this issue.

---

\(^5\)http://www.automotive-his.de

\(^6\)A tier 1 supplier is a company that provides products and services directly to one or more OEMs.
and decided to work together to create a collaborative platform. The basic idea was to collaborate on basic functionality to create a robust platform. This would then allow competition with new functionality, without worrying about portability or availability of low-level functions such as communication. These initial discussions resulted in the Automotive Open System Architecture, or AutoSAR. A project plan was released in 2003, and 2006 the first version of the architecture was completed, see figure 3.4 for an overview. At the top there is AutoSAR software. This is what software developers write and contains the useful functionality. The software sits on top of the AutoSAR runtime environment, which handles inter-process communication, hiding whether or not the processes are on the same physical device or communicating via a network. The runtime environment works together with the Basic Software (BSW) which provide functionality such as network communication and process finding in a distributed environment. Complex device drivers are user provided modules that enable peripheral hardware such as sensors or actuators.

The formal motivations of the group were:

- Management of E/E (Electronic/Electrical) complexity.
- Flexibility for product updates.
- Scalability of solutions across product lines.
- Improved quality and reliability of E/E systems.

The main objectives were modularity, configurability and transferability of software modules, and the standardization of their interfaces. A large number of specifications have been developed, including APIs for standardized ba-

---

Figure 3.4: An overview of the AutoSAR Architecture.
sic software (services and communication interfaces etc.) and operating system functions. The AutoSAR mantra is “Cooperate on standards, compete on implementation”[30]. Formally, the AutoSAR partnership has the following goals[4]:

- Implementation and standardization of basic system functions as an OEM wide Standard Core solution.
- Scalability to different vehicle and platform variants.
- Transferability of functions throughout the network.
- Integration of functional modules from multiple suppliers.
- Consideration of availability and safety requirements.
- Redundancy activation.
- Maintainability throughout the whole “Product Life Cycle”.
- Increased use of “Commercial off the shelf hardware”.
- Software updates and upgrades over vehicle lifetime.

3.5.1 Extensions to OSEK

All requirements for AutoSAR are given as extensions or limitations to the OSEK Specifications. The operating system must implement the entire OSEK API to be AutoSAR compliant, as AutoSAR is meant to be fully backwards compatible with existing OSEK applications and device drivers. The AutoSAR OS configuration’s scalability is summarized by its Scalability Classes. The AutoSAR OS supports all four classes, and the developer then selects the class most appropriate for their system. SC1 adds nothing more than optional stack monitoring. SC2 adds timing protection features. SC3 adds memory protection features, and is the Scalability Class we focus on in this report. SC4 is a combination of SC2 and SC3.[11]

OS-Application

The biggest difference over OSEK that the AutoSAR specification’s SC3 brings is a fault-containment region known as an OS-Application. An OS-Application groups one or more tasks with zero or more ISRs, hooks, alarms, events or resources. Internally the member objects of an OS-Application can communicate directly via shared memory associated with the OS-Application. Access to another OS-Applications objects such as events and alarms is in principal disallowed. Different OS-Applications cannot write to each others memory. Sharing functions, tasks etc. is made possible through operating system mechanisms, and the associated privileges must be specified at compile-time through the OIL-file. It should be noted that the OS-Application is a meta-object. That is to say, there is no specific code that runs as the OS-Application, it consists solely as the data that ties together its members and their access rights.[11]

The use of OS-Applications is voluntary. If no OS-Applications are specified, then there is simply no memory protection. However, if any OS-Application is
defined, then all OS objects must belong to an OS-Application. OS-Applications come in two variants: trusted and untrusted. A trusted OS-Application runs without any memory protection mechanism and has read and write access to all memory. If the MCU has support for execution modes, a trusted OS-Application will run in kernel mode. The idea is to reduce the overhead for programs that have been statistically proven to be correct off-line. Trusted OS-Applications can also export trusted functions; i.e. let other – potentially untrusted – OS-Applications call the function. There is an OS service that works as a middle man when using trusted functions. See figure 3.5 for an overview of the sequence of control. Parameters are passed as a pointer to a struct containing the function arguments.[11]

Figure 3.5: Sequence diagram over trusted functions.
An application or task should be free to run without the protection mechanisms in any scalability class. Due to the fact that timing requirements cannot be enforced for Category 1 ISRs\(^8\), they must belong to a trusted application. Likewise, the use of Alarm Callbacks – callback hooks that are executed upon an alarm going off – is forbidden in SC2 and higher due to the overhead associated with their use.[11][31]

The protection mechanisms should ensure that – if it is possible with the current hardware configuration – the OS code is protected from user code. Any violation of the protection requirements – as well as any instruction exceptions – will cause the new ProtectionHook to execute. The ProtectionHook is defined by the user, and returns a value indicating if the system is to be shut down, the application terminated and possibly restarted, or the current task should be terminated. In the case of OS service errors, there are now per-application ErrorHooks. Additionally, applications may define an individual Startup or ShutdownHook which is called on OS start or shutdown respectively.[11][31]

### 3.6 Future Compatibility

The requirements this chapter has presented are for the 2.1.0 version of the AutoSAR OS Specification. However, thorough study of the 3.0.1 specification, which was released during the course of our work, shows that no changes are needed to be 3.0.1 compliant, aside from supporting the use of XML in addition to OIL as a configuration language[32]. As it is a simple task to create an XML-parser using ANTLR and the existing OIL grammar file[33], this is not a particularly worrying hurdle.

### 3.7 The SWAP Consortium

SWAP, or Software Automotive Platform, is a joint venture within the Swedish automotive software industry. It is a cooperative effort between Enea, Mecel, Consat and QRtech that was started in 2007. The purpose of this effort is to increase the competitive edge of the region. They mean to increase the knowledge related to around AutoSAR and develop a prototype platform along with an implementation of the AutoSAR Basic Software (BSW). By increasing the knowledge of the venture partners about automotive software development and AutoSAR in particular they hope to empower the region when trying to win contracts with auto manufacturers.[6]

While QRtech will be responsible for the prototype platform, which will include both BSWs and a hardware test environment, Enea has been tasked with developing its OSE\(_{ck}\) operating system to be usable as a base for the system. To capitalize on the BSW implementations and to get a foot in the door for more interesting service and integration jobs Eneas existing sales channels will be used to commercialize any progress in the long run.[13]

---

\(^8\)Category 1 ISRs run in kernel mode and cannot be pre-empted by the OS.
Chapter 4

The OSE\textsubscript{ck} Operating System

This chapter will serve as an introduction to the OSE\textsubscript{ck} OS. It will cover the fundamentals of OSE\textsubscript{ck}, and will be prerequisite knowledge for understanding later chapters.

4.1 Overview of OSE\textsubscript{ck}

OSE\textsubscript{ck} is a small RTOS developed by Enea. It has a memory footprint that can be as low as 8kB, which is small enough to run on most Digital Signal Processors (DSPs). The core OS functionality is optimized for high performance, enabling application developers to get the most out of the underlying hardware. The kernel provides 43 system calls, although most applications can be written using only 5 basic calls.[7]

OSE\textsubscript{ck}'s minimalism and performance makes it a good choice for developing the OSEK and AutoSAR layers on top of it. However, there exists a wide variety of RTOSs on the market which could also be used. Besides high performance, OSE\textsubscript{ck} was also chosen because of its maturity and the possibility to edit its source code.

First the OS and its subsystems in general will be presented. Then an explanation of the details of linking and starting the OS will be made.

4.2 Subsystems of OSE\textsubscript{ck}

This section will introduce the important subsystems in OSE\textsubscript{ck}. There are four subsystems that will be explained in more detail:

**Build system:** The build system generates information about all the entities in the OS, such as pools, processes and ISRs, and compiles the system.

**Process management:** Processes are the actual threads of execution running on the OS.

**Pool management:** Pools are used to mark areas of memory available for dynamic or static allocation.
4.2. Subsystems of OSE\textsubscript{ck}

**Signal management:** Signals are used for inter-process communication (IPC).

![Diagram of OSE\textsubscript{ck} subsystems](image)

**Figure 4.1:** An overview of OSE\textsubscript{ck} and its subsystems.

The interactions between these four subsystems are depicted in figure 4.1.

### 4.2.1 Build System

In OSE\textsubscript{ck}, unlike most GPOSs, the OSE\textsubscript{ck} Kernel and its applications are not divided up into separate executable binaries\(^1\). Instead, both the OS and its applications are compiled and linked into one single monolithic executable, see figure 4.2. This executable is then programmed into memory. By doing this, there is no overhead introduced by an OS-loader for starting an application, as the only loading is done when programming the chip. However, some of the information such as process entry point gets lost since the applications themselves lack an executable format. To keep track of this information meta-data must be supplied to the OS and compiled into the executable.

The build system is responsible for generating and setting up this meta-data, or configuration, for the OS, see figure 4.3. The file `appcon.con` is central for the configuration of the OS. There the user configures various aspects of the system such as what processes exist, how many memory pools should exist, what hooks should be activated and so on. The configuration file is processed by a program called `mkconfig`. This program translates `appcon.con` into a C source file containing data structures the OS then uses. [34]

To get an idea of what this file looks like, a small example of a configuration is shown below:

```
POOL(0, 4096, 16, 32, 64, 128)
PRI_PROC(0, my_process, my_process_entry, 512, 10)
```

The first line sets up a pool with ID 0, a total size of 4 KB, with memory allocatable in 16, 32, 64 and 128 byte chunks. The second line declares a process with an automatically assigned process ID (PID) with the name “my\_process”.

---

\(^1\)As described in section 2.2.2.
The entry point of the process is called `my_process_entry()`, the process’ stack size is 512 bytes, and it should run with a priority of 10.

The example configuration would then be parsed into a C-file containing this information. The C-file gets compiled into an object file, and finally linked into the monolithic executable.

### 4.2.2 Process Management

The process management is responsible for creating, killing and switching between processes. Information about the static processes – such as priority, pool information and entry point – are generated by the build system. It is also possible to start processes dynamically in run-time by using the system call `create_process()`.

There exist two types of processes in OSEck: prioritized processes and interrupt processes. Prioritized processes are the most common type of process. A prioritized process must never return, and is therefore written as an infinite loop. Interrupt processes are invoked when there has been a hardware interrupt. Unlike a prioritized process, the interrupt process will run from the beginning to the end, unless an interrupt with a higher priority occurs.

There are two process information blocks the kernel handles: Process Description Block (PDB) and Process Control Block (PCB). The PDBs contain immutable information that is important for initializing a process. Setting up a process begins with allocating a PCB. Some of the data from the PDB associated with the process is then copied into the PCB along with the process’ state information. A graphical view of the relationship between these data structures can be seen in figure 4.4.
4.2.3 Pool Management

For memory allocation to work in OSEck, at least one pool must be defined. In its basic form, a pool takes a region of memory and keeps track of how much of that memory is allocated by the OS or processes[34]. Meta-data for the pool is stored at the start of the pool’s memory region. In figure 4.5 an example of a pool managing two free blocks is shown. Any process is free to allocate memory from any pool available in the system. There are several system calls that use pools to allocate memory. These are the two most important:

alloc(): Allocates a chunk of memory from the system pool (pool 0). If the memory in the system pool is exhausted, the system panics\(^2\). There is a special pool in OSEck called the system pool (pool 0) that must exist for the OS to function correctly.

free_buf(): Deallocates a previously allocated chunk of memory.

When using alloc(), it is only possible to allocate chunks of fixed sizes. If an application tries to allocate a larger chunk of memory than the largest defined chunk size, the system will panic.

\(^2\)In a system panic the OS enters an infinite loop and all further program execution is halted.
The system pool is used to allocate crucial data structures such as PCBs and process stacks. A process can also allocate memory from the system pool using alloc() (described above). However, it is also possible to allocate memory from an arbitrary pool using the system call s_alloc(). [35][34]

The configuration of pools is done in the configuration file appcon.con, as mentioned in section 4.2.1. It is possible to specify the total size of the pool, as well as the chunk sizes available and the ID of the pool. These pools are setup before the first process has started.

4.2.4 Signal Management

Signals in OSEck are used for passing messages between processes. Signals are allocated by using the system call alloc()[35]. This creates a buffer containing both the meta-data for the signal and its payload (see figure 4.6).
Every process has a list of pending signals. A signal can be sent between two processes using the `send()` system call. This system call links the signal onto the process’ signal list. A process accepts and receives a signal using `receive()`. When `receive()` executes it checks if there are any signals available in the process’ signal list. If there is a signal available, the signal is unlinked from the list and a pointer to its payload is returned. If no signal is available, the OS makes a context switch to allow other processes to continue execution.

