Porting AUTOSAR to a high performance embedded system

Author: Shuzhou Zhang
Supervisor: Avenir Kobetski
Examiner: Jakob Axelsson
Date: 1 September 2013
ABSTRACT

Automotive embedded systems are going through a major change, both in terms of how they are used and in terms of software and hardware architecture. Much more powerful and rapidly evolvable hardware is expected, paralleled by an accelerating development rate of the control software. To meet these challenges, a software standard, AUTOSAR, is gaining ground in the automotive field. In this work, experiences from porting AUTOSAR to a high performance embedded system, Raspberry Pi, are collected. The goal is both to present experience on the process of AUTOSAR porting and to create an AUTOSAR implementation on a cheap and widely accessible hardware platform, making AUTOSAR available for researchers and students.
# NOMENCLATURE

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>AUTOSAR</td>
<td>Automotive Open System Architecture</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>LIN</td>
<td>Local Interconnect Network</td>
</tr>
<tr>
<td>OEM</td>
<td>Original Equipment Manufacturer</td>
</tr>
<tr>
<td>E/E</td>
<td>Electronics &amp; Electrical</td>
</tr>
<tr>
<td>SWC</td>
<td>Software Components</td>
</tr>
<tr>
<td>RTE</td>
<td>Run-Time Environment</td>
</tr>
<tr>
<td>VFB</td>
<td>Virtual Function Bus</td>
</tr>
<tr>
<td>OSEK</td>
<td>Open Systems and their Interfaces for the Electronics in Motor Vehicles</td>
</tr>
<tr>
<td>MCAL</td>
<td>Microcontroller Abstraction Layer</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>ECUAL</td>
<td>ECU Abstraction layer</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HDMI</td>
<td>High-Definition Multimedia Interface</td>
</tr>
<tr>
<td>DSI</td>
<td>Display Serial Interface</td>
</tr>
<tr>
<td>CRT</td>
<td>Old Cathode-ray Tube</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computing</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>PCB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>TCB</td>
<td>Task Control Block</td>
</tr>
<tr>
<td>LIFO</td>
<td>Last In First Out</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>EB</td>
<td>External Buffer</td>
</tr>
</tbody>
</table>
# CONTENT

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>INTRODUCTION</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Related work</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Thesis Objective</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Outline of the report</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th>AUTOSAR STANDARD</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>AUTOSAR architecture</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>AUTOSAR methodology</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>AUTOSAR interfaces</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>HARDWARE AND SOFTWARE CONFIGURATION</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Hardware configuration</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Software configuration</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>SYSTEM KERNEL DEVELOPMENT</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Operating system fundamentals</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Kernel development</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Linker script</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5</th>
<th>HARDWARE DRIVER DEVELOPMENT</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Serial Peripheral Interface Bus</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>SPI specification in the AUTOSAR standard</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>Types definition</td>
<td>27</td>
</tr>
<tr>
<td>5.4</td>
<td>High level SPI functions in AUTOSAR</td>
<td>29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>EXPERIMENTAL SETUP</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Kernel test</td>
<td>32</td>
</tr>
<tr>
<td>6.2</td>
<td>CAN communication test</td>
<td>33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>CONCLUSIONS</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Conclusion</td>
<td>37</td>
</tr>
<tr>
<td>7.2</td>
<td>Future work</td>
<td>37</td>
</tr>
</tbody>
</table>

| Chapter 8 | ACKNOWLEDGEMENTS | 38 |

| Chapter 9 | REFERENSER / REFERENCES | 39 |
Chapter 1

INTRODUCTION

1.1 Introduction

Traditionally, automotive electronic control units (ECUs) have been quite resource constrained due to cost limitations. However, the development rate and the complexity of automotive software is starting to exceed the capacity of existent ECU hardware. Partly for this reason and partly motivated by falling hardware prices, high performance hardware solutions, similar to those used in applications such as mobiles, media and networking, will have to be considered. Also, following the trend of shorter development and product life times, new hardware solutions will be introduced more frequently in the near future.

For these reasons, there is a need for a realistic test and evaluation platform for the research and development of future automotive ECU architectures. At the very least, such platform should consist of a network of embedded systems, representative for the future types of automotive ECUs. It should conform to existing standards, be easily extendable, and preferably open-source.

In this work, first steps towards such a platform are taken. Briefly put, this report presents the experiences during porting of an operating system (OS), commonly used in automotive applications, to Raspberry Pi, a cheap and widely available high performance embedded platform. Traditionally, automotive ECUs are equipped with microcontrollers that include flash memory and static RAM on the same chip. Code is executed by direct readings, instruction by instruction, from the flash memory. The advantages are that the ECU design becomes simple and that the memory on-chip solution is robust and compact. The main disadvantage of this solution is again its simplicity, making it impossible to utilize the latest memory chip technologies due to different CMOS manufacturing technologies for SDRAM, Flash and CPUs.

In contrast, Raspberry Pis are based on microcontrollers with external flash memories for code storage and execution from RAM, which makes them suitable for high performance applications, such as video streaming. This is an important consideration in future automotive applications since performance and memory constraints are expected to grow considerably in a near future, when vehicles start to cooperate exchanging information with each other and surrounding infrastructure. Other advantages of Raspberry Pis are their low cost and wide availability, which was the final rationale behind using them in our work.

On the downside, the Raspberry Pi architecture requires more complex and time consuming start up procedures, copying and in some cases unpacking of code from ash to RAM. The runtime environment may also give other considerations such as memory protection against software failure and increased risk of single event upsets. Cache memory, pipelining and multi-core will add complexity to the software and system design from a real-time perspective due to variations in execution time.
For the last decade, the automotive industry has been developing a software architecture standard, currently prevailing in this business segment, called the Automotive Open System Architecture (AUTOSAR) [1][2]. The main idea of AUTOSAR is to decouple application software (ASW) from lower level basic software (BSW) by means of a standardized middleware called runtime environment (RTE). This allows running the same application software seamlessly on different hardware platforms, as long as the underlying hardware is linked with the RTE through appropriate BSW. The BSW consists of an operating system that has evolved from the OSEK standard; system services for e.g., memory management; communication concepts; ECU and microcontroller hardware abstractions; and complex device drivers for direct access to hardware.

There exist several implementations of the AUTOSAR standard. However, most of those are commercial products which often make them unsuitable for research purposes. In this work, ArcticCore [3] - one of the very few open-source AUTOSAR implementations - was chosen. The focus lies on extending the ArcticCore source code so that it can be runnable on Raspberry Pis. The details and experiences of the porting work are further described in chapter 4 and 5.

1.2 Related work

While the AUTOSAR standard is open, most of its implementations are commercial products, promoted by large companies, such as Bosch, Dassault Systemes, Vector Informatik GmbH, and dSPACE, to mention just a few. To the best of our knowledge, ArcticStudio OS is one of a few, if not even the only, widely-used AUTOSAR implementations under a GPL license, which is one of the reasons for choosing it in our work.

As a point of reference for this work, the open-source real-time OS for embedded devices, ChibiOS [4], was used. Chibi-OS has been ported to a number of different hardware platforms, including Raspberry Pi. However, its higher level structure differs substantially from AUTOSAR, which made it only useful for our purposes as a source of inspiration.

Another open-source RTOS, Trampoline [5], is an advanced OSEK-compliant academic project that among other things considers multicore issues, an important part of AUTOSAR 4.x. The reason for using ArcticCore here was its broader scope, including such AUTOSAR concepts as communication interfaces, high level ECU abstractions, etc.

In a recent MSc project [6], existing FlexRay communication drivers (on the uCAL level) were merged with ArcticCore’s corresponding ECUAL modules. Our work differs in two ways from that project. Firstly, both the FlexRay drivers and the underlying hardware were developed and owned by a company called QRTECH, which goes against the open-source vision of an automotive evaluation platform. And secondly, the experiences of the presented work go deeper and describe the whole chain of challenges when implementing AUTOSAR uCAL drivers on a new hardware.

Finally, Raspberry Pis were chosen due to their technical characteristics being in line with what is believed to be typical future automotive ECU architecture. In addition, their low cost and wide availability were important choice factors.

In conclusion, while quite some work has been done on AUTOSAR in industry, the academic world is trailing behind in this respect. A literature review that we did in preparation for this work revealed a lack of publications on the subject of AUTOSAR implementation, especially when it comes to its lower layers. The purpose of this paper is to fill this gap and present AUTOSAR porting experiences, together with a real application example that resulted from the porting work.
1.3 Thesis Objective

The thesis addresses several concerns. Firstly, experiences from the porting work, including typical pitfalls and opportunities, are presented, to serve as a basis for future porting work of AUTOSAR to new hardware. And secondly, an embryo of an open automotive hardware platform is demonstrated through an experimental setup, consisting of interconnected AUTOSAR-compliant Raspberry Pis, communicating through a Controller Area Network (CAN) bus, typically used in automotive applications.

1.4 Outline of the report

The report is organized as follows. In Chapter 2, some necessary background about the AUTOSAR standard is presented. In Chapter 3, the hardware and software configuration for this thesis is introduced. Chapter 4 shows the process about processor kernel development and Chapter 5 gives a description about SPI driver development. Chapter 6 is devoted to the experimental setup, while chapter 7 concludes the report.
Chapter 2

AUTOSAR STANDARD

Over the last few years, there has been a tremendous increase in the functional scope of automotive electronics and the application of software in vehicles. With the growing software amount, the complexity of vehicle electronic systems is also increasing. Hence, the challenge for future automotive engineers is to optimize whole electronic systems of vehicles and not single components, since the optimization work for components can be handled by the suppliers. In addition, with the rising variety of customer’s demand, vehicles are updated more often, making it necessary to improve the flexibility of automotive product modification. At the same time, in order to shorten the development time and cost, eliminating repetitive low-level development work, it is better to have scalable solutions which permit the development of standardized application software components that can be reused for several vehicle types. All these reasons make it imperative to have an industry-wide standardized software infrastructure.

For these reasons AUTOSAR was developed. The first version of the AUTOSAR standard was released in 2005 and currently the AUTOSAR v4.0 has already been released. Figure 1 shows the significant change in the relationship between software development and hardware of an automotive electronic system by using the AUTOSAR standard. The picture shows that the hardware and software is widely independent of each other which provides a possibility to reuse the software applications on different ECUs. This development processes is simplified. This reduces development time and costs. Lastly, reuse of software increases at OEMs as well as at suppliers, which enhances both quality of software and efficiency of development.

