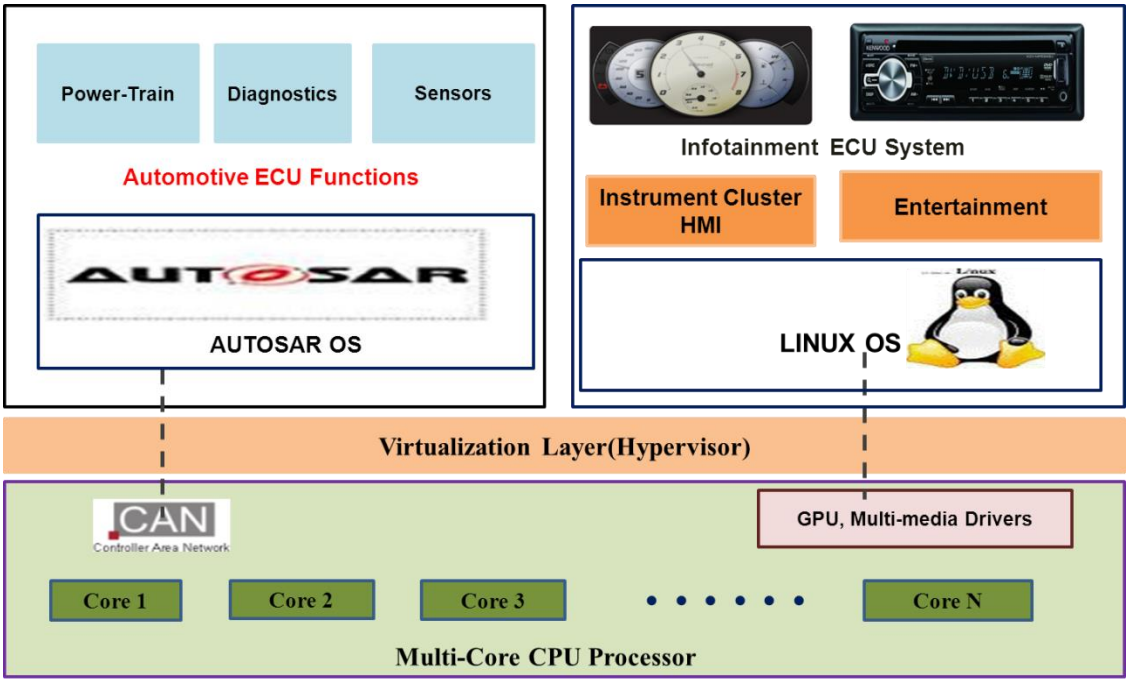
CHALMERS



AUTOSAR and Linux – Single chip solution

Implementation of Automotive Multipurpose ECU Prototype system using hypervisor solution

Master of Science Thesis in the Master Degree Program, Networks and Distributed Systems

KARTHIKEYAN RAVINDRAN
XINGGE XU

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, February 2015

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

AUTOSAR and Linux – Single chip solution

Implementation of Automotive Multipurpose ECU Prototype system using hypervisor solution

KARTHIKEYAN RAVINDRAN
XINGGE XU

© KARTHIKEYAN RAVINDRAN, February 2015.
© XINGGE XU, February 2015.

Supervisor and Examiner: **Elad Michael Schiller**

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: A Visual illustration of a Multipurpose ECU system is hosting both Linux and AUTOSAR automotive applications simultaneously on a common shared target platform using a hypervisor solution. Image is drawn in courtesy of the OpenSynergy's COQOS product. (http://www.opensynergy.com/en/products/coqos/products/details/)

Department of Computer Science and Engineering
Göteborg, Sweden February 2015

Abstract

The rapid explosion in the complex Electrical and Electronics (E/E) system components in modern automotive systems has resulted in an intricate vehicle on board system design with more than 50 Electronic Control Unit (ECU) nodes. These ECUs are interconnected by various communication networks, implementing diverse functionalities and running more than million lines of code. At one end, there are classical ECU systems implementing core automotive features like active safety, passenger comfort and convenience, etc. characterized by demand for hard real-time behavior and high reliability. At the other end, there are rapidly emerging resource intensive infotainment and connectivity ECUs offering rich multimedia, navigation assistance and Human Machine Interface (HMI) applications. There is a growing need among automotive suppliers and original equipment manufacturers (OEMS) for overall ECU system consolidation as well as integration and interaction between these two heterogeneous ECU systems in the future vehicles. AUTomotive Open System ARchitecture (AUTOSAR) is widely accepted as the standardized automotive software architecture for developing vehicular applications, and it uses OSEK real-time operating system (RTOS) specifications as the basis for AUTOSAR OS. It provides perfect support for the classical automotive control and real-time functionality; however, it fails to provide support for hosting infotainment and user applications that in turn limits the prospects of consolidated ECU integration by a common platform. Likewise, general purpose operating system (GPOS) such as Linux, Android, etc. are used for non-real-time infotainment and connectivity applications, and they are not inherently designed to support the real-time control applications.

In order to integrate the classical automotive ECU system, implementing core automotive features with the infotainment and connectivity ECU system, running HMI and user applications, automotive manufacturers have started considering the feasibility of deploying the virtualization technology in automotive embedded systems to support both AUTOSAR architecture and GPOS in a single platform. In this thesis work, we propose a design (with a corresponding pilot implementation) for a consolidated, multipurpose ECU prototype model for simultaneously hosting Linux and AUTOSAR applications in the same hardware platform using an existing hypervisor solution named as COQOS. We start by exploring the various consolidation strategies for the multipurpose ECU prototype and then propose and motivate a suitable hypervisor solution, hardware platform and reference applications for the prototype. Our study concentrates on exploratory design and implementation of the proposed prototype model of consolidated ECU system. This thesis work will serve as a good case study about integrating automotive application based on AUTOSAR and infotainment application based on Linux OS in a common platform and the necessary modifications required for this integration approach using virtualization technology. The report can be the basis for a comprehensive analysis about the key challenges and necessary technology enhancements required for implementing full-scale deployment of the multipurpose ECU system architecture using virtualization technology in the near future.

Keywords: Multipurpose ECU system, System consolidation, AUTOSAR 4.0, OSEK OS, Infotainment, COQOS Hypervisor, PikeOS microkernel, Resource partitioning,

Inter-partition communication, PICEA suite, POPULUS, functional safety, I.MX53 SABRE, Multicore architecture.

Preface

This Master thesis was a proposal made by MECEL AB to carry out research analysis regarding the technical strategies required for design and development of an integrated ECU demo system executing AUTOSAR and Linux automotive applications concurrently in a single chip target hardware using a Virtualization (hypervisor) solution. In this thesis work, we collaborated with OpenSynergy GmbH and utilized their hypervisor product solution, COQOS for the implementation of the multipurpose ECU system.

Mecel AB is software and systems consulting firm with more than 25 years of experience in developing solutions and services for the automotive industry. **OpenSynergy**, a specialist in automotive embedded systems, have developed COQOS, a standard based automotive virtualization software solution. It enables safe and secure integration of Linux/Android based infotainment, ADAS and Telematics software applications as well as AUTOSAR based core automotive functionalities to run on common target hardware.

Acknowledgements

During this thesis work, we have been in contact with many people who have provided a much-needed support and help. First we would like to thank our examiner and supervisor Elad Michael Schiller for his consistent support, encouragement and critical feedback during the course of our thesis work.

We would like to express our gratitude sincerely to our supervisor in MECEL AB, Lars Henrik Lottberg for his continuous guidance, interest and for spending extended time with us to sort the technical problems faced by us. We are also extremely grateful to Mathias Fritzson, who acted as a second supervisor mentoring our technical work and providing invaluable suggestions. It was a wonderful experience working with both of you and many thanks for the enthusiasm and sharing your broad knowledge and insights in the automotive domain with us.

We sincerely acknowledge the assistance extended by Anders Arnholm, Claes Ivarsson, Christopher Olofsson and several other software engineers from PICEA and POPULUS product teams. Also, many thanks to MECEL AB and their staff especially Jonas Arkensved and Karin Denti for giving this wonderful opportunity and providing us with the necessary equipment and resources to be able to finish this thesis work. We also sincerely express our gratitude to OpenSynergy GmbH for providing the COQOS hypervisor product solution and target hardware board with discounted price that was vital for our prototype development work at MECEL AB. Big thanks also goes to Andreas Haase and the R&D engineers from OpenSynergy, who were in touch with us to provide the essential software support and for participating in the professional discussions regarding this topic.

Finally, we would like to thank our parents, family and friends for their great love, understanding, motivation and continuous support.

Table of Contents

List of Figures

List of Tables

List of Abbreviations and Acronym Terms

ABS	*Anti-lock Braking System*
ADAS	*Advanced Driver Assistance Systems*
ARXML	*AUTOSAR Extensible Mark-up Language File*
AUTOSAR	*Automotive Open System Architecture*
BSP	*Board Support Package Software*
BSW	*Basic Software Layer*
CAN	*Controller Area Network*
CANIF	*Controller Area Network Interface Layer*
CODEO	*PikeOS Integrated Development Environment based on Eclipse platform*
COM	*AUTOSAR Communication Layer*
E/E	*Electrics/Electronics Systems*
ECU	*Electronic Control Unit*
GCC/GDB	*GNU Compiler Collection/Project Debugger*
GPOS	*General Purpose Operating System*
GPS	*Global Positioning System*
GPU/VPU	*Graphics/Visual Processing Unit*

HMI	*Human Machine Interface*
I.MX53 SABRE	*Smart Application Blueprint for Rapid Engineering based on i.MX53*
IDE	*Integrated Development Environment*
KVM	*Kernel-based Virtual Machine*
LXC	*Linux Containers*
MBD	*Model-Based Design method*
MCAL	*Microcontroller Abstraction Layer*
MOST	*Media Oriented Systems Transport standard for multimedia network*
MUXA	*Multi-channel multiplexer console and debugging utility*
MM/T	*Multimedia/Telematics*
OEM	*Original Equipment Manufacturer*
OpenGL/OpenVG	*Open Graphics Library/Open Vector Graphics standard*
OSAL	*Operating System Abstraction Layer*
OSEK	*Open Systems and their Interfaces for Electronics in Motor Vehicles*
PCP	*Priority Ceiling Protocol*
PDUR	*AUTOSAR Protocol-Data-Unit Router*
POSIX	*Portable Operating System Interface standard*
PSP	*Platform Support Package providing hardware platform dependent functions*
PSSW	*PikeOS System Software module*
RFS	*ROM File-System for PikeOS ROM image*
RTE	*AUTOSAR Run-Time Environment*
RTOS	*Real-Time Operating System*
SchM	*Basic Software Layer Scheduler Module*
SWC	*AUTOSAR Software Component*
TCB	*Trusted Computing Base of a computer system*
VFB	*Virtual Function Bus*
VMIT	*Virtual Machine Initialization Table for system partition configuration*
VMWARE/VMM	*Virtual Machine Monitor or Hypervisor*
V2V	*Vehicle-to-Vehicle communication systems*

1 Introduction

There has been a tremendous increase in the volume of automotive vehicles as well as new functionalities introduced in them in the last two decades. Also, there has been a significant demand from public transport governing agencies to develop functionalities that meet the legal, social and environmental requirements like increased convenience, safety and control, advanced driver assistance, eco-friendly sustainability, etc. [1, 2]. As a result, modern automotive systems have at least 50 Electronic Control Units (ECU) for small and midsized cars and more than 80 ECUs in highend luxury cars [3]. These ECUs in turn are interconnected by different communication networks and implement diverse functionalities. The current automotive embedded systems host both control functions and infotainment applications that are running on separate platforms like AUTomotive Open System Architecture (AUTOSAR) and general purpose operating system (Android, Linux, etc.) respectively [1, 4]. AUTOSAR is a standardized automotive software architecture that enables integration of automotive control and convenience functional modules. The operating system (OS) module in AUTOSAR architecture was based on the specification of a real-time OS known as OSEK and was designed particularly for development of automotive control applications [5, 6]. General purpose OS such as Linux, Android, etc. are used in the automotive embedded systems to support non-real-time infotainment and vehicular user applications. In current automotive systems, there is a growing need for merging and consolidating these heterogeneous functionalities in a common ECU platform since it is crucial to be scalable from the economic point of view and be cost effective with respect to energy efficiency, system complexity and development costs. AUTOSAR provides perfect support for the classical automotive control and real-time functionality; however, it fails to provide support for integrating infotainment and user applications [5]. In the same way, General Purpose Operating Systems (GPOS) are not inherently designed to support the real-time specifications and hence it cannot support automotive control functions. These limitations in AUTOSAR and GPOS are the main barrier towards the prospect of a consolidated ECU integration by a common platform.

In this thesis work, we look at the integration problem of consolidating these heterogeneous ECU systems, running on top of AUTOSAR and various GPOS. In this work, we consider only Linux as general purpose OS (GPOS) since it is predominantly (almost 80%) used for non-real-time user applications in automotive systems [7]. We propose how this integration problem could be solved by developing a common, consolidated ECU platform utilizing virtualization technology. Embedded virtualization solutions (*Hypervisor*) provide capabilities for dynamic partitioning of limited embedded system resources among the multiple execution environments and architectures [8]. We propose a pilot implementation of a multipurpose ECU system to solve the integration problem, and we show how AUTOSAR and Linux applications could be concurrently executed on a shared hardware platform using a virtualization solution while supporting automotive requirements such as less communication delay, fast boot up time, etc. The proposed solution facilitates a simpler system design with efficient processor (CPU) time sharing by using a flexible runtime scheduler, exploiting parallelism by partitioning mechanism and thus opens the door for improved performance of future automotive systems that have complex functionalities. This report presents the design, implementation of the pilot implementation and its results along

with the analysis of the results. This report will serve as a good case study, and it can provide valuable insights for a more comprehensive research analysis, required to address the key challenges explored here, further. It also briefly discusses the necessary technology enhancements that are required for full-scale implementation and deployment of multipurpose ECU systems using virtualization technology in the automotive domain in the near future.

1.1 Vehicular Embedded Systems

In the last two decades, complex, distributed electronic control units (ECUs) have completely replaced the mechanical or hydraulic components used in the automotive vehicles due to rapid progressions in the field of embedded systems. These electronic components offer several advantages over their predecessor mechanical components in terms of fuel consumption, controllability, cost reduction and reusability [1, 9]. The typical examples of ECUs used in the vehicles are engine management systems, body and brake control, digital instrument clusters, infotainment head unit, Internet connectivity box and convenience control modules, etc.

1.1.1 Increasing Cost and Development Complexity

There is an intense competition among vehicle manufacturers to offer more advanced and innovative electronic features and it has been their key differentiator or unique selling point. Almost 60 to 90% of the new components in a modern luxury vehicles fall into the Electrical and Electronics (E/E) systems category. The deployment of every new complex functionality in turn leads to the introduction of an ECU component in the automotive system [1, 10]. As a result, the E/E system design in the modern vehicles has evolved from few isolated monolithic ECUs to a complex, distributed, ECU system architecture with more than 50 computational nodes interconnected by in-vehicle networks like CAN, MOST, etc. [10]. It has led to increasing system complexity, high hardware, software development and maintenance costs for automotive manufacturers.

1.1.2 Growing Need for Integration of Heterogeneous ECU Systems

Apart from classical automotive functionalities, there are also other ECUs in the cars controlling in-vehicle infotainment systems like audio and rear-end video entertainment, navigation assistance and internet connectivity solutions, etc. [10]. According to research[3,9], future cars will require the critical active safety, control and driver assistance systems to interact with the intelligent navigation systems, internet connectivity and car-to-car cooperative communication systems to understand the real-time traffic information, road and weather conditions, etc. Hence, the infotainment and connectivity services should be seamlessly integrated with classical automotive control functions to provide new advanced features for automobiles. There are two use cases listed below that illustrates the requirement for interaction between these diverse functionality ECU systems in the next generation automotive systems.

(i) Cooperative Adaptive Cruise Control

The current implementation of cruise control system is based on the information obtained from onboard sensors and offline stored global positioning system (GPS) databases. However, future car generations require cooperative adaptive cruise control**.** It can take smart decisions for automatic braking and speed stabilization based on the information obtained from vehicle-to-vehicle (V2V) cooperative communication

systems and intelligent transportation connectivity solutions [9]. It can take into account real world traffic information such as congestions, crash situations as well as environment conditions. It can be extended for the collision avoidance systems as well [9]. To implement such intelligent adaptive systems in cars, the active safety, anti-lock braking system (ABS) and electronic stability control ECUs needs to interact frequently with the connectivity and telematics ECUs forming a tightly integrated closed loop system.

(ii) Connected Car and Autonomous Car

Ongoing research efforts are working towards the implementation of commercialized driverless and connected cars. In these cars, the infotainment and connectivity systems need to form a closed feedback loop with the core automotive functionality ECUs like active safety, chassis control, advanced driver assistance, etc. Figure 1.1 shows the smart, connected cars that operate with artificial intelligence algorithms that in turn require efficient input information from connectivity systems, cloud networks and vehicle infrastructure systems [9].

There are many other examples similar to the above two that demonstrates the growing need to implement the convergence of the real-time automotive functionality as well as the non-critical infotainment and vehicular user applications.

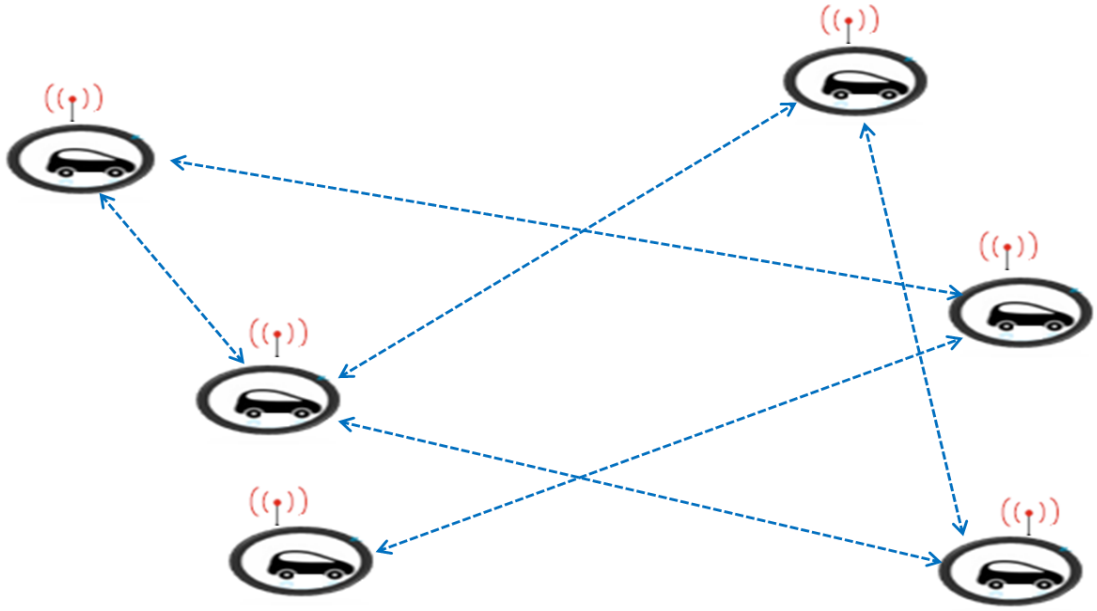


Figure 1.1 Connected Cars using wireless technology.

1.2 Motivation for consolidated platform

The best way to combat the integration and scalability issue of ECUs, as described above, in vehicular systems is by consolidation of ECUs of diverse functionality using a shared platform to integrate the infotainment services and the automotive control functions. Then the current vehicular system architecture containing more than 50 distributed ECU nodes will be replaced by few centralized ECUs running manifold vehicular applications in parallel on top of the modern, powerful multicore processors [11]. The consolidated platform is also crucial with respect to sustainable development

and society. The scalability issue of ECUs in automotive systems will lead to high economic costs to humans. Further increased power consumption due to the proliferation of ECUs will decrease the energy efficiency and emission control of automotive systems that in turn will impact the environment. Hence, it is crucial to resolve this proliferation issue. In the same way, as explained above, the solution to the integration problem will pave the way for developing advanced safety and driver assistance systems that can minimize human errors, avoid critical accidents and in turn save human lives.

1.3 Context of the research problem

The main impediment to the system consolidation of automotive control and infotainment ECUs is the lack of common OS platform for hosting these diverse functionalities in a single hardware system.

1.3.1 Automotive control applications and AUTOSAR standard

In order to counter the increasing challenges due to development complexity of the Electric/Electronic(E/E) systems, Original equipment manufacturers(OEMs), suppliers and tool developers have jointly developed the Automotive Open System Architecture (AUTOSAR) standard. It provides a common software infrastructure as well as standardized development platform for industry collaboration and partnership [4, 32]. AUTOSAR architecture defines various layered modules that are realized by various OEMs using different implementations. Software and hardware products following AUTOSAR standard can be easily integrated together and it finally improves the flexibility and scalability of the automotive E/E systems design and development process [32]. AUTOSAR architecture provides perfect support for the classical automotive functionality, that are inherently real-time applications with requirements for safety-critical timing constraints, high reliability and carefully defined specifications [5]. It can be achieved only by using a real-time operating system (RTOS) [1, 5] with real-time scheduling algorithm and task management. For the OS module, AUTOSAR architecture reuses the open systems and their interfaces for electronics in motor vehicles (OSEK) real-time OS specifications designed by the automobile industry consortium for the development of automotive ECU systems [6].

1.3.2 Infotainment applications and Linux

Apart from classical automotive functionalities, there are also other ECUs in the cars controlling in-vehicle infotainment and connectivity systems. Automotive manufacturers need to support seamless integration of portable consumer electronic (CE) devices and user applications in vehicular systems [10]. These infotainment and telematics applications use different general purpose OS platforms like Android, Linux, Windows, etc. Open source Linux OS is more commonly used for infotainment applications since it is cost efficient, flexible and easily portable to support various audio and graphics drivers [7].

1.3.2.1 Gaps in AUTOSAR standard and Linux

AUTOSAR standard mainly supports merging of automotive control and convenience function modules running on individual ECUs, in a common single-processor platform, and it is accomplished using the AUTOSAR software component (SWC)

implementation [32]. However, current AUTOSAR specifications do not provide support for infotainment and connectivity applications. Even though, it is extremely flexible and has a highly modular architecture, it lacks a flexible non-real-time scheduler to run these non-critical user applications [3]. Apart from this critical gap, AUTOSAR standard has the following limitations for supporting infotainment and telematics applications [5, 18].

a) No mechanisms to enable high-resolution graphics. Also, it does not support audio and video drivers.
b) The communications stacks used in Internet connectivity solutions (IP stack, Bluetooth, LTE, etc.) cannot be run on top of AUTOSAR architecture.
c) It cannot support algorithms that need to process large amount of data like image and video processing, etc.

Currently, there seems to be no intended plans or efforts by the AUTOSAR consortium to address the above challenges. Due to this limitation, now the infotainment, telematics and Internet connectivity applications run on top of various general purpose OS platforms like Android, Linux, Windows, etc. depending upon the OEMs and Tier1 suppliers. However, general purpose OS do not have real-time properties inherently and cannot be used for high critical automotive control applications. Therefore, GPOS also cannot be used for the consolidated platform, either. Therefore, it is important to find a consolidation strategy for the integration of these diverse functionality applications in a common, shared hardware platform. Since Linux is the most widely used (almost 80%) GPOS for automotive infotainment systems [7], we focus on Linux as the GPOS and thereby consider integrating Linux and AUTOSAR together on a single hardware platform in this thesis work.

1.4 AUTOSAR and Linux integration

In the modern vehicular systems, the core automotive control and infotainment applications have conflicting set of resource, architectural and performance requirements, and they are based on a different set of standards like AUTOSAR and Linux [3]. For instance, AUTOSAR architecture often integrates safety critical control applications like brake control system, diagnostic system, etc. that cannot be hosted by Linux OS. Also, AUTOSAR tasks require strict timing guarantees and need to be scheduled by a real-time task scheduler. In contrast, the infotainment and non-critical vehicular user applications hosted on top of Linux OS are nondeterministic and have soft or non-real-time tasks. The infotainment applications like digital dashboard and electronic instrument clusters have complex graphical user interfaces that would consume more resources for a considerable duration of time and thus require more computing power. Infotainment applications are implemented based on the specifications from consumer electronics domain and hence they deploy large, insecure code base and rich, open source APIs. However, automotive control applications utilize secure and minimal code base and are based on standard automotive specifications. The consolidation strategy for heterogeneous ECU systems should focus on providing support for these diverse set of requirements [3].

1.5 Technical goals and their challenges

In the present automotive systems, core automotive control and infotainment applications are implemented using AUTOSAR framework and Linux OS respectively, and they are executed on separate electronic control unit (ECU) systems. They interact with each other remotely using the vehicular communication networks for the transfer of data signals. There is an increasing need to implement the integration of these heterogeneous functionalities in a single ECU platform, for supporting the future car technologies and address the ECU proliferation and complexity issues in the current E/E system architecture in automotive vehicles. The designers of AUTOSAR and Linux have not explicitly considered the integration of these two systems and hence AUTOSAR and Linux cannot solve this integration problem by themselves. This thesis work considers this pressing, practical problem as a starting point for the problem definition.

Given the existing AUTOSAR and Linux implementations, one needs to find feasible strategy for consolidation of these heterogeneous systems and their diverse functionalities in a single, hardware platform and then implement sharing of the hardware resources and ensure strict separation between these functionalities [8, 13]. This thesis aims to explore several consolidation strategies and then finally design and implement a pilot model of multipurpose ECU system executing automotive control (AUTOSAR) and infotainment (Linux) applications in a single target platform. AUTOSAR architecture stack and Linux OS should be ported and executed on this common platform. The pilot implementation of the multipurpose ECU system should have the following capabilities to execute the AUTOSAR and Linux applications simultaneously on the same embedded system platform.

a) A flexible task scheduling mechanism that should give high priority to critical real-time automotive tasks while still providing best-effort to soft or non-real-time tasks.

b) Resource sharing and separation mechanism between the AUTOSAR and Linux application. Both applications should share the limited embedded system resources like memory and I/O drivers. At the same time, there should be a strict isolation and separation between the two applications i.e., applications should coexist, and a mal functioning application in one partition should not affect the application in the other partition.

c) Inter OS communication mechanism between AUTOSAR and Linux application to interact and transfer data and signals.

The pilot implementation should be subjected to measurement tests, and the measurement results should be evaluated to assess if the pilot implementation can demonstrate an equivalent or better performance when compared to current state of the art automotive systems.

The main challenge in this work was that we had to analyze and apply the theoretical concepts of virtualization and real-time systems practically in the realm of the automotive field and then design and implement a consolidated automotive ECU that is advanced than the current state of the art ECU system to solve the integration problem. The pilot implementation of multipurpose ECU system hosts both real-time automotive

and non-real-time infotainment applications. Automotive control applications have strict timing constraints and should execute within their specified deadlines. Infotainment applications need not respond quickly and can tolerate delay since they have soft or non-real-time requirements. We need to ensure that automotive application can finish all of its real-time tasks even in worst case scenarios. Also non-real-time infotainment application should not preempt the real-time automotive application and the infotainment application should still get a minimal proportion of the CPU time to be able to provide best-effort response. The challenge here was to study the complex scheduling mechanism of the common OS platform and then define a suitable scheduling model for efficiently allocating the CPU time slots between these diverse functional modules according to the requirements and the type of applications in the system. This required us to have significant knowledge in the task-scheduling techniques and fundamental concepts of virtualization. Likewise, we should design the resource partitioning mechanism for the common platform such that automotive control and infotainment applications should be spatially isolated from each other even though they are hosted on the same hardware platform and share the system resources. We need to ensure optimal memory foot-print and minimize the shared device access conflicts during runtime so that the multi-purpose ECU system can startup quickly. This necessitates us to estimate the memory requirements of automotive control and infotainment applications beforehand and the peripheral devices should be statically allocated among the partitions during configuration phase itself depending on their usage by the applications.

The inter-system communication between the applications should be designed such that it is localized in the same hardware platform and we need to ensure that it has better or equivalent transmission speed as that of the CAN communication networks used in the current state of the art automotive system. Also, we need to port the diverse OS such as AUTOSAR architecture that support real-time control applications and Linux OS that hosts non-real-time infotainment applications on the same hardware platform without the necessity for substantial modifications. In order to realize the above goals, it required us to have significant insights regarding AUTOSAR system architecture, Virtualization OS concepts, etc. The proposed design and implementation of multi-purpose ECU will help to achieve breakthroughs in the vehicular ECU development process and design of E/E system architecture in future vehicles. After the implementation of multi-purpose ECU system, we had to determine suitable tests to measure the nonfunctional parameters of the prototype model and then analyze those results with the related existing models or the state of the art system to demonstrate that our proposed prototype could be accepted for further analysis and applicability in future vehicular systems.

This thesis work requires us to have good understanding of knowledge and application of technologies from several domains such as

a) Real-time systems, for understanding the time-scheduling mechanism of the hypervisor for scheduling tasks with real-time and non-real-time requirements in AUTOSAR and Linux application.
b) Fault-tolerance, for ensuring isolation between the two OS partitions.
c) Operating system architecture and concepts, for understanding the virtualization techniques and porting of the AUTOSAR architecture and Linux OS on top of the virtualization layer.

d) Good knowledge of the state of the art automotive embedded systems to identify the requirements and suitable reference applications that could be deployed in the prototype model of multi-purpose ECU system.