In figure 4.7 there is an example of how the system operates when sending signals.

**Figure 4.7: Example of the signal system in action.**
4.3 Linking and Bootstrapping the System

So far, the mechanisms in OSE\textsubscript{ck} have been explained at a rather high level. However, some more detail is required about the build system to fully understand the later chapters. This section will primarily discuss booting and linking OSE\textsubscript{ck}\textsuperscript{3}.

4.3.1 Linking OSE\textsubscript{ck}

In the previous section general OS configuration was explained, but some details were intentionally left out. These details concern the linking of the OS. In practice, most embedded systems have a different memory layout and will require a customized memory mapping of the OS. A basic requirement for an OS to function correctly is to have the code and data located at their appropriate memory locations\textsuperscript{4}. The linker is responsible for mapping out the location of programs in memory. It is configured by a linker script. This script typically begins by defining a number of memory regions with permission information (read, write, execute). Then the different sections\textsuperscript{5} of the object files are placed into the defined memory regions.[36]

Below is an example of a link script file:

\footnotesize
\begin{verbatim}
This section can be skimmed through first, and is intended mostly to be a reference in later chapters.
\textsuperscript{4}E.g., not having mutable variables stored in ROM.
\textsuperscript{5}Sections contain different parts of the program. For example, .text contains the code and .bss contains non-initialized variables.
\end{verbatim}
4.3. Linking and Bootstrapping the System

/* This is a comment */
MEMORY
{
    /* Setup two memory areas: "rom" and "ram" */
    rom (rx) : ORIGIN = 0x00000000, LENGTH = 256K
    ram (rw) : ORIGIN = 0x20000000, LENGTH = 128K
}

/* Set program entry point to symbol ‘main’ */
ENTRY(main)

SECTIONS
{
    /* Create a new section called .text.
    * Put all (*) .text sections from all object files
    * passed to the linker into this new section .text.
    * Put the new .text section in memory area "rom".
    */
    .text : { *(.text) } > rom

    /* Same for .data. Put .data into "ram" */
    .data : { *(.data) } > ram

    /* Same for .bss. Put .bss into "ram" */
    .bss : { *(.bss) } > ram
}

Figure 4.8: Result of link script example.

The result of running this link script file together with two input object files are shown in figure 4.8.

In most cases the link script will be more advanced than this. The rest of this subsection presents some advanced link script techniques. The next section shows how these techniques are applied to an OSEck system.
Linking: Defining Symbols

It is possible to define symbols in the link script. These symbols contain the address of a certain memory location, such as the beginning or end of a particular section. A C-program can access these addresses by declaring the symbol defined in the linker script as an `extern` variable. The symbol then behaves as a reference to the specified address, which means that if the symbol is referenced (by using the address-of operator, `&`), the memory address assigned to the symbol in the link script will be returned.[36]

Linking: Load Address Vs. Virtual Address

There is a problem with directly loading all data into RAM from the off-chip programmer. If the system is restarted, all the data in RAM will be destroyed and the system will have to be reprogrammed. Therefore all the initial values of the data segment are kept in non-volatile memory and are copied into RAM memory at bootstrap. This means the data needs to have two addresses: a load address (ROM) and a virtual address\(^6\) (RAM). The load address is used by the bootstrap code to tell where the initial values are found, and the virtual address to tell where the data should be copied to. The load address is never used except in the bootstrap code. In normal programs, it is assumed the bootstrap code has copied the data to its correct virtual address.[36]

4.3.2 Bootstrapping the OS

The bootstrap routine in OSE\(_{ck}\) uses both of the two previously mentioned link script techniques for initializing the volatile memory of the system.

The routine starts off by setting the stack pointer to a valid address. The initial address of the stack pointer is defined as a symbol in the link script. An initial stack frame must be allocated for stack operations to function properly. After the stack is initialized, it will be possible to call and return from functions.

Next the memory is initialized: data needs to be copied from the flash memory to RAM. This is done by having three symbols defined in the link script: two symbols for indicating start and stop of the load address, and one symbol for telling the start of the virtual address. The second part of memory initialization is to zero the memory of the .bss section. Two symbols defined in the linker script are used to indicate the start and stop address of this section.

---

\(^6\)The term “virtual address”, or \(VMA\), is used by the GNU LD manual, but is not necessarily related to virtual address translation performed by an MMU. The VMA can just as well be a 1:1 mapped physical address.
4.3. Linking and Bootstrapping the System
Chapter 5

Updating OS Structures

To make OSE\textsubscript{ck} compatible with the requirements in the AutoSAR OS specification, a concept of memory protection must be introduced to the OS. OSE\textsubscript{ck} was originally not designed for processes running with memory protection, adding such a feature to the OS will affect most of the subsystems. This section will outline what design choices were made when updating the core of the OS to support memory protection mechanisms. For an overview of the context of this chapter as it relates to AutoSAR, see figure 5.1.

5.1 Introduction

As mentioned in previous chapters, OSE\textsubscript{ck} has many features making it an ideal choice for implementing the OSEK and AutoSAR layers on top of it, compared with implementing it on top of OSE. However, there are some features missing in OSE\textsubscript{ck} that must be added for making it compatible with the AutoSAR specification. One of these missing features are memory protection facilities.
When OSE\textsubscript{ck} was designed it was decided that no memory protection would be included since the focus of the OS was performance.

All the changes made to the OS should have a high portability factor. However, this thesis will only focus on functionality that can be implemented using the PowerPC Book-E architecture, which is the target architecture the AutoSAR layer should be running on.

As memory protection facilities are going to be introduced to OSE\textsubscript{ck}, an in-depth analysis of the subsystems of the OS is needed. This analysis will be used to determine which parts of the OS need to be modified. The following sections detail this analysis. First, a general discussion on the changes to the OS will be made. Then all the identified changes will be discussed in more detail in their own sections.

5.2 Overview

It may seem like a very simple goal to add memory protection to an OS; a user process should not be able to write data to another process’ memory and vice versa. Simple as this goal may seem, it requires the cooperation of many subsystems to function properly. Furthermore, the OS API should still behave as before introducing memory protection.

One way of identifying what has to be done for the introduction of memory protection in OSE\textsubscript{ck} is to make a dependency graph. By doing so, it is easier to plan how the memory protection facilities should be implemented. This dependency graph can be seen in figure 5.2.

![Dependency graph for OSE\textsubscript{ck} implementation.](image)

After analyzing the dependencies of the memory protection functionality, the following plan of implementation was made:

**User/Kernel-mode division:** A fundamental feature required for memory protection is the introduction of a privileged and non-privileged process concept in OSE\textsubscript{ck}. The processes in OSE\textsubscript{ck} are all currently running in kernel-mode, and all system calls are just ordinary function calls. To introduce user-mode processes in OSE\textsubscript{ck}, a trap interface must be intro-
Updating OS Structures

duced. The trap interface is not dependent on any other functionality, which makes it a good place to start the implementation.

**MMU/MPU support:** The next step is to implement support for the on-chip MMU. This allows the OS to divide memory into regions which can be used to protect processes from accessing each others memory. However, the kernel code and the user process code are right now mixed with each other. The kernel and the user processes needs to be put into different memory regions for the MMU/MPU functionality to work correctly. The different memory regions need to be represented in a kernel data structure so they can be programmed into the MMU/MPU.

**Updating the OS structures:** When the above listed features are implemented, all the fundamental parts of a working memory protection mechanism are in place. However, the OS needs to be aware of these features and make use of them. Further, some data structures may have to be updated to better fit the memory protection concept.

With these changes made, the OS supports enough memory protection mechanisms to be compatible with the AutoSAR standard.

In the following sections, the above listed items will be discussed in greater detail.

### 5.3 User-Mode Processes

One of the most basic mechanisms for making memory protection available in an OS is to divide up processes into the two categories: privileged and non-privileged. The privileged processes are allowed to execute any kind of instruction, while the non-privileged processes are only allowed to a subset of the total instruction set.

This division is made possible by using the hardware capabilities of the PowerPC architecture: There is a special processor instruction, `sc`, that can be used to enter kernel-mode from user-mode. When `sc` is executed, the CPU program counter jumps to a pre-specified exception handler and starts executing in kernel-mode[37]. From the exception handler, the selected system call can be executed with kernel-mode privileges. Finally, when the system call has finished, the CPU program counter is restored and the processor goes back to executing in user-mode.

To make it possible to execute system calls from user-mode in a controlled fashion, a trap interface must be designed. Enea has made the following demands on the trap interface implementation:

1. It should be possible to remove the trap interface if the target architecture does not support it or if all programs are supposed to be running in kernel-mode.

2. No more system calls than necessary should be linked in into the final executable. For example, if no process is using the system call `alloc()`, this call should not be linked into the executable.

3. The header file declaring the OS API should be kept unmodified.

---

1For a theoretical introduction, please see Section 2.1.1 Kernel-mode and User-mode
5.3.1 Implementation Feasibility Study

The design of a trap interface is a non-trivial task in two ways. First, a general design must be chosen. This design will decide how parameters should be passed to the trap interface and how they should be interpreted. Secondly, the chosen design must be integrated into the OS in a seamless way. The design should not make extensive modifications of the current OS API and should also be efficient.

General Design

There are many GPOSs running on the PowerPC architecture with support for a trap interface such as Linux and FreeBSD. The same thing applies also for RTOSs, such as OSE and Trampoline. To be able to select an implementation that is suitable for OSEck, it might be good to investigate the already existing implementations of trap interfaces in other GPOSs and RTOSs. Also, as the academic world seems to have ignored general trap interface design, there are no other sources of information.

Looking at existing implementations and by thinking about alternative solutions, there have been four general approaches identified for how a trap interface can be designed:

**Static table:** With the static table approach, there is a predefined system call table which contains all the addresses to the system calls. The kernel selects what system call to execute by using an index passed in a register before the execution of the sc instruction. This approach is easy to manage, but it requires that all the system calls are linked into the kernel (because all system calls are always referenced at least once). This approach was found in all of previously mentioned OSs. For general purpose OSs, this is probably the most suitable way of providing a trap interface.

For a monolithic RTOS, this approach might give bad link-time properties, as it requires all of the system calls to always be linked together with the executable – regardless if they were used by an application or not\(^2\).

**Dynamic table:** The dynamic table approach is when all system calls are installed in the system call table at run-time. This makes it possible to choose those system calls that should be available for a particular executable. Just like the static approach, an index is passed by the trap interface to indicate what system call to execute. When installing a system call, it could be done by either passing a predetermined index to the installer function (compile-time approach) or let the installer return an index for the function (run-time approach). The most efficient approach is to pass a predetermined index to the installer, as the index is given at compile-time. However, if the user installs very few system calls, it could result in a high fragmentation of the system call table. The run-time approach always has the most optimal space utilization regardless of the number of installed system calls and is thus not affected by this problem. A way of getting (almost) the same utilization with the compile-time approach is to sort the index numbers by their most frequent usage in

---

\(^2\)The reason for this is because all of the system calls are referenced at least once – in the system call table. The linker can therefore not omit any system calls as it believes all of the calls are used.
programs. By sorting the system calls by their usage frequency, it would also improve cache performance.

**On-demand:** An on-demand\(^3\) trap interface works by obtaining kernel-mode access exactly when it is needed. By defining the two functions *enter_kernel()* and *leave_kernel()*, it could be possible to enter and exit kernel-mode whenever this is requested. This approach can be implemented in two ways. The first way is to add a prologue and an epilogue to each system call for entering respectively exiting kernel-mode. The second way of implementing it would be to make a macro for each system call that wraps the two function calls around a system call. From an implementation perspective, such a mechanism is pretty straight-forward to implement, and would also result in a small (and thus maintainable) code base. Such an implementation would also result in no unused system calls being linked into the final executable. The downside with this implementation is that it is not really considered to be a true trap interface. Also, this functionality can be implemented as a special case of the other listed alternatives.

**Function pointer:** Finally, there is the function pointer approach which is a simplification of the table approaches. Instead of passing an index which should be used to look-up a function in a table, a direct pointer to the function is passed instead. The passed function is then executed in kernel-mode. This approach would, at least in a general purpose OS, probably never be considered because of the easiness of executing malicious code with kernel-mode privileges. However, for an embedded system, this approach might be acceptable. Also, it has many positive features regarding its implementation. It is one of the most efficient implementations as it does not require a table look-up. Furthermore, it solves the linking problem in a way that is transparent to the end-user (this responsibility is passed to the linker). Unfortunately the possibility to execute arbitrary code is undesirable. One way of solving this would be to declare a region of memory as safe. Only if the function pointer is within the safe bounds the function is executed. However, it is still possible to send a “valid” function pointer that points to somewhere in the middle of a system call.

