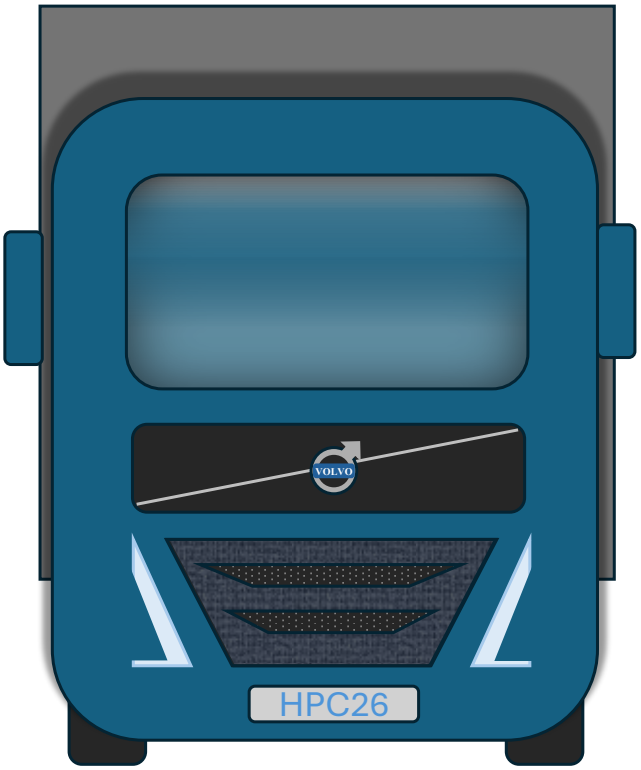
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



An Investigative Study of Software- and Hardware-Based Dependencies in Automotive Software

Master's thesis in Computer science and engineering

Friso de Muinck, Alexandra Lindvall

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Master's thesis 2026

An Investigative Study of Software- and Hardware-Based Dependencies in Automotive Software

Friso de Muinck, Alexandra Lindvall

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

An Investigative Study of Software- and Hardware-Based Dependencies in Automotive Software
Friso de Muinck, Alexandra Lindvall

Supervisor: Ahmed Ali-Eldin Hassan, Computer Science and Engineering
Advisor: Carl Hjerpe, Volvo AB
Examiner: Ahmed Ali-Eldin Hassan, Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

An Investigative Study of Software- and Hardware-Based Dependencies in Automotive Software

Friso de Muinck, Alexandra Lindvall
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern vehicles are increasingly software driven, shifting Electronic Control Unit (ECU) development toward virtualized environments to enable pre-silicon testing and bypass physical hardware bottlenecks. This thesis investigates the portability barriers of retrofitting a production-level Basic Software (BSW) stack from Volvo AB onto a virtualized Infineon TC39x microcontroller using a Synopsys Virtualizer Development Kit (VDK). The success of the virtualization was evaluated against two primary criteria. These were verification of hardware initialization and successful deployment of the Operating System (OS) scheduler.

The investigation revealed a distinct difference between the static initialization and dynamic execution phases of the software. During the static phase, procedural hardware dependencies were successfully mitigated. By developing Python-based workaround scripts to intercept memory-mapped Input/Output (I/O) and bypass simplified hardware ”stub modules” (the Memory Test Unit (MTU) and Power Management System (PMS)), execution time was iteratively extended. Furthermore, a ”warm start” memory injection strategy was implemented, capturing and mapping 7.45 MB of active Program and Data Flash from a physical reference ECU to overcome missing boot-sequence calibrations and diagnostic resets. This low-level register and memory manipulation enabled the virtual target to execute past early hardware checks and successfully initiate the OS on Core 0. However, when entering the dynamic phase at approximately 84 ms of runtime, the system encountered a terminal ”dependency explosion”. As execution transitioned to an asynchronous, event-driven environment, the OS kernel stalled due to missing timing-synchronized feedback from the Generic Timer Module (GTM) and Controller Area Network (CAN) controllers. This proves that functional stubbing is insufficient for Real-Time Operating System (RTOS) execution without synchronized peripheral simulations. Ultimately, this work documents the ”Retrofitting Trap” of forcing hardware-coupled binaries into virtual environments, highlighting the necessity for an industry paradigm shift toward simulation-aware, modular Hardware Abstraction Layer (HAL) to support scalable, cloud-native validation of software-defined vehicles.

Keywords: Computer, science, computer science, engineering, project, thesis, virtual ECU, Paravirtualization, BSW, hardware dependencies, Infineon AURIX TC39x, real-time operating system, software-defined vehicles.

Acknowledgments

We would like to express our deepest gratitude to our supervisor at Volvo AB, Carl Hjerpe, who, despite his busy schedule, managed to make time for us with support and guiding us to the finish line. A special thank you to the MEGA team at Volvo AB, who spent many hours helping us with debugging and showing us the ins and outs of their software, giving us many useful insights over the course of this thesis.

Thank you to Ahmed Ali-Eldin Hassan, our supervisor and examiner at Chalmers, who has been a continued help. Helping us by steering our report in the right direction and having a continuous dialog with us.

We would also like to thank Synopsys and Infineon, as well as pointing a special thank you to Rohit Kurup, for everything he taught us about the Virtualizer tool set and his patience when answering all of our questions.

Finally, we want to thank our families and friends for their unwavering encouragement during the highs and lows of our journey at Chalmers. Without their support, this work would not have been possible.

Friso de Muinck, Alexandra Lindvall, Gothenburg, 2026-06-04

Contents

List of Abbreviations

Below is the list of abbreviations used throughout the thesis listed in alphabetical order:

ADAS	Advanced Driver Assistance System	7
ARU	Advanced Routing Unit	7
ASIC	Application Specific Integrated Circuit	1
ATOM	ARU-connected Timer Output Module	7
AUTOSAR	Automotive Open System Architecture	2
BBB	Back Bone Bus	6
BMS	Battery Management System	3
BSW	Basic Software	v
BVT	Borrowed Virtual Time	17
CA	Cycle Accurate	10
CAN	Controller Area Network	v
CAN FD	Controller Area Network Flexible Data-rate	9
CGW	Central Gateway	3
CI/CD	Continuous Integration/Continuous Deployment	37
CPU	Central Processing Unit	4
DBT	Dynamic Binary Translation	10
DMA	Direct Memory Access	4
DPLL	Digital Phase Locked Loop	8
DSPR	Data Scratchpad RAM	8
E/E	Electrical and Electronic	5
E-Ray	FlexRay Protocol Controller	9
ECU	Electronic Control Unit	v
EMS	Engine Management System	3
EOL	End-Of-Line	15
GTM	Generic Timer Module	v
HAL	Hardware Abstraction Layer	v
HiL	Hardware the in Loop	5
HSI	Hardware-Software Interface	4
I/O	Input/Output	v
I2C	Inter-Integrated Circuit	10
IA	Instruction Accurate	10
ICU	Interrupt Control Unit	32
ISS	Instruction Set Simulator	10

List of Figures

List of Tables

1

Introduction

Modern vehicles are becoming increasingly software driven, with approximately 80% of vehicle functionality being software based [1]. To manage in-car functionality, a ECU is commonly used. ECUs host complex embedded software systems that control key features and functionality within a vehicle, such as the powertrain, braking, and other safety features, as well as less critical systems such as infotainment. In most ECUs the central component and main processing unit is a microcontroller [2]. Containing both input and output channels a large part of the signal processing within an ECU is executed on the microcontroller in addition to many control algorithms. However, due to the increasing complexity in functionality which is required from ECUs, the standard microcontroller no longer is sufficient as the sole computational component. Therefore, in many modern ECUs, the addition of Application Specific Integrated Circuit (ASIC) modules has become commonplace as a means of offloading computational tasks from the microcontroller in addition to providing the ability to perform more complex data processing and or signal monitoring [2]. As a result of this, the internal complexity of the ECU has drastically increased over the years with the introduction of ASICs, power management modules, watchdog protocols, and more, likewise increasing the microcontrollers dependency on peripheral circuitry and monitoring signals.

Traditionally, ECU software is developed and tested on physical hardware. However, this can create delays during development, as the hardware might not yet be available or specific fault conditions are difficult to reproduce. The need for a Virtual Electronic Control Unit (vECU) therefore becomes apparent to allow the development time within the automotive industry to decrease. This thesis will focus on attempting to successfully simulate a virtual microcontroller through retrofitting an existing production-level BSW. As the microcontroller is the central computing component of an ECU it is surmised that if the production software is successfully executed virtually, the model can be built upon further to create a full vECU in the future.

The increased complexity of embedded systems necessitates new strategies to automate the verification and design of safety-critical architectures [3]. The increased complexity also introduces complications in regards to virtualization and testing as more complex models are required to accurately represent the hardware virtually. Similarly, as the peripheral dependencies for microcontrollers in ECUs escalates, so

does the complexity of simulating the hardware. This has created the need for an approach in which peripheral dependencies can be mitigated by creating simplified models of external circuitry which produce expected inputs, allowing the microcontroller to perform as intended. The approach of simulating hardware partially and not recreating the full functionality is referred to as paravirtualization and allows for virtual hardware to behave similarly to physical hardware without requiring a full virtualization of every peripheral circuit [4].

Additionally, as a result of the lengthy development process of automotive platforms the use of legacy software is not an uncommon occurrence within the automotive industry [5]. Legacy software further increases the difficulty of virtualization, as the software is often developed around specific models and use cases, leading to tight dependencies on specific hardware components. By virtue of legacy software being built upon over large periods of time certain coding practices used throughout the software tend to be outdated [6]. This can lead to modern concepts such as abstractions and modularity, similar to the Automotive Open System Architecture (AUTOSAR) framework, to not be implemented throughout the entirety of a BSW.

Virtualization of such systems requires modification strategies to reflect how to best approach these issues of handling both dependencies related to inputs from peripherals and circuits within the ECU as well as the internal dependencies based on the structure of the software and development practices. It becomes an issue of to what degree paravirtualization and abstraction can offer developers the ability to execute production-level software for an ECU in a virtual environment without trading accuracy compared to the physical counterpart. If successful testing, debugging, and validation can begin earlier in the development process, supporting parallel workflows and faster iteration of the design [7]. It also allows for testing elements that are either too expensive or not possible on physical hardware, such as faults or broken behavior. The thesis therefore, aims to investigate what software- and hardware-based dependencies in automotive legacy software prevent fully transparent virtualization in a virtual microcontroller environment. In order to achieve this, production-level software will be tested in a virtual environment with a virtual model of the TC39x microcontroller, created by Infineon, servering as a baseline for testing.

2

Background

In traditional ECU software development, progress is often limited by hardware availability and the difficulty of reproducing specific error conditions. This slows down development, increases integration time, and makes fault handling tests complex and costly. As the complexity of systems has increased over the years, so has the demand for robust testing solutions. Virtual environments offer a cost-effective way to test software for complex systems early in the design process, before having access to the target hardware. There are several leading platforms for this kind of development, such as Synopsys and QEMU.

2.1 Electronic Control Unit (ECU)

The Electronic Control Unit (ECU) is an umbrella term for a larger electrical system that receives input signals, processes them, and distributes them to the desired destination within a system. The ECU is constructed of different electrical components, depending on the use case, within one housing to create a central component for the specific conditions required. Components such as microcontrollers, Read Only Memory (ROM), Random Access Memory (RAM), ASIC modules, and monitoring modules are some of the typical building blocks for an ECU [2]. A result of the high customization of an ECU is that, depending on the system in which it is implemented, the type of input signals as well as the signal processing can vary greatly. Input signals can vary between analog, digital, and pulse-shaped signals, often provided by sensors and other systems linked through communication interfaces. The microcontroller, often the main processing unit of the ECU, then executes control algorithms with the input parameters provided, and a processor runs backup checks on the data to ensure the integrity of the output data.

The ECU is sometimes referred to as the car's "computer". However, in practice, the vehicles control logic is not centralized within one unit but is distributed across several units, where each manages a different subsystem [8]. Common examples of different ECUs include the Engine Management System (EMS), Powertrain Control Module (PCM), and Battery Management System (BMS) [9]. All of these units can have different network protocols and are connected by a Central Gateway (CGW), which functions as a router and translates between them. Multiple units are then coordinated by the Vehicle Control Unit (VCU) which functions as a central node to oversee multiple subsystems [8].

2.2 Hardware-Software Interface (HSI)

The Hardware-Software Interface (HSI) is the functional boundary between the software and the hardware in the ECU [10]. The Central Processing Unit (CPU) cores within a microcontroller interact with peripherals by reading from and writing to specific memory addresses called Special Function Register (SFR) [11]. This is called Memory-Mapped Input Output (MMIO). Interrupt requests are handled by the cores interrupt router. When a peripheral task is finished, a signal is sent to the interrupt router, which then handles the interrupt. In virtualization, the timing of these interrupts is the most challenging part to replicate accurately [12]. In the HSI, there is also hardware-to-hardware data movement. The Direct Memory Access (DMA) engine offloads the cores by moving data from communication buffers to local memories without interventions from the cores [13].

