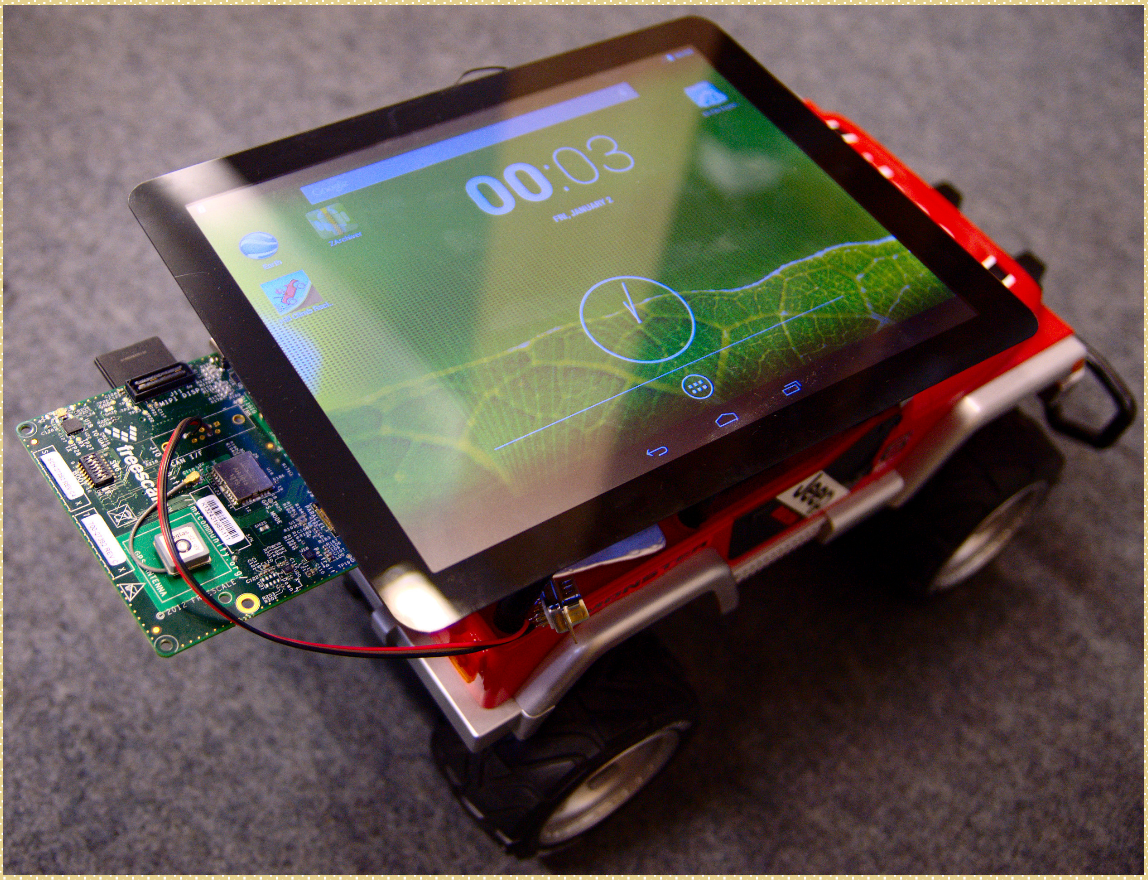


João Filipe Marques Serra

Multi-criticality Hypervisor for Automotive Domain

Master of Science in Electrical and Computer Engineering

July 2014

· U C ·

Universidade de Coimbra

· U C ·

Departamento de Engenharia Electrotécnica e de Computadores
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

A Dissertation
for Graduate Study in MSc Program
Master of Science in Electrical and Computer Engineering

Multi-criticality Hypervisor for Automotive Domain

João Filipe Marques Serra

Work Developed Under Supervision of
Prof. António Paulo Mendes Breda Dias Coimbra (DEEC-UC),
Prof. Tony Richard de Oliveira de Almeida (DEEC-UC),
Eng. Jorge Manuel Simões de Almeida (CSW)

Jury
Prof. Jaime Batista dos Santos (President)
Prof. António Paulo Mendes Breda Dias Coimbra (Vowel)
Prof. Mário Gonçalo M. V. Silveirinha (Vowel)

July 2014

Partially supported by the ARTEMIS-JU CONCERTO project (n.333053).
Work developed at Critical Software S.A., Coimbra.

Thanks

Never explain yourself. Your friends don't need it and your enemies won't believe it.
(Belgicia Howell)

At the end of so important stage for me, I can not let pass this opportunity to thank all the people that helped and supported me over the last years.

Thus, I want to thank all the colleagues I have worked with, both during the course and in Critical Software and iTGROW. Without them, many battles would have been harder and less pleasant to pass by than they actually were.

Then I extend my warmest thanks to my coordinators, Prof. Paulo Coimbra, Prof. Tony de Almeida and Eng. Jorge Almeida, whose support has been crucial to the achievement of this work.

Finally, I leave my deepest gratitude to my family and my friends, who are the people that supported me everyday, in the good and bad moments.

Abstract

It is wise to keep in mind that neither success nor failure is ever final.
(Roger Babson, American educator)

xLuna is a real-time kernel technology that enables concurrent mixed-criticality applications running simultaneously on the same hardware platform, bridging a safety critical application, hard-real-time task set and certifiable real-time operating system alongside a feature rich, non-secure, non-critical, non-real-time general purpose operating system.

The xLuna program, an hypervisor originally developed by Critical Software for space applications, has now a new development focused on the automotive domain extending the principles of its predecessor to new, multi-core hardware. To showcase this new approach, a certifiable real-time operating system is responsible to handle an eCall application, which is an European in-vehicle emergency call service, while running simultaneously with the general purpose operating system Android OS on the same hardware platform. This is done by sandboxing Android into a hardware enforced container. Space and time resource constrains are predefined, such as core affinity and static memory allocation.

The motivation of xLuna for automotive domain is to reduce the number of Electronic Control Units present in cars without reducing the number of features that they offer, by using a multi-core platform, allowing both critical and non-critical systems to run together in the same hardware platform.

This work focused on the solutions presented by the hardware used, and how xLuna takes advantage of the technologies it brings to enforce software and resource partitioning. It was accomplished to have Android running simultaneously alongside the real-time operating system FreeRTOS, each one running on only one core. While Android is responsible to offer an automotive infotainment experience, one FreeRTOS's task is responsible to simulate the eCall system, sending a SMS to a pre-determined mobile phone number when high acceleration values are detected.

The use of a modern multi-core platform running a mixed-criticality system is a emerging need for today's automotive domain. Integrating the eCall system with a system like this makes xLuna for automotive domain a great solution for this market.

Keywords: Embedded systems, mixed-criticality, multi-core, hypervisor, automotive, infotainment, eCall.

Resumo

É sábio ter em mente que sucesso e fracasso nunca são definitivos.
(Roger Babson, educador Americano)

xLuna é uma tecnologia de *kernel* de tempo-real que permite que aplicações com criticidade mista corram em simultâneo na mesma plataforma de *hardware*, juntando um sistema operativo de tempo-real certificável, contendo um conjunto de *tasks* de tempo-real rígido e aplicações de segurança crítica, com um sistema operativo não crítico, não seguro, rico em funcionalidades e que não é de tempo-real.

O programa do xLuna, um *hypervisor* originalmente desenvolvido pela Critical Software para aplicações destinadas para o espaço, tem agora uma nova abordagem focada no domínio automóvel, extendendo os princípios do seu antecessor para um novo *hardware* com um processador *multi-core*. Para dar a conhecer esta nova abordagem, um sistema operativo de tempo-real certificável é responsável por gerir uma aplicação eCall, que é um serviço de emergência europeu para veículos, enquanto que este corre simultaneamente com o sistema operativo Android na mesma plataforma. Isto é feito sendo o Android seguramente contido num *container* criado por auxílio do *hardware* presente na plataforma usada. Restrições aos recursos de tempo e de espaço são pré-definidas, tal como a afinidade aos *cores* do processador e a alocação da memória estática.

A motivação do xLuna para automóveis é reduzir o número de Unidades de Controlo Eletrónicas presentes nos automóveis sem reduzir as funcionalidades que elas oferecem, recorrendo à plataforma *multi-core*.

Este trabalho focou-se nas soluções fornecidas pelo *hardware* usado, e como o xLuna tira vantagem das tecnologias fornecidas para forçar o particionamento do *software* e dos recursos. Foi conseguido ter o Android a correr em simultâneo com o sistema operativo de tempo-real FreeRTOS, cada um deles a correr num só *core*. Enquanto que o Android é responsável por oferecer uma experiência rica em aplicações de multimédia, entertenimento e navegação, uma *task* do FreeRTOS é responsável por simular o sistema eCall, enviando uma SMS para um destinatário pré-definido assim que são detectados elevados valores de aceleração.

O uso de uma plataforma *multi-core* moderna a correr um sistema de criticidade mista é uma necessidade emergente para o domínio automóvel dos dias de hoje. Integrando o sistema eCall com um sistema deste tipo torna o xLuna para automóveis uma excelente solução para este mercado.

Palavras-chave: Sistemas embebidos, criticidade mista, *multi-core*, *hypervisor*, automóvel, *infotainment*, eCall.

Contents

Chapter 1

Introduction

If you have an apple and I have an apple and we exchange apples then you and I will still each have one apple. But if you have an idea and I have one idea and we exchange these ideas, then each of us will have two ideas.
(George Bernard Shaw, Irish dramatist)

This dissertation is based on an internship in business environment, more specifically at Critical Software S.A. (CSW) through iTGROW A.C.E.. Firstly, I had to apply for it and after an interview, a few tests and group activities, I was chosen. The internship opportunity that I had applied for was for the Mars Robotic Exploration Preparation (MREP) project where I would have to develop and perform benchmarks to a microprocessor that will integrate a rover designed to explore Mars. Because of some problems unrelated to CSW, this project was delayed for almost a year. This situation forced CSW to change my internship plan. Nevertheless, I stayed a few weeks at CSW studying and researching on this subject.

The new internship subject was related with a new approach of a CSW's product, the xLuna (described is section 2.1). Initially, this new approach xLuna was supposed to be an hypervisor[1] of type 2 for automotive domain. It would run within a real-time operating system (RTOS),

[1]Hypervisor is a piece of computer software, firmware or hardware that creates and runs virtual machines. They are classified into two types: type 1, which runs directly on the host's hardware to control the hardware and to manage guest operating systems, and type 2, which runs within a conventional operating system environment. More information please refer to [1].

responsible to handle the eCall system (described is section 2.3), and it would manage a guest operating system (OS), responsible for automotive infotainment[2], both running simultaneously on the same multi-core hardware (HW) platform. These OSs were planned to be QNX® Neutrino® RTOS (section 2.2.2) and Android [2], respectively the RTOS and the guest OS. But after some work and research on the HW and on the OSs chosen, some initial ideas were dropped leading to the choice of a new RTOS, the FreeRTOS^{TM} [3], and a change of type of hypervisor, now a type 1 hypervisor, for xLuna. In addition to this, xLuna also implements the eCall system. The first full working version of xLuna was accomplished at the end of last February. From that date on, the development team was reduced and those who stayed started working on new features and others details for better performance and security provided by xLuna.

This project allowed me to consolidate the knowledge acquired throughout the course by exploiting the concepts of operating systems and real-time operating systems to the limit.

In addition, this project was also very enriching for me regarding the first-job and team-work experience in the business environment but also regarding the automotive domain, which is a very interesting business area and that I will like to get involved into in the future.

1.1 Motivation

Nowadays, embedded systems are present everywhere. In cars, for example, there are dozens of ECUs (Electronic Control Units) that control one or more electronic safety systems, such as Anti-lock Braking System (ABS) or collision avoidance system, i.e., real-time safety critical systems. Now, with the introduction of in-car multimedia, navigation, web and social capabilities, the number of ECUs tends to increase even further, creating an ever growing need to reduce that number, minimizing production costs, cars' weight and energy consumption, without impacting performance, number of features offered and, above all, safety.

Multi-core processors can be a solution for that need, providing that systems with different safety integrity level (e.g., ABS and the Radio) do not coexist at the same processing unit. For this, it is necessary to isolate systems that would put human life at risk and, as such, have an higher level of certification requirements (such as ISO26262), from systems where a failure would

[2]Automotive Infotainment, or In-Vehicle Infotainment (IVI), is a collection of hardware devices present in nowadays' cars, or other kinds of vehicles, to provide audio-visual entertainment and automotive navigation systems.

not pose any threat to safety (e.g., the multimedia system). This isolation is usually provided by containing the non-critical system through the use of an hypervisor, allowing both critical and non-critical systems to run together in the same hardware (HW) platform.

1.2 Objectives

My role in this project was to develop some device drivers, like UART (Universal Asynchronous Receiver/Transmitter) and FlexCAN (Flexible Controller Area Network), and to deal with interrupts, by mapping and handling them. Furthermore, I supported the others members of the project whenever they needed me to. These tasks required a lot of knowledge about computer science, C programming language and assembly, more specifically ARM®[3] assembly.

The objectives of this dissertation were based on the xLuna project unique milestone. This milestone was to have a demonstrator of xLuna, which would run Android on one core and FreeRTOS on another, and simulate an eCall application, ready to be presented at BMW Supplier Event 2014, in February 2014.

1.3 Summary Of The Work Developed

At the beginning, it was written three tutorials – two that explained how to create an *Hello World!* application for both QNX and Android, and the other that explained how to build the Android source-code files. These tutorials are present in appendix (chapter A).

After that, when the development team of xLuna was complete, it started to be done the development and implementation of xLuna for automotive domain, in order to have the demonstrator ready for the milestone. This work included the development of the UART device drivers and the mapping and handling of the interrupts. The development of the device drivers of FlexCAN was done after the milestone, because this module did not entered in the demonstrator.

The work done by the others members of the development team had focused on the allocation of the static memory, the development of others device drivers and USB stack, etc. Apart from me and the *Project Manager*, the development team was composed by two senior engineers and

[3]ARM is a very popular family of computer processors based on a reduced instruction set computing (RISC) architecture. More information can be found in [4].

one junior engineer.

When the milestone was reached, a research work about AUTOSAR software architecture (described in section 3.1) started in order to study the integration of an AUTOSAR application in xLuna. A document that describes this study is presented in appendix (chapter A).

It was also written a paper about the xLuna for automotive domain (Multi-criticality Hypervisor for Automotive Domain). This article was submitted to INForum 2014, an event dedicated to Portuguese scientific works related with computer science. This paper is currently under evaluation.

1.4 Structure Of The Thesis

In addition to the current chapter, this document is split in five different chapters. They are the following:

- *Technical Background* – it is described the architecture of the xLuna for automotive domain and the projects that are related with it, as well as a market research;
- *Hardware, Software and Technologies Used* – in this chapter, the most relevant hardware, software and other technologies used in this work are properly detailed;
- *Development and Implementation* – this chapter contains all the work done related with the development and implementation of xLuna, as well as the problems found and its solutions;
- *Results* – in this chapter, it is showed the results of the work done in the latter chapter;
- *Conclusions* – it is commented the results obtained and described what future work can be done.

Chapter 2

Technical Background

An investment in knowledge pays the best interest.
(Benjamin Franklin)

In this chapter, it is described all the theory and background that supported the development and the implementation of xLuna Automotive, as well as the first xLuna architecture based on xLuna Space.

2.1 Introduction of xLuna

xLuna [5] is a real-time kernel technology that enables concurrent mixed-criticality applications running simultaneously on the same HW platform, bridging a safety critical application, hard-real-time (HRT) task set and certifiable RTOS, alongside a feature rich, non-secure, non-critical, non-real-time general purpose OS. The xLuna program, an hypervisor originally developed by CSW for space applications, has now a new development focused on automotive domain that extends the principles of its predecessor to new HW. In this new approach, the RTOS is responsible for simulating the eCall system, and it runs simultaneously with Android on the same platform, removing privileges of Android to access the memory space of the RTOS and splitting the cores of a multi-core processor to each OS.

This new approach of xLuna consists on an hypervisor of type 1 with a small footprint and uses ARM TrustZone® technology (section 2.2.3) to integrate FreeRTOS with Android to run on a Freescale™'s i.MX 6 Quad (section 3.3.1), an ARM Cortex™-A9 [6] based system-on-chip (SoC), platform. FreeRTOS is a free version of the certifiable RTOS[1] SAFERTOS® that provides HRT applications and interface with vehicle networks. Android brings a well-known platform for easier and faster application deployment as well as vendor supplied HW drivers for proprietary non-critical components, such as wireless communications, 3D graphics and sound interfaces, for a richer infotainment experience (Figure 2.1).

Figure 2.1: Preview of xLuna Automotive implementation in-car.

Furthermore, xLuna will hereafter implement an eCall application. eCall is a pan-European in-vehicle emergency call service based on 112, the common European Emergency number. It was created to improve the time response of emergency teams to car accidents all over European Union and it will be mandatory in all new cars commercialized in the European Union after October 2015 [7]. More information please refer to section 2.3.

The first and only milestone of xLuna was to have a demonstrator ready to be presented at BMW Supplier Event 2014, in München, Germany, in February 2014. This demonstrator would run Android on one core and FreeRTOS on another, and simulate a car accident and the eCall system, by sending a SMS for a predetermined mobile phone number when high accelerations were detected by a 3-axis accelerometer. Figure 2.2 shows the CSW's stand dedicated to xLuna Automotive.

This work has been partially supported by the ARTEMIS-JU CONCERTO project (n.333053) (section 2.4.1). This project aims to create a reference multi-domain architectural framework for complex, highly concurrent and multi-core systems. The work developed in xLuna has the objective to implement one of the Run Time Environments for the CONCERTO modelling language.

[1]A certifiable RTOS, in this case, it a RTOS that is pre-certified with safety and real-time standards.

CRITICAL Software has worked on more than 70% of ESA missions over the last ten years
Safety-critical & end-user applications in one system
DRIVING THE FUTURE IN VEHICLE INTEGRATION
- Fault injection & robustness testing
- Testing flexibility
- Hardware and software in the loop testing
- Functional safety assessment
- ISO 26262 compliance
Have you tried to do a recall in space?
DEPENDABLE TECHNOLOGIES FOR CRITICAL SYSTEMS
PORTUGAL

Figure 2.2: Stand of xLuna Automotive in BMW Supplier Event 2014.

2.2 xLuna Automotive Architecture

The xLuna Automotive emerged from the idea of applying the xLuna Space ideology to the automotive domain. In xLuna Space, the xLuna hypervisor places itself alongside the RTOS RTEMS[2] with the purpose to virtualize the well-known OS Linux, on an HW platform with a LEON2[3] processor.

2.2.1 Initial architecture

In the beginning, xLuna Automotive was supposed to have the same ideology of xLuna Space but with different OSs and HW – the QNX® Neutrino® RTOS (section 2.2.2) instead of RTEMS, Linux replaced by Android OS and, finally, the i.MX6Q processor instead of the LEON2.

[2]The Real-Time Executive for Multiprocessor Systems (RTEMS) is an open source fully featured RTOS that supports a variety of open standard application programming interfaces and interface standards such as POSIX and BSD sockets [8].

[3]LEON2 is a synthesisable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The model is highly configurable, and particularly suitable for SoC designs [9].

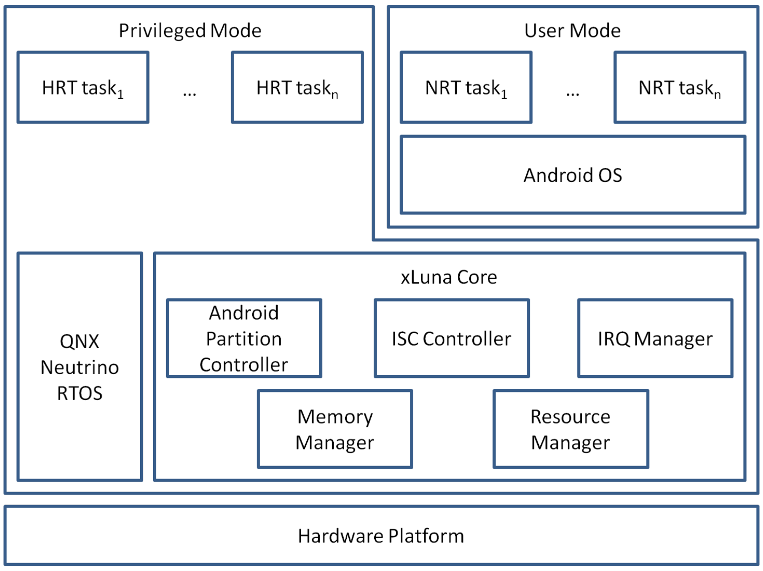


Figure 2.3: xLuna Automotive based on xLuna Space.

In Figure 2.3, it is represented the xLuna Space architecture modified for this new approach of xLuna. With this architecture, xLuna could be classified as an hypervisor of type 2, because it would use the RTOS features in order to provide its own functionalities, so that Android could be virtualized inside a task of the RTOS. These functionalities would be implemented in xLuna core, which would be composed by five different modules. These modules are described below:

- Android Partition Controller – the purpose of the Partition Controller would be to manage the partitioning service by using the needed QNX resources (like tasks) and by catching the needed interrupts to produce the necessary events to be delivered to Android;
- ISC (Inter-Systems Communication) Controller – the ISC Controller would provide synchronous/asynchronous bi-directional communication between HRT tasks, in the QNX space, and NRT (Non-Real-Time) tasks, in the Android space;
- IRQ Manager – the main purpose of the IRQ manager would be to process the interrupts from and to the Android kernel;
- Memory Manager – this module's purpose would be to satisfy the requirements of memory protection between QNX and Android, between Android kernel and Android processes, and among Android processes;
- Resource Manager – this module, which would be based on QNX's Resource Manager module, would allow the Android kernel to access additional hardware that would not be

managed neither by the Memory Manager module, neither by the Partition Controller.

This approach for the architecture of xLuna Automotive required more SW development and greater amount modifications in the RTOS kernel source code than the ideology adopted (section 2.2.4).

2.2.2 QNX Neutrino RTOS

QNX Neutrino RTOS [10] is a RTOS very used along the cars manufactures [11], which was the main reason why it was chosen to be the RTOS of the xLuna Automotive. As a micro-kernel-based OS, it is based on the idea of running most of the OS in the form of a number of small tasks. This allows developers to turn off any functionality they do not require without having to modify the OS itself – the tasks responsible for that functionalities simply will not run. In addition, QNX Neutrino RTOS is a message-passing-based OS, which means that the tasks communicate with each others through message-passing.

The QNX kernel only contains CPU scheduling, IPC (Inter-Process Communication), interruption redirection and timers, which means that everything else is done by user-space tasks. There is a special task within user-space task set known as `proc()`, which performs task creation and memory management by operating in conjunction with the kernel.

There is no device drivers in the kernel. In this case, the developers need to use QNX's Resource Manager (RM) module, whose goal is to present an abstract vision of some device. It consists on using standard POSIX[4] function calls that get translated to messages and then sent to RM.

QNX also allows the developers to create the interrupt handlers, or ISRs. The ISRs are a piece of code that determines what should happen when a specific interrupt is detected. They are called by the kernel and when they have finished, they tell the kernel what should it do by returning `NULL` if nothing is required to be done or an event otherwise. This event is then delivered to a new task.

Regarding the xLuna's approach considered until the time that this research was made, xLuna required a great amount of modifications in the RTOS source code in order to be possible to

[4]POSIX (Portable Operating System Interface) defines the application programming interface, along with command line shells and utility interfaces, for software compatibility with variants of Unix and other OSs [12].

implement the xLuna core. What is described above allowed the xLuna's development team to understand that QNX could not allow that modifications to be done or even provide the features that xLuna needed.

2.2.3 ARM TrustZone

Nowadays, there is an emerging use of multi-core in every kind of embedded systems, such as smartphones, tablets, TVs, cars, etc.. These systems handle highly sensitive data or information, like banking credentials of theirs users, for example. In addition, they are also becoming open software (SW) platforms that allow users to download and use third-party applications that can put these devices, and the data that are handled by these, in risk. In order to a device like these become a secure open SW platform, both HW and SW must work together to robustly secure the device from different types of attack.

The ARM TrustZone technology [13], which is present in ARM Cortex-A series processors, was developed to accomplish this security need. It provides the ability to create two environments or worlds, as referred in TrustZone terminology, with different privileges – the secure world and the normal world – providing a new HW assisted SW separation solution, splitting the already existing privileged/unprivileged domains (Figure 2.4).

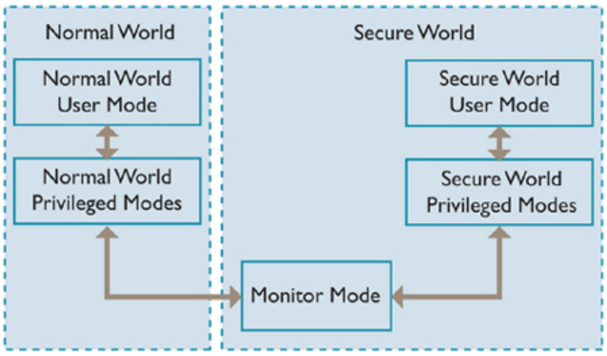


Figure 2.4: TrustZone technology architecture [13].