After investigating the various options for implementing a trap interface, two of the approaches were dismissed immediately: The static table and the on-demand approach. The static table approach was dismissed because of the impossibility of omitting unused system calls from the final executable. Even though this approach is very popular in general purpose OSs, it does not provide any direct advantage for OSE\(_{ck}\), with the exception of its simple implementation. The on-demand approach was dismissed mostly due to the fact that it is not really a trap interface - it is just a way of changing the execution mode at an arbitrary place in the code whenever it is demanded. In addition, the exact same functionality can be obtained by the other approaches (with just slightly more overhead).

\(^3\)The on-demand trap interface might not be considered a true trap interface as it does not automatically handle the transition between kernel-mode and user-mode, but it is listed for completeness.
5.3. User-Mode Processes

The dynamic table approach cannot, from an implementation point-of-view, be dismissed right away. Even though it requires some more configuration from the end-user and imposes extra start up time, it is still an alternative worth implementing and evaluating.

The function pointer alternative has not been found in any current real-time or general purpose OS. This could be considered as a sign of trouble, but on the other hand it might have been regarded as not secure enough. Even though safety is an issue, it is not as important as speed, size and simplicity. Therefore, this alternative can not be dismissed at this point. We will come back to the implementation of the trap interface in section 5.3.2.

Integration with OSEck

When integrating the trap interface with the rest of the kernel, it is important that the view to the user is the same as it was before the trap interface was introduced. Basically, all the system calls should go via the trap interface instead of just making the branch directly.

A simple way of doing this is to make a C-preprocessor macro that renames all of the system calls from e.g. syscall to trap_real_syscall. A system call will then be transformed by the preprocessor into a function call to the trap routine for the system call. It is also possible to do it the other way around: Rename all system call functions internally and instead let the trap functions take the name of its system call. The only difference between these two methods is that the first method mangles the names if a user process should be in user-mode, while the second method mangles the names if a process should be run in kernel-mode.

Figure 5.3: Example of how a weak alias works.

Another way of making the view to the user the same as before is to solve it in link-time. By declaring every system call as a weak alias to the trap function, it can be decided in link-time whether to use the trap interface or
not. A simplified example of how this technique could be applied is shown in fig 5.3. Even though this solution is very elegant from a user perspective (as it is completely transparent), it is also a very compiler dependent solution which in the end makes it an unsuitable alternative of implementation. Therefore, the previous alternative will be how trap functions and system calls will correlate to each other as it is more portable.

5.3.2 Implementation of the Trap Interface

Before looking at the details concerning their respective implementation there will be some general information regarding the function calling conventions on the PowerPC.

Following PowerPC Calling Conventions

For every processor family there exists a so called Application Binary Interface (ABI). The ABI is a set of architecture dependent conventions used for telling how low-level functionality should be implemented. Example of information stated in the ABI are how a stack frame should look, function calling conventions, but also more high-level information such as linking conventions and library interfacing. For this processor (PowerPC e200z6), the PowerPC e500 ABI will be used.

It is desirable for the trap interface to follow the function calling conventions of the PowerPC ABI. By doing so, a system call that is dispatched through the trap interface will look just the same as an ordinary function call to a system call: The system call can enter, exit and handle parameters in the same way it did as before.

![Standard stack frame for the E500ABI](image)

The layout of the stack frame is fundamental for the calling conventions in the PowerPC processor. For an illustration of the layout of a stack frame, please see figure 5.4. Every function is responsible for allocating and deallocating their own stack frame. [1]

In the stack frame, there are two things that are interesting when implementing the trap interface: the link register save word and the parameter save area. The link register save word is used to save the return address of a function call. The parameter save area is used to pass arguments to the called function.

The trap interface needs to make itself invisible to the system call and the caller. That means, the trap interface should not alter the parameters passed by
the caller or by changing the return address of a function in a way that breaks
the program flow. However, if the trap interface is completely passive, it will
return back to where it should, but with kernel-mode privileges. This is because
the return address is pointing back to the user code and not to the exception
driver where the kernel-mode privileges can be revoked. This problem can be
solved by inserting a stack frame in between the caller’s stack frame and the
system calls stack frame.

By inserting a new stack frame, the trap interface has taken on the respon-
sibility of passing the correct parameters to the system call. For most function
calls, this is not a problem, but when the number of arguments gets high, the
stack frame must be adapted to it. Passing parameters using only registers can
only be done for up to 8 arguments. If 9 or more parameters needs to be passed,
the new stack frame needs to copy the excessive parameters from the old stack
frame to the new stack frame. An illustration of the two types of stack frames
can be seen in figure 5.5. [1]

As seen in the illustration of the two stack frames, there are two more pieces
of information that are saved on the exception handler’s stack frame: Return
address and old machine state register. Both of these values are produced by
the system call instruction and are required for the CPU and the program flow
to be restored to the old state before the sc instruction was executed.

Implementation 1 - Dynamic Table

The first implementation that was carried out was the dynamic table imple-
mentation. For implementing the dynamic table approach into the kernel, the
following (simplified) steps must be taken:

Table API: There needs to be an API designed for installing trap functions
both statically and dynamically. The API needs an initialization function
that sets all unused trap numbers to point to a dummy function. If not,
the program flow could end up anywhere if the user tries to access a call number that is not installed. The size of the table can be as low as $46 \cdot 4$ bytes. This does not take into account the ability to install debugging system calls. Further, if AutoSAR should be able to install system calls, the table size would also need to increase. The minimal table size can be given from the following information: 46 system calls, 37 debug system calls, 47 AutoSAR RTE calls. That is, the minimal size of the table can be 130 entries.

**Stub functions:** For every system call there needs to be a stub function. The basic idea is that each stub function will insert the index associated with the system call into a register. When going to kernel mode, the exception handler can look at the content of the register and use it for the table look-up. Each system call will have their own stub function, but there will also need to be different types of stub functions. This has to do with the function calling conventions as described in the previous section. In OSEck, the largest number of parameters in a system call goes up to 11. Therefore, there needs to be a stub function with the ability to transfer 3 extra parameters to its stack frame (because up to 8 parameters can be passed using only registers).

**Exception handler:** When the exception handler executes, the CPU is running in kernel-mode. The exception handler should use the index passed from the stub function to do the table look-up. The found function is then dispatched. When the function has finished, the exception handler restores the system to the old state to before the exception was invoked.

**Installation:** A start-up function needs to be written for installing the system calls into the table. This must be done before any user process is started.

The results of the implementation can be seen in section 8.2.1.

**Implementation 2 - Function Pointer**

The second design to implement is the function pointer design. It is a bit simpler implementation than the dynamic table design, but have a few other problems that must be compensated for. The following steps were made when implementing the design:

**Stub functions:** All system calls will need a stub function that puts the address of the system call into a register. The implementation of the stub function is almost exactly the same as for the dynamic table implementation, but instead an address is passed instead of an index.

**Exception handler:** The exception handler code can be made a little bit simpler than in the dynamic table implementation. There is no need to make a table look-up for example. However, there needs to be a validation check to see if the function pointer is valid.

Only these two subsystems are required for the implementation of the function pointer design. No extra installation is needed to be done because all information regarding the system call is given in the stub function.
For the function pointer implementation to work correctly, the linker will need to be able to tell if a function should be linked into the executable or not. Normally, the linker looks at one object file at a time and decides whether to include it or not. That would require that each system call stub was located in its own separate file. Such a solution would work, but would not be very elegant. A way to keep the stubs in the same file and still allow the linker to filter away the unused stubs is to put each stub in a separate section in the object file.

To make the function pointer design a bit safer, the validation of the function pointer can be improved. The first way of validating the function pointer is to see if it points to somewhere inside the kernel. However, this does not guarantee the function pointer to point to somewhere within the function (and not at the entry point). A way of improving this would be to declare an area of memory only containing jump instructions to the system call entry points as safe. By doing this, the system can ensure the system call jumps to the start of a valid system call. Implementing such functionality would however make the implementation a static table approach (as all system calls are referenced), and thereby not be able to provide the good linkage features of a function pointer approach. Therefore, this protection scheme is not implemented. The results of the implementation can be seen in section 8.2.1.

Optimizations

Because it is expensive to change the execution mode, kernel-mode should only be entered when it is necessary. A check of the current execution mode can be performed in two ways:

**Run-time:** This is the most straightforward way. A check can be made in run-time to see if the execution mode is in kernel-mode or user-mode.

  If the processor is executing in kernel-mode, the function can be called directly. If not, it must be dispatched by the exception handler.

**Compile-time:** A compile-time solution would assume that some code always is executed in kernel-mode, making it unnecessary to go through the trap interface at all. An example of such code is for example the system calls. It is not uncommon that a system call uses other system calls to perform a certain task. When executing a system call, a call to another system call should behave the same as an ordinary function call (i.e. without going through the trap interface). However, if the code is considered outside the system call code, the dispatching of a system call must be made by the exception handler.

These two listed alternatives are not mutually exclusive. Because there will be a mix of processes running in both user-mode and kernel-mode, a run-time check can be made to tell the execution mode of the current running process. The compile-time approach is perfect to apply to code only running in kernel-mode, such as system calls. Both of the designs (dynamic table and function pointer) identified in the previous section can be well adapted to this solution.

Unfortunately, there were some practical difficulties with implementing a run-time check. A process running in user-mode is not allowed to see what mode it is running in – such a check would result in a program instruction exception. For this check to work, the state of the process must be maintained by OSEck for example in a dedicated general purpose register.
Changes in mkconfig

Finally, to specify if a process should run in user-mode or kernel-mode, a preference for this option must be added to the configuration program mkconfig.

With these alternations made, the OS has support for running processes both in user-mode and kernel-mode.

5.4 Adding MMU/MPU-support

The next step for introducing memory protection to OSEck is to implement support for the unit that primarily makes the protection of memory possible - the MMU. The MMU operates on fixed-sized pages by applying different access rights to them. Because of this fixed-sized division, code and data from different programs need to be separated from each other for the memory protection to function.

On the PowerPC processor for which this development was carried out there is an on-chip MMU. Some of the features present in the MMU are:

1. Highly integrated with the CPU - there exist a number of instructions designated for manipulating the state of the MMU.
2. There are 32 entries in the TLB. This enables address translations within 32 different memory regions to be made without making any change in the TLB.
3. Nine different page sizes ranging from 4kB to 256MB.
4. Memory protection for up to 255 processes.
5. No predefined page table format exists.

For an MMU on an embedded system, these specifications are rather high-end. Typically, the number of entries on the TLB is usually much lower than 32 and there is no support for virtual to physical address translation. Because OSEck should support such low-end systems, the memory model chosen should be adaptable to these.

5.4.1 Design

When making a design for a memory protection scheme, it is important that the scheme can be deployed on multiple platforms. The following decisions must be made:

Memory model: Decide whether the OS should deploy virtual memory or not.
Page table structure: Select the representation of the page table.
Page replacement algorithm: A scheme for how the pages in the TLB should be replaced by the OS.

\[4\] For more information about what a MMU is, please see section 2.3.4
5.4. Adding MMU/MPU-support

Memory Model

The first thing to decide when choosing a memory model for a system is whether the system should have support for virtual address translation or should solely divide the physical memory into pages with different access rights. The first option corresponds to MMU functionality, while the second option to MPU functionality. For an embedded system with statically linked in processes the best option is probably the MPU functionality. The choice is based on the following criteria:

1. MPU functionality can be introduced to the OS without making any changes to the process setup model. MMU functionality would require the programs to be loaded into memory at run-time, which would in turn introduce significant overhead to the OS.

2. MPU functionality is a subset of MMU functionality. To support systems not capable of making address translations, it is better to have a MPU memory model.

Page Table Structure

For representing the different memory regions and what access rights they have, a page table data structure must be introduced. The page table should have the following two features:

**Easy to configure:** As there exist no tool for automatically creating the page table, it must be configured “by hand”. This requires the page table format to be simple for a human to understand.

**Fast searches:** Searches in the page table should be fast to perform.

Because of the first listed feature, having something else than a simple array in memory could be confusing for the person configuring the page table. For each memory mapping, a new element is added to the page table. How fast searches can be achieved is described in section 5.4.2 Optimizations.