2.3 Surrounding Peripheral Units

The microcontroller is the main part of the ECU. However, due to the increased complexity of ECUs in recent years, they rely on supporting hardware, also called peripheral units. One of these is the System Basis Chip (SBC), which provides power supply to the microcontroller, manages external watchdogs, and handles the physical layer of communication [14]. External watchdogs function in a similar fashion as the microcontrollers internal watchdogs with the main distinction being that, in addition to verifying register values, they can handle more complex verification as well. Many external watchdogs are created within the BSW and are typically modeled around specific needs for a certain ECU and its peripheral hardware. The additional verification which commonly performed along with value verification is monitoring the functionality of the peripheral hardware. Wake-up calls are sent to ensure circuits are operational and produce a response. Once again in the event of a watchdog protocol flagging a failed check depending on the severity of the failure, a system reset can be triggered.

There is also a need for transceivers that handles communication. Certain microcontrollers contain a form of communication controller, but cannot talk to a physical wire directly. For this there requires to be a transceiver. The transceiver converts the digital logic signals from the Microcontroller Unit (MCU) into the differential voltages required for running, for example a CAN bus [15]. In vECUs, the transceiver is almost always stubbed and the virtual model usually passes the digital frame directly to a network simulation [16].

For the ECU to be able to control the actuators it also needs actuator drivers. These are high-side or low-side switches that handle currents and provide diagnostic feedback, like short circuit detection or over-temperature, back to the MCU via digital I/O or SPI [17].

2.4 Virtual Testing

A vECU can be categorized into multiple abstraction layers. The first level, level 0 seen in figure 2.1, is the lowest level and is often a Model the in Loop (MiL) setup. A MiL consists of the basic controller model and its purpose is to test the core control algorithm [18]. Moving up to higher abstraction levels, levels 1-3, see figure 2.1, the vECU is instead considered a Software the in Loop (SiL) and involves running the production ECU software stack on a PC simulation [18]. In the final stage, Hardware the in Loop (HiL), the software is deployed onto the physical ECU to verify its real-world performance [18]. The level 4, see figure 2.1, the vECU is the closest representation of physical ECU behavior and is the highest level of virtualization. When running software on a vECU at level 4 the model is no longer considered as SiL but instead referred to as a Virtual Hardware in Loop (vHiL).

Level 4 can be divided into two sublevels, 4a and 4b. Level 4a has a partial production stack, with specific hardware code being bypassed to the host which allows for a high speed simulation [19]. This method is referred to as stubbing in this thesis, and it is this approach that lays the ground for paravirtualization. Level 4b on the other hand has the full binary production stack, including drivers, and runs on simulated target hardware [19]. This level of virtualization represents the highest fidelity, with the model being a identical digital twin. However, it also has the highest performance overhead. This thesis will aim to perform simulation of the TC39x microcontroller, described in further detail in section 2.5, at level 4a through retrofitting and existing production-level BSW to execute on the virtual model.

The transition from SiL to vHiL is a critical distinction. Research published in the Winter Simulation Conference defines vHiL as a hybrid simulation that allows for the observation of peripheral resolution and processor clock speed effects without the real-time constraints of physical HiL [20]. By using emulators like QEMU or Virtualizer coupled with simulation frameworks allows for fault injection that would be impossible or dangerous on physical hardware [21], [22]. A vHiL provides a safe, reliable and realistic environment for validating complex systems at a lower cost and earlier in the design process than a traditional HiL [21].

2.4.1 ISO26262

During the development and testing of an Electrical and Electronic (E/E) system, to ensure functional safety in a road vehicles, an international standard called ISO26262 has been developed to assess and mitigate the risks and hazards potentially caused by a malfunction in these systems [23]. The standard aims to enable testing, debugging, and validation to begin earlier in the development process, supporting parallel workflows and faster iteration of the design [7]. Generally, safety critical systems undergo an extensive testing process ensuring functionality throughout their entire life-cycle. With the introduction of ISO26262 a framework for the testing of E/E has been created, establishing that all testing complies to a specified standard of functional safety in addition to the entire safety life-cycle of the developed system

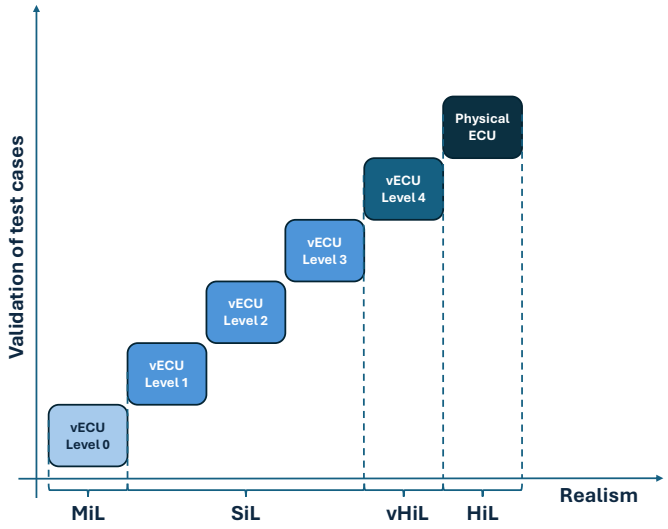


Figure 2.1: Conceptual Figure of the Abstraction Levels of Virtual ECUs

being verified [23].

Due to the limitations of this thesis, not simulation a full ECU, the ISO26262 standard is not followed to a larger extent. This being a result of that no functional testing will be conducted during the course of the project. However, an extended goal of this thesis is to enable for testing of a fully functional ECU. ISO26262 is therefore introduced and used as a point of discussion later in the thesis but not referenced during the main study.

2.5 Infineon TC39x

The microcontroller used in the target ECU is an Infineon TC39x. The TC39x contains six TriCore Cores, TC1.6.2P CPU, which use a superscalar architecture with three pipelines: Integer, Load/Store, and Loop, and have a clock speed of 300 MHz [14]. Four of these cores are equipped with Lockstep Comparator Logic (LCL) [14], which means that the processors can execute the same instructions simultaneously and have a hardware module compare their outputs after each cycle. This is a safety mechanism that makes the system fault tolerant [14]. A block diagram of TC39x can be seen in figure 2.2.

2.5.1 On-Chip Connectivity

The internal wiring on the chip consists of Shared Resource Interconnect (SRI) Fabric, System Peripheral Bus (SPB), and Back Bone Bus (BBB) [14]. The SRI Fabric is a high bandwidth crossbar that acts as a central module for communication and connects the TriCore CPUs and the DMA to high bandwidth memories, like Flash

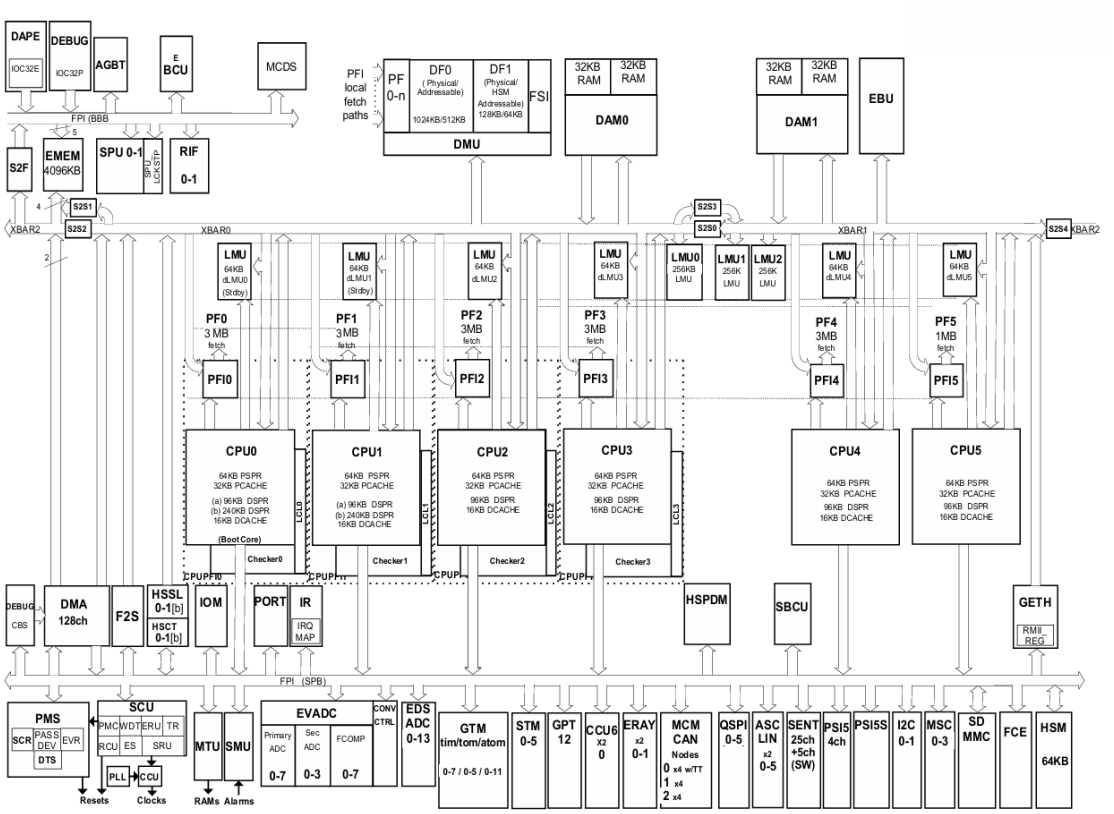


Figure 2.2: Block diagram of the TC39x. Credit original figure: AURIX TC3xx User's Manual, OPEN MARKET VERSION 2.0, Infineon

and Static Random Access Memory (SRAM), and other resources for instruction fetches and data accesses [14]. Due to the crossbar, the SRI fabric supports concurrent execution, making communication much faster than a traditional bus where only sequential communication is possible. The SPB connects the TriCore CPUs, the DMA module, and other SPB masters to medium- and low-bandwidth peripherals [14]. The SPB does not directly access the SRI Fabric but connects to it via a Serial Flash Interface to FIFO to Stream (SFI-F2S) bridge. In that way, the CPUs can control the slower peripherals as well [14]. The BBB is specifically used in Advanced Driver Assistance System (ADAS) and Emulation devices to connect ADAS resources to the TriCore CPUs, the DMA module, and the SPB masters [14]. The BBB is also not connected directly to the SRI masters but is accessed via a Serial Flash Interface to Stream to FIFO (SFI-S2F) bridge in a similar way as the SPB [14].

2.5.2 Generic Timer Module

To handle the high-speed digital I/O that is required for engine and motor control, the TC39x has a GTM [13]. The GTM is divided into clusters, each containing a specific sub-module [13]. The Advanced Routing Unit (ARU) allows sub-modules to exchange data, like duty cycles and timestamps, without involving the CPU [13]. This is a major factor in real-time performance. The submodules Timer Input Module (TIM), Timer Output Module (TOM), and ARU-connected Timer Output Module (ATOM)

then generate Pulse Width Modulation (PWM) signals or capture sensor timings [14]. TIM filters and characterizes incoming signals, e.g., from wheel speed sensors [13]. The TOM and ATOM generate the PWM signals for actuators like fuel injectors or motor controllers [13]. In addition to these input and output timer modules, the GTM also has a Multi Channel Sequencer (MCS), which is a programmable RISC-like core that can offload complex I/O tasks from the main TriCore CPUs [13]. Lastly, there is a module called Digital Phase Locked Loop (DPLL), which generates "angle clocks" (phase alignment for the output signals) for engine management based on crankshaft and camshaft signals [13].

2.5.3 Memory Configuration and Mapping

Each core in the TC39x has its own Program Scratchpad RAM (PSPR) and Data Scratchpad RAM (DSPR) [14]. Scratchpad memories are used because they typically have zero wait-states so every access takes exactly the same amount of clock cycles. This is vital for hard real-time automotive tasks. Core 0 and 1 have 240 KB of DSPR each, while the others have 96 KB of DSPR [14]. The microcontroller also has up to 16 MB of Program Flash, split into five 3 MB banks and one 1 MB bank [14]. The Local Memory Unit (LMU) provides 768 KB of global SRAM that is accessible by all masters [14].

To ensure data integrity the MTU controls and monitors initialization and tests the internal memory of the microcontroller [14]. Main features offered by the MTU are, data initialization, memory error correction and detection as well as alarm notification to the Safety Management Unit (SMU) and a unified interface to internal Secure Socket Shell (SSH) instances. Different modules within the microcontroller may contain multiple SRAM instances. Each SRAM is surrounded by a SSH module which each in turn are connected separately to the MTU via internal interfaces [14].

2.5.4 TC39x Safety Precautions

Since the TC39x is a safety-critical controller, the SMU is vital [14]. It provides a centralized interface for managing system faults, aggregating signals from both hardware and software safety mechanisms [14]. Each alarm can be independently configured to trigger internal recovery actions and/or external notifications through a standardized fault signaling protocol [14].