Also in this thesis work, we appropriately apply the existing concepts of the virtualization, which has already been used in other domains such as telecom, avionics, etc. into the automotive field and thereby propose a solution to the integration problem of core automotive and infotainment functionalities in vehicular systems.

1.6 Proposed problem solution, our contribution and key findings

Our thesis work is a case study and analysis about integrating core automotive applications, hosted on top of AUTOSAR framework and infotainment applications, hosted on top of Linux OS in a single, consolidated ECU platform. We implement a pilot model of multipurpose ECU, that would replace the current state of the art remotely connected ECUs, communicating using controller area network (CAN) buses, in vehicular embedded systems. Also, the communication between these diverse functionalities is localized in the same hardware in our proposed pilot model. In this report, we discuss the significant challenges in the implementation and evaluation of proposed pilot implementation to solve the above Integration problem.

This thesis work presents our proposed solution about how a scaled-down version of the multipurpose ECU system integrating mixed-critical (real-time and non-real-time) applications could be designed and implemented using virtualization technology. The prototype model was evaluated and then our results were critically analyzed by comparing them with other related or current state of the art systems whichever is applicable. In the Discussion section, we discuss and analyze our chosen design options such as time-scheduling, inter-system communication, etc. Also we discuss other design-alternatives that could be considered for the enhancement of our proposed prototype model. Further in section 9.3, we provide a brief overview about selected possible extensions or research-avenues that could be explored in future based on our thesis-work.

a) We explore several consolidation strategies for the design of the consolidated ECU system such as Linux containers (LXC) technology in combination with real-time Linux (RTLinux) patches [53], microkernel-based AUTOSAR architecture [35,36], etc. to solve the integration problem. Finally we motivate and propose to choose virtualization based system architecture (*Hypervisor*) [8,23,24] as a suitable consolidation strategy for the pilot implementation of multipurpose ECU system executing AUTOSAR and Linux applications concurrently in a shared hardware platform . The various consolidation strategies are explained in Section 5.

b) We set the requirements for the pilot implementation and based on that we motivate a suitable virtualization solution, target hardware and reference applications in the context of prototyping a multipurpose ECU. For virtualization solution, we analyzed several hypervisor products [55,58,59], finally choose COQOS[12] hypervisor solution and we motivate the rationale behind the selection of COQOS for our proposed prototype model. The requirements of the multipurpose ECU system and rationale for selection of hypervisor (COQOS) were presented in Section 5.

c) In order to facilitate the evaluation of the proposed prototype model, this report explains how following technologies provided by the hypervisor were studied and applied in the context of designing and developing our pilot implementation of multipurpose ECU system.

   1) Time partitioning algorithm, that incorporates both time-driven and preemptive priority-based scheduling, is used for simultaneously executing strict real-time (AUTOSAR), and non-real-time (Linux) processes [26, 27].

   2) Resource partitioning mechanism is utilized for creating logical partition containers for AUTOSAR and Linux applications and then allocates the embedded system resources between them. It also ensures strict separation between the two partitions.

   3) Inter-partition communication using sampling ports, to transfer the data and signals between the AUTOSAR and Linux partitions [26].

The report will present the theory of the above techniques in Section 4. Then the application of these technologies in the context of our pilot implementation is presented in the sections 6 and 7.

During our thesis work while designing the task scheduling-table using the PikeOS time partitioning mechanism, we observed that it is important to reduce the jitter and interference from other events during the execution of real-time application in order to guarantee determinism for real-time tasks. We determined that real-time tasks should be allocated spare time slots to complete their execution at worst-case scenarios to offset the extra processing overhead introduced by the virtualization layer as well as the system monitor or watchdog application. PikeOS microkernel provides dynamic scheduling mechanism that incorporates both time-driven and preemptive priority-based scheduling and also it enables the flexible re-utilization of unused time slots of real-time tasks by the soft or non-real-time tasks. We found this time-scheduling mechanism as a perfect fit for mixed-critical system with real-time and non-real-time applications. During resource partitioning, we found that the mechanisms for shared device access introduce more delay since the tasks need to pass through additional virtualization layer to access the devices at run-time and hence we statically allocated the devices between the partitions during configuration phase itself. We found that inter-OS communication can achieve a faster rate of transmission rate when compared to the current state of the art CAN communication networks. It eliminates the necessity for a complex communication protocol and reduces the traffic load in the external communication bus since the communication is localized on the same hardware platform.

d) In this pilot model, AUTOSAR stack is ported on top of the virtualization solution. The COQOS hypervisor is based on PikeOS microkernel and modifications were required in AUTOSAR OS abstraction layer in order to port the AUTOSAR stack on top of the PikeOS microkernel rather than standard OSEK OS and these modifications were presented in the sections 6 and 7.

In the AUTOSAR architecture, the BSW modules and SWC are dependent on the underlying operating system for scheduling and execution of the tasks and the applications. During this thesis work, we learnt that the OSEK OS concepts like task and event model, interrupt processing and resource management functionality needs to be migrated to the corresponding mechanisms in the proprietary OS such as PikeOS for porting and executing the AUTOSAR stack on top of a proprietary OS. This porting process required OS wrapper functions for emulating the appropriate AUTOSAR OSEK interfaces to BSW, SWC and RTE modules using the API functions of the proprietary OS. We analyzed the AUTOSAR application in our prototype to understand the necessary OS functions utilized by it. The automotive application designed in our prototype uses a quite limited and simple AUTOSAR OS model. The PikeOS OS concepts were compared to the OSEK interface specifications and studied to analyze if PikeOS can provide support for these standardized OS interfaces. If there was a one-to-one match, then the wrapper needs to translate and create mapping between the interfaces of both OS. We observed that COQOS virtualization solution has good support for porting the basic AUTOSAR architecture. It provides POSIX APIs to implement the wrapper functions for the OS abstraction layer to emulate the standardized AUTOSAR OS interfaces and to port the standard AUTOSAR BSW, RTE and SWC on top of the PikeOS microkernel.

We faced some incompatibility issues while integrating the code files of the standard AUTOSAR RTE module and the proprietary OS (PikeOS) module and found that the integration of AUTOSAR modules from different vendors is not a straightforward process. We noticed that the code generator tools and build system are quite vendor specific solutions and are not interoperable with each other. We were able to solve this issue by synchronizing the OS configuration files and then customizing build process for the integration of RTE module from standard AUTOSAR stack and OS module from proprietary OS (PikeOS) and then generating the code files for them [41]. Our findings show that it is possible to port the standard AUTOSAR stack on top of a virtualization solution with slight modifications if the virtualization solution provides good support for the AUTOSAR OS concepts and scheduling mechanism.

e) This thesis work also presents the measurement results and analysis of the non-functional parameters of pilot implementation like boot time, signal communication delay, and isolation property in Section 8.

The research analysis of this work includes a preliminary evaluation that demonstrates a basic validation of our proposed prototype model of multipurpose ECU designed using virtualization solution. This preliminary evaluation serves as a proof of concept that it could be accepted for deployment in future automotive embedded systems in order to solve the integration problem of core automotive control and infotainment functionalities. We evaluated the non-functional parameters like boot time, signal transmission delay, and isolation property and then the results were studied by a comparative analysis of related existing models and current state of the art system. From this analysis, we could infer that our proposed implementation of multipurpose ECU system achieves better or equivalent performance when compared with the existing systems. Our prototype model shows that it is possible to integrate the automotive control application modules based on AUTOSAR architecture with infotainment

applications based on Linux OS, and they can share the system resources on the same ECU hardware platform using the hypervisor solution. Also, the intercommunication between automotive control function and infotainment application can be localized in the same ECU hardware platform instead of using complicated in-vehicle communication networks. Thus in our thesis work, we were able to demonstrate that virtualization technology can be utilized to implement the seamless integration of classical automotive control functionalities and advanced infotainment functionalities and thereby support the future car technologies in next generation vehicular systems.

1.7 Scope and limitations

This report provides a proof of concept (POC) pilot implementation of multipurpose ECU system to study the integration problem in general. In this thesis work, we consider the integration of a reference automotive control application based on AUTOSAR framework and a reference infotainment HMI application executing on top of Linux OS, since AUTOSAR architecture and Linux OS are the predominantly used OS platforms in automotive ECU systems. However, the concepts used in this thesis work can also be considered for the integration of automotive applications using other OS platforms such as Android, GENIVI software architecture, etc. Moreover, we consider a single processor platform rather than multicore processor for out pilot implementation. However, this pilot implementation could be easily migrated towards a multicore processor platform and this report also theoretically discusses the enhancements needed for deploying virtualization technology in multicore processors. Further, this work carries out only limited evaluation and measurement tests, and it does not consider a comprehensive evaluation and analysis of safety, performance and security aspects of the multipurpose ECU system. As the prototype model is only a scaled-down mocked up implementation, full-scale performance measurements are outside of the project scope. The safety and security aspects of multipurpose ECU systems have been already discussed by Marko Wolf and his coauthors in several research articles [13, 14, and 15].

1.8 Significance and Contribution of the pilot implementation

The proposed demo prototype of multipurpose ECU system, implemented using virtualization solution, will pave the way for flexible utilization of limited resources in future automotive systems as well as reduction in hardware costs associated with ECU proliferation and communication interface overhead between the separate ECUs connected remotely[3]. It also addresses the growing need for interaction between the diverse functionality ECUs in next generation car technologies and can simplify the E/E system architecture in modern vehicles. Our work also demonstrates how AUTOSAR architecture could be ported on top of a hypervisor layer. The concepts used in this pilot implementation can be considered as a case study analysis for the selection and integration of a hypervisor solution in future vehicular ECU development process. It could potentially lay down the foundation for integration of infotainment head units, driver assistance system and connectivity units in a single hardware platform using virtualization technology.

1.9 Sustainable development and ethical aspects

This thesis work proposes a prototype of consolidated ECU platform which addresses significant problems with respect to sustainable development and society. The scalability issue of ECUs in automotive systems will lead to high economic costs to the humans. Further increased power consumption due to the proliferation of ECUs will decrease the energy efficiency and emission control of automotive systems that in turn will impact the natural environment. Our proposed solution of consolidated ECU system will solve the above issues w.r.t society and environment. The consolidated ECU model resolves the proliferation issue and provides the solution to the integration problem. It will pave way for integration and interaction of heterogeneous modules like core automotive control and infotainment functions in a single, consolidated ECU system that will in turn help in developing advanced safety and driver assistance systems that can minimize human errors, avoid critical accidents and in turn save human lives. Thus the outcome of this thesis work, i.e., prototype of multipurpose ECU system indirectly contributes to welfare of the society, improves the quality of life of human-beings and helps to provide a safe, natural environment.

1.10 Outline of the thesis report

The thesis report contains nine sections and is structured as follows

- Section 1 gives an introduction to our thesis work, problem background, problem definition and our contribution, significance and scope of this thesis work.
- Section 2 provides an overview about the research method deployed in our thesis work and the different phases of our thesis work.
- Section 3 presents the literature research of related work and briefly explains the current state of the art system and their limitations.
- Section 4 provides the theoretical overview of virtualization technology, AUTOSAR framework and the technical concepts applied in our thesis work
- Section 5 provides the requirements for proposed pilot implementation, overview of different consolidation strategies and functional components of the proposed prototype system.
- Section 6 contains the technical details about the design of the sub-components of the proposed prototype model.
- Section 7 provides an overview of the implementation and then integration of the different sub-components into a single, consolidated ECU system using the development tools provided by hypervisor solution.

- Section 8 presents the evaluation criteria, measurement results along with their analysis for the parameters of pilot implementation.

- Section 9 presents the reflections about our thesis work and selected potential extensions related to our thesis work that could be explored in future. It also consists of summary and conclusions of our thesis work.

2 Research Methods

A research method is a systematic and scientific way to solve a research problem on a particular topic. The research method for this thesis work, i.e., design, development and analysis, is based on Design Science Research Methodology for Information Systems Research proposed by Peffers et al. [16]. Derived from this Design Science Research Method, we propose to solve the integration problem in current automotive systems by constructing an artefact of consolidated multipurpose ECU system executing heterogeneous applications, real-time automotive control and infotainment application, simultaneously on a single hardware platform using a commercial virtualization solution. We decide to realize and implement this multipurpose ECU system by the prototyping method.

2.1 A case for prototyping processes

In this work, we carry out technical, applied research that focuses on proposing a practical design and implementation strategy for a multipurpose ECU system. The proposed pilot system with further improvements could be applied in a real world context in near future to address the gaps in existing state of the art systems and to provide a solution for a pressing practical problem i.e., integration problem in current automotive embedded systems. Also, we aim to derive further knowledge from the development and evaluation of the proposed artifact from an engineering perspective. Taking all these into consideration, we decide to accept prototyping as the main method for this thesis work instead of other methods like theoretical case study, formal proofs, survey, etc. Further from an engineering point of view, a tentative prototype model that could be built faster provides invaluable opportunities to get better understanding of the requirements, explore the design options, and experiment with our proposed strategy and implementation. Prototyping method helps to acquire useful insight about the diverse, complex aspects of the research problem and hence in our thesis work, we choose to deploy Proof of concept (POC) prototyping [42] that can be used to build a scale down, mocked up version with a potential design approach. This prototype model can simulate certain fundamental aspects of a complete engineering system and then helps to gain insight about the development and implementation process of the real multipurpose ECU system. The observation of the system and primary evaluation can be carried out under controlled conditions to analyze the functional behavior of the proposed model. Also, the limitations and necessary enhancements required to overcome them can be studied [42]. The research method for this thesis work was based upon Design Science Research Method, and prototyping process [16, 42], and it consists of several phases as shown in the Figure 2.1 below.

a) ***Identify and define the research problem:*** The research problem for the thesis was formulated and defined with the stakeholders (MECEL AB).This thesis work proposes design and implementation strategy for the prototype model of a consolidated, multipurpose ECU system using virtualization technology. The objectives and framework for the thesis work were set, and the context of the study was limited to the consolidation of automotive control and infotainment applications based on AUTOSAR architecture and Linux OS respectively.

b) ***Problem Assessment and Build on existing knowledge:*** Literature research on related topic was carried out to gain knowledge about the prior research work done and also to understand the gaps in AUTOSAR architecture and the existing design and implementation strategy of related prototype models.

c) ***Exploratory Analysis of Design Alternatives and Selection:*** The requirements are laid down for the proposed prototype model to understand the problem more clearly. Several design strategies for the proposed prototype model were analyzed against the specified requirements and then finally a consolidation strategy using virtualization solution is chosen for the design and implementation of the prototype model. In this work, the knowledge contribution is done using exaptation technique, i.e., nontrivial application of known solution to a new problem [17]. The virtualization is a proven concept already used in software infrastructure consolidation and other embedded domains (telecom and avionics). We apply and extend this strategy for solving a new problem, i.e., integration problem in automotive systems. The functional components of the prototype model were chosen so that it can closely simulate the functionality of an automotive ECU system.

d) ***Design and Implementation of Prototype model:*** A high-level system architecture consisting of these functional components was proposed for the prototype model. Then the design for various components of the prototype model like porting AUTOSAR and Linux application on top of hypervisor, configuration of hypervisor technologies such as resource partitioning, time partitioning, inter-partition communication, etc. were carried out in accordance with the set requirements and problem objectives. Once the design of the components of the prototype model was completed, we proceed to the implementation of the functional components as per the design and then finally integration of the sub-components into a complete prototype system.

e) ***Basic Evaluation and Analysis of the Implemented Prototype model:*** Finally controlled measurement tests were carried out on the prototype model, and the evaluation results were analyzed and compared with the current state of the art and other related artefact systems. Then the limitations and necessary improvements required for a full-scale implementation of the proposed artifact were analyzed and discussed.

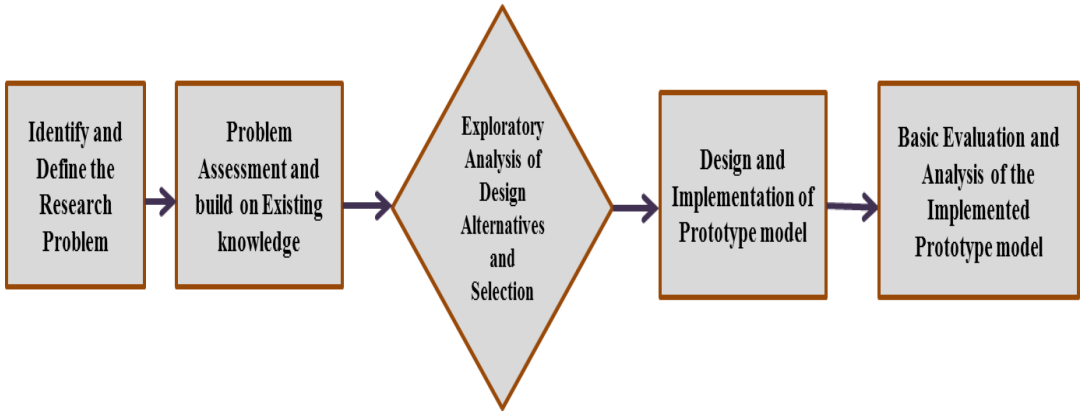


Figure 2.1 Research Method tailored from "Design science research [16]."

3 Related work and State of the art

Existing implementation deploys remotely connected ECU systems on separate processors for core automotive control and infotainment functionalities and they interact with each other using vehicular communication networks like CAN, Flexray, etc. The existing implementation has led to ECU proliferation that has resulted in a complex automotive E/E system architecture with more than 50 to 80 distributed ECU nodes. This has resulted in high hardware and software development costs, ECU to ECU communication network over-head, increased power consumption due to proliferation of ECU nodes and under-utilization of the additional CPU time of the modern processors.

Unlike the current state of the art systems, we report on a consolidated ECU system integrating core automotive control and infotainment functionalities executing on top of different operating systems on a common hardware platform using virtualization technology.

There are several benefits to this approach such as simplified system architecture with few consolidated ECU nodes that results in reduction of hardware and software development time and costs, integration and interaction of diverse functionalities in the same processor platform to support next-generation vehicular technologies, efficient utilization of peak processor potential by parallel execution of multiple applications and also decreases the need for external communication bus since the communication between the applications is localized in the same hardware platform. Further benefits are explained below in section 3.3.

In this section, we compare and discuss the significant aspects of this thesis work with respect to other related work and we show how our thesis work acts as an extension to the existing studies, analyses the challenges in solving the integration problem and the applicability of virtualization in automotive systems. Then it provides an overview of the current state of the art implementation for the interaction between core, automotive (AUTOSAR) and infotainment (Linux) ECU systems and the limitations in the current system architecture.

3.1 Literature research of related work

There has been several research works that seek to address the integration problem in automotive ECU systems. AUTOSAR consortium describes an approach to integrate basic AUTOSAR diagnostic applications in a Multimedia/Telematics (MM/T) ECU running on top of General purpose OS (GPOS) system like WinCE or QNX [18]. In this design, AUTOSAR Runtime Environment (RTE), Software components (SWCs) and specific Basic software (BSW) modules are integrated in a MM/T ECU, and they are made to believe that they are running in a complete AUTOSAR architecture by emulating the missing BSW modules, AUTOSAR OS module and the interfaces using the services provided by the underlying General purpose OS [18]. However, this simple integration method without using virtualization solution can be applicable only in limited situations. Most of the automotive applications have real-time requirements and in this architecture the GPOS does not differentiate between real-time and non-real-time tasks and also GPOS scheduler cannot be used for automotive applications that require critical timing guarantees. Likewise, it cannot be used for safety-critical functionalities

as the GPOS inherently does not have high safety integrity levels and hence crashes or errors in the underlying GPOS will impact the safety-critical automotive application running on top of the GPOS system. However, the concepts used in this study were quite interesting for us since in this thesis work, we take up a related approach when we try to port the AUTOSAR application on top of a proprietary OS platform (PikeOS microkernel).

The literature includes solutions that utilize hypervisor to solve the integration problem; implementation of automotive gateway based on an open source virtualization solution, KVM and Quest-V separation kernel for mixed criticality systems [19, 20]. The KVM hypervisor, unlike COQOS, is not based on microkernel implementation and can be run only on a processor that provides hardware support for virtualization. Quest-V is only a separation kernel and does not provide any virtualization services. It also depends on hardware-assisted virtualization for isolation and memory partitioning. This is a major limitation for OEMs as many commercial processors do not provide hardware extensions for virtualization. However, COQOS can be executed on most of the processor boards since it uses paravirtualization technique where guest OS like AUTOSAR and Linux needs to be modified and ported on top of PikeOS microkernel [12]. Also, KVM uses Linux as host OS and then utilizes real-time extensions and patches to support real-time tasks. It can support only soft real-time applications at best, and there will be some latency for hard-real-time applications. Also, security assurance level of Linux is relatively low for running safety-critical applications. However, PikeOS microkernel, chosen for our pilot implementation has been certified for compliance with stringent industrial safety and security standards and has already been deployed in mission-critical avionics projects [12, 26].

In the automotive gateway prototype, KVM implements a two-level hierarchical scheduling architecture and uses a combination of different task scheduling algorithms. It utilizes Sporadic Server (SS) or Constant Bandwidth Server (CBS) for the real-time tasks. The non-real-time tasks are allocated lower priority and can only be scheduled when the real-time tasks are idle using Completely Fair Scheduler (CFS) or Fixed Priority (FP) scheduler. Also, it uses virtual CPU (vCPU) framework, where multiple virtual CPUs (vCPUs) are created and assigned to the same physical processor core, and then it binds and allocates the tasks of the various guest OS (AUTOSAR and Linux) among these vCPUs [19]. In Quest-V architecture, the system is divided into several partitions, and each partition performs its local scheduling mechanism without using a complex global scheduler, and the CPU usage is effectively managed by a real-time kernel using an equivalent *vCPU* scheduling framework [20]. The time partitioning scheme used in PikeOS follows an analogous approach like KVM implementation where tasks in the various guest OS are allotted to logical time partitions and uses an advanced, flexible scheduler, combining time driven and priority-based scheduling, for executing real-time and non-real-time tasks [27]. The scheduling mechanism in the automotive gateway using KVM does not explain how it will handle the dynamic, system events like CAN communication interrupts, arrival of data from sensors, etc. PikeOS resolves the conflict between time driven and event driven real-time processes by using a novel approach where the scheduler needs to choose between two active time domains, foreground domain for time driven real-time tasks and background domain for event driven processes and non-real-time tasks [26, 27]. As a result, PikeOS partition scheduler uses a simplistic scheduling algorithm, and context switching overhead is fast

and bounded, i.e., O (1) since it needs to consider only two partitions during every context switch [26, 27]. However, the context switching overhead in vCPU framework will be high in the worst-case scenario, i.e., O(n**),** where n is the number of virtual CPUs. In the same way, the scheduling mechanism in automotive gateway using KVM and Quest-V architecture seems to be more complicated compared to PikeOS partitioning scheduler since they utilize a combination of several scheduling algorithms and hence they will have a large code base for the scheduling mechanism.

Quest-V architecture uses message passing based on shared memory regions for inter-partition communication [20]. The automotive gateway implementation also deploys shared memory regions in the KVM Linux kernel, that is accessed using a device driver interface by the application tasks in each guest OS [19]. In our implementation, we use simple sampling ports mechanism for transfer of data messages between the AUTOSAR and Linux applications. These sampling ports were based on socket based communication, and they never block or queue and provide much faster communication access when compared to the shared memory regions [29].

There seems to be very few research articles focusing on the challenges associated with porting of AUTOSAR stack on top of a virtualization abstraction layer. The automotive gateway implementation using KVM hypervisor uses only a customized AUTOSAR compliant OS, and it does not specify any details about the porting of this OS on top of the KVM hypervisor. The Quest-V architecture also mentions support for AUTOSAR stack but further details about the implementation are lacking in that report. Reinhardt, Kaule and Kucera specify the limitation of AUTOSAR in supporting mixed critical systems by itself independently and then outline a proposal for deploying AUTOSAR BSW in combination with an additional hypervisor layer for implementing a domain-oriented E/E design to solve the ECU scalability issues in automotive systems [21]. We take their analysis as a basis, and we focus on the challenge of porting AUTOSAR architecture on top of the hypervisor solution, PikeOS microkernel and then outline the necessary modifications required in the AUTOSAR OS abstraction layer and the standard AUTOSAR build process.

3.2 Current state of the art system

In the current automotive system architecture, infotainment and automotive control ECUs exists separately, and they are interacting with each other remotely using the vehicular communication networks for the transfer of data signals. Figure 3.1 depicts the example of a remotely connected ECU system. The current system does not address the ECU proliferation and the overall complexity issue of the vehicular interconnection wiring systems. Also, it hosts only a single control functionality in the modern, powerful processors and thus the excess CPU time is not effectively utilized [3, 13].

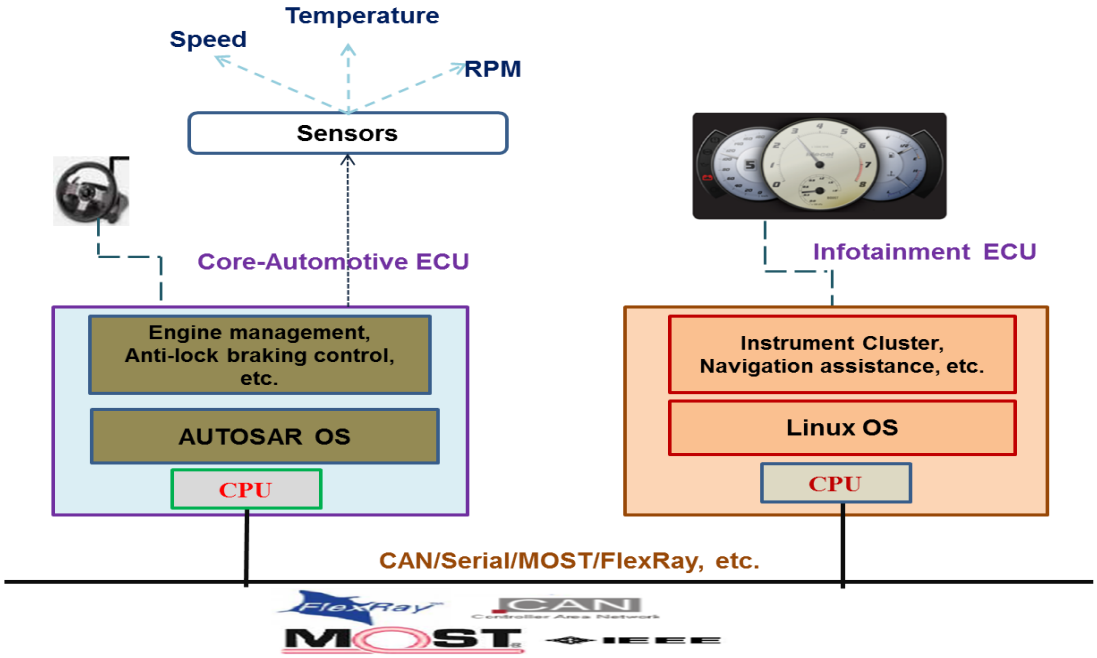


Figure 3.1 Remotely connected state of the art ECU systems.

3.3 Beyond the State of the art:

A consolidated ECU system can be implemented using the virtualization technology as shown in Figure 3.2, and it can support diverse OS on a common target platform to execute different functionality applications. This implementation offers several advantages and overcomes the existing limitations in the state of the art systems [24].

a) **Reduction of hardware units, development costs and system complexity**
The proposed multipurpose ECU systems will reduce the number of ECU units in cars considerably, and it will lead to simpler system architecture with few centralized, compact ECU nodes. This, in turn will reduce the development time and maintenance costs as it will be easier to carry out software development and upgrades [13].

b) **Minimizes ECU to ECU communication overhead and wiring system**
In the state of the art systems, the communication between the applications in different ECU nodes is handled by various communication protocols like CAN, Flexray, etc. running on complex interconnection wiring systems. It results in a communication overhead of nearly 100 to 1000 milliseconds for message transmission and a separate processor for handling these vehicular networks. In the proposed solution, the applications that require frequent interaction will be integrated in a single multipurpose ECU and also the communication will be localized in same processor platform [3].

c) **Efficient utilization of the processing capacity and system resources**
The proposed multipurpose ECU systems deployed using virtualization technology can consolidate and provide parallel execution of multiple functional applications on a single, powerful processor by exploiting their peak processor potential and facilitate efficient usage of shared hardware resources like memories and I/O peripheral devices.

d) **Increased flexibility and transferability of automotive software design**

The proposed ECU system using virtualization technology can easily support diverse operating systems and this enables developers to choose a flexible software platform for the hosting the automotive applications and also reuse and transfer the existing communication stacks and user applications directly into the automotive systems without need for significant adaptations [13].

