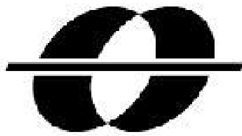

MÄLARDALEN UNIVERSITY
SWEDEN

App enabling environment to Volvo CE platforms

Thesis at
Volvo CE and SICS

Innovation, Design and
Engineering Department
Mälardalen University

by
Gerard Duff

Thesis Supervisor:
Sara Dersten

Technical Supervisor:
Dani Barkah

Examiner:
Jakob Axelsson

Start Date:
January 27th 2014

End Date:
June 13th 2014

Thesis Abstract

This thesis was submitted to the faculty of Innovation, Design and Technology, IDT, at Mälardalen university in Västerås, Sweden as a partial fulfillment of the requirements to obtain the M.Sc. in computer science, specializing in embedded systems. The work presented was carried out in the months January to June in 2014 partially in Volvo Construction Equipment, Volvo CE, Eskilstuna, and partially at Mälardalen university in Västerås.

Federated Resilient Embedded Systems Technology for AUTOSAR, FRESTA, is a collaborative project between Volvo and the Swedish Institute of Computer Science, SICS, that aims to make it possible to add third party applications to vehicle's computer systems without compromising system security and robustness. The mechanism is developed by SICS for AUTOSAR, AUTomotive Open System ARchitecture, an open standardized automotive software architecture for vehicles.

The following report documents the efforts to study and port the FRESTA mechanism to the Volvo CE platform, and develop a Java application to test the porting. The investigation will aspire to determine if it is feasible to introduce Java based third party applications to resource constrained embedded systems, without causing a deterioration in the predictability and security of the system.

Contents

1 Introduction 1

1.1 Motivation for Thesis . . . 2

1.1.1 Research Objective . . . 2

1.2 Related Work . . . 3

1.2.1 Third Party Application Development Platforms . . . 3

1.3 Conclusions and Observations . . . 6

1.4 Thesis Overview . . . 8

2 Background 9

2.1 Automotive Electronic Systems . . . 10

2.1.1 Electronic Control Units . . . 11

2.1.2 Communication . . . 11

2.2 Java . . . 14

2.2.1 Java Virtual Machine . . . 14

2.2.2 K Virtual Machine . . . 15

2.2.3 Characteristics of Java Virtual Machines . . . 15

2.3 Component Based Software Engineering . . . 19

2.3.1 Automotive Open System Architecture . . . 20

2.3.2 Volvo Software Platform . . . 21

2.4 FRESTA . . . 22

3 Porting a JVM to a Volvo CE Software Component 25

3.1 Porting Strategy . . . 26

3.1.1 Volvo CE Software Platorm Dilemma . . . 26

3.1.2 Initial Porting Plan . . . 27

3.1.3 Final Porting Plan . . . 28

3.2 Porting Implementation . . . 29

3.2.1 Development Environment . . . 29

3.2.2 Starting the Virtual Machine . . . 30

3.2.3 Setting up KVM defaults . . . 31

3.3 Porting Observations . . . 32

3.3.1 Challenges . . . 32

3.3.2 Related Porting Work . . . 33

3.3.3 Evaluation . . . 33

4 Java Applications in Resource Limited Embedded Systems 35

4.1 Application Development . . . 36

4.1.1 Human Machine Interface ECU . . . 36

4.2 Application Implementation . . . 37

4.2.1 Application Structure . . . 37

4.2.2 Preloading Applicaions 38
4.2.3 Calling Native OS Functionality 38
4.3 Volvo CE ECU Simulator 40
4.3.1 The Volvo CE Simulator Graphical User Interface . . . 41
4.3.2 The Volvo CE Simulator Display Menu 42
4.3.3 Running a Test Application in the Volvo CE Simulator 42

5 Results and Conclusions 45
5.1 Results 46
5.2 Future Work 46
5.2.1 Volvo CE Software Platform 47
5.2.2 Computing Power Increments 47
5.3 Conclusions 47
5.3.1 Final Synopsis 49

Bibliography 51

A Appendix: *Software List and Versions* **57**

Acknowledgments

First of all I would like to thank my technical supervisor Dani Barkah for all his help and support throughout the project. Any questions that I had were answered quickly and explained thoroughly.

I would like to thank Nils Erik Bånkestad for all his help and guidance throughout the thesis, and for all his feedback, suggestions and support during the thesis.

Thanks to SICS and Volvo CE for presenting me with the opportunity to complete a thesis in their project. At the SICS meetings all the members were friendly and welcoming, and working on the Volvo CE site was always a pleasure.

I would like to thank Mikael Silverberg for taking time to explain the ECU simulator, and for helping me with all my questions.

Thanks to Kurt Lundbäck and Mattias Gålnander for giving their time to introduce the world of component based software engineering and for their valuable input and ideas during my thesis.

I would like to thank Avenir Kobetski and Ze Ni for all their help and advice in relation to Java Virtual Machines. Their input was very helpful and enabled me to quickly get a better understanding of Java virtual machines.

Thanks to Jakob Axelsson for agreeing to be my examiner, answering any questions I had, and for all his support and guidance throughout the project.

I would like to thank Mälardalen university for giving me the opportunity to study at a wonderful university and for providing me with endless opportunities to improve myself as a person, and as an engineer.

And finally, a huge thank you to my supervisor Sara Dersten. She provided unending help, support and time throughout the project and I cannot thank her enough.

Gerard Duff

Västerås, Sweden, June 2014

List of Abbreviations

AUTOSAR	**AUT**omotive **O**pen **S**ystem **AR**chitecture
FRESTA	**F**ederated **R**esilient **E**mbedded **S**ystems **T**echnology for **A**UTOSAR
Volvo CE	**V**olvo **C**onstruction **E**quipment
SICS	**S**wedish **I**nstitute of **C**omputer **S**cience
ECU	**E**lectronic **C**ontrol **U**nits
CAN	**C**ontroller **A**rea **N**etwork
LIN	**L**ocal **I**nterconnect **N**etwork
HMI	**H**uman **M**achine **I**nterface
CPU	**C**entral **P**rocessing **U**nit
EEPROM	**E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory
ROM	**R**ead **O**nly **M**emory
CBSE	**C**omponent **B**ased **S**oftware **E**ngineering
VFB	**V**irtual **F**unctional **B**us
BSW	**B**asic **S**oft**W**are
RTE	**R**un**T**ime **E**nvironment
ASW	**A**pplication **S**oft**W**are
IDE	**I**ntegrated **D**evelopment **E**nvironment
RTOS	**R**eal **T**ime **O**perating **S**ystem
JVM	**J**ava **V**irtual **M**achine
KVM	**K**ilobyte **V**irtual **M**achine
JNI	**J**ava **N**ative **I**nterface
GC	**G**arbage **C**ollector
RTSJ	**R**eal **T**ime **S**pecification for **J**ava,
PS	**P**olling **S**erver
DS	**D**eferrable **S**erver
SS	**S**poradic **S**erver

OS	**O**perating **S**ystem
MVS	**M**icrosoft **V**isual **S**tudio
OEM	**O**riginal **E**quipment **M**anufacturer
OAA	**O**pen **A**utomotive **A**lliance
GENIVI	**GEN**eva **I**n **V**ehicle **I**nfotainment systems
App	**App**lication
GUI	**G**raphical **U**ser **I**nterface

Dedicated to my family and Kristina.

1

Introduction

The first chapter highlights the goals and motivation of the research conducted in this thesis, and provides an overview of the document structure.

Contents

1.1 Motivation for Thesis 2

1.1.1 Research Objective 2

1.2 Related Work 3

1.2.1 Third Party Application Development Platforms . . . 3

1.3 Conclusions and Observations 6

1.4 Thesis Overview 8

1.1 Motivation for Thesis

Customization is everywhere today. Electronic devices are no longer designed to be rigid units, where a change of functionality requires the purchase of a new device. The world is becoming smarter and devices more customizable to specific users. TVs are giving way to smart TVs, phones are making way for smart phones, and now automotive electronics are also starting to change.

Making vehicles customizable by downloading applications from trusted external servers, instead of through a return to the dealership, has major advantages for both the customer and seller. Customers can customize vehicles through the click of a button and car manufacturers and dealers are, through opening application stores, introducing a service industry to what was for the most part a “once-off purchase” goods industry.

1.1.1 Research Objective

Most of the work in the area of integrating third party applications into vehicles that will be discussed in section 1.2, will show that approaches appear to be aimed solely at infotainment systems. This approach is akin to treating a vehicle as a mobile phone by enhancing the infotainment system, but vehicles are more than mobile phones with greater functionality and capabilities, and it appears depreciative to limit third party applications to the infotainment system. A platform is needed to open selected signals in all the electronic systems in a vehicle to third party applications.

This raises a very interesting question. There is a vast ecosystem of application developers today, the majority of which use Java as the programming language of choice [1]. So if third party applications are to be easily integrated into a vehicles electronic systems, utilizing the large system of developers and application development processes already in place through mobile phone application development, Java may have to be introduced to vehicle systems. The question is “How will Java survive in resource constrained embedded systems?”.

Porting a full version of Java with all the capabilities of Java used in desktop computers or modern mobile phones is out of the question due to memory and processor constraints, so how does embedded Java fair? Is the subset of classes available to embedded Java enough to entice third party developers to design applications? How does including Java in a resource

constrained hard real time embedded system effect predictability and the overall security of the system?

This thesis report will endeavor to answer the questions posed, and investigate if it is feasible to have deterministic Java in resource constrained hard real time embedded systems.

1.2 Related Work

The automotive industry has recently started evaluating mechanisms to open automotive electronics to third party applications, with some platforms already being developed and prototype models displayed at car exhibitions [2,3]. There is a realization of the consumer need for levels of adaptability and customization in vehicles and efforts are being made to facilitate those desires.

A brief analysis of the current third party application automotive platforms will now be given in section 1.2.1. A state of the art report on third party application development platforms was compiled and the applicable findings are now discussed. This is not an exhaustive account, but rather a cross section of the options available in the marketplace as of June 2014.

1.2.1 Third Party Application Development Platforms

There are many platforms being developed today to integrate third party applications in automotive electronics with Original Equipment Manufacturers, OEMs, developing their own app stores, and accommodating existing app stores in systems [4]. This introduces significant challenges with regards to mixing infotainment systems with safety critical systems [5].

The challenge to establish systems where soft real time systems and hard real time systems cooperate without the detriment of either is significant. The underlying hard real time schedule must never be tampered with or compromised through the addition of third party applications, whilst ensuring a degree of reliability for the applications.

There also exists a responsibility to ensure the downloaded applications do not accumulate to become excessively distractive to vehicle operators. There must be a balance between functionality and user interface with regards to the number of apps downloaded to a vehicle.

1.2.1.1 General Motors OnStar

General motors established the OnStar platform in 1995 [6] and as of May 2005 it had more than 4 million customers [7]. It is an application that monitors vehicle diagnostics and utilizes a Global Positioning System, GPS, location to offer a variety of services to the user [8].

One area at which the application is aimed is theft prevention. OnStar has the ability to remotely deactivate vehicles. This means that vehicles involved in high speed police chases can be remotely shut down, potentially saving money on police resources and no longer endangering public lives [9]. This has successfully enabled police to halt high speed chases and recover stolen vehicles undamaged [10].

The OnStar system collects data using the on-board diagnostics system and a built-in GPS functionality [6]. It is installed in vehicles as original equipment during vehicle manufacturing and in consequence it is not possible for third party developers to install new applications on the system, but the extra functionality installed through OnStar at manufacturing time illustrates the potential advantages of installing applications, so called “apps”, in vehicles.

1.2.1.2 Ford and General Motors Infotainment Systems

At the Consumer Electronics Show, CES, in Las Vegas in 2014 Ford and General Motors announced the opening of vehicle infotainment systems to third party developers [2]. Applications can now be developed and submitted for distribution through the Ford application store, which when accepted, will be available to customers free of charge [11].

The three principle app development categories that Ford are focused on are: *news and information*, *music and entertainment*, and *navigation and location* [2]. The apps will have access to vehicles audio and display systems, along with some data from the engine such as mileage and speed. They will access the internet through a connected phone, or the cars own internet link [11].

Applications are expected to follow the Apple app model [12], with Ford stringent on acceptance criteria, and inclined to veto applications with potential to cause excessive distraction to the operator of the vehicle [13].

1.2.1.3 Cisco AutoGuard

At the 2013 North American International Auto Show [3], NAIAS, Cisco announced that it would be introducing a range of security products aimed at enhancing security in vehicle area networks [14].

This is not a platform for developing applications, but rather software to protect real-time over the air updates for vehicle electronic control units [15]. As such the system falls under the scope of this report and deserves mentioning as it brings to light the move developers and original equipment manufactures, OEMs, are making towards fully updateable and customizable vehicles.