An additional safety module within the microcontroller is the PMS. To ensure that the correct voltage is provided and internal circuitry is not damaged select bits within registers are configured as regulator bits [14]. These bits function as indicators which external software can be based around ensuring safe levels of voltage supply. However, when running in a virtual environment the PMS checks are redundant as no voltage supply is required for virtual circuits and these can also not be damaged through over voltage.

Another safety mechanism within the TC39x is the usage of internal watchdogs.

Within the microcontroller exist a collection of watchdog protocols which monitor a multitude of various functionality [14]. From specific register transfers to monitoring full system modules handling wake-up calls and clocking. If the system is performing as intended the watchdogs protocols are not meant to be intrusive or noticed. How watchdogs are implemented can vary greatly as they tend to be specifically modified for their application. However, generally the watchdog protocol implements a form of check which is constantly run and whenever it is flagged that the register or data transfer which it monitors is incorrect an alarm is triggered [14]. Depending on the severity of the alarm either an entire system reset can be triggered or the function which failed can be triggered again.

2.5.4.1 Communication Interfaces and Protocols

The primary ways the ECU communicates with other nodes are through the Multi-CAN Controller (MCMCAN) and the FlexRay Protocol Controller (E-Ray) [13]. The MCMCAN module supports both CAN and Controller Area Network Flexible Data-rate (CAN FD) and is designed for high-speed and reliable communication with transfer rates of up to 8Mbit/s [24]. It has up to four CAN nodes and the message RAM is shared across all CAN nodes [24]. The E-Ray is a specialized controller that is designed for the FlexRay communication protocol. It provides the hardware implementation for time-triggered communication which ensures that critical data is sent and received at precise predefined intervals [13]. The E-Ray module is essential for safety-critical automotive systems where guaranteed message deliver and timing are mandatory. Similarly to the CAN module, the E-Ray controller has its own dedicated access mechanisms to a message RAM for storing and retrieving communication frames [13].

2.5.5 Paravirtualization

As mentioned before there are different ways of building a vHiL, and it can be done with different levels of abstraction (level 4a or 4b). Paravirtualization is often implemented via the Xen hypervisor and reduces code size and potential security vulnerabilities in mixed-criticality systems [4]. Paravirtualization is a virtual machine abstraction that is similar, but not identical to the underlying hardware. Unlike traditional virtualization that attempts to provide 100% binary compatibility through full virtualization [4]. In paravirtualization the guest operating kernel must be modified to run on the hypervisor. In the application of this thesis the guest kernel is the production-level BSW and the hypervisor is the Synopsys tool Virtualizer. The application layer however should remain unchanged, making the user applications not requiring modification. The hypervisor runs directly on hardware, between the hardware and the OS. This means the hypervisor has a higher privilege level than the guest OS [4]. In practice this level of higher privilege allows for direct modification during the simulation which would otherwise not be possible. Examples of modifications relevant for the thesis are, direct register manipulation of the TC39x or machine instruction manipulation.

2.5.6 Full Virtualization

The foundation of a level 4b vECU is the Instruction Set Simulator (ISS), which interpret or translate the target's machine code [25]. There are two types of ISS, Instruction Accurate (IA) models and Cycle Accurate (CA) models. The IA model simulate the functional effects of every instruction but does not account fpr the exact timing in the CPU pipeline. QEMU is the industry standard for this. It uses Dynamic Binary Translation (DBT) to convert target insructions into host instructions in real-time [10]. The CA models are the more exact models. They simulate the complete CPU pipeline, with cache misses and bus contention. These kind of models provide the highest fidelity for scheduling analysis in real-time systems [26].

The peripheral units in the ECU is often modeled with Transaction Level Modeling (TLM) via SystemC as a full peripheral state machine. Instead of simulating every individual electrical signal (pin toggling), transactions (read/writes) to register addresses are simulated [10]. For he production BSW to run, the model must implement the exact memory map. If the BSW writes $0x01$ to address $0xFF00$ to start a timer, the model must recognize that write, start a virtual timer and eventually trigger a virtual interrupt [27].

If documentation of peripheral units, like an external ASIC, is missing, a common way to work around this problem is to use differential testing. In differential testing, I/O traffic like Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), and CAN, is captured from a physical ECU during operation [28]. The captured traces are then used to build a behavioral model that mimics the ASICs responses when the virtualized MCU sends specific commands. This allows the production binary to believe it is communicating with a real external chip [28].

Full virtualization of a system like an ECU leads to significant performance overhead. It is not just the microcontroller that needs to be virtualized, but also all the peripheral circuits, the watchdogs, CAN, LIN, Ethernet, etc. Paravirtualization results in a virtualized system that is not identical to its physical counterpart, but very similar and with significantly less performance overhead [4].

2.6 Structure of Automotive BSW

The BSW is the infrastructure that provides all the essential services the application layer needs to run [29]. With the BSW providing these services, the application layer can perform its functions without needing to know all the underlying structures, such as which pin is mapped to where and what all the memory addresses hold [29]. A conceptual figure of how the ECU software is structured can be seen in figure 2.3. The BSW is structured into four main layers. The first of these layers is the Services Layer, which is the highest level of the BSW. This layer handles the OS, system management, and communication protocols (such as CAN) and is mostly hardware-independent unless it expects a specific hardware timer to drive it [29]. The

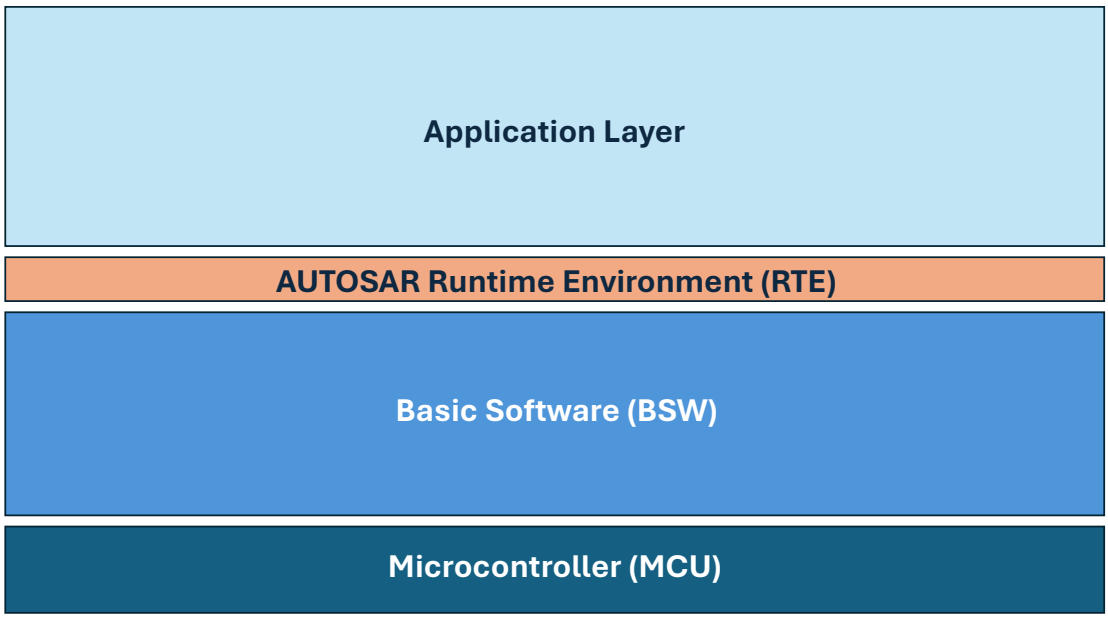


Figure 2.3: Software Layers of an ECU

second layer is the ECU Abstraction Layer, which handles all the services provided by the hardware to the ECU. These services can be communication mapped to specific busses, memory access, or I/O for sensors and actuators [29]. The third layer is the Microcontroller Abstraction Layer (MCAL) which abstracts the microcontroller architecture and provides basic drivers for the microcontroller [29]. This is the layer that sits closest to the hardware. The MCAL provides an upper-layer agnosticism where the layers above it, including the rest of the BSW and the application layer, can be unaware of the specific microcontroller being used. The fourth and last layer of the BSW is the Specific Driver Layer that holds drivers and special functionality code that are not part of the AUTOSAR standard [29].

2.7 AUTOSAR Standard for Automotive Basic Software

AUTOSAR is a global standard introduced in 2003 by multiple Original Equipment Manufacturers (OEM) to create a standardization of ECU functionality, development, and language for architectural models [30]. The standardization aims for increased scalability of different vehicles with the intent to improve software transferability between platform variants. An overarching goal of AUTOSAR is to reduce the time and cost of research and development by allowing basic software modules to be used in vehicles produced by different manufacturers and with electronic components from different suppliers. Another goal is to increase the use of ”off the shelf” solutions by minimizing the use of modules created for specific use cases, as this prohibits the scalability and transferability highlighted above.

The desire is that just the MCAL needs to be modified when transferring the BSW to new hardware [29]. The advantages of the AUTOSAR standard are the hardware portability, it makes it easier for companies to buy and implement sensors, actuators and other components essential in vehicles. However, it is also quite complicated and heavy. The abstraction layers (BSW and MCAL) take up significant RAM and Flash. It is also expensive. Licensing an AUTOSAR toolchain can cost millions of Swedish Krona (SEK) per year, with additional royalties per ECU produced. This leads to some companies or software teams not using it, or using it only to a certain extent. The advantage for developers not using AUTOSAR is that they have more freedom in how they choose to implement the code and do not have to deal with the potential performance overhead a standardized system could bring. However, it can also lead to a technical debt where projects that grow too big retain a lot of legacy code that is written to work with the specific characteristics of the physical microcontroller silicon, which makes it brittle in a virtual environment.

2.8 Peripheral Modeling vs Stubbing

There are different ways of building a virtual model of an ECU, and since this is a complex system some trade-offs need to made in order to have a functioning virtualization of the system. The different parts of the ECU can be modeled precisely using tools like QEMU or the SystemC language. In this method, the hardware peripherals (CAN controllers, timers) are virtuaized at the register level [10]. Full virtualization of both the microcontroller and the peripheral units is complicated and would lead to significant performance overhead, considering the size and the complexity of the system. Therefore, the usage of paravirtualization is common and peripheral units are stubbed to ”trick” the microcontroller that it is part of a physical ECU. There are different methods for doing this. One of them is functional stubbing, where the focus is input-output correctness. The stub returns a predefined value or a simple ”pass-through” [11]. By doing this, the software can be tested in its original state, thereby avoiding any modifications to the production code.

2.9 State of the Art

Within the automotive industry advances into virtual environments have been underway for multiple years. The extent of how much is virtualized has however steadily been increasing. Below early stages of virtual simulation and testing are presented as well as challenges related to increased virtualization within the automotive industry.

2.9.1 Automated Test of ECUs in a HiL Simulation Environment

As previously mentioned in the introduction of this thesis, vehicle complexity is making testing of software and ECUs increasingly relevant for each new vehicle brought to the market. This, however, is not a new phenomenon. In the paper ”Automated test of ECUs in a hardware-in-the-loop simulation environment” from 1999, the use

of HIL-rigs to test ECU functionality lifts many issues which, to this day, are still relevant [31]. The paper describes the application of real-time simulation tools and how automation techniques can be used during the development of automotive ECUs.

Using development tools such as MATLAB/Simulink an ECU model representing an internal combustion engine and drive train can be developed and verified [31]. Using the modeling tools to automatically generate code for high performance floating point hardware, a direct memory interface between a prototyping system and ECU occurs. This allows engineers to test real ECU hardware against a real time simulated engine and vehicle environment. In doing this, the HIL-rig allows for manually performable tasks to be automated, making it possible to trigger safety critical functions remotely and test the ECU under extreme driving conditions safely and remotely. An added benefit is that reusable scenarios in which the ECU can be tested are possible to create, making it possible to standardize the testing framework [31].

2.9.2 Virtualization Technologies for the Centralized Automotive E/E Architecture

The developments in automated testing of ECUs using HIL-rigs are still highly relevant since it allows for extensive testing of ECUs before prototypes of engines or vehicles are available. As the complexity and number of ECUs grow, so does the need for rigorous testing. Testing the software on real hardware becomes expensive, and often many teams share the same HiL-rigs, causing delays when a queue builds up. In the paper ”State-of-the-art virtualisation technologies for the centralised automotive E/E architecture”, the authors describe how vECUs allow developers to test and validate control software without relying on physical hardware [8]. This can ease the load on the HiL-rigs since developers can catch errors early using virtual HiL-rigs. With new tools emerging, it is possible to build a digital twin, which provides an accurate digital replica of the entire system. This allows for system-level monitoring and optimization in real-time [8]. It is also possible to test situations which are currently unable to be replicated on the physical HiL-rigs. One example is fault-injection with system failures that would break the physical hardware but can safely be tested on a vECU [8].

There are also challenges and research gaps when it comes to vECUs. One of these is Real-Time Performance where virtualization adds a layer of ”overhead” that can slow down processing [8]. Another challenge is safety standards. All virtual environments must still comply with ISO 26262, which requires rigorous functional safety analysis of the virtual machine monitor itself [8].