![Figure 1. The change of automotive software development](image)

This chapter gives a background on the AUTOSAR Standard, which covers three parts: AUTOSAR architecture, AUTOSAR methodology and AUTOSAR interfaces.
2.1 AUTOSAR architecture

The AUTOSAR Architecture [7] distinguishes on the highest abstraction level between three software layers: Application, Runtime Environment and Basic Software which run on a Microcontroller, as shown on the left side of Figure 2.

![Figure 2. AUTOSAR architecture layers](image)

2.1.1 Application layer

The application layer, also called AUTOSAR software layer, consists of AUTOSAR software components, as shown in the right side of Figure 2. The AUTOSAR Software Component (SWC) is a fundamental design concept of AUTOSAR, which is the basic structure of an AUTOSAR application. This means that an AUTOSAR application may include several AUTOSAR Software Components and each AUTOSAR Software Component encapsulates part of the functionality of the application. Each SWC is deployed, or mapped, on one ECU. For example, an automatic light adaptive application may consist of three AUTOSAR SWCs which are outside brightness detective component, a light request component and a light control master component [8]. The interaction between the component and other components is done through ports which are the information exchange points (providing data or services and requiring data or service) of SWCs.

2.1.2 Runtime environment layer

The Run-Time Environment (RTE) is the heart of the AUTOSAR ECU architecture. The RTE layer has two main functionalities: one is to provide the infrastructure services that enable communication between AUTOSAR SWCs and the other one is to act as the means by which AUTOSAR SWCs access basic software modules including the OS and communication services [9], as shown in Figure 2. From the viewpoint of the AUTOSAR Software Component, the RTE implements the Virtual Function Bus (VFB) functionality on a specific ECU.

The Virtual Functional Bus is the communication mechanism that allows SWCs to interact with each other. When the connections between AUTOSAR Software Components for a concrete system are defined, the VFB will allow a virtual integration of them. And when the virtual integration of SWCs mapped onto local connections (within a single ECU) or on network-technology specific communication mechanisms (such as CAN), the VFB is implemented by the concrete interface between the individual components and between the components and the BSW, so called RTE.

2.1.3 Basic software layer
Basic Software (BSW) is a standardized software layer, which provides low-level services to the application layer. As shown in Figure 2, the Basic Software Layer consists of operating system, services, communication, ECU abstraction and Complex Device Drivers. All these functionalities can also be sorted into another four sub-layers, which are Microcontroller Abstraction layer, ECU abstraction Layer, Services Layer and Complex Drivers layer, as shown in Figure 3.

![Figure 3. The AUTOSAR architecture including BSW sub-layers](image)

The Microcontroller Abstraction Layer (MCAL) is the lowest software layer of the Basic Software. MCAL is a software module that directly accesses on-chip MCU peripheral modules and external devices that are mapped to memory, and makes the upper software layer independent of the MCU. The MCAL consists of communication drivers (e.g. SPI, CAN, LIN), I/O drivers (e.g. ADC, PWM, DIO), memory drivers (e.g. internal Flash, internal EEPROM) and memory mapped external memory devices (e.g. external Flash), and microcontroller drivers (e.g. Watchdog, General Purpose Timer). The details are shown in Figure 4.

![Figure 4. Microcontroller Abstraction Layer](image)

The ECU Abstraction layer (ECUAL) makes upper layers independent of how the ECU is structured, and provides interfaces to the drivers in the Microcontroller Abstraction Layer and drivers for external devices. This is done through APIs for access to peripherals and devices regardless of their physical location, and their connectivity. Figure 5 gives an example to show the connection between the ECU Abstraction layer and MCAL.
The Services Layer is the highest layer of the Basic Software. The services layer offers operating system services, vehicle network communication and management services, memory services, diagnosis services and ECU state management.

The Complex Device Driver Layer (CDDL) is used for complex functions not found in other layers. This layer accesses the microcontroller (MCU) directly, as shown in Figure 6. CDDL is used for complex/non-standard sensor and actuator drivers, which is not explicitly specified by the AUTOSAR. These drivers do not follow the normal layered AUTOSAR architecture, but instead access both the microcontroller and RTE layers directly.

2.2 AUTOSAR methodology

AUTOSAR defines an approach called AUTOSAR methodology [10] for the process of system development, as shown in Figure 7 [11]. The figure roughly describes the design steps form system-level configuration to the generation of an ECU executable.

Firstly, the vehicle architecture design is defined. In this step, the main task is to select the software components and the hardware and the entire system constraints are declared. Lastly, in this step, all the necessary information is filled into the system configuration input XML file using appropriate templates. The next step is vehicle system design. The output of this step is a system configuration description XML file, which specifies the system information like bus mapping or topology and the mapping of SWCs to ECUs. Further on, it comes to the stage of single ECU system design and the main task of this stage is to extract the information from the system configuration description file for building a specific ECU. The next stage is the single-ECU system development. Here the application software is
developed, including task schedule, the RTE settings, BSW configuration and runnable entity assignment, etc. Last step is to generate an executable file and download it to the ECU. Typically, this step involves code generation and compilation and links everything together into an executable.

Figure 7. Design sequence using AUTOSAR Methodology

2.3 AUTOSAR interfaces

In order to reuse the software from different sources, the AUTOSAR defines a series of application interfaces which can be classified into 3 types. In Figure 2, these interfaces are given: "AUTOSAR Interface", "Standardized AUTOSAR Interface" and "Standardized Interface" [12].

An AUTOSAR Interface denotes software component interfaces. Each component that is connected to the RTE via an AUTOSAR interface provides and/or requires ports, through which it is connected to and interacts with other components, as shown in Figure 2.

A Standardized AUTOSAR Interface is an AUTOSAR Interface whose syntax and semantics are standardized in AUTOSAR. Typically, the Standardized AUTOSAR Interfaces are used to define AUTOSAR services, which are provided by the AUTOSAR BSW to the SWCs, as shown in Figure 2.

A Standardized Interface is typically used between software modules that are always on the same ECU. The operating system for instance has to provide a standardized interface to allow the RTE to consume services and to schedule other services, communication and ECU state management in the BSW layer, as shown in Figure 2.
Chapter 3

HARDWARE AND SOFTWARE CONFIGURATION

This chapter describes the hardware and software configuration of this thesis work. In the first section, an embedded systems hardware platform that used in this work is presented. In the second section, the software environment is described, which contains an integrated development environment (Arctic Studio) and an open source implementation of the AUTOSAR standard (Arctic Core) which is referenced by this thesis.

3.1 Hardware configuration

3.1.1 Raspberry Pi

The Raspberry Pi is perhaps the most inspiring computer available today. It is a circuit board, about the size of a credit card, with components and sockets stuck on it, as shown in Figure 8. Rather than a microcontroller board, the Raspberry Pi is a complete computer at a very cheap price.

![Raspberry Pi](image)

The heart of a Raspberry Pi is the same processor as in iPhone 3G and Kindle 2, so it owns abundant capabilities comparable to those powerful little devices. The processor of a Raspberry Pi is a Broadcom BCM2835 [13] chip (32 bit, 700 MHz) that contains an ARM central processing unit (CPU) – ARM1176JZF-S, and a video core 4 graphics processing unit (GPU). The CPU and GPU share memory between them. A Raspberry Pi owns rich functionalities. First of all, A Raspberry Pi supports three different kinds of video outputs: composite video, High-Definition Multimedia Interface (HDMI) video and Display Serial
Interface (DSI) video. Next, a Raspberry Pi provides an audio output via a 3.5mm jack or over HDMI (the HDMI port carries both the video signal and a digital audio signal).

A Raspberry Pi (in Model B) has two Universal Serial Bus (USB) ports. Another significant feature of a Raspberry Pi is that it can connect to the network through an onboard Ethernet network connector (RJ45) or by using a WiFi USB dongle in wireless. In wired network, with a cable connected between a Raspberry Pi and a router, a Raspberry Pi will automatically receive the details it needs to access the Internet through the Dynamic Host Configuration Protocol (DHCP). This assigns the Raspberry Pi an Internet Protocol (IP) address on the network, and tells it the gateway it needs to use to access the Internet (typically the IP address of your router or modem). In wireless network, a WiFi adapter is plugged in to one of the USB ports, and then there are no longer two free USB sockets available for a keyboard and mouse.

In addition, a Raspberry Pi’s eight General-Purpose Input/Output (GPIO), I2C bus and SPI bus also make it an ideal choice for experimenting with computer hardware and peripheral devices. Finally, a Raspberry Pi supports a lot of operating systems, such as Raspbian, RaspBMC, DexOS, Linux and so on. For booting an operating system on a Raspberry Pi, a SD card is needed with a bootloader and a suitable operating system image (the details are explained in chapter 4).

Above all, the Raspberry Pi is really a high performance embedded platform.

3.1.2 ARM processor fundamentals

**ARM processors and architecture versions**

An ARM processor is a 32-bit RISC processor, which comprises an ARM core (the execution engine that processes instructions and manipulates data) plus the surrounding components that interface it with a bus. Reduced instruction set computing, or RISC, is a CPU design strategy based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A computer based on this strategy is a reduced instruction set computer, also called RISC. The RISC design strategy allows a RISC processor to be simpler, and thus the processor can operate at higher clock frequencies.

![Figure 9. ARM based processor](image)

ARM Holdings itself does not manufacture its own electronic chips, but licenses its designs to other semiconductor manufacturers. Therefore, there are many variants of ARM processors, with different capabilities, and implementing different features. Figure 9 [14] gives an example of a typical ARM based microcontroller chip. The ARM 11 core is the central processing unit (CPU) of a microcontroller chip. In addition, a number of other components are required for the whole ARM11 processor-based microcontroller. After chip manufacturers
license the ARM11 core, they can put the ARM11 core in their silicon designs, adding memory, peripherals, input/output (I/O), and other features. Therefore, ARM11 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features.

However, all these ARM processors should implement a version of the ARM architecture which has been given the version name ARMv{n}. So the "ARM11 processor" refers to a processor of the ARM11 family based on the ARMv6 architecture. For example, the processor of Raspberry Pi is Bcm2835 produced by Broadcom Company, a member of ARM11 family, using an ARM1176JZF-S core which is based on an ARMv6 architecture.

**Processor modes**

ARM processors support different processor modes, depending on the architecture version. An ARM11 processor supports seven processor operating modes, which are User mode (USR), System mode (SYS), Fast Interrupt Request mode (FIQ), Interrupt Request mode (IRQ), Supervisor mode (SVC), Abort mode (ABT) and Undefined mode (UND). Each operating mode is used for a particular purpose and only one mode is in use at any time. Apart from the User mode that is a non-privileged mode, the other 6 modes are privileged modes that can fully access system resources.