TrustZone also avoids the performance penalty of a world switch every time an exception occurs in a different context of its destination world because each world contains its own exception handlers. The world switching is done by a CPU exception mode called *monitor*. This mode is entered by calling ARMv7A 'smc' (Secure Monitor Call) instruction. This instruction can be called from either the secure or normal worlds or by a secure IRQ (Interrupt Request)/FIQ (Fast

IRQ)[5] that would require a world switch by saving the current world context and restoring the context of the other world.

Contextualizing TrustZone into the xLuna's ideology, the development team noticed that TrustZone could have a major role on xLuna Automotive, because it can provide almost all of the features of the xLuna core modules.

2.2.4 Final architecture

Regarding sections 2.2.1, 2.2.2 and 2.2.3, it was decided to drop the idea for the xLuna Automotive architecture and the RTOS. Thus, xLuna is now using ARM TrustZone and an open-source RTOS. It was chosen the FreeRTOS because of its simplicity and, consequently, the fact that it is a certifiable RTOS. Therefrom, Figure 2.5 depicts the final architecture of the xLuna Automotive.

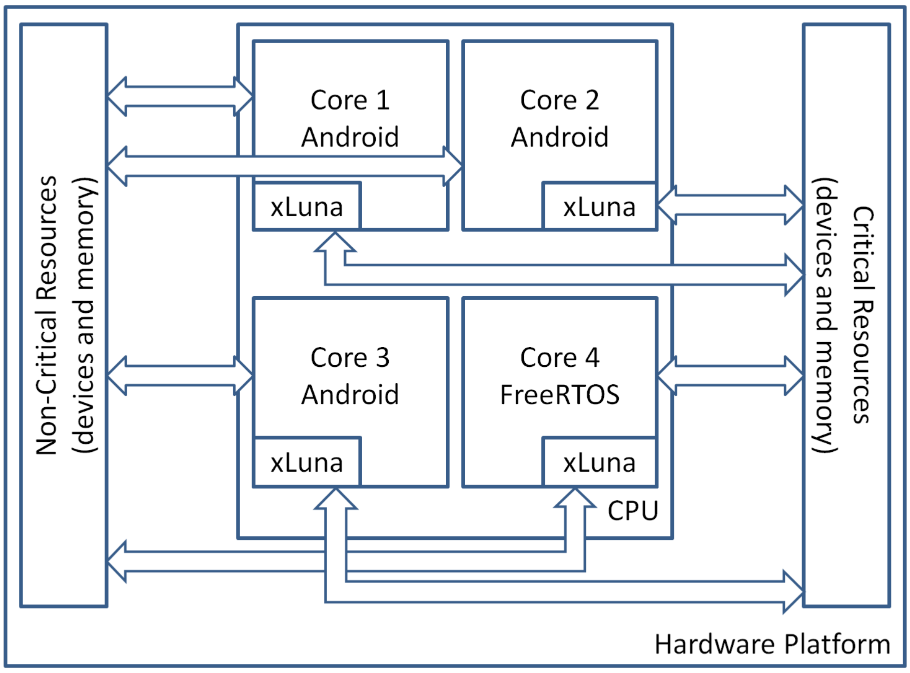


Figure 2.5: xLuna hypervisor implementation on a multi-core hardware platform.

xLuna thrives from TrustZone technology (section 2.2.3) placing itself in the monitor exception routines. It manages a fast, secure and certifiable entry point enabling non-critical SW to be sandboxed[6] into this HW enforced container, while ensuring the safety-critical environment has

[5]An FIQ is a feature of the modern ARM processors and it consists on an higher priority interrupt request. It is prioritized by disabling IRQ and other FIQ handlers during request servicing.

[6]In computer security, a Sandbox is a security mechanism for separating running programs. It is often used to execute untested code, or untrusted programs from unverified third-parties, suppliers, untrusted users and

no impact whatsoever. The fact of being an HW implemented container makes the SW required for managing this container substantially less than other virtualization options. Regarding that, xLuna can be easily converted into a certifiable solution while avoiding any performance penalties of SW emulations.

In order to manage the context switch and to pass the control from one world to the other, a secure Application Programming Interface (API) provides system calls, or monicalls (Monitor Calls) as referred in xLuna terminology, with a specific register set to pass as arguments. Setting these registers, followed by the ‘smc’ instruction, will make the current core to jump to the monitor exception handler where xLuna will then securely switch that core from non-secure to secure worlds and call the corresponding handling code to process the requested monicall and arguments.

xLuna also implements an Inter-Core Communication (ICC) mechanism that allows the secure SW to perform a secure request to a target core. It can for example stop/halt the non-secure SW in the target core specified by sending a call to that core.

As it is shown in Figure 2.5, in xLuna Automotive three cores of the processor are dedicated to Android and the fourth is dedicated to FreeRTOS. This attribution was made due the fact that Android needs more computation resources than FreeRTOS. It can be changed any time by using xLuna’s ICC to stop an Android core and then to start it again dedicated to FreeRTOS. Memory is also split into two regions – a larger one for Android and the other for FreeRTOS.

2.3 eCall

eCall is an electronic safety system that calls automatically the emergency services in case of a serious car accident, which is going to work all over the European Union (EU) plus Iceland, Norway and Switzerland.

An eCall event can be triggered automatically, when a car accident happens, or manually by simply pushing a button. Then, it is immediately established an 112 emergency call to the nearest emergency centre, Public-Safety Answering Point (PSAP) (Figure 2.6). It is also transmitted a Minimum-Set-of-Data (MSD) message over IP to that centre containing the geographic coordinates of the accident scene, the time of the occurrence, vehicle description, and others

untrusted websites [14].

eCall details, such as message ID, the trigger mode, etc.

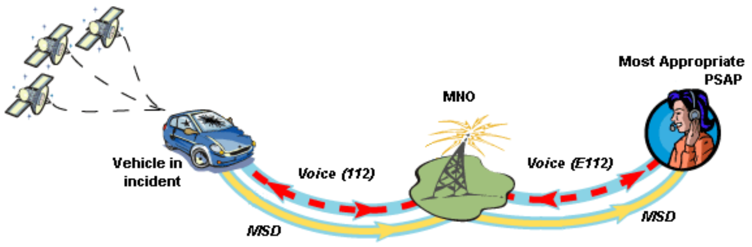


Figure 2.6: eCall system [15].

There are three participants involved in an eCall event – the car, the Mobile Network Operator (MNO) and the PSAP. Each one of them has to be prepared to support the eCall system by following a set of standards. So, xLuna will need to follow the standards dedicated to the cars' manufacturers – CEN EN 15722, 16072 and 16062.

The general benefits of eCall are the following:

- Faster response to the rescue;
- Faster implementation of the Traffic Incident Management process;
- Reductions in secondary incidents;
- More accurate information;
- Increased information for first responders provided by the MSD.

Furthermore, eCall will also result in faster treatment of injured people, by giving accident victims better recovery prospects. In financial terms, every year the EU's economic loss caused by road accidents amounts to more than €160 billion, but if all cars were equipped with the eCall system, up to €20 billion could be subtracted to that value [16].

In numbers, it is expected that eCall saves several hundred lives in the EU each year by reducing emergency services' response time by 50% in rural and 40% in urban areas [16].

eCall system is developed by the HeERO (Harmonised eCall European Pilot) project. Its main objective is to prepare for the deployment of the necessary infrastructure in Europe with the aim of making the eCall a reality. This project, which is partially funded by the European Commission, was divided in two phases – HeERO 1 and HeERO 2. The first lasted for three years, from January 2011 to December 2013, and it was composed by a nine European countries consortium. The second phase is currently on going and it is composed by six new countries plus the other nine pilot countries of HeERO 1. For more information, please refer to [17].

All EU member states are invited to be represented at the European eCall Implementation Platform (EeIP) in order to report on their national status of eCall deployment. Portugal entered the eCall pan-European project in 18 September 2007 [18]. In January 2011, it was created a implementation platform for eCall system in Portugal (PIeCall.pt) but apparently it was dropped because it could not be found any more information about it since its creation. Portugal did not entered in HeERO 1 neither in HeERO 2.

For more information about eCall, please refer to [15].

2.4 State of the art

This section will present and contextualize the other project related with xLuna. Furthermore, it will be listed and analysed the concurrence of xLuna, specially in automotive domain, and their main features. It will be also discussed the entrance of xLuna in the market and its strengths and weaknesses.

2.4.1 CONCERTO

As mentioned in sections 1.1 and 2.2.3, embedded systems are present everywhere and they are starting to use processors with a multi-core architecture. This kind of architecture can have a very important rule in the future of safety-critical real-time systems. To make it possible, across-the-board advances are demanded in all areas of design and development. CONCERTO will combine component-based design with model-driven development in order to deliver a reference multi-domain architectural framework for these systems, without compromising real-time, dependability and energy consumption restrictions.

CONCERTO (Content and cOntext aware delivery for iNteraCtive multimEdia healthcaRe applicaTiOns) is an European project partially funded by the ARTEMIS Joint Undertaking and composed by a European consortium. It aims to improve Model Driven Engineering[7] practices

[7]Model-Driven Engineering (MDE) is a software development methodology which focuses on creating and exploiting domain models, rather than on the computing concepts. For more information, please refer to [19].

and technologies for embedded systems. It consists on the creation of a framework to elevate the abstraction of increasing embedded systems complexity in automotive, aerospace, medical, petroleum and telecom domains. This can be done by following the next three aspects:

- Providing reference designs and architecture to enable a compositional approach to system construction for high performance multi-core platforms with monitoring at run-time;
- Providing a modelling framework developed to support a multi-view, hierarchical design space, with incremental development and early verification of extra-functional properties;
- Developing computing platforms for embedded systems to increase cross-domain reuse and interoperation through platform-independent compositional development of highly concurrent, complex multi-core systems.

The big impact of CONCERTO in the market is to provide the above mentioned capabilities to enable SW developers to take advantage of the power of newer and more advanced HW platforms without substantial increases on the complexity of system design. Therefore, European technology providers will benefit from reduced time-to-market despite the increasing contribution of embedded systems and SW and their increasing size and complexity. Advanced HW modelling capabilities to capture the full potential of new multi-core platforms will also increase the quality and reliability of European embedded systems based products and services, as well as they will provide new and innovative functionalities for the user, while providing tools to ensure high quality and highly reliable systems.

The main focus of CSW in CONCERTO project is to research on the extension of CONCERTO to the automotive domain, in particular to study the use of hypervisors for mixed-criticality multi-core systems. Regarding these, CSW will leverage on knowledge obtained in the aerospace domain and re-apply it to the automotive market. In addition, the development of a small demonstrator for a use case related to the automotive domain is also responsibility of CONCERTO's partners that have strong automotive domain experience, which includes CSW. This use case will encompass infotainment systems and safety-critical functionalities in the same ECU, in order to demonstrate the possibilities of having mixed-criticality environments running simultaneously on the same HW platform without interferences.

For more information about CONCERTO, please refer to [20].

2.4.2 Existing competitors solutions

In this subsection it is presented a market analysis of multi-core virtualization solutions, with a greater focus on the automotive related ones. The main goal of this analysis is to compare xLuna Automotive with its competitors in order to know what are its advantages and disadvantages.

Continental Higher System Integration

This product was developed by Continental, together with Sysgo Continental, aiming to integrate vehicle systems with multimedia applications, mobile entertainment technology, internet and cloud computing in a single vehicle domain. This product consists on an hypervisor that control different SW worlds, like AUTOSAR systems, GENIVI[8] or/and Android applications, running together on only one HW platform. To do this, these worlds are distributed into several virtual machines (VMs) and the hypervisor manages the resources requested by these VMs.

Figure 2.7: Continental and SYSGO integrate vehicle control electronics for instrument clusters and infotainment [22].

The increasing number of ECUs in today's cars is an aspect that was taken in to count

[8]The GENIVI Alliance is a non-profit automotive industry alliance committed to driving the broad adoption of an In-Vehicle Infotainment (IVI) open-source development platform. The GENIVI Alliance was founded on March 2, 2009 by BMW Group, Delphi, GM, Intel, Magneti-Marelli, PSA Peugeot Citroen, Visteon, and Wind River Systems [21].

by this partnership. Regarding that, it is also wanted to display every different, but relevant information, of all the ECUs present in the car using a user-friendly interface as well as all the GENIVI and/or Android applications in different screens, such as in the instrument cluster or in the middle console display.

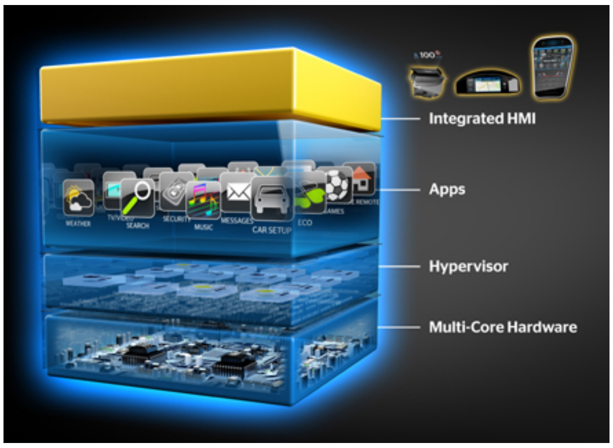


Figure 2.8: Integration via multi-core-hardware and Hypervisor [22].

To accomplish these, it is generated three VMs on a multi-core processor, which means that three SW technologies can run in parallel but completely independent for each other.

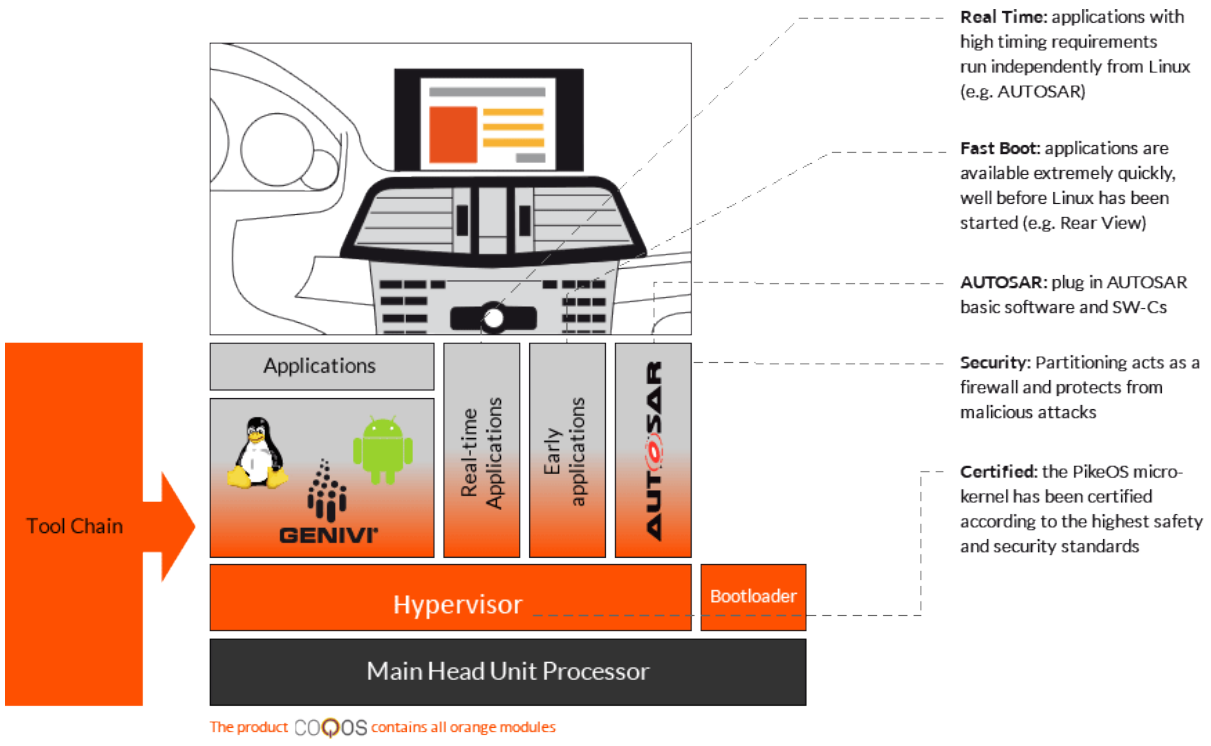
COQOS Operating System

This product was developed by OpenSynergy to provide user-friendly applications, internet and other infotainment funcionalities to the user without compromising the real-time/automotive requirements. The COQOS OS is composed by an hypervisor, which is based on the certified PikeOS micro-kernel, a Linux based OS to run the infotainment applications, AUTOSAR, which will run independently from Linux, and a bootloader.

According to COQOS's website, its benefits are:

- Possibility to run on compact microprocessors as well as high-performance multi-core processors;
- Assign different partitions to one core or several cores to one partition;
- Hypervisor certified with the highest requirements in terms of safety and security;
- Isolation between guest OSs;

- Integrating AUTOSAR seamlessly;
- Reduce time-to-market and low costs.



Real Time: applications with high timing requirements run independently from Linux (e.g. AUTOSAR)
Fast Boot: applications are available extremely quickly, well before Linux has been started (e.g. Rear View)
AUTOSAR: plug in AUTOSAR basic software and SW-Cs
Security: Partitioning acts as a firewall and protects from malicious attacks
Certified: the PikeOS micro-kernel has been certified according to the highest safety and security standards
Tool Chain
Applications
GENIVI
Real-time Applications
Early applications
AUTOSAR
Hypervisor
Bootloader
Main Head Unit Processor
The product COQOS contains all orange modules

Figure 2.9: System containing COQOS OS [23].

Evidence® open-source platform for automotive systems

This solution developed by Evidence combines Linux for human-machine interactions and a reliable certified RTOS based on OSEK developed by Erika Enterprise, specifically designed for automotive and industrial control, to run on the same multi-core processor. The hard real-time requirements are guaranteed by running control on a dedicated core, and the safety of the access to shared data is accomplished using mutual exclusion mechanisms. This solution was presented in Edinburgh at Automotive Linux Summit event, that occurred last October, where a demo running on a Freescale i.MX6 multi-core ARM SoC proved the feasibility and effectiveness of it.

A presentation of this solution can be found in [24].

fentISS XtratuM

XtratuM is a secure, reliable and efficient hypervisor created to enter the markets of aerospace, aeronautics, automotive and transportation and more, by Fent Innovative Software Solutions

S.L.. This hypervisor provides a framework to run several OSs in a partitioned environment and it can be used to build a multiple independent levels of security, always meeting safety critical real-time requirements. It supports four different processor architectures, such as ARM and x86, and its main features are:

- Para-virtualization technology;
- Designed to have a small footprint;
- Multi-plan support;
- Advanced monitoring and error reporting;
- Configuration and validation tool available;
- ARINC-653[9] flavour;
- Smart I/O multi/demultiplexing.

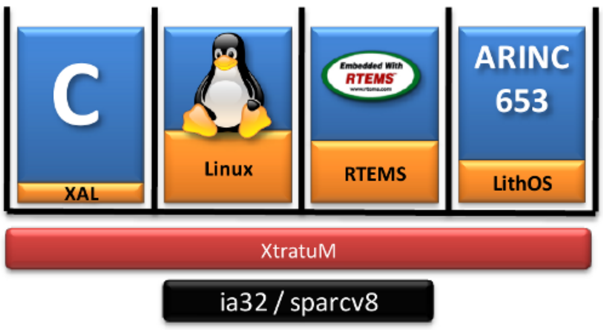


Figure 2.10: XtratuM partitions and execution environments [26].

Green Hills Software's INTEGRITY® Multivisor

INTEGRITY Multivisor was developed to consolidate disparate systems onto dedicated VMs running on the same platform. This means one HW platform can host arbitrary guest OS alongside a suite of real-time applications and they are flexibly scheduled across one or multiple cores. When available, INTEGRITY Multivisor will use the processor's hypervisor mode assistance to minimize modifications to guest OSs. Otherwise, intrusive modifications to the guest OS will

[9]ARINC 653 (Avionics Application Standard Software Interface) is a software specification for space and time partitioning in safety-critical avionics real-time OSs [25].

be the minimum as possible to maximize performance without sacrificing ease of migration and portability.

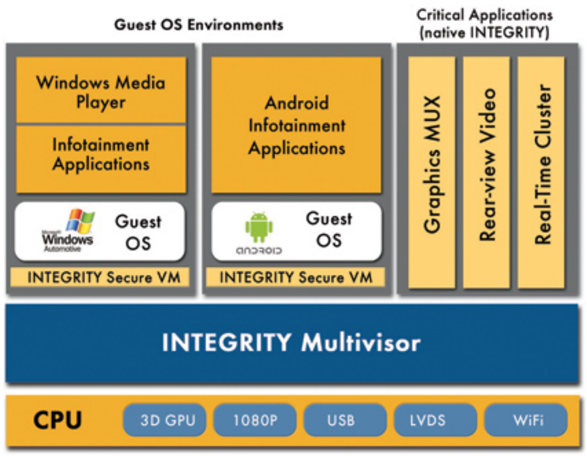


Figure 2.11: Virtualization architecture for Android infotainment systems [27].

The benefits of using INTEGRITY Multivisor as referred in Green Hills Software's website are, besides those in common with this kind of products:

- HW consolidation that decreases production costs;
- The ease of porting OSs to new HW gets faster time-to-market products;
- Better time-in-market by reusing legacy OSs and SW;
- Eliminating the need to port existing applications to new OSs;
- Hypervisor power management that turns this product a green one;
- EAL6[10] plus High Robustness-certified OS technology.

Mentor Embedded Hypervisor

This is a product developed by Mentor Graphics® and it consists on an hypervisor of type 1, with a small footprint and high performance, which uses the ARM TrustZone to control HW access and supports several OSs including the Linux distribution Yocto, Android, Mentor's Automotive

[10]The Evaluation Assurance Level (EAL1 through EAL7) of an IT product or system is a numerical grade assigned following the completion of a Common Criteria security evaluation, an international standard in effect since 1999. The increasing assurance levels reflect added assurance requirements that must be met to achieve Common Criteria certification [28].

Technology Platform, Nucleus RTOS and AUTOSAR. According to Mentor Graphics®, this hypervisor has the following key benefits for developers:

- Build secure and high-performance embedded systems at the highest privilege level in a system;
- Reduce test and debug times by consolidating multiple functions on a single multi-core platform;
- Take advantage of asymmetric multiprocessing (AMP) and symmetric multiprocessing (SMP), or a combination of both, to provide flexible scheduling algorithm that allows for a single Virtual Machine (VM) to run on a multiple cores, or a multiple VMs to run on a single core or multiple VMs to run on multiple cores;
- Partition memory and devices to prevent unauthorized access to sensitive resources;
- Support for ARM TrustZone secure system architecture that allows normal and secure applications to run within the same embedded system.

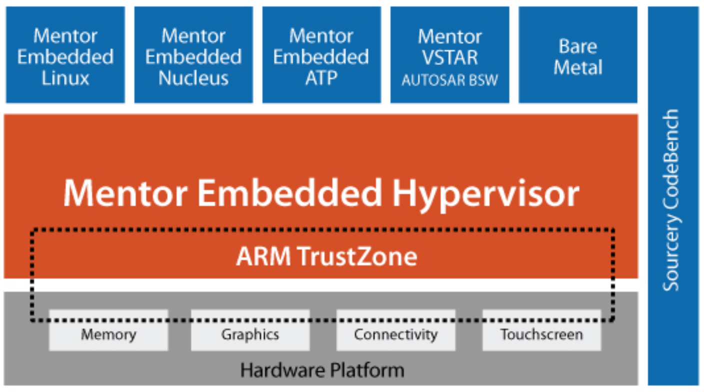


Figure 2.12: Mentor Embedded Hypervisor architecture [29].

A video of Mentor Embedded Hypervisor demonstrator can be seen in [30].

NI Real-Time Hypervisor

This is a National Instruments product and it has two different versions: one version that runs on a single multi-core PXI and other versions that runs on an industrial controller. The NI

Real-Time Hypervisor uses virtualization to run both LabVIEW Real-Time and either Windows XP or Red Hat Enterprise Linux OSSs simultaneously in parallel on a single processor.