1.2.1.4 Apple iOS in the Car

Apple announced plans to integrate iPhones and iPads with vehicles during the world developers conference in June 2013 [16]. When a customer enters the vehicle with an iPad/iPhone, the iPad/iPhone synchronizes with the infotainment display and customizes the display with a unique set of apps customized for each specific user [16]. While the extent of the access that the iOS has is still unsure, potential areas for development include maps, music, and a general extension of any current Apple device capability [17].

Apple's growing interest to integrate iOS with vehicles demonstrates the appetite among developers to develop apps for the automotive electronics world. Indeed a survey by Appcelerator in December 2013 showed that two thirds of developers listed iOS in the car as a platform that should be prioritized in 2014 [18], even though Apple have yet to even confirm if "third party" apps will be supported by iOS in the car.

1.2.1.5 Google Android Automotive Alliance

The Open Automotive Alliance [19], OAA, is an alliance of car manufacturers and the visual computing firm NVIDIA. They have come together to develop a platform that can bring the Google android experience to the dashboard of cars. The alliance are working toward the integration of android devices in vehicles to enable the vehicle itself as a connected android device.

The OAA are working closely with several road safety organizations like the National Highway Traffic Safety Administration, NHTSA, to ensure safety standards are adhered to and that drivers will not get overly distracted

by disruptive apps. The first car equipped with the platform is due for roll-out in the summer of 2014 bringing the 700,000 existing android apps to the dashboard [20].

1.2.1.6 GENIVI

GENeva In Vehicle Infotainment systems, GENIVI, is a non profit alliance dedicated to open source development of vehicle infotainment systems and applications [21]. GENIVI has over 160 members, and is working towards the creation of an open source Linux based infotainment system.

The problem with infotainment systems available today is the difficulty to introduce new software not just to different car models, but entire generations of cars. This is the driving force behind the GENIVI alliance [22]. If successful the platform would mean that developers could develop new software that could run on all vehicles with the GENIVI platforms. This would be a big step toward opening automotive systems to third party apps.

1.2.1.7 OpenXC Platform

Ford has been working on developing the OpenXC Platform, which strives to present the car in a manner similar to that in which developers view smart phones [23]. Drivers are installed on a small piece of hardware that can read vehicle sensors and control units, and convert them to a format that can be read by compatible android apps [24]. The OpenXC platform has been distributed to firms and universities and has already lead to the creation of some “vehicle-aware” applications [24].

One such application is used to notify contacts if a person will be late for a meeting. The application detects the vehicles location, calculates if the vehicle will reach it’s destination on time, and if it calculates that the vehicle will arrive late the application emails or texts the people the driver is driving to meet to let them know there will be a delay.

1.3 Conclusions and Observations

Considering the platforms that are currently available or in development leads to a predominant observation. There appears to be two distinct directions

and approaches for delivering applications to vehicles; integrating a mobile device with a vehicle to use it as a vessel to connect to apps through the mobile device, or an OEM run app store where apps are downloaded directly to the vehicle.

Both approaches have advantages and disadvantages. The advantages of the first method are:

- A large mobile developer ecosystem that is already in place.
- Application updates that are easily managed.
- Phone and car applications that are managed in the same place.

The big disadvantage of the first method is that the mobile device is only interacting with the infotainment system, and whatever signals the infotainment system has access to. This limits the potential of applications, and the solution of routing signals through the infotainment system introduces the headache of extra signals running through the controller area network bus, described in section 2.1.2.1.

A survey carried out by the center of automotive research during January 2014 [25] found that the average car now contains an average of 60 microprocessors and more than 10 million lines of software code. Confining applications signal access to just a single vehicle infotainment system seems excessively restrictive.

The advantages of the second method include:

- The potential to open all vehicle systems to applications.
- Greater customization possibilities.
- Greater diagnostic surveillance opportunities.

The second method has the potential to excel where the first method falls short. Turning a vehicle into a smart vehicle by opening up selected signals from all the vehicles systems offers virtually unlimited possibilities for developers. However this method comes intertwined with huge safety concerns.

This thesis work deals with the Java virtual machine approach that project FRESTA uses, as outlined in section 2.4, and it offers solutions to some of the safety concerns surrounding potential memory leaks in applications, by using garbage collection algorithms. The drawback to this method is that the garbage collector also presents some issues. The unpredictability of the garbage collector makes it tough to use in real time embedded systems so an approach to combat this uncertainty would be necessary.

The solution could possibly lie in a hybrid of both methods [26] through using embedded modems in vehicles for some functionality, with a mobile phone also being used for some applications. However along with combining

the advantages of both methods, the disadvantages of both would have to be managed, and amalgamating both methods would have to be well planned and organized.

1.4 Thesis Overview

The objectives of the work reported in this thesis are primarily to: 1) Study the Volvo CE platform and FRESTA mechanism; 2) Analyze what is necessary for the porting; 3) Port the FRESTA platform to the Volvo CE platform; and 4) Develop an application to test the porting. The structure of the thesis seeks to follow the objectives outlined while simultaneously investigate the feasibility of Java in resource constrained embedded systems.

Chapter 2 provides a brief background in the areas of automotive electronics, Java, and component based software engineering. It delivers an introduction on the technical background of the thesis work and describes some of the pertinent elements related to the investigation.

Chapter 3 is an analysis of what is necessary for the porting and the porting work. It starts by developing a porting plan, then details some of the porting steps taken, and ends with a discussion evaluating the porting process undertaken in this thesis and other related porting efforts.

Chapter 4 outlines the Java application development process. The reasons behind the application chosen, along with possible extensions to the application and developing applications for vehicles in general, are discussed.

Finally chapter 5 contains the results and conclusions observed, and endeavors to answer some of the questions regarding Java in resource constrained embedded systems.

2

Background

The second chapter introduces the software and technology used, and presents a background of the main technological components, in the thesis.

Contents

2.1 Automotive Electronic Systems 10

2.1.1 Electronic Control Units 11

2.1.2 Communication 11

2.2 Java 14

2.2.1 Java Virtual Machine 14

2.2.2 K Virtual Machine 15

2.2.3 Characteristics of Java Virtual Machines 15

2.3 Component Based Software Engineering 19

2.3.1 Automotive Open System Architecture 20

2.3.2 Volvo Software Platform 21

2.4 FRESTA 22

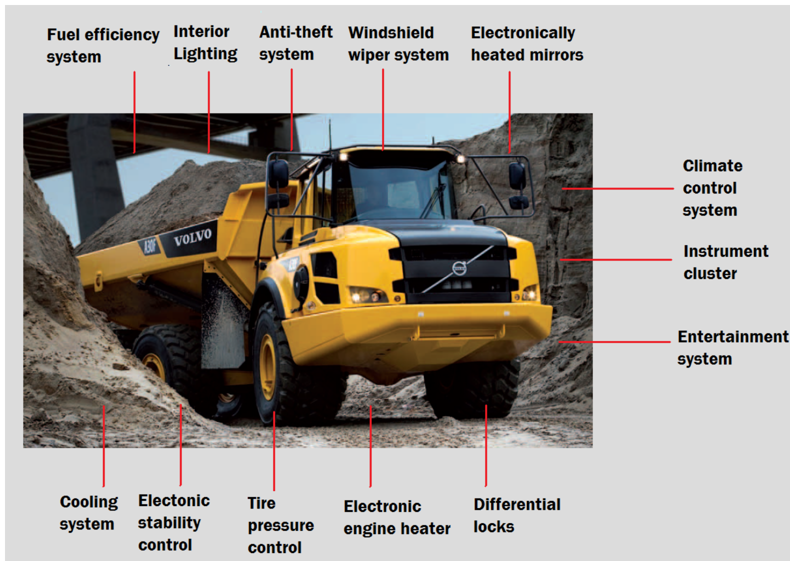

Figure 2.1: Automotive electronic systems in a typical vehicle

2.1 Automotive Electronic Systems

Auto electronics are becoming increasingly important in vehicles. Since the introduction of electronics for emission control in engines, the evolution of electronics in automobiles has advanced rapidly [27]. Auto electronics are used to control almost every system in a vehicle, with Figure 2.1 depicting a small cross section of the electronic systems available in vehicles.

Automotive electronics are now increasingly utilized to implement safety systems under the mantra "A smarter car is a safer car" [28], whilst simultaneously customers expect more infotainment applications integrated in vehicles. This has lead to an industry that has a growing demand for integrating applications with diverging real-time and criticality requirements on the same microcontroller [29].

Mixed criticality systems introduces challenges in the development of software components for vehicles. What happens when soft real time systems such as infotainment systems try to integrate with hard real time safety critical systems such as brakes or airbag deployment systems? Well that is one of the main questions this thesis sets out to answer.

Are mixed criticality systems viable in vehicles? How can system failures or integration bugs be prevented when applicaions are introduced to vehicle systems? Clearly integrating applications with safety critical platforms require

the art of limiting the applications influence while interacting and sharing resources [29].

2.1.1 Electronic Control Units

Electronic Control Units, ECUs, are microcontroller based systems in vehicles that control various functions. As vehicles come to incorporate an increasing number of electronic systems, the number of ECUs in vehicles rise. It is not uncommon for a single vehicle to include 40-70 ECUs [30] to regulate the various systems independently and cooperatively through communication channels. A typical block diagram can be seen in Figure 2.2 [31] .

Having an increasing number of microcontrollers in a vehicle boasts many advantages - a more modular design, an improved distribution of systems, etc. - but there are also disadvantages, mainly increasing system complexity and complicating safety designs. In 2013 a lawsuit was brought against Toyota after a sudden acceleration fault was found in several Toyota vehicles [32]. During the trial, embedded systems experts who reviewed Toyota's electronic throttle source code testified that Toyota's source code was defective, and that it contained bugs - including bugs that could cause unintended acceleration [33]. Clearly if there is to be a move towards "plug in applications" that can plug into an ECU, as is the background of this thesis, stringent safety strategies must be developed and rigidly enforced.

2.1.2 Communication

As eluded to in section 2.1.1, automotive systems are distributed across multiple ECUs. There are several communication methods implemented by vehicle manufacturers to communicate between various ECUs. The most widespread and commonly used are Controller Area Networks, CAN, and Local Interconnect Network, LIN.

CAN is generally the backbone of communication in vehicles, used to enable communication between the different ECUs of the vehicle. LIN is commonly used for enabling communication between peripherals, and between peripheral input/output devices and an ECU.

If applications are to be downloaded across multiple ECUs, and utilize signals from various different ECUs, a method of communicating through CAN and LIN may be needed. Both CAN and LIN will be described in sections 2.1.2.1 and 2.1.2.2 respectively.

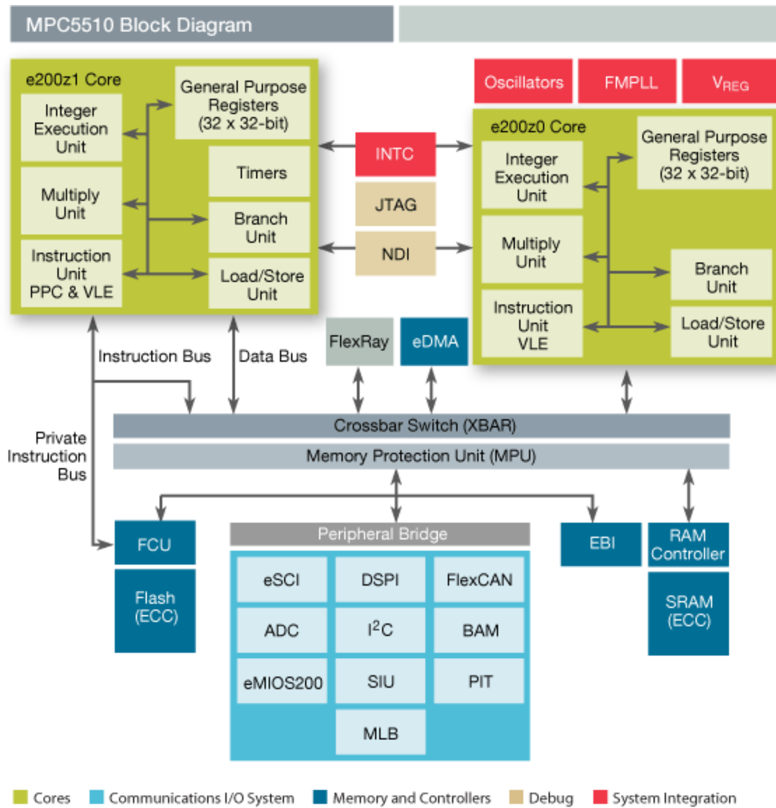

Figure 2.2: Typical block diagram of an ECU

2.1.2.1 Controller Area Network

Controller Area Networks, CAN, was first introduced in February 1986 by Robert Bosch GmbH [34]. It has grown to become the most commonly used communication bus system in vehicles in Europe [35]. It is widely used as it boasts predictable timing behavior on communication transmissions [36].