**Registers**

The registers are the most fundamental storage areas on the processor. The ARM has a total of 37 registers. These comprise 30 general-purpose registers, 6 status registers and a program counter.

![Register organization](image)

Figure 10. Register organization

Figure 10 shows the register organization of an ARM 11 processor. The registers R0 to R7 are unbanked registers and always available no matter which mode the processor is in. (Unbanked registers means the physical register referred to by each of them depends on the current processor mode.) These registers are truly general-purpose, with no special use being placed on them by the processors' architecture. Registers R8 to R12 have two banked physical registers each. (Banked registers means that each of them refers to the same 32-bit physical register in all processor modes) One is used in all processor modes other than FIQ mode, and
the other is used in the FIQ mode (R8_fiq to R12_fiq). The existence of separate FIQ mode versions of these registers allows very fast interrupt processing. This means that when the ARM processor switches into FIQ mode, the software does not need to save the normal R8–R12 registers, since FIQ mode has its own set that can be modified. Registers R13 and R14 have six banked physical registers each. One is used in User and System modes, while each of the remaining five is used in one of the five exception modes. R13 is also known as the Stack Pointer which stores the address of the last program request in a stack, while R14 is known as the Link Register that holds the address to return to when a function call completes. Each processor mode has its own R13 and R14 registers. This allows each mode to maintain its own stack pointer and return address. Register R15 that holds the Program Counter, is used to identify which instruction is to be performed next.

Current Program Status Register (CPSR) is an important register because it keeps the processor’s computation condition, controls the ARM mode switching, enables/disables the interrupts and switches between different instruction sets – ARM instructions set or THUMB set. Each exception mode also has a Saved Program Status Register (SPSR) that is used to preserve the value of the CPSR when the associated exception occurs. (User mode and System mode do not have an SPSR, because they are not exception modes.)

The format of CPSR and SPSR is shown in Figure 11. It contains 4 bits at the top known as the condition code flags, and 8 bits at the bottom known as the control bits.

<table>
<thead>
<tr>
<th>Bit</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>V</td>
<td></td>
<td>F</td>
<td>T</td>
<td>Mode</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11. Program status register

The condition code flags are in the top four bits of the register and have the following meanings:

- **N**: Negative; the Negative (sign) flag takes on the value of the most significant bit of a result. Thus when an operation produces a negative result the negative flag is set and a positive result results in the negative flag being reset. This assumes the values are in standard two's complement form. If the values are unsigned the negative flag can be ignored or used to identify the value of the most significant bit of the result.
- **Z**: Zero; the Zero flag is set when an operation produces a zero result. It is reset when an operation produces a non-zero result.
- **C**: Carry; the Carry flag holds the carry from the most significant bit produced by arithmetic operations or shifts. As with most processors, the carry flag is inverted after a subtraction so that the flag acts as a borrow flag after a subtraction.
- **V**: Overflow; The Overflow flag is set when an arithmetic result is greater than can be represented in a register.

The control bits, including the mode bits \( \text{M}[4:0] \), \( \text{I} \), \( \text{F} \) and \( \text{T} \) are at the bottom eight bits of the register. The \( \text{I} \) and \( \text{F} \) bits are the interrupt disable bits, which disable interrupts in the processor if they are set. The \( \text{I} \) bit controls the IRQ interrupts and the \( \text{F} \) bit controls the FIQ interrupts. The \( \text{T} \) bit represents the state of an ARM processor. If the bit is set to 1, the ARM core is executing THUMB code, which consists of 16-bit instructions. And if the bit is set to 0, the ARM core is executing ARM code, which consists of 32-bit instructions. In order to explain the mode bits, Table 1. Processor mode is given, which lists the various modes and the associated binary patterns.
In an ARM privileged mode, the CPSR can either be to read or written. However, in an ARM unprivileged mode, the control bits in the CPSR cannot be modified.

Table 1. Processor modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>MODE[4:0]</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>USR</td>
<td>10000</td>
<td>Unprivileged</td>
</tr>
<tr>
<td>FIQ</td>
<td>10001</td>
<td></td>
</tr>
<tr>
<td>IRQ</td>
<td>10010</td>
<td></td>
</tr>
<tr>
<td>SVC</td>
<td>10011</td>
<td></td>
</tr>
<tr>
<td>ABT</td>
<td>10111</td>
<td></td>
</tr>
<tr>
<td>UND</td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>SYS</td>
<td>11111</td>
<td></td>
</tr>
</tbody>
</table>

Interrupts, exceptions, and the vector table

An interrupt is usually defined as an event that changes the sequence of instructions executed by a processor. Normally, interrupts are issued by interval timers and I/O devices. For instance, the arrival of a keystroke from a user sets off an interrupt.

Exceptions, on the other hand, are caused either by programming errors or by anomalous conditions that must be handled by the kernel of the operating system. However, in the ARM architecture manuals, interrupts and exceptions are mixed together, that is, ARM defines an interrupt as a special type of exception. The ARM processor supports seven types of exceptions which are listed in Table 2. Each exception is handled by a specific exception handler. When an exception occurs, the processor halts execution after the current instruction. The state of the processor is preserved in the Saved Processor Status Register (SPSR) so that the original program can be resumed when the exception routine has completed. The address of the instruction the processor was just about to execute is placed into the Link Register of the appropriate processor mode. Then the processor will jump to a fixed address that corresponds to that exception. This fixed address, called the exception vector address, is located at the bottom 32 bytes of the memory map. These 32 bytes are called the exception vector table (shown in the Table 2).

Table 2. The exception vector table

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Processor Mode</th>
<th>Vector Address</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>SVC</td>
<td>0x00000000</td>
<td>1(high)</td>
</tr>
<tr>
<td>Undefined instructions</td>
<td>UND</td>
<td>0x00000004</td>
<td>6(low)</td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>SVC</td>
<td>0x00000008</td>
<td>6</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>ABT</td>
<td>0x0000000C</td>
<td>5</td>
</tr>
<tr>
<td>Data Abort</td>
<td>ABT</td>
<td>0x00000010</td>
<td>2</td>
</tr>
<tr>
<td>Interrupt</td>
<td>IRQ</td>
<td>0x00000018</td>
<td>4</td>
</tr>
<tr>
<td>Fast Interrupt</td>
<td>FIQ</td>
<td>0x0000001C</td>
<td>3</td>
</tr>
</tbody>
</table>
The Reset exception is the highest priority exception. When power is first applied to the processor, it will put 0x00000000 on the 32-bit address bus going to memory and receive its first instruction. An Undefined instruction exception occurs when an instruction cannot be recognized by the ARM processor. Software interrupts, or SWIs, are generated by ARM instructions. This causes an exception, and forces the processor into Supervisor mode. When an SWI exception happens, the processor fetches the instruction at the exception vector address 0x8 and then the SWI handler will be in charge. A Prefetch abort exception occurs when the processor tries to execute an instruction pre-fetched from an illegal address. When an abort is acknowledged by the processor, it will fetch the instruction in the exception vector table at address 0xC. An abort handler will then be responsible for either trying to fix the problem or die a graceful death. A Data abort exception happens when data transfer instruction tries to read or write at an illegal address. ARM cores have two kind of interrupt lines, one for a fast interrupt (FIQ) and the other one for a low-priority interrupt (IRQ). An Interrupt Request (IRQ) exception happens when a processor’s external interrupt request pin is asserted. And a Fast Interrupt Request (FIQ) exception occurs when a processor’s external fast interrupt request pin is asserted.

3.2 Software configuration

In this work, tools used for developing software applications are designed by ArcCore Company, including Arctic Studio and Arctic Core.

3.2.1 Arctic Studio

Arctic Studio [15] is an advanced IDE (Integrated Development Environment) built on top of Eclipse. It includes a complete C development environment based on Eclipse CDT and a custom made build environment based on MSYS [16]. By using Arctic Studio, C applications were developed for this thesis work.

3.2.2 Arctic Core

Arctic Core [17], one of the implementations of the AUTOSAR standard provided by ArcCore [18], holds all the source code and build scripts needed to build an AUTOSAR System. Included in the Arctic Core platform are a real time operating system, communication services supporting automotive communications standards such as CAN, LIN and more, memory services and drivers for a number of different microcontroller devices, such as FREESCALE MPC5xxx, HCS12XX, ARM Cortex M3 and R4. All the source code available for download is licensed under GPLv2 and can freely be used for education or evaluation.

Figure 12 (a) gives a brief overview of the Arctic Core file system. It describes the general behavior of AUTOSAR. For example, the “rte” contains the code that is needed for the AUTOSAR Runtime Environment (RTE) functionality to work correctly. The “system” folder contains the implementation of the AUTOSAR operating system. The “memory” folder contains the memory management service of the AUTOSAR OS. The “communication” folder contains the files that fulfill the communication functionality between the AUTOSAR OS and the AUTOSAR RTE. The “drivers” folder includes the implementation of some complex drivers.

There are two additional subdirectories in the Arctic Core folder, as shown in Figure 12(b), which were the ones that were modified in this thesis. One is a folder named “arch” which contains files used for all the architectures in Arctic Core and another one is a folder called “boards” which contains examples used for different embedded platforms. In our case, the “arm_v6” folder was designed to represent Raspberry Pi’s processor architecture. It includes three subfolders called “drivers”, “kernel” and “scripts”, shortly explained below.
• drivers – contains hardware drivers’ files of the processor, e.g. Spi.c, Can.c (see chapter 5)
• kernel – contains start up files of the processor, e.g. stack.c, interrupt.c (see chapter 4)
• scripts – contains linker files (see chapter 4)

As for the “Raspberry Pi” folder, the purpose is to support some examples for using the AUTOSAR OS on the Raspberry Pi platform. For example, the operating system test example (os_simple), the communication example (com_simple) and the RTE example (led_rte). In addition, there are some description files for the platform information (board_info.txt), example building configuration (build_config.mak) and the system memory usage (memory.ldf).