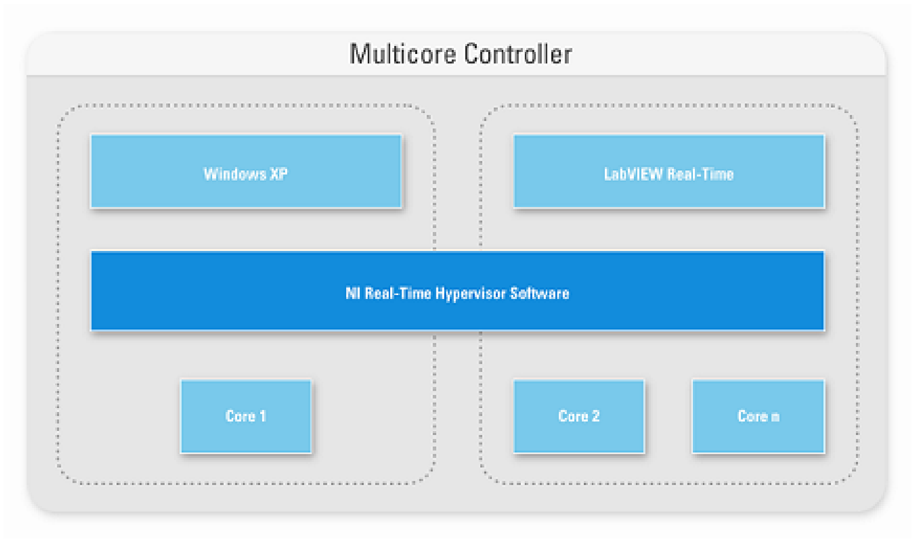


Figure 2.13: NI Real-Time Hypervisor for Windows SW runs Windows XP on one or more processor cores and LabVIEW Real-Time on the remaining cores [31].

The NI Real-Time Hypervisor is a bare-metal hypervisor which means that it is a virtualization SW package that runs immediately on the HW platform. Actually it can be divided into two SW packages: one where the host OS is Windows and other where the host OS is Linux. The NI Real-Time Hypervisor partitions I/O devices, RAM and CPU cores between OSs. Regarding that, when one of the OSs tries to access to shared resources or to the resources of the other OS, the hypervisor is automatically called to manage and control that access. This hypervisor also has a SW utility called NI Real-Time Hypervisor Manager that allows the user to assign individual I/O devices, RAM and CPU cores to either LabVIEW Real-Time or the host OS.

Red Bend's vLogix Mobile®

As the latter, this is also a type 1 hypervisor that runs directly on HW and schedules access to shared platform services by partitioning and allocating RAM and flash memory, for example, and separating VMs from one another. Virtual devices is a concept regarded in vLogix Mobile, and by this it is intended to say that devices, such as Wi-Fi, GPS and the screen, are virtualized so that the guest OSs can have access to a virtual view of them without being able to access them physically.

Some of the benefits announced by Red Ben of the vLogix Mobile are:

- Achieve faster time-to-market at lower costs;
- Porting capabilities;
- A optional module that provides absolute separation between VMs;
- A small footprint hypervisor (about 40000 lines of code);
- Pre-integrated with Android and available for multiple platforms for fast deployment.

This hypervisor was already integrated to a system for automotive purposes. Red Bend, Texas Instruments, QNX and Crack joined to create a demo (Figure 2.14) that illustrates a solution of virtualization to the automotive domain. This demo was composed by TI's Jacinto 6 board, QNX CAR Platform for infotainment, a QNX Neutrino RTOS based instrument cluster developed by Crank Software's Storyboard^{TM} Suite, and vLogix Mobile. This demo was presented in the International CES of 2014.

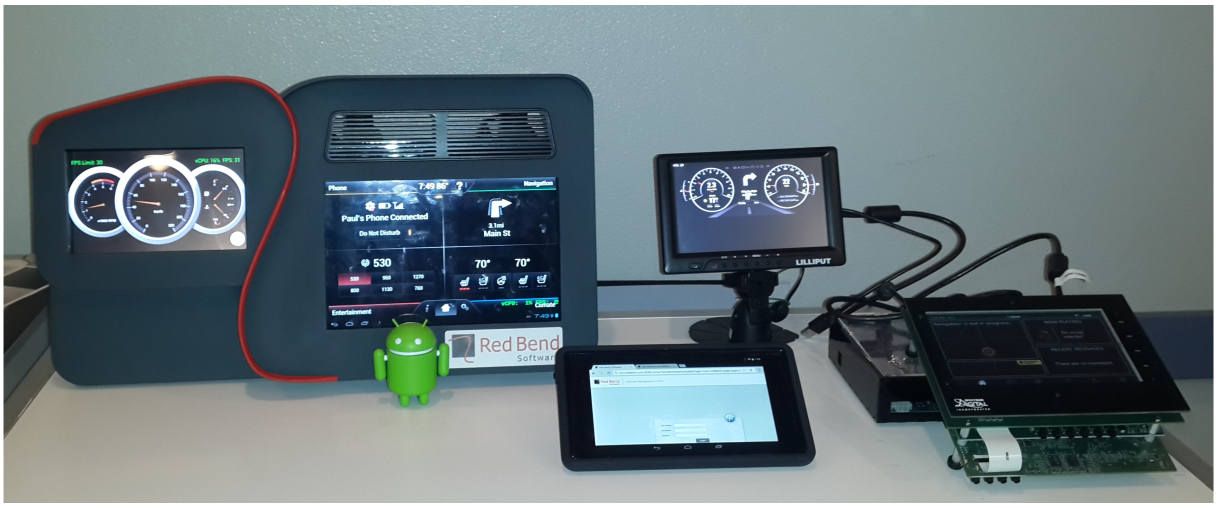


Figure 2.14: Demonstrator of the Red Bend, Texas Instruments, QNX and Crack joint [32].

SierraVisor Hypervisor

This Sierraware product is another hypervisor based on ARM architecture and, consequently, ARM TrustZone. It allows multiple OSs and RTOSs to run on a single multi-core processor in parallel using para-virtualization and full HW virtualization. Regarding the para-virtualization, it is inserted into the guest OSs non-intrusive hypercalls to minimize the modifications on them and the impact on performance, but also to allow the access to the secure OS for specific secure functionalities and data. The SierraVisor Hypervisor also provides both AMP and SMP for managing system utilization.

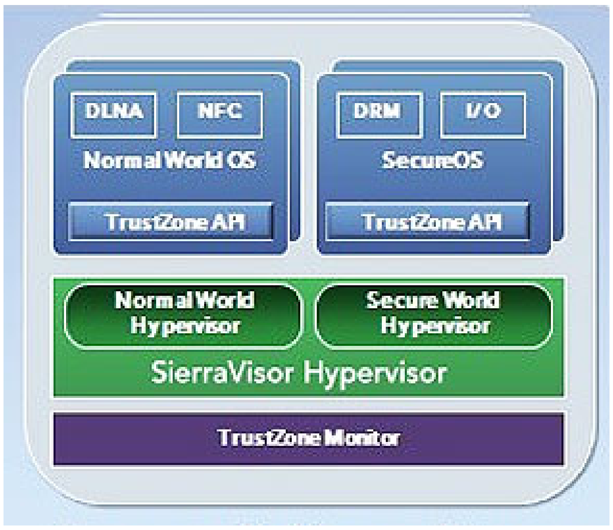


Figure 2.15: SierraVisor architecture diagram [33].

Besides the hypervisor, Sierraware also provides a utility that brings to equipment manufacturers the possibility to choose the right solution for their virtualization requirements and target architecture called SierraVisor Hypervisor Development Toolkit.

Wind River Hypervisor 2.0

The Wind River Hypervisor 2.0 is a type 1 hypervisor that was built for low latency, determinism and multi-core performance in the smallest footprint possible. It aims to consolidate multiple applications onto the same multi-core platform to increase capacity and reduce the size, weight, power and cost of embedded systems. It uses Wind River's VxWorks® to obtain a secure partition capability designed to separate applications of different levels of criticality in critical systems. Wind River Hypervisor supports six different processor architectures (Intel® Core^{TM}, Intel Xeon®, Intel Atom^{TM}, PowerPC e500 and e500mc, and ARM Cortex^{TM}-A9) and any OS that will be integrated with VxWorks and Wind River Linux.

The main features of this product according to Wind River are:

- Consolidate multi-criticality applications by, at the same time, complying with safety and security regulations;
- Meet real-time performance requirements even with non real-time applications running on

the same platform by using multiple OSs on multi-core processors supporting SMP and AMP;

- Migrate legacy SW with minimal requesting by isolating legacy applications into a single virtualized partition.

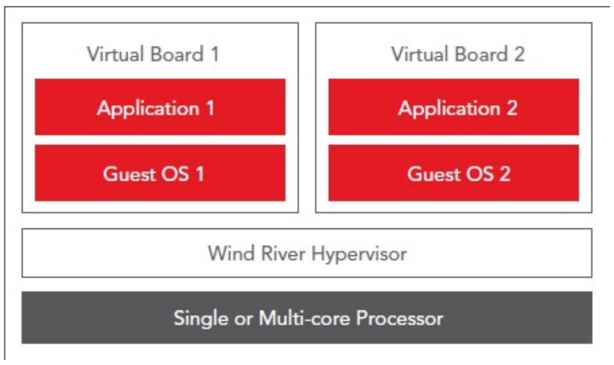


Figure 2.16: Virtualized system with Wind River Hypervisor 2.0 [34].

A video of Wind River Hypervisor 2.0 demonstrator can be seen in [35].

Summary

Comparing all the competitors of xLuna with xLuna itself, it can be said that xLuna and the existing solutions more related with the automotive domain are very similar. Almost all of them consist on an hypervisor of type 1 and introduce a multi-core processor solution, exactly as xLuna.

The use of ARM TrustZone technology is not so usual, but it seems to be the best approach to ensure that a non-critical system is sandboxed successfully, which means that the non-critical system does not impact the performance of the critical-real-time system. Considering this, only two competitors are at the same level of xLuna Automotive regarding the security of the whole system – Mentor Embedded Hypervisor and SierraVisor Hypervisor. Just note that the SierraVisor Hypervisor is not so focused in the automotive domain than Mentor Embedded Hypervisor and xLuna Automotive.

The fact of the xLuna's RTOS, the FreeRTOS, be a certifiable RTOS is also an advantage. There are competitors that also mention this feature, but not all of them.

In addition to this, xLuna Automotive will have implemented an eCall application, which, being eCall mandatory in all new cars commercialized in the European Union after October 2015, is a great advantage of xLuna when compared with its competitors.

The disadvantages that can be found when comparing xLuna with its competitors are mainly the lack of OS porting capabilities and the fact that xLuna Automotive does not support the AUTOSAR SW architecture. But in both cases, these are only disadvantages depending on the client, i.e., only if the client requires a different OS or support to AUTOSAR, the two disadvantages mentioned can be considered as that.

Concluding, xLuna Automotive has the potential to be on top of the market of the automotive multi-core automotive infotainment systems.

Chapter 3

Hardware, Software and Technologies Used

Make no little plans: they have no magic to stir men's blood... Make big plans: aim high in hope and work.
(Daniel Burnham, American architect)

In this chapter it will be mentioned the hardware, the software and technologies that had contributed significantly to the work developed. It will be detailed the ones which had specific and important features for this work.

3.1 AUTOSAR

With the increasing complexity of automotive applications, a group of automotive industry OEMs[1] created a joint group in order to develop and establish an open industry standard for automotive Electrics/Electronics (E/E) architecture. This joint group was created in 2012 and it is called AUTOSAR (AUTomotive Open System ARchitecture).

For more information related to AUTOSAR, please refer to section A.1.

[1]An Original Equipment Manufacturer (OEM) manufactures products or components that are purchased by another company and retailed under that purchasing company's brand name. More information can be found in [36].

3.2 CAN Protocol

CAN protocol is a vehicle bus protocol designed to allow different ECUs to communicate with each other without the control of an host computer and meeting the specific requirements of the automotive domain: real-time processing, required bandwidth, reliable operation in the EMI environment and cost-effectiveness.

For more information about CAN protocol, please refer to section A.2.

3.3 Hardware Used

In this section, it is presented the HW used in this work.

3.3.1 Freescale i.MX 6Quad

As mentioned in section 2.1, xLuna Automotive uses a multi-core SoC – the Freescale i.MX 6Quad processor [37].

For more information about i.MX 6Quad processors, please refer to section A.3.

3.3.2 ARM TrustZone dedicated hardware

Regarding the TrustZone technology, the ARM Cortex-A9 processors introduce a set of devices that brings support to what was described in section 2.2.3. These devices – TrustZone Address Space Controller (TZASC), Central Security Unit (CSU) and Secure Generic Interrupt Controller (GIC) – are aware of the access requesting world and so they can provide a set of specific functionalities that allows to create a safer and more secure container.

The TZASC device enables to configure the system memory up to 16 regions. Each region can be configured as read-only (RO), write-only (WO) or read-write (RW) to each world, in order to ensure secure memory access. In case of unauthorized access, either read or write, the transaction is denied, a secure IRQ is triggered and the interrupt handler, or interrupt service routine (ISR), is called to handle the situation.

The security enhanced GIC provides the capability to secure interrupts by blocking any configuration attempts from normal world SW, as well as the possibility to configure a secure interrupt to trigger a FIQ, instead of normal IRQ, that would jump to monitor mode.

The CSU controls the access to bus masters and slaves that are not aware of the currently running world, e.g., the CAN (Controller Area Network) bus interface device, the GPU or the USB. This controller sets what devices are authorized to be accessed by each world and, when an unauthorized access is detected, a secure interrupt is triggered calling the respective ISR to give back control to the secure world SW.

In addition to the latter devices, the TrustZone Watchdog triggers a secure interrupt if the non-secure SW monopolizes the core for a predetermined amount of time, forcing a switch to the secure world. This time interval, being configurable, allows the scheduler of the secure world OS to adjust the amount of computational time it will give to the normal world environment.

3.3.3 FlexCAN

The Flexible Controller Area Network (FlexCAN) module is a communication controller that implements the interface to the CAN protocol according to the CAN 2.0B protocol specification, which supports both standard and extended message frames. FlexCAN supports up to 64 Message Buffers (MBs) and each of them can be configured to receive (Rx) or to transmit (Tx).

For more information, please refer to section A.4.

3.3.4 IOMUX Controller

The IOMUX (I/O Muxing) Controller (IOMUXC) enables the SoC to share one pin's pad to several functional blocks by multiplexing the pad's I/O signals.

For more information about IOMUXC, please refer to section A.5.

3.3.5 Universal Asynchronous Receiver/Transmitter

The UART module is responsible to translate data between parallel and serial forms. It takes bytes of data and transmits them bit by bit sequentially. On the other side of the connection, a second UART re-assembles the bits into complete bytes. The *universal* term indicates that the developer can configure the bitrate and the data format which varies with the modes of operation of this module.

For more information about UART, please refer to section A.6.

3.3.6 Join Test Action Group and Open On-Chip Debugger

This section is dedicated to the HW and SW used for debugging purposes. Starting with JTAG (Join Test Action Group), which is implemented in a great part of the most modern processors, it is a standard for testing printed circuit boards and integrated circuits using boundary-scan, and electrical signals to communicate with the target. JTAG allows debuggers to communicate with a processor, letting developers to perform operations like single stepping and adding breakpoints. The debugger used in xLuna development was the well known Unix-like systems and C debugger, the GNU Debugger (GDB).

In order to be possible to use JTAG, it is needed an interface between the machine/PC which is debugging and the target. During xLuna development, it was used a Luminary Micro BD-ICDI-B, which is an In-Circuit Debug Interface (ICDI) board. This board has a Mini-USB port to connect to the PC and JTAG pins to connect to the target.

As GDB can not control the signalling of JTAG to the ICDI board, it was needed an extra SW to make the bridge between GDB and the drivers of the ICDI board. This bridge is done by Open On-Chip Debugger (OpenOCD). The OpenOCD interprets the GDB protocol and transforms it to JTAG signals. In order to run OpenOCD, two configurations files are needed – one for using with the ICDI board and another for the target processor – which are responsible to configure OpenOCD to the specific HW used. These configurations files are written in TCL/TK programming language.

For more information regarding JTAG and OpenOCD, please refer to [38] and [39], respectively.

Chapter 4

Development and Implementation

An expert is a man who has made all the mistakes which can be made in a very narrow field.
(Niels Bohr, Danish physicist)

In this chapter it will be explained all the work developed, as well as all the main decisions that were made, their consequences on the development part of the project and the difficulties that crossed in the way.

4.1 Preparation

The purpose of this section is to describe all the steps took to set-up all the tools and functionalities needed to provide a stable and consistent working base to the main work.

4.1.1 OpenOCD set-up for debugging via JTAG

The debug infrastructure for a project of this nature is crucial. The need of having the capability of debugging and accessing the registers in a very complex SW like this is huge, so that it is why this is the first step took in xLuna development.

Initially OpenOCD was installed and configured. To install it, it was needed to build it with a specific configuration to support the ICDI board (section 3.3.6). After that, OpenOCD needs two configuration files written in TCL/TK programming language – one for the interface/JTAG board and other for the target chip/board to debug. In the case of the ICDI board, it was needed to provide the ID of the board and initialize the values of FTDI registers. Regarding the configuration file for the target chip, the Debug Access Port (DAP) and the memory base addresses of the JTAG controller and of the target core to debug are used as parameters by OpenOCD functions that configure the SoC for JTAG debugging. These two configuration files are called every time the OpenOCD starts. The instructions followed can be found in [40], where it is explained how to install OpenOCD and configure it for ARM Cortex-M3.

When the project was near reaching the milestone, referred in section 2.1, a new need emerged to debug all cores simultaneously. To accomplish that, OpenOCD needed to be capable of doing that, and theoretically it was. It was spent exactly two weeks on trying to configure OpenOCD to debug all cores without the desired results. What was accomplished was the ability to read the value of two predetermined cores' registers but only one of them could be debugged with all the features provided by GDB. It was also discovered that the OpenOCD development team were still working on the multi-core debugging capabilities of OpenOCD for ARM based processors. Concluding, it was not possible to debug all cores with the full features of GDB at that time. This fact limited a lot the work being preformed.

4.1.2 UART set-up for providing a console

The debugging is not enough if the development team does not have an human-machine interface, like a console or a terminal. To provide one, there was only one possibility in this case, which was using an UART line.

Before starting the development work, it was needed to know how UART works and all information about its registers for this specific SoC. This kind of information is located in chapter 64 of i.MX6Q Reference Manual [41]. There, it could be found that UART has a Tx (Transmission) FIFO which holds the characters to print. These characters need to be write in a specific bit-field of UART Transmitter Register but only if Tx FIFO has free space. The others registers allow to configure UART as needed and check its working status.

Then, the development work started with a small function in ARM assembly that worked directly on the UART registers. It was resorted to Reference Manual to configure them but the function was not printing anything at first. After trying several different configurations, it was considered to use the configurations present in U-Boot[1] code but it was still not working. Then, it was realized that the problem was not in UART configurations. It was needed to configure the IOMUXC (Input/Output Multiplexing Controller) for a specific port to work in GPIO mode so that the CPU and the UART could be connected correctly. It was resorted to U-Boot source code again to discover how to configure the IOMUXC.

Later, with the integration of FreeRTOS, it was developed the device drivers for UART in C for future use in the project, but in this case it was followed the instructions and the information present in i.MX6Q Reference Manual.

4.2 The Real-time Operating System – FreeRTOS^{TM}

This section is reserved to the work done directly related with FreeRTOS. Most of the work done by all the development team of xLuna was on FreeRTOS domain. Thus, in this section most of the work is described.

4.2.1 Vector table and exceptions handlers set-up

The first task related with FreeRTOS that was needed to be done was to map the exception vector table. The exception vectors map all the ARM exceptions. In this case, they are the following:

- *Reset*;
- *Undefined instruction*;
- *Supervisor call (SVC)*;
- *Prefetch abort*;
- *Data abort*;
- *IRQ interrupt*;

[1]The U-Boot (Universal Boot loader) utility is a multi-platform, open-source, universal boot loader with comprehensive support for loading and managing boot images, such as the Linux kernel. For more information, please refer to [42].

- *FIQ interrupt.*

To accomplish this, it was necessary to implement a set of instructions, the vector table. Each instruction call the handler of the respective exception. This piece of code works like an `if..else` statement in C. When an exception occurs, the Program Counter (PC) go to 0x0 memory address by default and then the correct handler is called. Thus, in the initialization of xLuna, the vector table is copied to the initial memory address of the SABRE SD memory space. All the handlers used are provided by the FreeRTOS except the *reset* handler, which is the initialization code of all the system.

With the exception vector table mapped, the xLuna gained the capability of handling all kind of exceptions, including IRQs, which are essential to the correct functioning of FreeRTOS. But for the CPU to be able to receive IRQs it was needed to configure the GIC (section 3.3.2). Initially, it was used the assembly code from FreeRTOS to make GIC work but later in the project it was developed xLuna's own GIC device drivers in C. This work was based on chapter 3 of Cortex-A9 MPCore® Technical Reference Manual [43].

4.2.2 EPIT set-up for providing timer and tick capabilities to the OS

After xLuna had the ability to handle interrupts, it was necessary to implemented the tick[2] handler for FreeRTOS clock. To do that, it was needed to choose one timer from the two available in the SABRE SD – the General Purpose Timer (GPT) and the Enhanced Periodic Interrupt Timer (EPIT). The development team decided to go for EPIT because it is a timer that is capable of providing precise interrupts at regular intervals with minimal processor intervention, which is just perfect for a tick timer. So what it was needed to do was to develop the EPIT drivers and invoke the tick handler of FreeRTOS in the tick ISR, in order to provide the tick timer to the RTOS.

The drivers were developed following the code present in i.MX6Q SDK [45] and the guidelines present in chapter 24 of the i.MX6Q Reference Manual [41]. Then, it was developed a simple ISR that only printed a small debug message.

Initially the EPIT was working well but the IRQs were not getting caught or fired. After

[2]In computer science, a clock tick is the smallest unit of time recognized by a device. For personal computers, clock ticks generally refer to the main system clock, which runs at 66 MHz. This means that there are 66 million clock ticks (or cycles) per second [44].

analysing the value of the EPIT's registers, it could be concluded that the IRQs were not getting fired. So it was checked the configurations of EPIT and tried others configurations but the problem was on the wrong mapping of EPIT's registers structures in the code, i.e., the wanted values for the configurations were being written on different memory addresses than those that were supposed. After this task was accomplished with success, xLuna was ready to provide the tick timer to the RTOS and I changed the ISR to the FreeRTOS's tick handler. Then it was needed to test the work done by using the scheduler of FreeRTOS.

Initially it was created the `main()` function of FreeRTOS. In this initial phase, the `main()` just needed to have initialized the basic HW, create the OS's tasks desired and start the FreeRTOS's scheduler. Resorting to FreeRTOS's documentation [46], it was created two different tasks with different priority that would print different debug messages such that it is possible to distinguish them. To accomplish this task, it was needed to be sure that IRQs were enabled in secure world and disabled in normal world, which could be done using GIC (chapter 3 of the Cortex-A9 MPCore Technical Reference Manual [43]). After correcting some code mistakes, it was accomplished to have two RTOS's tasks running concurrently. This was a very important step for xLuna.

4.2.3 I2C (Inter-Integrated Circuit) and accelerometer set-up

This section is reserved for the development of I2C device drivers as well as the accelerometer drivers, which is an I2C peripheral. The main goal of using the accelerometer is to simulate a car accident by catching high acceleration values of the board in any direction. The accelerometer incorporated in SABRE SD board is a Freescale MMA8451Q, which is a smart low-power, three-axis capacitive micro-machined accelerometer with 14 bits of resolution.

From the i.MX6 Reference Manual (chapter 35) [41], it was known the need to configure the pads of the CPU I/O pins and the clocks for I2C device first. After consulting the SABRE SD's schematics [45], it was known what pins it was needed to use. Starting on IOMUXC, it was configured two pins in order to establish the connection between the CPU and the I2C device – one for bus clock and the other for bus data. Then, it was enabled the I2C clock that is controlled by Clock Controller Module (CCM) (chapter 18 of i.MX6 Reference Manual [41]).

In I2C device drivers it was implemented three public functions. One of them to initialize the device, which also configures IOMUXC and CCM like described above, another to read from a peripheral and the last one to write to a peripheral. The accelerometer drivers consisted in the initialization of the device and the data collection to get the acceleration values. As expected, all operations done directly with the device were made by using I2C functions to read and write. Both of these drivers' code were based on i.MX6 SDK.

This task lasted exactly two weeks. The troubles found were mainly related with making I2C bus working properly, because it was occurring an I2C arbitration lost error. It was taken a couple of days to understand the reason of I2C arbitration lost error. This error was occurring because, to make I2C bus work, all the peripherals present in the I2C bus must be powered on, which was not happening due the fact that IOMUXC was not configured properly. Some code mistakes, derived from the high complexity of the code used to access the registers of the devices, were also responsible for the long duration of this task.

After accomplish to get the acceleration values, it was created a FreeRTOS's task that was always getting the acceleration values and when high values were detected a LED blinked.

4.2.4 USB drivers and stack

As referred in section 2.1, for the milestone of xLuna, it has to be implemented an eCall application simulation, by sending a SMS to a pre-determined mobile phone number. So, a mobile communication modem was required. There were three options: (i) a 3G Mini-PCIe modem; (ii) a 3G plus WIFI Mini-PCIe modem; or (iii) a 3G USB modem. In this case, it was chosen the option (iii), a 3G USB modem. When compared to the others options, the option (iii) was the cheaper and the quicker to get because 3G USB modems are very common today. In terms of complexity of the work required, the option (iii) seemed the simplest one of the three options available. In fact, since the i.MX6Q SABRE SD supports to be the USB host, it was just necessary to develop the USB Stack, which is composed by, the USB Communication Device Class[3] module, the USB Host module and the device drivers for the USB 2.0 controller.