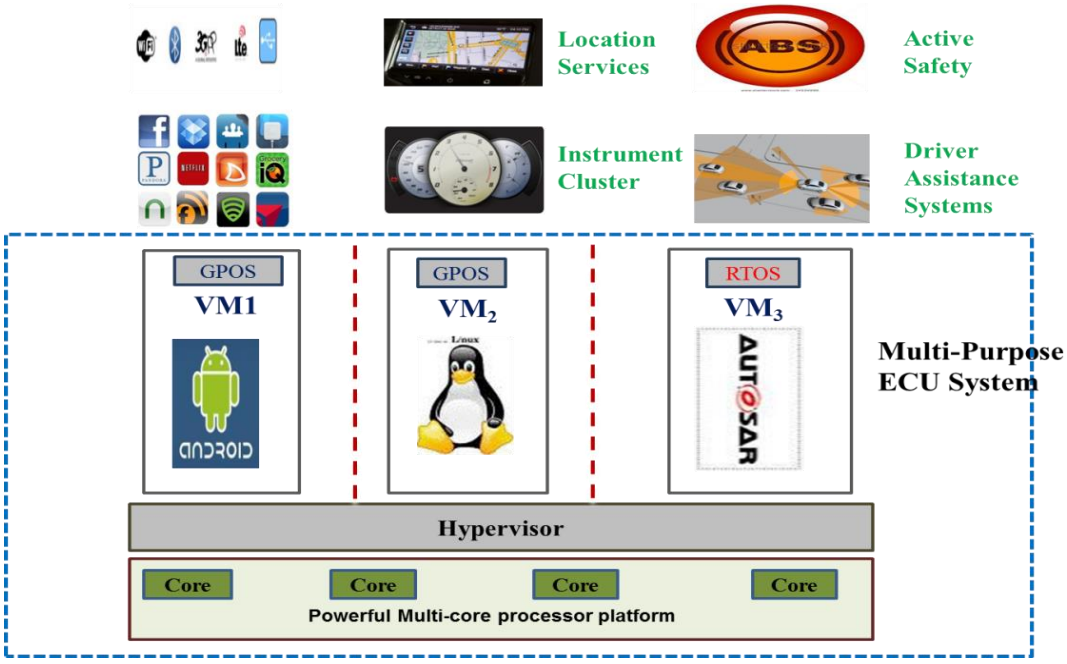


Figure 3.2 Example of a Consolidated ECU system using hypervisor solution supporting diverse functionality applications.

4 Background

This section introduces about virtualization solution; technical concepts applied in our pilot implementation such as time scheduling, resource partitioning of diverse operating systems (OS) and inter-partition communication mechanism using virtualization technology. It also describes how the hypervisor can support diverse operating systems and the tools used for implementation and integration of the components into a single system. Then this section highlights the fundamental concepts of AUTOSAR architecture modules related to this thesis work. The concepts explained in this section will help the reader clearly to understand the design and implementation of multipurpose ECU prototype that is described in the later sections.

4.1 Virtualization

Virtualization is loosely defined as a " framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation and many others [25]. "

Hypervisor is a software or firmware that realizes the virtualization technology in the computer or embedded systems [8]. Virtualization technologies like VMware, virtual desktop infrastructure, were initially used extensively in the server domain for hosting multiple logical server instances on a single server, and then gradually entered into the enterprise domain for software infrastructure consolidation. Then it was applied in the consumer space for simultaneously running multiple operating systems (OS) on a single PC [8, 23].

4.1.1 Deploying virtualization in embedded systems

Initially, embedded systems were mostly uni-application and uni-processor systems implementing simple and dedicated functionality. However, modern embedded systems have to implement several complex functionality domains and computationally demanding software applications with strict reliability and security. Thus, it can be seen that the embedded systems are reaching a degree of complexity and challenges comparable to that of the server enterprise space in terms of resource sharing and system consolidation. Most of the embedded systems use customizable system architectures that in turn are more suited for virtualization [22, 23].

Virtualization is seen as the next frontier by the embedded system architects and product development engineers to solve the decision challenges associated with increasing resource and application requirements of modern embedded systems. Also, the advent of the powerful multicore processors, demand for higher performance and lower power consumption has been a major driving force behind the deployment of virtualization in avionics, medical and industrial devices, next generation mobile platforms and automotive systems [11, 23]. The support for virtualization extensions by the processor chip providers has furthered the introduction of hypervisor technology in embedded systems [8]. The hypervisor solution offers the flexibility to host heterogeneous operating systems on the same multicore processor. It also implements robust reliability and fault containment mechanism to guarantee safe separation between mission critical, hard real-time applications and the general purpose, untrusted user applications.

4.1.2 Classification of virtualization technologies

Hypervisor can be classified into two types depending on how they are deployed and realized.

a) *Type1 or bare metal* hypervisor where the hypervisor executes directly on top of the hardware in the kernel mode and manages the hardware and different VMs[28]

b) *Type2 or OS level* hypervisor where the hypervisor runs as an application on top of a conventional OS environment and then manages the guest OS that runs at third level above the hardware [28].

Another type of classification is made depending on how the hypervisor virtualizes the guest operating systems.

a) "Full virtualization" technique that requires hardware processor support for virtualization and uses binary translation of OS instructions at runtime to trap into the VMM. There is no need for modification of the guest OS, and the guest OS is not aware that it is being virtualized. However, there will be a high-performance overhead due to the scanning and translation of OS binary instructions before execution [8, 23].

b) "ParaVirtualization" technique that requires modified guest OS to replace OS code instructions with hyper calls that communicate directly with the VMM. This results in high performance and low virtualization overhead and also guest OS are aware of being virtualized [23].

Type1 hypervisor with a thin hypervisor layer and small Trusted codebase (TCB) is suitable for virtualization in embedded systems as embedded systems have limited resources. Likewise, paravirtualization is more suited for embedded systems since it offers high performance and less code overhead and also the modification in guest OS is not complicated since the kernel code in embedded systems is usually small and is readily available to application developers.

4.2 COQOS Hypervisor solution

COQOS is a highly scalable software framework for implementing virtualization technology in the automotive domain [12]. It provides a flexible solution for automotive OEMs and suppliers to seamlessly integrate the infotainment head units, telematics and consumer electronics user applications along with the real-time AUTOSAR based automotive applications like advanced driver assistance systems(ADAS), chassis control, etc. on a single ECU system using a powerful processor[12]. In this thesis work, COQOS containing a lightweight PikeOS microkernel implementation [12, 26], that is comparable to a Type1 hypervisor, was deployed in our pilot implementation of Multipurpose ECU system. Paravirtualization was done for AUTOSAR and Linux guest OS, and they are ported on top of this hypervisor solution

It has been chosen since it provides the following features.

- ✓ Support for the AUTOSAR framework and paravirtualized Linux kernel versions.
- ✓ Support for various processor architectures and target platforms.

4.2.1 PikeOS Microkernel

The COQOS solution uses the SYSGO's PikeOS microkernel architecture [26]. The PikeOS microkernel has already been extensively deployed in mission-critical avionics projects, and it is certified for compliance with the stringent safety and security standards. It provides several state of the art functionalities and capabilities and this in itself is an extensive research topic [26, 27]. We briefly explain the technical aspects of PikeOS used and applied in the context of our pilot implementation like

i. Strict isolation and partitioning of resources using the resource partitioning technique.
ii. Time partitioning mechanism for flexible and efficient scheduling of real-time and non-real-time tasks.
iii. Inter-partition communication for implementing communication between the partitions for transfer of data signals.

4.2.2 Resource partitioning

Resource partitions are logical containers with statically defined set of hardware and software application resources with predefined access levels. PikeOS microkernel divides global kernel memory space into statically configurable subset of pools and then allocates it among the tasks in resource partitions at creation time. This mechanism ensures strict isolation between the partitions as a malfunctioning application in one resource partition can only exhaust the memory allocated to it and cannot access the memory pools of the other resource partitions. It is a simplistic solution since it maintains a minimal trusted code base and the only challenge is that the developer should predict the kernel memory requirements of the RTOS and GPOS beforehand. Likewise, RAM user memory resources are allocated to the partitions by PikeOS using predefined configurations. Address spaces that contain the corresponding RAM memory pages are created and allocated to each partition during system startup. Threads in a partition are always attached to an address space, and it can only access and manage the address space it owns [26, 29].

Additionally, access rights and communication rights are assigned to each thread to restrict and monitor their access to system call interface. Each access right enables a task to access a set of system calls, and the microkernel can check each system call of a task with their corresponding assigned abilities to ensure that it does not consume excess kernel resource or manipulate system settings. The abilities of a task are stored in the corresponding task descriptor table, and it cannot be changed or extended during the lifetime of the task. Also, a health monitor can be configured for error detection, fault handling and recovery of partitions and user applications executing inside a partition during runtime [29]. The above mechanisms guarantee strict separation and protection of system resources between the resource partitions.

4.2.3 Time partitioning

The pilot implementation of multipurpose ECU system will host both real-time AUTOSAR and non-real-time Linux applications. There should be a flexible mechanism to schedule both critical real-time and non-real-time tasks simultaneously. PikeOS implements an advanced time partitioning scheme that incorporates both time-driven and preemptive priority-based scheduling for running hard real-time tasks, as well as soft real-time and non-real-time tasks [26, 27]. In PikeOS, time partitioning can

be considered as a two-step process. PikeOS creates time partitions that are used for allocating a certain CPU time to each resource partition. The resource partitions in the PikeOS system are first assigned to the time partitions as shown in Figure 4.1.

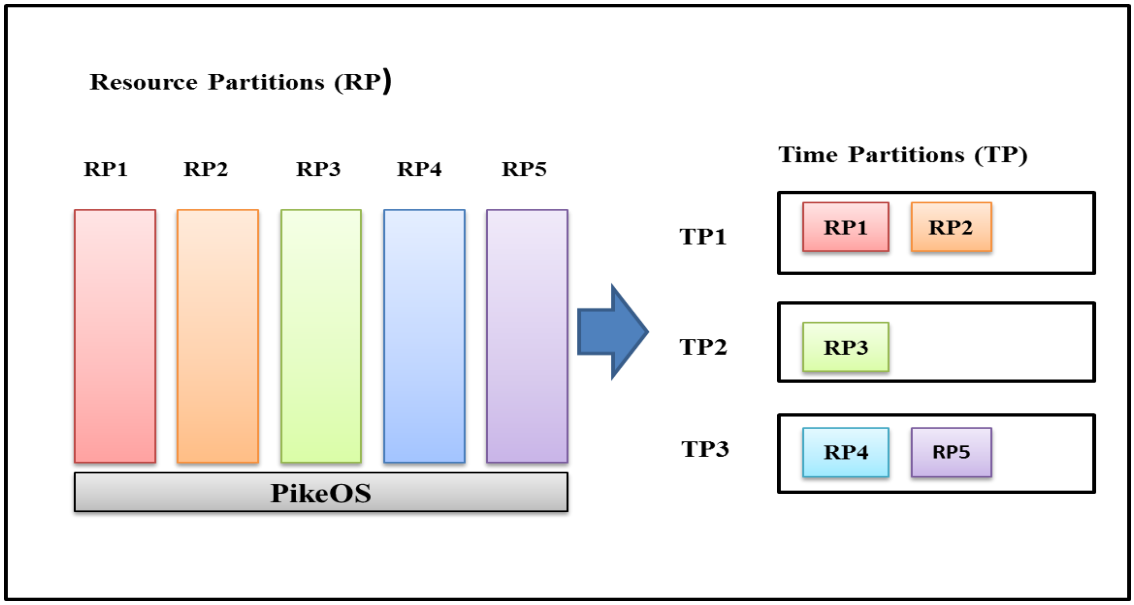


Figure 4.1 Assigning resource partitions to time partitions.

There is a major time frame of fixed duration that is cyclically repeated. This time frame is subdivided into several time slots of variable duration. The time partitions are allotted to the time slots in the major time frame as shown in Figure 4.2**.** Time driven scheduling mechanism schedules the time partitions in the major time frame. Within each time partition, the applications are scheduled by their priority according to preemptive priority based algorithm [26, 27].

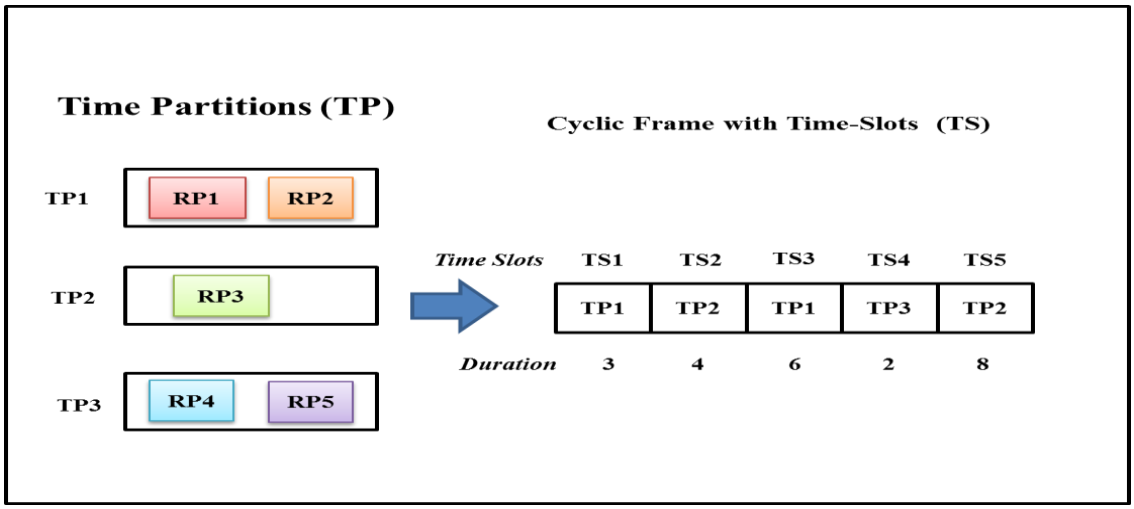


Figure 4.2 Assigning time partitions to time slots in the frame

This scheduling method is used in several RTOS. However the unique feature in PikeOS is that there are two time partitions that can be active at the same time; a special time partition with ID 0 (T_0), that is referred to as background time partition that is always active and another time partition, T_i chosen from $T_1, T_{2,..,} T_{N-1}$ (where N is the

highest time partition created) that acts as foreground time partition, switched cyclically by a time driven scheduler as shown in Figure 4.3.The real-time tasks are allocated in the time driven partitions $T_1, T_2, .., T_{N-1}$ and are given a medium to high-level priorities. The non-real-time tasks are allocated in the background time partition, T_0 and they are provided with a common low priority level. The safety-critical system services such as health monitor and watchdog are allocated to the background time partition, T_0 and they are assigned highest priority so that they can be activated instantly**.** This flexible scheduling mechanism ensures efficient load balancing among the different types of tasks. Strict, time drove real-time tasks get their fixed time slots that are calculated according to their worst-case execution times. If those tasks are completed earlier, then the remaining free time slices of their time slot will be assigned to the low priority, non-real-time tasks in the background partition using the PikeOS scheduler, thus effectively utilizing the excess computing time. Additionally PikeOS provides different scheduling schemes that can be created during system configuration, and it provides the ability to switch dynamically between these schemes during runtime [26, 27].

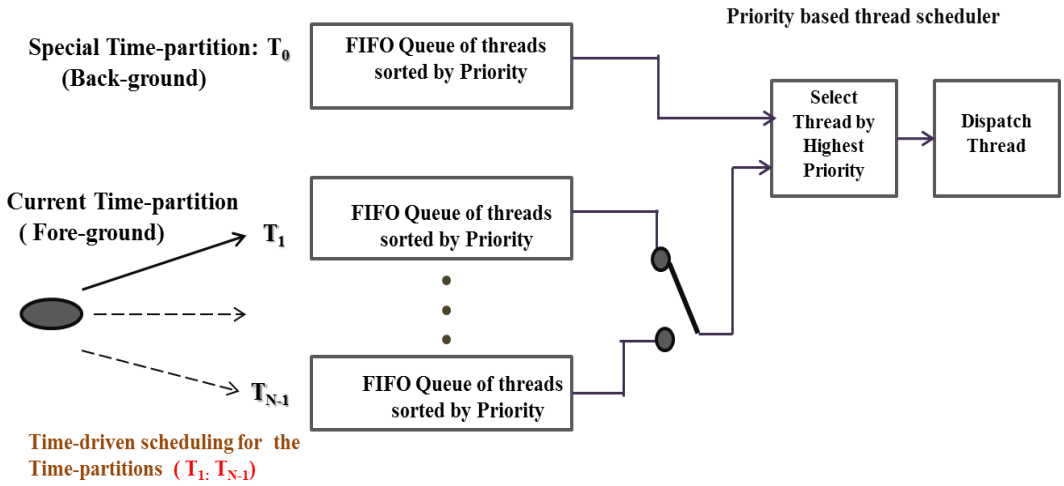


Figure 4.3 PikeOS Scheduler concept.

4.2.4 Inter-partition Communication

The resource partitions are strictly isolated and separated by using PikeOS micro kernel. However, it is also important for the applications in different resource partitions to communicate with each other in a controlled manner. PikeOS micro kernel provides three methods for implementing the inter-partition communication mechanisms [29].

(i) **Queuing ports**: It implements point to point communication channel between the resource partitions. Messages are stored in each channel up to a user defined maximum size in a First in First out (FIFO) queue. When reading or writing to the channel, the queue is blocked with a timeout value.

(ii) **Sampling ports**: This is equivalent to queuing ports; however, there is no queue to store the messages. The sampling ports consist of a message buffer of fixed size and when new messages arrive, old messages are rewritten. The messages can be updated periodically at a faster rate.

(iii) **Shared memory regions**: It is used for transferring large amounts of data. The shared memory segments are considered as shared file system and the read and write access to the shared memory is configured for each partition. Also, it needs application specific protocols for the synchronization and coordination of read, write access between the different partitions.

4.2.5 PikeOS support for virtualized AUTOSAR and Linux OS

In PikeOS, applications/user programs can be either executed directly on top of PikeOS microkernel interface or on top of resource partitions that host paravirtualized operating systems that are adapted to the PikeOS microkernel. These guest operating systems that can be run on top of PikeOS microkernel are known as personalities [26, 29]. In our prototype implementation, we use the following personalities provided by PikeOS, Embedded Linux (ELinOS) and Portable Operating System Interface (POSIX Real-time), for porting virtualized Linux and AUTOSAR OS respectively on top of PikeOS microkernel [26, 31].

a) ELinOS

ELinOS is an embedded Linux distribution provided by SYSGO AG [31]. ELinOS offers all the features used in industrial real-time applications and supports a broad range of BSPs, drivers and real-time hardware extensions. It provides the latest kernel version (2.6.35), for the Linux HMI demo application and is used to create the virtualized Linux OS that can be hosted in a partition on top of PikeOS microkernel.

b) Portable Operating System Interface (POSIX) Real-time

The PikeOS POSIX real-time personality is suitable for deployment of compact real-time systems in a partition and we use it to emulate the AUTOSAR OS since it provides good support for the concepts of AUTOSAR OS such as task model, events, scheduling, alarm mechanisms and resource management. POSIX Real-time personality provides POSIX API functions for the programming environment [29].

4.2.6 PikeOS Development and Integration Process

The PikeOS provides an Eclipse based IDE tool named as CODEO for cross compilation of software applications for the supported paravirtualized OS and then configuring them in separate partitions [30]. The applications can be compiled, linked and built into application binaries for the target hardware system using the PikeOS cross-compiler toolchain. The PikeOS system settings are configured using CODEO by the developer, and these parameters are automatically updated and stored in a configuration file in XML format known as VMIT.xml [30]. The specification of the system configuration settings such as partitions, their resource requirements, time scheduling parameters, inter-partition communication channels and ports and the applications that execute within the partitions are updated in this file. Finally, the partition configuration file (VMIT.xml), user application binaries and PikeOS binary objects (microkernel module and PikeOS system software) are assembled and built into a single binary file using a ROM image builder tool. This binary file is the PikeOS ROM image file that can be downloaded and initialized on an embedded hardware platform [30]. CODEO specifies the bootstrap mechanism for starting up the ROM image in the embedded target hardware.

4.3 AUTOSAR

AUTOSAR is an open and standardized software architecture framework jointly developed by vehicle manufacturers, major OEM suppliers and automotive tool developers in order to counter the increasing complexity of E/E systems in vehicles [4]. It provides a software infrastructure platform for collaboration, transferability, reusability and partnership between different stakeholders in the automotive domain while at the same time promoting competition on innovative features [32].

4.3.1 Modular, Layered Architecture

The AUTOSAR standardization provides a modular, layered ECU software architecture that enables model based development (MBD) of automotive embedded software functionality and also it helps to eliminate the boundaries between the diverse automotive functional domains. Furthermore, AUTOSAR provides software component framework that allows the description of automotive functionalities in terms of componentized software entities (SWC) that in turn helps to model the software for individual requirements. AUTOSAR provides a standard and simplified development infrastructure for ECUs, where the application software realizing automotive vehicle functions (SWC) are separated from the platform specific basic software modules (BSW), operating the ECU through a transparent middleware layer called Runtime Environment (RTE) as shown in the Figure 4.4. This design helps to develop the automotive software functionality without any knowledge about the corresponding ECUs [4, 32].

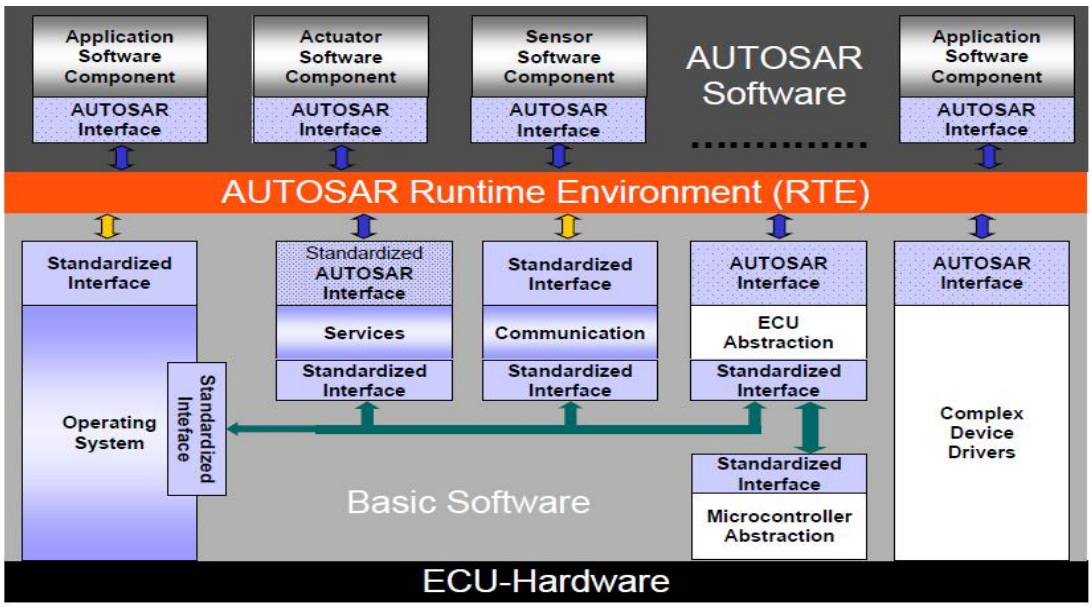


Figure 4.4 AUTOSAR Layered Architecture [32].

4.3.2 Basic software (BSW)

The Basic Software modules provide the infrastructural functionality on an ECU, and it contains standardized and ECU specific components. The Basic software modules are divided into sub layers as shown in the below Table 4.1 according to their functionality [32].

Services	Memory management, diagnostics, Watchdog, ECU state management, CRC, end to end protection and implements scheduler functions for BSW modules.
Communication	Vehicular network communication and management services
OSEK RTOS	Real-time scheduler, task and event management, alarms, resource allocation and interrupt handling services. Also handles assignment of cores to applications in multicore processor.
MCAL	Driver interfaces for I/O devices, communication, memory and microcontroller direct access and internal peripherals

Table 4.1 Overview of the list of functions offered by various BSW sub-layers

4.3.3 Micro Controller Abstraction Layer (MCAL)

MCAL contains the internal drivers for microcontroller, and it enables the upper layers of AUTOSAR stack to be independent of the hardware implementation [32]. In this thesis project, the CAN communication module in MCAL was utilized to receive and process the CAN frames containing the data values from a vehicular CAN network simulator tool, CANalyzer**.**

4.3.4 AUTOSAR Operating System

In order to maintain the backward compatibility with automotive legacy applications, the AUTOSAR OS is based upon standard OSEK OS specifications [6]. The essential features of AUTOSAR OS are [6]

i. It can be configured and scaled statically.
ii. Amenable to reasoning of real-time performance.
iii. Provides a priority based scheduling and protective functions at runtime.
iv. It can be hosted on the low-end controllers and without external resources.

4.3.4.1 AUTOSAR OS concepts

a) Task management

Tasks are schedulable entities of a software application that are executed by the scheduler. Each task can be in anyone of the states as explained below.

1. *Running*: The task is being executed, and CPU time is allocated to this task and in a uniprocessor platform, only one task can be in running state at any point in time.
2. *Ready*: The task has satisfied all the conditions for execution and waiting for scheduler to allocate the processor time to it.
3. *Waiting*: The task that is waiting for an event to occur to continue its execution.
4. *Suspended*: The task that has been terminated by the scheduler goes into this state until it is activated again by scheduler.

The tasks are allocated their priority statically during configuration phase so that scheduler can determine the precedence for the execution of tasks according to their priority [6].

b) Scheduler

It is a system application that determines the tasks that should be given access to the system resources i.e., processor, I/O devices, etc. OSEK OS uses event driven, fixed priority scheduling policy where the tasks are activated and deactivated by periodic timing or sporadic events and tasks are executed according to their allocated priority [6].

c) Event mechanism

AUTOSAR OS uses events for synchronization of tasks, i.e., state transition of tasks from running to waiting state and vice versa. Events can be expiry of a timer, reception of CAN messages, availability of a resource, notification of hardware interrupts, etc. [6].

d) Resource management

AUTOSAR OS module uses resource management for protection and coordination of shared resources such as memory, hardware devices, etc. for the different tasks. It uses OSEK priority ceiling protocol (PCP) to prevent deadlocks and priority inversion problem i.e., lower priority tasks indirectly preempting and delaying the execution of a higher priority task that is waiting for access to a shared resource [6]. According to this protocol all the shared resources are assigned static ceiling priority i.e., priority of highest priority task that will ever use that resource. Whenever a task acquires a resource, its priority will be temporarily raised to the ceiling priority of the resource, and it will be the only task that is in running state among all the tasks that share this resource and thus, deadlocks and priority inversion problems can never occur.

e) Alarms and Counters

The AUTOSAR mechanism provides alarms and counter mechanism for processing of recurring events, i.e., interrupts that occur at regular events, reception of periodic CAN frames, etc. A counter is equivalent to a timer. Each counter is associated with the alarm function that expires when the counter reaches a particular value and upon expiry of alarm, either the task is activated, or event is set for the task [6].

Apart from the above, AUTOSAR OS also carries out of processing of hardware and software interrupts by providing corresponding interrupt service routines (ISRs), mapping of runnable entities of the application to tasks, memory and timing protection, etc.

4.3.4.2 Operating System abstraction layer (OSAL)

In the AUTOSAR framework, standard OSEK OS specifications are used for the execution of AUTOSAR software components. If these software components are implemented to be run on top of a proprietary OS, then the OSEK OS interfaces should be emulated and provided by using an Operating System Abstraction Layer (OSAL) [5].

4.3.5 AUTOSAR Run Time Environment (RTE)

In the AUTOSAR framework, RTE is the middle layer that provides the application layer with the services from the BSW layer. All communication between SWCs, either inter ECU or intra ECU is accomplished by the RTE using the interfaces of Virtual Function Bus (VFB) i.e., mapping connectors and ports. It also takes care of the real-time scheduling of the SWCs [33]. The essential task of the RTE is to ensure that SWCs

are independent of the ECU on which they are mapped. As SWC's are dependent on the type of application, the RTE has to be tailored for the specific ECU in which it serves and the RTE will differentiate between different ECUs and the SWCs can remain same accomplishing the intended functionalities [33].

4.3.6 Software Component (SWC)

An AUTOSAR application consists of several SWCs interconnected by connectors [34]. The SWC is independent of the hardware infrastructure, and it encapsulates a part or the complete automotive functionality. SWC includes a formal description that contains details about the operations and data elements, communication properties, specific implementation, internal behavior (runnable entities, task and events), composition and required hardware resources [32]. The essential attribute of the SWC is that it should be atomic i.e., only one instance of a certain SWC can be present in a vehicle and that is assigned to one ECU [32].

4.3.7 Virtual Function Bus (VFB)

The virtual functional bus is the abstraction layer that comprises the interconnections and data exchanges between SWCs, other components and the system environment of the entire vehicle independent of any underlying hardware [34]. The functionality of the VFB is provided by ports, port interfaces and mapping connectors. Software components have the required port (Rport) as input and provided port (Pport) as output to interact and exchange the data elements with other software components. The ports should be associated with port interfaces, that define the services or data that is transmitted between the ports of different SWCs [34]. Commonly used port type interfaces supported by the VFB are client-server and sender-receiver interfaces.

In our application, we use the sender-receiver communication mechanism between the SWC and the BSW COM module via the RTE. This mechanism provides asynchronous (non-blocking) communication where a sender distributes information to several receivers, or one receiver gets information from several senders and there is no data or control flow response mechanism. The ports between the software components, that need to interact and communicate with each other, are linked using assembly connectors as shown in Figure 4.5 below [34].

