Porting an AUTOSAR-Compliant Operating System to a High Performance Embedded Platform

Shuzhou Zhang
School of Innovation, Design and Engineering
Mälardalen University
szg11002@student.mdh.se

Avenir Kobetski
Software and Systems Engineering Laboratory
SICS Swedish ICT AB
avenir@sics.se

Eilert Johansson
Software and Systems Engineering Laboratory
SICS Swedish ICT AB
eilert@sics.se

Jakob Axelsson
School of Innovation, Design and Engineering
Mälardalen University
jakob.axelsson@mdh.se

Huifeng Wang
School of Information Science and Engineering
East China University of Science and Technology
whuifeng@ecust.edu.cn

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.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Microprocessor/microcomputer applications, Real-time and embedded systems; D.4.4 [**Operating systems**]: Communications Management—*input/output, network communication*

General Terms

Design, Experimentation, Standardization.

Keywords

AUTOSAR, Raspberry Pi, Embedded Operating Systems

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 paper 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. For the sake of compliance with existing standards, the OS of choice relies on AUTOSAR, the prevailing software architecture standard in the automotive industry.

The results address 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.

The paper is organized as follows. In Section 2, some necessary background is presented. In Section 3, experiences of the porting process are detailed. Section 4 presents the experimental setup, while Section 5 shortly surveys the state of the practice, related to this work. Finally, Section 6 concludes the paper.

EWiLi'13 , August 26-27, 2013, Toulouse, FRANCE
Copyright retained by the authors.

2. BACKGROUND

In this section, basic information related to the AUTOSAR standard and the specifics of the hardware architecture behind Raspberry Pi is presented.

2.1 AUTOSAR

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) [2, 11].

AUTOSAR is a layered software architecture that decouples 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 abstraction layers (ECUAL and μCAL respectively); and complex device drivers for direct access to hardware, see Figure 1.

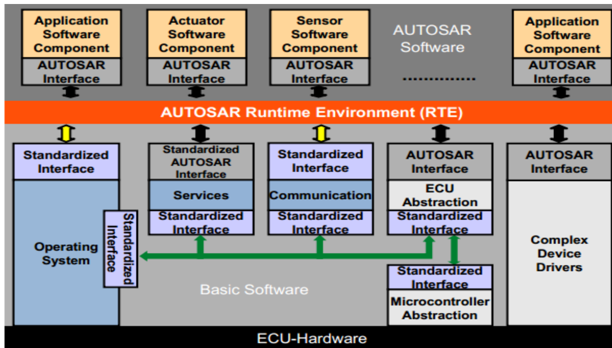


Figure 1: AUTOSAR layer structure

There exist several implementations of the AUTOSAR standard. However, most of those are commercial products which often makes them unsuitable for research purposes. In this work, ArcticCore [1] – one of the very few open-source AUTOSAR implementations – was chosen. ArcticCore contains among other a real-time OS, memory management, and ECUAL software. For some hardware architectures, mostly those that are currently popular in the automotive industry, such as Freescale MPC5xxx, also the μCAL software is implemented.

This work is based on AUTOSAR 3.2, with the focus on extending existing ArcticCore source code to make it runnable on Raspberry Pis. First, the ArcticCore OS kernel needs to be amended to allow the system to start up. Once this is done, the existing ECUAL code needs to be connected to the underlying hardware in the μCAL layer through appropriate driver development. The details and experiences of the porting work are further described in Section 3.

2.2 Raspberry Pi

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, see e.g. [13]. Other advantages of Raspberry Pis are their low cost and wide availability, which was the final rationale behind using them in our work.

A Raspberry Pi has a Broadcom BCM2835 system on chip (SoC) [10], which includes an ARM1176JZF-S 700 MHz processor, VideoCore IV GPU, and 512 megabytes of RAM. The microcontroller has a standard RJ45 based Ethernet port and some external GPIOs for adding sensors and actuators as well as other peripherals through SPI, IIC or UART ports. The performance is expected to be several magnitudes higher than a traditional microcontroller based automotive ECU.