The i.MX6Q has four USB 2.0 controller cores, but only two are used in SABRE SD board – USB 2.0 Controller Core 0 and USB 2.0 Controller Core 1. The Controller Core 1, or Host1 Core, is only used by the CPU to communicate with the Mini-PCIe adapter. The Controller

[3]USB Communications Device class (CDC) is a composite Universal Serial Bus device class which is used primarily for modems, but also for ISDN and fax machines and telephony applications for performing regular voice calls [47].

Core 0, or OTG Core, has the only USB port available for external USB devices or hosts. Thus, the OTG Core supports working in Host mode or Device mode.

This task lasted about one week. The work related with it was mainly research on USB stack and reading the chapter 65 of the i.MX6Q Reference Manual [41] in order to understand the working manner of the USB 2.0 Controller of the SoC.

Furthermore, it was necessary to protect the USB Controller. So, the normal SW's attempts to access this device were blocked in CSU device and the Linux kernel was modified in order not to initialize and use the USB Controller.

4.2.5 3G USB modem set-up for sending a SMS

The main objective of this task was to discover the commands, or messages, that were needed to configure the 3G modem so it would be ready to send a SMS. The 3G modem used in this work was the Huawei K3765 [48].

The first step was configuring the 3G USB modem to work as a modem, and not as a mass storage device. This is done by sending a message to the USB device, containing a code specific for the modem's vendor. After some research work to find the code[4], it was sent to the USB device, switching it to modem mode with success.

The second and final step develop a function that sends the SMS. The sending process is also done through messages, but in this case by exchanging several messages with the 3G modem. These messages are Hayes commands[5] To start the sending process, it is sent to the modem the command `AT+CMGF=1` that configures the modem to work in Text Mode. In Text Mode, SMS messages are represented as readable text. Then, and after receive the response message from the modem, it is sent the command `AT+CMGS=XXXXXXXXXXX`, where 'XXXXXXXXXXX' is the mobile phone number, that sets the destination number for the SMS. And finally, after receiving the response of the modem, it is sent the text of the SMS.

[4]The code used to switch the mode of the Huawei K3765 can be found in some forums on the internet, e.g., the forum of Draisberghof's USB_ModeSwitch, which is a mode switching tool for controlling multi-mode USB devices on Linux [49].

[5]The Hayes command set, in computer telecommunication, consists of a series of short text strings which combine together to produce complete commands for operations such as dialing, hanging up, and changing the parameters of the connection. The vast majority of dialup modems use the Hayes command set in numerous variations [50].

This task was the last one before reaching the milestone of the xLuna.

4.2.6 FlexCAN module set-up

The FlexCAN device, as described in section 3.3.3, implements the interface to the CAN bus. Being this bus the most used to connect ECUs in cars, and because it is a bus with some complexity, its drivers can not just provide a simple API with the usual `open()`, `close()`, `read()` and `write()`. Usually, a CAN interface API has functions to set the properties of the bus, get errors from the bus, send, receive and configure messages, etc. So, it was made a research about the CAN bus and what functions were essential to an API with this purpose and it was decided to focus on the API provided by AUTOSAR Basic Software component dedicated for CAN bus, present in chapter 8 of AUTOSAR Software Specification for CAN Interface [51].

For the development of FlexCAN drivers, it was needed to configure the I/O pins of the processor, to establish the correct connection between the FlexCAN and the i.MX6Q SoC, and the clocks for the CAN bus and for the FlexCAN device as well.

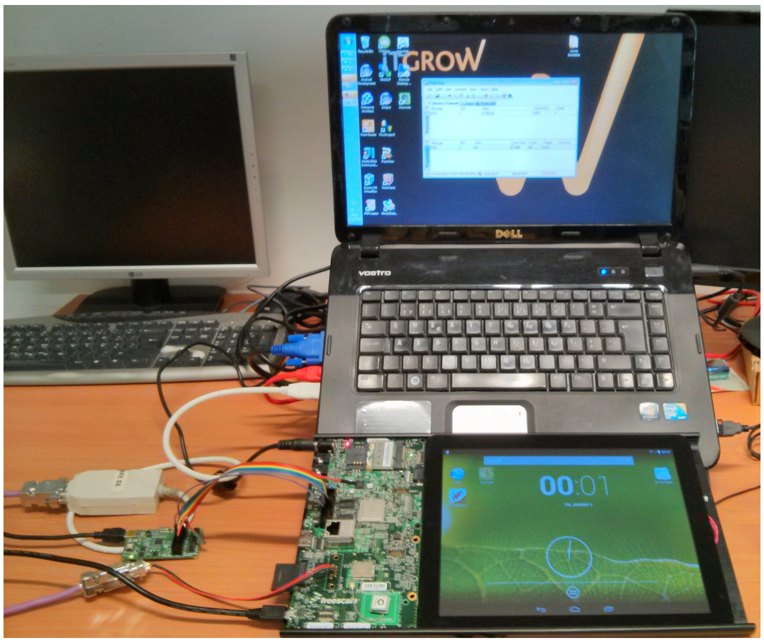


Figure 4.1: FlexCAN's tests and demonstrations work place.

The main difficulty of this task was related with the configuration of the I/O pins of the processor. Connecting the board to a computer using CAN bus, the PCAN-View SW and the PCAN-USB adapter (Figure 4.1), both from PEAK-System [52], it was not being possible to send or receive messages. The errors got could tell that the problem was physical. It could be either the I/O pins of the processor or incompatible configurations between the SW in the computer and the FlexCAN controller, like bitrate. It was followed the instructions present in chapter 26 of the i.MX6Q Reference Manual [41] and the schematics of the board [45], so the pins possibility was not an option. It was spent a couple of weeks trying to the get the right configurations on both sides of the CAN bus, but without success. After another review of the schematics of the board, it was found a signal, the *CAN1_STBY* signal, that is not present in Reference Manual but it is actually essential of proper functioning of the FlexCAN device. So, it was necessary to configure the IOMUXC in order to pad this signal to the SoC, and finally, it was accomplished to exchange CAN messages between two nodes, the computer and the SABRE SD board.

4.2.7 Develop a ring buffer for UART

As referred before (section 3.3.5), the UART is the device responsible to provide a console for both FreeRTOS and Android. But it has a limitation – the Tx FIFO size. It is so because, for example, when a big flow of characters comes from Android via monicall and UART's Tx FIFO gets full, the FreeRTOS needs to wait if it needs to print something. This delays the HRT tasks, turning xLuna system in a non-safe one.

In order to avoid this latency to the system, it was decided to create a ring buffer[6] and use it in the monicall dedicated to print a character. So, when a monicall to print a character is called, the character is written into the ring buffer. Furthermore, this methodology requests the existence of a RTOS's task to flush the buffer periodically whenever the Tx FIFO is free.

The implementation of the ring buffer creates a latency in the arriving of the console messages. But, in the other hand, it makes the system more reliable by ensuring that both OSs, but specially the RTOS, do not have to wait for a long time to write to the UART device. The disadvantage of this idea is the possibility of losing characters, which happens when the buffer is full and a timeout value is reached. But as this is just used for debug messages, it is not really important for the well functioning of the xLuna system.

[6]A ring buffer, circular buffer or cyclic buffer is a data structure that uses a single, fixes-size buffer as if it was connected end-to-end [53].

4.2.8 TZASC module set-up

The TZASC device, as described in section 3.3.2, is very important to ensure security to the system by providing protection to system memory against unauthorized accesses. In order to make it work properly, it is needed to configure it first, by defining the memory regions wanted and their permissions, and then enable the device.

It was created three memory regions. These memory regions are described below:

- Region 1 – A region with 512 MB and configured to allow RW accesses from both normal and secure SW;
- Region 2 – A region with 256 MB and configured to allow RW accesses from both normal and secure SW;
- Region 3 – A region with 256 MB and configured to allow RW accesses from secure SW.

Thus, the regions 1 and 2 make a 768 MB super-region, because TZASC does not allow to create one region with the size of 768 MB. This super-region is dedicated to Android but it is also accessible by FreeRTOS and xLuna SW (Figure 4.2). The region 3 is dedicated to FreeRTOS, while it can also be accessed by xLuna SW but not by Android.

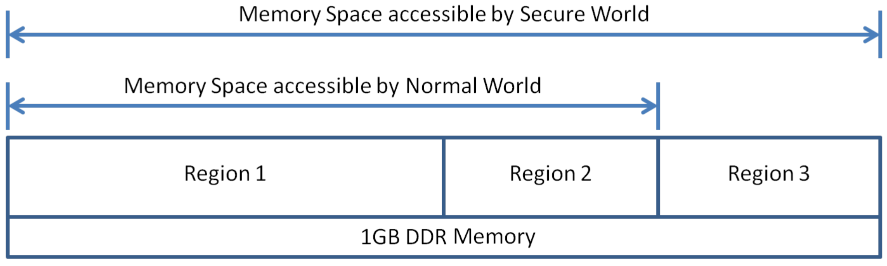

Memory Space accessible by Secure World
Memory Space accessible by Normal World
Region 1
Region 2
Region 3
1GB DDR Memory

Figure 4.2: DDR memory splitting diagram.

As TZASC is disabled, or bypassed as referred in i.MX6Q Reference Manual [41], by default, the signals between CPU and the system memory do not pass through it. In this condition, it is referred in TrustZone terminology that TZASC is bypassed. There are two different ways to enable it. The recommended one is to burn a fuse which ensures that TZASC is bypassed. Burning this fuse, the TZASC device become permanently enabled, which means that it can not be undone. That is why the development team of xLuna decided to go for the other possible way to enable TZASC.

The other possible way to enable TZASC is to configure IOMUXC (section 3.3.4), while ensuring that there is no DDR memory transaction occurring. This instructions are present in section 63.3 of the i.MX6Q Reference Manual. To do this, it was developed a set of instructions in ARM assembly that enables TZASC in IOMUXC registers. Then, these instructions were copied to the On-Chip RAM (OCRAM), a small internal RAM of the SoC, in order to ensure that there is no transaction being made between the CPU and the DDR memory.

The results were not as expected. After the enabling of TZASC, every writes made in any memory region, in any memory address, were not done with success. The first evidence spotted was right after the enabling of TZASC when a `push` instruction was made but the values wrote in the stack were not there after the `push` instruction. Logically, when the next `pop` instruction appeared, a data abort occurred and the system went down. The development team spent about a month trying to get the right configurations and the right enabling process of the TZASC but without success. It was tried to resort to the Freescale's support team but they could not help us.

More information about TZASC can be found in chapter 63 of the i.MX6Q Reference Manual [41] and in TrustZone Address Space Controller Reference Manual [54].

MMU as an alternative to TZASC

A document search was made in order to find possible solutions to the problem. In that search, it was considered the possibility of the Memory Management Unit[7] (MMU) blocks the unauthorized accesses to the system memory. This idea came up because the MMU enhances the TrustZone technology, which allows each core to have two different translation tables – one for the secure world and the other for normal world.

After one week researching on this subject, it was discovered that the TrustZone features in MMU could not substitute TZASC. It only blocks a normal SW's attempt to create a translation table of a range of memory addresses that overrides a secure range of memory addresses. For example, it is created three translation tables in secure world – one for ROM, internal RAM and other peripherals, another for the secure world and the last one for the normal world. After this, if a normal SW tries to create a translation table for the normal world with at least one memory

[7]In ARMv7a, a Memory Management Unit controls address translation, access permissions, and memory attribute determination and checking, for memory accesses made by the processor [55].

address in common with the ones of the translation table of the secure world, that operation will be blocked by MMU. This means that, if a normal SW tries to access directly to a secure memory address, MMU will not block it, otherwise to the TZASC. So, MMU is not an alternative to TZASC.

This research was based mainly on chapter B3 of the ARM Architecture Reference Manual for ARMv7-A and ARMv7-R architectures [55].

4.2.9 Code conventions and comments

After all the work done, it was necessary to be sure that the code produced was according to the CSW's Software Coding Standard for Safety Critical Systems. In order to do that, all the modules were reviewed and modified if needed, which demanded to study all the code produced for xLuna Automotive (more than 30.000 code lines). The code was also commented so that future members of the development team of xLuna can easily understand the code. Thus, the comments produced were compatible with a tool for generating documentation from programming code – the Doxygen [56].

4.3 The Non Real-time Operating System – Android

This section is reserved to the work related with Android. The Android version used in this work was the Jelly Bean 4.2.2 [57] patched with a Board Support Package (BSP), which was the only Android's BSP for i.MX6Q available when xLuna Automotive project started. Now it is available the Android Jelly Bean 4.3 and it can be found in [45].

4.3.1 Modify Linux kernel source code to print via monicalls

Both in xLuna Space and xLuna Automotive, it is interesting to have a console of the non-critical OS. In the case of xLuna Automotive, the Android OS uses the console as an output for logging the status of the OS, mainly at its initialization. In fact, the information printed while Android is initializing was crucial to this work. For example, it shows the status of the devices' initialization done by Android. As it was desired that Android would not have access to some HW, through the console it was possible to check what devices were having trouble to be initialized or fundamental for Android to run.

In order to have a console, the Android OS uses the UART device. So, UART was now being used by both normal and secure worlds, which could impact the critical-real-time SW performance significantly, as referred in section 4.2.7.

This problem could be solved using xLuna's monicalls. With the monicalls, FreeRTOS still has full access to the UART device but not Android. Instead of that, Android uses a monicall to print character-by-character. In turn, the xLuna puts the character received from Android into the ring buffer (section 4.2.7).

So, since the Android OS uses the Linux kernel (in xLuna, it was used the version 3.0.35 of the kernel [58]), it was necessary to modify this kernel in order to use the monicalls in printing functions. The files that needed to be modified were the UART device drivers' files. In Linux kernel, the i.MX6Q's UART device has two device drivers – one used before the serial driver has initialized and the other used after that. Both device drivers were modified in just one function in each one – the function responsible to print a character in the console. Since this function is the only one that writes directly on UART's Tx FIFO, its code was replaced by the calling of the monicall dedicated to print a character.

The main difficulty in this task was finding the UART's device drivers used in Linux kernel. In this case, these device drivers are located in the following kernel's files:

- `(Linux kernel source code directory)/drivers/tty/serial/mxc_uart_early.c`;
- `(Linux kernel source code directory)/drivers/tty/serial/imx.c`.

Chapter 5

Results

People with clear, written goals, accomplish far more in a shorter period of time than people without them could ever imagine.
(Brian Tracy, Author)

In this chapter, it is presented some results showing that the system developed is performing as expected. Two demonstration experiments were developed and are presented here.

5.1 xLuna Automotive's Prototype Demonstrations

This section is dedicated to describe the prototype implemented with the purpose to simulate a real-life application of xLuna Automotive. In this prototype, it is used a radio-controlled (RC) car toy with the SABRE SD platform on it (Figure 5.1).

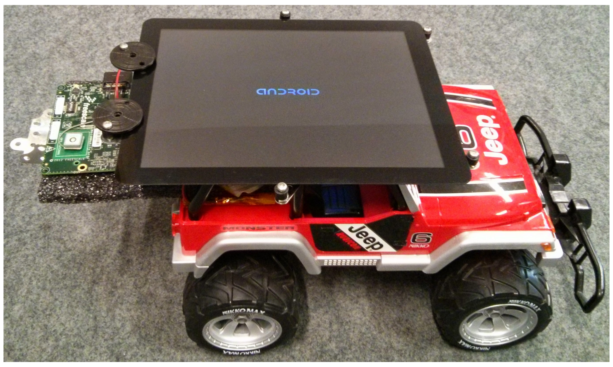


Figure 5.1: Prototype with SABRE SD platform on the RC car.

The SABRE SD platform's 10.1" touch-screen display is placed on the top of the RC car, while the SABRE SD board, its battery and the 3G modem are located below the display, in the rear part of the RC car (Figure 5.2). Both the display and the SABRE SD board were fixed using Meccano[1] pieces and the RC car's structure itself.

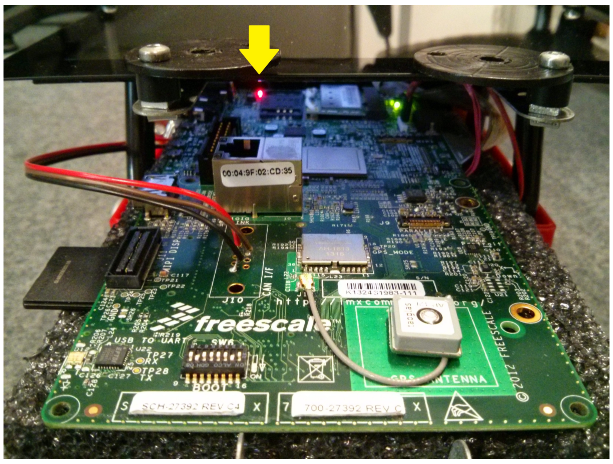
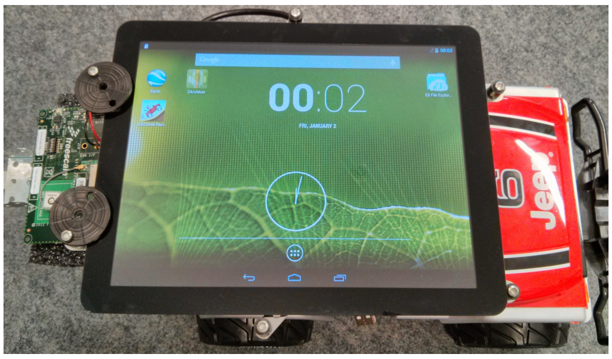
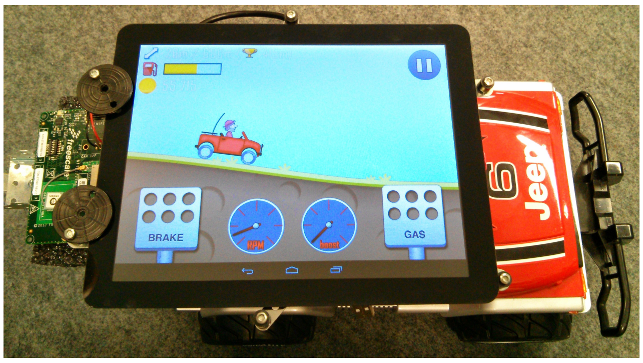


Figure 5.2: SABRE SD board on the RC car.

In Figure 5.3a, it can be observed Android running normally, i.e., it is possible to navigate through its application drawer, opening and using applications without restrictions. For instance, it can be played a 3D graphics game with a fair performance (Figure 5.3b).



(a) Android OS home-screen.



(b) 3D graphics game running on Android.

Figure 5.3: Android OS running on xLuna Automotive prototype.

In order to demonstrate that FreeRTOS is *alive* and running simultaneously alongside Android, a LED blinks with a period of one second as the only output dedicated to the RTOS, indicated by the yellow arrow in Figure 5.2.

[1]Meccano is a model construction system invented in the UK by Frank Hornby. It consists of re-usable metal strips, plates, angle girders, wheels, axles and gears, with nuts and bolts to connect the pieces [59].

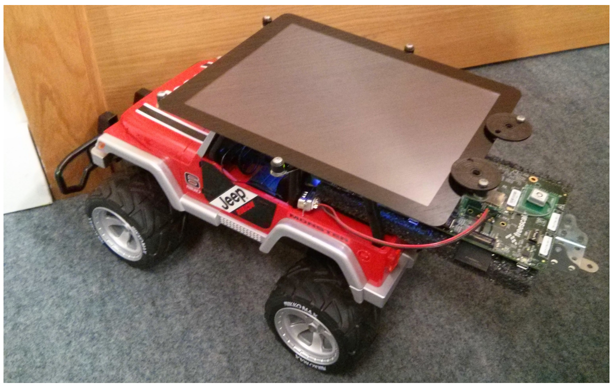
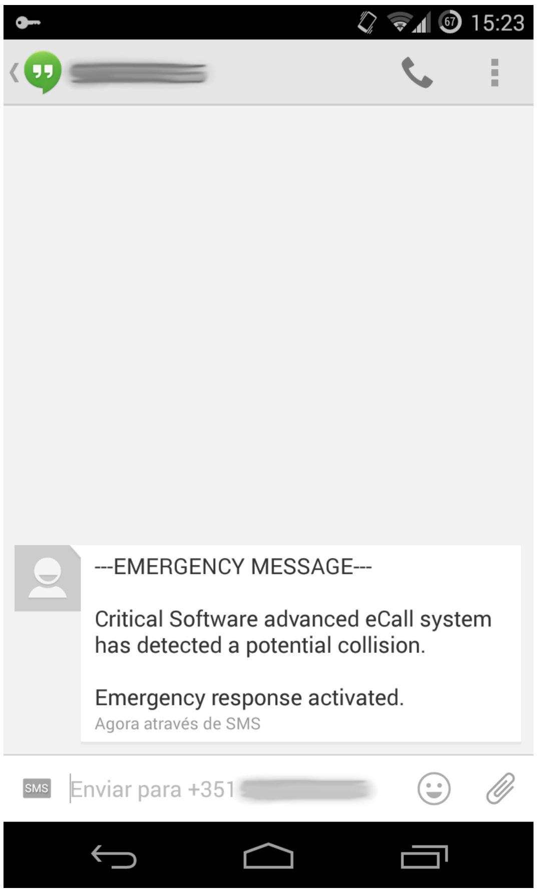


Figure 5.4: Car accident scene simulation with Android already halted.

Regarding the eCall system simulation, a car accident is simulated by driving the RC car against a wall (Figure 5.4). The objective is to create high accelerations values that causes the Android to halt and send a SMS to a pre-determined mobile phone number (Figure 5.5).



---EMERGENCY MESSAGE---

Critical Software advanced eCall system has detected a potential collision.

Emergency response activated.

Figure 5.5: SMS sent by the xLuna when a car accident is detected.

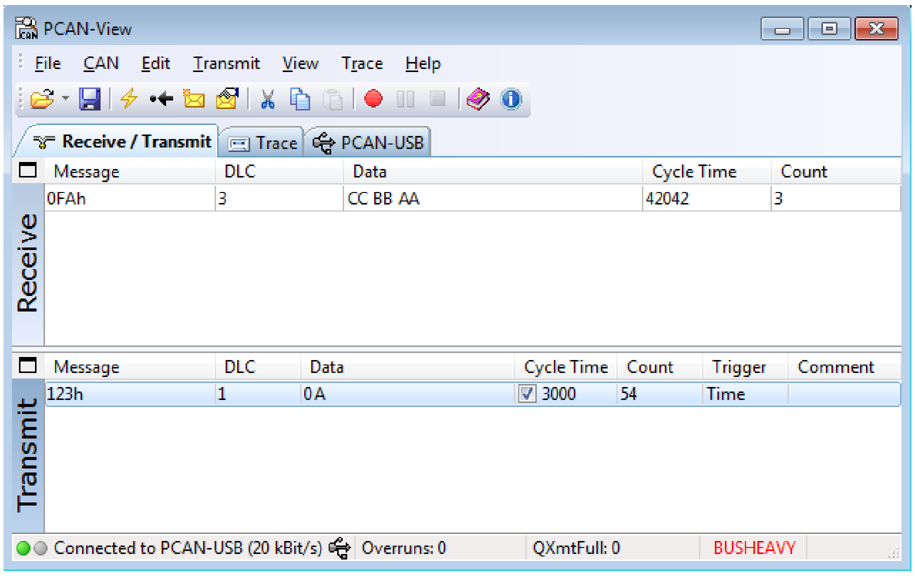
5.2 FlexCAN Demonstration

The demonstration of the implemented FlexCAN device drivers can not be done using the prototype described in section 5.1, because it is needed at least two CAN nodes, and there is only one within the prototype – the FlexCAN module itself. So, it was necessary to connect the SABRE SD board to a computer, as described in section 4.2.6. The procedure used in xLuna to test the FlexCAN is the following:

- In xLuna, create a FlexCAN demonstration application, using a RTOS's task, that sends a message and stays listening for receiving another message;
- Connect the SABRE SD to the computer;
- Open the PCAN-View SW in the computer;
- Configure the PCAN-USB adapter according with the configurations of the FlexCAN module;
- Set the PCAN-USB's node ID;
- Create a Tx message with the same node ID of the Rx message created in xLuna, configuring it to be sent periodically;
- Start xLuna.

In Figure 5.6, it can be visualized the created Tx message in PCAN-View SW. This message's data is `0x0A` and the destination node ID is `0x123`. This values were chosen just for testing.

In received section of PCAN-View, it is already the message transmitted by FlexCAN. It can be visualized its destination node ID, i.e., the PCAN-USB ID, which is `0xFA`, and the data transmitted, which is `0xAABBCC`. Please note that in a CAN bus message exchange, the data is transmitted only one byte at the time. That is why the data of the message received is presented backwards.