Page Replacement Algorithm

Most common page replacement algorithms work by replacing pages that have not been accessed for the longest time. This is normally indicated by a so called dirty-bit to tell whether a page has been accessed or not. In the PowerPC Book-E case, the dirty bit must be set in software by the user. This is done by inserting all the pages to the TLB, but without their respective permissions. When a process tries to access data within a page, an access exception will occur. The page table will then be searched, and the correct permissions for the page will be obtained. At the same time, a dirty bit can be recorded for the page. [38]

The presence of a dirty bit might introduce some unwanted overhead into the system. Also, it might not be possible for other architectures to support such a scheme. However, an efficient on-demand page replacement algorithm requires the presence of a dirty-bit.
Instead of implementing an on-demand page replacement algorithm, pages are replaced every time there is a context switch. For each process, there is a number of pages associated with that process that are inserted. This method does not use the capabilities of the target MMU to its full extent, but the method is compliant with MMUs with less TLB entries.

### 5.4.2 Implementation

This section describes the changes that would have to be made to the kernel code and to the build system.

#### Adding Support for MMU

Several functions for accessing and manipulating the MMU was implemented to support the features listed in the reference documentation. Such features included: Inserting pages, removing pages and searching for pages.

A start routine for inserting all the kernel pages into memory was written. There is no need to write an assembler-only bootstrap routine for inserting the basic kernel pages\(^5\)[39]. The MMU is reprogrammed with the kernel pages after the initial bootstrap routine (which handles the setup of a stack).

#### Dividing Tasks

Traditionally in OSE\(_{ck}\), the code and data of the kernel and the processes have been intermixed. For the hardware to be able to apply memory protection, the kernel and the processes needs to be separated into their own memory regions. This is done manually in the linker script. It requires all the processes that should belong together to be compiled into separate object files.

Pools are used to do stack and memory allocation for both the kernel and the user processes. Because of this, pools too also need to be separated for memory protection to function properly. A pool is defined in OSE\(_{ck}\) by editing the mkconfig configuration file. After parsing this configuration file, the pool is defined in the resulting C file as an ordinary uninitialized array assigned to a specific link section. To have the ability to separate a pool from the rest of the variables, each pool is defined in a separate section. By doing this, it is possible in the linker script to assign the pools to the correct memory region.

#### Integrating Page Table with Linker Script

When the linker script has been configured for separating the processes, the page table needs to setup the memory regions according to this configuration. For every memory region, two symbols representing the start and the stop of a region is defined in the linker script. The address to these symbols, which are set at link-time, can be used in the initialization of the page table.

\(^5\)This is thanks to the BAM (Boot Assist Module) on the MPC5516. It sets up enough memory regions to support both the use of code and a stack at the same time. If the BAM was not present, the bootstrap code would not be allowed to use the stack, and would have had 4KB of memory to make all initial MMU initialization.
New Bootstrap Mechanism

The bootstrap mechanism is among other things responsible for initialization of the .data and .bss section. As the memory has been divided up into several regions, the bootstrap mechanism must support initialization of multiple .data and .bss sections. This is done by constructing two tables which the bootstrap mechanism can use:

.data initialization table: This table contains information relevant for initialization of the .data segments. Every row in the table contains three pieces of information: the source address, the destination address and the size to copy.

.bss initialization table: The counterpart for the .bss initialization. Every row in the table contains two pieces of information: the start address and the size to clear.

Both of these tables are defined in linker script.

Optimizations

Searches in the page table could be optimized by dividing the pages into two types of pages: pages that are always present in the MMU, and pages that are possible to replace. By having a pointer to the start of the pages that are possible to replace, the search time can be optimized. This is because the invalidation protected pages are not necessary to include in a search when doing a context switch, and thereby replacing the old pages, as these pages are already present in the MMU. This can also improve the time spent searching for a free entry in the TLB.

5.5 Updating the OSEck Structures

For OSEck to make the most out of the newly added memory protection mechanisms, some of its internal data structures and system calls must be modified to support this. The following subsystems will be changed:

Process management: A new concept must be introduced into OSEck for handling processes active in the same of memory region.

Pool management: Pools should be owned by processes (or a group of processes) and be located within special regions of memory. Processes should be associated with a pool.

Signal management: Because OSEck has earlier used the lack of memory protection as a feature in the old signal system, a revised signal sending concept must be designed.

5.5.1 Process Management

For OSEck to handle multiple processes within the same memory protection region, a new concept needs to be introduced into the OS. This new concept is called a block. A block can group several processes together and apply the same
memory protection scheme for them all. This is preferable if processes should be able to share memory with each other. The block concept was implemented as an extra integer member in the PCB structure. However, this concept can be made much more advanced.

To group processes into blocks, an extra option was added in mkconfig. The block number is used by the MMU to set the \textit{PID0} register which is used to track the currently running process.

5.5.2 Pool Management

In OSE\textsubscript{ck}, the information regarding the pool is saved at the start of the pool. Because it is possible for a user to corrupt this data, the meta-data for the pool is moved into kernel-space instead. This is done by saving the meta-data in the system pool instead. By doing this, only system calls can access and alter the state of the pool.

All pool areas are assigned to special sections so they can be placed in the correct memory regions by the linker.

5.5.3 Signal Management

The signal management system needs to be revised in a similar way as the pool management. The signal is, exactly as for pools, saved before the payload of the signal. To prevent a user from accidentally overwriting the meta-data, it must be located in the kernel's memory protection region. More precisely, the meta-data of the signal is allocated from the system pool, while the payload is allocated from the process' associated pool. It is not however possible to completely separate meta-data from data for signals. Still, the data will need to know what meta-data it is assigned to. To keep this association, a pointer to the meta-data is saved before the payload of the signal. Even though it is possible to alter the address of this pointer, it is easy to verify that the pointer is valid. This is checked by testing if the meta-data indicated by the payload is pointing back to the payload. If it is not, the signal is invalid. Such a check could also trigger the execution of a memory protection exception as it is not possible to know whether the address pointed out by the signal header is in a different memory region or not. Another approach would be to use the address of the signal to look-up the correct signal administration block. However, to make this look-up quick, it would require an auxiliary dynamic data structure of some sort (such as a hash table). As this would add far more complexity to the current signal management system than the first alternative, this alternative is discarded.

Another change regarding the signal management is how signals are sent between processes. As it is not possible for a process in one memory protection area to access a signal allocated by a process operating in another memory protection area, a copy to the receiving process' area must be made.

\footnote{Many system calls take the user part of a signal as argument. These system calls need to have a method of looking up the meta-data for a signal looking only at the user part of a signal.}
5.5.4 An Example of Signal Management

To get a better picture of the modifications to OSE\textsubscript{ck}, an example is shown in figure 5.6 and 5.7. These two pictures shows an example of IPC between two processes (by using signals). A similar picture which shows the same scenario, but with the old signal management protocol, can be seen in figure 4.7.

Figure 5.6: Sequence diagram of the signal new signal system.
Figure 5.7: Sequence diagram of the signal new signal system continued.

1. Process 1 sees a new signal on its signal list. The signal is unlinked and processed.
2. Process 1 allocates a new signal, signal 2, that is accessible from the memory space of process 1 (when running in user mode). The signal is allocated just as in (3), except this time the signal is allocated from pool 1.
3. The payload of signal 1 is copied to the payload of signal 2.
4. As process 2 has lost ownership of signal 1 and the signal is not needed any more, the signal is freed. This is done by linking the signal to the free list of the pool which the payload was allocated from. It is then possible to reuse the signal in a future alloc() call.
5. A reference to signal 2’s payload is returned.
5.5. Updating the $\text{OSE}_{\text{ck}}$ Structures
Chapter 6

Turning AutoSAR OS Requirements into Design

This chapter will document the results of our second problem area as described in detail in section 1.2.2 on page 2, namely creating a mapping between AutoSAR requirements and OSEK. The first part of this problem area was to get an overview of the AutoSAR architecture and the system in its original state. To find what AutoSAR requires of a memory protection scheme, two primary documents – Specification of Operating System[11] and Requirements on Operating System[31] – were analyzed to identify requirements and implementation constraints. These were compared to the existing operating system and OSEK implementation to identify which services and functionality were missing. This analysis work resulted in a number of design choices which will be presented below. The context of this work, and it’s architectural place compare to OSEK and the rest of the AutoSAR architecture can be seen in figure 6.1.

Section 6.1 and 6.2 will present the analysis work and the data types and functions directly specified by the AutoSAR OS specification. Then section 6.3 will discuss the new data structures created to keep track of the new application concept. The following section presents the additions to the OIL language and Enea’s OIL parser, OILTool. The chapter concludes with two sections, the first about keeping tracking the active application at run-time and the second about an AutoSAR requirement that presented particular difficulty.
### 6.1 AutoSAR Requirements Analysis

A careful analysis of the specification documents was conducted to summarize the requirements that affected our project and constrained the final design. The analysis required several readings of the documents, as the language within the documents was bureaucratic and the organization of information occasionally awkward. A complete view of the system was difficult to achieve in a single reading. This work resulted in a list of applicable requirements and API features which pertained to the memory protections features that were to be implemented.

### 6.2 The OS API and AutoSAR Data Types

The AutoSAR OS standards specify an API so that all the functionality introduced – e.g., memory protection – can be accessed in a simple and standardized way. The data types defined by the API and their constants can be seen in table 6.1. All the data types correspond to a feature – `ApplicationType` and `ISRType` – or to the usage of new features – e.g., `MemoryStartAdressType` and `MemorySizeType` to check memory access rights.

There are also eight services specified in the API. They are enumerated in table 6.2. Here again we see a correlation to new concepts (applications and memory protection). The services pertaining to applications require a well-ordered collection of data about the applications and their members to be created. These data structures are discussed in the next section.

#### Table 6.1: Data types directly specified by AutoSAR.

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationType</td>
<td>INVALID_OSAPPLICATION</td>
</tr>
<tr>
<td>TrustedFunctionIndexType</td>
<td>N/A</td>
</tr>
<tr>
<td>TrustedFunctionParameterRefType</td>
<td>N/A</td>
</tr>
<tr>
<td>AccessType</td>
<td>N/A</td>
</tr>
<tr>
<td>ObjectAccessType</td>
<td>ACCESS</td>
</tr>
<tr>
<td>ObjectTypeType</td>
<td>OBJECT</td>
</tr>
<tr>
<td></td>
<td>ISR</td>
</tr>
<tr>
<td></td>
<td>OBJECT_ALARM</td>
</tr>
<tr>
<td></td>
<td>OBJECT_RESOURCE</td>
</tr>
<tr>
<td></td>
<td>OBJECT_COUNTER</td>
</tr>
<tr>
<td>MemoryStartAdressType</td>
<td>N/A</td>
</tr>
<tr>
<td>MemorySizeType</td>
<td>N/A</td>
</tr>
<tr>
<td>ISRType</td>
<td>INVALID_ISR</td>
</tr>
<tr>
<td>RestartType</td>
<td>RESTART</td>
</tr>
<tr>
<td></td>
<td>NO_RESTART</td>
</tr>
</tbody>
</table>

The AutoSAR OS standards specify an API so that all the functionality introduced – e.g., memory protection – can be accessed in a simple and standardized way. The data types defined by the API and their constants can be seen in table 6.1. All the data types correspond to a feature – `ApplicationType` and `ISRType` – or to the usage of new features – e.g., `MemoryStartAdressType` and `MemorySizeType` to check memory access rights.

There are also eight services specified in the API. They are enumerated in table 6.2. Here again we see a correlation to new concepts (applications and memory protection). The services pertaining to applications require a well-ordered collection of data about the applications and their members to be created. These data structures are discussed in the next section.
### 6.3 New Data Structures

The introduction of the application concept requires the creation of at least two new data structures. An application itself has no specific code and these data structures effectively are the applications. One structure is needed to keep track of the object memberships, and another to keep track of `ACCESSING_APPLICATIONS` — that is to say applications that are granted access to a particular object belonging to another application. These two can be combined. However, it was decided that the added overhead in getting the `ACCESSING_APPLICATIONS` information out of the data structure was prohibitive when the two groups of information would generally be accessed separately. The membership information is used primarily for the `CheckObjectOwnership` service, and the `ACCESSING_APPLICATIONS` information for the `CheckObjectAccess` service.