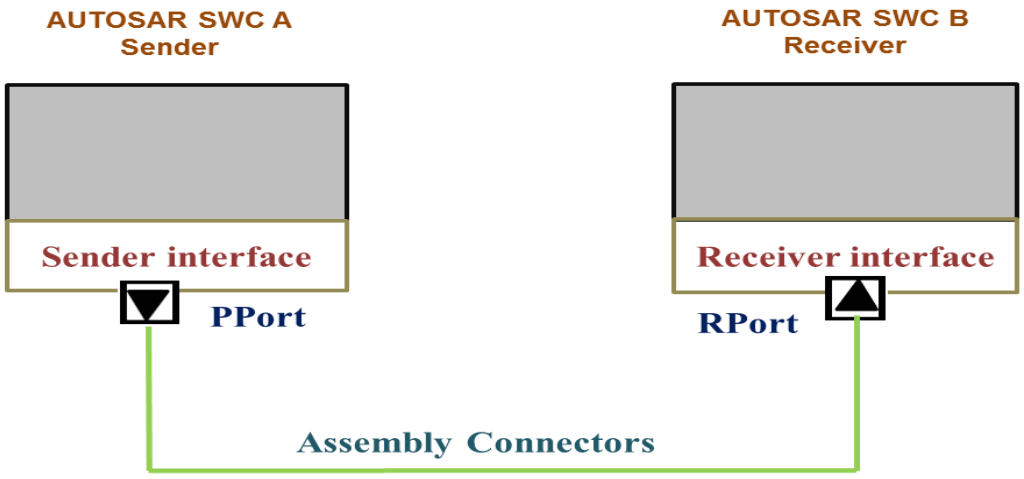


Figure 4.5 Sender Receiver communication between 2 AUTOSAR SWCs.

5 Requirements and Consolidation Strategies

In this thesis work, we propose to design a pilot implementation of multipurpose ECU in order to study and solve the integration problem of automotive control and infotainment applications. This section briefly explains the requirements for our pilot implementation, overview of alternate consolidation strategies and then we motivate the reasons behind the chosen consolidation strategy using virtualization solution. We highlight the functional components in our prototype model, particularly virtualization solution and the rationale for their selection.

5.1 Requirements for Multipurpose ECU system

System design always begins with setting requirements for the system in order to determine the design choices. The main challenge for the pilot implementation of multipurpose ECU system is to find a suitable common OS platform or system architecture for concurrently running the real-time automotive control and non-real-time infotainment applications on the same target hardware, sharing the system resources like CPU, kernel and RAM memory, I/O peripheral devices, etc. Also, the common OS platform should provide an Inter OS communication mechanism so that these diverse applications running on a shared hardware platform should be able to exchange data signals between them.

The common OS platform for the multipurpose ECU should have following functional requirements.

(i) Flexible scheduling mechanism

Automotive control applications must respond quickly and hence they need immediate access to resources. They have strict timing constraints and should execute within their specified deadlines. Infotainment applications need not respond quickly and can tolerate delay since they have soft or non-real-time requirements. The common OS platform should provide a dynamic scheduling mechanism for efficiently allocating the CPU time between the diverse functional modules such as real-time tasks in the AUTOSAR application and non-real-time tasks in the Linux application, by giving high priority to the real-time tasks while still providing best-effort to non-real-time processes.

(ii) Capability for partitioning of system resources and isolation between the partitions

The common OS platform should provide capabilities for sharing and partitioning of embedded system resources like kernel, user memory and I/O devices between the AUTOSAR and Linux applications. There should be strict separation and isolation between them; the different applications should safely coexist and execute in the common platform, and one faulty application should not affect the other applications in the system. In our pilot model, the execution of AUTOSAR and Linux applications should be strictly isolated from each other even though they are hosted on the same hardware platform and share the resources like CPU time, I/O peripheral devices, etc.

(iii)Communication between the applications

The common OS platform should provide a simple and faster way for the transfer of message signals between AUTOSAR and Linux applications, i.e., the communication between the diverse applications should be localized in the same hardware platform without going through the external CAN communication bus.

(iv) Support for AUTOSAR architecture and general purpose OS

The consolidated ECU system should have abilities to host diverse OS such as AUTOSAR architecture (real-time control applications) and Linux (non-real-time infotainment applications) without the need for substantial modifications.

Apart from the above, the common OS platform should be able to meet the below nonfunctional requirements that are measurable.

(i) Meet automotive requirements like fast boot time and minimum communication delay

The consolidated ECU system should satisfy the automotive real-time requirements like fast bootup time, minimum signal communication delay, etc. For instance, automotive real-time applications should be started instantly within 5 to 10 seconds as soon as the car is powered on. The boot time of the consolidated ECU system and signal communication delay between AUTOSAR and Linux application should have better or comparable performance when compared to the current state of the art, remotely connected ECU systems. This is important since automotive companies compete mainly on cost/performance ratio.

Based on above requirements, we analyze different design alternatives and select a suitable consolidation strategy for our prototype model.

5.2 Survey of consolidation strategies

We present various design strategies for the common system architecture of proposed prototype model of multipurpose ECU and then we motivate the reasons behind our chosen design strategy.

5.2.1 Linux containers and Real-time Linux patches

Linux containers (LXC) technology provides a lightweight virtualization for running multiple isolated Linux systems known as containers on a single Linux kernel and root file system (RFS), and it provides isolation of resources between the containers using kernel control groups [53]. Thus, Linux container technology is equivalent to a Type2 Hypervisor or operating system level virtualization implementation. This mechanism does not virtualize the hardware platform; it only provides separation between the different systems. OS level instructions are executed directly on the CPU and hence do not require dynamic translation as done in Paravirtualization. RTLinux, a hard real-time variant of Linux, that contains a small real-time kernel that coexists with the typical Linux kernel, can be used to support the real-time control applications. We consider using LXC in combination with RTLinux as a design alternative for the multipurpose ECU system. However, there were several shortcomings in this approach.

- ✓ Linux containers will support only Linux OS; AUTOSAR architecture needs to be ported on top of the Linux OS, and it requires substantial effort.

- ✓ Also, RTLinux contains only a simple priority-based scheduler for handling interrupts in general and hence it cannot completely replace the functionality of the OSEK RTOS. So, we need to modify the kernel to include feasible scheduling algorithms as per the requirements of specific automotive applications. This is quite complicated and hence, it cannot adequately support hard real-time applications.

- ✓ Additionally there is less domain isolation and separation since Linux kernel inherently does not have a strong security level assurance, and it cannot provide a high protection level like virtualization technology.

- ✓ Critical automotive applications cannot be booted up instantly as it needs to wait until the startup of Linux kernel and that itself takes almost 5 seconds.

5.2.2 Microkernel based AUTOSAR architecture

There are several literature research articles proposing an AUTOSAR compatible micro kernel [35, 36]. In this approach, minimum core and safety relevant functions like task creation, scheduling and memory access are carried out by the AUTOSAR OS. It is the only software module that executes in the privileged mode of the CPU, thus creating a thin microkernel layer with minimal codebase. To support heterogeneous operating systems, the task scheduling in microkernel-based AUTOSAR model needs to be modified to include a flexible scheduling algorithm for executing both real-time and non-real-time tasks [36]. Then AUTOSAR microkernel can be considered as a virtualization platform and other OS such as Linux/Android can be virtualized by adapting it to the VFB layer of AUTOSAR architecture as shown in the Figure 5.1. Also, AUTOSAR 4.0 provides memory partitioning and protection mechanism [5]. However, still developing an AUTOSAR compatible microkernel requires major architectural changes as explained above and significant technical breakthroughs would be needed for meeting the safety-critical requirements as well as providing support to infotainment and connectivity applications that use high-resolution graphics and complex communication stacks. This approach requires considerable effort and time (maybe years) and hence it is not a viable solution for our pilot implementation that we intend to develop within few months.

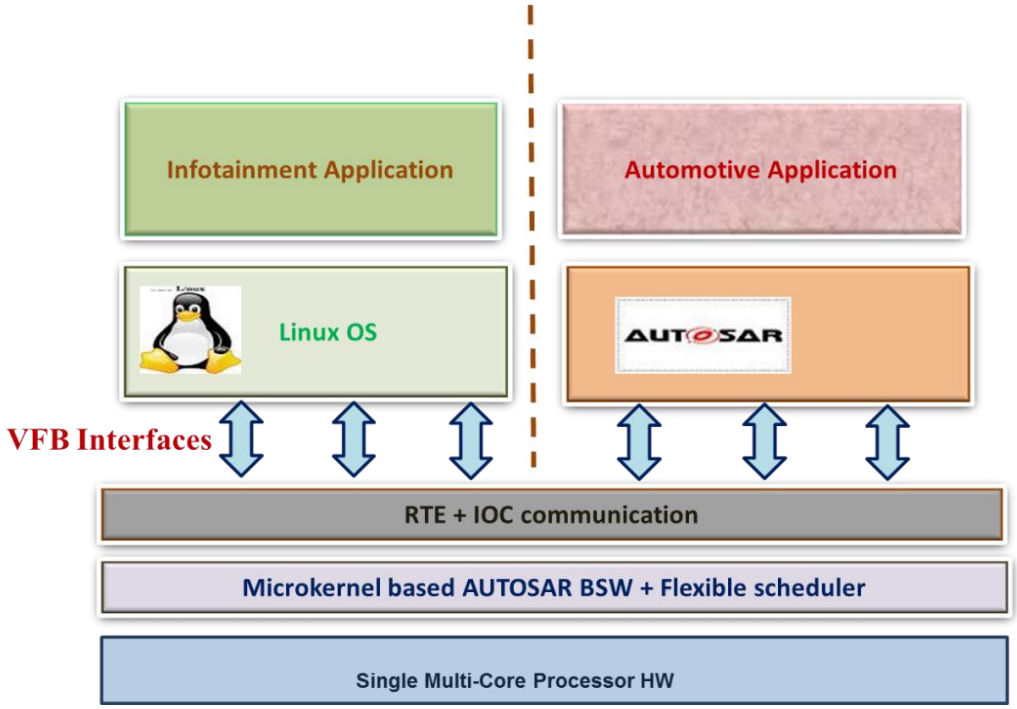


Figure 5.1 Microkernel based AUTOSAR architecture for running real-time and non-real-time applications.

5.2.3 Virtualization based system architecture

Virtualization solution has already been deployed in the embedded systems domain in avionics, telecom switches and next generation mobile devices for system consolidation and isolation of diverse functionalities [23]. At present, automotive embedded systems also implement several distributed functionality domains and computationally demanding software applications. Most of the automotive systems have started using general purpose software with extensive insecure code base and rich open source APIs [10]. Hypervisor technology provides a scalable architecture for multiple functionality domains, eases software design since it offers a common abstraction layer for developers without the need for complex porting process of diverse OS and hence the existing user applications can be easily reused in automotive systems without significant modifications. It can efficiently tap the processing power of the state of the art processors to provide peak performance potential by parallel execution of applications [24]. Also, several, diverse applications in automotive system can share the hardware resources like memory and I/O peripheral devices between them using hypervisor solution. The chosen hypervisor technology meets all the requirements for the pilot implementation of multipurpose ECU as listed in Section 5.1. It can provide support for hosting several diverse operating systems and then ensure separation and isolation between them. The applications in the diverse OS can interact with each other using inter OS communication mechanism provided by hypervisor technology. Most of the hypervisor solutions inherently deploy advanced, dynamic scheduling mechanisms. Also, most modern processors have high processing speeds and capabilities, and thus, it can offset the small processing overhead due to the additional hypervisor abstraction layer. Hence, we decide to select the system design using hypervisor technology as the consolidation strategy in order to solve the integration problem in automotive embedded systems.

5.3 System settings of the prototype model using Hypervisor technology

The functional components of the proposed pilot implementation of the multipurpose ECU are

a) COQOS as the hypervisor solution
b) I.MX53 SABRE ARD as the target hardware platform
c) AUTOSAR 4.0 and Linux (kernel:2.6.35) as the two OS partitions
d) AUTOSAR control application developed using MECEL PICEA suite [38].
e) Linux HMI Instrument cluster application developed using MECEL POPULUS suite.

5.3.1 Rationale for selection of functional components of the prototype model

The pilot demo system containing reference implementations of automotive real-time application based on AUTOSAR architecture and non-real-time infotainment application based on Linux OS, executing concurrently on a common processor platform was designed to serve as the scaled down, prototype model for multipurpose ECU that could be deployed in future automotive systems. The main challenge is to design a simplistic prototype that could model the final engineering system and hence the functional components of the pilot model were selected and designed so that they could simulate the functionality of the real working ECU system as close as possible. Also, it is important to keep in mind that the selected functional components should

have good interdependency so that they can be integrated holistically into a complete system.

5.3.2 Hypervisor solution

The choice of the hypervisor solution was critical for the implementation of multipurpose ECU prototype since it should provide capabilities for flexible scheduling of diverse applications, partitioning and isolation of resources and support for porting Linux and AUTOSAR OS architectures. There was several open source as well as proprietary hypervisor solutions applicable for embedded systems domain like Kernel-based Virtual Machine (KVM), Xen, OKL4, Wind River Hypervisor and COQOS. We carried out a comparative analysis of the features of these hypervisors, as shown in Table 5.1, in order to select a suitable hypervisor for our proposed pilot implementation of multipurpose ECU hosting both Linux OS and AUTOSAR architecture.

5.3.2.1 Comparative Analysis of the Hypervisor solutions

There are several open source hypervisor solutions like KVM and Xen, and we considered them for our prototype implementation [55, 57].

KVM: This is an open source hypervisor, and it utilizes Linux as the host OS. The guest OS are ported on top of the Linux kernel and it requires hardware virtualization extensions [55]. The main disadvantage in this is that Linux kernel does not provide support for hard real-time applications and at best it can be used only for soft real-time applications. It is mainly used in server systems and general embedded devices [54, 57].

Xen: This is most commonly used open source Type1 hypervisor, and it can support both full virtualization as well as paravirtualization if there are no hardware virtualization extensions available in the processor platform [54]. Since it is open source code, developers need to implement suitable and specific scheduling algorithms as per the requirements of the target applications. It is mainly used in mobile platforms and general embedded devices [55, 56].

Even though, we can avoid high costs by using an open source hypervisor, there is limited information available about the security assurance levels of open source hypervisors and this is critical for automotive embedded applications. Both KVM and Xen hypervisors do not provide support for AUTOSAR architecture and hence it requires extensive effort to port it on top of the hypervisor abstraction layer for our pilot implementation.

There were several proprietary hypervisor solutions like Wind River, OKL4 Microvisor and COQOS, and we considered them as well for our prototype implementation of multipurpose ECU system.

Wind River Hypervisor: This is a Type1 embedded hypervisor implemented by Wind River, and it provides support for real-time behavior and capabilities required for high performance embedded applications [59]. It delivers a thin virtualization layer with minimal codebase. It implements reliable and safe partitioning and also the scheduling mechanism can be configured or modified according to the requirement of the target applications. Even though, it provides support for the various processor architectural models, there was no pluggable board support packages (BSPs) available for Freescale boards at the time of our thesis work. Wind River Hypervisor solution is used in a wide

array of embedded applications like aerospace, telecom equipment, consumer and medical embedded devices, mobile platforms as well as automotive systems [59].

OKL4 Microvisor: This is an open source, microkernel based hypervisor solution provided by OK Labs. It has good support for real-time systems, resource management and a minimal codebase. It is mainly used in mobile platforms and supports several guest OS such as Linux distribution, Android, Symbian and Windows. However, there is not much support available for AUTOSAR architecture and hence extensive effort is needed to port it on top of OKL4 Microvisor [58, 60].

Hypervisor	*Support for AUTOSAR and Linux*	*Advantages*	*Drawbacks*
KVM	➢ Linux distributions can be easily ported. ➢ Requires much effort for porting AUTOSAR.	➢ Open source. ➢ No license cost. Easier modifications and upgrades.	➢ Not much information about security assurance level. ➢ Linux cannot provide support for hard, real-time applications ➢ Requires hardware virtualization extensions and hence cannot support all processor platforms
Xen	➢ Good support for porting Linux. ➢ Requires much effort for porting AUTOSAR.	➢ Open source. ➢ No license cost. Easier modifications and upgrades.	➢ Not much information about security assurance level. ➢ Requires suitable scheduling algorithms as per the requirements of target applications.
Wind River	➢ Good support for both Linux and AUTOSAR.	➢ Good support for Real time systems, Resource partitioning and Minimal code base.	➢ License cost is very expensive in the context of academic thesis work. ➢ Additional cost is required for providing BSPs for Freescale boards.
OKL4 Microvisor	➢ Good support for Linux. ➢ Requires much effort for porting AUTOSAR.	➢ Good support for Real time systems, Resource partitioning and Minimal code base.	➢ Mostly used for mobile platform and is not deployed for automotive ECU systems.
COQOS	➢ Very good support for both AUTOSAR and Linux	➢ Good support for Real time systems, resource partitioning and Minimal Code base. ➢ Meets stringent security and safety certifications. ➢ Specifically designed for Automotive industry.	➢ License cost is there but it is less than Wind River.

Table 5.1 Comparative analysis of hypervisor solutions for our prototype model.

COQOS: Open synergy's COQOS product is based on the SYSGO's PikeOS microkernel. The virtualization layer is very thin and is only about 3k lines of code. It

also provides full support for Linux, Android and AUTOSAR framework. Also, the microkernel allows safe and reliable partitioning of the processor resources. It also fulfills the highest safety and security standards as the PikeOS microkernel has been extensively used in avionics and safety critical applications. It also provides a configurable communication bridge between the Linux and AUTOSAR OS. COQOS uses paravirtualization instead of hardware-assisted virtualizations and provides paravirtualized versions of Linux and Android guest OS. Another appealing factor is that COQOS has been specifically designed for the automotive ECU systems and for integration of AUTOSAR based automotive application with Linux/Android based infotainment application [12,26].

5.3.2.2 COQOS Hypervisor solution

Finally, OpenSynergy's COQOS hypervisor was selected as the suitable virtualization solution for the pilot implementation of multipurpose ECU [12]. The primary motive behind the selection of the COQOS hypervisor solution was because it meets all the requirements as specified in Section 5.1.

(i) **Resource Partitioning**: As explained in Section 4.2.2, COQOS provides a resource partitioning mechanism to allocate system resources for different OS partitions. AUTOSAR and Linux OS can be hosted in separate partitions on a single hardware platform and share the system resources using this mechanism.

(ii) **Time Partitioning Mechanism**: As explained in Section 4.2.3, COQOS provides time partitioning mechanism to allot CPU time slots for different OS partitions hosting real-time and non-real-time applications. AUTOSAR real-time and Linux non-real-time applications can access the CPU time using this flexible time partitioning scheme.

(iii) **Inter system Communication Mechanism**: As explained in Section 4.2.4, COQOS provide several communication methods to enable inter OS communication between the partitions, and one of those methods was utilized for transfer of data signals between AUTOSAR and Linux application in our prototype model.

(iv) COQOS is specially designed for the automotive domain, and it had good support for paravirtualization of AUTOSAR architecture and embedded Linux OS [12].

(v) It is based on PikeOS microkernel that provides a lightweight abstraction layer with minimal code base. It has been extensively used in safety-critical avionics and industrial applications. It can fulfill hard, real-time features and is certified for compliance with the stringent safety standards (IEC61508, EN 50128, etc.)[26].

5.3.3 Target hardware

Freescale i.MX53 SABRE ARD was selected as the target hardware for the pilot implementation [40]. It provides

a) High processing speeds up to 1 GHZ for hosting the additional hypervisor layer.

b) FlexCAN interface module for CAN communication.

c) Support for Linux OS and capabilities for executing accelerated high definition 2D/3D graphics with minimal CPU loading to support the infotainment applications.

As a result, the other functional components such as hypervisor, automotive and Linux reference applications can be seamlessly integrated on top of this hardware platform.

The reference applications for the pilot model were selected and designed so that they could closely simulate the functionality of a real working ECU.

5.3.4 Automotive Reference application using AUTOSAR Architecture

We developed an automotive control application using AUTOSAR 4.0 architecture. Vehicular real-time values, such as speed, revolutions per minute (RPM), and engine temperature are sent to this control application periodically by a CAN network simulator tool. The AUTOSAR application mainly consists of an application SWC, COM communication stack, BSW OS module and a generated RTE module.

(i) The COM communication stack (BSW modules such as CanDrv, CanIf, PDUR and COM) is configured to receive the CAN frame containing the payload data from the CAN simulator tool, encapsulate the payload data and then finally deliver it to SWC through RTE module.

(ii) The SWC with receiver ports and sender-receiver interfaces is created for periodically reading the data values from the BSW communication layers via RTE. Also, it consists of a C function known as runnable, that implements the actual business logic of ECU functionality. The SWC would perform some processing with these data values for the engine control functionality of automotive ECU.

(iii) In the RTE module, a periodic timing event necessary for scheduling of the runnable is created. RTE read functions that enable the SWC to access the BSW communication modules and fetch the appropriate data signals, are implemented [33].

(iv) The BSW OS module consists of an OS task that is associated with the RTE timing event. The SWC will utilize this OS task for the periodic scheduling of the runnable function. The BSW scheduler contains the schedule table for the activation of tasks in the AUTOSAR application [5].

The AUTOSAR application is configured using the Volcano VSx tools, and it utilizes the BSW, RTE software libraries and generator tools provided by MECEL Picea suite [38**]**. This automotive application simulates a typical control process functionality that could be deployed in an engine management ECU.

5.3.5 Infotainment Reference application using Linux OS

We used an instrument cluster HMI application, developed using MECEL Populus suite as the Linux reference application [39**]**. The Linux application gets the input data values periodically from the automotive control application and then updates the gauges and indications like speed, RPM, engine temperature, etc. during runtime on the HMI display unit. This is a state of the art GUI application that deploys accelerated 2D/3D graphics with high resolution and frame rates. It typically resembles the dashboard application that could be used in the automotive infotainment head unit.

6 Design of components of Prototype model

Based on the selected consolidation strategy using virtualization technology, we propose a pilot implementation of multipurpose ECU system. In this ECU system, automotive control and infotainment applications based on AUTOSAR architecture and Linux OS respectively, are implemented so that they can concurrently execute in a single hardware platform using the COQOS hypervisor solution. The proposed solution facilities consolidation of heterogeneous, distributed ECU nodes and aims to solve the integration problem in the current state of the art vehicular systems. The real-time AUTOSAR and non-real-time Linux application are separated by allocating them to two different partitions that have their own memory space, and the partitions share the I/O peripheral devices in the hardware and the CPU time [29]. Then we implement inter-partition communication channels between the partitions so that the applications can transfer data signals between them. The high-level system design of the proposed prototype model as shown in the Figure 6.1 below consists of three software functional components namely,

- ✓ Partition 1 named as AUTOSAR partition containing the automotive control application based on AUTOSAR architecture.
- ✓ Partition 2 named as Linux partition containing the Linux kernel, root file system and HMI application based on Linux OS.
- ✓ Hypervisor solution, that provides the virtualization technologies.

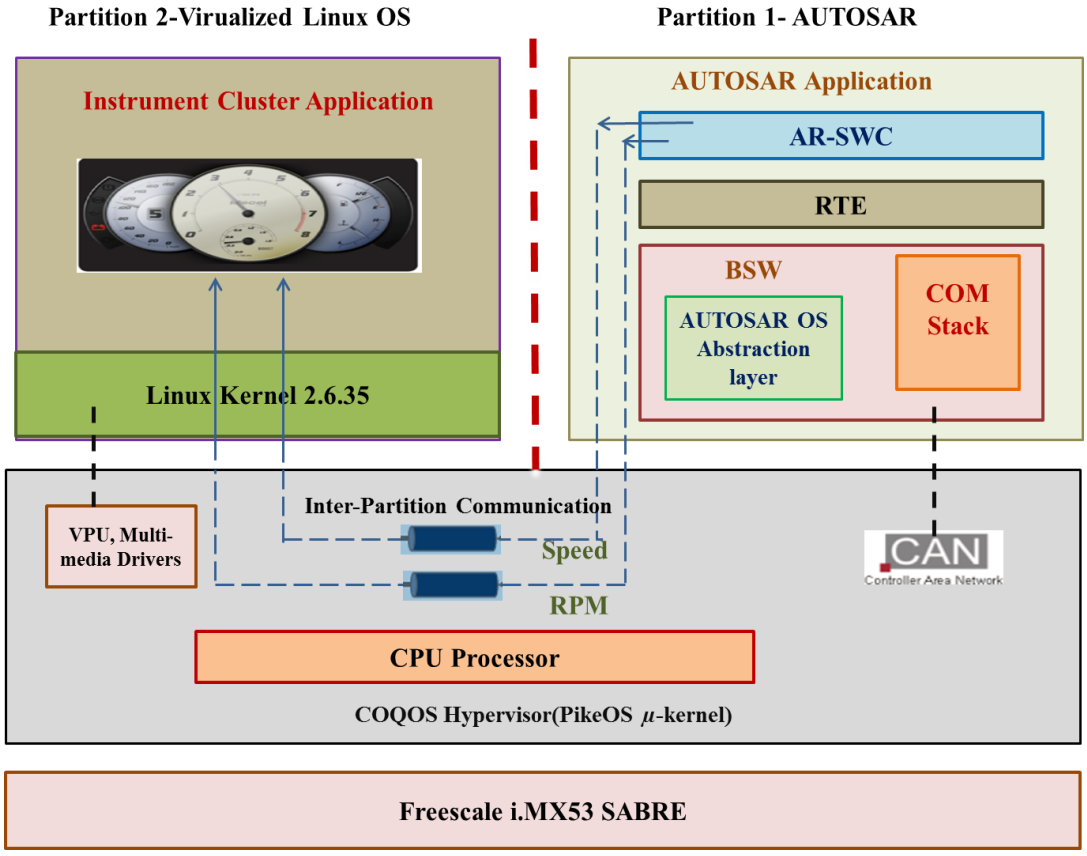


Figure 6.1 High-Level Design of the Prototype model.

Apart from the above, the prototype model also consists of the target hardware platform, I.MX53 SABRE ARD. However, the hardware platform is a commercial off-the-shelf (COTS) product that is supported by COQOS hypervisor and there is no particular design and implementation done in this hardware for our prototype implementation [12]. The three software functional components of the prototype model are designed such that they can enable parallel execution of AUTOSAR and Linux applications on the same hardware platform.

The main design objectives for the prototype model were

1. Porting of AUTOSAR application on top of hypervisor solution.

2. Porting of Linux application on top of hypervisor solution.

3. Enable parallel execution of AUTOSAR and Linux applications on the same hardware platform using the hypervisor solution. This involves
   a) Efficient processor (CPU) time sharing between the real-time AUTOSAR and non-real-time Linux partitions using PikeOS time partitioning mechanism.
   b) Sharing and strict separation of the hardware resources in target device (i.MX53 SABRE) between the two partitions using PikeOS resource partitioning mechanism.
   c) Communication interface to transfer the data signals in a simple and faster way between the partitions using PikeOS inter-partition communication method.

The design considerations, techniques and optimization for each of the functional components were carried out in accordance with the above objectives and to fulfill automotive requirements like quick startup time, less signal communication delay, strict isolation, etc. As in Design Science Research Method, we expose the basis of our design decisions and explain what design was done for each functional component and why it is being done [17].

6.1 Design of the AUTOSAR partition

The AUTOSAR partition consists of an automotive control application based on AUTOSAR framework, which could be ported on top of the hypervisor solution. The design considerations for the AUTOSAR partition are

a) Create a PikeOS compliant AUTOSAR architecture for the automotive application so that it could be integrated and executed on top of the PikeOS microkernel as shown in Figure 6.2.

b) Optimize the startup time of the AUTOSAR application to support the fast bootup requirement.

6.1.1 Design of PikeOS compliant AUTOSAR architecture

In the AUTOSAR architecture, the BSW modules and SWC are dependent on the underlying operating system for scheduling and execution of the tasks and applications [32]. Usually, the industry standard OSEK OS specification is used in the state of the art AUTOSAR architecture [6]. In this prototype design, we use the virtualization layer, PikeOS microkernel as the OS module to create a PikeOS compliant AUTOSAR architecture that could be integrated and executed on top of PikeOS microkernel as

shown in Figure 6.2 below. This involves two main modifications in the state of the art AUTOSAR framework.

a) As per AUTOSAR specifications, an OS abstraction layer (OSAL) need to be implemented that provides OS wrapper functions for emulating the appropriate AUTOSAR OSEK interfaces to BSW, SWC and RTE if a proprietary OS (PikeOS) is used instead of standard OSEK OS [5].

b) Integrate the state of the art AUTOSAR BSW, RTE and SWC with the OS module based on PikeOS and generate code files using AUTOSAR build process [41].

6.1.1.1 OS wrapper functions for emulation of OSEK OS interfaces and services

The OSEK OS concepts like task model, fixed priority based scheduling of tasks, event mechanism for synchronization of tasks, interrupt processing, resource management and alarms functionality[6] needs to be migrated to the corresponding mechanisms of PikeOS. AUTOSAR architecture implements these functions using standardized OS interfaces. First, we analyze our AUTOSAR application to understand the necessary OS functions utilized by it. When porting AUTOSAR RTE or BSW into propriety OS platforms, the third party OS need to reimplement these standardized OS interfaces using their own API functions to emulate OSEK OS [5]. The PikeOS mechanisms were compared to the OSEK interface specifications and studied to analyze if PikeOS can provide support for these standardized OS interfaces. If there is a one-to-one match, then the wrapper needs to translate and create mapping between the interfaces of both OS. The AUTOSAR application designed in our prototype uses a quite limited and simple AUTOSAR OS model.