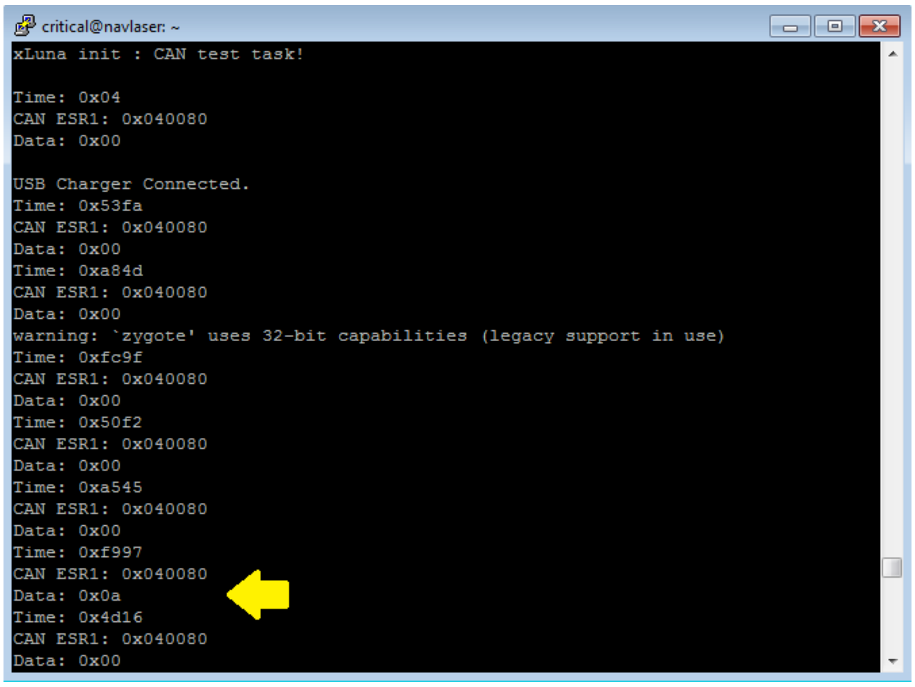


Message	DLC	Data	Cycle Time	Count
0FAh	3	CC BB AA	42042	3

Message	DLC	Data	Cycle Time	Count	Trigger	Comment
123h	1	0A	☑ 3000	54	Time	

Connected to PCAN-USB (20 kBit/s) | Overruns: 0 | QXmtFull: 0 | BUSHEAVY

Figure 5.6: PCAN-View trying to transmit a message and with a message received.



critical@navlaser: ~

```
xLuna init : CAN test task!

Time: 0x04
CAN ESR1: 0x040080
Data: 0x00

USB Charger Connected.
Time: 0x53fa
CAN ESR1: 0x040080
Data: 0x00
Time: 0xa84d
CAN ESR1: 0x040080
Data: 0x00
warning: `zygote' uses 32-bit capabilities (legacy support in use)
Time: 0xfc9f
CAN ESR1: 0x040080
Data: 0x00
Time: 0x50f2
CAN ESR1: 0x040080
Data: 0x00
Time: 0xa545
CAN ESR1: 0x040080
Data: 0x00
Time: 0xf997
CAN ESR1: 0x040080
Data: 0x0a
Time: 0x4d16
CAN ESR1: 0x040080
Data: 0x00
```

Figure 5.7: Console of xLuna.

In xLuna's side, it is printed periodically the bus time, the value of the FlexCAN's Error and Status 1 Register (ESR1) and the data presented in the Rx MB. Both the bus time and the value of ESR1 are just printed for debugging purposes. In Figure 5.7, it can be visualized these prints, as well as the value `0x0a` in the second data print, indicated by the yellow arrow. This means that the message transmitted by the CAN node PCAN-USB was received by the CAN node FlexCAN.

Chapter 6

Conclusions and Future Work

An expert is a man who has made all the mistakes which can be made in a very narrow field.
(Niels Bohr, Danish physicist)

6.1 Conclusions

xLuna Automotive architecture

The architecture that was selected showed that it is a better choice when compared to the architecture used in xLuna Space. In addition, the use of a multi-core processor also helps the xLuna Automotive in that comparison, since xLuna Space uses a single-core processor based on SPARC architecture.

The usage of TrustZone technology allows to develop an hardware enforced container for separation of software environments with a high-reliably protection against both software and physical attacks. xLuna brings TrustZone to deliver a faster mixed-criticality certifiable solution, enabling feature rich operating systems to coexist with a secure safety-critical real-time operating system, with a small secure para-virtualization layer.

With the monitor call system, new calls can be easily added to expand the underlying secure real-time operating system features to the non-secure environment. For example, it allows access for requesting information through the CAN bus by a secure API to a graphical user interface in the non-secure environment having only to certify what is below that API. This guarantees that there is no impact in the vehicle CAN network, even if the graphical user interface fails for some reason.

The developed system successfully allows to reduce the number of ECUs present in a car that has xLuna installed. Also, it ensures that, as referred above, the real-time operating system accesses the devices needed by the eCall application without any restrictions of time and permissions.

The implementation of the eCall application in xLuna is a relevant novelty. Without being necessary to add new redundant hardware (GPS, accelerometer, communication system, and a processor) to support eCall, a car with the xLuna system will have a feature rich operating system for infotainment, the Android, and, at the same time, the newest safety technology for cars in the European Union.

Moreover, the technologies used were new to almost all elements of the development team, making this an even harder work.

Summary of the main results

The results obtained were the expected. It was accomplished to have a demonstrator with Android running (on one core) simultaneously alongside a real-time operating system (on another core) on the same hardware platform. In addition, it was also implemented on this demonstrator an eCall application simulation, which sent a SMS whenever high acceleration values were detected by an on-board accelerometer.

It was accomplished to have the controller responsible to implement an interface to the CAN protocol working properly. As described in chapter 5, the xLuna Automotive is capable of transmitting and receiving CAN frames using the resources of the real-time operating system, while there is no impact on Android execution.

The objectives of this internship have been achieved with great success. In addition, it was used and consolidated the knowledge acquired mainly in *Operating Systems*, *Real-Time Systems*, *Architecture of Computers* and *Micro-processors Systems* courses.

6.2 Future Work

In the future, xLuna Automotive will have new versions. Considering its architecture, Android will run on three cores, while the real-time operating system will run on only one core. It will be also considered an architecture where the real-time operating system's core will be shared with Android, because the real-time operating system may not need so much computing power.

The TrustZone Address Space Controller will also be a subject for the future work. This device can have a very important role in terms of security within xLuna's software, ensuring that no non-authorized accesses to the memory are done.

The takeover of the devices made when an eCall event happens will be implemented in the future.

It is suggested to be implemented an Android application that would send a warning text message (SMS), when a car accident happens, to a set of contacts selected by the user of xLuna Automotive. This application would enable the user to choose what information would be sent in the warning message, e.g., the geographic coordinates of the accident scene, etc., and what contacts would receive the warning message. This could be a greater value feature for xLuna Automotive.

As a *beta* version of the Android application described above, the possibility to send a voice message instead of the SMS is very interesting in order to simulate the eCall system.

An Android application that would freeze the Android OS is also suggested to be implemented in the future, for demonstrations purposes. The objective of Android's freezing is to demonstrate that this freezing does not impact the whole system and that the real-time operating system still have the situation under control.

References

[1] Hypervisor, from Wikipedia, the free encyclopedia, `http://en.wikipedia.org/wiki/Hypervisor`, last accessed on July 14, 2014. 1

[2] Android, `http://www.android.com/`, last accessed on July 14, 2014. 1

[3] FreeRTOS: a Certifiable RTOS, `http://www.freertos.org/`, last accessed on July 14, 2014. 1

[4] ARM, The Architecture for the Digital World, `http://www.arm.com/`, last accessed on July 14, 2014. 3

[5] xLuna: a Real-Time, Dependable Kernel for Embedded Systems, `http://www.criticalsoftware.com/`, last accessed on July 14, 2014. 2.1

[6] ARM Cortex-A Series Applications Processors, `http://www.arm.com/products/processors/cortex-a`, last accessed on July 14, 2014. 2.1

[7] European Parliament: Parliament supports life-saving eCall system in cars, `http://www.europarl.europa.eu/news/en/news-room/content/20140224IPR36860/html/Parliament-supports-life-saving-eCall-system-in-cars`, last accessed on July 14, 2014. 2.1

[8] RTEMS Real Time Operating System (RTOS), `http://www.rtems.org/`, last accessed on July 14, 2014. 2

[9] LEON2 Processor, `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.gaisler.com/products/leon2/leon.html`, last accessed on July 14, 2014. 3

[10] QNX Neutrino RTOS, `http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html`, last accessed on July 14, 2014. 2.2.2

[11] QNX Car Dominance, http://berryflow.com/BBI/qnx-car-dominance/, last accessed on July 14, 2014. 2.2.2

[12] POSIX, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/POSIX, last accessed on July 14, 2014. 4

[13] ARM TrustZone: System Security by ARM, http://www.arm.com/trustzone, last accessed on July 14, 2014. 2.2.3, 2.4

[14] Sandbox (computer security), from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Sandbox_(computer_security), last accessed on July 14, 2014. 6

[15] European Commission, eCall: Time saved = lives saved, http://ec.europa.eu/digital-agenda/en/ecall-time-saved-lives-saved, last accessed on July 14, 2014. 2.6, 2.3

[16] About eCall, HeERO Project site, http://www.heero-pilot.eu/view/en/ecall.html, last accessed on July 14, 2014. 2.3

[17] HeERO: Harmonised eCall European Pilot, http://www.heero-pilot.eu/view/en/heero.html, last accessed on July 14, 2014. 2.3

[18] *Portugal adere ao sistema de alerta e-Call*, the Portuguese Government's *Ministério da Administração Interna* site, http://opiniao.mai-gov.info/2007/09/18/portugal-adere-ao-sistema-de-alerta-e-call/, last accessed on July 14, 2014. 2.3

[19] Model-driven engineering, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Model-driven_engineering, last accessed on July 14, 2014. 7

[20] The CONCERTO Project, partially funded by the ARTEMIS Joint Undertaking, http://www.concerto-project.org/, last accessed on July 14, 2014. 2.4.1

[21] GENIVI Alliance, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/GENIVI, last accessed on July 14, 2014. 8

[22] Higher System Integration in the Vehicle Cockpit, Continental site, https://www.conti-online.com/www/pressportal_com_en/themes/press_releases/3_automotive_group/interior/press_releases/pr_2014_02_25_domain_integration_en.html, last accessed on July 14, 2014. 2.7, 2.8

[23] COQOS Operating System, OPENSYNERGY site, http://www.opensynergy.com/produkte/coqos/, last accessed on July 14, 2014. 2.9

[24] A fully Open-Source platform for automotive systems: Embedded Linux + OSEK/VDX on a dual core CPU, https://www.youtube.com/watch?v=x_5bOkzU3H8, last accessed on July 14, 2014. 2.4.2

[25] ARINC 653, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/ARINC_653, last accessed on July 14, 2014. 9

[26] XtratuM, fentISS site, http://www.fentiss.com/en/products/xtratum.html, last accessed on July 14, 2014. 2.10

[27] Green Hills Software Expands INTEGRITY RTOS Support to Renesas' 2nd-Generation R-Car Product Family, Green Hills Software site, http://www.ghs.com/news/20130927_renesas_rcar.html, last accessed on July 14, 2014. 2.11

[28] Evaluation Assurance Level, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Evaluation_Assurance_Level, last accessed on July 14, 2014. 10

[29] Mentor Embedded Hypervisor site, http://www.mentor.com/embedded-software/hypervisor/, last accessed on July 14, 2014. 2.12

[30] Mentor Graphics ARM Embedded Software demos, https://www.youtube.com/watch?v=Hox4S-yROgo#t=63, last accessed on July 14, 2014. 2.4.2

[31] NI Real-Time Hypervisor Architecture and Performance Details, National Instruments site, http://www.ni.com/white-paper/9629/en/, last accessed on July 14, 2014. 2.13

[32] See How Four Industry Leaders are Making the Connected Car Less Expensive for Manufacturers and Safer for Drivers, Red Bend Software site, http://goo.gl/4irVZI, last accessed on July 14, 2014. 2.14

[33] SierraVisor Hypervisor Development Toolkit, Sierraware site, http://www.sierraware.com/arm_hypervisor.html, last accessed on July 14, 2014. 2.15

[34] Wind River Hypervisor site, http://www.windriver.com/products/hypervisor/, last accessed on July 14, 2014. 2.16

[35] Wind River Hypervisor for Automotive Security Demo at Embedded World 2012, https://www.youtube.com/watch?v=dOYkdLGWf6k, last accessed on July 14, 2014. 2.4.2

[36] Original equipment manufacturer, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Original_equipment_manufacturer, last accessed on July 14, 2014. 1

[37] i.MX6Q: i.MX 6Quad Processors – Quad Core, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q, last accessed on July 14, 2014. 3.3.1, A.3

[38] JTAG Technologies, http://www.jtag.com/, last accessed on July 14, 2014. 3.3.6

[39] OpenOCD - The Open On-Chip Debugger, http://sourceforge.net/projects/openocd/, last accessed on July 14, 2014. 3.3.6

[40] STM32/ARM Cortex-M3 HOWTO: Development under Ubuntu (Debian), http://www.fun-tech.se/stm32/OpenOCD/index.php, last accessed on July 14, 2014. 4.1.1

[41] i.MX6Q: i.MX 6Quad Processors Documentation, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q&fpsp=1&tab=Documentation_Tab, last accessed on July 14, 2014. 4.1.2, 4.2.2, 4.2.3, 4.2.4, 4.2.6, 4.2.8, A.3, A.5, A.4, A.5, A.6

[42] Das U-Boot – the Universal Boot Loader, http://www.denx.de/wiki/U-Boot/WebHome, last accessed on July 14, 2014. 1

[43] Cortex-A9 MPCore Technical Reference Manual, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0407i/index.html, last accessed on July 14, 2014. 4.2.1, 4.2.2

[44] Clock tick, from Webopedia, http://www.webopedia.com/TERM/C/clock_tick.html, last accessed on July 14, 2014. 2

[45] Software & Tools for i.MX6Q SABRE Platform for Smart Devices, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=RDIMX6SABREPLAT&fpsp=1&tab=Design_Tools_Tab, last accessed on July 14, 2014. 4.2.2, 4.2.3, 4.2.6, 4.3, A.5

[46] FreeRTOS API Reference, http://www.freertos.org/a00106.html, last accessed on July 14, 2014. 4.2.2

[47] USB Communication Device Class, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/USB_communications_device_class, last accessed on July 14, 2014. 3

[48] HUAWEI K3765, http://www.huaweidevice.com.eg/Product-Description/Data-cards-K3765.php, last accessed on July 14, 2014. 4.2.5

[49] Draisberghof's USB_ModeSwitch forums, http://www.draisberghof.de/usb_modeswitch/bb/viewtopic.php?p=1343, last accessed on July 14, 2014. 4

[50] Hayes Command Set, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Hayes_command_set, last accessed on July 14, 2014. 5

[51] Specification of CAN Interface, AUTOSAR site, http://www.autosar.org/download/R4.1/AUTOSAR_SWS_CANInterface.pdf, last accessed on July 14, 2014. 4.2.6

[52] PEAK-System, http://www.peak-system.com/?&L=1, last accessed on July 14, 2014. 4.2.6

[53] Circular buffer, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Circular_buffer, last accessed on July 14, 2014. 6

[54] CoreLink^{TM} TrustZone Address Space Controller TZC-380 Technical Reference Manual, ARM Infocenter, all ARM non-confidential Technical Publications, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0431c/index.html, last accessed on July 14, 2014. 4.2.8

[55] ARM Architecture Reference Manual ARMv7-A, ARM Infocenter, all ARM non-confidential Technical Publications, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html, last accessed on July 14, 2014. 7, 4.2.8

[56] Doxygen, Generate documentation from source code, http://www.stack.nl/~dimitri/doxygen/, last accessed on July 14, 2014. 4.2.9

[57] Android 4.3, Jelly Bean, http://www.android.com/about/jelly-bean/, last accessed on July 14, 2014. 4.3

[58] The Linux Kernel Archives, https://www.kernel.org/, last accessed on July 14, 2014. 4.3.1

[59] Meccano, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Meccano, last accessed on July 14, 2014. 1

[60] AUTOSAR, AUTomotive Open System ARchitecture, http://www.autosar.org/, last accessed on July 14, 2014. A.1

[61] CAN protocol, CAN in Automation (CIA), http://www.can-cia.org/index.php?id=systemdesign-can-protocol, last accessed on July 14, 2014. A.2

[62] CAN bus, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/CAN_bus, last accessed on July 14, 2014. A.2

[63] Low-voltage differential signaling, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Low-voltage_differential_signaling, last accessed on July 14, 2014. 1

[64] SABRE Platform for Smart Devices based on the i.MX 6 Series, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=RDIMX6SABREPLAT, last accessed on July 14, 2014. A.4, A.3

[65] Universal Asynchronous Receiver/Transmitter, from Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter, last accessed on July 14, 2014. A.6

Appendix A

Long lists and other material

In this chapter, it is presented the following contents:

- AUTOSAR A.1;
- CAN Protocol A.2;
- Freescale i.MX 6Quad A.3;
- FlexCAN A.4;
- IOMUX Controller A.5;
- Universal Asynchronous Receiver/Transmitter A.6;
- Tutorial to build Android A.7;
- Tutorial to develop an *Hello World!* application for Android A.8;
- Tutorial to develop an *Hello World!* application for QNX A.9;
- AUTOSAR OS and RTE emulator research A.10.

A.1 AUTOSAR

AUTOSAR is a standardized and open SW architecture that can be used for the several in-vehicle systems without compromising the quality and cost-efficiency. AUTOSAR aims to the improvement of innovative electronic systems on automotive vehicles through the achievement of modularity, scalability, transferability and re-usability technical goals.

Its scope includes all vehicle domains, and this standard serves as a platform upon which vehicle applications can be implemented. This process aims the minimization of the barriers between functional domains and the mapping of functions and functional networks almost independently of the associated HW. In order to achieve the previously referred technical goals, AUTOSAR provides a common SW infrastructure for automotive systems, based on standardized interfaces, including different layers.

The illustration presented in Figure A.1 describes an AUTOSAR architectural overview:

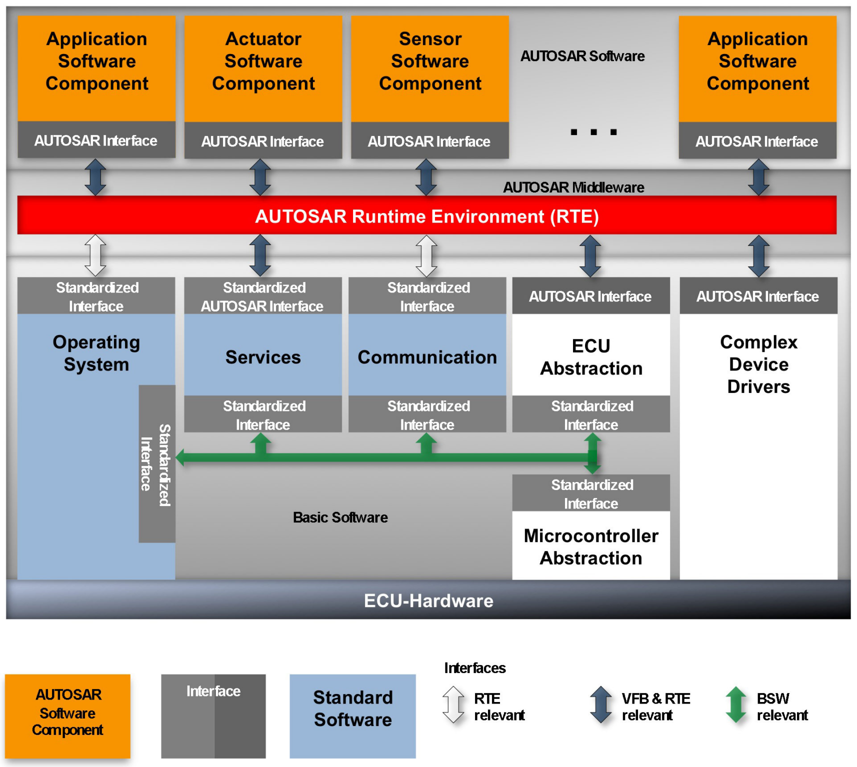


Figure A.1: AUTOSAR ECU SW architecture [60].

The relations between the SW and abstraction layers of AUTOSAR SW architecture and the ECU micro-controller components/devices are well visible in Figure A.1.

AUTOSAR main goal is to achieve the functional interfaces standardization across manufacturers and suppliers and also the interfaces standardization between different SW layers.

From bottom to top, the AUTOSAR layers are composed by:

- Basic Software (BSW) – The standardized SW layer that provides the services to the AUTOSAR SW components by interacting with the hardware. The functional part needs the basic SW layer to run, but the BSW layer does not perform any functional job itself. It contains standardized and ECU specific components, mapped in the services layer, the communication layer, the operating system layer, the micro-controller abstraction, the ECU abstraction layer and the complex device drivers layer;

- Runtime Environment (RTE) – RTE implements the communication functions for inter- and intra-ECU information exchange. It provides communication abstraction to the Software Components attached to it, which is independent from whether intra-ECU or inter-ECU (CAN, etc.) communications channels are used. RTEs are ECU specific and need to be tailored according to the communication requirements of the Software Components running on top of them;

- Virtual Functional Bus (VFB) – The sum of all communication mechanisms and interfaces to the BSW result on this layer of abstraction (technology independent). VFB functionality is provided by well-defined communication patterns;

- Software Component (SWC) – Each Software Component within an ECU encapsulates part of the application functionality, as an application on AUTOSAR consists of interconnected SWCs. SWCs contain well defined interfaces, described and standardized within AUTOSAR. SWCs are ECU specific and shall only be assigned to one ECU within a vehicle scope, although they can be reused. SWCs are ECU, micro-controller and other SWCs independent. The SWCs are responsible for the functionality features of an AUTOSAR application.

For more information related to AUTOSAR, please refer to [60].

A.2 CAN Protocol

This protocol consists on message exchanging between nodes. Each node has an ID and when the transmitting node is preparing the message to be transmitted, it encapsulates the destination node ID into the message. After the message is sent, the listening nodes will try to match the ID present in the transmitted message with its own ID. If the result of the matching process is positive, the respective listening node will receive that message. Apart from the ID, a message has also the data to be transmitted and its length in bytes, and other more information fields.

The transmission process bases itself on a binary model of ‘dominant’ and ‘recessive’ bits, where the dominant ones are 0 and the recessives ones are 1. This model is mostly used when two or more nodes are trying to transmit a message at the same time and it is needed to choose the first to transmit. That selection is made using priorities. Each node has a priority that is represented by its own ID, so the lower binary number its ID is the higher priority the node will have. This part of the transmission is called arbitration process.

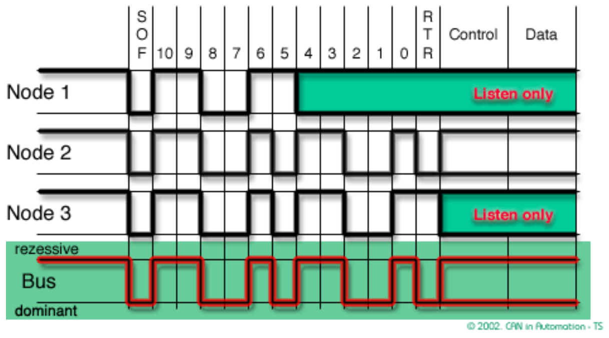


Figure A.2: Arbitration process with three nodes [61].

In Figure A.2, it is represented an arbitration process with three nodes trying to transmit at the same time. The arbitration winner is node 2. It is so because it has an higher priority than node 1 and, despite having the same destination ID of the node 3, the latter is a data request frame, making node 2 more high-priority than node 3.

For more information about CAN protocol, please refer to [62].

A.3 Freescale i.MX 6Quad

It contains a quad-core ARM Cortex-A9 processor running up to 1.2 GHz with 1 MB of L2 cache and 64-bit DDR3 or 2-channel, 32-bit LPDDR2 support. In addition, i.MX 6Quad family processors also integrates a video processing unit, a 2D/3D graphics processing unit and several others devices that make these processors qualified for automotive, industrial and consumer markets – such as FlexCAN (section 3.3.3), HDMI v1.4, LVDS[1] PCI Express®, MIPI camera

[1]Low-Voltage Differential Signaling (LVDS) is a technical standard that specifies electrical characteristics of a differential, serial communications protocol. The typical applications are high-speed video, graphics, video camera

port and SATA-2 (Figure A.3).