Figure 12. Arctic Core file system
Chapter 4
SYSTEM KERNEL DEVELOPMENT

This chapter presents kernel development part of the AUTOSAR porting process on the Raspberry Pi. The first part of this chapter introduces some basic concepts about general operating systems. The second part focuses on the four core steps that were taken to set up the AUTOSAR OS kernel and prepare it for running on a Raspberry Pi. This kernel development process includes initialization, memory modeling, context switch and exception handling. Initialization is executed when the operating system starts up. Data structures, global variables, and hardware are set up in this stage in order to prepare necessary configuration for the operating system. Memory handling builds the system and task stacks so as to determine how much memory is available for different system tasks. A context switch is needed to handle scheduled tasks. Finally, the method for handling interrupts and exceptions is a critical part of the architecture design of the operating system. The third part of this chapter is devoted to the description of a simple linker script designed for this thesis.

4.1 Operating system fundamentals

An operating system (OS) is a collection of software that manages computer hardware resources and acts as an interface between the user of a computer and the computer hardware. In simple words, “OS is the software that controls the hardware”. Nowadays, operating systems are not only used in computers but are installed in almost all user friendly machines operated by humans like mobile phones, robots, electronic control units of electronic systems in a vehicle, etc.

ARM processors support over 50 operating systems which can be divided into two main categories: real-time operating systems (RTOSs) and platform operating systems. A RTOS is an operating system that supports real-time applications and embedded systems by providing logically correct result within the deadline required. For example, the AUTOSAR OS is a RTOS, which is an extension of the industry standard OSEK OS [19]. Platform operating systems require a memory management unit to manage large, nonreal-time applications and tend to have secondary storage. For example, Windows and Linux are traditional platform operating systems.

4.1.1 Firmware and Bootloader

An operating system needs a particular type of environment before it can operate on a hardware device. Thus, firmware and bootloader are needed. Firmware is a fixed and small program that is generally responsible for basic low-level hardware in a system. Normally, firmware is stored in Flash, ROMs, PROMs or EPROMs and cannot be modified by users. This means that when a microcontroller is produced by the manufacturer, usually there is a firmware inside the chip. A bootloader is the first executed code after the operating system resets on a hardware target. It aims to bring an operating system to a state in which it can perform its main function.
4.1.2 Initialization

Initialization starts after the bootloader relinquishes the control of the processor. The goal is to initialize the operating system, including data structure initialization, global variable initialization, hardware initialization, operating system configuration and so on. If the operating system is pre-emptive, such as the AUTOSAR OS, a periodic timer interrupt is set up during this stage.

4.1.3 Memory model

A memory model is an important part of an OS. It defines how much memory may be used by programs. The hardware, the operating system, the choice of programming language and its compiler all contribute to the memory model. Memory handling involves setting up the system stack segments, heap segment and code segment. The position of the stacks determines how much memory is available for the OS. The heap is used for dynamic memory allocation. For example, if a block of memory is allocated by using malloc(), it uses memory in the heap. As for the code segment, it presents the area which contains executable instructions and it is placed below the heap or stack in order to prevent heap and stack overflows from overwriting it [20].

4.1.4 Context switch and task scheduling

In an OS, a task is a basic program unit that users interact with when performing a certain job. A task in an OS is represented by a data structure known as a task control block (TCB), also called process control block (PCB). A TCB contains important information about the specific task, including task id, task context, task priority and an initial context, as shown in Figure 13 [21]. The TCB specifies all the parameters necessary to schedule and execute a task.

Normally, a task may exist in any of the four states, including RUNNING, SUSPENDED, READY and WAITING, as shown in Figure 14. During the execution of an application program, individual tasks are continuously changing from one state to another. But only one task can be in the running mode at any point of the execution.

Context switching happens when a new task has been scheduled for execution. During a context switch, the running task is stopped and another task is given a chance to run. The operating system kernel must stop the execution of the running task, copy out the values in hardware registers to its TCB, and update the hardware registers with the values from the TCB of the new task. Also, context switching may occur as a result of an external interrupt. For example, a timer interrupt causes the operating system to make a scheduled task active.

The scheduler is the soul of an OS. It is virtually an algorithm that decides which task has the right to use the processor next [22]. In a real time operating system, a real-time scheduler should know the deadlines of all tasks. The scheduler should base decisions on a comparison...
of the deadlines of tasks that are in the ready queue such that all tasks meet their timing constraints. There are a lot of scheduling algorithms in real time systems, such as rate monotonic algorithm, deadline monotonic algorithm, and earliest deadline first algorithm, etc [23].

Figure 14. Task state model

Figure 15. Boot sequence of Raspberry Pi
4.2 Kernel development

Here, the OS fundamentals that needed to be implemented in this thesis work are described.

4.2.1 Bootloader for the Raspberry Pi

To boot a Raspberry Pi and make an OS run on it, four files are needed, bootcode.bin, loader.bin, start.elf and kernel.img. All these files can be obtained from the Raspberry official forum [24].

Figure 15 shows the boot sequence of a Raspberry Pi. When a Raspberry Pi is turned on, its GPU core starts working first while the ARM core is off. At this point the SDRAM is disabled. Then the GPU starts executing the first stage bootloader which is developed by the manufacturer and stored in ROM on the SoC and enable the SD card. In the second stage, the first stage bootloader reads the SD card and loads bootcode.bin from the SD card into the L2 cache and runs it. The file bootcode.bin has the duty to enable the SDRAM memory and load the binary file, loader.bin from the SD card into RAM. The file loader.bin contains the code that is able to load the firmware - start.elf. Finally, the file start.elf loads kernel.img and the OS is running. By default, the kernel.img is a linux kernel that is provided by the Raspberry Pi foundation. But in this thesis an AUTOSAR OS kernel image was developed instead of it.

4.2.2 Initialization

Normally, there are two main stages of initializing an operating system on ARM architecture – startup and execution of the OS initialization [25]. To port the AUTOSAR OS to the Raspberry Pi, in the initialization stage, the above 2 steps should be implemented as well. During the startup stage, the startup code sets up stacks of the seven ARM modes. In the OS initialization stage, hardware drivers, operating system configuration and a periodic timer are initialized. When all these stages are ready, the first task can be executed.

The startup sequence for Raspberry Pi was implemented in the assembly file crt0.s. The processor starts with a reset vector when the Raspberry Pi is powered on. It means that the processor will jump to a fixed address via a signal sampled at reset automatically, which is the address 0x00000000H of the RAM, as shown in Figure 16. The reset vector is always the first instruction to be executed and contains a branch to the reset handler which is the place to execute the reset code.

```
.........
| Interrupt Handler |
.........
| Reset Handler |
.........
| Fast Interrupt Request |
| Interrupt Request |
| Reserved |
| Data Abort |
| Prefetch Abort |
| Software interrupt |
| Undefined Instruction |
| Reset |
```

Figure 16. The execution flow of reset handler
In the reset handler, we should initialize stack pointers for all ARM modes. This process is implemented by writing different mode bits (M[4.0] in Figure 11) to CPSR in order to switch the processor to a particular ARM mode and setting the stack register (R13) for this mode. The stack register is allocated according to pre-defined stack sizes for the different modes. The following instructions give an example of how to set the stack of the UND mode.

```assembly
/*switch to undefined instruction mode*/
msr CPSR, #MODE_UND | I_BIT | F_BIT
/*set base address (store in R0) for the stack*/
mov sp, r0
/*set UND mode’s stack size*/
ldr r1, =_und_stack_size_
/*prepare the base address for the next mode’s stack */
sub r0, r0, r1
```

Once the stacks are set up, each mode’s stack pointer points to the stack space of that operation mode, so that when the program runs into an exception mode, the interrupted task’s information can be protected into the stack pointed to by R13. When an application program returns from the abnormal pattern, by recovering the interrupted stack, the normal execution of the program can proceed contiguously.

As part of the OS initialization, a periodic timer interrupt should be implemented in this stage because it is related to the hardware. A timer is an important part of a real-time OS, providing a system tick for the OS. In the AUTOSAR OS, the system tick is defined to 1 millisecond. When the timer is activated, a counter will start to decrement its value, which in our case is 1000 (because in our system one system tick is 1µs), in a specified timer register. Once the value reaches zero, an interrupt is raised. In this way, the timer generates an interrupt each 1 millisecond. In the timer interrupt service routine, the AUTOSAR OS will check the scheduler table to decide which task should be run next.

4.2.3 Memory Model

We implemented a simple memory model for the AUTOSAR OS, as shown in the Figure 17. ARM physical addresses start at 0x00000000 for the RAM. A vector table (see also Figure 16) is built from 0x00000000 to 0x000000020. Then the code section, heap and stack are arranged. The address range from 0x20000000 to 0x20FFFFFF is used for microcontroller registers. The hardware interfaces can then be controlled by accessing corresponding addresses in this area.

Typically, a task’s address space consists of a text segment, a data segment, a stack segment and a heap segment at runtime. A text segment is the area where the executable instructions reside in. A text segment is often read-only, to prevent a program from accidentally modifying its instructions.

An initialized data segment is usually called the Data Segment. A data segment is a part of the virtual address space of a program, which contains the global and static variables that are initialized with nonzero values by the programmer. The portion of the executable file containing the data segment is the data section. Since the values of the variables can be modified at run time. This segment can be sorted as a read-only part and an initialized read-write part. For example, a global string defined by `char s[] = “thesis report”` in C would be stored in the initialized read-write area. And a constant value, such as `const* char string = “thesis report”`, would make the string literal “thesis report” to be stored in the initialized read-only area, while the character pointer variable `string` would be placed in the initialized read-write area.

An uninitialized data segment is often called the “bss” (BSS stands for ‘Block Started by Symbol’.) segment. All global variables and static variables that are initialized to zero by
default or do not have explicit initialization in source code would be kept in this segment. For instance a variable declared “static int i;” would be stored in the BSS segment.

The stack segment is where local (automatic) variables are allocated. The data is popped up or pushed into the stack according to the Last In First Out (LIFO) rule. The stack holds local variables, temporary information/data, function parameters, return address and the like. When a function is called, a stack frame (or a procedure activation record) is created and pushed onto the top of the stack. When a function returns, the stack frame is popped from the stack. Typically, the stack grows downward.

The heap is where the dynamic memory (obtained by malloc() in C) comes from. Everything on the heap is anonymous, thus only parts of it can be accessed through a pointer. As memory is allocated on the heap, the task’s address space grows. Although it is possible to give memory back to the system and shrink a task’s address space, this is almost never done because it is normally re-allocated to other tasks. Freed memory (free() in C) goes back to the heap, creating what is called holes. Typically, the heap grows upward. This means that successive items that are added to the heap are added at addresses that are numerically greater than previous items.