2.9.3 A Model-Based Design and Verification Framework for Virtual ECUs

Using a virtual model to connect to real sensors and actuators can therefore give much needed insight in how the ECU will perform in real life scenarios. The study

”A Model-Based Design and Verification Framework for Virtual ECUs in Automotive Seat Control Systems” from late 2025 shows that using a combination of a Simulink model, AUTOSAR compliant code and a vECU connected to a combination of simulated and physical I/O the gap between a physical ECU and vECU is minimized [32]. Using this method the experimental results showed that the test setup had comparable timing behavior and execution in relation to physical ECU execution which leads to the ability for more extensive testing without the use of prototype hardware [32].

2.9.4 Virtualization for Automotive Safety and Security Exploration

With the increasing complexity of E/E systems in vehicles, there is also increased vulnerability to electronic and software failures as well as cyber-attacks. The key issue lies in that connectivity has been introduced in a congregation of components that were not originally designed with connectivity in mind [33]. Originally ECUs were designed to communicate via CAN in a closed system. Therefore there were no need for safety measures like authentication and validation. This increased connectivity has introduced a risk of exploitation by hackers.

There is also an issue of failures. Modern vehicles are distributed systems with several ECUs and sensors and actuators that are vulnerable to software and electronic failures. The paper ”Virtualization for Automotive Safety and Security Exploration” shows how failures can be prototyped and address the risks and failure classifications defined by the ISO 26262 functional safety standards, focusing on how electronic component failure rates and hardware modes impact the overall system [33]. The platform that was used facilitated the swapping of components, which enabled designers to simulate system-level failures in a virtual environment early in the design process [33]. They could also simulate the absence of a component to model a complete hardware failure or a lost electrical connection by connecting the relevant communication port to a ”blank process” [33]. The researchers set up a simulated steering scenario where the driver was going to make a right turn. They simulated a failure in the load motor actuator which resulted in a lack of applied torque, leading to an error where the wheels fail to turn [33]. This case demonstrates that malfunctions in an ECU lead to severe computational dysfunctions, which can prevent a driver from completing basic maneuvers like a successful turn. This paper shows that virtualization can be very useful early in the design process, and designers can simulate broken behavior of the ECU and the peripherals without any risk of breaking physical components.

2.9.5 Automotive Network Mapping

Another paper that address the weaknesses regarding security on automotive systems is ”CANvas: Fast and Inexpensive Automotive Network Mapping” [34]. They used a fault injection technique called forced ECU isolation. CANvas directly induces errors by driving domain bits to the CAN transceiver during the data field of

the message. They triggered "bus-off" states by incrementing an ECU's Transmit Error Counter (TEC) until it exceeded 255. They did that with all but one ECU, successfully isolating it in the system. With this mapping, the paper illustrates how security weaknesses can be identified related to a lack of fault tolerance in message filtering. The authors discovered that many ECUs in the tested vehicles do not implement strict hardware filters which makes them susceptible to cross-bus attacks. This means that non-critical ECUs, like a telematics unit, could potentially receive and then attack a safety-critical ECU, like the engine control module through the bus-off technique [34].

2.9.6 Parallel End-Of-Line (EOL) Testing

One of the bottlenecks in automotive production is End-Of-Line (EOL) testing. In traditional EOL testing of ECUs each input and output pin or functional block is tested sequentially. With each request and response on the CAN bus taking a specific amount of time, and modern ECUs having up to 140 pins, sequential testing becomes a time consuming task [35].

The researchers of the paper "Modeling of the Automatic Testing Process of Electronic Control Units in the Automotive Industry" investigated the possibility of parallelizing the EOL testing phase [35]. They started by analyzing the traditional execution time for every I/O pin and created an algorithm for parallel execution where they grouped test tasks so that multiple signals or functional blocks were interrogated or simulated simultaneously [35]. They then used an EOL station with sensors, switches and resistors and measured the ECU's response via the CAN line [35]. The results were a significantly faster execution of the EOL tests, and by speeding up the slowest station the overall capacity of the assembly line increased without needing to add additional physical space [35].

This paper shows the impact that software can have on the capacity of both production and development in the industry, and similar adaptations can be made in research and development. The difference is that in production there needs to be tests in the physical component since it is end-of-line, while in research and development a lot of testing early on can be moved to software and to virtual environments. By leveraging this technique, errors can be caught early, before being sent as a target test to a physical HiL-rig, which in turn increases the testing capacity without needing to add additional physical space.

2.9.7 Xen and the Art of Virtualization

Virtual models have become increasingly important to the industry. There are a lot of different techniques for building a virtual model of a system and for determining the appropriate abstraction level for that model. In a paper from 2003, "Xen and the Art of Virtualization", a paravirtualized model was presented. Xen is a high-performance Virtual Machine Monitor (VMM) that was designed to partition a single x86 server into multiple resource-managed virtual machines [4]. The goal of the paravirtualiza-

tion approach is to represent a virtual machine abstraction that is similar, but not identical, to the underlying hardware [4]. With full virtualization, the model becomes accurate with high fidelity, but also incurs significant performance overhead. With paravirtualization, the significant performance overhead and complexity is avoided [4].

The hypervisor, also known as the VMM, is the software layer that operates at a higher privilege level than the supervisor code of the operating system it hosts. It partitions the physical resources of a computer, like the CPU, memory, and devices, between multiple virtual machines and is responsible for safety, isolation and validating the calls from the guest OS for operations like page-table updates to ensure one domain cannot adversely affect another. In this case, Xen is the hypervisor and runs on the highest privilege level, which is Ring 0. The guest OS is modified to run at a lower privilege level (Ring 1), keeping it isolated from both the hypervisor and Ring 3 applications [4]. The privilege levels are hardware-enforced mechanisms that limits what a particular piece of software can do on a computer's processor. The privilege levels acts as a security clearance. The higher the clearance, the more sensitive commands are allowed to be executed. Ring 0 is the most privileged one and can be thought of as the "inner circle". In a standard computer, the OS kernel lives here and are the only code that is allowed to talk directy to the hardware, manage memory, or stop the CPU. Rings 1 and 2 are intermediate levels and were originally designed for device drivers or OS services, but are on modern systems, like Windows or Linux, almost never used. Ring 3 is the least privileged, and is where applications run. They cannot touch the hardware directly, but must ask the OS in Ring 0 to do things for them. In a normal setup, the OS has the highest privilege and sits in Ring 0. but in the virtualized Xen environment, it is moved to run in Ring 1, and instead the hypervisor, Xen, runs in Ring 0 [4]. This is to prevent the guest OS from crashing the whole machine, or spying on other virtual machines. If the OS tries o execute Ring 0 commands while it is in Ring 1, the processor will block it and trigger a fault, allowing Xen to step in, validate the request, andexecute it safely on the OS's behalf. [4]. This structure allows Xen to keep multiple operating systems isolated from each other while ensuring the hardware remains secure. In the Xen model, the OS is aware of that it is running on a VMM and is modified to use an idealized virtual machine interface [4].

The guest OS is responsible for allocating and managing hardware page tables, while Xen validates all updates to ensure safety and isolation [4]. Policy is separated from mechanism and a special domain called Domain0 is created at boot to host management software and control the creation and deletion of other domains [4]. Xen follows a design principle of separating policy from mechanism [4]. The hypervisor itself provides only basic control operations, such as scheduling the CPU or filtering network packets, and Domain0 is a specific privileged domain that is created at boot time that is permitted to use the control interface [4]. Xen hosts application-level management software used to create or terminate other domains, set resource allocations and manage virtual network or block services [4]. I/O data is moved via asynchronous shared-memory buffer descriptor rings, which provide high-performance communication between domains and the hypervisor [4]. Instead of hardware interrupts, Xen uses a lightweight eventdelivery system used for asyn-

chronous notifications, and for CPU scheduling, a Borrowed Virtual Time (BVT) algorithm is used to provide low-latency wake-ups for domains receiving events [4].

The authors of this paper showed that for many workloads, the XenoLinux which is a ported version of Linux 2.4, performs within a few percent of native, unvirtualized Linux [4]. By using industry-standard tools (SPEC WEB99 and OSDB) for evaluation, they showed that Xen significantly outperforms other solutions like VMware Workstation 3.2 and User-Mode Linux (UML) in system-wide benchmarks [4]. Xen provides strong performance isolation and ensures that even malicious domains running disruptive workloads (like fork bombs or disk hogs) do not significantly impact the performance of other domains. This was done with relatively little effort when it comes to porting of the OS. Linux required modifying roughly 3,000 lines of code, which corresponds 1.36% of the x86 codebase [4].

The Xen project successfully implemented a, at the time, new approach on virtualization. One of the future goals of the Xen project is to develop so called XenoServers, which aims to provide an internet-scale computing infrastructure where users can dynamically instantiate at ISP locations and pay for the resources they consume [4]. They also want to further improve I/O and memory performance by implementing a shared universal buffer cache and a ”last-chance page cache” [4].

2.9.8 Relating to State of the Art

The testing of automotive software has steadily been turning towards being less in actual hardware and more in controlled simulated environments. As presented above the SiL solutions using MATLAB/Simulink and other forms of HiL solutions are nowadays relatively common and used extensively throughout the industry. This thesis aims to further explore, through a scientific method, the possibilities of testing without the use of a full production version of hardware as many of the papers presented above have done. Trying to push the earlier discoveries to a higher abstraction level of an ECU and similarly as the 2003 paper, ”Xen and the Art of Virtualization” establish the boundaries and requirements currently existing for executing a BSW in a virtual environment.

3

Approach

To achieve the outlined aim of the thesis, a literature study has been conducted to investigate the current usage of vECU within the automotive industry, as well as to what level these vECUs represent their physical counterparts. With the VDK, that was provided by Synopsys, there were a few known limitations called stub-modules. These needed to be taken care of via python scripts that manually set control bits in the virtual registers. The second thing that needed to be done was to replicate the memory state that the physical ECU was in when power was turned on. Lastly, to combat the dependencies on external hardware, the initialization functions of the peripheral units where removed from the startup sequence. A road-map of the virtualization approach used during this thesis can be found in figure 3.1.

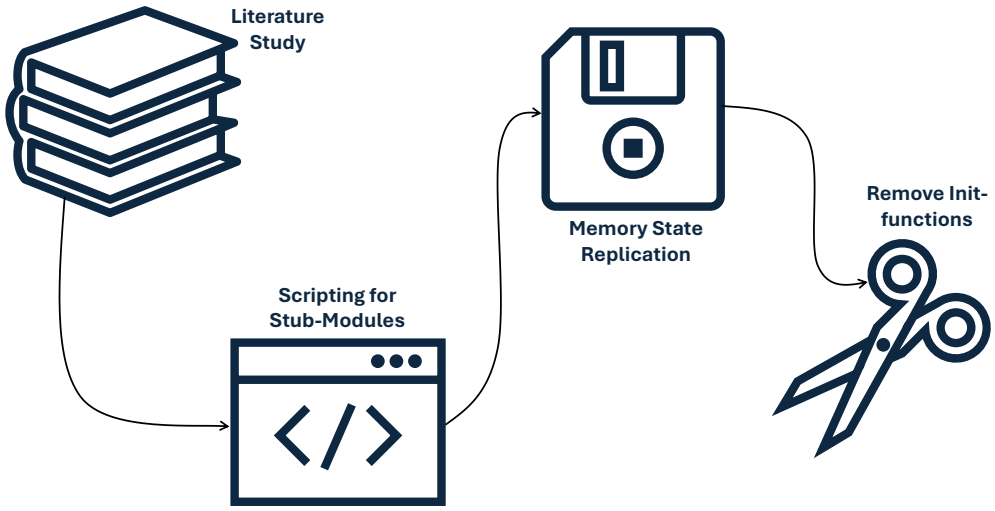


Figure 3.1: Road-map of the virtualization approach used in this thesis

3.1 Tools

The primary tool used during this thesis was Synopsys Virtualizer. Virtualizer allows for the creation of virtual prototypes of select target hardware [16]. Virtualizer

contains a library of processors and peripheral models with out-of-the box compatibility which makes it possible to integrate across compilers, libraries and debug environments [16]. Using these processes it is possible to create a Synopsys VDK which is a digital twin of an electronic system combined with software tools for development and testing [16].

The library mentioned above that is used by the Virtualizer toolkit consist of TLM blocks which are descriptions of hardware components behaviors [16]. These are the fundamental building blocks on which the full VDK is built. A VDK is a collection of TLMs which are a functional representation of how a hardware component performs. When creating larger and more complex virtual hardware model, such as an ECU, it is possible to use multiple VDKs or TLM blocks to simulate interconnected hardware. In this thesis a VDK for the microcontroller TC39x from Infineon was provided, which is the same microcontroller as in the ECUs that are the target systems for simulation.

The TLMs are based on SymstemC, making it possible for a user to create their own TLM in the case that an existing one in the library does not satisfy the needs for the current project [16]. However, throughout this thesis no additional VDKs or TLMs were created as the complexity of these were too large to fit within the scope of this thesis. Instead, python scripts were made to generate expected signals for the microcontroller as a paravirtualization approach. Specific registers and data within the microcontroller required for startup, was also manipulated through the use of python scripts and directly with the help of a debugger.