CAN is a broadcast bus which uses deterministic collision resolution to control access to the bus [36]. If messages with multiple priorities are sent simultaneously, the message with the highest priority will always succeed; while a message with a lower priority will fail, know that it has failed, and try to send again. This is due to the CAN arbitration technique.

The CAN message ID field is used to decide which message gains access to the bus when collisions occur. If a message sends a '0' then all stations watching the bus will observe a '0'. If multiple messages are sent simultaneously, any message containing '0' gets sent. If a message that has a '1' tries to send

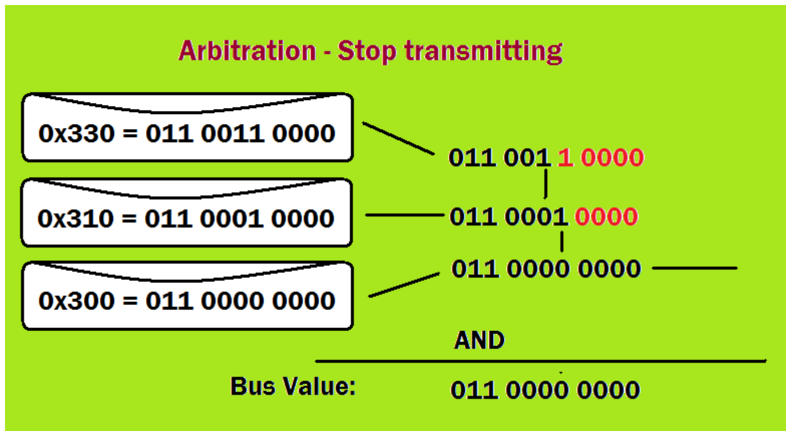

Figure 2.3: CAN arbitration

against a message that has '0', the later message will be sent. In effect the only time a message with a '1' will win an arbitration is if all messages being sent send a '1' at the same time. In effect the CAN bus acts like a large AND-gate. Figure 2.3 depicts an arbitration race [37].

The CAN standards advanced collision protocols and transmission predictability make it a desirable communication technique for system to system communication in vehicles.

2.1.2.2 Local Interconnect Network

Local Interconnect Network, LIN, is a communication protocol widely used in vehicles. It was developed by Audi, BMW, Daimler-Chrysler, Motorola, Volcano, Volvo, and Volkswagen [35] to be an inexpensive network communication method [38]. It was also designed with low power systems in mind and as such includes mechanisms to send particular nodes in and out of "sleep" mode to conserve energy [39].

LIN utilizes a solitary wire to implement the LIN communication bus, as opposed to CANs twisted wire approach, and uses the vehicle chassis as the return path for the current [40]. It is based on a master slave network, with a capacity of up to 16 slaves, where master nodes are used to control the traffic on the network and eliminate collisions [36].

LIN is smaller and generally slower than CAN and as such is only used to connect peripherals to ECUs or to integrate actuators and sensors [40].

2.2 Java

The world of third party applications today is built on Java. Java standard edition 7 has close to 4,000 standard class libraries, which represents a wealth of programming experience that could potentially be introduced to embedded systems [41]. The old Java motto of "Write once, run anywhere" has revolutionized software development and enabled developers to create highly portable software with minimum effort, but can this motto be enforced with resource constrained embedded systems?

Moves toward turning Java into a viable option for embedded systems development has already begun with The Real Time Specification for Java [42], RTSJ, that is a collection of specifications for real time applications developed for Java. Amongst other things, it specifies three types of tasks [43]:

1. Critical tasks that cannot be preempted.
2. High priority tasks that cannot tolerate unbounded preemption
3. Low priority tasks that can tolerate delays.

A more detailed insight into the specification is given in the book "Real Time Specification for Java" [44].

2.2.1 Java Virtual Machine

Java is at heart a virtual machine environment. This has advantages when developing software for embedded systems as it is possible to start developing and testing large parts of applications on desktops, without the need for hardware [45]. Java has the advantage of abstracting the software language from the hardware, increasing application portability. This has made embedded system developers begin to warm to Java over recent years [46]. Java applications run on top of a runtime environment which in the case of this thesis will be the K virtual machine.

2.2.2 K Virtual Machine

There are currently many Java virtual machines available aimed at small embedded devices. The trade-off between features and memory is a reoccurring theme throughout the offerings, and most commonly features can be disabled to reduce the memory footprint. After an analysis of the current Java virtual machines was made it was decided to use the K virtual machine, KVM, or Kilobyte virtual machine as it is also sometimes referred to.

The KVM is a run time environment aimed at devices with limited memory and processing power, that applications designed using a subset of Java classes can run on [47]. Oracles KVM was the JVM used for embedded devices before a newer and larger JVM called Hotspot [48] was introduced and replaced it. As KVM was aimed at older devices the memory footprint is quite low, approximately 30 Kilo bytes, and it has been successfully ported to many embedded devices.

2.2.3 Characteristics of Java Virtual Machines

The main characteristics of JVMs will now be discussed. The typical principal components of a JVM, and how the K Virtual Machine used in this thesis adapts or approaches these components, will be briefly summarized.

2.2.3.1 Development Flow

The normal development sequence when working with Java is to compile the source code and then the virtual machine verifies and interprets the Java byte code, however as the standard Java verifier is larger than KVM itself a different approach is undertaken. The verification step is split into two phases. The code is compiled and pre-verified on the host, and then the Java byte code is verified and interpreted in the KVM on the target device. The Java virtual machine interfaces with native OS functions through a Java native interface.

2.2.3.2 Java Native Interface

The Java Native Interface, JNI, is used to control the placement of objects in physical memory and to access input/output devices on the embedded system [49]. It is the interface that connects the JVM to the peripherals of the embedded device and allows access to outside signals from native applications. It is also a method to call native code from Java applications.

KVM does not utilize a JNI in an attempt to conserve memory. Instead it has its own K native interface which generates a native function table automatically at runtime. This native function table is contained in the "nativeFunctionTableWin.c" file. This table is used to interface with underlying OS functions and, along with the Garbage Collector, is one of the two most important aspects of any JVM.

2.2.3.3 Garbage Collector

Any object created by a Java application is stored on the JVM heap and there is no way to free these objects from memory directly from code [50]. That duty falls to the Garbage Collector, GC, which is a task that runs and frees all unused objects from the heap.

The unpredictability of memory management due to not knowing when the GC will operate, and how much time it will take to complete, is one of the major issues when associating Java with real time systems [43]. If Java is to be successfully included in real time embedded systems then an approach to make the GC more predictable is needed. Even though real-time garbage collection is one feature that is not mandatory in the Real Time Specification for Java, RTSJ, [51] there are many methods and approaches to GC in place in todays embedded Java world, some of which will now be discussed.

Stop the World

One method of garbage collection is a "Stop the world" approach. This stops every other Java task while the garbage collector works. If the garbage collector is then set to the highest priority, the worst case estimation time can be then estimated using the heap size [52].

Incremental Garbage Collection

Incremental garbage collection aims to be a less disruptive influence on tasks in the JVM. Its goal is to interweave garbage collections

with tasks to prevent unbounded delays. This can introduce memory fragmentation.

Although some studies suggest memory fragmentation may not be such a big problem [53], these only take medium term programs into consideration, and do not sufficiently investigate the long term effects of memory allocation. There are hybrid approaches to combat this [54], and also some algorithms have been developed to specifically combat memory fragmentation [55].

Parallel Real-Time Garbage Collection

Parallel real-time garbage collection can be implemented on multi-core processor systems [56], but as this thesis' focus is resource constrained systems, parallel garbage collection falls outside the scope of this paper.

Region Based Memory Management

Region based memory management [43] is an approach to combat unbounded blocking pauses to threads caused by GC. It works by creating a stack of regions, where objects are saved in regions based on their thread or method.

The advantage of this approach is that if it is combined with a traditional automatic GC, it can work quite well. Soft real time tasks use heap memory as normal, which is cleaned by the automatic GC, but hard real time systems allocate objects to a scoped memory region, which is released upon completion of the hard real time task. This means the automatic GC can be turned off when hard real time tasks are executing and the system becomes more predictable.

Ravenscar-Java Profile

Another approach is the Ravenscar-Java profile [57] which is a subset of the RTSJ that only allows the allocation of objects during a designated initialization phase. The disadvantage with this approach is that it is all done statically.

K Virtual Machine GC

The KVM approach is as expected, small and simple. It is based on a non-copying collector to save on memory and utilizes a mark and sweep algorithm [58]. As the GC is non-incremental it performs best with small heaps (30-512 Kilo bytes) as larger heaps would result in long GC periods.

2.2.3.4 Romization

Romization is a method where a memory image of the Java runtime environment is dumped from a deployment host and then copied onto the target device [59]. This has numerous advantages.

Firstly the startup time of the application will be reduced as no pre-compilation will be needed so the device is ready to use immediately. Secondly, Java class loading is computationally heavy and it is desirable to remove this process from resource constrained embedded systems. The Romization process can be utilized with the KVM, and as a result the “ROMjavaWin.c” is generated.

2.2.3.5 Aperiodic Events in Real-Time Java Systems

Java systems cannot be limited to purely periodic tasks and must have capabilities to handle aperiodic tasks. Usually systems have a mix of hard periodic tasks with soft aperiodic tasks. There must be an approach that preserves the deadlines of the hard periodic tasks while minimizing the soft aperiodic task response times and maximizing processor utilization.

One approach is to give all aperiodic tasks a lower priority than the hard periodic tasks, but while this is easy to implement, it does not address minimizing aperiodic response times. The more common approach is to implement a periodic task server [60]. Some server approaches include:

Polling Server

The Polling Server, PS, has identical features to a normal periodic task. It has access to a queue of aperiodic tasks and operates under the first in first out, FIFO, assumption. The main disadvantage of this approach is if a task is released just after the activation of the server task, it has to wait until the next activation period before being run. KVM uses the polling approach for event handling.

Deferrable Server

The Deferrable Server, DS, addresses the weakness of the PS. It keeps its capacity to execute aperiodic tasks even if the queue is empty. It can preempt lower priority tasks to execute but this violates the assumption that all aperiodic tasks are soft and shouldn’t preempt the hard real-time tasks of the system

Sporadic Server

The Sporadic Server [61], SS, has an execution time it can consume

each period. If the aperiodic task queue is empty it delays activation until a task arrives. The capacity is restored based on the replenishment time and the replenishment amount. This server is theoretically the best [60] but also more complex to implement.

Slack Stealing Approaches

These approaches are based around calculating the amount of time a periodic task can be suspended without missing a deadline. It then uses this "slack" time to execute aperiodic tasks.

2.2.3.6 Java Green Tasks

Java green tasks are tasks created by a JVM without invoking any underlying Operating System, OS, capabilities. The JVM maps multiple Java threads to a single system thread [62].

Each Java thread is assigned a context with a program counter which informs which instruction is to run next, and a stack for thread variables and other bookkeeping tasks [63]. The Java threads share a common heap for dynamic objects [50].

Green threads, or tasks as they are also known, provide a useful function in that it becomes possible to dynamically create threads in a system where that functionality had not previously existed. Green threads provide a mechanism that could provide a solution to one of the obstacles in developing an application platform, as discussed in section 2.4.

2.3 Component Based Software Engineering

As automotive electronics increase in both size and complexity, it becomes advantageous that Component Based Software Engineering, CBSE, approaches are practiced in the vehicle industry. Breaking software up into reusable components helps reduce development time and system complexity.

CBSE has been used to successfully construct complex desktop applications, and as embedded systems get more complex it stands to reason that the approach should be investigated for the embedded world. CBSE is based on software components that are self contained tasks or functions, or a combination of other software components, that are used to construct a program by connecting together the components like pieces of a jigsaw puzzle. Two CBSE approaches are AUTOSAR for which the FRESTA approach outlined in section

2.4 is aimed, and the Volvo CE software platform which is the foundation of this thesis work.

One of the main differences between AUTOSAR and the Volvo CE software platform is the communication protocols between the software components. In AUTOSAR communication between components is handled by a Virtual Functional Bus, VFB, where components send data to a virtual communication bus and it manages the software components connections and communication. Components do not need to know where other components are located when they are connected together.

The Volvo CE software platform approach differs as component ports must be connected together. An output port of a component must be connected to the input port of the desired component. This is implemented through the Integrated Development Environment, IDE, and does not have to be coded. Both approaches are discussed in sections 2.3.1 and 2.3.2.