The application membership data structure needs to keep track of each set of objects of a given type for a given application. For example, an application may have three tasks and a counter and alarm for each. Since the number of object types and applications are constant at configuration time, a two-dimensional array would be appropriate to keep track of the member lists. The member objects are then enumerated in a linked list, see figure 6.2. A linked list is a data structure in which the objects are arranged in a linear order, where the order is determined by a pointer in each object[40]. Searching a linked list takes $O(n)$ time[40]. In this case $n$ is rarely larger than 16 or 32, so this is quite acceptable.

---

#### Table 6.2: API services directly specified by AutoSAR.

<table>
<thead>
<tr>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetApplicationID</td>
</tr>
<tr>
<td>GetISRID</td>
</tr>
<tr>
<td>CallTrustedFunction</td>
</tr>
<tr>
<td>CheckISRMemoryAccess</td>
</tr>
<tr>
<td>CheckTaskMemoryAccess</td>
</tr>
<tr>
<td>CheckObjectAccess</td>
</tr>
<tr>
<td>CheckObjectOwnership</td>
</tr>
<tr>
<td>TerminateApplication</td>
</tr>
</tbody>
</table>

---

Figure 6.2: Example of the Application data-structure.
A simplified approach was used with the second data structure. It’s purpose is to keep track of the access rights an object may grant other applications. Here a simple array was created for each object type, with an entry for each object of that type. This entry consists of another array of applications which are granted access. An example is shown in figure 6.3. There is one of these structures for each object type, with each object of the type having one sub-array.

Figure 6.3: Example of the Accessing Applications data-structure.

6.4 Updating OIL and Its Tools

The OIL specification as prescribed by OSEK/VDX is at version 2.5. As AutoSAR is built upon the OSEK OS standard, the configuration language is also extended. The grammar needed only the minor adjustment of adding a single key-word (APPLICATION). However, AutoSAR also specifies that the parser is to validate the data in the configuration file, something not required by OSEK. The base OIL file which contains the implementation section, used to specify which objects and options are available, also needed a major update. The changes to the OIL grammar were to add the APPLICATION token, and the APPLICATION_TYPE symbol to tie to the object definition – like Identifier is tied to sum in tab. 3.2.

6.4.1 Updating OILTool

The OSEK Implementation Language (OIL) specifies that a system configuration is to be written in two parts. The first part defines what objects are available and what options they contain. This will be discussed more in the next section. The other part contains the actual system configuration. Since the OIL file itself describes what is considered valid contents, there is little effort required by OILTool to assure more than grammatical correctness. However, it is still required to validate more complex relationships in the input and generate output.

All input validation stems from the AutoSAR specifications. To aide in minimizing time spent searching for faults, the tools used with the AutoSAR system are to help find errors as early as possible. This is done by traversing the AST provided by the parser and checking the contents against a set of rules. For example, there’s a check to ensure that if the Scalability Class isn’t set to include memory protection then there isn’t any option enabled that is part of the memory protection mechanisms. The exact verifications now performed by OILTool are outline in Appendix B.
6.4.2 Updating OIL Implementation Definition

As mentioned before, the OIL file used to configure a system is composed of two parts. As there is a mechanism to include the contents of another file within the OIL file, one generally has the *implementation* portion in a separate file, as it will only need to be updated on major changes in the hardware environment or OS. This file also needs to correctly detail which objects are declarable. The updating of this file required adding the `APPLICATION` object and adding appropriate options to several other objects. Appendix A contains the updated definition file.

6.4.3 Compilation of Scalability Class Information

AutoSAR specifies that as part of its scalability, there should be no redundant code compiled or linked with the system. This means that if the Scalability Class is set to SC1 or SC2, the system services relating to memory protection shouldn’t be compiled.

The first obvious way of preventing this compilation was through optional compilation as provided by the C pre-processor. OilTool generates a header file which defines the scalability class that the implementation files then can include and simply skip the relevant code. The advantage here is that it is very simple to implement and requires no user intervention. Unfortunately, it also means that the end user has to have the system source code to be able to compile everything from scratch. While this might not be a problem within the context of SWAP, it can be seen as less than ideal if the system is used with an actual client. In that scenario one would ideally deliver a pre-compiled library file.

A second option, which takes into account the possible desire to keep the source-code out of the end user’s hands, is to simply create the four possible library configurations – one for each scalability class. The actual creation of these libraries can be automated by the build system. Compilation time would be increased by a factor four, but the total build time would still be well under a minute. This solution requires that OilTool can somehow influence which library is linked with the user’s system.

The third option would be to create one library, but with any part that should only be optionally compiled as a separate object file. This way when something should be unusable and thusly isn’t referenced it isn’t linked. To ensure that this works a check at the start of each such optional function is required to make sure the scalability class defined includes the function. This would be done with a global constant, which can be linked, whereas the first option used `#define` statements which only have file scope and require the recompilation of the AutoSAR layer.

In the end it was decided to go with the second option, creating four separate libraries. This required the least amount of additional coding and is simple to implement.

6.4.4 Implementing ShutdownHook

There was no concept of shutdown in OS, hence no ShutdownHook. The StartupHooks are declared in appcon.con and then the start-up function is generated by mkconfig. For symmetry one might want to build a similar system for
58 6.5 Keeping Track of the Current Application

Keeping Track of the Current Application

There are several functions that the OS API is to provide that require knowledge of which application is currently running. An example is the processing of the ProtectionHook. One of the possible outcomes is to kill the current application. A more direct need is found in the GetApplicationID service call.

A simple way of doing this would be to get the current process ID and go through the various application’s member lists, and returning the result via GetApplicationID. However there is a problem with this approach. Doing a full run through of the member list is $O(n^2)$ in the worst case and something we’d rather do as infrequently as possible. Further, the fact that it runs so infrequently means there may be a problem. Since we cannot guarantee that the command was most recently called from the current Task or ISR we have no way of tracing what application is running.

The alternative is to implement a permanent swap-hook. This function, which resides in the OS, would on a context switch check which application owns the incoming process. This is stored in a global variable in the system’s protected memory region. The look-up is then $O(1)$. There is an overhead introduced into the context-switch when storing the global variable, however the use of applications is meant to be limited to the development phase of the software, and not intended to be used after final deployment. The overhead would not be incurred in the final system. It’s therefore reasonable to assume that this penalty is acceptable.

6.6 Requirement OS208

One AutoSAR requirement in particular seemed to changed the extent of the memory protection considerably. Requirement OS208 states: “The Operating System should prevent write access to the private stack of Tasks/Category 2 ISRs of a non-trusted application from all other tasks/ISRs in the same OS-Application.”[31] This single sentence increased the complexity of the required solution a lot.

If you protect only OS-Applications, then applying that mechanism is relatively simple. Each TLB entry – see section 2.3.4 – is marked with the applications ID number, and that number is associated with all member tasks and interrupts. This way, all internal access is allowed, but other applications are completely blocked from wrongfully accessing the memory. Hardware support is guaranteed as long as there is an MPU or MMU.

However, by making the task_ISR stack private from other members of the
same application, the memory protection region must be reprogrammed on a per-process basis, see figure 6.4. Where before there was overhead for switching between two OS-Applications, there is now overhead for all switching.

Refer to figure 6.4. Depicted is an application with three tasks, symbolized by the three darker boxes at the bottom of the main application box. The light-gray region is the applications shared memory for direct communication. When task one is running as in 6.4(a), the memory protection region which is stored in the TLB and page-table is as shown by the dotted line. However, on a context switch to task 2, the region changes to that shown in 6.4(b). This means that the TLB must be flushed of the entries private to task 1, and replace them with those private to task 2.

A context switch is now burdened with a look-up in the page table and the loading of at least one TLB entry. Depending on where all this information is stored, this could mean reading several bytes of information from the slowest memory in the system, i.e. external flash. A context switch may have gone from a handful of clock-cycles to one or more hundreds of clock cycles.

The added difficulty of this addition, along with the unavoidable degradation of performance mean that we must be very skeptical. The main goal with AutoSARs memory protection is the introduction of the application concept. Getting that in place and functioning is a higher priority than completeness of conformance. Furthermore, the AutoSAR standard stipulates that memory protection should scale with the hardware. There are features in more advanced MMUs that make this sort of division easier to work with.

It was discovered after a few weeks, by way of reading the next revision of the specification, that the language was ambiguous. The use of should could indicate may – as it is explicitly written in the later draft – rather than shall. That is to say, the implementation of OS208 is entirely optional, and we can therefore ignore it. This level of protection is equivalent to per-thread stack protection, which no modern OS supports today, much less an embedded RTOS.
Chapter 7

Integrating $\text{OSE}_{\text{ck}}$ and AutoSAR

This section will explain how the memory protection mechanisms in the AutoSAR OS specification were integrated with the new memory protection implementation in $\text{OSE}_{\text{ck}}$.

7.1 Overview of integration

To make the implemented AutoSAR functionality work as intended there needs to be some unification of the two implementations. This chapter will outline what adaptations were necessary to make AutoSAR run on top of $\text{OSE}_{\text{ck}}$, and what extra functionality AutoSAR required from the OS to function properly.

7.2 Adapting AutoSAR to $\text{OSE}_{\text{ck}}$

With the $\text{OSE}_{\text{ck}}$ OS specification a completely new memory protection model has been introduced. The OILTool and the test suites need to be updated to accommodate this.

7.2.1 Additional Information From OIL Files

The implementation of memory protection features in $\text{OSE}_{\text{ck}}$ required that certain $\text{OSE}_{\text{ck}}$ specific information had to be added to the OIL files. The way that $\text{OSE}_{\text{ck}}$ handles IPC coupled with the AutoSAR requirement of per application private memory meant that each application needed a private pool. The specification of APPLICATION objects in the OIL file were therefore given a POOLID attribute.

The output from OIL Tool was also updated to give more explicit information in the configuration file. The process that each task and category 2 ISR maps to was bound to both it’s application and the application’s pool. Tasks belonging to trusted applications were also identified as running in kernel-mode, while non-trusted ones were marked as running in user mode.
Declaration of Application Information
The output from OILTool has to among other things generate data-structures keeping track of application membership and application access rights when in scalability class SC3 or SC4. First a solution with malloc() and an initializer function called through the StartupHook was considered. However, it was later decided that since a fully AutoSAR-compliant OS is supposed to be entirely static we should also create these data structures statically. OILTool was updated with data structures to keep track of all this data and generate C code to declare the structures statically.

7.2.2 User-mode/Kernel-mode for AutoSAR
As the data structures AutoSAR operates on should be protected, all the AutoSAR API functions should be executed through the trap interface. Adding the API functions to the trap interface was at this point trivial, as the same techniques had been applied when implementing the trap interface for OSEck.

Further, all tasks for OSEck had to start their execution in kernel-mode. The reason for this is that before a task reaches its entry point, the AutoSAR API layer will have to set up some task specific information. When this information has been initialized, the process can begin the execution at its entry point. It is then up to the AutoSAR API layer to switch to user-mode before letting the process run.

7.2.3 Handling Protection Errors
The introduction of the ProtectionHook and its functionality meant that OSEck had to have some sort of stub in the interrupt vector table (IVT). The ProtectionHook was to be called on – among other things – processor exceptions such as division by zero. This meant that the interrupt vector for each machine instruction exception needed to directly or indirectly call the ProtectionHook routine.

7.2.4 Interrupt Handling
Category 2 ISRs are run with the same privileges as their owning applications. They are also allowed to use system services. To achieve this, there has to be some sort of hand-off. After an exception the processor is in kernel-mode. To ensure that memory protection and user-mode are both correctly enabled a stub is needed to enable user mode and set up a correct environment before the ISR process is swapped in. This stub is called OSInt().

OSInt() is installed - by OILTool - into the interrupt vectors associated with any category 2 ISRs. This installation is done via the appcon.con file and mkconfig. A list of ISR processes associated with these interrupts is also created, allowing OSInt() to dispatch the correct process following an interrupt.

7.2.5 New Test System
To test the added functionality new tests were needed in the existing test system. However, the old test system - originally used for testing the OSEK layer - was not compliant with the new memory protection model. It was also too limited
for the new tests that should be performed. Both of these factors led to a rewrite of the test system. The new test system is depicted in figure 7.1.

![Diagram of test system](image)

**Figure 7.1:** A rough sketch of the new test system.

This figure shows three processes which all reside in different memory regions. The master process is responsible for handling the execution of the tests and book keeping the results of the executed tests. For each test a start signal is sent to either task1 or task2. On reception of the start signal, the task starts executing the next test and sends back the result to the master process which logs the results in RAM. Because some tests will yield different results when executing in different applications, a test system with two slave tasks was designed instead of having just one.