4.2.4 Context Switch

When a new task has been scheduled for execution, the new and old tasks have to be swapped using a context switch. To achieve this, the ARM context switch splits the activity into two stages. This process is given in Figure 18 [26]. In the first stage, the state of the

![Figure 17. System memory layout](image)
current task must be saved somehow, so that, when the scheduler gets back to the execution of this task, it can restore its state and continue. The state of the current task includes all the registers that the task may be using, especially the program counter, together with any other OS specific data that may be necessary. This data is usually stored in a TCB. In the second stage, the registers with data from the new task's TCB should be loaded. In doing so, the program counter from the TCB is loaded, and thus execution can continue in a new task. The new task is chosen from a task queue according to its priority.

Figure 18. The sequence of context switch

Figure 19. Task level context switch
The task level context switch is shown in Figure 19. A task is assumed to run in ARM mode and uses the SYS registers. The processor's SP(R13) points to a location into the current task's stack. In this thesis, os_ArchSwapContext(old_pcb, new_pcb) is called when a context switch happens. Firstly, the context of the current task is saved onto its own stack, that is, saving the return address (R14_sys) and the information of registers (from R0 to R12). Secondly, the stack pointer of the old task being switched out is saved into the current task TCB. This means that the R13_sys (SP) is saved. Thirdly, the stack pointer is loaded from the OS TCB of the new task. Now the SP points to the new task's stack frame which looks identical (except for the contents) to the stack frame of the task that were switched out. Lastly, the context of the new task is pulled of the stack, which means that the contents in registers (from R0 to R12) of a new task are loaded into the system mode registers as well as its address (PC) and linker register. Then the microcontroller resumes the new task.

4.2.5 Exception handling

Two types of exceptions were implemented in this porting work, the reset exception as mentioned before and the IRQ exception. In our case, a non nested interrupt handler structure was chosen since it is suitable for an initial porting stage when there are not many interrupt sources. When more sources are added, a nested interrupt handler will be needed to make execution of the AUTOSAR OS on a Raspberry Pi more efficient. This will be implemented at a later stage.

Figure 20. Nonnested interrupt handler

Figure 21. The interrupt vector List in the AUTOSAR OS

Figure 20 describes the execution flow of a non-nested interrupt handler. When an interrupt occurs (e.g. timer interrupt or SPI transmit/receive interrupt), an IRQ exception is triggered and the ARM processor disables further IRQ exceptions from occurring. Upon entry to the interrupt handler, the handler code saves the current context of non-banked registers. The handler then identifies the interrupt source according to the interrupt's number and executes the appropriate interrupt service routine (ISR). For example, SPI ISR is called when an SPI transmit interrupt occurs. In AUTOSAR OS, all ISRs have already been registered in an interrupt vector list during the OS initialization stage, based on the priority of source interrupts. The following instruction is used to registered a SPI interrupt service
routine: ISR_INSTALL_ISR2("SPI0", Spi_Isr, BCM2835_IRQ_ID_SPI, 15, 0); Here, the "SPI0" is the name of the timer interrupt service routine (Spi_Isr). BCM2835_IRQ_ID_SPI is the name of the interrupt source. The number 15 is the priority of the SPI interrupt in the AUTOSAR OS. In addition, the interrupt vector list is shown in Figure 21, which has two ISRs registered in. Finally, upon return from the ISR, the interrupt handler restores context. Finally, interrupts are enabled again and the task which was interrupted can continue its execution.

4.3 Linker script

While writing a multi-file program, each file is assembled individually into object files. Then the linker uses a linker script to combine a number of objects and archive files, relocate their data and tie up symbol references to make a final executable file, as shown in Figure 22. A linker script is used to describe how the sections in the input source files are mapped into the output file, and to control the memory layout of the output file [27]. To write a linker script, a linker command language is used. In this thesis, a simple linker script is implemented, which is described in the following sections.

4.3.1 ENTRY

The ENTRY command is the first instruction of the linker script. The ENTRY command takes one argument like this: ENTRY(_start). This defines the entry point of the linked program with the symbol name “_start”.

4.3.2 OUTPUT_FORMAT

The OUTPUT_FORMAT command has one or several arguments. It specifies the output format of the executable. In this thesis, it is like this:

OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm","elf32-littlearm")

There are three arguments in this instruction, which means that the output executable file is of the ELF type, follows the ARM architecture instructions and Little-Endian (default).

4.3.3 MEMORY

The MEMORY command declares one or more memory regions with attributes specifying whether the region can be written, read or executed. This is mostly used in embedded systems where different regions of address space may contain different access permissions. In this thesis, it is defined like this:
MEMORY
{
  ram: org = 0x8000, len = 0x06000000 - 0x20
}

It defines the start address (0x8000) and the size of RAM (95M) for placing the executable file.

4.3.4 SECTIONS

The SECTIONS command is used to describe the memory layout of the output file, as shown in the following code snippet.

```
SECTIONS
{
  . = 0;
  .text :
  {
    _text_start_ = .;
    *(.text)
    _text_end_ = .;
  } > ram
  .data :
  {
    _data_start = .;
    *(.data)
    _data_end = .;
  } > ram
  .bss :
  {
    _bss_start = .;
    *(.bss)
    _bss_end = .;
  } > ram
}
```

The first line inside the SECTIONS command sets the value of the location counter, which is the start address for allocating the “.text” part in RAM - zero in the example. It is incremented by the size of the section. The second line defines an output section, .text. The .text section contains all the binary instructions that C and assembly programs were compiled/assembled into and are typically put in program memory. The expression *(.text) refers to all .text input sections in all input files. Here, the start address and end address are not specified, which are arranged by the linker itself. The .data section contains all initialized global and static variables, as mentioned before. Then last is .bss section, which consists of uninitialized static and global variables. All these sections are saved in RAM which is specified by the mark “>ram”. This is the linker script structure used in this thesis.
Chapter 5

HARDWARE DRIVER DEVELOPMENT

This chapter describes the process of developing a SPI driver according to the AUTOSAR SPI standard requirements. For helping readers easily understand this process, some basic concepts about SPI are introduced first, e.g. SPI communication mechanism. Then some SPI related concepts and terms used in AUTOSAR standard are presented, e.g. channels, jobs and sequences. At last, the implemented functions are described, e.g. Spi_init() and so on.

5.1 Serial Peripheral Interface Bus

Serial Peripheral Interface (SPI) bus is a four-line, synchronous, serial bus that is widely used to connect a microcontroller to peripheral circuit chips. Each device has one input line and one output line, and data is exchanged in full-duplex mode. SPI operates in a master-slave topology where the microcontroller is the master while the peripheral chips respond as slaves.

An SPI protocol specifies 4 signal wires:

- Master Out Slave In (MOSI) - MOSI signal is generated by Master, and recipient is the Slave.
- Master In Slave Out (MISO) - MISO signal is generated by slaves, and recipient is the Master.
- Serial Clock (SCLK or SCK) - SCLK signal is generated by the Master to synchronize data transfer between the master and the slave.
- Chip Select (CS) - CS signal is generated by Master to select individual slave/peripheral devices.

A typical hardware setup for SPI communication is given in Figure 23. The communication is initiated by the master all the time. Normally, there are three steps for setting up the communication between a master and a slave. Firstly, the master configures the clock, using a frequency, which is less than or equal to the maximum frequency that the slave device supports. Secondly, the master selects the desired slave for communication by asserting the Chip Select (CS) pin of that particular slave-peripheral. At last, a full duplex
data transmission can occur during each clock cycle. That means that the master sends a bit on the MOSI line; the slave reads it from that same line; the slave sends a bit on the MISO line; the master reads it from that same line.

### 5.2 SPI specification in the AUTOSAR standard

The AUTOSAR standard defines a hierarchical structure of sequences, jobs and channels [28] to describe data transmission process on a SPI bus. A sequence contains one or several jobs, which are in turn composed of channels with the same CS signal. A channel is the actual place holder for the transmitted data.

In Figure 24 [28], an example of SPI communication, packaged in the AUTOSAR way, is shown. Transmission of a sequence is initiated via an API call, such as Spi_SyncTransmit. The sequence consists of two jobs, job n and m. At first, job n arbitrates the bus. After the data transfer of channel x is finished, the next channel of job n gets started without releasing the bus. When the transmission of both channels is finished, the bus is released by job n and job m starts to transmit data until the sequence a is finished.

![SPI transmission structure according to the AUTOSAR specifications](image)

### 5.3 Types definition

To configure sequences, jobs and channels mentioned above, their supporting data structures should be implemented. Here, five data structures are introduced based on some basic types from the AUTOSAR standard definition.

#### 5.3.1 Spi_ConfigType

The Spi_ConfigType (as shown in the following code) is the main data structure that contains the others and includes configurations for channels, jobs, sequences, and external device structures (Spi_ChannelConfig, Spi_JobConfig, Spi_SequenceConfig, and Spi_ExternalDevice respectively). In this way, all the necessary information can be passed in one block to the Spi_Init function for the initialization of the SPI Driver.

```c
typedef struct Spi_Config
{
  uint8 SpiMaxChannel;
  uint8 SpiMaxJob;
  uint8 SpiMaxSequence;
  // All data needed to configure one SPI-channel
  const struct Spi_ChannelConfig * SpiChannelConfig;
} Spi_Config;
```
// The communication settings of an external device.
const struct Spi_ExternalDevice * SpiExternalDevice;
// All data needed to configure one SPI-Job
const struct Spi_JobConfig * SpiJobConfig;
// All data needed to configure one SPI-sequence
const struct Spi_SequenceConfig * SpiSequenceConfig;
} Spi_ConfigType;

5.3.2 Spi_SequenceConfig

The data structure of Spi_SequenceConfig (as shown in the following code) should contain a flag used to specify whether this sequence can be interrupted by another sequence, a variable to specify the sequence's name, a sequence finish end notification function and an array of pointers to jobs in this sequence.

typedef struct Spi_SequenceConfig
{
    // allows or not this Sequence to
    // be suspended by another one.
    unsigned SpiInterruptibleSequence;
    // a reference to a notification function.
    void (*SpiSeqEndNotification)();
    // Spi_SequenceType is unsigned char type
    Spi_SequenceType SpiSequenceId;
    // A sequence references several jobs
    // Spi_JobType is unsigned short type
    Spi_JobType JobAssignment[SPI_MAX_JOB+1];
} Spi_SequenceConfigType;

5.3.3 Spi_JobConfig

The data structure of Spi_JobConfig should contain the job's name, a parameter to specify the priority of the job, an array of pointers to channels in this job, a job finish end notification function and a parameter that identifies the SPI hardware allocated to this job.

typedef struct Spi_JobConfig
{
    Spi_JobType SpiJobId;
    // the symbolic name
    uint32 SpiHwUnit;
    // a reference to a notification function.
    void (*SpiJobEndNotification)();
    // Priority of the Job, range from 0 to 3
    unsigned SpiJobPriority;
    // A job references several channels.
    Spi_ChannelType ChannelAssignment[SPI_MAX_CHANNEL+1];
    // Reference to the external device used by this job
    Spi_ExternalDeviceType Type DeviceAssignment;
} Spi_JobConfigType;

5.3.4 Spi_ChannelConfig

The data structure of Spi_ChannelConfig (as shown in the following code) should include parameters to describe the channel's name, the type of buffer to be used for this channel (either an external or internal buffer), the width of a transmitted data unit (8 bits or 16 bits), the maximum size of data buffers in case of external/internal channels, and a flag to define the first starting bit (LSB or MSB) for the transmission.