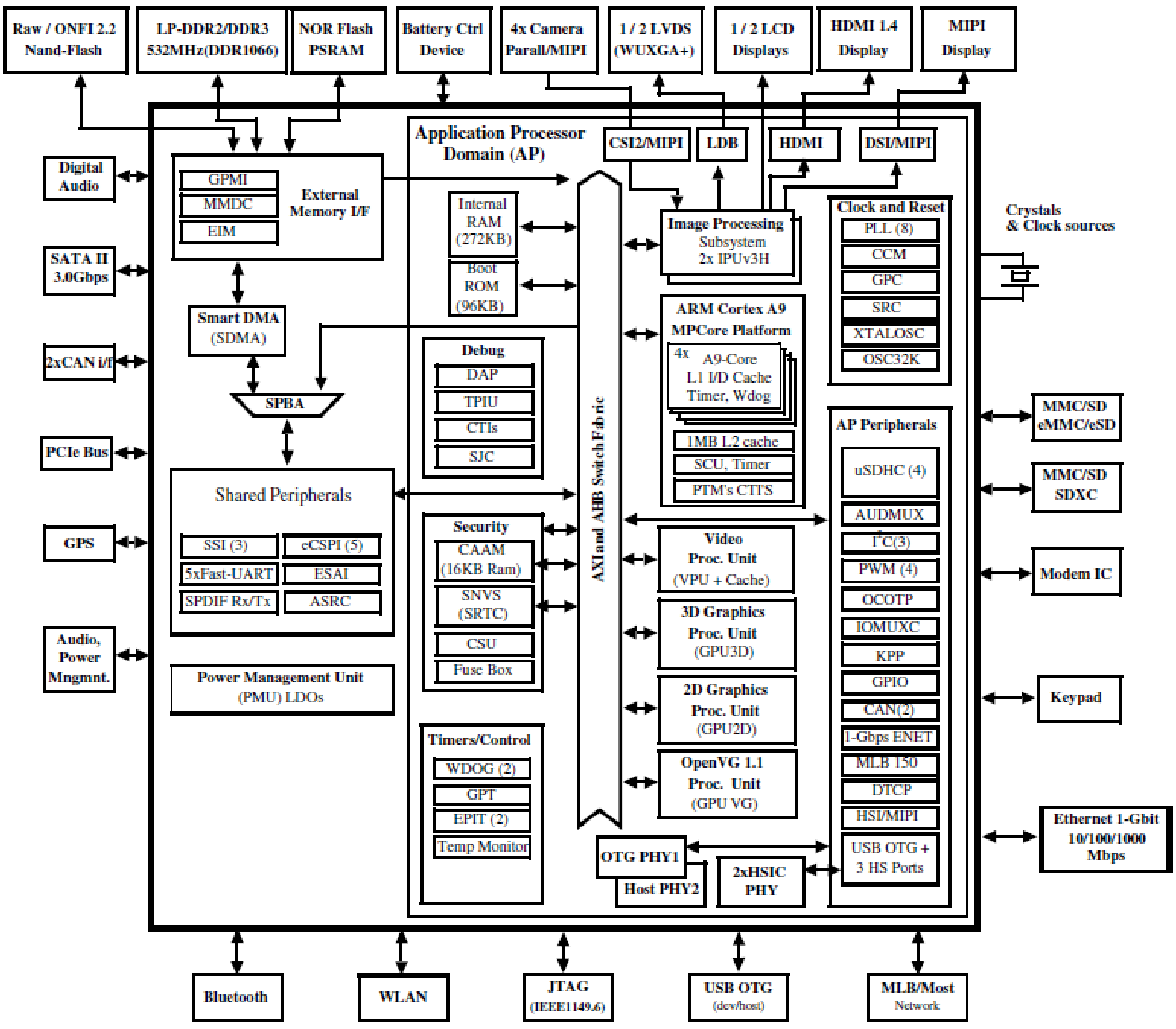


Figure A.3: Simplified block diagram of i.MX 6Quad from i.MX6Q Reference Manual [41].

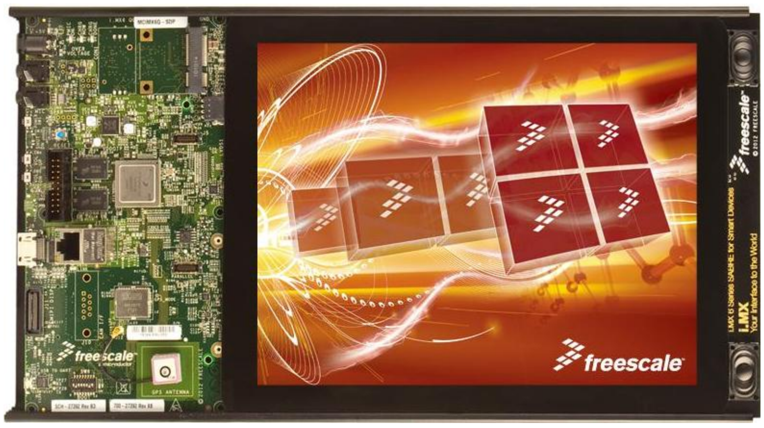


Figure A.4: SABRE Platform for Smart Devices based on the i.MX 6 Series [64].

data transfers, and general purpose computer buses [63].

In the development of xLuna Automotive, it was used a Freescale i.MX6Q SABRE Platform for Smart Devices (SABRE SD) [64], which includes a SABRE SD board, based on an i.MX 6Quad processor, and a 10.1" LVDS display with capacitive sensing (Figure A.4).

For more information about i.MX 6Quad processors, please refer to [37].

A.4 FlexCAN

In i.MX6Q, there are two FlexCAN modules, which means that this SoC supports up to 128 MBs (Figure A.5).

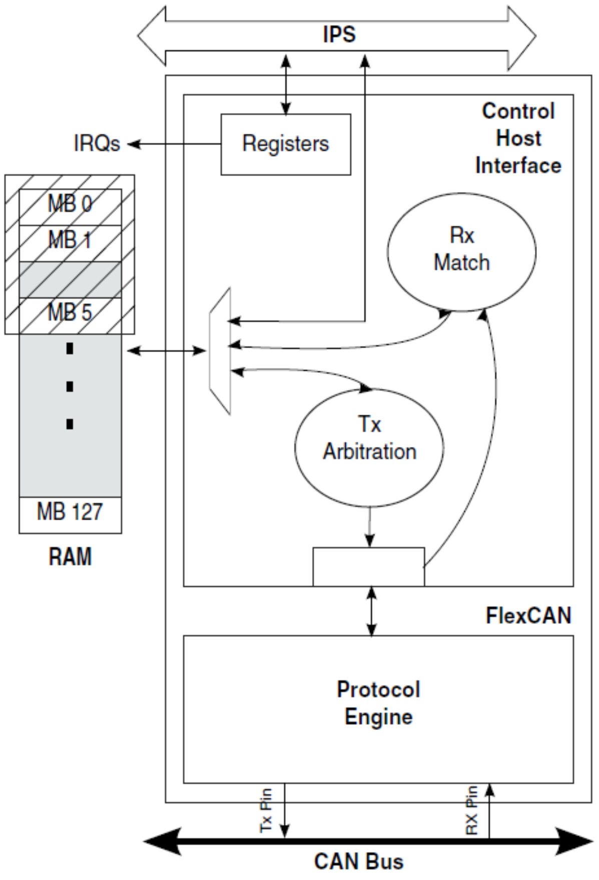


Figure A.5: FlexCAN block diagram from i.MX6Q Reference Manual [41].

The Rx process is essential composed by another process, the matching process. This process consists on the scanning of all the *active* and *empty* Rx MBs to find the one with the same ID of the incoming frame where it will be stored. It is called *matched structures* to the MBs that has the same ID of the incoming frame, but only one, the *matching winner*, will receive the frame. The selection of the matching winner is made between two possible ways:

- The first free-to-receive MB firstly and the last non free-to-receive MB secondly;
- Lowest MB number first.

The incoming frame is initially stored in an hidden MB called Rx Serial Message Buffer (Rx SMB) and at the end of the matching process the data contained in it is moved to the matching winner and its status is updated to *full*.

The Tx process is also mainly composed by another process, but in this case it is the arbitration process. This process scans the *active* Tx MBs searching the one that holds the message to be sent in the next opportunity, the *arbitration winner*. The arbitration process chooses the arbitration winner by one of two possible ways:

- Lowest MB number first;
- Highest MB priority first.

Once the arbitration winner is found, its content is copied to an hidden MB called Tx Serial Message Buffer (Tx SMB) and the MB's status switches to *inactive*.

For more information, please refer to i.MX6 Reference Manual [41].

A.5 IOMUX Controller

The IOMUX (I/O Muxing) Controller (IOMUXC) enables the SoC to share one pin's pad to several functional blocks by multiplexing the pad's I/O signals. Each one of these pads has up to 8 muxing options, or ALT modes as referred in i.MX6 Reference Manual [41], which can have a specific settings, such as hysteresis field and pull up or keeper. Each external SoC device requires its own pad settings. That settings are specified in the device's documentation.

This controller was essential in xLuna development because, as referred above, there was needed to configure the pads of the I/O signals of the processor in order to make the on-board devices to work properly. As it was needed to develop the device drivers of all the devices used, the IOMUXC was presented in the most of them.

In addition to Reference Manual, the I/O signals of a specific device can be found with more detail in SABRE SD board schematics [45].

For more information about IOMUXC, please refer to i.MX6 Reference Manual [41].

A.6 Universal Asynchronous Receiver/Transmitter

There are three modes of operation of UART: serial RS-232 Non-Return-to-Zero (NRZ) mode, 9-bit RS-485 mode and IrDA mode. The RS-232 NRZ mode consists on a 7 or 8 data bits for characters. The NRZ designation means that the binary one is represented physically by a positive voltage and the binary zero is represented physically by a negative voltage. The 9-bit RS-485 mode only differs the latter on the number of data bits, being in this case 9 data bits for characters. Both of RS-232 and RS-485 modes are limited to a bitrate of at most 5 Mbit/s. Finally, the IrDA mode consists on frames with 8 data bits for characters and with a transfer speed up to 115.2 Kbit/s. In this case, the binary one is represented physically by no pulse, while the binary zero is represented by a positive pulse.

The UART modules had a very important role in xLuna development by providing the console of the board. It was used in the RS-232 mode.

For more information about UART, please refer to [65] and to chapter 64 of i.MX6Q Reference Manual [41].

Memorandum

Header			
Author:	João Filipe Serra, Intern	**Doc. Reference:**	<CSW-PPPP-2011-TTT-NNNNN>
Contact:	jfserra@itgrow.pt	**Date:**	2013-10-17
Version:	<X.xx>	**Project name:**	<Project Name>
Purpose			
Explain step-by-step how to set up your local work environment to build the Android source files in Ubuntu 12.04 amd64			

Notifications		
Name	**Company**	**Contact**
<Name>	<Company>	<E-mail>

Installing Java Development Kit 6

1 – Register at www.oracle.com.
2 – Go to *Downloads->Java for Developers*.
3 – Click on *Previous Releases*.
4 – Click on *Java SE 6*.
5 – Click on *Java SE Development Kit 6u45*.
6 – Download the Linux x64 (no rpm) version.
7 – Open the terminal and to the downloaded file location and set it executable:

7.1 – *chmod a+x jdk-6u45-linux-x64.bin*

8 – Execute the file:

8.1 – *./jdk-6u45-linux-x64.bin*

9 – Change the new folder name from *jdk1.6.0_x* to *java-6-oracle*:

9.1 – *mv jdk1.6.0_x java-6-oracle*

10 – Move that folder to */usr/lib/jvm*:

10.1 – *mv java-6-oracle /usr/lib/jvm*

11 – Download a tool to help you defining the new directory of Java:

11.1 – *wget http://webupd8.googlecode.com/files/update-java-0.5b*

12 – Set that file executable:

12.1 – *chmod a+x update-java-0.5b*

13 – Run it with *root* permissions:

13.1 – *sudo ./update-java-0.5b*

14 – In the new window, select *java-6-oracle* and click *Ok*.

15 – Finally, check if your switch was successful by executing the following command lines:

15.1 – *java –version*

15.2 – *javac –version*

Installing Required Linux Packages

1 – Open the terminal and execute the following command lines:

1.1 – *sudo apt-get install git gnupg flex bison gperf build-essential zip curl libc6-dev libncurses5-dev:i386 x11proto-core-dev libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown libxml2-utils xsltproc zlib1g-dev:i386*

1.2 – *sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/libGL.so*

NOTE: After the step 1.1, some systems get damaged on graphics drivers and Ubuntu does not start up. So, you may need to reinstall graphics drivers or, if you are using VirtualBox like I do, you just have to reinstall VirtualBox Guest Additions.

Configuring USB Access

1 – Configure the system so that regular users can directly access USB devices. For that, create a file */etc/udev/rules.d/51-android.rules* as the root user and copy the following lines in it (replace <username> by the actual username):

```
# adb protocol on passion (Nexus One)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e12", MODE="0600",
OWNER="<username>"
# fastboot protocol on passion (Nexus One)
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", ATTR{idProduct}=="0fff", MODE="0600",
OWNER="<username>"
# adb protocol on crespo/crespo4g (Nexus S)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e22", MODE="0600",
OWNER="<username>"
# fastboot protocol on crespo/crespo4g (Nexus S)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e20", MODE="0600",
OWNER="<username>"
# adb protocol on stingray/wingray (Xoom)
SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", ATTR{idProduct}=="70a9", MODE="0600",
OWNER="<username>"
```

# fastboot protocol on stingray/wingray (Xoom)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="708c", MODE="0600", OWNER="<username>"
# adb protocol on maguro/toro (Galaxy Nexus)
SUBSYSTEM=="usb", ATTR{idVendor}=="04e8", ATTR{idProduct}=="6860", MODE="0600", OWNER="<username>"
# fastboot protocol on maguro/toro (Galaxy Nexus)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e30", MODE="0600", OWNER="<username>"
# adb protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d101", MODE="0600", OWNER="<username>"
# adb protocol on panda (PandaBoard ES)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="d002", MODE="0600", OWNER="<username>"
# fastboot protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d022", MODE="0600", OWNER="<username>"
# usbboot protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d00f", MODE="0600", OWNER="<username>"
# usbboot protocol on panda (PandaBoard ES)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d010", MODE="0600", OWNER="<username>"
# adb protocol on grouper/tilapia (Nexus 7)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e42", MODE="0600", OWNER="<username>"
# fastboot protocol on grouper/tilapia (Nexus 7)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e40", MODE="0600", OWNER="<username>"
# adb protocol on manta (Nexus 10)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4ee2", MODE="0600", OWNER="<username>"
# fastboot protocol on manta (Nexus 10)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4ee0", MODE="0600", OWNER="<username>"

Installing Repo

1 – Make sure you have a bin/ directory in your home directory and that it is included in your path:

1.1 – *mkdir ~/bin*
1.2 – *PATH=~/bin:$PATH*

2 – Download the Repo tool and ensure that it is executable:

2.1 – *curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo*
2.2 – *chmod a+x ~/bin/repo*

3 – Create an empty directory to hold your working files (replace WORKING_DIRECTORY by the working directory name you want):

3.1 – *mkdir WORKING_DIRECTORY*

3.2 – *cd WORKING_DIRECTORY*

4 – To bring down the latest version of Repo with all its most recent bug fixes, run the following command line:

4.1 – *repo init -u https://android.googlesource.com/platform/manifest*

5 – Finally, run the following command line (this can take several hours because it will be downloaded a great amount of data):

5.1 – *repo sync*

Building and Installing Android Source Tree

1 – In your working directory, run the following terminal command line to initialize the environment:

1.1 – *. build/envsetup.sh*

2 – Choose to build for the emulator:

2.1 – With limited access (suited for production):

2.1.1 – *lunch full-user*

2.2 – Like *user* but with root access and debuggability (preferred for debugging):

2.2.1 – lunch full-userdebug

2.3 – With all debugging enabled:

2.3.1 – *lunch full-eng*

3 – Build the source with the following command line:

3.1 – *make –jN*, where N is the number of tasks that is between 1 and 2 times the number of hardware threads on the computer being used for the build.

Run the Build

1 – As the emulator is added to your path automatically by the build process, to run your build on the emulator, type:

1.1 – *emulator*

Any Questions or Doubts and More Information

- Explore the left menu in *Source* section of source.android.com. There you can access different kinds of tutorials and some tricks to build and develop your own build of Android.

Memorandum

Header			
Author:	João Filipe Serra, Intern	**Doc. Reference:**	<CSW-PPPP-2011-TTT-NNNNN>
Contact:	jfserra@itgrow.pt	**Date:**	2013-10-23
Version:	<X.xx>	**Project name:**	<Project Name>
Purpose			
Explain step-by-step how to set up your Windows local work environment to develop Android applications and develop a *Hello World* application			

Notifications		
Name	**Company**	**Contact**
<Name>	<Company>	<E-mail>

Downloading and Installing the Android Software Development Kit (SDK)

1 – Go to developer.android.com.
2 – Click on *Get the SDK* button on the bottom of the page.
3 – Download the Android SDK by clicking on the blue button on the right of the page.
4 – Unzip the downloaded *.zip*. The folder contained in the archive is where the IDE and all the tools for the purpose are located, so it is from this folder that you will be using the work resources.
5 – To open the IDE, which in this case is Eclipse, execute *eclipse.exe* located in eclipse folder.
6 – You are ready to start developing.

Creating a New Android Virtual Device (AVD)

1 – Open the IDE.
2 – Go to *Window->Android Virtual Device Manager*.
3 – Click on *New...* button.
4 – Give it a name.
5 – Choose what device you want. I chose *Nexus S* because it has a nice screen aspect ratio and it uses little RAM.
6 – Choose the target, which is the Android version that the AVD will be running. You must choose it with special attention because the AVD just will be able to run applications compatible with the target you chose.

NOTE: The available APIs are the ones that you have installed. You can install what API you want in Android SDK Manager window, which can be open in *Window* submenu in the toolbar.

7 – Click on *Ok* and it is done.

8 – You can start it whenever you want by clicking on *Start...* in AVD Manager window.

Creating a New Android Application Project

1 – Go to *File->New->Android Application Project*.
2 – Fill the empty fields.
3 – Select *Minimum Required SDK*, *Target SDK*, *Compile With* and *Theme* according to your preferences, and click *Next*. Just note that *Compile With* option is the Android version that your AVD needs to have to run your new application.
4 – On the next screen you can configure your project. It is recommended leaving the default selections.
5 – Customize the application icon or leave it as the default icon.
6 – Select an activity template. The more simple and common activity template is *BlankActivity*.
7 – Once you finished creating your project, you will be shown two files in the main screen: the *java* file of the activity and its graphical configurations file (*.xml*). Let's begin by modifying the *.xml* file:

7.1 – There are two ways to edit the graphics of the application: the graphical layout and the manual editing of the *.xml* file. The lower level you work, the more concise your work will be, so it is recommended to edit manually the *.xml* file, with some assistance of graphical layout like adding and positioning the widgets. So, let's add a *Button* from *Form Widgets* in *Graphical Layout* tab by dragging it to the activity screen;
7.2 – Go to *.xml* file code tab;
7.3 – Find the code <*Button*. The space between <*Button* and the next */>* is the area of the button settings. Now, let's associate its click event to a function, which you will create later in *java* file. To do that, you need to add the following code line in the button settings area:

7.3.1 – *android:onClick="button_click"*

7.4 – As you can see, it already exist a *TextView* widget. Note that all widget have their *id*. You can change them or just take note of them because you will need them later.

8 – Now let's do the programing:

8.1 – Go to your activity *java* file tab;
8.2 – By default, two functions are created when you create an activity but right now you will only edit one of them, the *onCreate* function. Let's declare the handlers of the widgets you have created:

8.2.1 – *TextView textView1 = (TextView) findViewById(R.id.textView1)*;
8.2.2 – *Button button1 = (Button) findViewById(R.id.button1)*;

8.3 – Through the handlers you can access and configure all the aspects of the widget in question. So, let's change the text of the TextView widget and the Button widget:

8.3.1 – *textView1.setText("Hello World! Please click the button below...")*;
8.3.2 – *button1.setText("Click me!")*;

8.4 – Now let's create the function that was associate to the Button widget:

8.4.1 – *public void button_click(View view) { }*

8.5 – Finally, let's give to it some work to do:

8.5.1 – Declare the handlers:

8.5.1.1 – TextView textView1 = (TextView) findViewById(R.id.textView1);
8.5.1.2 – Button button1 = (Button) findViewById(R.id.button1);

NOTE: You need to declare the widgets handlers in all functions you want to use them.

8.5.2 – Change the TextView text:

8.5.2.1 – *textView1.setText("Text changed!");*

9 – Save all files: *File->Save all*.
10 – You can run or debug the project in *Run* submenu in the toolbar. So, let's run the application.
11 – Select *Android Application* and then *Ok*.
12 – The AVD will be open and run the application.

Any Questions or Doubts and More Information

- Consult the content in *Develop* menu in developer.android.com.

Memorandum

Header			
Author:	João Filipe Serra, Intern	**Doc. Reference:**	<CSW-PPPP-2011-TTT-NNNNN>
Contact:	jfserra@itgrow.pt	**Date:**	2013-10-16
Version:	<X.xx>	**Project name:**	<Project Name>
Purpose			
Explain step-by-step how to install and configure QNX Software Development Platform 6.5.0			

Notifications		
Name	**Company**	**Contact**
<Name>	<Company>	<E-mail>

QNX Account and Licensing

1 – Register at www.qnx.com.
2 – Get a non-commercial license:

2.1 – At the top of the page, go to *Downloads->QNX Product Evaluation*;
2.2 – On the right side of the page, click on *Other License Keys->Non-commercial*;
2.3 – Fill the fields and ask for the license on the bottom of the page.

Download the Software

1 – Log in at www.qnx.com.
2 – At the top of the page, go to *Downloads->QNX Software Development Platform 6.5.x*.
3 – Download the software:

3.1 – For the target system:

3.1.1 – For PCs with APIC chipsets (x86):

3.1.1.1 – *QNX Software Development Platform 6.5.0 - QNX Neutrino RTOS Installation and Boot CD [X86-only for APIC chipsets];*

3.1.2 – For all targets:

3.1.2.1 – *QNX Software Development Platform 6.5.0 - QNX Neutrino RTOS Installation and Boot DVD [All targets].*

3.2 – For the development host:

3.2.1 – For Windows Hosts:

3.2.1.1 – *QNX Software Development Platform 6.5.0 - Windows Hosts*;
3.2.1.2 – *QNX Software Development Platform 6.5.0 Service Pack 1 - Windows Hosts*.

3.2.2 – For Linux Hosts:

3.2.2.1 – *QNX Software Development Platform 6.5.0 - Linux Hosts*;
3.2.2.2 – *QNX Software Development Platform 6.5.0 Service Pack 1 - Linux Hosts*.

Install QNX Neutrino Real-Time Operating System

1 – Install QNX Neutrino RTOS by one of the following:

1.1 – Burn the CD/DVD, reboot the PC with the burned media in the optical drive and proceed with the installation;
1.2 – Create a Virtual Machine with 1GB of RAM (recommended), 3GB of disk space (minimum) and with only one CPU (on multi-core processors PCs), start the Virtual Machine with the CD/DVD ISO mounted on a virtual drive and proceed with the installation.

2 – After finishing the installation, log in as root with no password.
3 – At www.qnx.com, log in and go to *Downloads->QNX Software Development Platform 6.5.x*.
4 – Download *QNX Software Development Platform 6.5.0 Service Pack 1 - Neutrino Hosts*.
5 – Open the terminal, navigate to the file location and make it executable, using the following command line:

5.1 – *chmod a+x qnxsdp-6.5.0SP1-xxxxxxxxxxxx-nto.sh*

6 – Now, run it:

6.1 – *./qnxsdp-6.5.0SP1-xxxxxxxxxxxx-nto.sh*

7 – Connect the target to the same network of the host.
8 – Get the target IP, by opening the terminal and execute the *ifconfig* command.
9 – Ping the target in the host, to make sure that they are in the same network.

Install QNX SDP in Windows Hosts

1 – Execute the file *qnxsdp-6.5.0-xxxxxxxxxxxx-win32.exe* and proceed with the installation (it is advisable install in the default directory).
2 – After finishing the installation, log out and log back in again, so that the environment is set up correctly.
3 – Execute the file *qnxsdp-6.5.0SP1-xxxxxxxxxxxx-win32.exe* and proceed with the installation.

Install QNX SDP in Linux Hosts

1 – In order to run the installer and the IDE on Ubuntu 64-bit, you need to install the 32-bit libraries by executing the following command line in the terminal:

1.1 – *sudo apt-get install ia32-libs*

2 – Open the terminal, navigate to the location of the files downloaded before and make them executable, using the following command lines:

2.1 – *chmod a+x qnxsdp-6.5.0-xxxxxxxxxxxx-linux.bin*
2.2 – *chmod a+x qnxsdp-6.5.0SP1-xxxxxxxxxxxx-linux.bin*

3 – Execute the file *qnxsdp-6.5.0-xxxxxxxxxxxx-linux.bin* and proceed with the installation (it is advisable install in the default directory).
4 – After finishing the installation, log out and log back in again, so that the environment is set up correctly.
5 – Execute the file *qnxsdp-6.5.0SP1-xxxxxxxxxxxx-linux.bin* and proceed with the installation.

Configure QNX Momentics IDE and Create a New Project in Hosts

1 – Open QNX Momentics IDE:

1.1 – In Windows, double click in QNX Momentics IDE icon on the desktop;
1.2 – In Linux, open the terminal, go to */opt/qnx650/* and execute the file *run_qde.sh*.