3.2 Execution of BSW Within the Virtual Environment

Using the provided VDK the BSW, available from Volvo AB, was executed within the virtual environment to investigate what barriers existed to prevent full execution of the software. Milestones were created to accurately represent how far through the startup sequence the virtual microcontroller had run, creating a quantifiable measurement of portability. To bypass hardware dependencies modules were stubbed and modified to further progress through the startup sequence with the ultimate goal of running the OS on the microcontroller and performing a simple component execution.

3.2.1 Bypassing Hardware dependencies

In the VDK provided by Infineon there are known limitations that make the virtual microcontroller function differently than its physical counterpart. One of these limitations are stub modules, which are components that have been simplified compared to the physical microcontroller. The MTU and PMS are both known stub modules and were both required to be modified in order for software to progress through the startup sequence. Using the BSW of an existing project from Volvo AB in the Virtualizer toolkit it was verified that these modules were initially hit and forced a

reset prohibiting a full startup. The execution of the same BSW on a physical ECU was compared to the execution within the toolkit on the virtual TC39x in order to examine the functionality of the non stubbed versions of the same modules. Using debuggers, to investigate register behavior in both versions of the microcontroller, as well as investigating what type of safety checks existed within the software to ensure these modules functioned as intended, python scripts were created to mimic the behavior of these modules within the virtual environment. These scripts monitored the behavior of the virtual microcontroller waiting for the software to perform safety checks, ensuring the modules functioned as intended, and provided the expected input values for the VDK into the correct registers to mimic normal operation.

The safety checks performed based on the MTU and PMS modules are checks within the software to ensure that the microcontroller and by extension the ECU are performing as intended and not experiencing hardware faults. Therefore these checks are built into the software and created specifically around the expected performance of the ECU based on extensive testing and previous observations. Multiple of these checks exist within the BSW and are all required to pass in order for the startup sequence to proceed and the microcontroller to boot into OS. Additionally watchdog protocols check register values and code execution monitoring the startup sequence for faults. Similarly to before debuggers on both physical- and virtual hardware were used in addition to investigating the source code and differences in performance was studied. Multiple python scripts were created to bypass external hardware dependencies by simulating expected behavior and providing inputs to the virtual microcontroller when required.

In the final steps before the microcontroller fully booted into the OS a large number of hardware dependencies became clear. Prior the dependencies had mainly been internal or required one time inputs. The initial OS boot checks required not only an initialization of multiple ASIC but also continued responses from many of these to fully boot into the OS. To avoid fully recreating the circuits required, due to the complexity of their nature, the dependencies were attempted to be removed from the software. Multiple approaches were used to workaround these dependencies such as, adjusting the BSW to remove functions requiring external dependencies, using python scripts, adjusting the execution of the virtual microcontroller and jumping over the specific machine code instructions as well as forcefully injecting values into registers with debuggers, bypassing certain dependency checks.

3.2.2 Minimizing Discrepancies Between Physical and Virtual

Upon further investigation it was discovered that the software required specific values which had been initialized in the physical hardware through running a boot sequence which was not performed in the virtual hardware. Due to this a watchdog protocol was initiated to enable a reset of the system preventing the startup to proceed. The boot sequence executes once on the physical hardware initializing the memory and stores it in specific registers which are accessed by the software when requested. As

a result it was required for a similar initialization to be performed at the startup of the virtual model. The initialized memory blocks were extracted from the physical ECU. A python script was then created with the purpose of initializing the correct memory before the software entered the startup sequence. This new ”boot sequence” was required to run at the start of every simulation as memory was not preserved between simulations and required very specific ordering as the software has been built around these values being initialized within the specified registers.
Once the correct memory had been initialized the virtual model encountered a check which was reliant on the ports of the microcontroller to be verified in order for the software to function as intended after startup. The virtual model of TC39x did not have the required functionality enabling the check to pass. An addition to the VDK was therefore created to simulate the required ports enabling the check to be passed.

3.3 Evaluation of Virtualization Transparency

The virtual environment and how the software has been executed has continuously been compared to a physical ECU running the same software, containing the same microcontroller. The degree to which the software has been modified and to what extent the creation of external models, in the form of scripting functionality and memory injection, has been necessary have been recorded to evaluate the amount of modification required for a production-level BSW to run in a virtual environment. This was done to quantify the amount of modifications required for the software to run on the virtual hardware. This measurement was also used as a baseline for evaluating how much of the functionality in the virtual simulation was lost due to stubbing and jumping instructions. If it was required for the BSW to lose a large amount of functionality and processes the argument of virtualization allowing for initial testing would lose strength. The virtual model is required to maintain a level of accuracy compared to the physical hardware in order for the results to be desirable.

3.4 Success Criteria

The evaluation of the success for the thesis has been divided into two major success criteria:

1. Verification of the BSW hardware initialization.
2. Successful deployment of the OS.

The first criteria focuses on the initialization of all hardware components within the microcontroller. As the hardware components are within the microcontroller there are fewer dependencies and the initialization is of a more linear fashion. Therefore, this first criteria was formulated as an intermediate goal. Success was determined by verifying that all mandatory initialization functions were executed during the startup sequence. Once all hardware was initialized the microcontroller proceeds with safety checks and boot up of the OS. To measure success, virtual environment

execution was compared against a baseline list of which startup functions execute during each stage and on which core. This comparison determined the progress of the virtual microcontroller within the startup sequence.

The second criteria confirms the microcontroller's readiness for high-level tasks. With a complete boot of the OS the scheduler performs tasks and components created by the development team within the software. Therefore to verify OS functionality and execution a test component was created to validate the scheduler. The component was configured to increment a variable every 160 ms, providing a measurable indicator of a functioning OS. As OS boot up is the final stage of the startup sequence for the microcontroller, execution of the test component would prove normal operation within the test environment.

3.5 Limitations

This thesis will not cover creation of a full vECU for simulation of an entire ECU. Instead, a microcontroller will be simulated using a VDK as a virtual model with python scripts mimicking basic functionality of external circuits allowing the virtual microcontroller to pass safety checks within the BSW. These limitations have been set as a result of the complexity of recreating a full ECU would not be possible within the time frame of the project. Additionally, if successful, entering the OS on the microcontroller would allow for testing of basic functionality and components which could be built upon further with future work making this a justifiable goal for this thesis. Throughout the work of this thesis limitations relating to the tool set as well as the structure of the software have been discovered however, these will be discussed more extensively in the results and discussion.

Furthermore, the boot software will be excluded from the virtual microcontroller execution. Instead, a "warm start" will configure the necessary calibration values and headers. Because Virtualizer overwrites virtual memory between runs, executing the boot software would require integrating it with the BSW, a task excluded from this project due to the thesis time frame constraints.

4

Results

In this chapter, the technical outcomes of the virtualization process are presented. The results are characterized by the iterative modifications required to progress the production BSW through the startup process on a virtual target. The final state achieved was the transition to the OS task entry level, where further execution was halted by hardware dependencies.

4.1 Bypassing Hardware Dependencies

Upon initial execution of the BSW on the virtual target, the VDK encountered immediate failures in core 0 of the virtual TC39x, see figure 4.1. Here a chart is shown which shows the functions executed on the VDK and how long these are executed for. The official function names have been omitted due to a privacy agreement with Volvo AB. In figure 4.1 it can be seen that once the startup function has begone execution a memory fault is triggered at roughly 2.7 ms. The microcontroller is then stuck in an execution loop as this memory fault triggers the MTU due to it being modeled as a simpler version of its physical counterpart, a so called ”stub module”, it did not provide the register feedback expected by the production software.

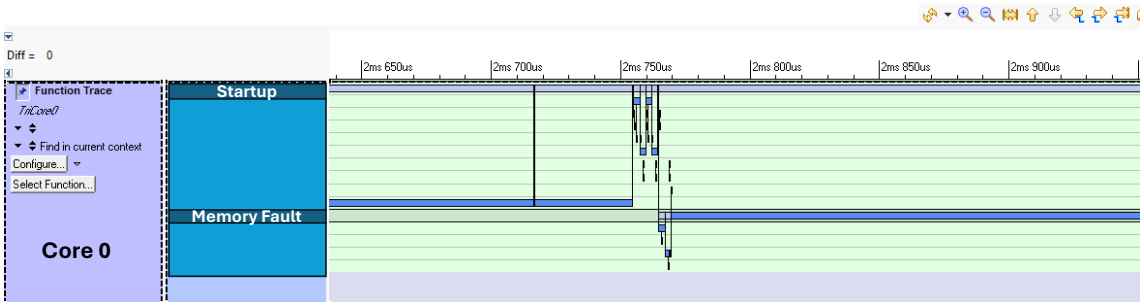


Figure 4.1: First Encountered Failure Without Modification

To resolve this issue, Python-based workaround scripts were developed to intercept MMIO calls forcing the stubbed MTU module to return a positive result on the memory check allowed the virtual TC39x to proceed further through the startup sequence. Once passed a second fault was triggered, which resulted in a systematic reset, see figure 4.2.

As shown in figure 4.2 the MTU workaround which was added allowed for further

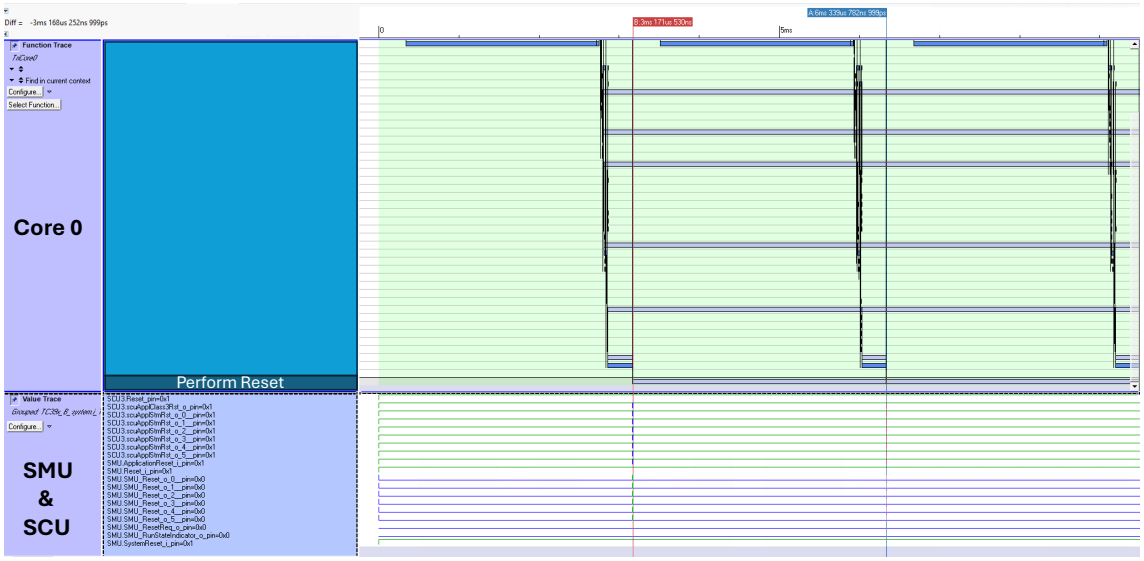


Figure 4.2: Resets After Modifying MTU Stub Module

progression through the startup sequence. Instead of a function being executed repeatedly as before, a system reset was being triggered at roughly 3.1 ms. This can be seen by the reset pins being activated in the lower half of the functions charts indicating a total system reset. The system then performs the same startup and is once again forced to a reset which can be seen at 6.3 ms.

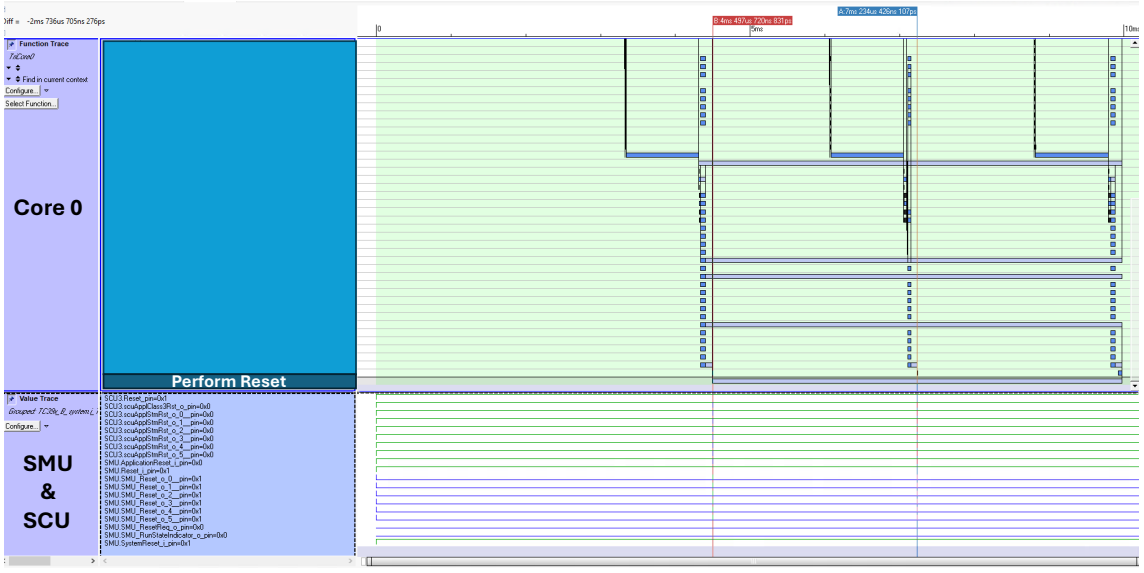


Figure 4.3: Resets After Modifying PMS Stub Module

It was discovered that this reset was due to the PMS which similarly as the MTU

is a known stub module. With the PMS not providing the correct ”Ready” and ”Stable” bits the system believed that it was not receiving the correct voltages for function and therefore forcing a system reset. For the interception, the scripts forced the peripheral status registers to return these ”Ready” and ”Stable” bits, bypassing the voltage-level checks that would otherwise trigger a reset. This addition allowed for the system to believe the PMS functioned as intended and we were able to once again proceed further into the startup sequence, see figure 4.3.