2.3.1 Automotive Open System Architecture

AUTomotive Open System ARchitecture, AUTOSAR, is a worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry [64]. AUTOSAR is common set of language and methodology standards assembled to allow a global industry standard in all software development for vehicles.

The standard is a framework where the application is composed from reusable components that can be embedded in a specific vehicle using a configuration scheme [65]. It decouples the basic software that needs to exist in all ECUs and can be standardized from the application software [65]. It was, according to the AUTOSAR consortium, already in use in 25 million ECUs in 2011. This figure is expected to rise to 300 million in 2016 [66, 67].

AUTOSAR is structured around a layered software architecture that contains three levels: the Basic SoftWare (BSW), a middleware called the RunTime Environment (RTE), and the Application SoftWare (ASW) [66, 67]. The BSW layer contains the ECU operating system, device drivers, and other system services. The RTE is where communication management takes place. Communication between different software components and different software layers is carried out here. The ASW layer contains all the software components to carry out various system functionality. A more detailed explanation is given in Axelsson2013 [65].

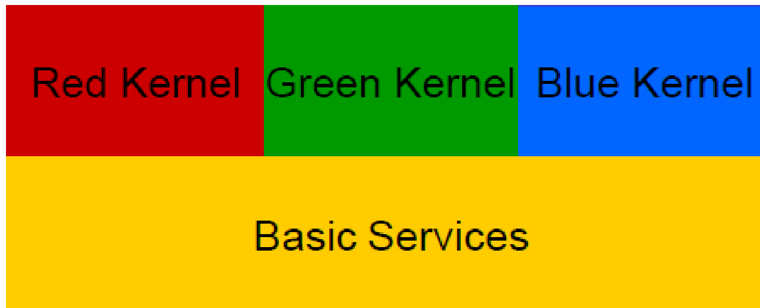

Figure 2.4: Volvo software platform kernels

2.3.2 Volvo Software Platform

The Volvo CE software platform is a component based software architecture that runs on a real time operating system. It is similar in approach to AUTOSAR as it seeks to decouple hardware and software as much as possible. It strives to achieve this through CBSE which also introduces advantageous properties like portability, and re-usability. The platform was designed to be simple to use, yet have enough features to adequately handle a complex embedded system. The overall goal is to make embedded systems as predictable and resource friendly as possible, and as such make the operating system desirable for complex embedded systems applications.

The Volvo CE software platform is based on three kernels as shown in Figure 2.4. The red kernel is used to manage red threads which are off-line assigned hard real time tasks. The green kernel handles interrupts in the system. The blue kernel handles the spare execution time left over from the red threads. Blue threads are also assigned off-line and utilize the leftover processor capacity after the red tasks have been scheduled.

There are two main phases when developing software for the platform. The first phase is developing in the Volvo CE Integrated Development Environment, IDE, where all tasks, queues, schedules and software components are designed. The tasks are assigned priorities, periods and deadlines and connected together as desired.

The platforms IDE consists of four main parts :

1. Designer: This is the tool for graphically designing the components of the system.
2. Compiler: The compiler for the system.

3. Builder: Builds the code.
4. Coder: Generates the specifications of the Real Time Operating System, RTOS, as dictated in the designer.

The Volvo CE software platform uses modes to define the states a system goes through. For example an ECU could have a startup mode, followed by a running mode, and a termination mode. Each mode can be treated as a self contained application and depicts the operating conditions of the system for each state.

The second phase of the platform development process is to add functionality to the tasks. After the system is designed in the Volvo CE software component designer the relevant files are imported to a code development IDE, like Microsoft Visual Studio [68] or Eclipse [69], and task functionality is developed by adding code to the task shells auto-generated by the designer. The real time operating system is then imported and everything is compiled.

2.4 FRESTA

Federated Resilient Embedded Systems Technology for AUTOSAR [70], FRESTA, is a collaborative project between Volvo Group [71], Volvo Car [72], and the Swedish Institute of Computer Science [73], SICS, to open the computer systems of cars to the market of applications or “apps”. The goal of FRESTA is not to develop apps, but to develop a platform to turn automotive systems from closed developing environments into “platforms for innovations” [70]. The challenge presented is to give external suppliers the ability to develop apps that can easily be integrated into an automotive system, without the loss of the vehicle’s critical systems security or robustness.

The FRESTA platform works by putting a Java Virtual Machine, JVM, in an AUTOSAR software component. The JVM runs on a plug in run time environment [65], and the JVM then handles the plug in applications [74] through the use of Java green threads. The basic FRESTA component diagram is represented in Figure 2.5. The Volvo CE software platform is similar to AUTOSAR with the same ideological structure and design principles, meaning that the FRESTA platform should be easily ported.

Java green threads, or threads, are user level threads implemented without the need of underlying operating system capabilities [75]. This is an ideal scenario as it allows applications downloaded from a secure server by the JVM to be dynamically assigned to tasks without changing the off-line schedule. The JVM is assigned to a large Volvo CE software platform blue

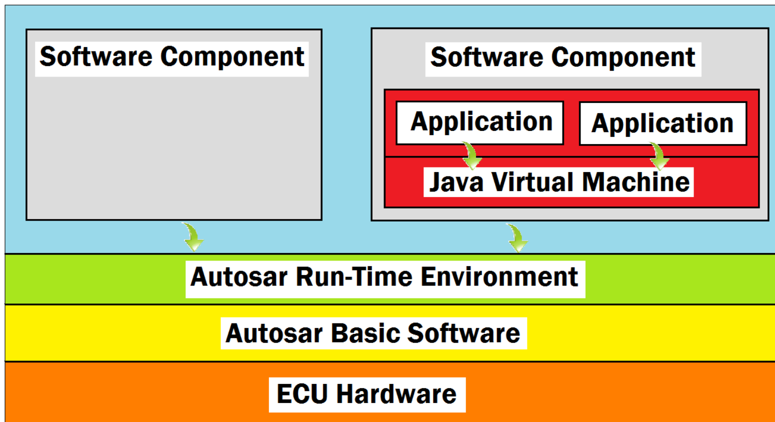

Figure 2.5: FRESTA component diagram

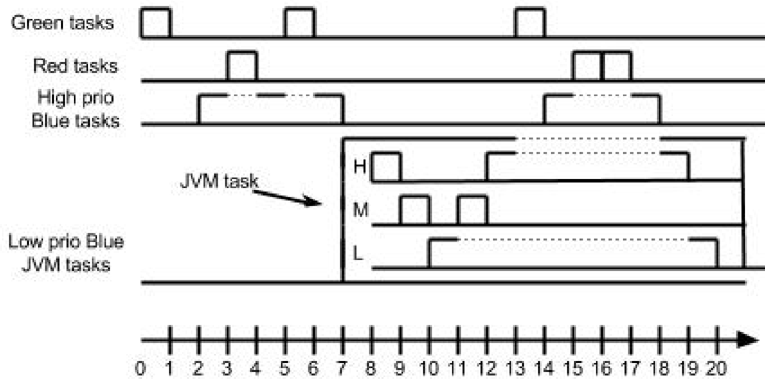

Figure 2.6: Volvo platform timing example

task before runtime and then the JVM can create Java green tasks within this Volvo CE software platform blue task as required. The underlying operating system still views the JVM as a single blue task within the Volvo CE software platform, even though the task could consist of several Java green tasks as represented in Figure 2.6.

Java green threads have three different priority levels: high, medium and low [76]. All the active threads are kept in a simple linked list and threads execute on the basis of highest priority first [58]. This is not the most optimum scheduling algorithm for threads and limits the scope of scheduling many tasks or applications with different priority levels, however its simplicity contributes to a lower overall memory footprint. Green threads are discussed in more detail in section 2.2.3.6.

The FRESTA platform is in an advanced stage of development and has

already been ported to a Raspberry Pi [77] for testing [78]. It has also been shown that Java can be successfully integrated into AUTOSAR [79].

The implementation goal of this thesis was to port the FRESTA platform to a Volvo CE software component, and develop an app to test the porting. The thesis goal was to take a JVM and put it in a software component. The first goal was to choose a JVM with a small enough memory overhead to fit in an ECU. The JVM should ideally have the ability to create and manage green threads, and be open-source. These criteria lead to a straight choice between two JVMs; the Kilobyte Virtual Machine [80], KVM, originally developed by Sun and then Oracle [81], or Squawk [82] which is an open source project hosted on java.net [83].

The K virtual machine was chosen as there exists a greater catalog of support documentation which is vital when it comes to completing a project in a short to medium timescale. The disadvantages with using the KVM was that it uses an older version of Java so it is not future proof, but as a technical exercise to investigate the viability of running a JVM in a resource limited embedded system it fits the profile perfectly.

The second goal of the thesis was to design an application to test the porting. A simple application to flash a light on the dashboard would be the starting goal for an application with scope for enhancing the application to a more useful real world application.

3

Porting a JVM to a Volvo CE Software Component

This chapter describes the porting process to port a Java Virtual Machine to a Volvo CE platform software component.

Contents

3.1 Porting Strategy 26

3.1.1 Volvo CE Software Platorm Dilemma 26

3.1.2 Initial Porting Plan 27

3.1.3 Final Porting Plan 28

3.2 Porting Implementation 29

3.2.1 Development Environment 29

3.2.2 Starting the Virtual Machine 30

3.2.3 Setting up KVM defaults 31

3.3 Porting Observations 32

3.3.1 Challenges 32

3.3.2 Related Porting Work 33

3.3.3 Evaluation 33

3.1 Porting Strategy

Porting is a term used to describe the modification of a software platform to integrate with an environment in which it may not have been originally designed to operate. In the case of this thesis the Federated Resilient Embedded Systems Technology for AUTOSAR, FRESTA, plug and play platform for software applications, more specifically the Java K virtual machine (KVM) component, was ported to a Volvo CE software component.

3.1.1 Volvo CE Software Platorm Dilemma

As discussed in section 2.3.2, the Volvo CE software platform is an off-line scheduled system. The big problem with introducing applications into an off-line scheduled system is that the operating system usually does not have the capability to dynamically create tasks. This means either creating tasks pre-runtime to assign to applications as they are downloaded, or including a mechanism to create tasks during runtime.

The first approach would mean a limit on the number of applications that an Electronic Control Unit, ECU, could run. It would introduce a memory limit on designing applications as applications could not exceed the assigned task size designated to application tasks. Another disadvantage would be potential memory waste. If an application was smaller than the memory assigned to a task for an application, then there would be unused leftover memory, and in embedded systems all memory is crucial.

The second option is to include a mechanism to dynamically create tasks when required for applications and is the main reason why a Java virtual machine was chosen to port to a Volvo CE software component. The Java green task mechanism described in section 2.2.3.6 that is used in Java Virtual Machines, JVMs, is ideally suited for dynamically creating tasks as tasks are created without invoking the underlying Operating System, OS. This would eliminate the disadvantages of the aforementioned first approach.

The disadvantage with the second approach is that the applications could becomes slower, or lag if too many applications are downloaded to the JVM. Given the advantages and disadvantages of both methods it was decided to progress with the second approach.

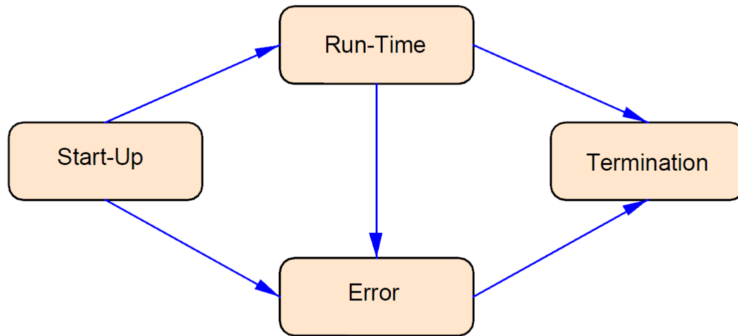

Figure 3.1: Planned Java virtual machine modes

3.1.2 Initial Porting Plan

The initial porting plan was to split the KVM component into four modes as depicted in Figure 3.1.

1. Start-Up: The initial startup phase of the KVM. The KVM is initialized.
2. Runtime: The runtime mode of the KVM. The main operating mode of the KVM.
3. Termination: The termination mode of the KVM. How the KVM shuts down.
4. Error state: The error state. What happens if an error occurs during startup, runtime or initialization mode.

Within the KVM component there are two main software components as depicted in Figure 3.2. The first is to handle the threads, the second to handle the download of Java applications. As the target ECU in this thesis does not have the ability to download applications, that component will be ignored for now, but should be included in future work.