2 – After choosing your workspace directory, go to *File->New->Project->QNX C Project*.
3 – Choose the project name (*hello_world* for example), make sure that *Generate default file* is checked and leave *Add project to working sets* unchecked.
4 – Choose the target architecture (x86) in *Build Variants* tab.
5 – In the target terminal execute *qconn* command.
6 – Go to *Window->Open Perspective->QNX System Information* to add the target to the project.
7 – In the empty *Target Navigation* view, press the right mouse button and select *New QNX Target...*.
8 – Give a name to the target and write its IP in the field provided.
9 – Go back to *C/C++ Perspective* (*Window->Open Perspective->Other...->C/C++*).
10 – Press the right mouse button on the project name and select *Properties*.
11 – On the left menu, select *C/C++ Project* and, in *Build Variants* tab, check if the right target architecture is selected and click *Ok*.
12 – After IDE rebuild the project, you can do the programming. Let's just change the string in *printf()* and write *Hello World!\n*.
13 – To run the project, you need to create your own launch configurations:

13.1 – From the dropdown menu beside the *play* green icon on toolbar, select *Run Configurations...*.
13.2 – On the left menu, select *C/C++ QNX QConn (IP)* and click on *New Launch Configuration* icon on toolbar.

13.3 – In *Main* tab, select your project in *Project* field, click on *Search Project...*, select your binary (the binaries with debug information include a suffix of *_g*), select your target and click *Apply*.

13.4 – Now you can *Run* the project.

14 – To run the project in debug mode, you have to select your binary that includes the suffix of *_g* in launch configurations, like described above.

Any Question or Doubts

- Consult the installation notes of the respective software at www.qnx.com, *Downloads->QNX Software Development Platform 6.5.x*.
- Consult the documents present at www.qnx.com, *Downloads->QNX Software Development Platform 6.5.x->QNX SDP 6.5.0 SP1 Documentation in PDF*.

MEMORANDUM

HEADER

AUTHOR:	José Rui Sampaio, <Position>	DOC. REFERENCE:	<CSW-PPPP-2011-TTT-NNNNN>
CONTACT:	<E-mail>	DATE:	2014-03-14
VERSION:	<X.xx>	PROJECT NAME:	<Project Name>

PURPOSE

<Short description>

NOTIFICATIONS

NAME	COMPANY	CONTACT
<Name>	<Company>	<E-mail>

DESCRIPTION

AUTOSAR OS Emulator Research

TABLE OF CONTENTS

TABLE OF TABLES

TABLE OF FIGURES

1. INTRODUCTION

1.1. OVERVIEW

AUTOSAR applications implementing a car’s ECU are designed to run within AUTOSAR architecture, assuming the presence of certain services implemented as the BSWMs. One of these modules is the AUTOSAR OS layer, which defines the operating system on which AUTOSAR applications run. It is an automotive oriented OS, based on the industry standard OSEK OS, containing a restrictive feature set to fulfil the AUTOSAR requirements specification.

To run AUTOSAR on general-purpose/proprietary OSs, AUTOSAR OS defined interfaces need to be provided as an Abstraction Layer or emulated.

The document refers to the research performed on AUTOSAR OS and RTE Layers in order to integrate AUTOSAR application on xLuna by creating the referred abstraction layer.

1.2. SCOPE

The document shall be used within the scope of xLuna project.

1.3. AUDIENCE

The document intended audience is xLuna project’s team.

1.4. DEFINITIONS AND ACRONYMS

TABLE 1 presents the list of definitions used throughout this document. The definitions presented in [AD-1] are also applicable.

NAME	DESCRIPTION
Applicable Document	A document is considered applicable if it complements this document. All its content is directly applied as if it was stated as an annex of this document.
Reference Document	A document is considered a reference if it is referred but not applicable to this document. Reference documents are mainly used to provide further reading.

TABLE 1: Definitions

TABLE 2 presents the list of acronyms used throughout this document. The acronyms presented in [AD-1] are also applicable.

ACRONYM	DESCRIPTION
AD	Applicable Document

ACRONYM	DESCRIPTION
AUTOSAR	AUTomotive Open System ARchitecture
BSW	Basic Software
BSWM	Basic Software Module
CSW	Critical Software, S.A.
ISR	Interrupt Service Routine
OS	Operating System
RD	Reference Document
RTE	Run-Time Environment
SPEM	Software Process Engineering Meta-Model
SWC	Software Component
TBC	To be confirmed
TBD	To be defined
VFB	Virtual Function Bus

TABLE 2: Acronyms

1.5. APPLICABLE DOCUMENTS

TABLE 3 presents the list of the documents that are applicable to this document. A document is considered applicable if it complements this document. All its content is directly applied as if it was stated as an annex of this document.

APPLICABLE DOCUMENT	DOCUMENT NUMBER
[AD-1] CSW Glossary, Critical Software, S.A.	CSW-QMS-2002-GLS-0353
[AD-2] Specification of Operating System, AUTOSAR Administration, 09/10/2013	AUTOSAR_SWS_OS
[AD-3] Specification of RTE, AUTOSAR Administration, 29/10/2013	AUTOSAR_SWS_RTE
[AD-4] Requirements on Operating System, AUTOSAR Administration, 31/10/2013	AUTOSAR_SRS_OS
[AD-5] Requirements on RTE, AUTOSAR Administration, 24/10/2013	AUTOSAR_SRS_RTE
[AD-6] General Specification of Basic Software Modules, AUTOSAR Administration, 31/10/2013	AUTOSAR_SWS_BSWGeneral

APPLICABLE DOCUMENT	DOCUMENT NUMBER
[AD-7] Specification of Basic Software Mode Manager, AUTOSAR Administration, 31/10/2013	AUTOSAR_SWS_BSWModeManager

TABLE 3: Applicable documents

1.6. REFERENCE DOCUMENTS

TABLE 4 presents the list of reference documents. A document is considered a reference if it is referred but not applicable to this document. Reference documents are mainly used to provide further reading.

REFERENCE DOCUMENT	DOCUMENT NUMBER
[RD-1] Demonstration of Integration of AUTOSAR into a MM/T/HMI ECU, AUTOSAR Administration , 23/11/2010	AUTOSAR_TR_IntegrationintoMMTHMIECU
[RD-2] Main Requirements, AUTOSAR Administration, 31/10/2013	AUTOSAR_RS_Main
[RD-3] Technical Overview, AUTOSAR Administration, 27/04/2011	AUTOSAR_TechnicalOverview
[RD-4] OSEK/VDX Operating System Specification 2.2.3, OSEK, 17/02/2005	os223

TABLE 4: Reference documents

2. BACKGROUND

2.1. INTRODUCTION

xLuna is an hypervisor/monitor based software that controls OSs access to hardware. xLuna project for the automotive domain intends to perform the described functions for a general purpose OS and a RTOS running within the same infotainment board.

The main goal of the research specified in this document is the integration of AUTOSAR application on xLuna project for the automotive domain. To do it, it is intended to run AUTOSAR applications on a general-purpose RTOS, like FreeRTOS.

Documentation present on AUTOSAR official site contains a document demonstrating how multimedia, telematics and human machine interface applications could be developed using AUTOSAR architecture [RD-1]. Integration is proposed for SWCs and BSWMs. However, it only contains a proposal and not a concrete implementation of such application, as the complexity of the integration process is proportional to the complexity of the AUTOSAR application itself. Some of the proposed concepts can be used as a starting point for the integration of AUTOSAR on xLuna automotive project. An overview of some basic concepts of the AUTOSAR architecture is firstly presented.

2.2. OVERVIEW

Conceptually, the layer that is placed on top of the architecture is the SWCs layer. The SWCs are completely unaware of every layer placed below the RTE, and they use the latter to perform all communication activities with other modules. Each instance of a SWC can be classified as Atomic, as each instance of it should only be present in one ECU within a vehicle.

SWCs can be classified as “application software-components” and “sensor-actuator software-components”. The first ones perform functionalities that are not hardware-dependent. Due to that fact, they are portable and can be easily integrated on other AUTOSAR applications (but not within the same vehicle). Such integration process complexity increases for “sensor-actuator software-components” as these are hardware dependent, more-specifically, sensor or actuator dependent. It must be ensured that the target ECU contains all the devices required by these SWCs.

A more generic representation of a SWC can be considered. Single components (application and/or sensor-actuator) can be logically interconnected into “compositions”, which is the same to say that several components that can be packaged as a single component. These are considered to be more “ECU-portable” than atomic components as they can be part of different ECUs within the same vehicle.

The BSWMs are placed below the RTE. These are the modules responsible for directly accessing the ECU services through the ECU abstraction layer, and other BSWMs. By doing so, they can provide a service required by the SWCs, by the RTE or by other BSWMs. The BSW layer is necessary to run the functional part of the software. It contains standardized and ECU specific modules to provide different services as required. These modules are tailored to each AUTOSAR application, which means they are not hardware and ECU independent.

SWCs and BSWMs layers cannot directly communicate with each other. They do it through the use of the RTE layer. In fact, the only layer the SWCs rely on is the RTE, and SWCs use it to communicate with other SWCs and with the BSWMs. The RTE is the realisation of the interfaces defined by the VFB.

The VFB is an abstraction of the components communication interconnections within the AUTOSAR architecture. Communication is performed through the usage of ports. A port always belongs to one component. It represents the point of interaction of that component with other components. A port can provide or require a particular service or data element.

In the AUTOSAR architecture, ports are characterized by port-interfaces. Each BSWM and SWC is configured with its own interfaces. This allows SWCs to communicate with each other, whether they are on the same ECU or not. It also allows SWCs to communicate with BSWMs of the same ECU containing ports and runnables. A port can be:

- Sender-receiver port-interface – Provides message passing facility;
- Client-server port-interface – Provides function invocation.

As the RTE implements all the communication infrastructure facilities of the AUTOSAR architecture, and it is conceptually placed between the SWC and the BSW layer, it can be classified as the “heart” of the architecture. It can be logically divided into the following subparts:

- Components communication;
- SWCs scheduling.

The Basic Software Scheduler must also be taken into account. It performs the scheduling activities for the BSWMs. It cannot be clearly separated from the RTE as they are strongly linked, since the same OS Task may be used to schedule SWCs and BSWMs.

Normally SWCs and BSWMs are designed to run within AUTOSAR architecture, depending on other BSWMs for normal functioning. An infotainment board can be classified as a multimedia, telematics and human machine interface (MM/T/HMI) ECU. To integrate AUTOSAR applications on such boards we need to guarantee that all the necessary SWCs and BSWMs are implemented, or, at least, emulated.

2.3. STATE OF THE ART AND XLUNA

The first approach proposed in [RD-1] is a minimalist one. It intends the implementation of an RTE emulator that performs the communication functionalities between the SWCs and a general purpose OS, providing the required interface and calls for the SWC runnables. The general-purpose OS implements all the BSWMs needed functionalities. The following figure illustrates the latter:

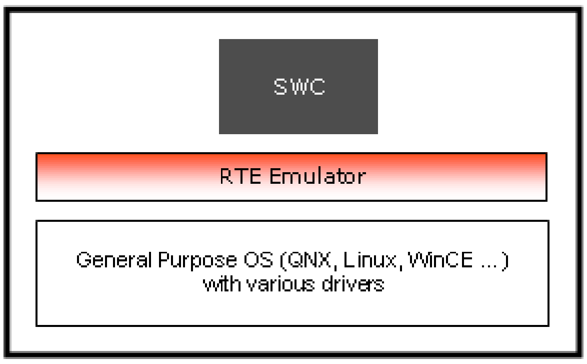


Figure 1 – RTE Emulator

This implementation is not a feasible one, regarding the scope of xLuna, due to the specific nature of the RTE layer. Although being a generic layer, suitable to all possible combinations of SWC and BSW layers, each AUTOSAR application implementation involves the generation of a tailored RTE. This is due to the variation of ports required by the SWCs and BSWMs.

Regarding the previously presented idea, it can be concluded that for each set of SWCs a different RTE emulator would need to be implemented. This would only be feasible if a RTE emulator generator were to be used, as the required implementation effort would be too great.

Another downside is that the SWCs may require BSWMs whose functionalities are not implemented by the general-purpose OS. This would mean that the required BSWMs functionalities needed to be emulated, increasing the required implementation effort, leading to a hard to maintain system.

It is then proposed, in [RD-1], the emulation of the required BSWMs and Interfaces. Regarding xLuna scope, and to keep the implementation as simple as possible, the best approach would be to automatically generate all the AUTOSAR architecture possible components and keep implementation as minimal as possible, reducing the effort to be spent on emulating the needed BSWMs and Interfaces. It would only be necessary to implement a layer of abstraction for the BSWMs to communicate with the ECU hardware and to the generic OS.

The following figure demonstrates how xLuna automotive project intends to integrate AUTOSAR application on an infotainment board with a quadcore cpu:

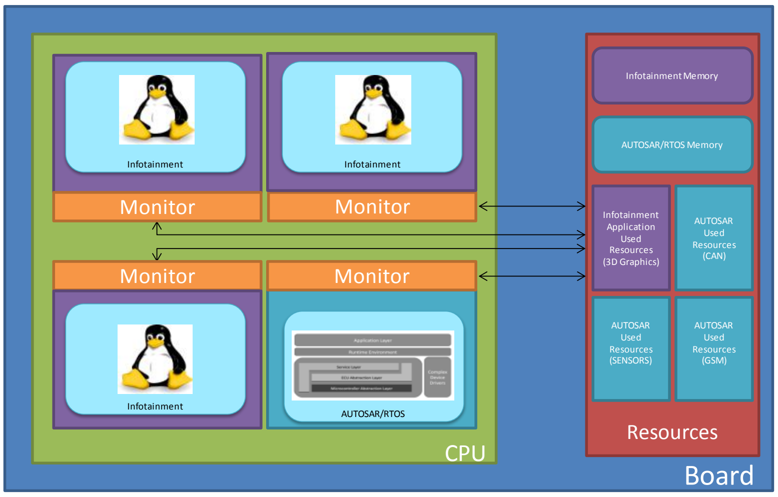

Infotainment
Infotainment
Monitor
Monitor
Monitor
Monitor
Infotainment
AUTOSAR/RTOS
CPU
Infotainment Memory
AUTOSAR/RTOS Memory
Infotainment Application Used Resources (3D Graphics)
AUTOSAR Used Resources (CAN)
AUTOSAR Used Resources (SENSORS)
AUTOSAR Used Resources (GSM)
Resources
Board

Figure 2 – AUTOSAR integration on xLuna

Regarding the scope of the xLuna project, the last proposed emulation approach of [RD-1] would be the more feasible one. It suggests the implementation of an AUTOSAR OS emulator and of a layer of driver emulators for the service modules (BSWMs).

The implementation process effort of the emulation modules is reduced as the SWCs, the RTE, and the BSWMs layers are automatically generated, remaining the AUTOSAR OS emulator and the driver emulators to be implemented.

This approach is demonstrated by the following figure:

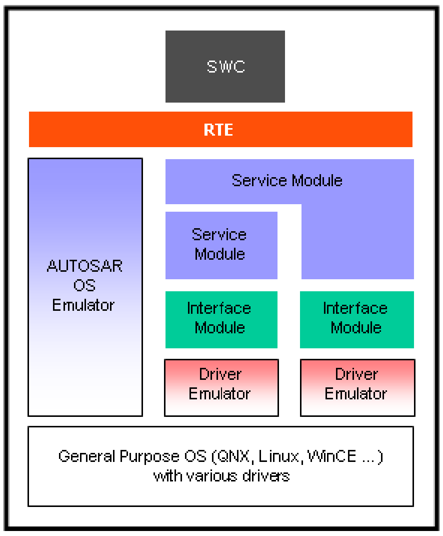


Figure 3 – Suggested emulation approach for AUTOSAR integration on xLuna

The use of a general-purpose OS on AUTOSAR is a delicate matter. It strongly depends on the AUTOSAR application itself, especially on all the necessary SWCs and BSWMs. This means there may be AUTOSAR applications for which the tailoring of the driver emulators will be complex. The latter will vary from application to application.

As for the AUTOSAR OS emulator, the implementation process can be generic. It is necessary to identify the OS services provided by the AUTOSAR OS layer to other BSWMs and to the RTE, and map its functionalities to the general-purpose OS being used. For the xLuna project, the mapping must be performed for FreeRTOS.

The RTE is responsible for SWCs and BSWMs scheduling. SWCs are composed by runnables on AUTOSAR arquitecture. The latter are mapped to tasks that execute the runnables when they are triggered. Tasks are triggered by the RTE upon occurrence of a certain event. An event can occur upon different circumstances, like data reception or transmission or due to an alarm that has expired. All the previous services need to be provided by the AUTOSAR OS, or, in this case, by the AUTOSAR OS emulator.

The AUTOSAR OS emulator will actuate as the bridge between the RTE, BSWMs and the general-purpose OS. It must guarantee that all the services provided by the AUTOSAR OS are provided to the RTE and to the BSWMs. If these services cannot be directly mapped or don’t exist, the emulator must contain the implementation of the AUTOSAR OS missing functionalities.

As the objective of this approach is to have the maximum number of AUTOSAR components automatically generated by AUTOSAR compliant code generation tools, it must be guaranteed that all the general-purpose OS provides all the necessary services to them via the AUTOSAR OS emulator. If not, the necessary services/functionalities must be implemented.

A research for identification of needed OS services is further provided. Full understanding of these is required in order to implement the AUTOSAR OS emulator.

2.4. AUTOSAR LAYERED ARCHITECTURE

Having detected the AUTOSAR OS emulator and the Driver emulator/s as the implementation targets for integration of a general-purpose OS in AUTOSAR, the next step would be to map these components on the layered architecture of an ECU AUTOSAR application.

As can be concluded from the analysis of the following figure, an AUTOSAR-based application comprises the following layers:

- Application (SWCs) – Contains the application, the sensor/actuator and the “composition” components. Communicates with the other layers’ components through the RTE (ECU and hardware independent);
- AUTOSAR Runtime Environment – Hardware independent layer that provides intra- and inter- communication facilities to the SWCs (application specific);
- Services Layer – Provides basic services to SWCs and BSWMs. It is composed by communication, services and operating system blocks (ECU and hardware specific);
- ECU Abstraction Layer – Comprises the BSW standardized interfaces and handlers that perform the abstraction of the peripherals connected to the CPU (ECU and hardware dependent);
- Microcontroller Abstraction Layer – Contains drivers that abstract direct access to microcontroller internal peripheral and memory mapped external devices. Makes all upper layers microcontroller independent (microcontroller dependent);
- Complex Device Drivers – Comprises drivers that implement special functioning and timing requirements to handle evaluate complex sensors values and control complex actuators by accessing microcontroller specific interrupts and/or peripherals directly. Specific non-standardized drivers must be placed in this layer (ECU, microcontroller and hardware dependent);

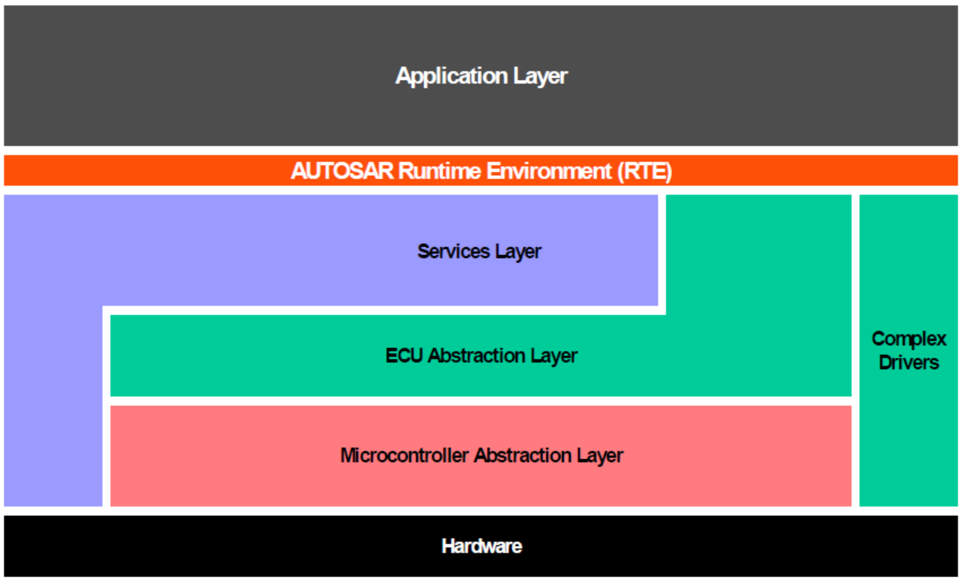


Figure 4 – AUTOSAR application layered architecture

The following figure illustrates a more modularized view of the architecture of an ECU AUTOSAR application by detailing each of the BSWMs layers in different parts, constituted by modules. There are approximately 80 BSWMs defined. Their implementation/generation varies according to the implemented SWCs and specific AUTOSAR application needed services. Due to that fact, the following figure solely presents a generic and more abstract representation of each layer.

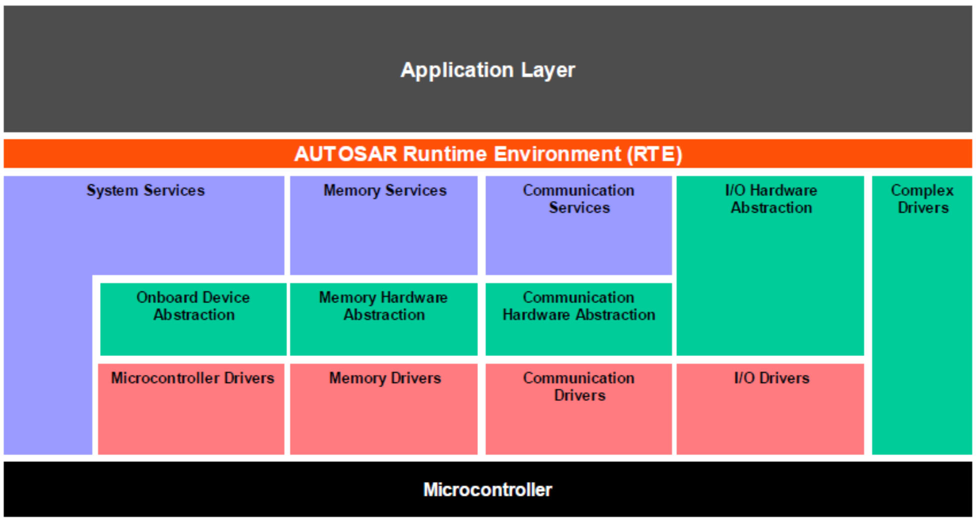


Figure 5 – AUTOSAR application layered architecture – BSW layers detailed

Through comparison of the previous figure with Figure 3 – Suggested emulation approach for AUTOSAR integration on xLunathe following mapping can be performed:

- AUTOSAR OS Emulator – Is going to be integrated as a module of the System Services layer, which is part the Services Layer;
- Driver Emulator/s – Will constitute the Microcontroller Abstraction Layer. May also have some modules as part of the Complex Device Drivers layer.

All the other modules can be configured and automatically generated by an AUTOSAR compliant code generation tool. Normally, these tools guide the user through the configuration and generation process, identifying all the necessary modules to be configured and generated, according to the initially specified SWCs.

2.5. LAYERS MODULES DETAIL

The AUTOSAR architecture sub-layers and correspondent layers on which the implementation/integration process is to occur are detailed in the following sections.

2.5.1 SERVICES LAYER

The Services layer can be divided in 3 different parts:

- Communication Services – Contains modules that implement vehicle network communication interfaces like CAN, LIX, FlexRay and MOST. It provides:
  - A uniform interface to the vehicle communication network for SWCs communication;
  - Uniform network management services;
  - A uniform interface to the vehicle network for diagnostic communication;
  - SWCs with protocol and message properties abstraction;
- Memory Services - Contains modules that implement the management of non-volatile data, like the NVRAM manager. It provides:
  - SWCs with uniform access to non-volatile data;

    - Abstraction from memory locations and properties;
    - Mechanisms for non-volatile data management;

- System Services - Contains modules and functions that implement system services to be used by modules of all layers. It provides basic services to SWCs and BSWMs and contains:
    - The real-time operating system – The AUTOSAR OS emulator must implement all the services provided by the AUTOSAR OS to the other modules, using the general-purpose OS. The emulator must be integrate in this layer in order to provide the required AUTOSAR OS services;
    - The error manager;
    - Library functions – CRC, interpolation, and others;

2.5.2 ECU ABSTRACTION LAYER

The Microcontroller Abstraction layer can be divided in 4 different sub-layers:

- I/O Hardware Abstraction – Consists on a group of modules which abstracts the ECU hardware layout, pin connections and signal level inversions, and its location, on-chip or on-board, except the sensors/actuators. The task of this group of modules is:
    - To represent I/O signals as they are connected to the hardware, e.g. current, voltage, frequency, etc.;
    - To hide hardware and layout properties from higher software layers.
- Communication Hardware Abstraction – As a hardware abstraction layer, it consists on a group of modules with the same goals of the above (to abstract from the location of the ECU hardware and its layout) but related with communication hardware, e.g. LIN, CAN, MOST, FlexRay, etc. The task of these modules is to provide equal mechanisms to access a bus channel regardless of its location.
- Memory Hardware Abstraction – In addiction of being a hardware abstraction layer, this group of modules has the task to provide equal mechanisms to access internal and external memory devices.
- Onboard Device Abstraction – Contains drivers for ECU onboard devices which cannot be seen as sensors/actuators. The task of this group of modules is to abstract from ECU specific onboard devices.