There were some tests the test system was not capable of recover from, e.g. protection errors. This is because the test system was not written with a customized error hook to handle recovery from such fatal errors. Getting the test system to work for these cases is left as future work.

### 7.3 Implementing OSE\textsubscript{ck} features requested by AutoSAR

Most of the functionality requested by the AutoSAR OS standard was possible to implement without doing any further additions to the OS (other than introducing memory protection). However, there are some operations that the AutoSAR layer is incapable of handling, and needs to be addressed directly in the OS. In this section, these features will be presented.

#### 7.3.1 Killing/restarting Applications

Killing or restarting whole applications is a non-trivial task, because it often requires several processes to be restored to their initial state. Because OSE\textsubscript{ck} does not offer a way to restart a process, doing this in the AutoSAR layer is even trickier.

There were two steps taken to make this functionality work:

\[\text{Such as tests for } \text{GetApplicationID}().\]
1. A special system call \texttt{(restart\_process())} was written to restart a single process. However, this system call has a special prerequisite; to ensure correct functionality, this system call should not be executed by the process itself. It has to be executed by another process which doesn’t share the victim’s stack.

2. To fulfill the prerequisite, a new high-priority process called \texttt{autosar\_daemon} was introduced. By sending signals to this process about what application should be killed, it can safely reset all tasks, alarms, resources and counters belonging to the application without the risk of ending up in an invalid state.

This functionality also makes it possible to rewrite some OSEK API calls pertaining to task restart/termination to make them cleaner.

### 7.3.2 Page Table Operations

Some AutoSAR OS operations required information on memory protection access rights. To support this, basic page table search functionality was added in OSE\textsubscript{CK}. There was not much time spent designing the interface for page table searching. This is because the interface should be designed with other architectures in mind as well. However, studying all the architectures that would be subject to the memory protection facilities are beyond the scope of this report. It was therefore decided that the interface should have just enough functionality for the AutoSAR layer to work. A refinement of this interface is left as further work.
Chapter 8

Outcomes and Evaluation

This chapter will discuss the results of the project. Specifically, it will enumerate the deliverables created and the key concepts and measurements produced thereby.

8.1 Implementation of AutoSAR API

The actual contents of the API were only a small portion of the material that needed to be programmed. There were additions to the OIL translation tool, new data structures to design and generate, and functional testing to be done on the API components. To speed up work with OILTool and the system in general a GUI application was created to generate OIL files.

8.1.1 Addendum to OILTool

A large portion of the tangible work associated with implementing AutoSAR went into the additions to OILTool. Adding the OIL file validation to OILTool took most of the time. An list of what verifications the parser now applies to input files is found in Appendix B. The OIL Implementation definition was updated to accept the various types that are new in the AutoSAR architecture – this file with new additions marked is in appendix A.

Changes in how mkconfig expected its input were incorporated to facilitate the new features in OSEck. Applications also received a Pool attribute so as to tie together OSEck with the AutoSAR API layer.

8.1.2 OILGen

A simple GUI application was created to ease testing OILTool and simplify system configuration. Figure 8.1 shows the main screen of the program. It was written in C# using Visual Studio, and consists of a series of screens where configuration data can be entered using text areas and check boxes. The configurations binary representation can be saved to file to be brought up again at a later time. The system exports the data to a well formed OIL file.
8.1.3 Additional Data Structures

To evaluate the new data structures an interview was conducted with Mathias Bergvall who works at Enea. He is an experienced programmer of embedded systems and has worked extensively with OSE and OSE\textsubscript{ck}. Generally the structure seemed quite applicable for RTOS applications, being neither too large nor adding unnecessary overhead. The linked lists\textsuperscript{1} could be optimized to a static array, if there were to be a well-defined NULL value entry that could be placed at the end of the array. Also, a slight optimization would be to cache the owning application's ID with each task or ISR after the first look-up. Some concern was also mentioned about the possibility of a sparse matrix\textsuperscript{2} \cite{41}.

The sparsest configuration would be several applications with one task each. In this situation you would have five null-pointers for each application. With an absolute maximum of 15 applications due to restrictions in the number of pools available under OSE\textsubscript{ck}, this gives a maximum of 300 bytes of wasted memory. If we take a conservative estimate of 1 MB ROM memory - the hardware system used in this project had 2 MB - for a regular system, this represents 0.028\% of the total memory area. The data in this structure is also static, so it will never have to be copied into RAM - a much more restricted resource. This is a negligible amount of memory, and the static nature of all the variables left Mathias with very little concern about the viability of the data structure\cite{41}.

8.2 Extensions to OSE\textsubscript{ck}

There have been various changes made incorporated into OSE\textsubscript{ck} to make it support memory protection. This section will show the results of the implementation. At the end of this section, a presentation of the introduced overhead for each extension will be presented.

\textsuperscript{1}For a review of the data structure in question see section 6.3.

\textsuperscript{2}A sparse matrix is one with more empty cells than used, e.g. several applications with only tasks would leave most of the array empty.
### Outcomes and Evaluation

<table>
<thead>
<tr>
<th>Subsystem/Function</th>
<th>Number of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table API - Static installation</td>
<td>≈ 20</td>
</tr>
<tr>
<td>Table API - Dynamic installation</td>
<td>≈ 29</td>
</tr>
<tr>
<td>Stub function up to 8 params.</td>
<td>10</td>
</tr>
<tr>
<td>Stub function 9 params.</td>
<td>13</td>
</tr>
<tr>
<td>Stub function 10 params.</td>
<td>15</td>
</tr>
<tr>
<td>Stub function 11 params.</td>
<td>17</td>
</tr>
<tr>
<td>Exception handler</td>
<td>21</td>
</tr>
<tr>
<td>Initialization</td>
<td>≈ 15 + 23 · Number of system calls</td>
</tr>
<tr>
<td>Initialization, no debug calls</td>
<td>≈ 15 + 23 · 46 ≈ 1073</td>
</tr>
<tr>
<td>Initialization, with debug calls</td>
<td>≈ 15 + 23 · (46 + 37) ≈ 1924</td>
</tr>
<tr>
<td>Initialization, debug + AutoSAR API</td>
<td>≈ 15 + 23 · (46 + 37 + 47) ≈ 3005</td>
</tr>
</tbody>
</table>

Table 8.1: Number of instructions used in the dynamic table implementation.

<table>
<thead>
<tr>
<th>Subsystem/Function</th>
<th>Number of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stub function up to 8 params.</td>
<td>12</td>
</tr>
<tr>
<td>Stub function 9 params.</td>
<td>15</td>
</tr>
<tr>
<td>Stub function 10 params.</td>
<td>17</td>
</tr>
<tr>
<td>Stub function 11 params.</td>
<td>19</td>
</tr>
<tr>
<td>Exception handler</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 8.2: Number of instructions used in the function pointer implementation.

#### 8.2.1 Trap Interface

**Dynamic Table Implementation**

The result of the implementation was as expected: a high overhead was introduced in the start up of the OS (for installing the trap functions), but the stub functions are quite small.

Table 8.1 shows roughly how many instructions there are for each part of the different subsystems. Because the Table API and the installation routines are coded in C, counting the instructions a bit harder. However, the instruction count should be regarded as worst case.

There are two routines for installing a system call. The static installation routine installs a system call which has already been assigned an index. The dynamic installation routine searches the trap table for an empty index to install a system call. The routine then returns the index of where the system call was installed.

The initialization routine installs all the desired system calls into the trap table using the static installation routine.

**Function Pointer Implementation**

Table 8.2 shows how many instructions were used for the routines for the two subsystems. This method was coded completely in PowerPC assembler, and only involved writing system call stubs and the exception handler for system calls.
8.2. Extensions to OSECK

Overhead

Overhead is added to every system call which is being dispatched through the trap interface. When the number of instructions of a system call is high, the overhead added by the trap interface might not make much of a difference to the overall execution time. However, for a system call composed of very few instructions the overhead is much more noticeable, as will be shown.

The number of extra instructions added to a system call can be found by adding together the instruction count for exception handler and the stub function associated with the system call. Therefore, the overhead for a system call dispatched by the trap interface can mathematically be expressed as follows:

\[
\frac{\text{Stub function} + \text{Except. handler} + \text{System call}}{\text{System call}} - 1 = \text{Instruction overhead}
\]

The highest overhead can be identified by looking at the system call with the lowest instruction count. The system call with the lowest number of instructions is \texttt{current_process()}\(^3\), which is compiled as only 5 instructions on a PowerPC processor. Looking at the function pointer implementation, the instruction overhead for this call is:

\[
\frac{12 + 18 + 5}{5} - 1 = 600\% \text{ overhead}
\]

The same calculation for the dynamic table implementation is:

\[
\frac{10 + 21 + 5}{5} - 1 = 620\% \text{ overhead}
\]

This calculation shows the function pointer approach has, for the worst case, a lower performance overhead than the dynamic table approach.

Measuring the instruction count for a system call is generally hard, as the result may vary depending on the data passed to it or the current state of the OS. Analyzing the average instruction count for the remaining system calls is left as further work.

8.2.2 Adding MMU/MPU-support

The implementation resulted in the following:

**MMU bootstrap routine:** The system successfully installs memory pages into the TLB specified by the page table. This imposes an initial overhead which is proportional to \(O(n)\) where \(n\) is the number of invalidation protected pages\(^4\).

**Page replacement:** When there is a context switch, the system successfully refreshes the pages in the TLB. The overhead introduced by this mechanism is proportional to \(O(m)\) where \(m\) is the total number of user pages (i.e. non-invalidation protected pages).

---

\(^3\text{current_process()}\) always executes the same number of instructions regardless of the system state.

\(^4\)Pages which are never replaced in the TLB.
Process protection: If there are two processes in two different memory regions and one of these process tries to access the data of the other process, an exception is triggered. This protection is handled entirely in hardware.

As it is, the MMU is tightly coupled with the rest of the target code. It is not possible to turn off the initialization of the MMU in the OS start up. It is also not possible to (completely) turn off the code responsible for page replacement at context switch.

8.2.3 Updating \texttt{OSE}_{\texttt{ck}} Structures

The following changes have been made to the structures for these different areas of responsibility:

Processes: A block concept has been introduced to \texttt{OSE}_{\texttt{ck}}. This means that it is possible to group processes that should reside in the same memory region together. Also, a scheme for process creation has been made (the created process simply inherits the parent’s memory region).

Pools: There has been a separation between the pool meta-data and the pool contents. The meta-data is now allocated in the system pool. The new scheme occupies 4 more bytes of data compared to the previous scheme.

Signals: A separation between the signal meta-data and the signal payload has been made. As in the previous case, all the meta-data is allocated in the system pool. If signals are sent between different pools, the contents of the signal is copied from one pool the other. This scheme makes it possible for processes to do IPC just as before memory protection was introduced. Overhead is introduced when sending a signal to a process which has a different pool, as it requires copying from the old pool to the new pool.

The updates for the \texttt{OSE}_{\texttt{ck}} structures were tested by a newly written test system which was designed for correctly handling user mode processes. There already exist a very powerful testing system for testing the functionality of the various system calls, but this was not used as it is not designed for handling processes running in user mode.
Chapter 9

Conclusion & Future Work

This chapter will present a discussion of the work and the results presented in the last chapter. We will try to convey whether or not the goals were met, as well as outline things left to do or to improve in the current implementation.

9.1 Solidifying AutoSAR Requirements

Implementing the AutoSAR API was essentially an exercise in data management. The functions themselves were all small and well-defined, and the data types allowed for flexibility in implementation. The AutoSAR specifications are well thought out, and any vagueness is generally there to allow for flexibility in how you implement the feature concerned. The one place where you must come up with a design is in the management of data. There are of course several updates that result from integration as well, and how much this design work will be depends entirely on how “in tune” with AutoSAR the OS is already. In our case, OSEck had a very good match up between features and demands from AutoSAR.

9.1.1 OILTool

The OILTool updates required reworking the configuration data into code which generates the data structures for application access, membership and trusted functions. This would have been greatly simplified if the program had been implemented in a higher level language than C. The code generated by ANTh is object-oriented, and the extra code required to implement an object-oriented approach in C made the code very unintuitive and hard to update.

9.1.2 Application Member List

The application member list represents most of the design work put in to this problem area. While it has been deemed suitable for its purpose, there are a few questions about it left unanswered. First of all, we have the possibility of a sparse array in a low-memory system. The memory loss is relatively low (just under 1% in a 32KB system and 0.0028% in a 1MB system), and we have no statistics about how common sparsity is. The only system we’ve seen, the test system, contained at least one of every object for each application. An investigation of
common industry systems would be needed to find out the likelihood that this is an issue. The effort required along with the small amount of spill mean that this is a negligible loss, at least at the current time.