3.1.2.1 Thread Management Component

The Volvo CE software platform firstly schedules all the red tasks according to their periods to ensure each task meets its deadline. The leftover utilization time is given to blue tasks. Blue tasks can have a priority from 1-15, 15 being the highest, with the OS using priority 1 and priority 2 for idle tasks.

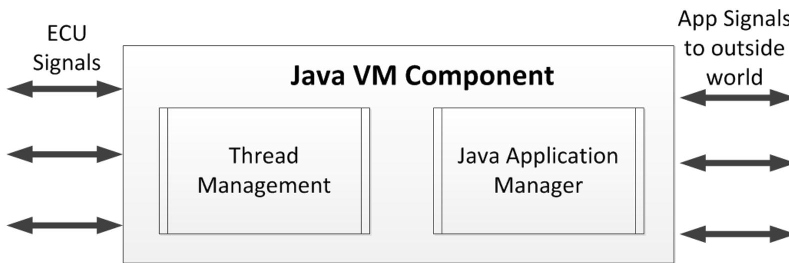

Figure 3.2: Java main component overview

Ideally the blue task containing the JVM should have a task priority of 3 or 4 so that it does not interfere with other higher priority blue tasks scheduled in the ECU, but will have preference over the idle tasks of priority 1 or 2, ensuring that it is not starved of processing time.

The main JVM component contains a thread management component to manage the green Java threads. The thread management component as depicted in Figure 3.3 is based on the Real Time Specification for Java, RTSJ, assumption that there are three types of tasks in a real time system [84]. The three types of tasks assumed are: hard real time tasks that should not be preempted, soft real time tasks that can be preempted, but not for an unbounded amount of time, and non real time tasks that have no hard or soft timing requirement.

This principle compliments the Java green task mechanism used by the KVM JVM as Java green threads can have three priority levels. Application threads will be assigned to the three priority Java green threads. The RTSJ also assumes a preemptive garbage collector, GC, so the standard cannot be fully implemented as part of this thesis, but having the component structure in place for upgrading the JVM to a professional light weight JVM in the future would be preferable.

3.1.3 Final Porting Plan

Due to project timing constraints it was decided to firstly implement just a single mode, single component JVM, and time permitting extend the design to the complete initial plan.

This component would be initiated be a button press which would initialize the KVM and run a preloaded application. The plan was to try and make things as simple as possible to complete the project successfully and on

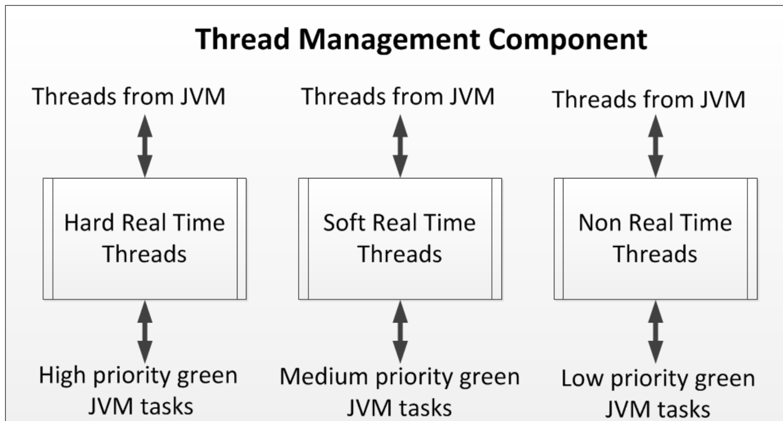

Figure 3.3: Thread management component overview

time.

3.2 Porting Implementation

This section will discuss the process undertaken to port the KVM to a Volvo CE platform software component. The major decisions taken and implementation approach will be discussed in the following sections.

3.2.1 Development Environment

The port of the KVM immediately presented several obstacles. The first decision was to chose an Integrated Development Environment, IDE, as the Volvo CE software platform, KVM, and ECU simulator code are all developed using different development environments.

Volvo CE software platform components are developed in a specialized IDE, and then code to add functionality to the components is generally developed through Eclipse. However as the ECU simulator is based on code developed in Microsoft Visual Studio, MVS, MVS had to be the IDE of choice. The Volvo CE software platform components and functionality code for the human machine interface, HMI, ECU and the KVM code was then added to a MVS project.

3.2.2 Starting the Virtual Machine

The default main file included in the KVM is configured for command line startup. KVM is called from the command line, with the options of:

1. Adding a class path of an application to run.
2. Including the debugger code.
3. Defining the heap size.

To start the KVM without using the command line, the code simply needs to call the "StartJVM()" function with the appropriate arguments.

3.2.2.1 Defining the Heap Size

The heap size must be larger than 16 kilobytes, and less than 64 megabytes - the maximum heap size allowed by the Garbage Collector, GC. In reality the GC is optimized for small heaps so if the heap size exceeds a few megabytes the GC pauses will become longer. If no heap size is defined it runs at a default 256 Kilobytes.

An estimate of the memory available to the KVM was needed to decide what size to set the heap for a successful porting . Volvo CE has a tool that analyzes the amount of memory an ECU application takes up on an ECU and displays the quantity of remaining memory on the ECU. This tool was used to estimate the free memory on the ECU, and the heap size was set accordingly.

3.2.2.2 Debugging mode

One of the goals of the thesis was to keep the memory footprint as small as possible and for that reason the debugging mode was set to 'OFF' as default.

3.2.2.3 Loading Applications in the KVM

The KVM comes with two main methods of loading applications. The first is passing the class path of the application through the command line. The other option is through using an optional Java Application Manager, JAM, feature which is also equipped to handle the loading of multiple applications downloaded from the internet or secured server.

Passing arguments through the command line was not a viable implementation option as the KVM will be initiated through a button press. JAM is a feature must be enabled and incorporated in the long term, as the project progresses to include the capability of downloading applications from the a secured server. The ECU in this thesis however, did not have the ability to connect to a secured server so the setting up and usage of JAM falls outside the scope of this particular thesis.

This meant that an alternative method was needed. The approach undertaken was to pre-load the applications as class files at compilation time and hard code the KVM to run the application of choice as default.

3.2.3 Setting up KVM defaults

Every time a KVM is ported it must include several port specific files [85]. These files are used to define the KVM functionality and characteristics desired. These files will now be discussed in more detail.

machine_md.h

This file was used to declare the system processor architecture and definitions. Definitions and Macros included in this file override any included in the “main.h” file.

runtime.h/runtime_md.c

These files are used for port specific functions. Functions to warn the user in case of errors, functions to initialize the KVM or modify the file or class operations can be found here.

3.3 Porting Observations

A number of observations were drawn from the porting work of the thesis. First of all, the coding required in this effort was minimal. Setting some defaults and customization values, and pointing the software component to the KVM is all that was needed. Getting the KVM operational, and adapting a developing environment to integrate the JVM with the Volvo CE software platform required the vast majority of effort.

3.3.1 Challenges

As stated previously the greatest challenges involved were to get the KVM operational, and to find a developing environment suitable to both platforms. Firstly getting the KVM up and running proved trickier than anticipated. The KVM is an old technology, with the last update almost half a decade ago. Building the KVM required “Makefile” functions available in the Windows XP platform but which are no longer supported by Windows Vista and newer versions of Windows. A version of Cygwin [86] was installed with the required “Makefile” functionality to overcome this and the KVM was built. The linux version of KVM was never fully investigated as the Volvo CE software platform used for the thesis was based in Windows.

KVM came with a Microsoft VS, MVS, project file that could be loaded to modify the code. This was a Microsoft VS 6 project file, an old version of MVS, but luckily MVS 12 converts older MVS projects to the newer project format with relative ease. Two of the files included in this project file, “nativeFunctionTableWin.c” described in section 2.2.3.2 and “ROMjavaWin.c” described in section 2.2.3.4, are not included with the project file and must be built through the KVM “Makefile” build process before the MVS project file becomes usable.

Working with the code of an Electronic Control Unit, ECU, was another challenge faced in this project. As discussed in section 2.1.1 ECUs are increasing in complexity and lines of code. Normally when beginning with a new software platform a simple “Hello, World” program is the starting point. With the Volvo CE software platform a simple “Hello, World” is not simple.

Integrating the OS required for the specific ECU with the software components is challenging. The most accessible starting point was a demo

application of an ECU which, even though stripped of any unnecessary functionality, was a substantially sized application. Gaining the understanding required to modify some functionality was non trivial.

3.3.2 Related Porting Work

During the investigative phase of the thesis a number of similar porting efforts was uncovered. Some of the more interesting and related portings are listed below. The porting efforts of note include:

Kaffe ported to L4 :
This was an attempt by Böttcher2004 [87] to port the Kaffe Java virtual machine to the L4 microkernel platform runnin a Dresden Real-Time Operating System. It is worthy of mention as it uses a thread mapping approach as opposed to using green threads.

AUTOSAR ported to Raspberry Pi :
This porting work by Zhang et al.2013 [88, 89] consisted of documenting the experiences of porting AUTOSAR to an embedded system, the Raspberry Pi.

Java in AUTOSAR :
This paper by Wawersich et al.2011 [90] documents the porting of the KESO KVM to AUTOSAR with the goal of enabling Java development in an AUTOSAR environment.

3.3.3 Evaluation

The overall goal of the FRESTA project is to enable vehicles with a platform for Java application development. The goal of this thesis was to port the FRESTA platform to the Volvo CE software platform. Selecting a suitable Java virtual machine, and then getting it operational was time consuming and meant that the thesis only progressed to working with an ECU desktop simulator, as opposed to in a full vehicle.

A number of issues remain to be resolved to complete the porting to a vehicle. This porting thesis work was solely aimed at including a JVM component on a single ECU. Efforts are required to extend the work to include several ECUs via the CAN communication discussed in section 2.1.2.1. An

investigation into which signals should be made available for Java applications should also be conducted, and a study on what effect sending extra CAN messages from applications have on the overall security and robustness of the vehicle communication systems.

Away from the technical challenges of porting the platform, a safety investigation is required to study the impact that extra applications will have on the operators of vehicles. The potential gain from including apps in vehicles should not interfere with, or decrease, the concentration levels of vehicle operators.

4

Java Applications in Resource Limited Embedded Systems

This chapter describes application development for the Java virtual machine that was ported to the ECU.

Contents

4.1 Application Development **36**
4.1.1 Human Machine Interface ECU 36
4.2 Application Implementation **37**
4.2.1 Application Structure 37
4.2.2 Preloading Applicaions 38
4.2.3 Calling Native OS Functionality 38
4.3 Volvo CE ECU Simulator **40**
4.3.1 The Volvo CE Simulator Graphical User Interface . . . 41
4.3.2 The Volvo CE Simulator Display Menu 42
4.3.3 Running a Test Application in the Volvo CE Simulator 42

4.1 Application Development

This chapter will now detail the Java application development process undertaken to design Java applications to test the porting. The decision process, application structure, and implementation challenges will be discussed.

The basic idea was to develop a simple application, app, to test if the porting was successful. It is not within the scope of the thesis to develop a technically challenging app, just a simple application to prove the viability of the porting. With this in mind it was chosen to develop a simple app to flash a parking brake light on a display connected to the human machine interface electronic control system, HMI ECU. A button press would call a Volvo CE software platform component, which would call the KVM, which would run a basic app. This app would be the proof of concept, and show that the porting was possible.

A second app with a never-ending loop would be designed to test if other HMI functionality was disrupted by the running Java app. Ideally an application should run without interfering with, or blocking any other process in the ECU.

4.1.1 Human Machine Interface ECU

The target Electronic Control Unit, ECU, for this thesis was the Human Machine Interface, HMI, ECU. An example of the various input/output mechanisms in a typical HMI ECU is depicted in Figure 4.1. The HMI is the interface between the user and the vehicle with input stalks and buttons, and output screens and dials.

Processing Power:

The processing power in typical HMI units comes from the Central Processing Unit, CPU, of the ECU. This CPU usually comes in the form of a microcontroller. The unit commonly has multiple memory types with flash, external read only memory, RAM, and EEPROM frequently available.

Communication:

The typical HMI ECU connects to an instrument cluster and an information display and has input/output stalks, along with a key-pad, that typically communicates via LIN. The HMI module generally communicates with other ECUs via CAN.