The AUTOSAR standard defines two kinds of channel buffer, one is the Internally buffered Channels (IB) and the other is Externally buffered Channels (EB). The IB to transmit/receive data is provided by the drivers like SPI driver, internal flash driver, etc. The (EB) to transmit/receive data is provided by the user.
typedef struct Spi_ChannelConfig {
    // Symbolic name
    Spi_ChannelType SpiChannelId;
    // Buffer usage with EB/IB channel
    Spi_BufferType SpiChannelType;
    // the width of a transmitted data unit.
    uint32 SpiDataWidth;
    // the default value to transmit.
    uint32 SpiDefaultData;
    // the maximum size of data EB buffers
    Spi_NumberOfDataType SpiEbMaxLength;
    // the maximum number of data IB buffers
    Spi_NumberOfDataType SpiIbNBuffers;
    // defines the first starting
    // bit for transmission.
    Spi_TransferStartType SpiTransferStart;
} Spi_ChannelConfigType;

5.3.5 Spi_ExternalDevice

The data structure Spi_ExternalDevice (as shown in the following code) which is used to describe an external device, contains the following parameters: the communication baudrate; a symbolic name to identify the CS used for this job; the active polarity of Chip Select (standard high or standard low); chip select functionality is on/off; the SPI data shift edge (data shift with leading or trailing edge); the SPI shift clock idle level (shift clock idle low or idle high); the timing between clock and chip select.

typedef struct Spi_ExternalDevice {
    // the communication baudrate
    uint32 SpiBaudrate;
    // Symbolic name to identify the CS used for this job
    uint32 SpiCsIdentifier;
    // defines the active polarity of Chip Select.
    uint8 SpiCsPolarity;
    // defines the SPI data shift edge.
    Spi_EdgeType SpiDataShiftEdge;
    // enables or not the Chip Select handling functions.
    uint8 SpiEnableCs;
    // defines the SPI shift clock idle level.
    uint8 SpiShiftClockIdleLevel;
    // Timing between clock and chip select
    uint32 SpiTimeClk2Cs;
    // Job finish end notification function
    void (*SpiCsCallback)(int);
} Spi_ExternalDeviceType;

5.4 High level SPI functions in AUTOSAR

There are 14 standard functions defined in the SPI driver specification of AUTOSAR standard and the porting job included implementation of these functions. In this thesis, 9 of them are implemented and described here, which are needed to set up the communication between the microcontroller and an external device.

5.4.1 SPI_Init

The function Spi_Init provides the service for SPI initialization. The flow chart of this function is shown in Figure 25. In details, there are three steps to achieve the initialization. Firstly, data structures stored in SPI_ConfigType are initialized, including sequence, job,
and channel structures. Secondly, configure the SPI controller of the processor (BCM2835 in our case). This means that the SPI relevant registers are initialized according to the data in Spi_ExternalDevice. Also, a ISR is registered into the system during this sub-stage. At the end of the Spi_Init function, the state of the SPI driver is set to SPI_IDLE, while the result of SPI transmission is set to a default value (SPI_SEQ_OK and SPI_JOB_OK). The initialization of the SPI module happens at the AUTOSAR OS initialization stage.

5.4.2 SPI_DeInit

Figure 26 shows the flow chart for the function SPI_DeInit. It provides the service to de-initialize the SPI driver, which is to put already initialized microcontroller SPI peripherals into the same state when the device is powered on. Otherwise, the function SPI_DeInit returns failed if the status of the SPI Driver is BUSY.

5.4.3 SPI_SetupEB

The function Spi_SetupEB provides the service to setup external buffers and data lengths for a given channel. Figure 27 shows the flow chart for this function. It takes 4 input parameters, which are the specified channel, pointers to the source and destination data buffers, and the length (in bytes) of the data to be transmitted and/or received. The execution flow of SPI_SetupEB is like this: firstly, it checks whether the SPI module is activated and whether the channel is valid. If so, the function Spi_SetupEB shall update the buffer pointers and length attributes of the specified channel with the provided values.

5.4.4 Spi_SyncTransmit

The function Spi_SyncTransmit provides the synchronous transmission service to transmit data on the SPI bus. This means that the transmission service is blocked during the ongoing transmission until the transmission is finished. The flow chart of this function is shown in Figure 28(a). When this function is called, a sequence is said to be in transmission which means that all the jobs and channels that belong to this sequence are being processed. At the same time, the SPI Driver status is set to SPI_BUSY, the Sequence status is set to SPI_SEQ_PENDING and the Job status is set to SPI_JOB_PENDING by calling the API service SPI_GetStatus (see 5.4.6), Spi_GetSequenceResult(see 5.4.7) and Spi_GetJobResult(see 5.4.8). After that, the data in each channel will be transmitted or received by calling the SPI driver interface in the microcontroller layer.

5.4.5 Spi_AsyncTransmit
The function Spi_AsyncTransmit provides the asynchronous transmission service to transmit data on the SPI bus. The flow chart of this function is shown in Figure 28(b). An asynchronous transmission means that the user calling the transmission service is not blocked when the transmission is ongoing. This is the main difference with the function Spi_SyncTransmit. To achieve this, the data transmission and reception must be based on SPI interrupts.

![Flow chart of Spi_AsyncTransmit and Spi_SyncTransmit](image)

**Figure 28. Spi_SyncTransmit and Spi_AsyncTransmit**

5.4.6 Spi_GetStatus

The function Spi_GetStatus offers the functionality to specify the status of the SPI Driver software module, that is, BUSY or IDLE. If a sequence is being processed, this function returns a flag called SPI_BUSY. Otherwise, it returns a flag called SPI_IDLE to show that the SPI module is available.

5.4.7 Spi_GetSequenceResult

The function Spi_GetSequenceResult returns the last transmission result of the specified sequence. The return value can be either succeeded (SPI_SEQ_OK) or failed (SPI_SEQ_FAILED).

5.4.8 Spi_GetJobResult

The function Spi_GetJobResult returns the last transmission result of the specified job. The return value can be either succeeded (SPI_JOB_OK) or failed (SPI_JOB_FAILED).

5.4.9 Spi_GetHWUnitStatus

The function Spi_GetHWUnitStatus return the status of the specified SPI hardware microcontroller peripheral: BUSY or IDLE.
Chapter 6

EXPERIMENTAL SETUP

This chapter is devoted to the experimental setup and result analysis. In the first section, a kernel test shows that the kernel development work, see Chapter 4, was done correctly. The second section presents a CAN communication test to demonstrate the SPI drive development work, see Chapter 5.

6.1 Kernel test

The purpose of this experiment is to test some basic OS functionally such as the initialization of the AUTOSAR OS, and the context switch of the AUTOSAR OS scheduling.

6.1.1 Experiment description

This test designs 4 Tasks, 3 of them are with the same priority and an IDLE Task with the lower priority. The execution process is as follows:

1. Task1 executes; Task1 activates Task2; Task1 changes to waiting state and waits for the next execution point.
2. Task2 executes; Task 2 activates Task3; Task 2 changes to waiting state and waits for the next execution point.
3. Task3 executes and terminates.

6.1.2 Result

Figure 29 shows the result of this test, observed by an oscilloscope. Channels 1, 2, and 3 present the execution flows of Tasks 1, 2, and 3 respectively, while channel 4 shows the IDLE Task. Normally, the channels output high level (+5 voltage) during the task's runtime, while a low level output (0 voltage) means that the task is not executing. The exception is channel 4, the IDLE Task. When this channel keeps high level for a long time, this does not mean that the task is running but that it is preempted by other tasks with higher priority. In addition, it continues executing after all the high priority tasks have gone to the waiting state.

Figure 30 shows the Gantt chart for this experiment. Firstly, Task1 executes and then goes to the waiting state. Next, Task 2 starts executing and then goes to the waiting state. Thirdly, Task 3 executes and finishes its job. Lastly, IDLE Task executes several times until it is preempted by Task1.

This test proofs that the porting of the AUTOSAR OS was successfully initialized and that the context switch is working.
6.2 CAN communication test

Nowadays, Controller Area Network (CAN) is one of the main communication methods in vehicles. In order to demonstrate our porting work, we set up a CAN bus communication system between two Raspberry Pis. Since a Raspberry Pi lacks a CAN interface itself, external CAN bus boards were used as intermediaries, connected to the SPI interface of the Raspberry Pis, as shown in Figure 31.

6.2.1 CAN controller board

The CAN bus board was made up of two main components: a CAN controller (MCP2515) and a CAN transceiver (MCP2551), as shown in Figure 32. The MCP 2515 CAN Controller can be added to a CAN bus system where CAN driver is not already embedded into the
processor, e.g. the processor of Raspberry Pi. The device uses a high speed SPI protocol to configure the CAN controller to allow the system to transmit and receive CAN messages.

The MCP2551 CAN transceiver is an interface between the CAN controller and the CAN bus. It converts between the logic levels of the CAN controller and the differential signal levels of the CAN bus.

6.2.2 Experiment description

The experiment is designed like this: one Raspberry Pi is in charge of sending CAN messages and the other one is in charge of receiving them, through two CAN bus boards, as shown in Figure 31. Figure 33 (a) shows the flow chart of the CAN communication initialization, CAN message creation and CAN message sending. As can be seen from the Figure 33 (a), before a CAN message can be transmitted, CAN controllers must be initialized. This includes resetting the MCP2515 chips, setting the CAN communication mode, configuring the speed of the CAN bus, as shown in Figure 33 (b). All this configuration data is transmitted from the Raspberry Pis to the CAN controllers.