The idea of caching application IDs with tasks and ISRs after the first call to GetOwningApplication() is quite interesting. Unlike making the member arrays static this would involve a bit more work. Clearly there is a performance gain to be won if there are several applications or each application has numerous entries under a given object type. Better and more thorough performance measurements should be run to see how much overhead repeated calls to the service really add before delving in to modify all the data structures and functions involved in such a caching scheme.

9.2. Updating a Minimal OS

This section will discuss the implementation of the various subsystems altered in OSEck.

9.2.1 Trap Interface

Looking at the instruction count of the two trap interfaces, there is not a big difference, except for the initial overhead of the dynamic table approach. This implies that the two approaches are roughly equally fast. Because of this, the trap interface cannot be selected solely based on its performance – other factors must be included as well.

Looking at ease of use for the two trap interfaces, the function pointer approach has a clear advantage over the dynamic table approach. Because the function pointer approach does not concern the user with configuring which function should be used in the program, it becomes much better integrated into the rest of the system. It does not require any extra configuration, and imposes only a minimal overhead for each system call. As of this, it is possible to distribute a kernel with both the ability to execute system calls normally and with the trap interface without any extra configuration. Thus, the trap interface can be supported by only a certain architecture and does not require any modifications to code for other target architectures.

As shown in chapter 8 in section 8.2.1 the overhead of executing system calls through the function pointer trap interface can be as high as 600%. This can in the worst case imply that system calls which was previously regarded as small and efficient now have to be used more carefully with respect to performance. Whether this overhead is acceptable or not must be decided by the application developer. In the development stage, these numbers might be acceptable as it can help the developer find bugs, but at the production stage, it might not. The trap interface, however, is possible to circumvent, and it is thereby possible for the application developer to choose whether to enable it or not.

Further, some system calls does not necessarily need kernel-mode access rights to be able to perform their tasks. current_process for example only finds the process ID for the current running process – it does not alter the system state in any way. For such system calls to function correctly, the kernel data structures needs to be readable from user-mode.
9.2.2 MMU/MPU Support

The overhead and complexity in the chosen solution is acceptable for a low number of pages, but if the number of pages increases in the system, another strategy would have been more appropriate (such as using a hash-table).

9.2.3 Updating OSE\textsubscript{ck}

Because the OSE\textsubscript{ck} data structures are much more general than they were before, some extra overhead has been introduced. The behavior of the updated data structures cannot be disabled within the same kernel distribution. I.e., the code needs to be recompiled for the old signal scheme to be used. However, the new signal scheme should work just as well as the old, but with some extra overhead. Measuring the exact overhead of typical applications is a subject for future work.

9.2.4 General Discussion of Memory Protection

Adding memory protection to an embedded system can be made without introducing too much overhead. However, it is not always the overhead of the machine that is interesting, it is also about the overhead of the human: the consequence of introducing memory protection in OSE\textsubscript{ck} is that the user has yet another layer of configuration to do. The link script and page table must be hand coded and tailored to each new system. This is probably also the only way it can be done, although the automatic generation of a linker script would be a great help. It is not completely impossible to do – at least not for some parts of it. The hardest part is ensuring that pages are located in non-overlapping regions.

For an example of this, consider the two applications in figure 9.1. The first application occupies 11 KB of data and thereby gets assigned a 16K page\(^1\). This leads to problems for the second application, whose memory region is included in the 16K page. Application 1 can access the data of application 2 as a consequence.

There are two link time solutions to this problem. The first solution is to enforce the correct alignment of the next memory section. This would allow the number of pages to be kept low, but it would increase the overall fragmentation of memory in the system. Implementing this feature in the current linker script

\(^1\)As this is the closest larger page size for the current MMU.
language is somewhat tricky, due to a lack of expressiveness\(^2\). The second solution is to divide the memory area into several pages (of size 4K). This method is better from a fragmentation point of view, but it would require more pages to be managed in the TLB, and it would also require the programmer to split the data sections (which is by itself a complicated task) by hand. Neither of these solutions are perfect from a user point of view, but the first solution is probably the best. It is possible to solve the problem at run time as well. This is done by using the first solution, but letting the kernel use a sizing algorithm to find the most appropriate split sizes of a page.

### 9.3 Future Work

This section will tell what can be made to improve the implemented solutions for both OSEck and AutoSAR.

#### 9.3.1 Improvements on OSEck

There have been some major changes in the core of the OSEck OS. As always, it is hard to get anything right the first time, and the features implemented in OSEck are no exception.

**Performance Measurements**

All of the features introduced has imposed extra overhead to all system calls in OSEck, especially to the system calls used for signal management. However, no in-depth performance measurements of how the various system calls are affected nor how a typical application is affected has been done\(^3\). Such measurements could give a hint of possible performance bottlenecks.

**An Improved Test System**

A test system was developed to verify the correct functionality of the added features, but it was not as extensive as it could have been. There was an already existing powerful test system developed for testing the OS functionality, but that system was not adapted to run with user mode processes. Because of the short time span, porting the existing test system was not an option. This should be done at some point.

**MMU Interface**

A better interface for the MMU should be defined. As only one type of MMU has been inspected, designing such an interface was regarded as futile, the design may not take other MMU designs into consideration.

Such a design should specify how the kernel or a kernel mode process can:

---

\(^2\) More specifically, the absence of a logarithm and exponential operation. These operations facilitate the finding of the correct page alignment.

\(^3\) There exists a performance measurement test for OSEck, but it had not been confirmed working correctly on the PowerPC platform. Because of this, the test could unfortunately not be used.
Specify page format: The page format is something that is always changing from architecture to architecture. The design should specify whether a general page layout should be used, or if the layout should change for each architecture. The existing design uses the page table format specified in the e200z6 manual.[37]

Make TLB alterations: The design should provide a way to make insertions/deletions of the pages existing in the TLB. The current implementation supports these operations, but the functions involved in this are regarded as “internal” (not to be used directly by users).

Make process switching: The kernel needs an interface for replacing the pages on either a context switch or a TLB-miss exception. Currently, there has been no functionality developed for handling TLB miss exceptions, as all relevant pages are supposed to be in the TLB at all times (this is a consequence of the page replacement algorithm of choice).

Bootstrap the MMU: To ensure that the system boots with the correct pages present, the interface should require a start up routine for the MMU to be implemented.

To verify the correctness of such an interface, memory protection mechanisms should be implemented on another architecture as well. Also, a decision whether it should be possible to disable the MMU for an architecture must be made.

9.4 Improvements on AutoSAR

The application member list should be updated to use an array instead of a linked list. While doing the necessary adjustments to the functions and services that use the list one could go ahead and add functionality for application ID caching in tasks and ISRs. To bolster the OILTool program for the future one could use the ANTr grammar to generate an intermediate XML document, and re-write the generating portion to accept XML rather than OIL. This would make it compatible with AutoSAR 3.0 and higher.
9.4. Improvements on AutoSAR
Glossary

ABI
Application Binary Interface. A set of rules defining how a program should behave on a low level when interfacing with other applications, the operating system and other libraries. In practice, this means that an ABI defines data size types, alignments, function calling conventions etc.

API
Application Programming Interface. A set of functions, macros and or data types by which an advanced functionality is exposed. For example, the POSIX API let’s the programmer open and read files, without the need to understand the underlying file system.

AST
Abstract Syntax Tree. A data-structure which represents a source code file as a tree.

AutoSAR
Automotive Open System Architecture

DSP
Digital Signal Processor. A specialized microprocessor designed specifically for digital signal processing, i.e. run some algorithm on a stream of digital data.

E/E
Electronic/Electrical. A generalization of components in a complete system which may be both digital and analog.

ELF
The Executable and Linking Format. ELF is a binary file format, designating the file as an object file or executable.
Exception
An anomalous state caused by a logical error in a program. For example, a divide-by-zero exception may be generated by the ALU to signify that a certain division operation is impossible to perform.

Hook
A Hook function is implemented by the user and invoked by the operating system in the event of something triggering that hook. For example, StartOS() triggers StartupHook.

Interrupt
A hardware or software signal to the processor which causes normal execution to be halted momentarily while an interrupt handler is run.

IPCIter-Process Communication. A system by which two processes can exchange data.

IVT
Interrupt Vector Table. A predefined region of memory which contains pointers to code that should be run to handle interrupts.

Kernel
The central part of many computer operating systems which manages the system’s resources and the communication between hardware and software components.

Loader
A loader loads a program into memory and prepare it for execution (sets registers, jumps to the entry point, etc). The input to a loader is an executable, for example a file in the ELF-format.

OEM
Original Equipment Manufacturer. A company that produces hardware to be sold under another company’s brand.

OIL
OSEK Implementation Language. Used to configure OSEK and AutoSAR systems.

OSE
Operating System Embedded. A real-time OS from Enea.
OSEck
OSE Compact Kernel. A smaller implementation of OSE aimed at DSPs.

OSEK

Panic
In a system panic the OS enters an infinite loop and all execution is halted. This is done as a response to a fatal error during run-time.

Pre-emption
The act of temporarily interrupting a task being executed by a computer system, without it’s cooperation, and with the intention of resuming the task at a later time. Pre-emption is what allows a system to run several programs “simultaneously”, by rapidly switching between them and executing them during a short period of time.

RAM
Random Access Memory. Memory allowing the system to read or write to any part of it without having to access it in sequential order.

Real-time
A system that responds to events or signals within a predictable amount of time. Specifically the response time must be within a known limit. Systems can be either soft or hard real-time systems. Soft real-time means that a delayed result has a reduced value, but may still be usable. Hard real-time means that if a value comes after it’s deadline it is discarded, and may lead to failure of the system.

ROM
Read Only Memory. Memory which can only be read from during the course of normal operation. The programming of the memory can range from being set in the manufacturing process to being done by a debugging tool on demand.

SWAP
Software Automotive Platform. A joint venture within the Swedish automotive industry.
**TLB**
Translation Look-aside Buffer. A table used in conjunction with an MMU to speed up page look-ups.

**Trap**
A trap is a processor instruction used to trigger a software interrupt and thereby enter kernel-mode. The purpose of the trap instruction is either to: (1) Make a system call or (2) Enter a debugging state. For this thesis, we will use definition (1).
The following is the implementation portion of the OIL file to be used with an AutoSAR system. Parts added in this thesis are clearly marked with comments.

```c
IMPLEMENTATION AUTOSAR_OSEck {

    OS {
        ENUM [
            STANDARD,
            EXTENDED
        ] STATUS;

        BOOLEAN STARTUPHOOK = FALSE;
        BOOLEAN ERRORHOOK = FALSE;
        BOOLEAN SHUTDOWNHOOK = FALSE;
        BOOLEAN PRETASKHOOK = FALSE;
        BOOLEAN POSTTASKHOOK = FALSE;

        BOOLEAN USEGETSERVICEID = FALSE;
        BOOLEAN USEPARAMETERACCESS = FALSE;

        BOOLEAN USERESSCHEDULER = FALSE;

        UINT32 SYS_STACKSIZE = 1024;

        ENUM [
```
BCC1,  
BCC2,  
ECC1,  
ECC2  
] CC = BCC1;

BOOLEAN SYS_ERRORHOOK = FALSE;

UINT32 SYS_CLOCK = 40000000;

ENUM [
  far,  
near
] APP_CODE_REGION = far;

// Additional attributes  
UINT32 MAXPOOLS = 4;

ENUM [
  AUTOPOOL,  
  MANUALPOOL {  
    UINT32 POOLID;  
    UINT32 SIZE;  
    UINT32 BUFSIZE1;  
    UINT32 BUFSIZE2;  
    UINT32 BUFSIZE3;  
    UINT32 BUFSIZE4;  
    UINT32 BUFSIZE5;  
    UINT32 BUFSIZE6;  
    UINT32 BUFSIZE7;  
    UINT32 BUFSIZE8;  
  }  
] POOL[];

BOOLEAN BUFFER_CHECK = FALSE;  
BOOLEAN PARAMETER_CHECK = FALSE;  
BOOLEAN DEBUG_HOOKS = FALSE;  
BOOLEAN DEBUG_INFO = FALSE;  
BOOLEAN ILLUMINATOR = FALSE;  
BOOLEAN STATISTICS = FALSE;