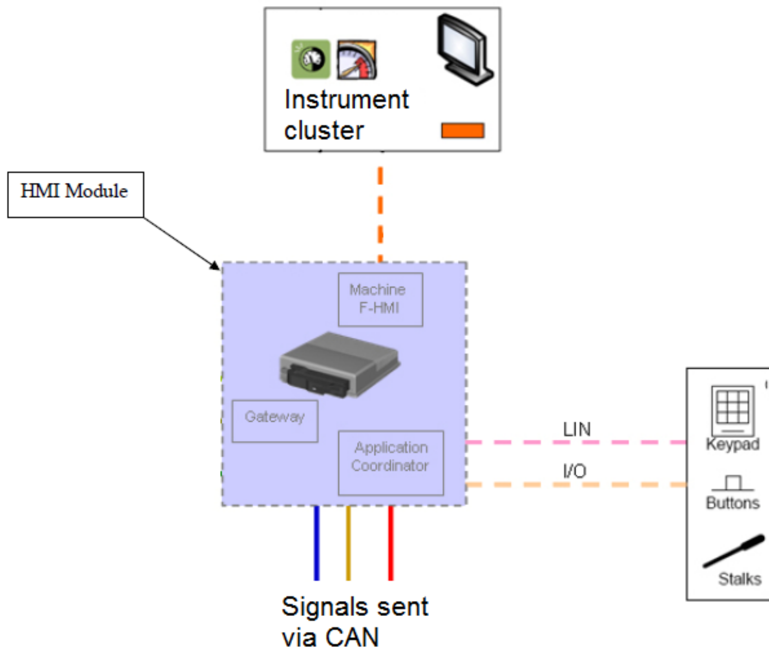

Figure 4.1: Typical HMI electronic control unit block diagram

4.2 Application Implementation

The following sections will describe the implementation considerations undertaken while developing a simple Java app to test the porting. The structure of the application will be discussed in section 4.2.1, followed by the class handling method in section 4.2.2, and finally a method to call native code is discussed in section 4.2.3.

4.2.1 Application Structure

The basic structure of the application is similar to most apps developed for mobile phones [47]. There are three main sections to the app as depicted in Figure 4.2. They are:

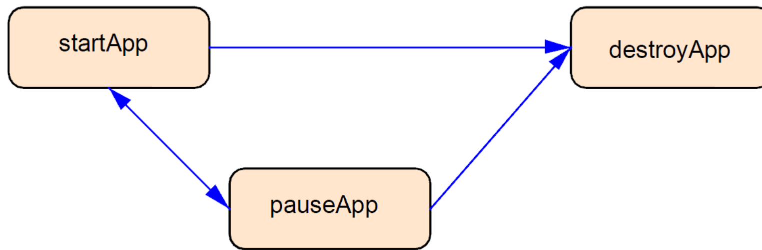

Figure 4.2: Application modes

1. startApp: This is the class called to begin the application and run the main application code.
2. pauseApp: This is called when the app is paused, like closing background activities or record stores
3. destroyApp: Cleanup anything that the garbage collector wouldn't necessarily manage.

4.2.2 Preloading Applicaions

Preloading apps into the KVM, or romizing, is the process where all the core classes are preprocessed into an image form. This image is then burned into ROM where the classes appear to be preloaded and linked. The normal build process for the KVM is as follows:

1. The core classes, and any classes that are added to the classes source folder, are built into a zipped file "classes.zip".
2. The makefile in the Java Code Compact, JCC, directory extracts any unneeded classes from the zip, and creates a new zip file.
3. The new zip is then processed into a C source file called "ROMJavaWin.c" which will be compiled and linked into the KVM.

4.2.3 Calling Native OS Functionality

Usually when developing applications in Java, access to the underlying operating system is through the Java Virtual Machine Java Native Interface, JVM JVI, however the KVM has forfeited this functionality to keep the overall

memory footprint as small as possible.

This means that instead of being able to write a mixture of Java and Native code in applications and expecting the JVM to manage it, all native code must be included in the build process. In the desktop application world this may be seen as a disadvantage, but in automotive electronics this is a very advantageous property. It ensures that Java applications will only have access to the native code functions included by developers at build time, sandboxing the KVM and only giving access to underlying OS signals that the original equipment manufacturer, OEM, decides are safe.

The KVM grants access to the underlying OS by building native code files into a native table c file called "nativeFunctionTable.c" at build time. The steps undertaken to access Native code from KVM will now be discussed.

Naming Convention:

The first consideration when writing functions to allow access to native code is the name of the class. The KVM has a specific naming convention for files that are to be built into the native table.

All files must start with "Java_" and include the class path with periods replaced by underscores. The name of the method must also be included. So a native method called "print", that was part of a class with a class path of "nativeClasses.printClass" would have to be called "Java_nativeClasses_printClass_print" .

Include files:

The native code file must include the "global.h" file. This is because it contains functionality that the native code may need to reference.

The "global.h" file includes methods of passing arguments to and from native code functions. This functionality is described in more detail in Topley2002 [76].

Java Arguments:

Arguments are not passed in and out of functions in the usual way, instead they are pushed and popped to and from the stack using methods declared in the "global.h" file. To access an argument in a native function the Java argument is popped from the stack, and to return values from the native function they are pushed onto the stack.

It is important and all Java arguments passed to the native function are popped before the function returns. If they are not it could cause the KVM to crash [76].

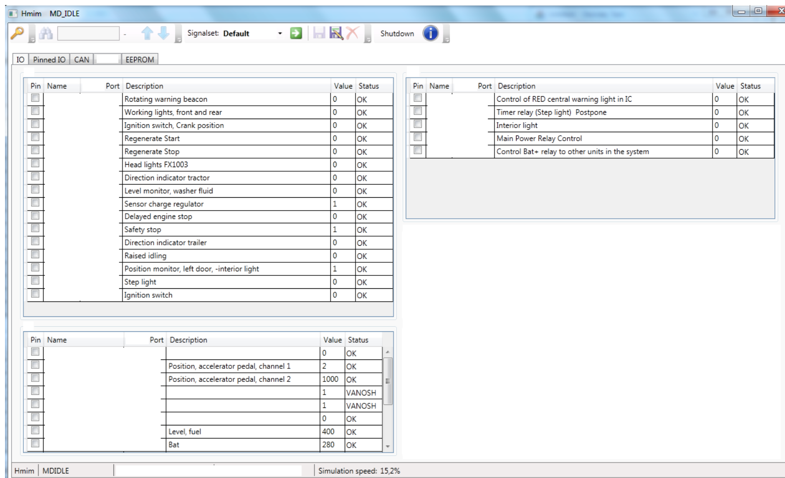

Figure 4.3: Volvo CE simulator check-box options

4.3 Volvo CE ECU Simulator

The Volvo CE Electronic Control Unit, ECU, simulator is an emulator used to replicate ECU software performance in a desktop environment without the need of a physical ECU. This was an incredibly useful software, as testing an application in a desktop environment is less time consuming than debugging the actual hardware. Having a virtual prototype can greatly speed up the development process.

The Volvo CE simulator comes with a checkbox option manager to choose ECU functionality. Multiple options are available to manipulate the ECU behavior as can be seen in Figure 4.3. The settings can be changed simply by changing the relevant check-box. This is a practical method of swiftly changing various ECU functionality, without the need to search through thousands of lines of code and manually change flags to enable the desired process.

The Volvo CE simulator works by adding ECU project files to the Volvo CE simulator directories. The system is based on a server client architecture with each ECU as a client. The drawback of the simulator is that it does not emulate a real-time environment, i.e. the systems processing speed, but rather an approximation.

The porting work carried out in this thesis was developed using the Volvo CE simulator as a platform to verify functionality. It was advantageous to have the ability to quickly test code to confirm performance after development.

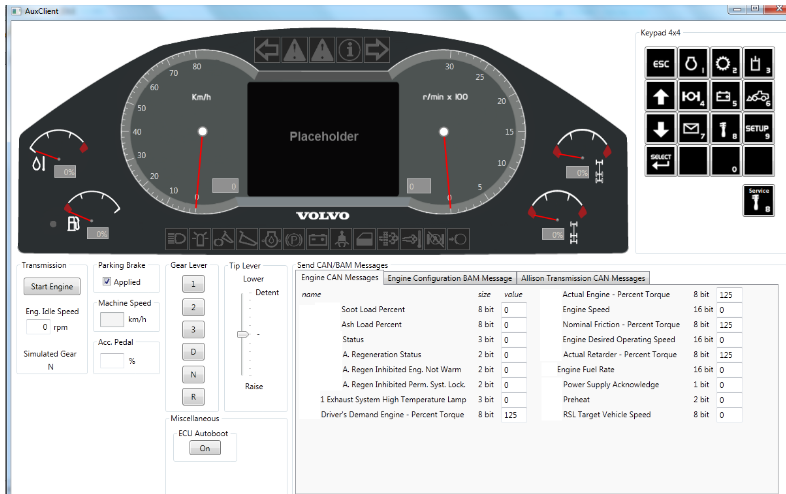

Figure 4.4: The Volvo CE simulator GUI

4.3.1 The Volvo CE Simulator Graphical User Interface

The Volvo CE simulator generates a heads up Graphical User Interface, GUI, as can be seen in Figure 4.4. The simulator GUI is used to visually confirm that the relevant ECU is operating as expected after adding modifying the system code as described in section 4.3.3. This is a simple way to quickly test and verify ECU functionality without having the actual ECU hardware present.

There is an emulated keypad to the left of the GUI which is used to simulate input from a user, and a selection of dials and LEDs to demonstrate ECU activities. The emulator can be put into gear, the desired speed in kilometers per hour, and the speedometer dial increases to the desired speed. There is also a sub-menu in the bottom corner to show CAN messages sent to other ECUs. This will be useful when the system is extended to include Java virtual machines across multiple ECUs. The CAN messages transmitted and received can be tracked through this useful menu.

The aim of the application developed to test the porting was to flash the parking break LED on the row of LEDs under the speedometer, by clicking one of the emulated buttons on the left hand side of the GUI. The button to be pressed and the LED in question is depicted in Figure 4.6.

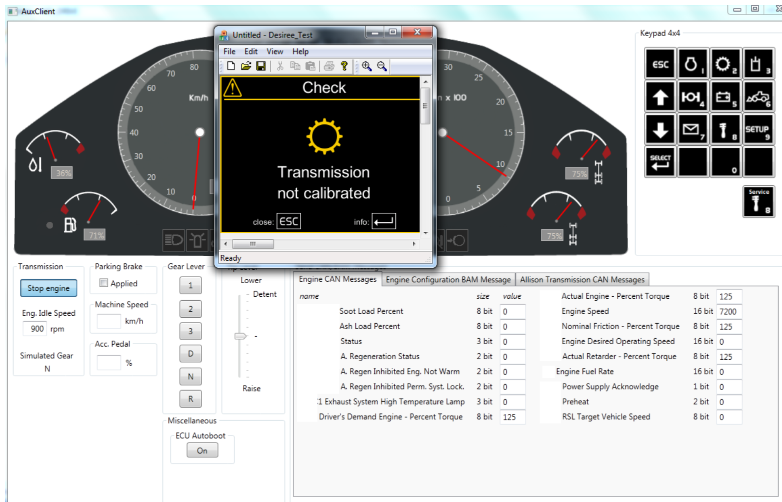

Figure 4.5: The Volvo CE simulator Display Menu

4.3.2 The Volvo CE Simulator Display Menu

The Volvo CE simulator also includes a display screen, similar to the LCD display on an actual vehicle, that is shown in Figure 4.5. This display is used to convey error messages, along with general functions like time and date. The screen is used to display a menu and has capabilities of checking various system functions. The display is interfaced through the input buttons seen on the left hand side of Figure 4.4.

4.3.3 Running a Test Application in the Volvo CE Simulator

The Volvo CE simulator is based on Microsoft Visual Studio, MVS, projects, so the ECU code to be simulated has to be in that format. To run the simulator the relevant ECU MVS project is included in the correct Volvo CE simulator directory. The Volvo CE simulator then emulates the ECU behavior based on the software project added to the directory.

The functionality of the new code is then tested by observing how the system reacts under different inputs from either the emulated buttons, or the check box options described in section 4.3. If the system does not behave as expected the code can be modified and the simulations run again.

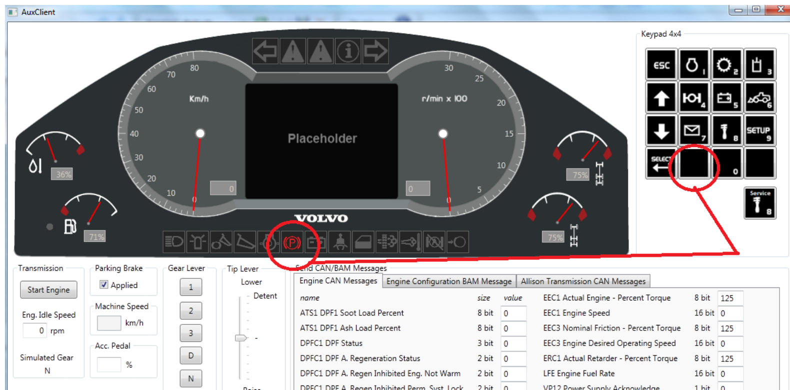

Figure 4.6: Button and Parking Brake LED

The goal of the simulation part of the porting was to develop the system so that a button pressed would call the KVM which would call an application to flash a led on the heads-up GUI. Button 'X' was chosen as the button as it did not have any operations assigned to it as default, and the parking brake LED was chosen as the LED to be flashed, as depicted on Figure 4.6.