Once the setup is completed, i.e. the CAN initialization task has returned successfully, the scheduler switches from initialization to communication tasks and data can be sent. Figure 33 (c) shows the CAN_send_message which consists of getting a free transmit buffer from the CAN controller, writing CAN message to that free buffer and sending them. And the Figure 33 (d) shows the execution flow of the reading of a CAN message.

The communication data between a Raspberry Pi and a CAN controller is sent using the AUTOSAR compliant SPI functions, see Section 5.5. The following instructions show an example of how the SPI functions were used.

```c
uint8 cmdbuf[] = {0xC0};
Spi_SetupEB(SPI_CH_CMD, cmdbuf, NULL, sizeof(cmdbuf));
test = Spi_AsyncTransmit(SPI_SEQ_CMD);
```

Here, a CAN controller reset command (0xC0) sequence is to be transmitted on the SPI bus. It is assumed that Spi_Init has already been called during the initialization stage. First, an external buffer needs to be set for the command channel, with the command data in the source data buffer. The destination buffer is set to NULL because no data needs to be read back from the SPI bus. Finally, the command sequence is transmitted.

6.2.3 Result

A PC terminal was used to display the result information of the test, as shown in Figure 34. In order to make the CAN message (A CAN Message is created by user, which is transmitted on the CAN-bus. It can be numbers or characters.) continuous sending and receiving, the test code was placed into a for-loop. In Figure 34(a), it shows the following:

1. This is a CAN-bus test.
2. CAN controller sends a message
3. CAN message is successfully transmitted.

The Figure 34 (b) shows:

1. This is a CAN-bus test.
2. CAN controller successfully receives a CAN message.
3. The CAN messages are 00000001, 00000002 and 00000003 (these messages are what the CAN transmitter sends, which are created by the user, as shown in Figure 33 (a)).
As explained above, the CAN communication process consists of a CAN initialization sequence, CAN message creating sequence, CAN message sent sequence and CAN message reading sequence. All these sequences are sent through the AUTOSAR compliant SPI functions which were developed during this thesis, see Chapter 5. Therefore, the successful CAN communication demonstrates the validity of the AUTOSAR SPI driver development work.
Chapter 7

CONCLUSIONS

7.1 Conclusion

In this report, an approach for porting AUTOSAR to an ARM based platform, Raspberry Pi, was presented. Based on ARM architecture specifications, four main kernel development processes were performed, including initialization, memory modeling, exception handling and context switching, which allowed an AUTOSAR compliant OS to start up on a Raspberry Pi. Subsequently, the development process of a SPI driver/handler was documented to demonstrate the steps that are needed to develop a driver for a specific hardware that meets the AUTOSAR standard.

In addition, in order to demonstrate the practical value of our work, a CAN bus communication system was built, allowing two Raspberry Pis to successfully communicate with each other through a CAN bus. The actual connection between the Raspberry Pis and the CAN system was done through the above mentioned SPI interface.

Therefore, we believe that this work is of value for researchers and developers that need to port AUTOSAR to different embedded platforms, providing them with a base of experiences to speed up their development. Also, this work marks the initiation of an open hardware platform for research and experimentation on advanced automotive ECUs.

7.2 Future work

In the future, we plan to extend this work by adding other common I/O and communication functionality, including support for the serial port, PWM, and Ethernet communication, which is expected to be the next communication standard in automotive applications and is included in AUTOSAR 4.0.

Once fully AUTOSAR compliant Raspberry Pis are up and running, they will be interconnected to simulate a network of vehicle ECUs. This will allow to run high level automotive application software in a realistic lab environment, providing opportunities to test completely new concepts. One of such concepts that we believe has an important potential is related to the federations of embedded systems (FES) [29], and the means of easily installing new software in the AUTOSAR framework on running vehicles [30], similarly to how it is done with apps in smart phones. With the experimental platform in place, it will be possible to take the step from theoretical visions of FESs to actual demonstrations and evaluations of the concepts.
Chapter 8

ACKNOWLEDGEMENTS

During 6 months hard work in this thesis, I finish it on time. Along the way, I get a lot of help and support from my friends, family and professors. At the moment, I am deeply grateful to all of you.

My deepest gratitude goes first and foremost to Avenir Kobetski, my supervisor, for his constant encouragement and guidance. He did put a lot of effort into this thesis and has walked me through all the stages of this work. Without his great contribution to the paper — “Porting an AUTOSAR-Compliant Operating System to a High Performance Embedded Platform”, it cannot be published on time. Without his philosophical guidance — “keep the inner peace”, I think I had already stuck in the thesis. Without his consistent and illuminating instruction, this thesis would not have reached its present form. And his meticulous working attitude pushed me to a new altitude of my study.

Second, I would like to express my heartfelt gratitude to Professor Jakob Axelsson and Expert Eilert Johansson, who gave me a chance to do this thesis and instructed and help me a lot during this thesis. Eilert’s selfless assistance in hardware test leaded me out of the darkest period of time during the implementation.

Third, I would thank to Professor Damir Isovic from Mälardalen University and Professor Huifeng Wang from East China University of Science and Technology, who made my two years’ study in Sweden possible and wonderful.

Last, my thanks would go to my beloved family for their loving considerations and great confidence in me during these two years. With my grandfather’s letters and my parents’ home video meetings, my homesickness did reduce a lot.

Västerås, July 2013
Shuzhou Zhang
Chapter 9

REFERENSER / REFERENCES


[5] Jean-Luc Béchennec, Mikael Briday, Sébastien Faucou and Yvon Trinquet, Trampoline - an open source implementation of the osek/vdx rtos speci_cation, In 11th Int. Conf. on Emerging Technologies and Factory Automation (ETFA'06), Prague, 2006


[19] OSEK Group, OSEK/VDX operating system specification, version 2.2.3, 2005


A. APPENDIX – start up file

.extern __ram_end__
.extern __und_stack_size__
.extern __abt_stack_size__
.extern __fiq_stack_size__
.extern __irq_stack_size__
.extern __svc_stack_size__
.extern _bss_start
.extern _bss_end
.extern IrqHandler
.extern Irq_Entry
.extern main

.set   MODE_USR, 0x10
.set   MODE_FIQ, 0x11
.set   MODE_IRQ, 0x12
.set   MODE_SVC, 0x13
.set   MODE_ABT, 0x17
.set   MODE_UND, 0x1B
.set   MODE_SYS, 0x1F

.set   I_BIT, 0x80
.set   F_BIT, 0x40

.section .init
.code 32
.balign 4
.globl _start
_start:
//All the following instruction should be read as: Load the address at
//symbol into the program counter.
ldr  pc,reset_handler    //Processor Reset handler
ldr  pc,undefined_handler //Undefined instruction handler
ldr  pc,swi_handler     //Software interrupt / TRAP (SVC)
ldr  pc,prefetch_handler //Prefetch/abort handler
ldr  pc,data_handler           //Data abort handler
nop
ldr  pc,irq_handler     //IRQ handler
ldr  pc,fiq_handler     //Fast interrupt handler

.code 32
.balign 4
/* Here we create an exception address table! This means that reset/hang/irq can
be absolute addresses*/
reset_handler:
.word ResetHandler
undefined_handler:
   .word UndHandler
swi_handler:
   .word SwiHandler
prefetch_handler:
   .word PrefetchHandler
data_handler:
   .word AbortHandler
   .word 0
irq_handler:
   .word IrqHandler
fiq_handler:
   .word FiqHandler
.code 32
.balign 4
.global ResetHandler

ResetHandler:
//In the reset handler, we need to copy our interrupt vector table to
//0x0000, its currently at 0x8000
mov r0,#0x8000
mov r1,#0x0000
ldmia r0!,{r2,r3,r4,r5,r6,r7,r8,r9}
stmia r1!,{r2,r3,r4,r5,r6,r7,r8,r9}
ldmia r0!,{r2,r3,r4,r5,r6,r7,r8,r9}
stmia r1!,{r2,r3,r4,r5,r6,r7,r8,r9}
ldr r0, =__ram_end__
/* Undefined */
msr CPSR_c, #MODE_UND | I_BIT | F_BIT
mov sp, r0
ldr r1, =__und_stack_size__
sub r0, r0, r1
/* Abort */
msr CPSR_c, #MODE_ABT | I_BIT | F_BIT
mov sp, r0
ldr r1, =__abt_stack_size__
sub r0, r0, r1
/* FIQ */
msr CPSR_c, #MODE_FIQ | I_BIT | F_BIT
mov sp, r0
ldr r1, =__fiq_stack_size__
sub r0, r0, r1
/* IRQ */
msr CPSR_c, #MODE_IRQ | I_BIT | F_BIT
mov sp, r0
ldr r1, =__irq_stack_size__
sub r0, r0, r1
/* Supervisor */
msr CPSR_c, #MODE_SVC | I_BIT | F_BIT
mov sp, r0
ldr r1, =__svc_stack_size__
sub r0, r0, r1
/* System */
msr CPSR_c, #MODE_SYS | I_BIT | F_BIT
mov sp, r0
mov r0, #0
ldr r1, =__bss_start
ldr r2, =__bss_end
bssloop:
cmp r1, r2
strlo r0, [r1], #4
blo bssloop
 /* msr CPSR_c, #0x1F*/
bl main
b __main_exit_handler

.weak __main_exit_handler
.globa __main_exit_handler
__main_exit_handler:

.loop: b .loop

.code 32
.balign 4
.weak UndHandler

UndHandler:
     .weak SwiHandler

SwiHandler:
     .weak PrefetchHandler

PrefetchHandler:
     .weak AbortHandler

AbortHandler:
     .weak FiqHandler

FiqHandler:

     .global _unhandled_exception

_unhandled_exception:
     b _unhandled_exception
B. APPENDIX – context switch

`#define _ASSEMBLER_
#include "asm_offset.h"
#include "arch_stack.h"
.extern Os_Sys
.extern Irq_Entry
#define IRQ_ENABLE()   cpsie  i
#define IRQ_DISABLE()  cpsid  i
#define REG_SAVE r0-r12