With the addition of the workaround for passing the voltage checks required by the software with the help of mimicking PMS functionality it was discovered that the virtual TC39x once again hit systematic resets. These resets were however, being triggered further into the startup sequence with the first being triggered after a runtime of 4.5 ms.

With the bypassed functionality of these two stub models it was determined that the reset triggered at 4.5 ms was not due to hardware dependencies. This being based on the fact that no additional known stub modules were required for functionality, It was also observed that internal watchdog protocols were being triggered relating to memory value faults. Connecting debuggers to both the virtual VDK and physical ECU allowed us to determine that certain memory areas that were meant to be initialized were missing in the virtual environment.

4.2 Memory State Replication and Injection

A significant barrier to the startup process was the absence of calibration values and headers that were present in the physical ECU upon startup. Without these, the virtual TC39x entered a continuous reset loop due to failed integrity checks. The solution to these issues involved a two-step injection process. First the physical reference ECU was flashed to extract the missing memory areas. The resulting memory image was extracted and mapped onto the VDK’s virtual SRAM and Flash.

The memory that was flashed from the physical ECU was the Program Flash and the Data Flash. In table 4.1 the total amount of memory extracted is showed. We successfully managed to flash all 750 KB from the Data Flash. However, when we tried to extract the Program Flash, we ran in to the issue that certain parts of the memory areas were under memory protection and could not be extracted. We extracted as much as possible from the Data Flash and in the end we managed to flash 6.7 MB of 14.7 MB and injected this to the corresponding memory areas of the virtual microcontroller. In total we managed to extract and inject 7.45 MB of 15.45 MB in the virtual microcontroller, as shown in table 4.1.

The result of the memory extraction and injection was that we managed to get passed the frequent resets and core 0 tried to enter the initial tasks of the OS, see figure 4.4. However further execution of the OS was prohibited by the initialization of external peripheral units. These take place during level 2 of startup in core 1, as shown in figure 4.4.

Table 4.1: Result from memory extraction

Memory Section	Extracted	Total
PROGRAM_FLASH	6.7 MB	14.7 MB
DATA_FLASH	750 KB	750 KB
Total	**7.45 MB**	**15.45 MB**

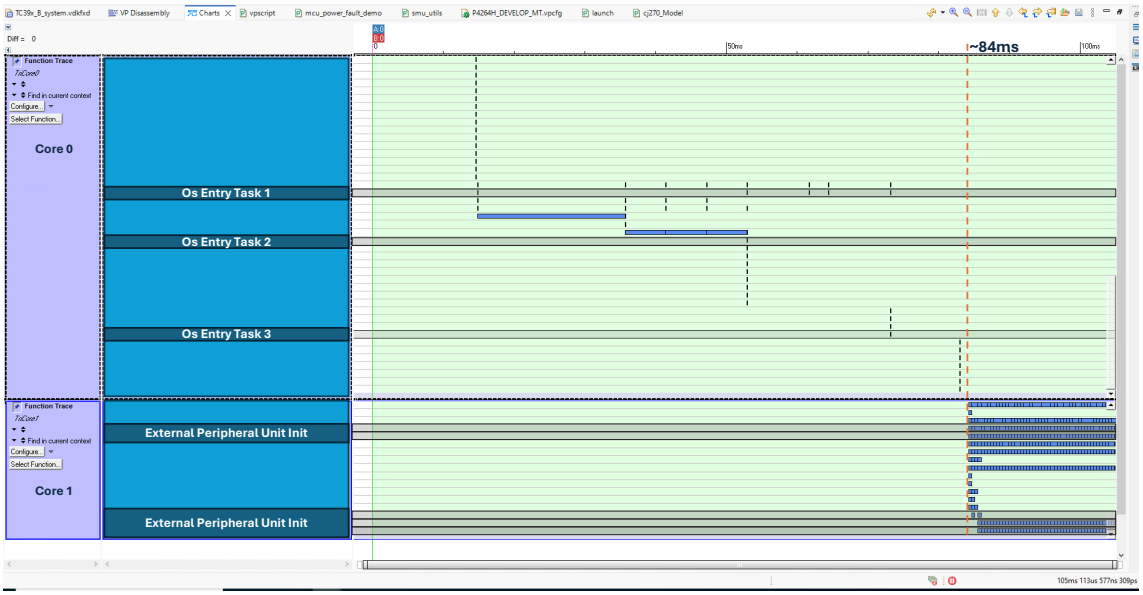


Figure 4.4: Execution after injecting the memory with values from the physical microcontroller

4.3 VDK Hardware Model Verification

After the memory had been injected the virtual microcontroller proceeded to initialize external hardware. Ports which handled communication with these external units where then also initialized to establish communication. During the initialization of Port 1, a systematic failure was identified in the virtual hardware model itself. The Port Control Select Register (PCSR) was incorrectly blocked, preventing the software from configuring the pin directions. This result was verified through register-trace analysis, which lead to a collaboration with the vendor (Infineon). A revised VDK model was issued where the PCSR blocking logic was corrected. This allowed the BSW to proceed past the HAL initialization of the digital I/O.

4.4 Modification of External Hardware Initialization

To allow for the execution of OS entry tasks on core 1 of the virtual TC39x, external peripheral dependencies were required to be removed. The driver functions for the external peripherals were run continuously, see figure 4.4, preventing the software

from progressing as no inputs were provided due to the peripheral hardware not being connected in the virtual environment.

Modifying the production binaries by directly adjusting the BSW allowed for some additional progression through the startup sequence. In figure 4.5 it can be seen that on core 1 of the virtual TC39x OS entry tasks are being executed and the microcontroller run-time is progressing past 84 ms. In total 9 out of 37 functions were stubbed out of the startup sequence in order to reach OS entry on core 1. The functions targeted were mainly functions relating to the initialization of external hardware drivers as well as communication protocols with external circuitry.

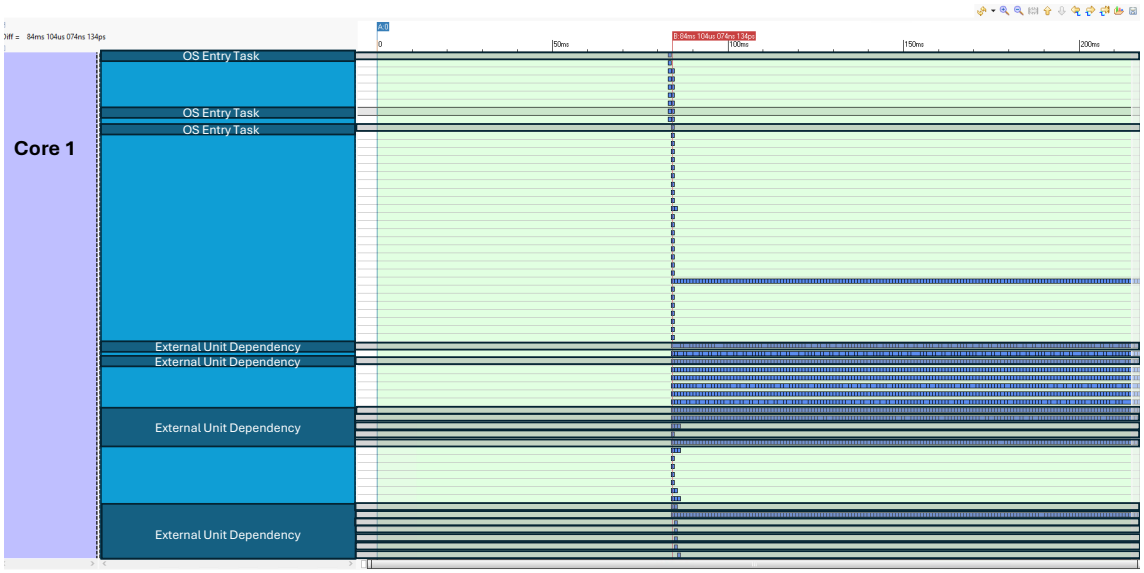


Figure 4.5: Execution After Stubbing Peripheral Hardware Initialization Functions

The culmination of these modifications allowed the virtual microcontroller to reach the initial OS entry tasks in core 1. At this point, the kernel attempted to schedule the first high-level application tasks, where among others, we had put a basic component called MyFirstComponent. However, execution reached a ”dependency explosion” where the OS expected feedback from the GTM and CAN interrupts. As these were not modeled to support the timing sensitive OS scheduler, the system reached a terminal state. This demonstrates that for Level 4 virtualization, functional stubbing of individual registers is insufficient to support a full-scale BSW without a synchronized peripheral simulation.

4.5 Summary

In table 4.2 a chronological overview of the modifications and the results achieved is provided. These results demonstrate an iterative progression through the BSW

startup sequence and was achieved by systematically addressing the hardware dependencies observed. The process began with low-level register manipulation via Python scripts to bypass the MTU and the PMS checks, which allowed the BSW to execute for 4.498 ms before resetting, instead of the initial 2.750 ms as before any modifications were made. This was followed by a memory injection phase, where 7.45 MB of extracted physical ECU data (100% of the Data Flash and roughly 45% of the Program Flash) was mapped to the VDK. This step resolved integrity-check failures and allowed the software to reach the OS entry level at approximately 84 ms.

The final phase involved the identification of a hardware model error in the PCSR logic, which was corrected by the vendor, and the stubbing of 9 out of 37 startup functions related to external peripherals. While these modifications enabled the TC39x to reach the OS task scheduling stage on Core 1, the execution ultimately reached a terminal state due to a ”dependency explosion” triggered by missing GTM and CAN interrupt feedback.

Table 4.2: Summary of the modifications made and the results

Modification	**Execution time**	**Comment**
No modifications	2.750 ms	Starts resetting
MTU Workaround	3.172 ms	Got past the MTU check, but still resetting
PMS Workaround	4.498 ms	Got past the PMS check but still resetting
Memory Injection	∼84 ms	See the first entry functions of OS in core 0, gets stuck on initialization of external units
Removing Init Functions	∼84 ms	See entry functions of the OS in core 1, still have dependencies on external functions

5

Discussion

While we did not succeed in fulfilling both of the success criteria for this thesis, the results highlight some important insights regarding virtualization in the automotive industry. In this section we will discuss the techniques used during this project, the tradeoffs of these techniques, as well as the conclusions that can be drawn.

5.1 Analysis of the Virtualization Approach

One of the methods used to work around the issue regarding the virtual microcontroller having a different memory state than the physical microcontroller was to flash the physical ECU and inject that state into the virtual environment. This solved the issue for us but this physical-to-virtual state transfer technique has flaws. In a perfect virtualization, the software starts from address 0 and initializes its own memory, calibrations and stack. By doing the state transfer we did not simulate these processes, but instead the result of the processes.

This method is very brittle. It worked well for us under these specific circumstances. However, if the boot process changes or other calibration values are used in other versions of the BSW, this method will most likely fail. Naturally this process can be redone with a new ECU that is known to be working well, but that is the main issue with this method. There must be a working ECU before testing in the virtual environment. One of the main goals of having a virtual environment is that software can be tested before a physical test rig is present. For this virtual model to be a viable pre-silicon verification tool, the environment must be capable of autonomous initialization rather than relying on state-transfer from physical targets.

Additionally, the solution presented is only viable for the model of ECU used during this thesis. Meaning, for this to be a viable solution for other systems and models of ECU each would require their own version of the memory to be loaded into the virtual environment. This very quickly becomes an issue not only within companies, as many automotive companies produce multiple ECUs, but also if this model was to be applied over the automotive industry by and large.

When reaching the OS execution, the system functions as a dynamic interrupt-driven environment. Before this however, the initialization phase is of a more static nature with a checklist of functions that were to going to be executed. During

the initialization phase the system is relatively predictable, and it progresses in a linear fashion systematically performing checks and proceeding onto the next after completion. These forms of checks are relatively easy to replicate through the usage of stubbing and register injection as we had done.