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 flash 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.

3. THE PORTING PROCESS

This section discusses our experiences from the AUTOSAR porting process on Raspberry Pi. The first part introduces 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 modelling, exception handling and context switch.

Initialization is the first code that 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 either the tasks or the system. The method for handling interrupts and exceptions is a critical part of the architecture design of the operating system. Finally, a context switch is needed to handle scheduled tasks.

The second part focuses on the development process of a Serial Peripheral Interface (SPI) driver according to AUTOSAR standard requirements. This is just an example to show a general method for developing a hardware driver on Raspberry Pi that complies with the AUTOSAR standard.

3.1 Kernel development process

A Raspberry Pi's boot process starts with a small ROM based primary boot loader copying a file containing a secondary boot loader from SDHC card into the L2 cache of the microcontroller. The secondary boot loader initializes

SDRAM and loads a third boot loader into SDRAM which starts up the operating system. The bootloader of Raspberry Pi can be obtained from the Raspberry official forum, including bootcode.bin, loader.bin and start.elf. It boots ARM from the 0x00008000 in the ARM address space, and that is the ARM entry point and location for the first ARM instruction that we can control.

3.1.1 Initialization

Normally, there are three main stages of initializing an operating system on ARM architecture – startup, executing process control block (PCB) setup code, and executing the C initialization code [4]. During the startup stage, an exception vector table should be built. Table 1 shows such a vector for the ARM architecture. The vector table contains the addresses that the ARM processor branches to if an exception is raised.

Program execution starts with a reset exception, which triggers the reset handler. In the reset handler, the exception vector table needs to be copied to the address 0x00000000, because initially it is at the address 0x00008000 due to the boot process of the Raspberry Pi. After that the Undefined, Abort, FIQ, SVC, IRQ and System base stack registers are set up. Once the stacks are set up, the processor is switched back into the SVC mode, which allows the rest of the initialization process to continue.

In the PCB setup stage, there are four major parts of the PCB that have to be initialized: the program counter, the link register, the user mode stack, and the saved processor status register (which in the ARM case means registers r13, r14, r15, and spsr) for each task.

A timer is an important part of a real-time OS, providing a system tick for OS. In the ArcticCore OS, the system tick is defined to 1 millisecond. Hence, a timer should be enabled in the C initialization stage. When the timer is activated, a counter will start to decrement the value, which in our case is 1000, in the specified timer register. Once the value reaches zero, an interrupt is raised. In the timer interrupt service routine, the AUTOSAR OS will check the scheduler table and alarm to decide which task should be run next.

3.1.2 Memory Model

We implemented a simple memory model for the AUTOSAR OS, as shown in the Figure 2. ARM physical addresses start at 0x00000000 for the RAM. An interrupt table is built from 0x00000000 to 0x000000020. Then the code section, heap and stack are arranged. A good stack design tries to avoid stack overflow because it may cause instability in embedded systems [4, 14]. Hence, we put the interrupt stack at the top of the memory above the user stack. In this way the vector table will not be corrupted when a stack overflow occurs, and so the system has a chance to correct itself when an overflow has been identified. The address range from 0x20000000 to 0x20FFFFFF is used for microcontroller registers. The hardware interfaces can then be controlled by accessing corresponding address in this area.

3.1.3 Interrupts and Exceptions 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.

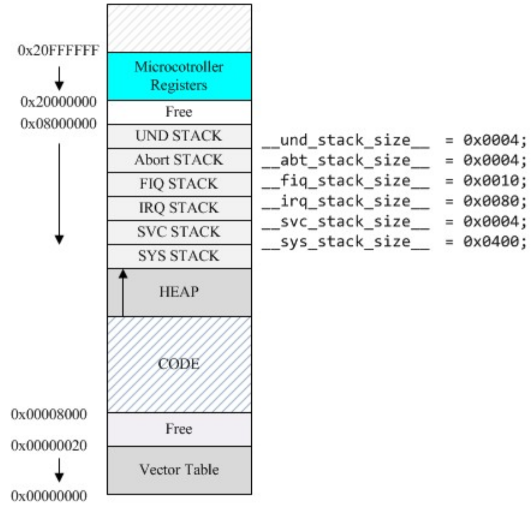


Figure 2: System memory layout

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 3 describes the process about how a non-nested interrupt handler is implemented.