.section .text
.code  32
.balign 4
.global IrqHandler
IrqHandler:
/* Setup return address. This requires subtraction from LR.*/
sub   lr, lr, #4
Svc_Handler:
/* Store return stuff on system mode's stack*/
srsdb sp!, #0x1f
/*Switch to system mode.*/
cpsid i, #0x1f
/*Push registers on stack*/
push    {r0-r12,lr}
sub   sp,sp,#C_SIZE
mov   r4,#LC_PATTERN
str   r4,[sp,#C_CONTEXT_OFFS]
/*Put stack as first arg to Irq_Entry*/
mov  r0,sp
/* When at interrupt nest count = 0, load interrupt stack*/
ldr      r4,=Os_Sys
ldr      r5,[r4,#SYS_INT_NEST_CNT]
cmp      r5, #0
bgt      arggg
ldr      sp,[r4,#SYS_INT_STACK]
arggg:
IRQ_DISABLE()
bl      Irq_Entry
IRQ_ENABLE()
mov     sp, r0   /*pop from returned stack*/
/* Do a normal exception return */
add     sp,sp,#C_SIZE
/*Restore registers*/
pop     {r0-r12,lr}
// Return using stuff from stack.
rfeia sp!
/**
 * Os_ArchSetSpAndCall
 * @param sp Pointer to the stack
 * @param f  Pointer to the function to call
 */
.global Os_ArchSetSpAndCall
.type Os_ArchSetSpAndCall, %function
Os_ArchSetSpAndCall:
    mov    sp,r0
    mov    lr,r1
    bx     lr

/**
 * Os_ArchSwapContext
 *
 * @param r0 - pcb for old task
 * @param r1 - pcb for new task
 *
 */
.global Os_ArchSwapContext
.type Os_ArchSwapContext, %function
Os_ArchSwapContext:
    // Save function call registers
    push    {r0-r12,lr}
    // Store the context frame
    sub     sp,sp,#C_SIZE
    // Save small-context indicator
    mov     r4,#SC_PATTERN
    str     r4,[sp,#C_CONTEXT_OFFS]
    // store old stack for old task
    mov  r4,sp
    str     r4,[r0,#PCB_STACK_CURR_P]
    // Flow down
    // R1 - new PCB
    .global Os_ArchSwapContextTo
    .type Os_ArchSwapContextTo, %function
Os_ArchSwapContextTo:
    // Get stack for new task
    ldr    r2,[r1,#PCB_STACK_CURR_P]
    mov     sp,r2
    // Set new current pcb
    ldr    r5,= Os_Sys
    str    r1,[r5,#SYS_CURR_PCB_P]
    // Restore C context
    ldr    r6,[sp,#C_CONTEXT_OFFS]
    cmp    r6,#SC_PATTERN
    beq     os_sc_restore
    cmp    r6,#LC_PATTERN
    beq     os_lc_restore
    os_stack_problem:
        os_stack_problem

    /* Restore the small context. Cases:
     * "Normal" context switch between processes.
     * We are in handler mode (this task preemted another task in interrupt
     * context). We need to terminate handler mode ( set to LR=0xffff_fff9 )
     */
    os_sc_restore:
        add     sp,sp,#C_SIZE
        pop     {r0-r12,lr}
        bx      lr
/* Restore the large context. Cases: */
/* 1. Directly from IRQ_Handler() */
/*  (the preempted task got swapped in directly) */
/* 2. The preempted task, got preempted by a task and */
/*     we have already returned from handler mode. */

os_lc_restore:
    /* Do a normal exception return */
    add    sp,sp,#C_SIZE
    // Restore registers
    pop    {r0-r12,lr}
    /* Enable interrupts */
    // Return using stuff from stack.
    rfeia sp!
C. APPENDIX – linker script

OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm","elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
__und_stack_size__ = 0x0004;
__abt_stack_size__ = 0x0004;
__fiq_stack_size__ = 0x0010;
__irq_stack_size__ = 0x0080;
__svc_stack_size__ = 0x0004;
__sys_stack_size__ = 0x0400;
__stacks_total_size__ = __und_stack_size__ + __abt_stack_size__ +
__fiq_stack_size__ + __irq_stack_size__ + __svc_stack_size__ + __sys_stack_size__;
MEMORY
{  
  ram : org = 0x8000, len = 0x06000000 - 0x20
}
__ram_start__ = ORIGIN(ram);
__ram_size__ = LENGTH(ram);
__ram_end__ = __ram_start__ + __ram_size__;
SECTIONS
{  
  . = 0;

  .text : ALIGN(16) SUBALIGN(16)
  {  
    .text = .;
    KEEP(*(.init))
    *(.text)
    *(.text.*)
    *(.rodata)
    *(.rodata.*)
    *(.glue_7t)
    *(.glue_7)
    *(.gcc*)
    *(.ctors)
    *(.dtors)
  } > ram

  .ARM.extab : {*(.ARM.extab* .gnu.linkonce.armextab.*)}
  __exidx_start = .;
  .ARM.exidx : {*(.ARM.exidx* .gnu.linkonce.armexidx.*)} > ram
  __exidx_end = .;
  .eh_frame_hdr : {*(.eh_frame_hdr)}
  .eh_frame : ONLY_IF_RO {*(.eh_frame)}
  . = ALIGN(4);
  _etext = .;
  ._textdata = _etext;
  .data :
  {  
    .data = .;
    *(.data)
    . = ALIGN(4);
    *(.data.*)
    . = ALIGN(4);
    *(.ramtext)
    . = ALIGN(4);
    _edata = .;
  } > ram
  .bss :
  {  
    _bss_start = .;
* (.bss)
  . = ALIGN(4);
* (.bss*)
  . = ALIGN(4);
* (COMMON)
  . = ALIGN(4);
  _bss_end = .;
} > ram

  .uninit ALIGN(0x10) (NOLOAD):
  { *(.winidea_port .ramlog .dem_eventmemory_pri); }  > ram
  /* Stabs debugging sections. */
  .stab  0 : { *(.stab) }
  .stabstr 0 : { *(.stabstr) }
  .stab.excl 0 : { *(.stab.excl) }
  .stab.exclstr 0 : { *(.stab.exclstr) }
  .stab.index 0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }
  .comment 0 : { *(.comment) }

  /* DWARF debug sections.
        Symbols in the DWARF debugging sections are relative to the beginning
        of the section so we begin them at 0. */
  /* DWARF 1 */
  .debug 0 : { *(.debug) }
  .line  0 : { *(.line) }

  /* GNU DWARF 1 extensions */
  .debug_srcinfo 0 : { *(.debug_srcinfo) }
  .debug_sfnames 0 : { *(.debug_sfnames) }

  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges 0 : { *(.debug_aranges) }
  .debug_pubnames 0 : { *(.debug_pubnames) }

  /* DWARF 2 */
  .debug_info 0 : { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev 0 : { *(.debug_abbrev) }
  .debug_line 0 : { *(.debug_line) }
  .debug_frame 0 : { *(.debug_frame) }
  .debug_str 0 : { *(.debug_str) }
  .debug_loc 0 : { *(.debug_loc) }
  .debug_macinfo 0 : { *(.debug_macinfo) }

  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 : { *(.debug_weaknames) }
  .debug_funcnames 0 : { *(.debug_funcnames) }
  .debug_typenames 0 : { *(.debug_typenames) }
  .debug_varnames 0 : { *(.debug_varnames) }

}

PROVIDE (end = .);
  _end = .;

  __heap_base__  = _end;
  __heap_end__   = __ram_end__ - __stacks_total_size__;
  __main_thread_stack_base__ = __ram_end__ - __stacks_total_size__;
D. APPENDIX – SPI driver – code snippet

```c
static void Spi_InitController(void) {
    uint32 control = 0;
    IRQ_DISABLE2 |= BIT(22);
    // This can be optimized to setting two masks.
    bcm2835_gpio_fnsel(7, GPFN_ALT0);   // SPI0_CE1_N.
    bcm2835_gpio_fnsel(8, GPFN_ALT0);   // SPI0_CE0_N.
    bcm2835_gpio_fnsel(9, GPFN_ALT0);   // SPI0_MOSI.
    bcm2835_gpio_fnsel(10, GPFN_ALT0);  // SPI0_MISO.
    bcm2835_gpio_fnsel(11, GPFN_ALT0);  // SPI0_SCLK.
    control &= ~SPI_CS_REN;
    SPI0_CS = control | SPI_CS_CLEAR_TX | SPI_CS_CLEAR_RX;
    SPI0_CLK = 0x100;
    ISR_INSTALL_ISR2("SPI0", Spi_Isr, BCM2835_IRQ_ID_SPI, 15, 0);
}

static void Spi_Isr(Spi_UnitType *uPtr) {
    // write
    if (SPI0_CS & SPI_CS_DONE) {
        uint8 *count = &(Spi_DataExchange.n);
        if (*count > 0) {
            /* Fill FIFO */
            uint8 *txbuf = Spi_DataExchange.txbuf;
            while ((SPI0_CS & SPI_CS_TXD) &&  *count > 0) {
                SPI0_FIFO = Spi_DataExchange.txbuf != NULL ? *(txbuf)++ : 0;
                --*count;
            }
        } else {
            /* Deactivate Transfer and disable SPI interrupts.*/
            SPI0_CS &= ~(SPI_CS_INTD | SPI_CS_INTR | SPI_CS_TA);
            read_fifo(uPtr);
        }
    } else if (SPI0_CS & SPI_CS_RXR) { /* read */
        read_fifo(uPtr);
        Spi_Rx_FIFO(uPtr);
    }
}

static void Spi_Poll(Spi_UnitType *uPtr) {
    uint32 rx;
    // SPI0 select
    SPI0_CS &= ~SPI_CS_CS;
    // clear tx and rx
    SPI0_CS |= SPI_CS_CLEAR_TX | SPI_CS_CLEAR_RX;
    // transfer
    SPI0_CS |= SPI_CS_TA;
    uint8 *count = &(Spi_DataExchange->n);
    uint8 *txbuf = (uint8 *)(Spi_DataExchange)->txbuf;
    uint8 *rxbuf = (uint8 *)(Spi_DataExchange)->rxbuf;
    Spi_DataExchange->rn = 0;
    while(*count > 0){
        // wait for TXD
        while(!((SPI0_CS & SPI_CS_TXD)));
        // Write to FIFO
        SPI0_FIFO = Spi_DataExchange->txbuf != NULL ? *txbuf++ : 0;
        while(!((SPI0_CS & SPI_CS_RXD)));
        rx = SPI0_FIFO;
        rxbuf++;
        Spi_DataExchange->rn++;
        // Prepare for the next
    }
}
```
--*count;
}

// Wait for DONE to be set
while (!(SPI0_CS & SPI_CS_DONE));
SPI0_CS &= ~SPI_CS_TA;
//unselect
SPI0_CS |= SPI_CS_CS;