Once the system has passed the initialization phase it enters the OS state, were execution shifts to asynchronous, event-driven execution. The OS starts a number of different schedulers. It sets hardware timers, like the GTM, and expects to wake up in several different time intervals. At the same time, it expects the CAN controller to be pulling data into the memory and the Interrupt Control Unit (ICU) to be sending signals. Now the software is no longer the main controlling factor, but instead becomes a slave to timing and hardware events, not just in the microcontroller but also in the external peripheral units. If the hardware does not behave as expected by the software following precise timing, the software logic breaks.

This is the reason we saw that the execution on our virtual model entered a terminal state where it first got stuck on the entry tasks of the OS, and then entered a phase of checking for peripheral hardware which never was exited. This is a technical wall that we hit. In the static phase we only had to worry about specific registers that the hardware was looking for at that time. In the dynamic phase the OS activates everything at once. The explosion of dependencies can be caused by different factors. One of these could be that the OS requires the GTM, the System Timer Module (STM) and the DMA to all be synchronized. If the virtual model fails this, the software will regard this as a hardware failure. Another possible explanation is cross module reliance. The OS scheduler might depend on an interrupt from the CAN controller and if that interrupt never comes because the CAN is not fully modeled, the OS hangs. This can lead to the watchdog timer, which is a separate hardware dependency, reacting to this and triggering a hard reset. A third reason why the OS enters a terminal state is that a feedback loop might occur. If one signal is missed this causes the OS to call an Error Handler. The Error Handler tries to write a log, which requires the SPI bus. If the SPI bus is not yet modeled, the Error Handler fails, which triggers a deeper exception.

These issues would not be possible to solve with the techniques already used in this project. There are two main ways to move forward from here. One of them is to model the external peripheral units exactly, to be able to model the entire electrical system with all the complex interconnected timings. To be able to do this you would need detailed descriptions of the chips from the vendors. This would soon turn into a complicated and time-consuming task. Another option would be to focus the stubbing technique on interrupts and have a script handle the different dependencies on the fly.

The modifications done throughout the thesis have proven that it is possible to modify the hardware dependencies of a microcontroller through paravirtualization without performing changes to the BSW. Nevertheless, this is only true for the initial hardware initialization phase before OS execution. While useful for testing simple functionality and dependencies, such as MTU and PMS, we have also shown that as

the complexity of the software execution and dependencies increases these changes are no longer viable. The complexity of most ECUs within the automotive industry make the requirement of an OS customary meaning this can be considered a common issue which will be faced during the process of retrofitting a general production level BSW.

5.2 Trade-offs Between Stubbing and Exact Models

When virtualizing hardware for simulation it is necessary to consider the tradeoffs between using paravirtualization to create a model which is similar but not an exact replica of the physical hardware contra creating a one to one model. As often is the case, time and resources become a main deciding factor. In theory, creating a full virtual twin of a microcontroller and by extension an ECU would yield more precise results in terms of simulation compared to physical execution. However, this increases performance overhead of the simulation significantly and as a result the length of simulation time also increases due to the complexity of the models which are used. Additionally, modeling peripheral circuits introduces a new issue as these rarely are created by the same company that is producing the ECUs. This leads to external ASICs being implemented without full knowledge of the functionality and them becoming a form of hardware black box. Within the automotive industry this is not an uncommon occurrence as ECU complexity leads to many teams working on the same models simultaneously and not having extensive insight into the full hardware.

Creating a simplified model through stubbing specific hardware and removing non essential functionality then offers a reduction in time spent before having a running simulation in addition to minimizing performance overhead. However, if too much functionality is stubbed or removed the integrity of the simulation is also lost. It therefore becomes tradeoff of how much functionality can be removed without critical safety or functionality being lost. Due to the nature of software within the automotive industry being very tightly fitted around specific hardware dependencies and functionality, stubbing can be an extensive process. However, successful paravirtualization makes it possible for full unchanged binaries to run in the virtual environment. As a result the simulation can be deemed to perform similarly to its physical counterpart as both are running the same binaries. In our case, the initial hardware dependencies,MTU and PMS, were mainly checks to ensure normal functionality and could be bypassed through creating models which manipulated the registers required to pass the checks built into the BSW. Here, stubbing is an excellent solution as the complexity of the models required is relatively simple and does not require multiple types of functionality within the same model. Once entering the OS, the previously described ”dependency explosion” makes stubbing more difficult. To bypass this, dependencies can directly be removed within the software however, the integrity of the simulation drops significantly as the same binaries are no longer being executed on hardware and in the virtual environment.

Another issue raised by modifying the BSW directly is at which point is it no longer a virtual simulation of hardware but instead a SiL solution? The advantages of simulating hardware is that it allows for testing before physical hardware is available as well as fault injection testing which is difficult or not possible on hardware. However, to achieve a desirable level of accuracy for these test the virtual model must remain precise compared to the physical components. We have proven that for full simulation of a level 4 virtualization more accurate models of external circuitry are required as simply stubbing and modifying the BSW becomes a far to complex process for accurate simulation. If extensive modification is required, at what point is the virtualization no longer a satisfactory representation of the hardware? This is a research question which we raise for further study. Potentially using a virtual microcontroller which has been modified to different levels and comparing how performance compares to its physical counterpart and determining at what point performance no longer is satisfactory.

5.3 Portability Within the Automotive Industry

Within the automotive industry the introduction of the AUTOSAR standard was hoped to increase the modularity of both hardware and software for automotive solutions. Using modular code and following an industry wide standard could remove many of the issues which we have encountered throughout the thesis. The introduction of modular software blocks could reduce the amount of black boxes and increase collaboration within the industry as well. The increased portability and modularity does however, result in an increase in complexity. Modular software and hardware which follows the AUTOSAR standard is far less specific as it is meant to be applicable for many use cases. The integration therefore becomes far more complex as the standardized modules need to be adjusted to fit the more specific needs of the application.

To strengthen the claims of the specificity with regards to automotive applications the port initialization fault in the VDK discovered in this thesis can be highlighted. The issue which we encountered relating to there being a fault in the provided VDK which had previously not been encountered by other users. The VDK used throughout this thesis is a finished production model which has already been in use for an extended period of time by other users and companies. The fact that no other users have encountered this issue shows that it is a form of hardware functionality which most likely is not taken advantage previously. This strengthens the argument of each company has a very specific set of needs which might not be shared with others making it extremely difficult if not impossible to adjust for every form of application. Through the use of standardization and modularity these forms of specific usage of the hardware can be minimized which ultimately would lead to a decrease in the complexity of retrofitting. Making it a more viable option in terms of virtualization.

In reality however, many companies do not implementing such modularity and

standardization as the implementation is too difficult to justify the amount of work required. The use of legacy software being one of the main factors contributing to the difficulty of implementing modular and standardized software. This is a trend which can be observed not only over the industry as a whole but even within the same company. Projects are rarely built from the ground up but instead, improved upon over time. The effort to convert an entire project to following a new standard would be immense. We have observed that there are projects within the Volvo infrastructure which use the AUTOSAR standard. However, despite this the usage of non standardized software and hardware is still very prevalent as it requires less manpower and time to implement. The same issues which we have run into in terms of strict hardware dependencies and the usage of black box software and hardware means that the act of porting existing software requires an extensive investigation. This is a trend which can be observed over the industry as a whole as many companies still produce hardware and software which does not follow the standard.

5.4 The ”Virtual-First” Paradigm Shift

The main issue encountered during this project was that the BSW was so tightly coupled with the hardware. This made it hard to retrofit the BSW to the virtual environment. When contacting Synopsys, the company providing the tool that we used for the virtualization process, it became clear that these types of tools most often are used to develop a virtual environment alongside the software that is meant to run on it. The BSW is production level software and was never designed to be virtualized. Instead of building it alongside the virtual environment, we tried to force it into a virtual mold. This is a reactive process and we were constantly trying to catch up with the hardware dependencies. This is the main issue with retrofitting. Every time the hardware vendor updates a register or the BSW team adds a new safety check, the virtual environment will break. It is high-maintanence and a low-scalability approach.

In a ”Virtual-First” paradigm, the VDK is the primary development target, not the physical ECU. With this approach, software and hardware are developed in parallel using Co-Verification. Co-Design eliminates the need for the reactive methods that we had to use. If the software is built on the virtual model from the start, the virtual model becomes a tool in the design process of the software, and the physical hardware is the final deployment target.

Currently most BSW is aware of the hardware and it expects specific electrical behavior. To ease the transition into virtual driven development, the software should contain an option to be aware of the virtual environment. This could, for example, be done with a specific configuration flag in the BSW: `IS_VIRTUAL_TARGET = TRUE`. If the BSW is developed with this option and if the flag is set to true, the software could bypass timing-critical loops or use functional shortcuts for complex peripherals like watchdogs. Instead of the ”dependency explosion” where the OS crashes because a timer is not initialized correctly, a simulation-aware OS would trust the virtual timers provided by the simulator. This approach would create a clean separation between functional logic and hardware binding. If the binding is modular, it would

be possible to swap a physical driver for a virtual driver without modifying the core application.

The automotive industry is moving, and has come a long way, toward Software Defined Vehicle (SDV). In this world the hardware is just a generic ”server on wheels” and the software is updated Over the Air (OTA) every month. With this shift in the automotive industry the demand for testing software is increasing constantly, and running thousands of test cases for a global fleet becomes increasingly demanding. Physical rigs are expensive, they break and they take up floor space. For a modern DevOps approach for the workflow, you can run software in the cloud, and instead of doing nightly tests on physical test rigs, spin up thousands of instances of the virtual environment in a server farm and run tests in parallel.

5.5 Architectural Generalization and Industry Implications

While this research focused on the Infineon TC39x, the barriers identified, specifically the ”Dependency Explosion” at OS entry, represent a system challenge in the automotive industry. As the automotive industry shifts toward SDVs, the ”Retrofitting Trap” documented here serves as a cautionary framework for any organization attempting to virtualize hardware-coupled binaries.

Most microcontrollers, like ARM Cortex-R, RISC-V, PowerPC, follow a similar boot flow to the TC39x, with a static initialization followed by the concurrent execution of the RTOS [36][37][38]. The ”Dependency Explosion” that we saw is not specific to the Infineon architecture, but a general characteristic of RTOS-based embedded systems. Our results suggest that the stubbing of initialization functions of external units is universally viable for the initial phase of hardware initialization verification, but insufficient for scheduler level verification. To avoid the ”Dependency Explosion” that we encountered as we got further into the startup sequence of the BSW, simple stubbing of functions is not sufficient, a more rigorous modeling effort with register accurate modeling enabling unchanged BSW execution is required. This is a tradeoff that the industry as a whole could face when it comes to virtualization of these types of systems. Regardless of chip architecture, any virtualized target will face a fidelity threshold once the interrupt controller and system timers are activated.

The different components of the ECU are made by multiple different vendors. In our case, the microcontroller was made by Infineon, while some of the external units were made by Bosh. This translates generally to the automotive industry, where these kinds of hardware components are rarely developed in-house, but rather outsourced to external vendors. This results in trade-offs that automotive companies have to make. On one hand, it is less work and cheaper in the long run to buy hardware instead of develop it in-house. On the other hand, the company gets less insight in the exact structure of the hardware and these units can be treated more as a ”black box” which can create a portability debt. This reliance on ”black box” drivers

makes it more complicated to create portable solutions for the ECU. The authors of the paper "State-of-the-art virtualisation technologies for the centralized automotive E/E architecture" argue that true hardware/software decoupling is the primary technical hurdle for SDVs and require a fundamental shift in how dependencies are managed [8]. Additionally, the authors of the paper "Automating Safety and Security Co-design through Semantically Rich Architecture Patterns" show with their work on automated co-design using architecture patterns that manually stubbing is a time-consuming task and prone to human errors [3]. Our project shows that without a virtual-aware HAL, the cost of virtualization, with the man-hours spent on trying to resolve all the hardware dependencies, may exceed the cost of physical hardware in the short term. For a microcontroller to be truly ready to be virtualized, the HAL must be designed with modular hardware bindings. In that way, the software would be able to point to a virtual or a physical driver at compile time.

Our approach of injecting the virtual memory with extracted memory from the physical ECU is often referred to as Warm-Start Virtualization. This approach relies on having a physical ECU "seed" a virtual model, which creates a circular dependency. You need hardware to test the software that is meant to replace the hardware. For the industry to scale, especially in pre-silicon phases, virtual models must be able to support autonomous initialization and not be dependent on the hardware already being present. This is often called Cold-Start Virtualization, where the virtualized model is initialized on its own in the same way that the physical ECU is when it is turned on. Our findings show that Warm-Start methods can be a viable debugging tool, but is a failed verification strategy.

With the automotive industry moving more and more towards SDV, there is a growing need for parallelized testing in the cloud. In a globalized supply chain, different teams, often in different time zones, share the same software. If the software is "hardware-agnostic" through virtualization, the industry can move toward cloud-native Continuous Integration/Continuous Deployment (CI/CD), which could help standardize testing across global teams and in the end be a cheaper alternative than testing on physical test rigs in multiple locations. For example, if one developer is waiting on one of ten physical rigs in a lab in Sweden, they can spin up 1000 instances of the virtual model in the cloud, which multiple teams can use at the same time. Our project has identified some of the specific technical blockers, with the interrupt synchronization and timer logic, that currently prevent this cloud-native future from being "plug-and-play".