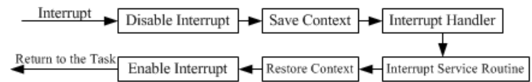


Figure 3: Nonnested interrupt handler

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 during the OS initialization stage, based on the priority of source interrupts. Upon return from the ISR, the handler restores the context. Finally, interrupts are enabled again and the task which was interrupted can continue its execution.

3.1.4 Context Switch

When a new task has been scheduled for execution, the new and old tasks have to be swapped with a context switch. To achieve this, the ARM context switch splits the activity into two stages [5, 6]. In the first stage, the state of the 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 data structure called PCB. In the second stage, the registers with data from the new task's PCB should be loaded. In doing so,

Table 1: ARM processor exception vector

Exception	Mode	Main purpose	Address
Reset	SVC	Initializes the system	0x00000000
Undefined Instruction	UND	Software emullayeration of hardware coprocessors	0x00000004
Software Interrupt	SVC	Protected mode for operating systems	0x00000008
Prefetch Abort	Abort	Memory protection handling	0x0000000C
Data Abort	Abort	Memory protection handling	0x00000010
Interrupt Request	IRQ	Interrupt request handling	0x00000018
Fast Interrupt Request	FIQ	Fast interrupt request handling	0x0000001C

the program counter from the PCB is loaded, and thus execution can continue in a new task. The new task is chosen from a task queue according to its priority.

In the porting work, two types of context switch needed to be implemented. One is the interrupt context switch that was already described in Section 3.1.3. And another one is the task context switch, shown in Figure 4. A task is assumed to run in ARM mode and uses the SYS registers. Firstly, the context of the current task should be saved onto its own stack. Secondly, the stack pointer of the old task being switched out is saved into the the current task PCB. Thirdly, the stack pointer is loaded from the OS PCB of the new task. Lastly, the context of the new task is pulled off the stack. Then the microcontroller resumes the new task.

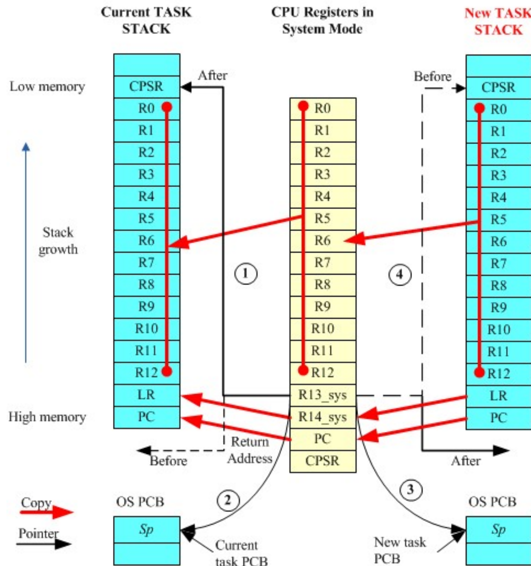


Figure 4: Task context switch

3.2 Driver development process

In this subsection the process of developing μCAL software that follows existing AUTOSAR specifications is exemplified through the development of a Serial Peripheral Interface (SPI) driver. Firstly, some SPI related concepts and terms used in the AUTOSAR standard are presented. Secondly, data structures and functions that are used for communication are described.

3.2.1 Background

The SPI (Serial Peripheral Interface) is a 4-wire synchronous serial interface. Data communication is enabled by a Chip Select wire (CS) and transmitted via a 3-wire interface consisting of serial data output (MOSI), serial data input (MISO), and a serial clock (CLOCK).