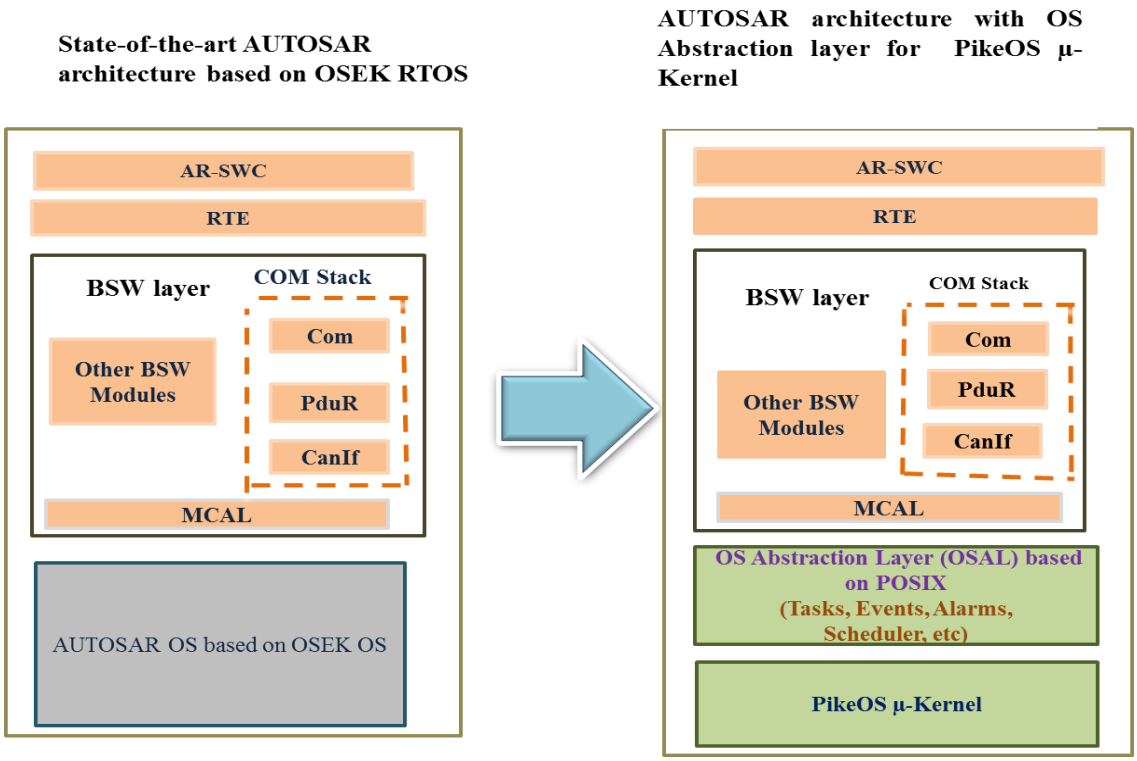


Figure 6.2 Migration to PikeOS compliant AUTOSAR architecture.

a) Task modeling:

The reference AUTOSAR application consists of a basic OS task model instead of an extended one for scheduling of runnable entities of the application. In basic task model, a task has three basic states: running, ready and suspended [6]. We need to provide the interfaces for state transition between these three states in our OSAL wrapper functions.

b) Event, counter and alarm mechanism:

In reference AUTOSAR application, the OS tasks are associated with periodic timing events for the activation and execution of the tasks by the scheduler. In AUTOSAR OS model, events are registered with specific alarm functions and these functions set or reset the events. These alarm functions have predefined timing counter values, and when the counter reaches the specified value, the alarm function is called that in turn triggers or resets the associated event [6]. So we require interfaces to increment counter value, set or cancel the alarm function, etc., in the OSAL wrapper functions.

c) Interrupt processing

In our reference application, CAN frames are sent from the simulator tool, and it in turn triggers the CAN hardware interrupt signals. This requires the OSEK interfaces to trap and process these CAN interrupt signals [6]. Therefore, we need corresponding interfaces in OSAL wrapper functions to enable, disable and handle the CAN interrupts.

d) Scheduling and resource management:

A scheduling service is required to allocate system resources to tasks. When it is called, the scheduler will search for any task that is in the ready state. The task in the ready state with the highest priority will get the system resources and will be scheduled to running state [6]. There should also be a mechanism for protection and coordinated access to common resources such as memory, I/O devices, etc. that are shared between different tasks to prevent deadlocks and priority inversion problem. This is done by OSEK priority ceiling protocol (PCP) that guarantees mutual exclusion for the common resources shared among different tasks [6]. We need corresponding interfaces in OSAL wrapper function to implement the scheduling and resource management mechanisms.

PikeOS provides several personalities to paravirtualize and emulate a guest OS on top of the PikeOS microkernel. We use the PikeOS POSIX real-time personality to emulate the standardized OSEK OS interfaces since it provides good support for real-time systems [26, 29]. The COQOS solution provides POSIX compliant APIs [12] and the OSAL wrapper function utilizes these POSIX API functions for translating and mapping the OSEK interfaces of AUTOSAR OS in PikeOS, and this is explained in the Implementation section.

e) Optimization in AUTOSAR application

PikeOS provides facilities only for initialization of partitions and scheduling of the applications inside the partition according to the time partitioning mechanism. However, the immediate execution and startup of the application depends upon the design of the application. AUTOSAR OS uses a standardized interface; StartOS to initialize the AUTOSAR operating system and application during system startup or reset [5, 6]. StartOS interface carries out the initialization steps as shown in the Figure 6.3, and it calls the StartupHook routine and the user can put all the OS dependent initialization code in this routine [6]. In our prototype system, the fast startup of the real-time

AUTOSAR application is an essential requirement for automotive ECU system, and this depends upon OSEK Startup interface (StartOS) [6].

PikeOS did not directly provide the StartOS interface function. Instead, it had an equivalent OS interface named as "START_AUTOSAR" that was used for starting the OS services. The Startup functionality was coded and implemented in the OS abstraction layer (OSAL) by using this PikeOS interface (START_AUTOSAR), as per the specifications provided by OSEK OS. The initialization procedure in StartupHook routine was customized so that it is not complicated and lengthy to execute. We placed limited and necessary initialization procedures in the StartupHook function to ensure quick startup of AUTOSAR application and at the same time we ensured that AUTOSAR application could be loaded and executed correctly.

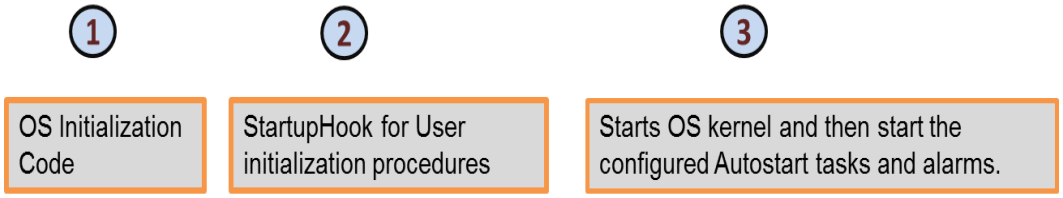


Figure 6.3 StartOS functionality for AUTOSAR application startup.

6.1.2 Integration of standard AUTOSAR stack and PikeOS OS module

During the porting of AUTOSAR stack on top of PikeOS microkernel, we integrate the standard BSW, RTE and SWC modules from MECEL AUTOSAR stack along with proprietary OS module from COQOS. As explained above, the OS module is based on PikeOS microkernel and contains the OS abstraction layer developed by POSIX API functions. However, the integration of AUTOSAR modules from different vendors is not a straightforward process as the code generator tools and build system are quite vendor specific solutions and are not interoperable with each other. This is a typical system integration problem that arises while linking of modules from different subsystems. We provided a strategy to customize the standard build process in order to integrate the standard AUTOSAR stack along with an OS module based on other Propriety OS such as PikeOS.

The standard build process can be used to generate C source and header files containing the application specific configuration parameters for BSW, MCAL modules and SWC using the configuration files (Arxml) and SWC coding implementation since SWC and BSW modules are not directly dependent on the OS module in AUTOSAR architecture [41]. However in AUTOSAR framework, RTE and OS modules are interdependent since the SWC and BSW scheduler uses the RTE layer to access OS module for task scheduling, resource allocation and communication [5]. The problem is that RTE module is from standard stack provided by MECEL and OS module is based on a proprietary OS like PikeOS and is provided by COQOS. The OS header files (Os.h, Os_Cfg.h, MemMap.h and Ioc.h) contain the configuration information of OS such as task, application, alarms, resources, etc. as well as the declarations and definitions of the OS interface functions. These header files are referenced by the BSW scheduler and SWC to access the OS services through RTE interfaces and hence it is important that the OS interface functions should be mapped correctly with the generated RTE code

[41]. We implemented a strategy for the integration of RTE and OS modules from different vendors and then carry out C code file generation as shown in the Figure 6.4. We synchronized the OS configuration parameters such as tasks, counter and accessed resources in the OS configuration file (OS.arxml) as per the PikeOS microkernel and our AUTOSAR application settings. Using COQOS generator tools, we generated the code files (source and header files) for PikeOS compliant OS module. Then we used MECEL generator tools to automatically create the RTE code (source and header files) containing the configuration of OS dependent entities such as RTE events and runnable that are then remapped according to the updated OS configuration file. These generated source and header files (code files) are finally integrated together during build process to generate the complete AUTOSAR system executable [41].

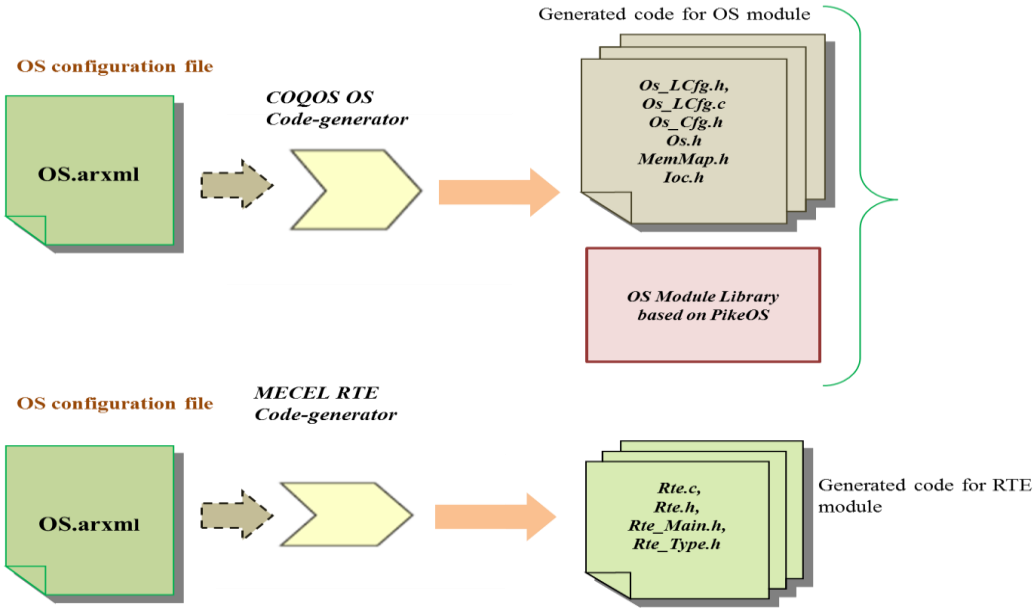


Figure 6.4 Strategy for generation of C code files for Standard RTE module and Proprietary OS module based on PikeOS.

6.2 Design of Linux partition

When compared to AUTOSAR framework, porting of Linux OS on top of PikeOS microkernel does not require much effort as COQOS already provides an embedded Linux distribution named as ELinOS with the latest kernel version (2.6.35) for creating a virtualized Linux OS partition[31]. Unlike AUTOSAR partition that consists of only AUTOSAR application, the Linux partition consists of the kernel, root file system, dependency library files and Linux application. The main challenge was to port a lightweight Linux partition to ensure minimal memory footprint that enables quick startup of the Linux HMI application. We achieved this by removing the unnecessary features and then optimizing the necessary functionalities in the Linux partition.

a) Minimal set of root file system

The Busybox package, that emulates the standard UNIX executables in a lightweight manner in single binary file, is chosen for providing the functionality of root file system since it occupies less memory size and is more suited for small embedded systems.

b) Using Loadable kernel modules and dependency library files

The device drivers for graphics support were added as loadable kernel modules so that there was no need for recompilation, and it saves much memory. Also, the performance will be faster since kernel modules are loaded into the kernel only when required and will be unloaded after the usage. Similarly, we load the necessary dependency files for graphics support (OpenGL/OpenVG) as dynamically linked shared libraries (.so) that in turn will reduce the startup time of the Linux kernel.

c) Other optimizations

The startup Init script was simplified by removing the commands for initialization of unused system services and also the necessary tasks in the Init function were startup in parallel to reduce boot time. The ext4 file format is chosen for the Linux image since it provides faster file system checking and loading during bootup. Also, we optimized the boot loader module (U-boot) by removing the CRC check for checking the data integrity of image during startup and then disabling the unnecessary features in the boot loader module that in turn will reduce the boot time of the Linux kernel.

6.3 Hypervisor technologies

The COQOS hypervisor solution provides abilities for partitioning of resources, inter-partition communication, and flexible scheduling of real-time and non-real-time tasks [12]. However, it is not a typical off-the-shelf virtualization solution and the design and implementation of these technologies should be customized and applied in the context of our prototype model and the requirements.

6.3.1 Partitioning of system resources and separation

The embedded system platform has limited resources like kernel, user memory, I/O peripheral devices, etc. unlike a general purpose computer. COQOS hypervisor provides facilities for allocation of system resources as well as separation between them to ensure fault containment between the multiple operating system partitions.

6.3.1.1 Resource allocation for the partitions

The real-time AUTOSAR and non-real-time Linux application in the hardware platform (I.MX53 SABRE ARD) are separated by allocating them to two different resource partitions that have their memory space, and they share the hardware I/O peripheral devices and processor CPU time. Additionally there is a default system partition known as partition 0 that acts as a watchdog to monitor and carry out fault-handling of the other user partitions, and applications hosted in a partition [29].

The following are design considerations for the resource partitioning and separation between AUTOSAR and Linux user partitions.

a) Ensure an optimal memory footprint and resource allocation (I/O peripheral devices and memory) for the partitions to enable quick initialization of the PikeOS kernel and minimal task latency for the applications.

b) Assign the necessary access privileges and rights for the tasks to ensure strict separation and protection of the resources.

a) I/O Peripheral devices

First the peripheral devices in the target hardware needs to be efficiently allocated to Linux and AUTOSAR partitions to optimize the resource management and minimize the device access conflicts during runtime, that in turn will reduce the task latencies. PikeOS provides facilities for both static allocations and sharing of devices between the partitions [29]. In our system design, we have tried to statically allocate the I/O peripheral devices between partitions as much as possible. The mechanisms for shared device management will add more overhead and delay since the tasks need to pass through the additional PikeOS software abstraction layer to access the shared devices at runtime. We identified the devices in the target system that needs to be statically allocated between AUTOSAR and Linux partitions according to their usage as shown in Table 6.1 below. These devices can be quickly accessed from AUTOSAR and Linux partitions without any delay, and there is no need to pass through the PikeOS virtualization layer.

I/O Peripheral device	Partitions	Usage
Graphical and video processor	Linux	For the graphics and frame buffering of HMI application
FlexCAN module	AUTOSAR	For CAN communication
Display Screen	Linux	To display the HMI application
Internal RAM module	Linux	To provide faster CPU cache access for the resource demanding graphics applications

Table 6.1 Static allocation of devices in Hardware platform to AUTOSAR and Linux partitions.

However certain I/O peripheral devices such as Serial interface, Ethernet and Hardware timer module (GPT) needs to be shared between the AUTOSAR and Linux partitions as shown in Table 6.2 below. Usually, these shared devices are allocated to one of the resource partitions and then they are emulated as external files to other partition so that it can access it. However in our design, the devices that need to be shared were assigned to the system partition (partition 0) and then these devices are emulated as external file descriptors to both AUTOSAR and Linux partitions as shown in Figure 6.5 below. These two user partitions can register with these emulated devices, open and close them whenever they require access to these shared devices. The resource management mechanism in PikeOS guarantees synchronized and mutually exclusive access for the shared devices between the user partitions. The main advantages of allocating them in the system partition is that even if one of the resource partition is restarted or shutdown, it will not affect the other partition since the device will always be available in the

default system partition,0. The system partition is always available and will be affected only when the entire PikeOS system is reset or shutdown [29].

I/O Peripheral device	Partitions	Usage
Ethernet	AUTOSAR/Linux (Assigned to PikeOS System partition, **Partition 0**)	For network access to monitor and debug AUTOSAR and Linux partition
Serial interface console	AUTOSAR/Linux (Assigned to PikeOS System partition, **Partition 0**)	For console output and configuration of the boot script
Hardware timer(GPT) module	AUTOSAR/Linux (Assigned to PikeOS System partition, **Partition 0**)	Used for time synchronization

Table 6.2 List of Shared devices between AUTOSAR and Linux partitions

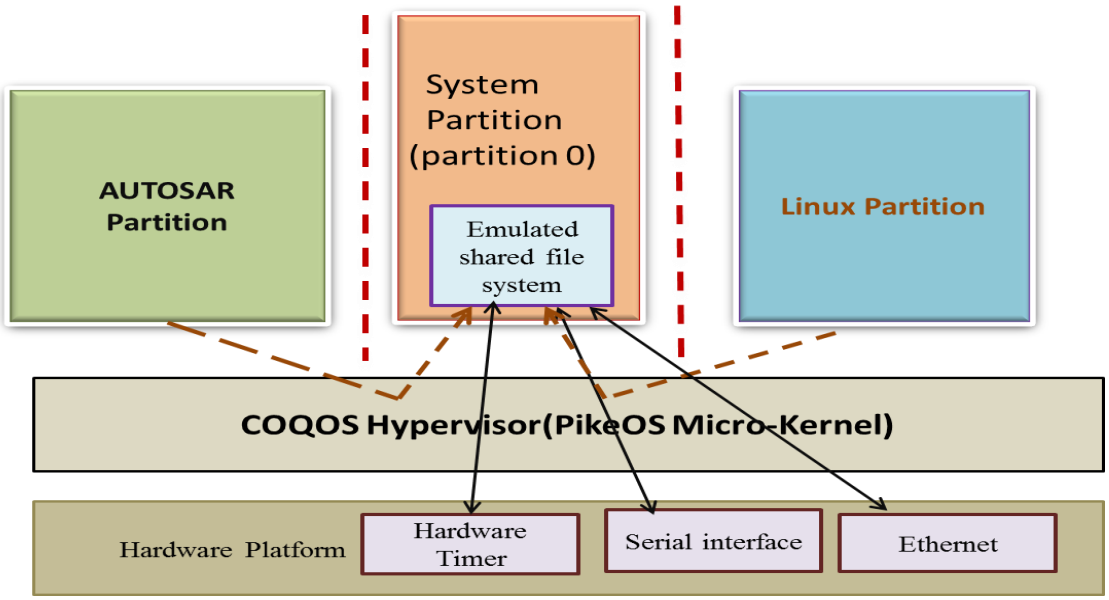


Figure 6.5 Method for sharing of devices between user partitions using System partition.

b) Memory allocation:

The virtual memory in PikeOS is logically divided into two regions, kernel and user [29]. These memory regions need to be allocated sufficiently for AUTOSAR and Linux partitions and also the memory footprint of the partitions needs to be optimized as the PikeOS system can startup quickly if the memory footprint is optimal. For real-time AUTOSAR partition, the kernel memory requirements are relatively deterministic, and there is not much variance between the worst-case and average-case memory requirement. It depends upon the number of AUTOSAR tasks, software component and associated runnables. The AUTOSAR application implemented in our prototype is quite simple and consists of limited tasks and runnables. AUTOSAR development tools were used automatically to calculate the trusted code base and stack size of the tasks and runnable, and they provide a realistic prediction of the kernel memory size required for the AUTOSAR application. We allocated 0.4 MB as the kernel memory for the AUTOSAR application. Likewise, memory map file of the application binary of the AUTOSAR application was used to estimate the user memory for the AUTOSAR application.

However for Linux partition, it is not quite straightforward to determine the memory requirements. We executed the Linux application directly on the target platform and determined the user and kernel memory requirement. We allocated 50 MB as user memory for the Linux partition, and 10% of user memory is used for the kernel memory. Apart from static memory allocation, the following optimizations were also done for the AUTOSAR and Linux partitions

- ✓ A parent task can spawn many child tasks, and these child tasks can consume much memory. As we had only one application in AUTOSAR partition, the child task was limited to 1.

- ✓ To further reduce the RAM memory usage, the read only parts of the applications in PikeOS were not copied to the RAM filesystem. Instead, they were executed directly from the SD Flash memory and mapping is provided in the RAM to the load address of the data segments of these applications.

- ✓ In order to prevent unnecessary caching that degrades performance, caching is inhibited for the I/O memory access and caching is allowed only RAM and kernel memory access.

6.3.1.2 Access privileges for Resources

As explained above, static allocation of devices and memory pools to partitions will provide strict separation between the partitions. However, this is not adequate and additionally we need to ensure that the devices, memory regions or file systems do not unintentionally change the system settings. PikeOS provides options for assigning the access rights to the resources in each partition in order to restrict their access to system interface [26].

- ✓ The memory regions for kernel, RAM and statically assigned I/O devices in each resource partition were given read, write and execute access for the assigned partitions so that they can only exclusively access them.

- ✓ The shared peripheral devices between the user partitions were given only the read and write access rights for the user resource partitions and only the system partition has the execute access for these devices.

6.3.2 Design of Inter-partition communication

The primary design considerations for inter-partition communication between Linux and AUTOSAR partitions are as follows

a) Ensure fast transfer of data signals between the partitions: The inter-partition communication should provide better or equivalent transmission speed as that of the CAN communication networks used in the current state of the art automotive system.

b) Make it compatible with the AUTOSAR communication framework: AUTOSAR standard specifies that the interaction between AUTOSAR and non-AUTOSAR application should be modeled based on AUTOSAR semantics such as port, connectors and interfaces to ensure interoperability and easier migration [34].

c) Minimal error checking mechanism for controlled communication between AUTOSAR and Linux application

PikeOS offers three mechanisms for inter-partition communication mechanism; Shared memory segments, Queuing ports and Sampling ports [29]. The shared memory segments are considered as shared file system, and their file access attributes needs to be configured for each partition. Also, it requires application specific protocols for the synchronization and coordination of read, write access of shared memory segments between the partitions. Queuing ports as the name indicates, uses FIFO queuing and blocking mechanisms that in turn adds more latency to the message transfer [29]. In our design, we have chosen sampling ports for inter-partition communication mechanism since sampling ports are simple to implement, do not block or queue and are quite faster. The sampling ports consist of a message buffer of fixed size, and the messages are updated periodically at a fast rate, i.e., even 2 to 30 milliseconds. The sampling ports are connected between the partitions using the communication channels.

The sampling ports along with the communication channels provide the analogous abstraction as that of AUTOSAR provider and receiver ports connected by assembly connectors, used for the communication between different SWCs [34]. In our design, the Linux application is considered equivalent to an AUTOSAR SWC and the implementation of inter-partition communication was modeled based on AUTOSAR sender-receiver communication mechanism as depicted in Figure 6.6 below [34]. Also, we ensure error checking for the inter-partition communication by implementing a simple validation for checking the data integrity of the signals transferred from AUTOSAR to Linux partition. This validation is done to detect and check that the data is not unintentionally changed or corrupted due to hardware or software faults that could occur during inter-partition communication.

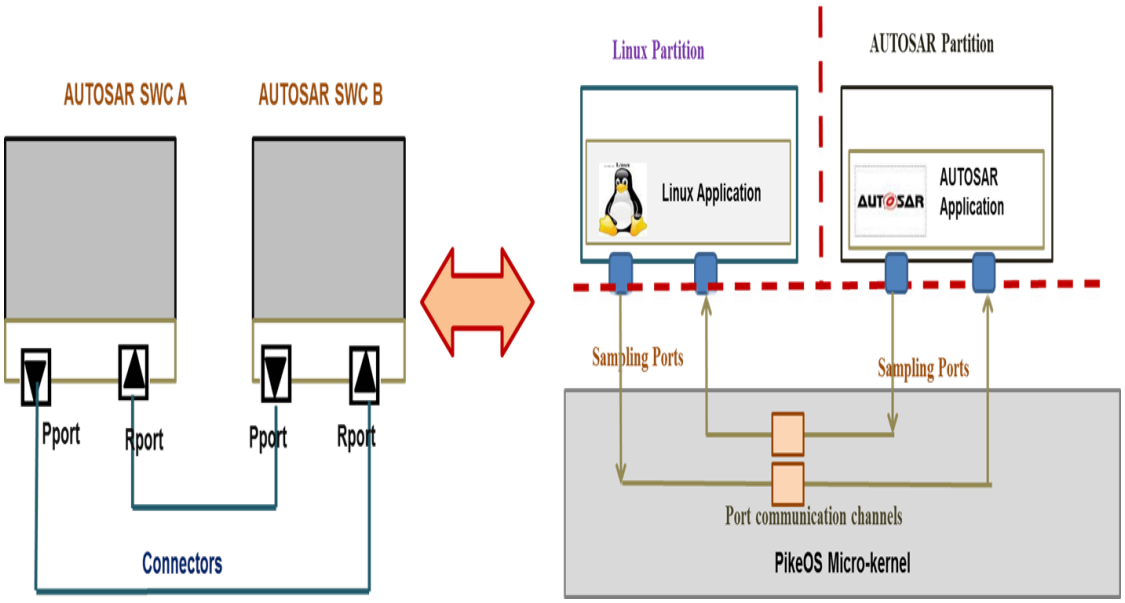


Figure 6.6 Modeling of PikeOS inter-partition communication based on AUTOSAR communication specification.

6.3.3 Time partitioning mechanism for AUTOSAR and Linux partitions

PikeOS provides time partitioning mechanism for sharing the CPU time slots between the partitions [29]. However, the configuration of this CPU time sharing should be done by the designer efficiently to allocate the CPU time between the partitions according to the requirements and depending on the type of applications in the system. The prototype model of multipurpose ECU consists of real-time AUTOSAR application and non-real-time Linux application.

The time partitioning mechanism consists of two phases [30].

a) To allocate time partitions (TP) to different resource partitions such as Linux and AUTOSAR.
b) To allocate time slots/duration (TS) to these time partitions.

The following are the primary design considerations for the time partitioning scheme

i. PikeOS should provide good real-time performance because the AUTOSAR control application has real-time constraints such as deadlines, quick response time, etc. and it should be allocated a fixed and sufficient processor time to ensure determinism and it should finish all of its tasks even in the worst-case scenario. Traditionally, latency sensitive applications have been assumed not to be compatible with virtualization. This is because virtualization often incurs additional processing-time overhead and variability. The time partitioning scheme for AUTOSAR partition should also take this factor into consideration.

ii. Even though, Linux partition does not have real-time constraints and is of less priority when compared to AUTOSAR partition, it should still get a minimal proportion of the CPU time to be able to provide best-effort response. It should not be starved of CPU time slots by the AUTOSAR application.

iii. Both AUTOSAR and Linux partition should get fixed, static time slots and also, the non-real-time Linux partition should not impact the timing behavior of real-time AUTOSAR partition.

In order to achieve the first purpose, we choose to measure the jitter of the periodic task in AUTOSAR application. Jitter is the deviation of an assumed periodic signal in electronics and telecommunication. In reality, applications may get interference from a variety of other events such as from events from other applications, kernel events, the hardware interrupt, etc. All of these interferences may cause a delay and thereby cause jitter for the periodic task. This makes it difficult to trace the source of jitter correctly. Therefore, we should try to reduce interference from other factors and make the jitter value influenced more by the hypervisor solution in our design. For the second purpose, we need to allow Linux system getting access to the CPU time slots whenever the real-time task is idle.

Taking these into consideration, we design a simple task in AUTOSAR application that runs a periodic task to read CAN signals from the simulator tool and pass it to the RTE module. This periodic task is cyclic for every 500ms, and this task would process the CAN signals. The task can tolerate a delay of 0.5 seconds and hence the maximum deadline for the tasks is fixed as 500ms. Real-time tasks are quite deterministic, and we approximately estimated the actual execution time of this task to be less than 20 milliseconds, and it should be scheduled and executed within the allocated time window even during the worst-case scenario. During this time window for real-time task, other tasks are not scheduled by the hypervisor in order to reduce the interference. As the Linux application does not have real-time constraints, we set the Linux partition to be scheduled later when compared to AUTOSAR partition. Also, it is assigned lower priority so that the Linux tasks cannot preempt AUTOSAR task when both arrive at the same time. This will reduce the interference and jitter period for the real-time automotive task.

Computing the time window for Linux partition is not quite straightforward and easy since Linux is a non-real-time OS, and hence it is nondeterministic. In AUTOSAR partition, all the processes are contained within the AUTOSAR application. However, Linux partition can have many system daemon background processes apart from the processes associated with Linux HMI application. Therefore, it is difficult to model the process behavior of the Linux partition. Usually, the HMI application takes 100 milliseconds for execution. However, this is always not correct and applicable since it may even take more time due to memory caching, execution of background and system services, etc. depending on the conditions during runtime. Hence, we give more time slots to Linux partition since it requires more CPU time to execute its processes. This ensures best effort response for Linux partition and also it will not be starved of access to CPU time slots.