5.6 Conclusion

This thesis investigated the portability of production-level BSW to a virtualized environment, using a Synopsys VDK to model the Infineon TC39x microcontroller. By evaluating the barriers between the virtual model and a functioning execution of the BSW, this research has provided a technical analysis of the gaps between current hardware-centric automotive software and the requirements of a virtual-first development paradigm.

The investigation demonstrated that while virtualization of the initial BSW startup sequence is achievable through register-level stubbing and state injection, a significant barrier for portability still exists at the transition to the OS execution. The success of the project was split across two phases of execution. These were the static phase and the dynamic phase. In the static phase, python-based workaround scripts and memory state replication could bypass early hardware-dependencies such as the PMS and MTU initialization. In the dynamic phase, the project reached a ”technical wall” upon OS entry. The transition from a more linear, procedural initialization to an asynchronous, interrupt-driven environment caused a ”dependency explosion”. This revealed that Level 4 functional stubbing is insufficient for supporting an RTOS without timing-accurate peripheral models.

The primary contribution of this work is the identification of the ”Retrofitting Trap” in automotive virtualization. The results acquired during this thesis shows that attempting to adapt production binaries into a virtual environment is a reactive and brittle process. This research highlights that for virtualization to be a scalable solution, the industry must move toward autonomous initialization within the virtual model.

Furthermore, this thesis raises critical questions regarding the definition of virtualization transparency. By documenting the degree of modification required to reach execution milestones, this work provides a baseline for evaluating the trade-offs between simulation fidelity and development speed.

To overcome the barriers identified in this research, future work should focus on three primary areas. The first of these is the development of a virtual-aware HAL, where the implementation of a BSW configuration flag needs to be investigated. This would allow the software to gracefully handle missing or simplified hardware peripherals without triggering a terminal reset. The second area that needs to be thought of is the possibility to automate interrupt stubbing. This means move beyond manual register scripting and toward an automated framework for handling asynchronous dependencies during the initialization of the OS. The third area to be investigated is the possibility to have cloud-native integration. That is, evaluating how the ”virtual-first” paradigm can be integrated into existing CI/CD pipelines to enable parallelized, high-volume testing of SDV architectures.

The automotive industry is at a crossroads, moving toward a future defined by software rather than hardware. As this thesis has shown, the current reliance on physical hardware for software verification is a bottleneck that cannot be solved by simple retrofitting alone. While the technical challenges of full-scale BSW virtualization are significant, they are not hopeless. The shift toward a ”virtual-first” paradigm, where software is co-designed with its digital twin, is no longer a luxury, but a necessity for the next generation of safe, reliable, and software-defined vehicles.

Bibliography

[1] M. Vogel et al., "Metrics in automotive software development: A systematic literature review," *Journal of Software: Evolution and Process*, vol. 33, no. 2, e2296, 2021, e2296 smr.2296. DOI: https://doi.org/10.1002/smr.2296. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2296. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2296.

[2] M. Kaiser, "Electronic control unit (ecu)," in *Gasoline Engine Management: Systems and Components*, K. Reif, Ed. Wiesbaden: Springer Fachmedien Wiesbaden, 2015, pp. 254–259, ISBN: 978-3-658-03964-6. DOI: 10.1007/978-3-658-03964-6_16. [Online]. Available: https://doi.org/10.1007/978-3-658-03964-6_16.

[3] Y. G. Dantas and V. Nigam, "Automating safety and security co-design through semantically rich architecture patterns," *ACM Trans. Cyber-Phys. Syst.*, vol. 7, no. 1, Feb. 2023, ISSN: 2378-962X. DOI: 10.1145/3565269. [Online]. Available: https://doi.org/10.1145/3565269.

[4] P. Barham et al., "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[5] A. S. Thangarajan, M. Ammar, B. Crispo, and D. Hughes, "Towards bridging the gap between modern and legacy automotive ecus: A software-based security framework for legacy ecus," in *2019 IEEE 2nd Connected and Automated Vehicles Symposium (CAVS)*, 2019, pp. 1–5. DOI: 10.1109/CAVS.2019.8887788.

[6] R. Diane Caballar and C. Stryker. "What is legacy code?" [Accessed: 29 April 2026], IBM. [Online]. Available: https://www.ibm.com/think/topics/legacy-code.

[7] V. Rupanov, C. Buckl, L. Fiege, M. Armbruster, A. Knoll, and G. Spiegelberg, "Early safety evaluation of design decisions in e/e architecture according to iso 26262," in *Proceedings of the 3rd International ACM SIGSOFT Symposium on Architecting Critical Systems*, ser. ISARCS '12, Bertinoro, Italy: Association for Computing Machinery, 2012, pp. 1–10, ISBN: 9781450313476. DOI: 10.1145/2304656.2304658. [Online]. Available: https://doi.org/10.1145/2304656.2304658.

[8] Z. Guo, K. Koufos, M. Dianati, and R. Woodman, "State-of-the-art virtualisation technologies for the centralised automotive e/e architecture," *Frontiers in Future Transportation*, vol. 6, p. 1 519 390, 2025.

[9] H. Askaripoor, M. Hashemi Farzaneh, and A. Knoll, "E/e architecture synthesis: Challenges and technologies," *Electronics*, vol. 11, no. 4, p. 518, 2022.

[10] S. Dingler and F. Boenke, "Toward automated virtual electronic control unit (ecu) twins for shift-left automotive software testing," *arXiv preprint arXiv:2602.18142*, 2026.

[11] A. K. Sundar Rajan and M. Nirmala Devi, "Virtualizing an automotive state-of-the-art microcontroller: Techniques and its evaluation," in *Automotive embedded systems: key technologies, innovations, and applications*, Springer, 2021, pp. 19–36.

[12] H. Kim, J. Kwak, and J. Cho, "Autosar-compatible level-4 virtual ecu for the verification of the target binary for cloud-native development," *Electronics*, vol. 13, no. 18, p. 3704, 2024.

[13] Infineon Technologies AG, *Aurix™ tc3xx user's manual (part 2)*, V 2.0.0, Describes the TC3xx Platform family peripherals and connectivity, Infineon Technologies AG, Munich, Germany, Feb. 2021.

[14] Infineon Technologies AG, *Aurix™ tc3xx user's manual (part 1)*, V 2.0.0, Describes the TC3xx Platform family architecture and core subsystem, Infineon Technologies AG, Munich, Germany, Feb. 2021.

[15] R. B. GmbH, *Automotive handbook*. John Wiley & Sons, 2022.

[16] Synopsys, Inc., *Virtualizer: Vdk creation & deployment tools*, Datasheet, Synopsys, Inc., Mountain View, CA, 2024. [Online]. Available: `https://www.synopsys.com/content/dam/synopsys/gated-assets/verification/virtualizer_ds.pdf`.

[17] N. Navet and F. Simonot-Lion, *Automotive embedded systems handbook*. CRC press, 2017.

[18] S. Yadav, R. T. Ugale, and R. Goyal, "Development of virtual test environment for vehicle level simulation," in *2023 3rd Asian Conference on Innovation in Technology (ASIANCON)*, IEEE, 2023, pp. 1–6.

[19] Synopsys, Inc., *Synopsys automotive vdk for level 4 virtual ecu abstraction*, Datasheet, Synopsys, Inc., Mountain View, CA, 2023. [Online]. Available: `https://www.synopsys.com/content/dam/synopsys/verification/datasheets/automotive-fact-sheet.pdf`.

[20] J. Zhou, W. Han, T. Zhuang, and J. Kong, "High-performance network virtualization with safety for automotive virtualization os," in *2025 IEEE Smart World Congress (SWC)*, IEEE, 2025, pp. 811–818.

[21] M. Abboush, C. Knieke, and A. Rausch, "A virtual testing framework for real-time validation of automotive software systems based on hardware in the loop and fault injection," *Sensors*, vol. 24, no. 12, p. 3733, 2024.

[22] J. Reitz, A. Gugenheimer, and J. Roßmann, "Virtual hardware in the loop: Hybrid simulation of dynamic systems with a virtualization platform," in *2020 Winter Simulation Conference (WSC)*, IEEE, 2020, pp. 1027–1038.

[23] R. Debouk, "Overview of the second edition of iso 26262: Functional safety—road vehicles," *Journal of System Safety*, vol. 55, no. 1, Mar. 2019. DOI: `10.56094/jss.v55i1.55`. [Online]. Available: `https://jsystemsafety.com/index.php/jss/article/view/55`.

[24] Infineon Technologies AG, *Aurix™ tc3xx microcontroller training: Mcmcan can interface*, version V1.0, Training Document, Infineon Technologies AG, Sep. 2020. [Online]. Available: `https://www.infineon.com/assets/row/public/`

documents/10/56/infineon-aurix-tc3xx-can-interface-training-en.pdf?fileId=5546d46274cf54d50174da4ee8cb226c.

[25] N. Amringer and P. Asemann, "Simulation of virtual ecus in the context of ecu consolidation," in *Proceedings of the 9th AutoTest Technical Conference: Test of Hardware and Software in Automotive Development*, Stuttgart, Germany, Oct. 2022. [Online]. Available: https://www.researchgate.net/publication/364694688_Simulation_of_Virtual_ECUs_in_the_context_of_ECU_Consolidation.

[26] V. Weaver, "Are cycle accurate simulations a waste of time," 2008.

[27] A. Schlie, C. Wille, and I. Schaefer, "Virtual prototyping for early development of automotive software," in *2018 IEEE/ACM 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, Describes the use of SystemC and TLM-2.0 for register-accurate modeling to enable unchanged BSW execution., 2018, pp. 34–37. DOI: 10.1145/3196478.3196484.

[28] S. Biewer, P. Scheurer, J. Haupenthal, and H. Hermanns, "Introduction to automated hardware-in-the-loop testing," *IEEE Design Test*, vol. 35, no. 3, pp. 43–50, 2018, Discusses the capture of hardware signals to create simulation models for components where internal documentation is unavailable. DOI: 10.1109/MDAT.2018.2810332.

[29] M. Becker, D. Dasari, V. Nélis, M. Behnam, L. M. Pinho, and T. Nolte, "Investigation on autosar-compliant solutions for many-core architectures," in *2015 euromicro conference on digital system design*, IEEE, 2015, pp. 95–103.

[30] M. Staron and D. Durisic, "Autosar standard," in *Automotive Software Architectures: An Introduction*. Cham: Springer International Publishing, 2017, pp. 81–116, ISBN: 978-3-319-58610-6. DOI: 10.1007/978-3-319-58610-6_4. [Online]. Available: https://doi.org/10.1007/978-3-319-58610-6_4.

[31] R. Boot, J. Richert, H. Schutte, and A. Rukgauer, "Automated test of ecus in a hardware-in-the-loop simulation environment," in *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design (Cat. No.99TH8404)*, 1999, pp. 587–594. DOI: 10.1109/CACSD.1999.808713.

[32] A. Yang, W. J. Han, H. S. Cho, D.-W. Koh, and J.-G. Kim, "A model-based design and verification framework for virtual ecus in automotive seat control systems," *Electronics*, vol. 15, no. 3, 2026, ISSN: 2079-9292. DOI: 10.3390/electronics15030569. [Online]. Available: https://www.mdpi.com/2079-9292/15/3/569.

[33] M. R. Kabir and S. Ray, "Virtualization for automotive safety and security exploration," in *2023 IEEE 16th Dallas circuits and systems conference (DCAS)*, IEEE, 2023, pp. 1–4.

[34] S. Kulandaivel, T. Goyal, A. K. Agrawal, and V. Sekar, "{Canvas}: Fast and inexpensive automotive network mapping," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 389–405.

[35] A. Bogorin-Predescu, A. M. Ţıţu, N.-M. Niţă, and V.-L. Domnariu, "Modeling of the automatic testing process of electronic control units in the automotive industry," *International Journal of Mechatronics and Applied Mechanics*, no. 12, pp. 148–155, 2022.

[36] Advanced Micro Devices, Inc. “Arm cortex-r5f and cortex-r52 processor boot code.” Describes low-level boot and initialization sequence for Cortex-R processors, Accessed: May 13, 2026. [Online]. Available: https://docs.amd.com/r/en-US/oslib_rm/Arm-Cortex-R5F-and-Cortex-R52-Processor-Boot-Code.

[37] DeepWiki. “Risc-v boot process overview.” Overview of multi-stage boot flow for RISC-V platforms including ROM, bootloaders, and OS startup, Accessed: May 13, 2026. [Online]. Available: https://deepwiki.com/riscv/meta-riscv/5.2-boot-process.

[38] S. Lee. “Rtos bootloader essentials: A deep dive.” Explains role of bootloader in hardware initialization and RTOS startup, Accessed: May 13, 2026. [Online]. Available: https://www.numberanalytics.com/blog/rtos-bootloader-essentials-deep-dive.