The AUTOSAR standard defines a hierarchical structure of sequences, jobs and channels [7] 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 5, 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.

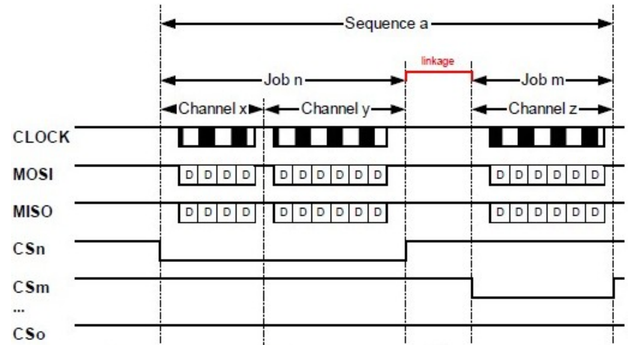


Figure 5: SPI transmission structure

3.2.2 Types definition

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

The Spi_ConfigType 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 handler/driver.

The data stucture of Spi_SequenceConfig should contain a flag used to specify whether this sequence can be interrupted by another sequence, a variable to specify the sequence's name, and an array of pointers to jobs in this sequence.

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, and a parameter that identifies the SPI hardware allocated to this job.

The data structure of Spi_ChannelConfig should include parameters to describe the channel's name, the buffer type 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.

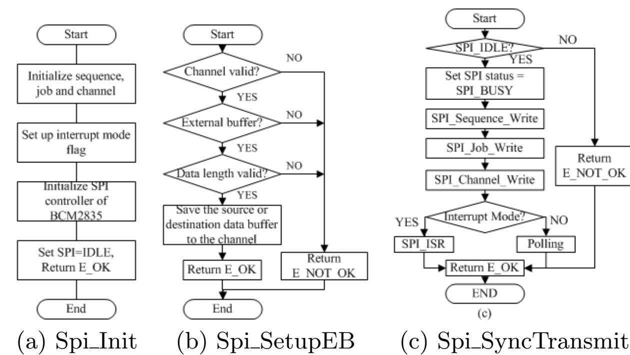
At last, a data structure to describe an external device is needed. The Spi_ExternalDevice structure should contain parameters to define the communication baudrate, the active polarity of Chip Select (standard high or standard low), the SPI data shift edge, and the SPI shift clock idle level.

3.2.3 Function

There are 14 standard functions defined in the SPI handler/driver specification of AUTOSAR standard and the porting job included implementation of these functions. Due to space limitations, only 3 main functions, needed to set up the communication between the microcontroller and an external device, will be described here.

The function Spi_Init provides the service for SPI initialization. The flow chart of this function is shown in Figure 6(a). First, all data structures stored in SPI_ConfigType are initialized, including sequence, job, and channel structures. The SPI controller of BCM2835 is also initialized here. At the end of the Spi_Init function, the state of the SPI handler/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 function Spi_SetupEB provides the service to setup external buffers and data length for a given channel. 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. Figure 6(b) shows the flow chart for this function.



(a) Spi_Init (b) Spi_SetupEB (c) Spi_SyncTransmit

Figure 6: Flow charts of three SPI functions.

The function Spi_SyncTransmit provides the service to transmit data on the SPI bus. Wheownsn 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. The data in each channel will be transmitted or received by calling the SPI driver interface in the microcontroller layer. The flow chart of this function is shown in Figure 6(c).

4. EXPERIMENTAL SETUP

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 by using 2 Raspberry Pis, as shown in Figure 7. Since Raspberry Pi lacks CAN interface itself, an external CAN bus board that contains a CAN controller (MCP2515 chip) and a CAN transceiver (MCP2551 chip) was used and controlled by a Raspberry Pi through the SPI interface, described above.