PikeOS system partition consists of critical services such as system monitor and watchdog applications for error detection and fault recovery. These critical services should be always executed instantly if errors or faults occur in the user partition or applications, and these critical services should be able to preempt the running AUTOSAR and Linux processes. Therefore, we should give PikeOS system partition the highest priority and make sure that the critical services are always available. However, these faults rarely occur and are not periodic [26, 29]. So, the system partition

processes will not consume more than 10 to 20 milliseconds during normal scenarios when there are no faults in the PikeOS system. Thereby it will not interrupt the execution of AUTOSAR and Linux partitions during normal cases and will not cause any interference.

6.3.3.1 Time partition allocation

After the above analysis, we start to design the time partitioning scheme. TP0 is allocated to PikeOS system partition, TP1 to Linux partition and TP2 to AUTOSAR partition. The special background partition, TP0 is allocated to PikeOS system partition with the highest priority. This is because PikeOS system partition monitors the other two user partitions and should provide quick response for external errors and internal fault situations. Hence, it should always be available and executed instantly during fault conditions. From a safety point of view, such kind of critical application should be temporally separated from non-critical and untrusted applications. AUTOSAR and Linux resource partitions are allocated to normal, foreground time partitions and the AUTOSAR partition is assigned a higher priority compared with Linux. Higher priority enables tasks in AUTOSAR partition to be scheduled earlier than tasks in Linux partition and also protects it from preemption by non-real-time tasks in the Linux partition.

6.3.3.2 Time slots allocation

PikeOS system partition is allocated to TP0 and therefore we do not need to consider how it is scheduled since it will be taken care by PikeOS microkernel by default [29]. AUTOSAR and Linux partitions are allocated to normal foreground time partitions, and it requires a scheduling table. The scheduling table is designed for the allocation of time slots to the AUTOSAR and Linux partitions and is shown in Table 6.3 below.

Time Slots	Offset	Duration	Time Partition
TS0	0	3	TP2 **(AUTOSAR)**
TS1	3	12	TP1 **(Linux)**

Table 6.3 Configuration of Scheduling Table for AUTOSAR and Linux partition.

For each time frame, AUTOSAR partition may get only three time slots (we use the default value of 10ms for each time slot) and Linux systems get 12 time slots. This is because, in our AUTOSAR partition, we have one simple task to read CAN signals and pass it to RTE module. Even though, the total execution time is only 20 milliseconds at average-case for the AUTOSAR application, we have allotted 30ms taking into account the processing time overhead introduced by the virtualization layer. Linux partition gets 12 time slots, and it can access the CPU resources when the automotive control task is idle. The scheduling table defines the offsets and durations for each time partition, and each time partition may be activated only during its time slot. According to our requirement, automotive application should be scheduled first and hence the offset value of AUTOSAR time partition, TP2 is set as 0. In our design, the total time frame for one scheduling cycle takes 150 milliseconds, and this value is much smaller than the deadline duration for our designed AUTOSAR task, i.e., 500 milliseconds. In the worst

case scenario, when all the processes are ready at the same moment, TP0 will get the CPU resource first, followed by the AUTOSAR control task and Linux HMI application.

Our scheduling mechanism allows safety-critical processes in the system partition like system monitor and watchdog applications to be scheduled instantly and have quick response time to external errors and internal faults. Our designed AUTOSAR task and Linux processes are scheduled based on the above schedule table. The AUTOSAR task allocated to time partition, TP2 is assigned higher priority than the Linux processes in TP1. During each scheduling cycle, TP2 will be scheduled first and gets access to CPU time immediately. When the AUTOSAR task in TP2 is idle, then the Linux processes in TP1 would be able to borrow the free, unused CPU time slots dynamically from AUTOSAR partition, TP2 and then utilize it for their execution. This design mechanism ensures best-effort performance for Linux application and allows efficient usage of CPU resources. Both AUTOSAR and Linux applications will get static, time slots and the priority assignments will ensure that the non-real-time Linux application cannot preempt or impact the real-time AUTOSAR application that requires strict timing guarantees. Thus, our designed scheduling mechanism satisfies the runtime requirements of both AUTOSAR and Linux partitions.

7 Implementation and Integration

This section shows how the different components of the prototype model are configured and implemented as per the design techniques discussed in the previous section and how they are integrated into a single system image, that could be run on top of the selected hardware platform (I.MX53 SABRE ARD), using the development tools provided by COQOS hypervisor solution. The AUTOSAR and Linux applications are implemented independently, and the executable files are generated. The PikeOS system configurations like resource partitioning, time scheduling and inter-partition communication mechanism are configured using a PikeOS configuration tool known as CODEO **[**30**]**. The application binaries and PikeOS system configuration are combined into a single integration system. This PikeOS integration system can be considered as the prototype model of multipurpose ECU system. The PikeOS integration system in turn is converted into a bootable PikeOS ROM image that can be downloaded and executed on the embedded target, i.e., i.MX53 SABRE ARD platform.

7.1 Implementation of PikeOS compliant AUTOSAR application

This implementation consists of two phases;

1. As described in the Design section 6.1.1, OS wrapper functions are implemented using POSIX API's for translating and emulating the AUTOSAR OSEK interfaces in PikeOS microkernel. These wrapper functions provide an OS abstraction layer (OSAL) for porting the standard AUTOSAR RTE and BSW modules on top of the proprietary OS platform, PikeOS microkernel.

2. Build the AUTOSAR system executable binary using suitable toolchain.

7.1.1 Translation of standardized AUTOSAR interfaces using POSIX API's

a) Task model

The OSEK interface functions for the task model are ActivateTask() and TerminateTask() that implement the state transition of the AUTOSAR tasks[5,6]. PikeOS provides schedulable runnable entities known as threads, and these threads are used to emulate the OSEK task mechanism [26]. Using POSIX pthread APIs, we implement the standardized AUTOSAR interfaces in OSAL wrapper for emulating state transition of the tasks, to set scheduling priority for tasks and assign them to the schedule table equivalent to the OSEK task concept [6].

b) Events, Alarms and Counters

The OSEK interface functions for the event functionality are SetEvent() and ClearEvent() that are used to set and reset events for the associated tasks[5,6]. PikeOS has an event mechanism associated with each thread to wait and resume using events, and this is used to emulate the OSEK event model [26]. Using POSIX pthread condition variable functions, cond_signal and cond_wait, we implement these standardized AUTOSAR interfaces in PikeOS wrapper for setting and clearing events and enable the state transition of a thread to suspended and running states. Also OSEK alarm and counter interfaces, SetAlarm() and IncrementCounter() are emulated using POSIX timers and POSIX alarm function. When the counter reaches the predefined timer value set by the associated alarm function, the alarm expires and it will generate a POSIX

signal, SIGALRM to notify the event mechanism that in turn will activate, restart or suspend a thread associated with the alarm function.

c) Scheduler and Resource management

The OSEK OS uses event driven, fixed priority scheduling that is also supported by the PikeOS implementation and therefore the POSIX API, sched_yield() is used by the tasks to get access and relinquish the CPU resource. The resource management mechanism of OSEK OS was realized using the pthread mutex functions by considering the shared resource as a critical section and thereby mutual exclusion is enforced. The mutex locking and unlocking functions implement the GetResource() and ReleaseResource() OSEK interfaces respectively. Also, whenever a thread acquires a resource, its priority is raised to the maximum controlled priority of the corresponding user partition. This is equivalent to the OSEK resource ceiling priority for protection of shared resources from deadlocks and priority inversion problem [6].

The translation and mapping of the AUTOSAR OSEK interfaces in the PikeOS microkernel using POSIX API functions are shown in the Table 7.1 below.

OS Concept	AUTOSAR interfaces	POSIX API functions
Task model	ActivateTask() TerminateTask()	pthread_create() pthread_join() pthread_kill() pthread_cancel() pthread_setschedparam()
Event Model	SetEvent() WaitEvent()	pthread_cond_signal pthread_cond_wait()
Counters and Alarms	SetAlarm() IncrementCounter()	alarm() & SIGALRM POSIX signal timer_settime()
Resource management	GetResource() ReleaseResource()	pthread_mutex_lock() pthread_mutex_unlock()
Scheduler	Schedule()	Sched_yield()

Table 7.1 Translation of AUTOSAR interfaces using POSIX API functions in PikeOS

7.2 Generate PikeOS compliant AUTOSAR system executable binary

After updating the OS abstraction layer(OSAL) as explained above, the code files(source and header files) for the RTE module based on standard AUTOSAR stack and OS module based on PikeOS microkernel are generated by customizing the build process as described in Design section 6.1.2. The complete system executable was

created for AUTOSAR application using these generated code files by the AUTOSAR build process [41]. The gcc 4.4 cross compiler toolchain was used for build process since it supports POSIX APIs, and it is also supported by the hardware platform (I.MX53 SABRE) used in our pilot implementation. All the makefiles in the AUTOSAR application are updated according to the new toolchain to setup the compilation and build process, and an AUTOSAR application binary executable file that could be run on top of PikeOS microkernel was created.

7.3 Generation of application binary files for the Linux partition

In this prototype model, ELinOS, embedded Linux distribution provided by SYSGO is used for providing the kernel and root file system for the virtualized Linux partition [31]. As described in the Design section, a minimal set of root file system and Linux kernel are configured using ELinOS to ensure less memory footprint for the Linux partition. Once they are configured, the Linux HMI application, dependency library files, and loadable kernel modules for the graphics support are copied into the root file system. ELinOS contains its development environment and tools for creating the application binary files for the Linux partition. It creates two application binaries, one for the Linux kernel and other for the Linux root file system.

7.4 PikeOS System configuration

Once the application binary files for AUTOSAR and Linux partitions are created, the PikeOS system mechanisms like resource partitioning, time scheduling of partitions and inter-partition communication needs to be configured according to the design settings discussed in the previous section. PikeOS provides an Eclipse based integration development environment named as CODEO for system configuration and integration of components into a single target image that can be deployed on the hardware platform [30]. It provides a graphical editor for the configuration of PikeOS system properties like resource partitioning, time scheduling of partitions and defining inter-partition communication channels and ports. Using this project configurator editor, we configure the PikeOS system settings as defined in the Design section.

7.4.1 Virtual Machine Initialization Table

PikeOS consists of a table known as Virtual Machine Initialization Table (VMIT). As the PikeOS system settings are configured using CODEO, they are automatically updated in a file in XML format known as VMIT.xml, which contains the VMIT table. The specifications of the PikeOS partitions, their resource requirements, time scheduling parameters, inter-partition communication settings and the applications that execute within the partitions are updated in this file [29]. During the build process, this file is compiled into a binary module and included in the target image. The target image is loaded into the hardware platform and when the PikeOS system is startup, PikeOS microkernel reads this VMIT binary module and creates the resource partitions, inter-partition communication channels and time scheduling of the tasks with the configured settings in the hardware platform [30]. The VMIT specification file is divided into the following major subsections:

a) **Partition table:** Contains resource settings of the partition like memory resources, allotted I/O peripheral devices, applications contained in the partition, etc.

b) Connection table: Contains the inter-partition communication channel settings.

c) Schedule table: Contains the time partitioning parameters of the time partitions like offset, duration, etc.

7.4.2 Resource partitioning

The resource partitioning mechanism is configured using the CODEO project configurator. First using CODEO, the two user partitions namely, AUTOSAR and Linux are created. By default, PikeOS creates a system partition known as service partition that acts as a watchdog, to monitor the user partitions and for debugging purpose [29]. The following settings were configured for the resource partitions as per the specifications in the Design section**.**

- ✓ Allocation of kernel and user memory regions to user partitions.
- ✓ The I/O peripheral devices that are exclusively used by each partition were allotted to the AUTOSAR and Linux partition. The physical address of the I/O memory regions is associated with the virtual address of the resource partitions.
- ✓ Certain I/O peripheral devices need to be shared between the AUTOSAR and Linux partitions. These devices were assigned to the system partition (partition 0) and configured as external file descriptors to both AUTOSAR and Linux partitions.
- ✓ The access privileges of the partition for the assigned resources were configured. Also, the maximum priority of the AUTOSAR and Linux partitions are set.
- ✓ Then the AUTOSAR application binary and the Linux application binaries are copied to the AUTOSAR and Linux partitions respectively.

7.4.3 Time scheduling mechanism

As explained in the design strategy, time partition and time slot allocation for Linux and AUTOSAR user partitions are configured using the PikeOS configuration editor. The scheduling table containing the offset value and time duration for the AUTOSAR and Linux partition are configured. Also, the priority of the AUTOSAR and Linux time partitions are configured using PikeOS configuration editor, CODEO [30].

7.4.4 Inter-partition Communication mechanism

As specified in the Design section, the inter-partition communication was implemented by communication channels consisting of sampling ports. The source and destination sampling ports on both Linux and AUTOSAR partitions are configured, and the attributes of the sampling ports such as the direction (Sender/Receiver), refresh time, maximum message size are setup using the CODEO project configurator editor [30]. Sampling ports consists of a message buffer that can store two messages at a time, and the configured refresh rate determines the validity or timeout period of the messages. The sampling ports between the two partitions are linked by the inter-partition communication path known as channel [29]. The properties of the channel such as endpoints (source and destination ports) and port type are also setup using the CODEO

configuration editor, and the configured channel will connect the sampling ports in each partition and establishes a communication path between the two partitions.

7.4.4.1 Coding of sender and receiver functions for transfer of data signals

As the sampling ports and communication channels are configured, a path is established for the data transfer between the two partitions. The channels contain an I/O buffer and utilize message passing protocol, allowing data to be transferred in variable sized messages up to the maximum message size that is specified [29]. In the prototype model, the exchange of data signals like speed and revolutions per minute(RPM) takes place between the SWC in automotive control application in AUTOSAR partition and the HMI application in Linux partition using the inter-partition communication channels.

Sender function in AUTOSAR SWC

```
/* Read speed data value from CAN communication bus
periodically via RTE */

Speed =
Rte_DRead_AppSwCpType1_RPortPrototype1_VariableDataP
t1();
wait(100);

/* If it is a new value, compute checksum for new value and
immediately write to sampling ports */
If (Speed != oldSpeed)
{
/* generate checksum for data-value */
Send_Checksum = Calc_check(Speed)

/* Write data and check-sum to sampling ports */
fd = open("/sport/speed", O_WRONLY);

sportioc_speed.sport_op.write.buf[0]=&Speed;
sportioc_speed.sport_op.write.buf[1]=&Send_Checksum;
r_ioctl = ioctl(fd, SPORT_WRITE, &sportioc_speed);
}
}
```

Receiver function in Linux Application

```
/* Read the sampling port periodically */

fd = open("/dev/vmsport0", O_RDONLY);
ioc.sport_op.read.buf [0]= =&Speed;
wait(100);

/* If new value, read the checksum and check if it is indeed
correct!! */

If (Speed != oldSpeed)
ioc.sport_op.read.buf [1]= =&receive_Checksum;
r_ioctl = ioctl(fd, SPORT_READ, &ioc);

/* function to validate the checksum for data-value */
Valid = validity(r_ioctl)
/* If checksum valid then update the HMI display unit
if (Valid)
{
// call the HMI notification functions //
...

}
```

Figure 7.1 Pseudo codes for the inter-partition communication mechanism between the AUTOSAR SWC and HMI application.

In order to realize the communication mechanism, functions need to be coded using the PikeOS sampling port APIs. We coded sender and receiver functions on AUTOSAR and Linux partitions respectively that are analogous to the AUTOSAR RTE module read and write functions to emulate the AUTOSAR sender-receiver communication mechanism as presented in Figure 7.1 above [34]. The sender function was placed in the

runnable entity of the SWC. The two data elements, Speed and RPM, were defined with the AUTOSAR data types and associated with the corresponding two sampling ports. The SWC periodically reads the speed and RPM values, (every 500 milliseconds) sent to the CAN communication bus via the RTE functions. Whenever there is an updated value for speed or RPM, a simple checksum (Ex: square of the data value) is calculated based on the updated data value. Then the updated data values and the calculated checksum are written to the AUTOSAR sampling ports. The communication channels act as a link between the partitions and transfers the data values and checksum to destination sampling ports in the Linux partition. The receiver function is placed in the Linux application, and it periodically polls and reads the values from sampling ports. Whenever there is an updated value, it reads the corresponding checksum. It computes the checksum with the same algorithm, which is used in sender function, and compares the computed checksum and received checksum. This simple checking is done to detect and ensure that the data is not unintentionally changed due to hardware or software faults during inter-partition communication. If they are equal, then the Linux application notifies these values to the HMI unit that in turn updates the associated speed and RPM indication in the instrument cluster display. The sender function was implemented in the SWC entity in AUTOSAR application in AUTOSAR partition, and the receiver function was implemented in the Linux application in Linux partition.

7.5 Integration and generation of ROM image for the target hardware

The application binaries for the AUTOSAR and Linux partition are generated, and the PikeOS system settings that consist of configuration of resource partitioning, time scheduling of partitions and inter-partition communication settings are stored in the PikeOS configuration file, VMIT.xml. Apart from this, there are also PikeOS system binary objects such as kernel, PikeOS system software module (PSSW) and platform support package (PSP) for the target platform [30]. These binary objects were provided along with PikeOS software, and they are directly used without any additional configuration in our prototype implementation.

The prototype model of multipurpose ECU system using PikeOS microkernel is transformed into a PikeOS integration project, and it consists of three parts

1. Application binaries for AUTOSAR and Linux partitions
2. PikeOS Configuration file, VMIT.xml
3. PikeOS binary objects such as kernel, PSSW and Platform support package.

This integration project is converted into a single bootable ROM image using PikeOS ROM image generation tool as shown in Figure 7.2 below

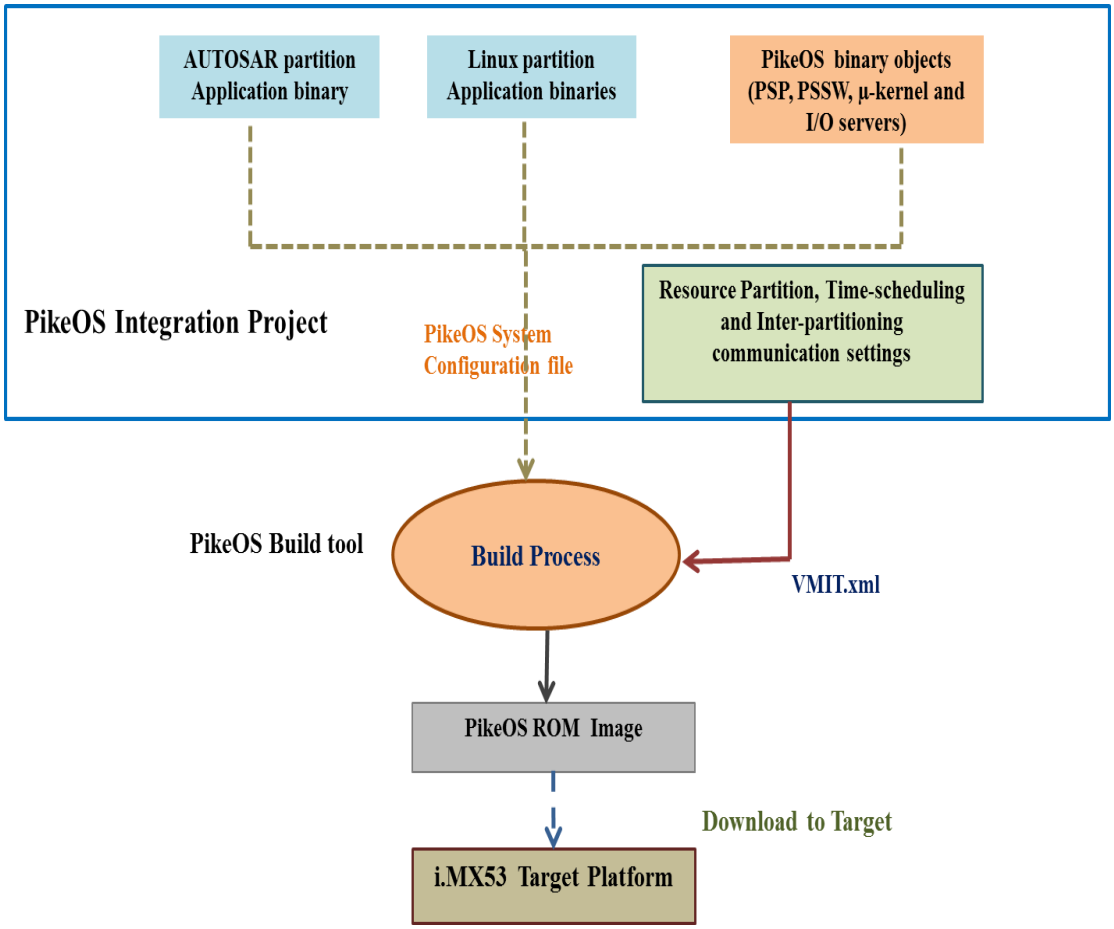


Figure 7.2 Overview of PikeOS Build and image generation process.

The ROM image generation tool integrates all the components of the integration project into a composite ROM image file as shown in Figure 7.3 below [30].

Boot loader	Microkernel	PSSW	VMIT	Application Binaries

Figure 7.3 PikeOS ROM Image layout.

7.6 Running the PikeOS image on the Hardware platform

The PikeOS ROM image containing AUTOSAR and Linux partitions on top of the PikeOS microkernel is downloaded to the target system, I.MX53 SABRE board. Once the board is switched on, the image starts booting up, and the boot loader starts the PikeOS microkernel. The PikeOS initialization steps in the target hardware are shown in the Figure 7.4 below. The microkernel gets all the necessary information about the partitions, applications in partitions, assigned resources, time scheduling and inter-partition communication parameters from the VMIT file. Then it loads and initializes the AUTOSAR and Linux partitions with the allocated resources according to VMIT file and a partition daemon thread is started for each partition. The partition daemon manages the partitions and then loads and starts up the AUTOSAR and HMI application processes in the corresponding memory regions in each partition. Then the kernel also

sets up the application scheduling and inter-partition communication between the two partitions as specified in the VMIT.xml [29].

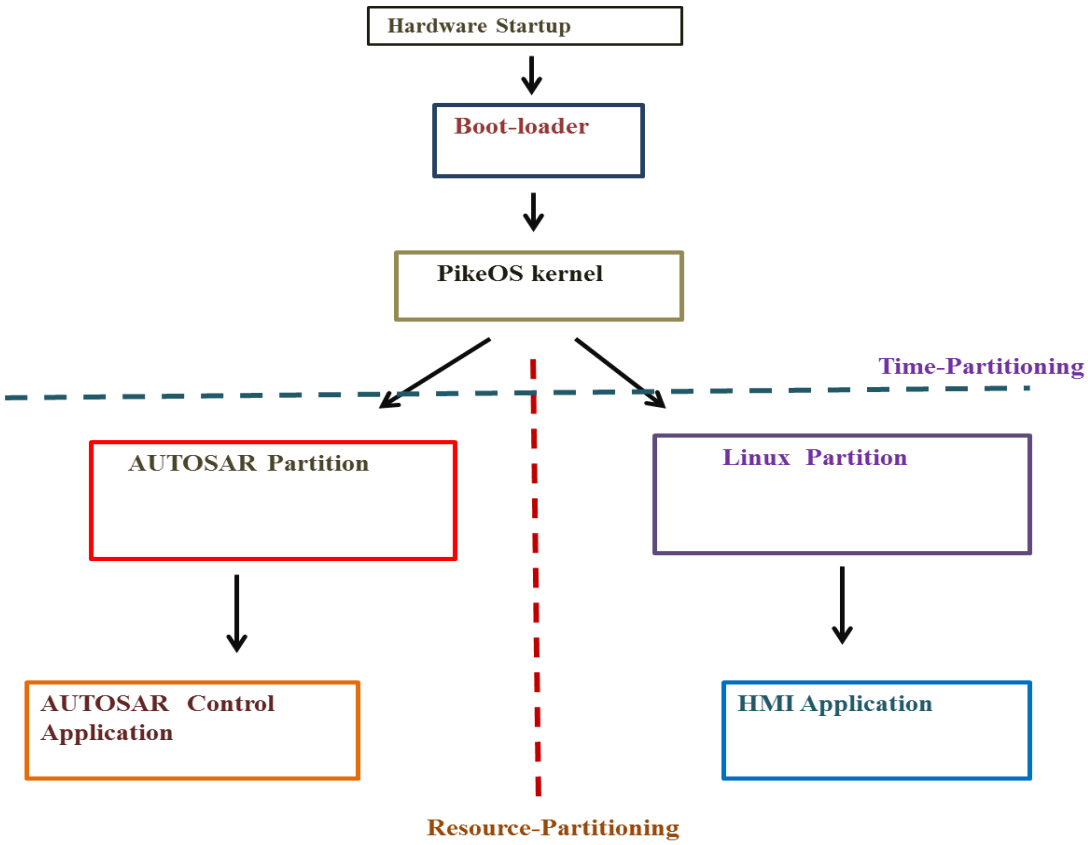


Figure 7.4 PikeOS Initialization steps in Target hardware.

The working prototype model, i.e., PikeOS system image with the AUTOSAR and Linux applications executing simultaneously on separate partitions in the common target platform, i.MX53 SABRE board demonstrates the concept of a consolidated ECU system. It can be observed that AUTOSAR and Linux HMI applications were able to initialize and execute concurrently on the shared hardware platform, and it proves that the AUTOSAR and Linux OS were correctly virtualized on top of PikeOS microkernel. The AUTOSAR application could process the frames sent from the CAN simulator tool and also these values were transferred to the Linux application through the inter-partition communication i.e., sampling ports and channels. Thus, the inter OS communication mechanism could be implemented correctly between AUTOSAR and Linux application in different partitions [29]. Thus, this working prototype model demonstrates the concept of a consolidated, multipurpose ECU implemented using virtualization technology.

8 Measurement Results and Analysis

We carried out measurements and testing in our proposed prototype model of multipurpose ECU for the evaluation of the nonfunctional requirements like boot time, signal communication delay and separation between the diverse applications. The measurement results were compared with the equivalent existing models or current state of the art system and then analyzed to assess the applicability of our proposed prototype model for real-world automotive systems. The boot time of the multipurpose ECU, hosting two heterogeneous automotive applications that are executing concurrently on the same hardware platform, was measured to be nearly 2.2 seconds. The communication between the applications was localized on the same hardware system instead of using an external communication bus and hence we could achieve a faster communication rate of 30 to 40 milliseconds (ms) for the transfer of data signals in our prototype model. Also, we could ensure isolation and separation between the diverse OS partitions even though they are hosted on a shared hardware platform. From our measurement results and analysis, we could infer that our proposed implementation of multipurpose ECU does achieve equivalent or better performance for these parameters, i.e., boot time, message transmission time and separation between applications, when compared to related existing models or the state of the art ECU system.

8.1 Evaluation Strategy and Measurement parameters

In this report, we present the design and implementation of an artifact i.e., multipurpose ECU system. This artifact was realized by a prototype implementation and hence a full-scale evaluation and testing is not required. For our prototype model of multipurpose ECU system, there were five requirements defined in Section 5.1. Some of the functional requirements such as porting of AUTOSAR and Linux OS on a common platform, flexible time scheduling between the different applications, resource sharing mechanism for allocation of system resources among different applications, and inter-system communication for exchange of data signals between AUTOSAR and Linux applications are met by the design and implementation of the prototype model itself using the hypervisor solution. However, some measurement tests under simple conditions were carried out to check the applicability of our proposed prototype model in the automotive systems. The following parameters were measured in our prototype implementation of consolidated ECU system.

a) Boot time of the system

The startup time of the automotive ECU systems is an essential requirement. It is not allowed to use few minutes to boot up the system like a general purpose computer. Most of the vehicular system functionalities require their applications to be started up within 10 to 20 seconds after the car is started. The critical automotive applications such as a rearview camera, engine management system must be available instantly within 3 to 5 seconds after power on. Even less critical applications such as audio FM systems and infotainment screens must be available within 20 seconds after power on for good user experience [43]. In the state of the art systems, speed and RPM values are required to be read soon after the car is started. In the multipurpose ECU prototype, AUTOSAR application and Linux application should be able to startup as soon as the common hardware platform (I.MX53 SABRE board) is powered on. Though there is no specific limit to the boot time, we expect this value to be as small as possible and hence for our

multipurpose ECU prototype, we set the benchmarking value for boot time to be less than 5 seconds.

b) Signal communication delay

In our prototype model, the data signals were transmitted between the applications in different partitions using inter-partition communication mechanism. It should not introduce high delay when compared to the state of the art vehicular communication networks. There are several vehicular communication bus standards like Controller area network (CAN), Local interconnect network (LIN), Media oriented systems transport (MOST), FlexRay, etc. For our analysis, we consider only the CAN communication bus since it is predominantly used in the in-vehicle networks [44]. In the CAN communication networks, the delay for transfer of data signals takes 100 to 1000 milliseconds depending on network load and the priority of signals [44]. For our prototype model, we expect the value of signal transmission delay to be within smaller range, and less than the state of the art CAN communication bus since our communication mechanism is localized on the same hardware and hence it should be much faster. So for our multipurpose ECU prototype, we set the benchmarking value for signal communication delay to be within the range of 50 to 100 milliseconds.

c) Isolation and separation between the applications

The prototype model should provide strict isolation between the concurrently executing applications even though they share the same hardware platform and system resources such as CPU, I/O devices, etc. In our prototype, there are two partitions running on the same processor and the hypervisor manages these partitions. The hypervisor module should ensure strict isolation between the partitions. The faults and errors in one partition should not affect the execution of the application in the other partition or the underlying hypervisor layer. This isolation property is checked by a simple test where we continuously reboot or halt one of the partitions and observe the execution behavior of the application in the other partition.