5

Results and Conclusions

This chapter discusses the observed results and conclusions drawn from the investigative work.

Contents

5.1 Results **46**
5.2 Future Work **46**
5.2.1 Volvo CE Software Platform 47
5.2.2 Computing Power Increments 47
5.3 Conclusions **47**
5.3.1 Final Synopsis 49

5.1 Results

The applications show that it is possible to enable an Electronic Control Unit, ECU, to run Java applications without disrupting vehicle functionality. Although the application was only a flashing LED, it could easily be enhanced to turn on the vehicle headlights or one that can utilize other signals.

As an example the application could possibly be extended into an application that connects to the internet, checks if the weather is overcast in the area and automatically turns on the headlights. Alternatively an app could be developed to communicate with the vehicle Global Positioning System, GPS, and turn on the vehicle indicator in advance as it comes to junctions on the route. The possibilities are endless.

5.2 Future Work

There is a lot of potential for future work with regards to the thesis work. Firstly the entire porting should be subjected to an exhaustive testing process. There is also scope to conduct a study on safety, not just in the safety of the system itself, but how users interact with the extra applications added to vehicles and if the added distraction of the applications contributes to an excessive distraction to the vehicle operator.

The porting plan applied in this project should be extended to incorporate the full porting plan initially envisaged in section 3.1.2. At the beginning of the thesis a decision was made to use the K virtual machine as the Java Virtual Machine, JVM, of choice for the thesis porting. During the months that followed project FRESTA decided that the Squawk virtual machine mentioned in section 2.4 may be a preferable alternative so an investigation into porting Squawk may be required.

This project focused on a single ECU, and it could be extended to add JVMs to multiple ECUs. This would involve studying if applications could be spread over several ECU's, and if so how would communication happen. The communication would most likely be over the Controller Area Network, CAN, described in section 2.1.2.1, but how would the extra CAN traffic introduced by applications effect the overall communication systems in the vehicles.

5.2.1 Volvo CE Software Platform

This thesis was based on the older Volvo CE software platform. However there is a newer version of the platform available and one of the areas of future work would be adapting the porting to run on the latest platform.

5.2.2 Computing Power Increments

ECUs are gaining more memory and more processing power with each passing year. This means that there will be increasing amounts of memory available to implement better garbage collection algorithms, and enough memory to have a fully preemptive garbage collector. Better and more predictable JVMs could be studied and implemented as more computing power becomes available.

5.3 Conclusions

At the beginning of the thesis a number of questions were asked. Given the research conducted a number can now be answered.

"How will Java survive in resource constrained embedded systems?"
Although the viability of Java virtual machines has been proved to an extent in this thesis work, there is still a way to go before real time Java can be used in hard real time resource constrained embedded systems. The main reasoning behind this conclusion is that the Garbage Collector, GC, technology available today is not quite predictable enough.

The pre-study phase of this project uncovered a number of professional lightweight virtual machines that could boast a fully preemptive garbage collector, but further study revealed that there are still sections where the GC cannot be preempted. These sections do however have a Worst Case Execution Time, WCET, so the question becomes "Does the advantages the GC provides - no memory leaks etc. - justify developing embedded systems in Java at the expense of processing time for the GC and longer WCETs for hard real time tasks?". That would come down to the system requirements for different

projects.

Of course one option would be to disable the GC and follow the Ravenscar profile detailed in section 2.2.3.3. This profile strives to minimize the use of dynamically created objects amongst other things, but rather than introducing limitations to creating applications perhaps the best approach would be to find a feasible approach to overcome the disadvantages of the GC given the tremendous advantages.

"Porting a full version of Java with all the capabilities of those used in desktop computers, or modern mobile phones, is out of the question due to memory and processor constraints, so how does embedded Java fair?"

The answer is quite well. There are constraints introduced due to a limited number of classes in the JVM, its not the full Java set of classes on a desktop, but there are enough to develop useful applications. Another solution would be to use a different JVM with more classes available.

"Is the subset of classes available to embedded Java enough to entice third party developers to design applications?"

This question was partly answered above but to expand a little - developers should not look at the limitation of classes available as a disadvantage, rather a design challenge. The fact that not all Java classes are available is also a timely reminder that the application is being developed for an embedded system, and as such a little constraint may be needed. i.e. An application should not be overly resource hungry, or excessively intrusive that it causes an exorbitant distraction to the user.

"How does including Java in a resource constrained hard real time embedded system effect predictability and overall security of the system?"

In the design approach taken by the FRESTA platform, the answer is not at all. The JVM is sandboxed from all other important signals. As the KVM is running in a lower priority task than all hard real time tasks in the system, it does not have a negative impact on predictability or overall security of the system. The applications themselves will never be hard real time tasks in this environment, as the JVM is running in a lower priority task, but there is still great potential to develop innovative applications for the system.

"Is it feasible to have deterministic Java in resource constrained hard real time embedded systems?

There is evidence that embedded Java virtual machines, especially the professional expensive JVMs, are quite close to having deterministic Java. The

solution available is a mostly preemptive garbage collector that has small, deterministic, stop the world parts. For these systems the WCET of every task is increased by the estimated bounding time of the GC.

Fully introducing unrestricted Java means introducing a GC of some type, which means extra processing time is needed. This is an obvious disadvantage as every resource is crucial in embedded systems, but the advantages gained over not including a GC makes a strong case for deterministic Java. A system that can almost never break due to memory errors is quite advantageous, and that asset is one that may make developers in the future consider embedded Java a little more strongly.

5.3.1 Final Synopsis

The thesis set out to investigate some of the questions associated with integrating a platform for Java application development in a resource constrained embedded system. The answers revealed that it is possible to include the mechanism and that the platform could go some way to alleviating many of the disadvantages concerned with mixed criticality systems in vehicles.

The Java founding motto of "Write once, run anywhere" has revolutionized desktop software development and mobile phone development by enabling developers to create highly portable software with minimum effort, but if the Java virtual machine, JVM, of choice was the K virtual machine this motto would be broken as Java applications developed for mobile phones would have to be modified to run with the reduced class library of the K virtual machine.

The way around this could be by choosing a different Java virtual machine like Squawk, or a professionally licensed JVM that includes a full library of Java classes. Overall including Java as a programming language of choice is quickly becoming more of an option as embedded system resources grow and JVMs become smaller and more efficient.

Bibliography

[1] Mark Kirby. (2010, June) Mobile app development trends – what languages should you be learning? Mark Kirby - Mobile Developer. [Online]. Available: http://mark-kirby.co.uk/2010/mobile-app-development-trends/

[2] D. Newcomb. (2014, Feb.) Ford, GM open their dashboards to outside developers. WIRED. [Online]. Available: http://www.wired.com/autopia/2013/01/ces-2013-ford-gm-app-developers/

[3] J. Rohatynski. (2014, Jan.) NAIAS newsroom. North American International Auto Show. [Online]. Available: http://naias.mediaroom.com/index.php?s=27808

[4] Susan Kuchinskas. (2014, Feb.) State of the automotive app store. Telematics Update. [Online]. Available: http://analysis.telematicsupdate.com/infotainment/state-automotive-app-store-part-i

[5] Schneider, Jörn, "Overcoming the Interoperability Barrier in Mixed-Criticality Systems," in *Concurrent Engineering Approaches for Sustainable Product Development in a Multi-Disciplinary Environment*, Stjepandić, Josip and Rock, Georg and Bil, Cees, Ed. Springer London, 2013, pp. 1093–1104.

[6] J. Laukkonen. (2014, Feb.) GM's OnStar Service: How Does It Work? About.com. [Online]. Available: http://cartech.about.com/od/Safety/a/Gms-Onstar-Service-How-Does-It-Work.html

[7] S. Freeman. (2006, Feb.) How OnStar Works. How Stuff Works. [Online]. Available: http://auto.howstuffworks.com/onstar.html

[8] G. Motors. (2014) OnStar homepage. [Online]. Available: https://www.onstar.com

[9] C. Woodyard. (2014, Feb.) Device can remotely halt auto chases. ABC News. [Online]. Available: http://abcnews.go.com/Business/Autos/story?id=3706113

[10] M. Rohlin. (2014, Feb.) OnStar stops truck that was carjacked at gunpoint. Los Angeles Times. [Online]. Available: http://latimesblogs.latimes.com/technology/2009/10/onstar-gps-carjacking.html

[11] T. Simonite. (2013, Jan.) GM and Ford open up their vehicles to app developers. Technology Review. [Online]. Available: http://www.technologyreview.com/news/509736/gm-and-ford-open-up-their-vehicles-to-app-developers/

[12] Apple. (2014, Feb.) Develop apps for iPad. Apple Developer. [Online]. Available: https://developer.apple.com/ipad/sdk/

[13] K. Fitchard. (2013, Jan.) At CES the connected car became truly connected. GIGAOM. [Online]. Available: http://gigaom.com/2013/01/12/at-ces-the-connected-car-became-truly-connected/

[14] E. Olman. (2014, Jan.) AutoGuard: Keeping your Car Safe from Hacks. Cisco. [Online]. Available: http://blogs.cisco.com/sp/autoguard-keeping-your-car-safe-from-hacks/more-134620

[15] P. Roberts. (2014, Jan.) Cisco eyes security services for connected cars. The security ledger. [Online]. Available: https://securityledger.com/2014/01/cisco-eyes-security-services-for-connected-cars/

[16] R. Faas. (2014, Feb.) How Apple's iOS in the car could transform the way you drive. CITE World. [Online]. Available: http://www.citeworld.com/mobile/23015/Apple-iOS-in-the-car

[17] iOS7. (2014, Feb.) iOS in the Car. Best passenger ever. Apple. [Online]. Available: http://www.apple.com/ios/whats-new/

[18] M. Rosoff. (2013, Dec.) Mobile developers have high hopes for iOS in the car. CITE World. [Online]. Available: http://www.citeworld.com/development/22796/ios-car-high-hopes

[19] Google. (2014, Feb.) Open Automotive Alliance homepage. Open Automotive Alliance. [Online]. Available: http://www.openautoalliance.net/about

[20] Rose Etherington. (2014, Jan.) Google joins forces with major auto brands to bring Android to car dashboards. de Zeen magazine. [Online]. Available: http://www.dezeen.com/2014/01/07/google-launches-open-automotive-alliance-for-android-connected-cars/

[21] GENIVI. (2014, Feb.) GENIVI Homepage. GENIVI. [Online]. Available: http://www.genivi.org/

[22] Ed Scannell. (2009) GENIVI Alliance driving linux infotainment stack. InformationWeek. [Online]. Available: http://www.informationweek.com/genivi-alliance-driving-linux-infotainment-stack/d/d-id/1077263?

[23] OpenXC. (2014, Jan.) The OpenXC Platform. OpenXC. [Online]. Available: http://openxcplatform.com/overview/index.html

[24] Jon Stewart. (2012) Connected cars open up to apps and the cloud. BBC. [Online]. Available: http://www.bbc.com/future/story/20120719-road-opens-for-connected-cars

[25] Kim Hill,Debra Menk,Bernard Swieck, and Joshua Cregger. (2014, Jan.) Just How High-Tech is the Automotive Industry? Center for Automotive Research, CAR. [Online]. Available: http://www.cargroup.org/?module=Publications&event=View&pubID=103

[26] Susan Kuchinskas. (2012) Telematics and the hybrid approach to content delivery. Telmatics update. [Online]. Available: http://analysis.telematicsupdate.com/infotainment/telematics-and-hybrid-approach-content-delivery

[27] W. Ribbens and N. Mansour, *Understanding Automotive Electronics*, ser. SAE-R. Newnes, 2003. [Online]. Available: http://books.google.se/books?id=lu9BhR2T20YC

[28] TRW, "Cognitive Safety Systems," TRW, Feb 2012. [Online]. Available: http://www.trw.com/electronic_systems

[29] J. Schneider, "Overcoming the interoperability barrier in mixed-criticality systems," in *Concurrent Engineering Approaches for Sustainable Product Development in a Multi-Disciplinary Environment*, J. Stjepandic, G. Rock, and C. Bil, Eds. Springer London, 2013, pp. 1093–1104. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-4426-7_92

[30] U. Drolia, Z. Wang, Y. Pant, and R. Mangharam, "Autoplug: An automotive test-bed for electronic controller unit testing and verification," in *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*, Oct 2011, pp. 1187–1192.