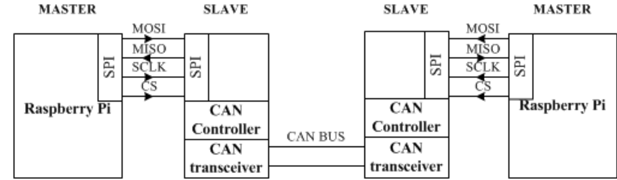


Figure 7: CAN bus communication system

Before a CAN message can be transmitted, CAN controllers must be initialized. This includes resetting MCP2515 chips, configuring the speed of CAN bus, etc. All this configuration data is transmitted from the Raspberry Pis to the CAN controllers. Once the setup is configured, the scheduler switches from initialization to communication tasks and data can be sent using the AUTOSAR compliant SPI functions, see Section 3.2.

An example of how the SPI functions were used is shown in Figure 8. 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.

```
uint8 cmdbuf[] = {0xC0};
Spi_SetupEB(SPI_CH_CMD, cmdbuf, NULL,
            sizeof(cmdbuf)/sizeof(cmdbuf));
Spi_SyncTransmit(SPI_SEQ_CMD);
```

Figure 8: An example of SPI transmission.

5. STATE OF THE PRACTICE

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, ChibiOS [3], an open-source real-time OS (RTOS) for embedded devices, was used. Chibi-OS has been ported to a number of hardware platforms, including Raspberry Pi. However, its higher

level structure differs substantially from AUTOSAR, making it only useful for our purposes as a source of inspiration.

Another open-source RTOS, Trampoline [9], 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 [12], existing FlexRay communication drivers (on the μCAL 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 goes deeper and describes the whole chain of challenges when implementing AUTOSAR μCAL 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.

6. CONCLUSIONS AND FUTURE WORK

In this paper, 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 modelling, 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.

As a consequence, 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.

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) [13], and the means of easily installing new software in the AUTOSAR framework on running vehicles [8], similarly to how it is done with apps in smartphones. 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.

7. ACKNOWLEDGMENTS

This project is supported by Vinnova (grant no. 2012-02004), Volvo Cars, and the Volvo Group.

8. REFERENCES

[1] ArcticCore product page. http://www.arccore.com/products/arctic-core/.

[2] Autosar consortium web. http://www.autosar.org.

[3] ChibiOS/RT homepage. http://www.chibios.org/.

[4] D. S. Andrew N.Sloss and C. Wright. *ARM System Developer's Guide*. Morgan Kaufmann Publishers, 500 Sansome Street, Suite 400, San Francisco, CA 94111, 2004.

[5] ARM. *ARM1176JZF-S Technical Reference Manual*. ARM, 2004.

[6] ARM. *ARM Compiler toolchain-Developing Software for ARM Processors*. ARM, 2011.

[7] AUTOSAR. *Specification of SPI Handler/Driver for AUTOSAR*. AUTOSAR Official, Version 3.2.0, 2011.

[8] J. Axelsson and A. Kobetski. On the conceptual design of a dynamic component model for reconfigurable autosar systems. In *5th Workshop on Adaptive and Reconfigurable Embedded Systems, Philadelphia*, 2013.

[9] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline - an open source implementation of the osek/vdx rtos specification. In *11th Int. Conf. on Emerging Technologies and Factory Automation (ETFA'06)*, Prague, 2006.

[10] B. Corporation. *BCM2835 ARM Peripherals*. Broadcom Corporationl, Broadcom Europe Ltd. 406 Science Park Milton Road Cambridge CB4 0WW, 2012.

[11] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, 2009.

[12] J. Jansson and J. Elgered. Autosar communication stack implementation with flexray. Technical report, Chalmers University of Technology, 2011.

[13] A. Kobetski and J. Axelsson. Federated robust embedded systems: Concepts and challenges. Technical report, Swedish Institute of Computer Science, 2012.

[14] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, UPPER SADDLE RIVER, NEW JERSEY 07458, 2007.