8.2 Measurement setup:

The PikeOS system in our prototype model consists of a system partition known as service partition (partition 0) apart from the user partitions, AUTOSAR and Linux. This service partition provides console access to control, debug and monitor the user partitions and applications running in each partition. PikeOS provides a tool named as MUXA tool that can be run on a host PC, and it provides bidirectional virtual channels for remotely connecting and communicating with system partition to monitor the user partitions and applications executed in the target hardware from the host PC [29, 30]. This is equivalent to a virtual terminal application such as Telnet that provides bidirectional communication facility. Using this MUXA tool, we established channel connections with the system partition (partition 0) from a host PC to check the status information of the user partitions and applications in each user partition, inter-partition communication channels, ports and running processes. Also, it is possible to halt or reboot the user partitions and the target system from this MUXA tool [30]. We carried out our measurement tests using the MUXA utility provided by PikeOS.

8.3 Boot time of the prototype system

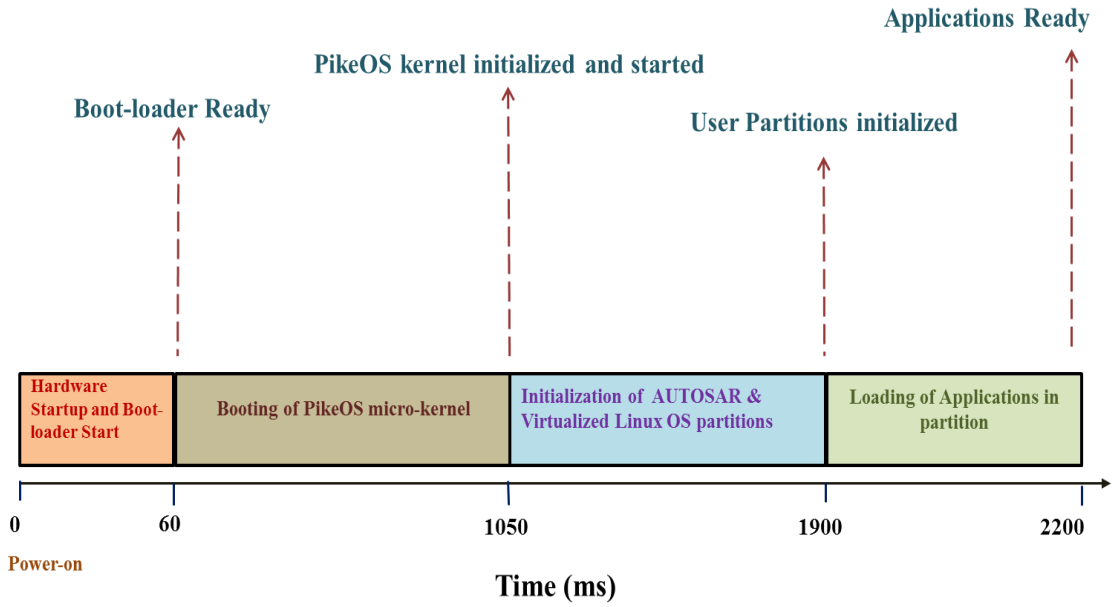


Figure 8.1 Measurement time of PikeOS Startup phases.

We inserted some code in the Init module of the PikeOS to dump the initialization times of the boot loader, PikeOS partitions and applications in the console display after switching on the hardware platform. The complete bootup time was approximately 2200 milliseconds (2.2 seconds) as shown in the above Figure 8.1.

8.3.1 Analysis of the Boot time results:

As shown in the above Figure 8.1, the startup of the PikeOS system consists of four phases. The first phase depends completely on the hardware platform. The next three phases depend on the hardware as well as the software configuration of the kernel, partitions and applications in the PikeOS system [29]. The time measurements for the initialization times of the boot loader, kernel, PikeOS partitions and applications were presented in Table 8.1.

Initialization Phases	**Time (milliseconds)**
Boot loader ready since power on, $\mathbf{T_1}$	**60 milliseconds**
Initialization of PikeOS kernel since power on, $\mathbf{T_2}$	**1050 milliseconds**
Initialization of user partitions since power on, $\mathbf{T_3}$	**1900 milliseconds**
Startup of application since power on, $\mathbf{T_4}$ ***Complete bootup time of PikeOS system***	**2200 milliseconds**

Startup of application since PikeOS kernel startup = $\mathbf{T_4 - T_2}$	**1150 milliseconds**
Startup of Application since initialization of partition = $\mathbf{T_4 - T_3}$	**300 milliseconds**

Table 8.1 Time measurements for PikeOS Initialization Phases.

In our proposed design, we have chosen the following design options and optimizations so that there is not much latency or delay while booting up the system.

a) In the resource partitioning mechanism, devices in the hardware platform were carefully considered according to their utilization by user partitions and most of the devices were included statically and given direct exclusive access in the user partitions. Only three I/O devices were dynamically shared between the user partitions and they were included directly in the system partition. Thus, we minimized the functionality of the system partition that acts as a partition manager in PikeOS system. This helped to minimize the latency in the second phase during loading of the PikeOS kernel since it takes less time to initialize the system partition as it has minimal I/O devices assigned to it.

b) PikeOS system carries out the memory mapping and management before the initialization of the user partitions [29]. It was important to ensure a minimal, as well as sufficient memory footprint for the AUTOSAR and Linux partitions. We analyzed the memory requirements of the AUTOSAR and Linux partitions and then allocated minimal and average-case memory space for kernel and RAM user memory regions of the AUTOSAR and Linux partition. As a result, the memory mapping for the partitions will be carried out quickly, that in turn reduced the startup time of the partitions during the third phase.

c) The AUTOSAR partition contains only a simple AUTOSAR application, and we optimized the StartOS functionality of the AUTOSAR application to immediately start up the AUTOSAR application. The Linux partition is a typical OS partition consisting of Linux kernel, root file system and libraries apart from the HMI application. We ported a lightweight Linux partition with minimal set of root file system, kernel modules, etc. so that the Linux partition and applications could be initialized quickly. Thus, we optimized the startup time of the applications during the fourth phase.

The measured boot time of our prototype model using PikeOS is compared other equivalent in-car systems as shown in the Figure 8.2 below.

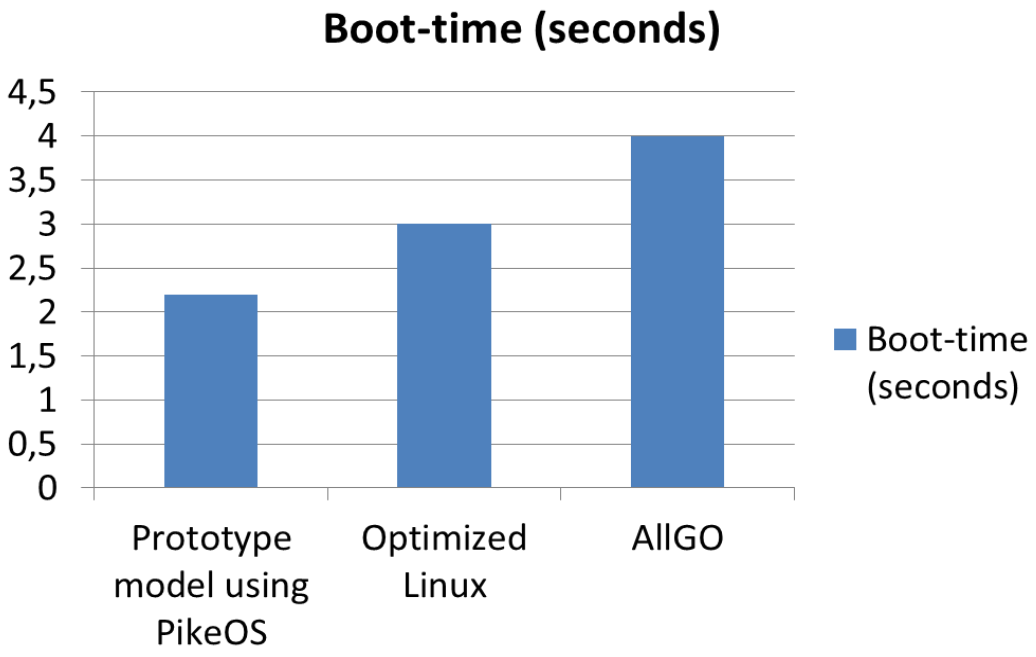


Figure 8.2 Boot time for our prototype model, Optimized Linux and AllGO.

As it can be seen, our pilot implementation using PikeOS has better performance compared with the Optimized Linux and AllGO Android systems [45, 46]. Even though we do not have much information about the type of hardware and applications deployed in these systems, we can safely assume that our prototype model is more complicated since it executes two state of the art heterogeneous automotive applications on the same inexpensive, hardware platform.

PikeOS utilizes a thin, microkernel while other systems use a monolithic kernel. Apart from this factor, the fast bootup performance also depends on the hardware, software configuration of the kernel, partitions and applications in the PikeOS system [29]. There are certain other options to increase the bootup performance of our system. In our proposed design, after the PikeOS kernel is started up, the Linux and AUTOSAR partition were started up simultaneously by the kernel. PikeOS provides a mechanism known as modular boot and using this concept; it is possible first to load only part of the PikeOS image and thereby bootup the PikeOS kernel and the real-time AUTOSAR partition earlier [12, 29]. Then the rest of the image can be loaded later, and the Linux partition can be startup. This in turn will provide a faster startup time for the AUTOSAR application and slower startup time for Linux application compared to our approach. The designer can choose either of these approaches, and it purely depends on the fast boot requirements for the applications considered for the multipurpose ECU system. In automotive system, infotainment and automotive applications should be startup immediately after ignition on, and they require more or less equivalent bootup times.

8.4 Signal communication delay

In our prototype model, the speed and RPM data signal values are transmitted from the AUTOSAR to Linux application through the inter-partition channels and it is necessary that inter-partition communication occur without considerable delay. We measured the

time for the inter-partition communication by inserting code in the sender and receiver functions in the AUTOSAR and Linux application respectively to dump the time stamps for the transmission and reception of these data signals. The measured inter-partition communication time for data signals was 30 to 40 milliseconds (ms).

8.4.1 Analysis of inter-partition communication delay

In the state of the art vehicular architecture, the remotely connected ECUs uses different communication buses like CAN, MOST, LIN, FlexRay, etc. The CAN communication networks are predominantly used and therefore it is considered for our comparative analysis [44]. In vehicular networks, many ECU nodes will be ready to transmit at the same time, and they share a common communication medium, CAN network. CAN communication networks utilize a message based protocol and prioritized arbitration scheme for determining the transfer of messages according to their configured priority value [44]. The signals are encapsulated in a message frame known as CAN frame according to a standard format, and the protocol is used for formatting and de-formatting of signals, error handling, priority arbitration, transmission and reception of messages, etc. The delay associated with transmission of a message in the CAN networks consists of

a) Message delay: encapsulation and decapsulation of signals into a fixed frame format, error checking, etc.
b) Queuing delay: The time that the message needs to wait in the queue until it gets a slot to transmit according to its priority.
c) Transmission delay: Transmission time for the message depends upon data rate of communication medium and propagation distance.

Thus, the delay associated with the messages could be between 100 to 1000 milliseconds depending upon the traffic load in the network and the distance that the message needs to traverse from sender to receiver ECU node [44]. Also, this CAN communication mechanism utilizes additional processor power and requires complex wiring mechanism [24]. Two ECUs in automotive systems, for instance, an infotainment and powertrain control ECU usually transfer 10 to 40 signals between them, and they have to contend with the signals transmitted by the other 50 to 60 ECU nodes [44].

In our prototype design, the ECU applications that interact with each other frequently can be integrated in a single consolidated platform, and they utilize simple inter-partition communication mechanism using sampling ports and channels. This helps to reduce the traffic load on the external CAN communication bus since the communication between these applications is localized on the same hardware platform and also the transmission rate is faster, i.e., 30 to 40 milliseconds(ms). There is also no need for the formatting and de-formatting of signals into a standard message frame and a complex communication protocol as there is no necessity for contention with message signals from other ECUs. Further, the signal propagation distance is significantly reduced by integrating these two ECU systems on the same platform. Thus in our proposed implementation of multipurpose ECU system, these 10 to 40 data signals can be transferred by configuring the necessary sampling ports and channels between the infotainment and powertrain control applications executing in separate partitions in the same hardware platform.

8.5 Isolation and separation between the applications

The two heterogeneous applications in our prototype implementation are separated and isolated by our design configurations even though they share common resources such as I/O devices, CPU processor, inter-partition communication channels and the underlying PikeOS microkernel. We carried out a simple testing procedure to check this separation property. We used Muxa tool [30] to connect to the system service partition (partition 0) and then we continuously rebooted/halted one of the user partitions, and we observed the execution state of the application in the other user partition. The AUTOSAR application continued its execution and was processing the CAN frames without any impact when the Linux partition was restarted or halted. Likewise, we could observe that the Linux HMI application could continue its execution behavior without any interference when the AUTOSAR partition was restarted or halted. From this simple testing, we can logically conclude that the partitions are strictly isolated, and the faults or errors in one resource partition do not propagate to the other. In the state of the art systems, the applications are isolated by executing them on separate dedicated hardware platforms that are remotely connected. However, this incurs hardware cost and complicated wiring mechanisms. In our prototype model, we have implemented following design options to ensure the strict separation of applications executing on a single, shared hardware platform.

a) Spatial isolation was implemented by statically allocating the devices used by each user partition to them exclusively and also each user partition was given sole access rights to these devices so that other user partition cannot access it. The kernel and RAM user memory in the hardware system were partitioned into virtual memory regions and then statically allocated to the user partitions with suitable access rights during design phase. Thus, the memory pools are statically allocated to the user partitions during system startup, and they can only access this allocated memory space and cannot intentionally or unintentionally access the code and data regions of other application.

b) The partitions still share some I/O devices and the CPU processor time. Temporal isolation was guaranteed by allocating the shared I/O devices in the system partition that acts a partition manager, and each user partition has been provided with limited access to these I/O devices. Also, the dynamic resource management protocols in the PikeOS ensure mutual exclusion to these shared devices for the two user partitions [29]. Likewise using time partitioning mechanism, the user partitions were allotted fixed and static time slots so that the software processes in each partition are also separated [29]. Thus, a faulty or malfunctioning process in one application cannot acquire the complete CPU time or cause a denial-of-service (DoS) attack on other processes. It can only utilize the time slots allotted to it and cannot affect the time scheduling of the processes in other partitions.

From these, we can infer that the two heterogeneous applications in the prototype implementation of multipurpose ECU are strictly separated, albeit they share the same hardware platform and thus they have equivalent isolation and separation property as that of the state of the art, remotely connected ECU systems.

8.6 Significance of our measurement results and analysis

Automotive companies compete mainly on cost/performance ratio. Hence, it is important that our proposed pilot implementation of multipurpose ECU should meet the nonfunctional requirements like fast boot time, less message transmission time and strict isolation between the applications. The boot time of the multipurpose ECU is nearly 2.2 seconds even though it hosts two heterogeneous automotive applications, executing concurrently on the same hardware platform. The communication between the applications is localized on the same hardware system instead of using an external communication bus and hence we could achieve faster communication rate of 30 to 40 milliseconds (ms) for the transfer of data signals in our prototype model. Also, we could ensure spatial and temporal isolation between the applications even though they are hosted on a shared hardware platform. From our measurement results and analysis, we could infer that our proposed implementation of multipurpose ECU does achieve equivalent or better performance for these parameters i.e., boot time, message transmission time and separation between applications, when compared to related existing models or the state of the art ECU system. Even though, we have carried out limited measurement tests under simple conditions, these results are significant since we could demonstrate a basic validation of our prototype model. It could serve as a simple proof of concept that our proposed model of multipurpose ECU using virtualization solution could be accepted for deployment in automotive embedded systems in order to solve the integration problem of core automotive control and infotainment functionalities. This prototype model can be introduced in future vehicular systems as a full scale implementation after further research study and comprehensive analysis, with more advanced technological enhancements.

9 Discussion and Conclusions

Here we reflect and discuss about our design options for the prototype model, challenges faced during the design and evaluation phase, limitations and other design alternatives that could be considered for the refinement of our proposed prototype model. This self-introspective analysis will help to provide essential inputs and valuable insights that can be the basis for further comprehensive analysis required for full-scale implementation and deployment of a multipurpose ECU system in the automotive domain.

9.1 Reflections on Design and Implementation phase

We started the design phase in our thesis work by setting the requirements for the prototype model of multipurpose ECU system. We consider the conflicting set of functional requirements for diverse OS platforms like real-time AUTOSAR architecture and non-real-time Linux OS as well as the nonfunctional automotive requirements like quick boot time, minimal signal communication delay and isolation of applications, etc. Based on these requirements, we explored several consolidation strategies to integrate the AUTOSAR and Linux operating systems on the same hardware platform [35, 36, 53]. Significant architectural changes were required in the standardized automotive framework; AUTOSAR in order to support the non-real-time infotainment applications. These modifications require major effort and time and hence we did not consider this approach for our prototype model. We choose virtualization since it is a proven technology that has already been widely used in avionics and telecom domain for consolidation of mixed critical systems (hard real-time and non-real-time) [23]. Even though, virtualization technology does add some processing overhead due to the additional abstraction layer, this can be easily mitigated by the high-performance processor platforms available in the market. The choice of hypervisor for the prototype model was an important design choice. There were several open source hypervisor solutions available like KVM, OKL4 and Xen, etc. However, they provide support mainly for different variants of Linux OS used for mobile applications, and there was not much support available for the paravirtualization of AUTOSAR real-time OS. Also, open source solutions inherently do not have strong security assurance levels and hence it could not be applicable for safety-critical automotive applications. The main driving force behind the choice of commercial COQOS hypervisor was that it fulfilled all the requirements set by us for the prototype model of multipurpose ECU. It is specifically targeted for the automotive industry and has good support for paravirtualization of AUTOSAR architecture [12]. Also, it has high-security assurance level since it complies with the stringent security and safety standards [26]. However, the main limitation is that COQOS is a closed source, proprietary software solution and hence our design and implementation strategy is bounded to the options and techniques available in the COQOS solution. We studied the concepts and technologies available in COQOS solution and applied these concepts to the implementation of our prototype model considering our problem objectives and requirements.

9.1.1 Porting of AUTOSAR and Linux

COQOS has good support for porting the basic AUTOSAR architecture. It provides POSIX APIs to implement the wrapper functions for the OS abstraction layer to port the standard AUTOSAR stack on top of the PikeOS microkernel [12]. These wrapper functions were used to emulate the standardized AUTOSAR OS interfaces in PikeOS and then the standard AUTOSAR BSW, RTE and SWC could be ported on top of the PikeOS microkernel. The reference AUTOSAR application in our prototype model utilized only the basic set of AUTOSAR OS features and hence we were able to translate the OSEK interfaces used by the AUTOSAR OS functions, in PikeOS microkernel using OSAL wrapper. However, most AUTOSAR applications use extended OS functions, and it is important to check if it is possible to the port the complete state of the art AUTOSAR architecture on top of PikeOS microkernel. To achieve this, PikeOS needs to provide support for emulating the complete set of standardized OSEK OS interfaces. We faced some incompatibility issues while integrating the code files of the standard AUTOSAR RTE module and the proprietary OS (PikeOS) module, and then generating the complete system executable. We were able to solve this issue by synchronizing the OS configuration files and then customizing build process for the integration of RTE module from standard AUTOSAR stack and OS module from PikeOS and then generating the code files for them [41].

We used the proprietary Linux distribution (ELinOS) [31] provided by COQOS to create the virtualized Linux OS with the latest kernel version (2.6.35) and a minimal set of root file system. We ported a lightweight Linux partition with minimal memory footprint for the quick startup of Linux partition in PikeOS system.

9.1.2 Resource partitioning

We ensured spatial isolation between the AUTOSAR and Linux partition by static allocation of I/O peripheral devices to user partitions and memory segmentation technique. The AUTOSAR and Linux partitions were allocated fixed, static set of memory segments instead of a global, dynamic memory. As a result, the kernel and RAM memory requirements of AUTOSAR and Linux needs to be calculated beforehand, and it can only utilize the allocated memory segments at runtime [26]. For real-time OS, it is relatively easy to estimate the memory requirements. However for General Purpose OS (GPOS) such as Linux and Android that host resource-intensive infotainment applications, it can cause minor performance impact for HMI display and buffering of the video and audio systems. Even though, this approach is quite inflexible sometimes, it can guarantee strict separation between the trusted and untrusted applications in different partitions. We used a resource partitioning mechanism with minimal memory footprint and static allocation of devices to the user partitions to optimize the boot time of the prototype model. In our prototype model, there were only two or three devices that need to be shared between the partitions and these were allotted to the system partition. However, in some cases, there will be several I/O devices that need frequent shared access between the partitions. There will be some latency since we use the system partition as proxy, and the tasks need to pass through the additional PikeOS hypervisor layer to access the shared devices at runtime. Hence, there will be an additional delay when applications access the shared devices via system partition when compared to accessing them natively. Hence, efficient shared device management techniques such as mixed criticality locks or non-blocking wait-free

protocols [47, 48] should be deployed in order to guarantee a minimum latency for the real-time applications.

9.1.3 Inter-partition communication

PikeOS offers shared memory segments for transferring data signals. However, it will introduce latency since it needs application specific protocols for the synchronization of read and write access between the partitions [29]. Sometimes it may even take more than one to two seconds for the transfer of signals between the applications due to the synchronization and waiting delay for reading and writing the signals. In our prototype implementation, the inter-partition communication was realized using PikeOS sampling ports mechanism. As a result, there was no synchronization or blocking delay, and we could achieve a fast communication rate of 30 to 40 milliseconds (ms) for the transfer of signals between the AUTOSAR and Linux applications. However there is a limitation on the number of signals that could be transmitted between two applications using the sampling ports mechanism. We also showed how the PikeOS inter-partition mechanism between AUTOSAR and Linux partition can be modeled analogous to the AUTOSAR sender receiver communication mechanism between 2 SWCs, using PikeOS sampling ports and channels.

9.1.4 Time partitioning

In our prototype model, time scheduling requirement was not complicated, and we implemented a simple time partitioning scheme consisting of fixed time duration slots for AUTOSAR and Linux partitions. However, certain automotive applications need dynamic scheduling mechanisms. For Ex: A safety-critical application that needs to be available quickly requires the complete CPU time exclusively for itself while starting up, and after it is initialized, it can share the CPU time with other applications. PikeOS offers advanced time partitioning schemes to support these dynamic scheduling mechanisms, and it could be considered for the complex time scheduling requirements of automotive ECUs [27, 29].

9.2 Reflections on Evaluation and Results

We analyzed the parameters of the prototype model like boot time, message transmission delay and isolation between the applications. The expected boot time of the automotive applications should be within 4 to 8 seconds [43]. We optimized the boot time of the prototype model by allocating minimal memory footprint for the partitions and static allocation of devices to the user partitions instead of allocating all the peripheral devices in the system partition. Thus, we could achieve a boot time of nearly 2.5 seconds for our proposed prototype model of multipurpose ECU system. In the state of the art vehicular architecture, CAN communication protocol was used for the transfer of message signals between different ECU nodes. In this protocol, the delay for the message communication would vary considerably between 100 to 1000 milliseconds depending on the traffic load in the network and the distance that the message needs to traverse [44]. The sampling ports in PikeOS provide a fast communication rate compared to the state of the art CAN communication networks and since the communication interface is localized on the same hardware platform, the delay for the inter-partition communication was reduced to 30 to 40 milliseconds (ms) in our prototype model. The isolation between the applications was tested by simply rebooting

one of the partitions and checking the execution behavior of the application in the other partition. We could observe that there was no impact on the other partition when one of the user partitions was rebooted or halted continuously. Thus we can infer that we have achieved good temporal and spatial isolation between the diverse applications executing in the same hardware system by our proposed design and configuration in the prototype model of multipurpose ECU system.

9.2.1 Proposed measurement tests for further evaluation of the prototype model

According to Alexandra Aguiar and Fabiano Hessel, virtualization solution should guarantee reliability and fault isolation to safety-critical embedded systems [28]. The basic idea behind system partitioning is that if a fatal error occurs in a partition, it should be localized within that partition, and it should not impact the underlying hypervisor layer and other partitions [28]. In order to test this concept completely, further comprehensive fault injections tests such as buffer overflow and denial-of-service attacks should be carried out. Also, fault testing for the inter-partition communication layer between the partitions could not be carried out due to time constraints. It was also important to measure the additional latency and overhead caused by the virtualization abstraction layer when compared to the native system since automotive companies compete mainly on cost/performance ratio. The GPU performance of the HMI application could be evaluated by measuring the frame rates of the HMI display using GPU performance tools. This would show the impact of the additional hypervisor layer overhead on the GPU performance. Likewise, the jitter for the CAN frames needs to be computed by measuring the time difference between frame transmission from the CAN network simulator tool and arrival time at the SWC entity in AUTOSAR application. These jitter measurements should be carried for various transfer rates such as 10, 100 and 1000 milliseconds to determine the average-case jitter and latency due to the processing time overhead introduced by the additional hypervisor abstraction layer. These measurement tests were planned initially but could not be carried out in our thesis work. The COQOS product license was expensive, and we could utilize it only for a limited duration i.e., one month and hence we could not carry out the above measurement tests due to time constraints.

As explained above, we integrate and suitably apply the knowledge derived from several related areas like real-time systems, operating system concepts, virtualization technology and automotive systems for the design and implementation of the prototype model of multipurpose ECU system. In our thesis work, we utilize the proven concept of virtualization technology, which has already been deployed in other domains, into automotive systems. We apply the virtualization technology concepts for designing and implementing our proposed prototype of multipurpose ECU for resolving the integration problem in automotive systems. The knowledge from operating system architecture was utilized for porting the AUTOSAR architecture based on OSEK OS, and Linux OS on top of the hypervisor layer. The concepts from real-time systems were deployed in implementing the time partitioning and resource partitioning mechanism for our proposed prototype model using hypervisor technology. Also, basic comprehension of the automotive domain was required for setting the requirements and for the high-level design of the prototype model of ECU system.

9.3 Selected extension topics

We provide a brief overview about selected possible extensions that could be explored in future based on our thesis work. We discuss the research improvements and necessary engineering measures required so that virtualization solution can gradually find its way successfully into the real world automotive platforms in the near future.

9.3.1 Multicore processors and Virtualization

In order to truly unlock the benefits of parallelism using multicore processors, we need to dynamically manage and allocate the cores to different partitions with diverse configurations like Symmetric multiprocessing (SMP), Asymmetric multiprocessing (AMP) and a mix of both depending on the system load conditions at runtime and priority levels of applications in the multipurpose ECU [49,50]. The virtualization layer provides the above capabilities for flexible scheduling of cores to different partitions [50]. Our prototype model consists of a dual partitioned ECU system running real-time and non-real-time applications, and it was implemented on a single processor platform.

Using multicore processors, the hypervisor solution can implement either of the below mechanisms to improve performance. Also, the partitions can be securely isolated from each other by using different cores that in turn can increase reliability [37, 51].

a) Statically allocate several cores to the real-time OS partition and a single core to non-real-time, general purpose OS partition during system configuration at startup.

b) Dynamically allocate cores to OS partitions depending on their priority levels or demand for processor utilization at runtime.

c) Use a mixed configuration in a tri-core processor, where two cores are statically allocated to the OS partitions at system startup and one core is dynamically scheduled across the partitions during runtime depending on a partition's privilege level or processor utilization.

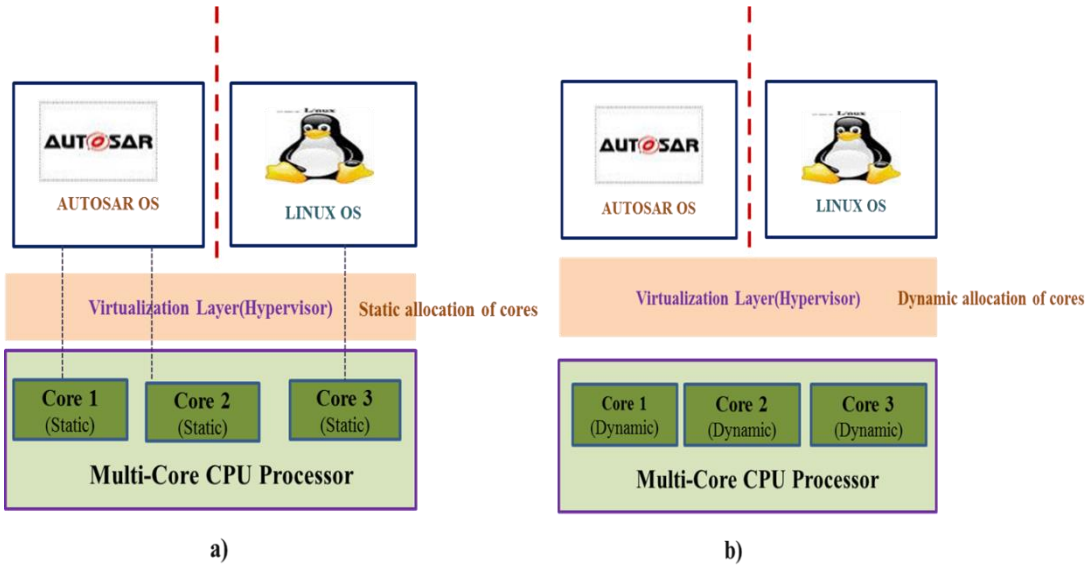


Figure 9.1 Different strategies for allocation of cores using hypervisor a) Static allocation of cores at Initialization and b) Dynamic allocation of cores at Runtime.