STRING SWAP_HOOK = "NO_HOOK";  
STRING SEND_HOOK = "NO_HOOK";  
STRING RECEIVE_HOOK = "NO_HOOK";  
STRING ALLOC_HOOK = "NO_HOOK";  
STRING FREE_HOOK = "NO_HOOK";  
STRING RESTORE_HOOK = "NO_HOOK";  
STRING CREATE_PROCESS_HOOK = "NO_HOOK";  
STRING KILL_PROC_HOOK = "NO_HOOK";  
STRING ERROR_HOOK = "NO_HOOK";
STRING CREATE_POOL_HOOK = "NO_HOOK";
STRING RESET_POOL_HOOK = "NO_HOOK";
STRING KILL_POOL_HOOK = "NO_HOOK";
STRING ERROR_HANDLER = "NO_HANDLER";
STRING POWER_OFF_HANDLER = "NO_HANDLER";
STRING LINK_HANDLER_PROC = "NO_HANDLER";

UINT32 MAXPROCESSES = 32;

// Following options added for AutoSAR compatibility
BOOLEAN PROTECTIONHOOK;
BOOLEAN STACK_MONITORING;
ENUM WITH_AUTO [SC1, SC2, SC3, SC4] SCALABILITYCLASS = AUTO;

APPLICATION{

 BOOLEAN [
 TRUE {
 STRING NAME;
 },
 FALSE
 ] TRUSTED_FUNCTION[];
 BOOLEAN TRUSTED = FALSE;
 BOOLEAN STARTUPHOOK;
 BOOLEAN SHUTDOWNHOOK;
 BOOLEAN ERRORHOOK;
 BOOLEAN [
 TRUE {
 TASK_TYPE RESTARTTASK;
 },
 FALSE
 ] HAS_RESTARTTASK;
 TASK_TYPE TASK[];
 ISR_TYPE ISR[];
 ALARM_TYPE ALARM[];
 COUNTER_TYPE COUNTER[];
 RESOURCE_TYPE RESOURCE[];
 MESSAGE_TYPE MESSAGE[];

 UINT32 POOLID;
}

/*********************************************************************************/
* TASK implementation *
*****************************************************************************

TASK {
  UINT32 [0 .. 31] PRIORITY;
  ENUM [
    NON,
    FULL
  ] SCHEDULE;
  UINT32 ACTIVATION = 1;

  BOOLEAN [
    TRUE {
      APPMODE_TYPE APPMODE[];
    },
    FALSE
  ] AUTOSTART;

  RESOURCE_TYPE RESOURCE[];
  EVENT_TYPE EVENT[];
  MESSAGE_TYPE MESSAGE[];

  //Option added for AutoSAR compatibility
  APPLICATION_TYPE ACCESSING_APPLICATION[];

  // Additional attributes
  UINT32 STACKSIZE = 1024;
};

*****************************************************************************
* COUNTER implementation *
*****************************************************************************

COUNTER {
  UINT32 MAXALLOWEDVALUE;
  UINT32 TICKSPERBASE;
  UINT32 MINCYCLE;

  //Option added for AutoSAR compatibility
  APPLICATION_TYPE ACCESSING_APPLICATION[];

  // Additional attributes
  ENUM [
    SOFTWARE,
    HARDWARE
  ] TYPE;
};
ALARM
{
  COUNTER_TYPE COUNTER;
  ENUM [
    ACTIVATETASK {
      TASK_TYPE TASK;
    },
    SETEVENT {
      TASK_TYPE TASK;
      EVENT_TYPE EVENT;
    },
    ALARMCALLBACK {
      STRING ALARMCALLBACKNAME;
    },
    // Additional attributes
    INCREMENTCOUNTER {
      COUNTER_TYPE COUNTER;
    }
  ] ACTION;

  BOOLEAN [
    TRUE {
      UINT32 ALARMTIME;
      UINT32 CYCLETIME;
      APPMODE_TYPE APPMODE[];
    },
    FALSE
  ] AUTOSTART;

  //Option added for AutoSAR compatibility
  APPLICATION_TYPE ACCESSING_APPLICATION[];
};

RESOURCE
{
  ENUM [
    STANDARD,
    LINKED {
      RESOURCE_TYPE LINKEDRESOURCE;
    },
    INTERNAL
  ] RESOURCEPROPERTY;
};
//Option added for AutoSAR compatibility
APPLICATION_TYPE ACCESSING_APPLICATION[];
};

/***********************************************************************
* EVENT implementation *
***********************************************************************
EVENT {
  UINT64 WITH_AUTO [0x1 .. 0x80000000] MASK;
};

/***********************************************************************
* ISR implementation *
***********************************************************************
ISR {
  UINT32 [1, 2] CATEGORY;
  RESOURCE_TYPE RESOURCE[];
  MESSAGE_TYPE MESSAGE[];

  // Additional attributes
  UINT32 [1 .. 31] INTERRUPTVECTOR;
  UINT32 [1 .. 31] PRIORITY;
};
};
Appendix B

OIL Verification Requirements

This appendix summarizes the requirements on verification of input OIL files to an OIL parser. In this report the OILTool application was extended to validate OIL files according to these requirements.

<table>
<thead>
<tr>
<th>GEN011</th>
<th>Trusted and Non-trusted Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>An OS-Application shall be marked as either trusted or non-trusted.</td>
<td></td>
</tr>
<tr>
<td>Use case</td>
<td>The application object is defined in the OIL file with the parameter <code>BOOLEAN TRUSTED</code>;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS000</th>
<th>Stack Monitoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>The operating system must implement stack monitoring. In scalability class 3 and 4 it will be mandatorily enabled.</td>
<td></td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS012</th>
<th>AlarmCallbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>The use of AlarmCallbacks is forbidden if the system is configured as scalability class 2, 3 or 4.</td>
<td></td>
</tr>
<tr>
<td>Use case</td>
<td>If an alarmcallback is defined in scalability class 2 or higher, the system generator will report an error and no system will be generated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OIL000</th>
<th>OIL Application Inclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>If at least one object of type APPLICATION is present, any TASK, ISR, MESSAGE, ALARM, RESOURCE or EVENT object not found in exactly one APPLICATION objects appropriate member list will result in a fatal error.</td>
<td></td>
</tr>
<tr>
<td>Use case</td>
<td>An orphaned task is included in the configuration file, whereupon the parser returns an error.</td>
</tr>
<tr>
<td>Use case</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>[OIL002]</td>
<td>Correct Use of SCALIBILITYCLASS parameter. If the SCALIBILITYCLASS parameter is NOT set to SC3 or SC4, and there exists an APPLICATION object, the parser shall issue a fatal error.</td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
<tr>
<td>[OIL003]</td>
<td>Enforcement of ALARMCALLBACKS. If the SCALIBILITYCLASS parameter is NOT set to SC1, and there is an ALARM object with the ACTION parameter set to ALARMCALLBACK, the parser shall issue a fatal error.</td>
</tr>
<tr>
<td>Use case</td>
<td>A malformed configuration is put through the parser which generates an error.</td>
</tr>
<tr>
<td>[OIL004]</td>
<td>Enforcement of Category 1 ISRs. If the SCALIBILITYCLASS parameter is set to SC3 or SC4, and there is a APPLICATION object with the parameter TRUSTED set to FALSE that owns an ISR object with CATEGORY set to 1, the parser shall issue a fatal error.</td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
<tr>
<td>[OIL008]</td>
<td>SCALIBILITYCLASS = SC1. If SCALIBILITYCLASS = SC1, then no application may be specified. The generated configuration file shall ensure that no memory protection features are to be enabled.</td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
<tr>
<td>[OIL009]</td>
<td>SCALIBILITYCLASS = SC2. If SCALIBILITYCLASS = SC2, then no application may be specified. The generated configuration file shall ensure that no memory protection features are to be enabled.</td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
<tr>
<td>[OIL010]</td>
<td>SCALIBILITYCLASS = SC3. If SCALIBILITYCLASS = SC3, then all objects must belong to applications. The generated configuration file shall ensure that applicable memory protection features are to be enabled.</td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
<tr>
<td>[OIL011]</td>
<td>SCALIBILITYCLASS = SC4. If SCALIBILITYCLASS = SC4, then all objects must belong to applications. The generated configuration file shall ensure that applicable memory protection features are to be enabled.</td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
<tr>
<td>[OIL017]</td>
<td>OSEK Extended Mode. In scalability classes SC3 and SC4, OSEK error information must be in extended mode. If extended mode isn’t enabled and error shall be given.</td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
<tr>
<td>[OIL018]</td>
<td>Scalability Class Auto. If the scalability class is set to Auto, the correct scalability class should be discovered prior to any tests requiring the scalability class.</td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
<tr>
<td>[OIL019]</td>
<td>Trusted Non-trusted Mix</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>If a non-trusted application exports a trusted function, the parser shall generate a fatal error.</td>
<td></td>
</tr>
<tr>
<td>Use case</td>
<td>–</td>
</tr>
</tbody>
</table>
Bibliography


http://www.eetasia.com/ART8800540693499495_NT_b26a47a5.HTM.


http://www.vinnova.se/misc/VINNOVA-projekt/Projekt—Listhuvud/15534/.


[21] OSEK/VDX. Osek/vdx operating system. Version 2.2.3.


[31] AUTOSAR GbR. Requirements on operating system. V2.0.2Rev0014.


Index

— A —
ANTlr, 17
API, 16, 20, 54
Application, 26, 54
AutoSAR, 1, 19, 53

— B —
BSW, 4, 23
Buffer overflow, 11

— C —
Compilation, 10

— D —
Data structures, 55, 66
Data structures
AST, 18, 56
Linked list, 55
DSP, 25

— E —
ECU, 4, 15
Enea, 1, 23, 25
Execution mode, 8
Execution mode
Kernel-mode, 8
User-mode, 8

— H —
HIS, 19
Hook, 21
Hook
ErrorHook, 23
ProtectionHook, 23, 62
ShutdownHook, 23, 57
StartupHook, 23

— I —
IPC, 26, 61

— L —
Linking, 10, 31
Linking
Link script, 10, 31

— M —
MCU, 1, 4
Memory, 9
Memory
Memory map, 9
Memory protection, 11, 13, 53,
56, 61
Non-volatile, 9
Volatile, 9
mkconfig, 26, 65
MMU, 14, 58

— O —
OIL, 16, 56
OIL
OILTool, 56, 58, 62, 65
Operating system, 7
Operating system
Application, 21
GPOS, 10, 26
Kernel, 7
RTOS, 25, 66
OS , see Operating system 7
OSEck, 2, 25
OSEK, 15, 21, 53, 58
OSEK
Alarm, 15, 21
Event, 15, 21
ISR, 15, 21, 25, 58, 62
Resource, 16, 21
Task, 15, 21, 58

— P —
Paging
Page table, 45, 59, 64, 68, 75
Page-replacement, 14
Pages, 14
Pointer dereferencing error, 11
Pool, 25, 26, 65
Pool
  Pool management, 28
Process, 7, 25, 26
Process
  Interrupt process, 27
  Prioritized process, 27
  Process management, 27

— R —
RTOS, 2

— S —
Scalability class, 21, 57
Segmentation, 14
Signal, 26
Signal
  Signal management, 29
Stack, 7, 10, 29, 33, 58
SWAP, 1, 23, 57
System call, 8, 25

— T —
TLB, 14, 58
Trap interface, 8, 36, 62, 67
The car industry has created a series of standards called AutoSAR as a response to the increasing number of processors in modern vehicles. Among these specifications is one for real-time operating systems (RTOS). This RTOS standard includes requirements for memory protection. This thesis outlines the work involved in introducing the memory protection outlined in this specification in the OSECK operating system. The work consisted of updating the operating system, implementing the AutoSAR OS API, and updating the suite of tools used to build the finished system.

The AutoSAR specifications were found to be very thorough and well thought out. The OS API was successfully implemented, and the data-structures needed to permit its functionality. The existing software tools were updated to conform with the new requirements from AutoSAR, and additional software was created to ease the configuration process.

Memory protection was successfully implemented in the OSECK operating system, including two implementations of the trap interface. The memory protection functionality adds yet another layer of user-configuration to the operating system. Also, additional overhead for system calls, context switches and message passing is expected. A general evaluation of how OSECK application performance is affected is beyond the scope of this thesis, but preliminary studies of additional instruction counts on certain system calls have been performed.
Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enskilda kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera åktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är krävande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner.

The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Per Fagrell
Richard Eklycke
Linköping, March 13, 2009