[31] Freescale. (2014, May) MPC5510: Qorivva 32-bit MCU for Body Electronics Applications. MPC5510 block diagram. [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC5510

[32] J. Trop, “Toyota seeks a settlement for sudden acceleration cases,” The New York Times, Dec 2013. [Online]. Available: http://www.nytimes.com/2013/12/14/business/toyota-seeks-settlement-for-lawsuits.html

[33] J. Yoshida, “Toyota case: Single bit flip that killed,” EE Times, Oct 2013. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1319903

[34] CiA, “CAN history,” CAN in automation, Feb 2014. [Online]. Available: http://www.can-cia.de/index.php?id=161

[35] A. Emadi, *Handbook of Automotive Power Electronics and Motor Drives*, ser. Electrical and Computer Engineering. Taylor & Francis, 2005. [Online]. Available: http://books.google.se/books?id=c984D31D2sQC

[36] H. Hansson, J. Carlson, D. Isovic, K. Lundqvist, T. Nolte, M. Ouimet, P. Pettersson, S. Punnekkat, and C. Seceleanu, *Real-Time Systems*. Fraunhofer IESE, February 2010, full text not available. Contact http://www.zfuw.de (or the authors) if you are interested in using the book. [Online]. Available: http://www.es.mdh.se/publications/1775-

[37] K. Tindell, H. Hansson, and A. J. Wellings, “Analysing real-time communications: controller area network (can),” in *Real-Time Systems Symposium, 1994., Proceedings.*, Dec 1994, pp. 259–263.

[38] R. Zurawski, *Embedded systems handbook*. CRC Press, 2005.

[39] N. Navet and F. Simonot-Lion, *Automotive embedded systems handbook*. CRC press, 2008.

[40] CVEL, “Automotive data communication buses,” The Clemson University electornics laboratory, Feb 2014. [Online]. Available: http://www.cvel.clemson.edu/auto/auto_buses01.html

[41] S. Ritter, “Top ten reasons for using java in embedded apps,” *Java Magazine*, pp. 20–25, Jan 2013.

[42] G. Bollella and J. Gosling, “The real-time specification for java,” *Computer*, vol. 33, no. 6, pp. 47–54, Jun 2000.

[43] M. Higuera-Toledano, S. Yovine, and D. Garbervetsky, “Region-based memory management: An evaluation of its support in rtsj,” in *Distributed, Embedded and Real-time Java Systems*, M. T. Higuera-Toledano and A. J. Wellings, Eds. Springer US, 2012, pp. 101–127. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-8158-5_5

[44] J. Gosling and G. Bollella, *The Real-Time Specification for Java*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[45] T. Barr and S. Meloan, “Embedded everywhere,” *Java Magazine*, pp. 20–25, Jan 2013.

[46] D. Mulchandan. (1998, May) Java for embedded systems. Wind river system. [Online]. Available: http://www.computer.org/internet

[47] Q. H. Mahmoud, *Learning wireless java*. O’Reilly Media, Inc., 2002.

[48] Oracle. (2014, Mar.) Oracles Hotspot Java virtual machine. Oracle. [Online]. Available: http://www.oracle.com/technetwork/java/embedded/downloads/javame/java-embedded-java-me-download-359231.html?ssSourceSiteId=ocomen

[49] P. Dibble, J. Hunt, and A. Wellings, “Programming embedded systems: Interacting with the embedded platform,” in *Distributed, Embedded and Real-time Java Systems*, M. T. Higuera-Toledano and A. J. Wellings, Eds. Springer US, 2012, pp. 129–158. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-8158-5_6

[50] B. Venners, *Inside the Java virtual machine.* McGraw-Hill, Inc., 1996.

[51] P. Basanta-Val and J. Anderson, “Using real-time java in distributed systems: Problems and solutions,” in *Distributed, Embedded and Real-time Java Systems*, M. T. Higuera-Toledano and A. J. Wellings, Eds. Springer US, 2012, pp. 23–44. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-8158-5_2

[52] M. Zabel, T. Preusser, P. Reichel, and R. Spallek, “Secure, real-time and multi-threaded general-purpose embedded java microarchitecture,” in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, Aug 2007, pp. 59–62.

[53] M. S. Johnstone, “Non-compacting memory allocation and real-time garbage collection,” Tech. Rep., 1996.

[54] D. F. Bacon, P. Cheng, and V. T. Rajan, “A real-time garbage collector with low overhead and consistent utilization,” *SIGPLAN Not.*, vol. 38, no. 1, pp. 285–298, Jan. 2003. [Online]. Available: http://doi.acm.org/10.1145/640128.604155

[55] H. G. Baker, Jr., “List processing in real time on a serial computer,” *Commun. ACM*, vol. 21, no. 4, pp. 280–294, Apr. 1978. [Online]. Available: http://doi.acm.org/10.1145/359460.359470

[56] F. Siebert, “Concurrent, parallel, real-time garbage-collection,” *SIGPLAN Not.*, vol. 45, no. 8, pp. 11–20, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837855.1806654

[57] B. Dobbing, “The ravenscar profile for high-integrity java programs?” in *Proceedings of the 10th International Workshop on Real-time Ada Workshop*, ser. IRTAW ’00. New York, NY, USA: ACM, 2001, pp. 56–61. [Online]. Available: http://doi.acm.org/10.1145/374370.374382

[58] F. Yellin, “Inside the K Virtual Machine,” *Sun’s 2000 Worldwide Java Developer Conference*, 2000. [Online]. Available: http://www.mobilejava.co.kr/bbs/temp/cldcboard/Inside/%20the/%20KVM.pdf

[59] A. Courbot, G. Grimaud, J.-J. Vandewalle, and D. Simplot-Ryl, “Application-driven customization of an embedded java virtual machine,” in *Embedded and Ubiquitous Computing – EUC 2005 Workshops*, ser. Lecture Notes in Computer Science, T. Enokido, L. Yan, B. Xiao, D. Kim, Y. Dai, and L. Yang, Eds. Springer Berlin Heidelberg, 2005, vol. 3823, pp. 81–90. [Online]. Available: http://dx.doi.org/10.1007/11596042_9

[60] D. Masson and S. Midonnet, “Handling non-periodic events in real-time java systems,” in *Distributed, Embedded and Real-time Java Systems*, M. T. Higuera-Toledano and A. J. Wellings, Eds. Springer US, 2012, pp. 45–77. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-8158-5_3

[61] B. Sprunt, L. Sha, and J. Lehoczky, “Aperiodic task scheduling for hard-real-time systems,” *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989. [Online]. Available: http://dx.doi.org/10.1007/BF02341920

[62] M. Sung, S. Kim, S. Park, N. Chang, and H. Shin, “Comparative performance evaluation of java threads for embedded applications: Linux thread vs. green thread,” *Information Processing Letters*, vol. 84, no. 4, pp. 221 – 225, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019002002867

[63] B. Sanden, “Coping with java threads,” *Computer*, vol. 37, no. 4, pp. 20–27, April 2004.

[64] AUTOSAR, “AUTomotive Open System Architecture homepage,” AUTOSAR, Jan 2012. [Online]. Available: http://www.autosar.org

[65] 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*, April 2013. [Online]. Available: http://www.es.mdh.se/publications/3264-

[66] S. Schmerler and R. Rimkus, “Autosar — shaping the future of a global standard,” *ATZelektronik worldwide*, vol. 8, no. 1, pp. 42–45, 2013. [Online]. Available: http://dx.doi.org/10.1365/s38314-013-0147-0

[67] F. Kirschke-Biller, “Autosar-a worldwide standard current developments, roll-out and outlook,” in *15th International VDI Congress Electronic Systems for Vehicles, Baden-Baden, Germany*, 2011.

[68] Microsoft. (2014, June) Microsoft Visual Studio. Microsoft. [Online]. Available: http://www.visualstudio.com/

[69] Eclipse. (2014, June) Eclipse CDT tool. CDT Project. [Online]. Available: http://www.eclipse.org/cdt/

[70] SICS. (2014, Feb) FRESTA - apps in vehicles. SICS. [Online]. Available: https://www.sics.se/projects/fresta-apps-in-vehicle

[71] Volvo Group. (2014, May) Volvo Group Global. Volvo Group. [Online]. Available: http://www.volvogroup.com/group/global/en-gb/Pages/group_home.aspx

[72] Volvo Car. (2014, May) Volvo Car Corporation. Volvo Car. [Online]. Available: http://www.volvocars.com/pages/default.aspx

[73] SICS, “Swedish ICT homepage,” SICS, Jan 2014. [Online]. Available: https://www.sics.se

[74] A. Kobetski and J. Axelsson, “On the technological and methodological concepts of federated embedded systems,” in *First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering*, May 2013. [Online]. Available: http://www.es.mdh.se/publications/3268-

[75] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, “Java™ on the bare metal of wireless sensor devices: The squawk java virtual machine,” in *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, ser. VEE ’06. New York, NY, USA: ACM, 2006, pp. 78–88. [Online]. Available: http://doi.acm.org/10.1145/1134760.1134773

[76] K. Topley, *J2ME in a Nutshell: A Desktop Quick Reference*, ser. In a Nutshell (o’Reilly) Series. O’Reilly, 2002. [Online]. Available: http://books.google.se/books?id=ieBA3-Q-V6sC

[77] Raspberry Pi. (2014, May) Raspberry Pi homepage. Raspberry Pi. [Online]. Available: http://www.raspberrypi.org/

[78] S. Zhang, A. Kobetski, E. Johansson, J. Axelsson, and H. Wang, "Porting an autosar-compliant operating system to a high performance embedded platform," in *3rd Embedded Operating Systems Workshop*, August 2013. [Online]. Available: http://www.es.mdh.se/publications/3203-

[79] C. Wawersich, I. Stilkerich, and M. Stilkerich, "The Use of Java in the Context of AUTOSAR 4.0," in *Embedded World Proceedings & Conference Materials*, K. Scheinig, Ed., Nürnberg, Germany, 2011. [Online]. Available: http://www4.cs.fau.de/Publications/2011/wawersich-11-ew.pdf

[80] Oracle. (2014, Mar.) Oracles K Java virtual machine. Oracle. [Online]. Available: http://www.oracle.com/technetwork/java/ds-137153.html

[81] ——. (2014, June) Oracle homepage. ORACLE. [Online]. Available: http://www.oracle.com/index.html

[82] Derek White. (2014, May) Squawk Development Wiki. Java.net. [Online]. Available: https://java.net/projects/squawk/pages/SquawkDevelopment

[83] Java.net. (2014) Homepage. Java.net. [Online]. Available: https://www.java.net

[84] T. Higuera, V. Issarny *et al.*, "Analyzing the performance of memory management in rtsj," in *Symposium on Object-Oriented Real-Time Distributed Computing: ISORC 2002*, 2002, pp. 26–33.

[85] *KVM porting guide,CLDC version 1.1.* Sun Microsystems, 2003.

[86] R. Hat, "Cygwin," 2005. [Online]. Available: https://www.cygwin.com/

[87] A. Böttcher, "Port of the java virtual machine kaffe to drops by using l4env," 2004.

[88] S. Zhang, A. Kobetski, E. Johansson, J. Axelsson, and H. Wang, "Porting an autosar-compliant operating system to a high performance embedded platform," *ACM SIGBED Review*, vol. 11, no. 1, pp. 62–67, 2014.

[89] S. Zhang, "Porting autosar to a high performance embedded system," 2013. [Online]. Available: http://www.idt.mdh.se/examensarbete/index.php?choice=show&lang=en&id=1442

[90] C. Wawersich, I. Thomm, and M. Stilkerich, "The use of java in the context of autosar 4.0," *Embedded World, Nuremberg, Germany, March*, 2011.

A

Appendix: *Software List and Versions*

Contents:

Software List *A List of the software, and versions used throughout the thesis*

Software List

- *Volvo Software Platform v2.4.4:* Component based software development IDE.
- *Oracle K Virtual Machine v1.1:* A Java Virtual Machine aimed at resource constrained embedded systems.
- *Eclipse "Juno":* Software development IDE used to develop FRESTA platform.
- *Microsoft Visual Studio 12:* Used to integrate KVM and ECU code.
- *FRESTA:* App enabling environment.
- *Volvo CE Simulator v2.2.1:* Simulation tool for ECU development.
- *TexWorks v0.4.3:* All official documentation were produced using LaTeX and TexWorks.
- *TimesTool v1.3 beta* Mode diagrams were constructed using TimesTool
- *Microsoft Project and Visio:* All project planning was done using Microsoft project and Visio.
- *Windows 7 and Windows Vista:* Development was carried out on windows OS platforms.
- *Cygwin GNU bash, version 4.1.10(4)-release (i686-pc-cygwin):* A set of tools to provide a Unix-like environment for Microsoft Windows.