9.3.2 Fault monitoring and energy efficiency in Virtualized automotive ECUs

Fault detection, fault management and fault recovery are required to ensure the correctness of safety-critical, automotive applications in a multipurpose ECU system. As a result, the functionality of the health monitoring and system watchdog monitor applications in the current hypervisor solution should be enhanced. Hypervisors should implement an advanced, configurable watchdog as system application that performs runtime monitoring of the automotive applications and implement corrective actions if the system does not function correctly. It should check if the state transitions or runtime behavior of the executing system are in line with the requirement specification properties and state information of the system specified at design time [50]. It should take suitable corrective actions in case of faults and maintain the fail-operation mode even in the presence of critical errors. The watchdog application should also verify the timing properties of interrupts and task scheduling mechanism of the system to guarantee timing guarantees. This watchdog monitor can run on a separate core and the automotive applications to be monitored are implemented on other cores in a multicore ECU system to ensure that there is no performance impact on the automotive applications due to computational overhead of the watchdog monitor application [50].

Likewise, power monitoring and energy efficiency is crucial due to the limited battery supply available in the automotive ECU systems. The watchdog system application can monitor the runtime energy consumption and based on that, corrective actions such as temporarily turning the idle software units to sleep or hot standby mode and dynamic voltage scaling can be carried out in the multipurpose ECU system[52].

9.4 Summary and Conclusions

Automotive OEMs and manufacturers face multitude of technical and financial challenges due to the proliferation of core automotive control and infotainment functionalities in vehicular systems, and the increasing necessity for providing integration and interaction between these heterogeneous functionalities for supporting the next generation vehicular technologies [3, 9]. In this thesis work, we look at the integration problem of consolidating heterogeneous automotive applications, running on top of AUTOSAR architecture and Linux OS, in separate ECU platforms. We propose a pilot implementation of a multipurpose ECU system, in which AUTOSAR and Linux applications could be concurrently executed on a shared hardware platform, to solve the integration problem. Initially, we set the requirements for this proposed pilot implementation of multipurpose ECU system. Then our thesis work explores several consolidation strategies based on these requirements and finally chooses the strategy of using embedded virtualization technology to solve the integration problem in automotive ECU systems. We demonstrate this concept by proposing the design along with corresponding implementation for a prototype model of multipurpose ECU executing real-time control and non-real-time infotainment applications simultaneously, on top of AUTOSAR architecture and Linux GPOS respectively in a shared hardware platform using a virtualization solution. In this report, we present our chosen consolidation approach using a commercial virtualization solution named as COQOS and it is based on PikeOS microkernel [12, 26]. Then we chose a suitable hardware platform, automotive control and infotainment reference applications based on AUTOSAR architecture and Linux OS respectively for the prototype model. The standard AUTOSAR stack was ported on top of the PikeOS microkernel virtualization

layer by implementing necessary wrapper functions in the OS abstraction layer (OSAL) to emulate the standardized OSEK interfaces in PikeOS microkernel [5]. Linux GPOS could be easily ported on top of PikeOS microkernel without much effort by utilizing the Linux distribution provided by hypervisor [31]. Then partitioning and sharing of system resources such as memory and I/O peripheral devices, CPU time sharing between AUTOSAR and Linux applications were configured using the COQOS hypervisor. Intersystem communication was also implemented to exchange data signals between AUTOSAR and Linux application executing in different partitions. The techniques presented by COQOS hypervisor were applied in the context of the design and implementation of our proposed prototype model of multipurpose ECU to satisfy the nonfunctional automotive requirements like fast bootup time, less signal transmission delay and strict isolation between the applications. The subcomponents of the prototype model were configured separately and then integrated into a single system image using the COQOS development and compilation tools [30].The PikeOS system image was loaded in the target hardware platform and we were able to execute the two reference applications concurrently in the selected hardware platform using COQOS hypervisor, and the application in different partitions were able to intercommunicate and transfer data signals between them.

Finally, we carried out a basic testing of the prototype model for measuring and evaluating the nonfunctional parameters like boot time, message transmission delay and isolation property and then the results were studied by a comparative analysis with related existing models and current state of the art system. The analysis of our measurement results shows that our proposed prototype model achieves better or equivalent performance when compared to the current state of the art ECU systems. Thus, we have demonstrated that it is possible to integrate the automotive control application modules based on AUTOSAR architecture with infotainment applications based on Linux OS, and they can share the system resources such as memory, I/O devices and CPU processor time on the same ECU hardware platform using the hypervisor solution. Also, the intercommunication between automotive control function and infotainment application can be localized in the same ECU hardware platform instead of using complicated in-vehicle communication networks. Thus in our thesis work, we were able to demonstrate that virtualization technology can be utilized to implement the seamless integration between classical automotive control systems and the advanced infotainment systems in the future car technologies.

However, it should be kept in mind that this is a single case study, and the prototype system is quite limited. Also, it is critical to carry out a comprehensive analysis and validation of the prototype system to guarantee that it meets the automotive performance and safety requirements. We could not carry out a full scale evaluation and further refinement of the prototype model due to lack of time, and financial constraints. The deployment of virtualization solution in next generation vehicular ECU systems is becoming inevitable and has already generated active interest among automotive manufacturers [13, 24]. We envision that our work and results presented in this report will serve as a useful stepping stone for exploring further research avenues in this fledgling technical field. We expect that the outcome of this thesis work will provide valuable insights for further research study, focusing in depth on the various technical aspects discussed in this report and will pave way for implementing virtualization technology in automotive embedded domain in near future.

References

[1] F. Simonot-Lion and Y. Trinquet, “Chapter 1. Vehicle Functional Domains and Their Requirements”, in *Automotive Embedded Systems Handbook*, Ed. 2008, Industrial Information Technology series, CRC Press.

[2] EU - European Commission transport council, “DIRECTIVE- 2010/40/EU OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on the framework for the deployment of Intelligent Transport Systems”, 7 July 2010. [Online]. Available at: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2010:207:0001:0013:EN:PDF

[3] A. Hergenhan and G. Heiser, “Operating systems technology for converged ECUs,” in 6th Embedded Security in Cars Conference (escar), ISITS, Nov 2008, Hamburg, Germany.

[4] *AUTOSAR ProjectObjectives*, R4.0 V3.0 Rev 3, AUTOSAR, December 2011, [Online]. Available at: http://www.autosar.org/fileadmin/files/releases/4-0/main/auxiliary/AUTOSAR_RS_ProjectObjectives.pdf

[5] *Specification of Operating System*, R4.0 V5.0 Rev 3, AUTOSAR, November 2011, [Online]. Available at: http://www.autosar.org/fileadmin/files/releases/4-0/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf

[6] *OSEK - Operating System Specification*, Version 2.2.3, OSEK/VDX, February 2005, [Online]. Available at: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf

[7] IHS Automotive, “Linux to Take the Lead in Automotive Infotainment Operating System Market”, Southfield, Michigan, November 2013, [Online]. Available at: http://press.ihs.com/press-release/design-supply-chain-media/linux-take-lead-automotive-infotainment-operating-system-mar

[8] J. Fornaeus, “Device hypervisors”, in Proceedings of the 47th Design Automation Conference, (DAC), 13-18 June 2010, ACM/IEEE, Anaheim, California, USA, pp. 114 - 119.

[9] R. Bishop, *Intelligent Vehicle Technology and Trends*, Norwood, MA: Artech House, 2005.

[10] A. Sangiovanni-Vincentelli, “Automotive electronics: Trends and challenges”, in Convergence, October 2000, Detroit, MI, USA.

[11] D. Kleidermacher, “Chapter 7: System Virtualization in Multi-core Systems”, in *Real World Multi-core Embedded Systems*, Ed. March 2013, Newnes, pp. 227 - 267.

[12] OpenSynergy GmbH , “COQOS Operating System-Datasheet”, 2013, [Online]. Available at: http://www.opensynergy.com/fileadmin/user_upload/Datenblaetter/COQOS-Datasheet.pdf

[13] J. Pelzl, M. Wolf, T. Wollinger, “Virtualization Technologies for Cars - Solutions to increase safety and security of vehicular ECUs”, in Automotive - Safety and Security, 19 - 20 November 2008, Stuttgart, Germany.

[14] A. Groll, J. Holle, M. Wolf, T. Wollinger, “Next Generation of Automotive Security: Secure Hardware and Secure Open Platforms”, in 17th ITS World Congress (ITS Busan), 25 – 29 October 2010, Busan, Korea.

[15] F. Stumpf, C. Meves, B. Weyl, M. Wolf, “Security Architecture for Multipurpose ECUs in Vehicles”, in 25th Joint VDI/VW, Automotive Security conference, 19 – 20 October 2009, Ingolstadt, Germany.

[16] K. Peffers, T. Tuunanen, M.A. Rothenberger and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," in *Journal of Management Information Systems*, 2007, pp. 45 - 78.

[17] V. Vaishnavi, B. Kuechler "Design Research in Information Systems", 2005. [Online]. Available at: http://www.desrist.org/design-research-in-information-systems

[18] *Demonstration of Integration of AUTOSAR into a MM/T/HMI ECU*, V1.0.0, AUTOSAR, November 2010. [Online] Available at: https://www.autosar.org/fileadmin/files/releases/AUTOSAR_TR_IntegrationintoMMTHMIECU.pdf

[19] Z. Gu, Z. Wang, S. Li, H. Cai, "Design and Implementation of an Automotive Telematics Gateway based on Virtualization", in Proceedings of the 2012 IEEE 15th International Symposium on Object Component Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 11 April 2012, IEEE Computer Society, Washington, USA, pp. 53 - 58

[20] Y. Li, R. West, and E. Missimer, "The quest-v separation kernel for mixed criticality Systems", in Proceedings of the 1st International Workshop on Mixed Criticality Systems, (WMC), Real-Time Systems Symposium (RTSS), 3 December 2013, pp. 31 - 36.

[21] D. Reinhardt, D. Kaule, and M. Kucera, "Achieving a Scalable E/E-Architecture Using AUTOSAR and Virtualization", in SAE International Journal of Passenger Cars- Electronic and Electrical Systems, April 2013, pp. 489 - 497.

[22] R. Kaiser, "Chapter 15. Applicability of Virtualization to Embedded Systems", in *Solutions on Embedded Systems*, Ed. November 2011, Springer, LNEE - Lecture Notes in Electrical Engineering, pp. 215 - 226.

[23] G. Heiser, "The role of virtualization in embedded systems", in Proceedings of the 1st workshop on isolation and integration in embedded systems (IIES), April 2008, ACM, New York, USA, pp. 11 - 16.

[24] M. Strobl, M. Kucera, A. Foeldi, T. Waas, N. Balbierer and C. Hillbert, "Towards automotive virtualization", in International Conference on Applied Electronics (AE), IEEE, 10-12 September 2013, pp. 1 – 6.

[25] A. Singh, "An Introduction to Virtualization", January 2004. [Online]. Available at http://www.kernelthread.com/publications/virtualization

[26] R. Kaiser, S. Wagner, "Evolution of the PikeOS Microkernel", in Proceedings of the 1st International Workshop on Microkernels for Embedded Systems (MIKES), January 2007, National ICT Australia (NICTA), Sydney, NSW, Australia.

[27] R. Kaiser, "Alternatives for Scheduling Virtual Machines in Real-Time Embedded Systems", in Proceedings of the 1st workshop on isolation and integration in embedded systems (IIES), April 2008, ACM, New York, USA, pp. 5 - 10.

[28] A. Aguiar and F. Hessel, "Embedded systems' virtualization: The next challenge?", in 21st IEEE International Symposium on Rapid System Prototyping (RSP) , 8-11 June 2010, Fairfax, VA, USA, pp. 1 - 7.

[29] *pikeOS - PikeOS Fundamentals*, Version S3136-1.8, PikeOS v3.2, © Copyright 2005 – 2012, SYSGO AG, 2012, Klein-Winternheim, Germany.

[30] *pikeOS - Using PikeOS*, Version S3136-1.11, PikeOS v3.2, © Copyright 2005 – 2012, SYSGO AG, 2012, Klein-Winternheim, Germany.

[31] *ELinOS – A Development Environment for Embedded Linux Applications*, 14th Edition, © Copyright 2001 - 2010, SYSGO AG, October 2010, Klein-Winternheim, Germany.

[32] *AUTOSAR Technical Overview*, R3.2 V2.2.2 Rev 1, AUTOSAR, April 2011, [Online]. Available at: https://www.autosar.org/fileadmin/files/releases/3-2/main/auxiliary/AUTOSAR_TechnicalOverview.pdf

[33] *Specification of RTE*, R4.0 V3.2.0 Rev 3, AUTOSAR, October 2011, [Online]. Available at: https://www.autosar.org/fileadmin/files/releases/4-0/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf

[34] *Specification of VFB*, R3.2 V1.3.0 Rev 2, AUTOSAR, May 2012, [Online]. Available at: https://www.autosar.org/fileadmin/files/releases/3-2/main/auxiliary/AUTOSAR_SWS_VFB.pdf

[35] D. Haworth, *An AUTOSAR-compatible microkernel for systems with safety-relevant components*, Herausforderungen durch Echtzeitbetrieb, Informatik aktuell, Heidelberg,© Springer-Verlag Berlin, 2012.

[36] S. Faucou, P. Emmanuel Hladik, A. Marie.Déplanche, and Y. Trinquet "Overview of microkernel standards for real-time in-vehicle embedded systems", in Taiwanese-French Conference on Information Technology (TFIT), March 2008, National Taiwan University, Taipei, Taiwan, pp. 28 – 39.

[37] K. Senthilkumar, R. Ramadoss, "Designing Multi-core ECU Architecture in vehicle networks using AUTOSAR", in Third International Conference on Advanced Computing (ICoAC), IEEE, 4-16 December 2011, Chennai, India, pp. 270 – 275.

[38] MECEL AB, "Product brief – Mecel Picea Suite – Datasheet", 2013, [Online]. Available at: http://www.mecel.se/products/mecel-picea/Product.Brief.Mecel.Picea.pdf

[39] MECEL AB, "Product brief – Mecel Populus Suite – Datasheet", 2013, [Online]. Available at: http://www.mecel.se/products/mecel-populus/Product.Brief.Mecel.Populus.pdf

[40] Freescale, "i.MX53 SABRE Platform for Automotive Infotainment - Factsheet", 2013, [Online]. Available at: http://www.freescale.com/files/32bit/doc/fact_sheet/IMX53SBRAUTFS.pdf

[41] *AUTOSAR Methodology*, R4.0 V2.1.0 Rev 3, AUTOSAR, November 2011, [Online]. Available at: http://www.autosar.org/fileadmin/files/releases/4-0/methodology-templates/methodology/auxiliary/AUTOSAR_TR_Methodology.pdf

[42] J.E. Urban, "Software Prototyping and Requirements Engineering", Rome Laboratory, Rome, NY, June 1992 [Online]. Available at: https://www.thecsiac.com/sites/default/files/SW%20Prototyping%20and%20Requirements%20Engineering.pdf

[43] D. Stolarz, "Reducing OS Boot Times for In-Car Computer Applications", 22 April 2004, [Online]. Available at: http://www.linuxjournal.com/article/7544

[44] R.I. Davis, A. Burns, R.J. Bril, and J.J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised", Real-Time Systems, April 2007, pp. 239 - 272.

[45] E. Brown, "Linux boots in 2.97 seconds", 7 November 2008, [Online]. Available at: http://archive.linuxgizmos.com/linux-boots-in-297-seconds

[46] AllGo Embedded Systems, "AllGo Systems announces Android Automotive IVI system solution with 4 seconds boot up time", October 2013, [Online]. Available at: http://www.allgosystems.com/for_release_oct_2013.php

[47] J. Schneider, "Overcoming the Interoperability Barrier in Mixed-Criticality Systems", in Proceedings of the 19th ISPE International Conference on Concurrent Engineering (CE), September 2012, Trier, Germany.

[48] M. Herlihy, “Wait-free synchronization”, ACM Transactions on Programming Languages and Systems (TOPLAS), January 1991, pp. 124 - 149.

[49] N. Böhm, D. Lohmann, and W. Schröder-Preikschat, “Multi-Core Processors in the Automotive Domain: An AUTOSAR Case Study”, in Proceedings of the Work-in-Progress Session of the 22nd Euromicro Conference on Real-Time Systems (ECRTS), 6-9 July 2010, Brussels, Belgium, pp. 25 - 28.

[50] J. Makitalo, “RECOMP-Deliverable D3.1: Virtualization: Requirement definition and Mechanism description”, ARTEMIS-2009-1, Second Project Periodic Report, 2012.

[51] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion, “Multi-source and Multicore Automotive ECUs-OS Protection Mechanisms and Scheduling”, in Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE), 4-7 July 2010, Bari, Italy, pp. 3734 - 3741.

[52] R. Zalman, B. Böddeker, “AUTOSAR at the Cutting Edge of Automotive Technology”, in Proceedings of the 7th International Conference on High-Performance and Embedded Architectures and Compilers (HIPEAC), January 2012, Paris, France.

[53] R. Rosen, “Resource management: Linux kernel Namespaces and cgroups”,
Haifux, May 2013, [Online]. Available at: http://www.haifux.org/lectures/299/netLec7.pdf

[54] D. Petrovic, A. Schiper, "Implementing Virtual Machine Replication: A Case Study Using Xen and KVM," in IEEE 26th International Conference on Advanced Information Networking and Applications (AINA), 26 - 29 March 2012, pp. 73 - 80.

[55] N. Franssens, J. Broeckhove, P. Hellinckx, "Taxonomy of Real-Time Hypervisors," in Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 28-30 October 2013 , pp. 492 – 496.

[56] S. Xi, J. Wilson, C. Lu and C. Gill "RT-Xen: Towards real-time hypervisor scheduling in Xen," in Proceedings of the International Conference on Embedded Software (EMSOFT), 9-14 October 2011, pp. 39 - 48.

[57] A. Kivity, “KVM: The Linux Virtual Machine Monitor”, Proceedings of the Linux Symposium, 2007.

[58] A. Iqbal, N. Sadeque, and R. I. Mutia, "An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems, Technical Report, Department of Electrical and Information Technology, Lund University, Sweden, 2009, [Online]. Available at:
http://www.eit.lth.se/fileadmin/eit/project/142/virtApproaches.pdf

[59] Wind River, “Wind River Hypervisor product description”, 2013, [Online]. Available at:
http://www.windriver.com/products/product-notes/wind-river-hypervisor-product-note.pdf

[60] G. Heiser, B. Leslie, “The OKL4 microvisor: convergence point of microkernels and hypervisors”, in Proceedings of the first ACM Asia-Pacific Workshop on systems, 30 August 2010, New Delhi, India.

Appendix A – Proposal for Master Thesis

(i) **Background:**

Embedded systems or ECU units in the automobiles have been increasing in complexity and they support real-time automotive control applications as wells as non-real-time applications like Infotainment, HMI and Navigation assistant systems. Normally the real- time tasks are executed by AUTOSAR based RTOS and non-real-time processes are executed by general purpose OS like Android/Linux/Windows etc. Currently both the OS have different requirements and they do not share resources like processor and memory.

(ii) **Purpose:**

The purpose of this Master Thesis is to combine an AUTOSAR OS with a Linux OS on the same target hardware, single chip solution. Since the AUTOSAR OS and Linux OS have different requirements a virtualization solution (hypervisor) is required. The Master Thesis shall implement a demo system containing a reference implementation of AUTOSAR OS, Linux OS and Hypervisor on a target hardware fulfilling automotive requirements.

(iii) **Description:**

This Master Thesis contains a theoretical part to learn more about; AUTOSAR, Linux, Hypervisor, Tool chain/Debugger and Target hardware.

Definition of the OS and hypervisor solution to be used is done on the first phase of the Master Thesis. Requirement analysis is done to map the automotive requirements with theoretical solutions and thereafter have software architecture. Software design containing configuration of the system including Inter-system communication between AUTOSAR OS and Linux OS is done.

Implementation of demo system is done on pre-defined target hardware. This includes making sure that the hardware is configured and setup properly. The OS are working accordingly and that necessary demo application(s) is developed to demonstrate features/functions for performance measurements. Carry-out measurements on the demo system and map the measurement results against automotive requirements. Finally we analyze and comment on the results. There will also be some generic and customer specific automotive requirement specifications available for this kind of ECU (System).

(iv) **Limitations:**
The thesis is not intended to do the complete integration of the AUTOSAR architecture part with Linux applications. It will only implement a pilot project of basic demo applications with the integration of AUTOSAR and Linux OS on top of the hypervisor solution as a proof of concept. Also this thesis project is dependent upon the availability of the target hardware platform and the hypervisor solution.

Appendix B – Tasks List for Master Thesis

Tasks for the Master Thesis are divided into two phases namely the **Research study phase** and **Implementation phase**.

a) Phase 1 – Research Study Phase

1. Investigation/Survey on selection of the hypervisor product solution and the target hardware platform.

2. Study about the selected hypervisor product solution and the target hardware platform and understand their functionality.

3. Study about OSEK OS/AUTOSAR interfaces and understand how to integrate and interface it with the hypervisor abstraction layer.

4. Survey on Linux kernel and distribution and understand how to integrate it with the hypervisor.

5. Explore and understand how AUTOSAR and Linux OS communicate with each other using the Inter OS communication mechanism provided by the hypervisor product solution. Analyze and study the interfaces and libraries provided by the hypervisor solution.

6. Study about the automotive performance requirements like fast boot up and quick loading and how it is done in the current implementation.

7. Complete the interim report for the Phase-1.

b) Phase 2 – Design and Implementation Phase

1. Design software architecture for the integration of the AUTOSAR & Linux with the hypervisor solution.

2. Prepare the development environment, debugging methods and run the hypervisor on the intended target hardware.

3. Select the suitable sample demo applications. Develop the demo applications and execute it on the virtualized platform and check their functionality as proof of concept.

4. Integrate the AUTOSAR OS and Linux by using the interfaces of the hypervisor solution. Develop the communication module that allows AUTOSAR & LINUX OS to communicate the data signals with each other on top of the hypervisor abstraction layer.

5. Carry-out basic measurement testing of the demo system to ensure that it conforms to the standard automotive requirements and analyze the measurement results.

6. Reflect and evaluate the measurement results and compare it with the current state of the art ECU system and related automotive ECU models.

7. Testing, Optimizing and Refactoring the solution (optional and if necessary).

Appendix C – Software products and Tools used in our Master Thesis

The following software products and tools were used during the course of our Master thesis-work.

a) AUTOSAR development tools

1. MECEL PICEA SUITE

This was developed by MECEL AB and it is a comprehensive solution for effective development and seamless integration of the AUTOSAR compliant ECUs. It was mainly developed for automotive ECU suppliers. It contains the AUTOSAR basic software modules, RTE, tool for configuring the RTE, BSW modules and editing the application software components (SW-C). PICEA SUITE provides complete eclipse based tool-chain support to integrate the various description elements together into an ECU application system. The configuration of BSW modules and RTE, as well as the implementation of the Application SWCs can be done effectively using the Picea suite. Then the code and software executable of RTE and BSW for the ECU can be built using the generator tools and make-file build system. Then Application SWCs can be integrated on the automotive ECU system.

Source: http://en.wikipedia.org/wiki/Mecel

2. Volcano VSx tool-suite

This is an integrated tool suite developed by Mentor Graphics, for the top-down automotive system and ECU design. It provides automotive software and electronic systems design, development, integration and verification. This tool-suite is built upon Eclipse framework and it enables the design of the software and hardware architecture of an AUTOSAR system, mapping of the application software components to ECUs and the design of the communication networks between the different ECUs. It can generate the AUTOSAR ECU extract and other AUTOSAR ECU system configuration files.

Source:
http://s3.mentor.com/public_documents/datasheet/products/vnd/VolcanoAUTOSAR_solutions.pdf

b) HMI development tools

Mecel Populus HMI Suite is a complete package-tool developed by MECEL AB for designing, implementing and deploying state-of-the-art, HMI based user interfaces for automotive and distributed embedded systems. It ensures effective software life-cycle management while minimizing the time and cost associated with system engineering and verification of HMI applications. The brief description of functional components of Populus suite can be found below.

(i) **Populus Editor:** The Populus editor is a window application used for specifying the complete graphical layout, visual and the HMI flow/logic of the user interface. It creates the HMI database which controls all the aspects of HMI display Unit like screen layout settings, animation effects, text-entries, handling user inputs and reacting to events. Using the HMI database, it also configures behavior of Populus engine, the run-time software component, which interacts with the FU applications and generates the graphical interface on the target display unit. Populus editor also generates FU interface class and FIL which enables the engine to interact and get the data from the FU application

(ii) **Functional Unit:** Functional Unit defines the actual business-logic implementation code for the HMI application. It can be located internally in the same ECU system or distributed in external ECU. FU communicates with the Populus engine via a communication protocol known as ODI and periodically notifies the indications and changes to update HMI Unit. Populus editor automatically generates the FIL (Java or C/C++ interfaces) using the FU-SDK libraries which takes care of implementing the ODI protocol and enables the interaction between the decoupled HMI display and FU application modules. In addition, the FU modules are completely re-usable.

MECEL Populus Suite handles efficient and complete software life-cycle management of HMI application project like design, development, testing and integration into the target system. Also late HMI and FU changes can be easily added.

Source: http://en.wikipedia.org/wiki/Mecel

c) LTIB Tool-Chain

LTIB (Linux Target Image Builder) project is a simple tool that can be used to develop and deploy BSPs (Board Support Packages) for various target platforms and processor architectures like PowerPC, ARM, Freescale, etc. Using this tool, the developer will be able to develop a GNU/Linux image for the specific target platform. It supports building of root file system, boot-loader and Linux kernel image for multiple target platforms and the developer can choose different system configurations (e.g. toolchain selection, kernel selection, package selection, etc.) during the build process.

Source: http://savannah.nongnu.org/projects/ltib

d) COQOS Virtualization solution development tools

1. CODEO

CODEO is the PikeOS IDE (Eclipse based) used for application development, system configuration, integration and deployment of PikeOS projects in embedded target platform. It also provides project configurator editor, a special graphical editor for project definition and system configuration properties like resource and time partitioning, application scheduling and defining the properties of inter-partition communication channels and ports. The CODEO IDE also provides capabilities like debugger for debugging the PikeOS application projects in the target environment using GDB tools and tracer analyzer tool for adding trace points in each partition and then analyze the response time, latency, CPU time and scheduling behavior, etc. Trace events are displayed as icons in the CODEO Trace Tool oscilloscope. It also provides MONITOR tool for examining the partition information and system parameters. There is a CPU emulator program called QEMU which can be used for booting the PikeOS application projects virtually in the host PC without the real hardware [29, 30].

Source: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/

2. ELinOS

ELinOS is a development environment for building embedded Linux distribution and is provided by SYSGO AG. ELinOS offers all the features used in industrial real-time applications and supports a broad range of BSPs, target boards, processor architectures, drivers and real-time hardware extensions. It provides the latest kernel version and is used to create the virtualized Linux OS that can be hosted in a partition on top of PikeOS microkernel. ELinOS is mainly used for industrial applications and the CODEO IDE can be used as the application development environment for creating the embedded Linux distribution using ELinOS [30, 31].

Source: http://www.sysgo.com/products/elinos-embedded-linux/

3. MUXA

PikeOS provides a utility named as MUXA tool that can be run on a host PC, and it provides bidirectional virtual channels for remotely connecting and communicating with the system partition in the target platform to monitor the user partitions and applications executed in the target hardware from the host PC. MUXA is a utility which provides console and debugging functionalities for the various personalities/partitions in the PikeOS system. In order to check the system status of the different partitions, Muxa provides a multi-channel multiplexer to have several virtual bi-directional communication channels between the partitions in the embedded target system and the host PC with a single physical link [29, 30].

Using this MUXA tool, it is possible to check the status information of the user partitions and applications in each user partition, inter-partition communication channels, ports and running processes. Also, it is possible to halt or reboot the user partitions and the target system remotely using this MUXA tool. The Muxa utility should be configured and setup on both host side (PC) and target platform side [29, 30].

On the target side, the Muxa application registers as a file provider, allowing applications to access it via set of predefined file functions. On the host side, the host Muxa application listens on the physical link between target and host on one side and provides a command interface and TCP/IP port connections for the channels on the other side. Muxa connections can be used for both binary data exchange and character oriented console communication [29, 30].

e) CANalyzer- Vehicular CAN network simulator tool

CANalyzer is a CAN bus communication simulator and analysis tool provided by Vector Informatik GmbH. This is primarily used by automotive and electronic control unit suppliers to simulate, measure, and then analyze the in-vehicle network communication traffic in automotive systems. Using this tool, we can send and receive the CAN messages from the automotive ECU systems. It supports CAN, FlexRay, MOST as well as LIN communication bus systems.

Source: http://en.wikipedia.org/wiki/CANalyzer