2.5.3 MICROCONTROLLER ABSTRACTION LAYER

Also the Microcontroller Abstraction layer has been subdivided into 4 different layers:

- Drivers for digital and analog inputs and outputs;
- Drivers for ECU onboard communication devices and for vehicle communication;
- Drivers for on-chip memory devices and for external memory devices.
- Drivers for the microcontroller internal peripherals and functions for direct microcontroller access;

Regarding the scope of the intended implementation, a multimedia, telematics and human machine interface (MM/T/HMI) board, more specifically an infotainment board, is to be used. Due to that fact, all the required modules of this layer need to be implemented.

2.5.4 COMPLEX DEVICE DRIVERS LAYER

The Complex Device Drivers layer is used to place specific software implementations. It is required that the interfaces of these implementations to the AUTOSAR architecture are implemented according to AUTOSAR port and interface specifications.

This layer can contain:

- Sensor and actuator control drivers – normally contains direct access to microcontroller specific interrupts and peripherals;
- Non-standardized drivers – drivers of hardware that is not currently supported by AUTOSAR architecture;

Regarding the scope of the intended implementation, there may be modules that need to be implemented as part of this layer.

3. GENERATION PROCESS

This chapter provides an overview of the AUTOSAR Methodology. Based on that, it then regards the issue of the code generation process in order to highlight the layers where the intended integration process is to occur and correspondent modules dependencies.

3.1. AUTOSAR METHODOLOGY

3.1.1 INTRODUCTION

The implementation of an AUTOSAR application must be compliant with a series of specifications that are part of the AUTOSAR methodology. The latter is neither a complete process description nor a business model. It consists of a workflow that defines different activities and the correspondent required inputs and implemented outputs.

AUTOSAR Methodology is described using a subset of the Software Process Engineering Meta-Model standard. The following definitions describe the elements of this standard:

- Work-Product – Information or physical entity produced/used by an activity. In AUTOSAR it can be a:
    - XML document;
    - c-Document (source code file) – “*.c”;
    - obj-Document (object file);
    - h-Document (header file) – “*.h”;
- Activity – Piece of work composed by tasks, operations and actions performed by a person or group of persons;
- Guidance – Elements associated with activities that represent additional information or tools to be used to perform the correspondent activity.

The following image contains the graphical representation of the previously described elements:

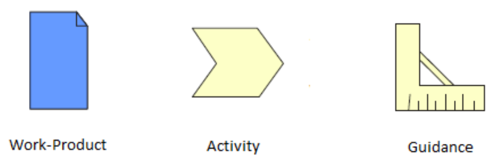


Figure 6 – SPEM Elements graphical representation

Graphical representation is also defined for:

- The flow work-products and activities - represented by a straight line with an arrowhead;
- The dependencies between different elements - represented by a dotted line with an arrowhead;

The following figure provides an overview of the methodology used to implement AUTOSAR compliant applications:

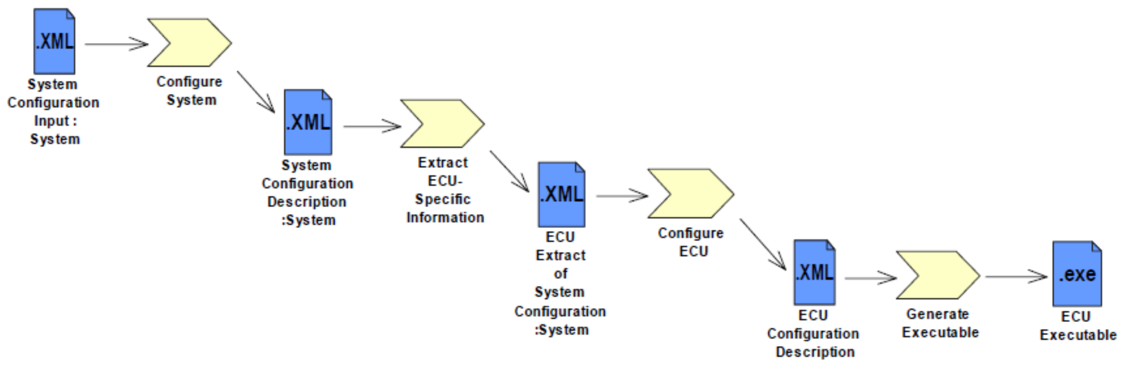

.XML
System Configuration Input : System
Configure System
.XML
System Configuration Description :System
Extract ECU-Specific Information
.XML
ECU Extract of System Configuration :System
Configure ECU
.XML
ECU Configuration Description
Generate Executable
.exe
ECU Executable

Figure 7 – AUTOSAR Methodology Overview

This document only provides a very generic overview on the methodology, as it is much more complex. For further detail on methodology refer to chapter 3 of [RD-3].

3.1.2 STEPS DESCRIPTION

The first step on implementing an AUTOSAR application is to provide information at system level by defining the System Configuration Input (in XML). This requires that all system level decisions (regarding software, hardware and overall system constraints). The definition of the System Configuration Input requires the filling and editing of the proper AUTOSAR templates. This information will regard:

- Software Components – description of the SWCs API (ports, data type, and others);
- ECU Resources – Specification of CPU, memory, peripherals, sensors and actuators per ECU;
- System Constraints - Specification of constraints regarding bus signals, topology and mapping of SWCs;

The configuration of the System Configuration Input can be performed using an AUTOSAR compliant proper tool. AUTOSAR methodology allows for a high degree of reuse at this level.

In the Configure System activity the mapping of the SWCs to the correspondent ECUs is performed, regarding the resources and timing requirements. This activity creates as an output the System Configuration Description. It is a system-level activity that must be performed per system. The remaining activities must be performed for each ECU present in the system.

The next activity represented in Figure 7 – AUTOSAR Methodology Overview regards the extraction of the ECU needed information from the System Configuration Description, and is called Extract ECU-Specific Information. The output of this activity is the ECU Extract of System Configuration.

The latter serves as input to the next activity to be performed, the Configure ECU activity. In this activity all the implementation required information is specified, like:

- Task Scheduling;
- Required BSWMs identification;
- BSWMs configuration;
- Mapping of Runnables to Tasks, and others;

All the ECU defined information is gathered and the output to this activity is the ECU Configuration Description. The ECU Configuration Description is organized in individual structures containing the configuration values for the different BSWMs. Modules commonly used values must be synchronised between the different BSWMs configurations. Standardized synchronisation functionality is not provided by AUTOSAR as it is assumed that during generation of the BSWMs input information provided to a module is synchronised with information provided by the modules it relies on. ECU-specific runnable software can be generated from this work-product.

The last activity is the Build Executable activity. It performs the generation of an ECU-specific executable based on the information defined on the ECU Configuration Description. This step involves:

- Generation of RTE and BSW code;
- Compilation of code – Generated code or available SWCs source-code;
- Linking of everything and creation of the executable;

There are additional steps to be performed in parallel to the previously described ones in order to correctly integrate the SWCs into the system:

- Generation of the SWCs API;
- Implementation of the SWCs functionalities;

3.1.3 AUTOSAR TOOLING

There is a variety of tools available that generate code and allow the configuration of AUTOSAR compliant applications. The majority of these tools follow a standard procedure for code generation, based on AUTOSAR methodology. Normally this procedure consists on fewer steps than the ones previously described. It regards the following scopes:

- AUTOSAR Application – Firstly the SWCs are configured - ports and interfaces specification, runnables configuration, required sensors/actuators definition. The outputs of this stage, per SWC, are a XML document describing the SWC specification and a source code and header file specifying the API of the SWC. No functionality is implemented, solely the structure on which it will be further built on. Implementation of SWCs functionality can be performed at any stage before executable generation by editing the SWC correspondent source code files. AUTOSAR compliant tools normally allow configuration of the SWCs, edition of source code and automatic generation of code;
- System – System level tasks are performed after the SWCs are defined. The previously defined SWCs are mapped into ECUs. The communication network facilities are defined and the required hardware is identified and its topology defined. Some tools allow communication description files to be imported for reuse purposes in order to ease the communication network configuration task. The outputs of this tasks are only XML documents: an AUTOSAR System Description containing the mapping of the SWCs into ECUs, the ECU-Extract containing the specification of each ECU required modules and, for some tools, the BSWMs configuration files can also be generated at this stage;
- ECU – Using system-level tasks outputs as inputs, ECU configuration tasks can be performed. The ECU required BSWMs are identified and properly configured. The RTE layer must be configured. If an ECU is to use any complex device drivers that are not included in AUTOSAR architecture they must be integrated at this point. The Operating System and the Microcontroller Abstraction Layer modules must also be configured. When the configuration tasks are finished, the AUTOSAR BSW and RTE code is automatically generated. The generated code can be further edited. The outputs of the ECU configuration tasks are the files containing BSW and RTE source code;

Basically, AUTOSAR compliant tools allow SWCs, BSWMs and RTE definition, configuration, automatic code generation and code edition. When the AUTOSAR application is completed, and the code is ready, it can be compiled and linked into executables by the tools, in order to run of each system ECU.

The AUTOSAR application generation process is illustrated by the following figure:

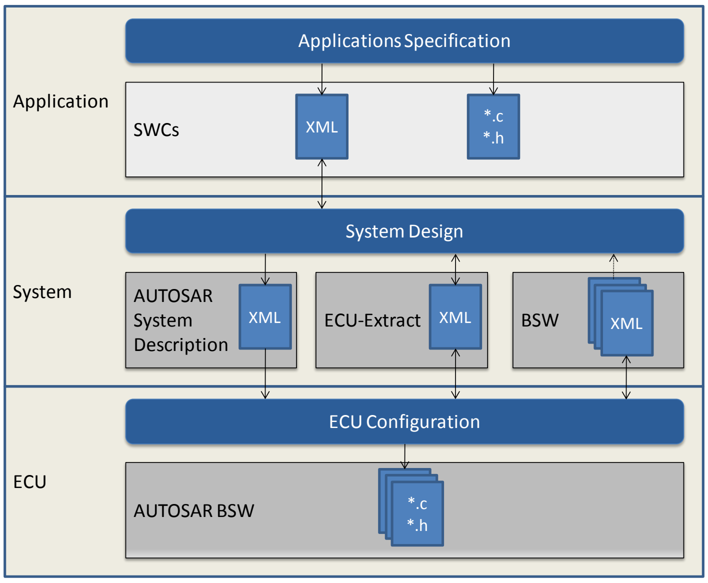


Figure 8 – AUTOSAR Tooling Overview

3.2. RTE GENERATOR

The RTE generator consists on a tool, or set of tools, that perform all the necessary tasks to generate RTE and BSW scheduler API and source code. Due to that fact, it is part of the ECU configuration tasks of AUTOSAR methodology. For detail regarding the RTE Generation Process refer to chapter 3 of [AD-3].

As RTE interacts with the AUTOSAR OS module, the dependencies that exist regarding that module are to be identified. In chapter 3, section 3.3 of [AD-3] it is referred that the “RTE Configuration Editor” can use the information contained in the “Specification of Timing Extensions” in order to create and check the configuration of the RTE Event to an OS Task mapping.

In section 3.4.1 of [AD-3], regarding BSW Scheduler Generation Phase, it is referred, by requirement [SWS_Rte_07569], that even if mapped RTEEvents are not permitted, and runnable calls are not generated into OS task bodies, all the BSW scheduler configuration related OS task bodies are generated.

It is referred in requirement [SWS_Rte_07085], in section 3.4.3.1 of [AD-3] that the RTE may depend on elements of other BSWMs. It that is the case, the full qualified path name of such elements must be provided to the RTE Generator. An example is provided regarding dependence on “TaskType”, an “ImplementationDataType” element of the OS module.

In section 3.7 of [AD-3] is referred that the RTE Generator may update/change configuration information of other BSWMs if the external configuration switch strictConfigurationCheck is set to false. An example is provided referring that in case an OSTask referenced by the RTE configuration is not configured in the OS module configuration the RTE Generator may change the OS module configuration to contain that OSTask.

4. OS INTERACTION

This chapter describes the interaction of the AUTOSAR OS module with other BSWMs.

4.1. API

This section resumes the API provided by the AUTOSAR OS module. For further detail please refer to chapter 8 of [AD-2].

4.1.1 CONSTANTS

The OS module shall provide the following constants to the RTE and to other BSWMs. For further detail refer to section 8.1 of [AD-2].

Name	Type	Description
AppModeType	Enumeration	AppMode of the core shal be inherited from another core.
TotalNumberOfCores	Scalar	The total number of cores

TABLE 5: OS API Constants

4.1.2 MACROS

The OS module shall provide the following macros to the RTE and to other BSWMs. In the OS module all the API macros are function-like macros. For further detail refer to section 8.2 of [AD-2].

Name	Arguments Type	Return Value	Description
OSMEMORY_IS_READABLE()	AccessType	Not equal to zero if memory is readable.	The return value of the service Check[Task|ISR]Memory Access() is used as argument for this macro.
OSMEMORY_IS_WRITEABLE()	AccessType	Not equal to zero if memory is writable.	The return value of the service Check[Task|ISR]Memory Access() is used as argument for this macro.
OSMEMORY_IS_EXECUTABLE()	AccessType	Not equal to zero if memory is executable.	The return value of the service Check[Task|ISR]Memory Access() is used as argument for this macro.

Name	Arguments Type	Return Value	Description
OSMEMORY_IS_STACKSPACE()	AccessType	Not equal to zero if memory is stack space.	The return value of the service Check[Task|ISR]Memory Access() is used as argument for this macro.

TABLE 6: OS API Macros

4.1.3 TYPE DEFINITIONS

The OS module shall provide the following types to the RTE and to other BSWMs. For further detail refer to section 8.3 of [AD-2].

Name	Type	Constants	Description
ApplicationType	Scalar	INVALID_OSAPPLICATION	This data type identifies the OS-Application.
ApplicationStateType	Scalar	APPLICATION_ACCESSIBLE APPLICATION_RESTARTING APPLICATION_TERMINATED	This data type identifies the state of an OS-Application.
ApplicationStateRefType	Pointer	None	This data type points to location where a ApplicationStateType can be stored.
TrustedFunctionIndexType	Scalar	None	This data type identifies a trusted function.
TrustedFunctionParameterRef Type	Pointer	None	This data type points to a structure which holds the arguments for a call to a trusted function.
AccessType	Integral	None	This type holds information how a specific memory region can be accessed.
ObjectAccessType	Scalar	ACCESS NO_ACCESS	This data type identifies if an OS-Application has access to an object.
ObjectTypeType	Scalar	OBJECT_TASK	This data type identifies an

Name	Type	Constants	Description
		OBJECT_ISR OBJECT_ALARM OBJECT_RESOURCE OBJECT_COUNTER OBJECT_SCHEDULETABLE	object.
MemoryStartAddressType	Pointer	None	This data type is a pointer which is able to point to any location in the MCU address space.
MemorySizeType	Scalar	None	This data type holds the size (in bytes) of a memory region.
ISRType	Scalar	INVALID_ISR	This data type identifies an interrupt service routine (ISR).
ScheduleTableType	Scalar	None	This data type identifies a schedule table.
ScheduleTableStatusType	Scalar	SCHEDULETABLE_STOPPED SCHEDULETABLE_NEXT SCHEDULETABLE_WAITING SCHEDULETABLE_RUNNING SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS	This type describes the status of a schedule.
ScheduleTableStatusRefType	Pointer	None	This data type points to a variable of the data type ScheduleTableStatusType.
CounterType	Scalar	None	This data type identifies a counter.
ProtectionReturnType	Scalar	PRO_IGNORE PRO_TERMINATETASKISR	This data type identifies a value which controls further actions of the OS on return from the

Name	Type	Constants	Description
		PRO_TERMINATEAPPL PRO_TERMINATEAPPL_RESTAR T PRO_SHUTDOWN	protection hook.
RestartType	Scalar	RESTART NO_RESTART	This data type defines the use of a Restart Task after terminating an OS-Application.
PhysicalTimeType	Scalar	None	This data type is used for values returned by the conversion macro
CoreIdType	Scalar	OS_CORE_ID_MASTER	CoreIDType is a scalar that allows identifying a single core. The CoreIDType shall represent the logical CoreID.
SpinlockIdType	Scalar	INVALID_SPINLOCK	SpinlockIdType identifies a spinlock instance and is used by the API functions: GetSpinlock, ReleaseSpinlock and TryToGetSpinlock.
TryToGetSpinlockType	Enumeration	TRYTOGETSPINLOCK_SUCCESS TRYTOGETSPINLOCK_NOSUCC ESS	The TryToGetSpinlockType indicates if the spinlock has been occupied or not.
IdleModeType	Scalar	IDLE_NO_HALT	This data type identifies the idle mode behavior.

TABLE 7: OS API Type Definitions

4.1.4 FUNCTION DEFINITIONS

The OS module shall provide the following functions to the RTE and to other BSWMs. The use of such services may be restricted and depends on the context they are called from. The table only contains an overview of the functions. The functionality specification of each function is detail in terms of software specification requirements. For further detail refer to section 8.4 of [AD-2].

Name	Arguments Type	Return Type	Description
GetApplicationID	Void	ApplicationType	This service determines the currently running OS-Application (a unique identifier has to be allocated to each application).
GetISRID	Void	ISRType	This service returns the identifier of the currently executing ISR.
CallTrustedFunction	TrustedFunctionIndexType TrustedFunctionParameterRefType	StatusType	A (trusted or non-trusted) OS-Application uses this service to call a trusted function.
CheckISRMemoryAccess	ISRType MemoryStartAddressType MemorySizeType	AccessType	This service checks if a memory region is write/read/execute accessible and also returns information if the memory region is part of the stack space.
CheckTaskMemoryAccess	TaskType MemoryStartAddressType MemorySizeType	AccessType	This service checks if a memory region is write/read/execute accessible and also returns information if the memory region is part of the stack space.
CheckObjectAccess	ApplicationType ObjectTypeType Void (object to be examined)	ObjectAccessType	This service determines if the OS-Applications, given by ApplID, is allowed to use the IDs of a Task, Resource, Counter, Alarm or Schedule Table in API calls.
CheckObjectOwnership	ObjectType Void (object to be examined)	ApplicationType	This service determines to which OS-Application a given Task, ISR, Counter, Alarm or Schedule Table belongs.
StartScheduleTableRel	ScheduleTableType	StatusType	This service starts the processing of a schedule table at "Offset" relative to

Name	Arguments Type	Return Type	Description
	TickType		the "Now" value on the underlying counter.
StartScheduleTableAbs	ScheduleTableType TickType	StatusType	This service starts the processing of a schedule table at an absolute value "Start" on the underlying counter.
StopScheduleTable	ScheduleTableType	StatusType	This service cancels the processing of a schedule table immediately at any point while the schedule table is running.
NextScheduleTable	ScheduleTableType ScheduleTableType	StatusType	This service switches the processing from one schedule table to another schedule table.
StartScheduleTableSynchron	ScheduleTableType	StatusType	This service starts an explicitly synchronized schedule table synchronously.
SyncScheduleTable	ScheduleTableType TickType	StatusType	This service provides the schedule table with a synchronization count and start synchronization.
SetScheduleTableAsync	ScheduleTableType	StatusType	This service stops synchronization of a schedule table.
GetScheduleTableStatus	ScheduleTableType ScheduleTableStatusRefType	StatusType	This service queries the state of a schedule table (also with respect to synchronization).
IncrementCounter	CounterType	StatusType	This service increments a software counter.
GetCounterValue	CounterType TickRefType	StatusType	This service reads the current count value of a counter (returning either the hardware timer ticks if counter is driven by hardware or the software

Name	Arguments Type	Return Type	Description
			ticks when user drives counter).
GetElapsedValue	CounterType TickRefType TickRefType	StatusType	This service gets the number of ticks between the current tick value and a previously read tick value.
TerminateApplication	ApplicationType RestartType	StatusType	This service terminates the OS-Application to which the calling Task/Category 2 ISR/application specific error hook belongs.
AllowAccess	Void	StatusType	This service sets the own state of an OS-Application from APPLICATION_RESTARTI NG to APPLICATION_ACCESSIB LE.
GetApplicationState	ApplicationType ApplicationStateRefType	StatusType	This service returns the current state of an OS-Application.
GetNumberOfActivatedCo res	Void	Uint32	The function returns the number of cores activated by the StartCore function. This function might be a macro.
GetCoreID	Void	CoreIdType	The function returns a unique core identifier.
StartCore	CoreIdType StatusType Pointer (out)	Void	It is not supported to call this function after StartOS(). The function starts the core specified by the parameter CoreID. The OUT parameter allows the caller to check whether the operation was successful or not. If a core is started by means of this function StartOS shall be called on the core.

Name	Arguments Type	Return Type	Description
StartNonAutosarCore	CoreIdType StatusType Pointer (out)	Void	The function starts the core specified by the parameter CoreID. It is allowed to call this function after StartOS(). The OUT parameter allows the caller to check whether the operation was successful or not. It is not allowed to call StartOS on cores activated by StartNonAutosarCore. Otherwise the behaviour is unspecified.
GetSpinlock	SpinlockIdType	StatusType	GetSpinlock tries to occupy a spin-lock variable. If the function returns, either the lock is successfully taken or an error has occurred. The spinlock mechanism is an active polling mechanism. The function does not cause a de-scheduling.
ReleaseSpinlock	SpinlockIdType	StatusType	ReleaseSpinlock releases a spinlock variable that was occupied before. Before terminating a TASK all spinlock variables that have been occupied with GetSpinlock() shall be released. Before calling WaitEVENT all Spinlocks shall be released.
TryToGetSpinlock	SpinlockIdType TryToGetSpinlockType Pointer (out)	StatusType	TryToGetSpinlock has the same functionality as GetSpinlock with the difference that if the spinlock is already occupied by a TASK on a different core the function sets the OUT parameter "Success" and returns with E_OK.
ShutdownAllCores	StatusType	Void	After this service the OS on all AUTOSAR cores is shut down. Allowed at TASK

Name	Arguments Type	Return Type	Description
			level and ISR level and also internally by the OS. The function will never return. The function will force other cores into a shutdown.
ControlIdle	CoreIdType IdleModeType	StatusType	This API allows the caller to select the idle mode action which is performed during idle time of the OS (e.g. if no Task/ISR is active). It can be used to implement energy savings. The real idle modes are hardware dependent and not standardized. The default idle mode on each core is IDLE_NO_HALT.

TABLE 8: OS API Functions

4.1.5 IOC

The Inter OS-Application Communicator is responsible for the communication between OS-Applications, more specifically, for the communication crossing core or memory protection boundaries. Its internal functionality is closely connected to the Operating System. Memory protection boundaries are a characteristic of OS-Applications and special communication mechanisms are needed to cross them.

AUTOSAR software modules from BSW and SWC most belong to an OS-Application. It is expected that BSWMs are trusted code, but they may be defined as more than one OS-Application. IOC provides communication services between OS-Applications, in particular over core boundaries in Multi-Core systems.

For further detail on IOC please refer to section 7.10 of [AD-2].

4.1.5.1 Constants

The OS module shall provide the following IOC related constants to the RTE and to other BSWMs. For further detail refer to section 8.5.3 of [AD-2].

Name	Type	Description
IOC_E_OK	Std_ReturnType	No error occurred
IOC_E_NOK	Std_ReturnType	The total number of cores
IOC_E_LIMIT	Std_ReturnType	In case of “event” (queued) semantic, the internal buffer within the IOC communication service is full (Case: Receiver slower than

Name	Type	Description
		sender). This error produces additionally an Overlayed Error on the receiver side at the next data reception.
IOC_E_LOST_DATA	Std_ReturnType	In case of “event” (queued) semantic, this Overlayed Error indicates that the IOC service refuses an IocSend request due to internal buffer overflow.
IOC_E_NO_DATA	Std_ReturnType	In case of “event” (queued) semantic, no data is available for reception.

TABLE 9: OS API IOC Constants

4.1.5.2 Function Definitions

The OS module shall provide the following IOC related functions to the RTE and to other BSWMs. The table only contains an overview of the functions. The functionality specification of each function is detail in terms of software specification requirements. For further detail refer to section 8.5.4 of [AD-2].

Name	Arguments Type	Return Type	Description
IocSend_<IocId>[_<SenderId>]	<Data>	Std_ReturnType	Performs an "explicit" sender-receiver transmission of data elements with "event" semantic for a unidirectional 1:1 or N:1 communication between OS-Applications located on the same or on different cores. <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication. <SenderId> is used only in N:1 communication. Together with <IocId>, it uniquely identifies the sender. It is separated from <IocId> with an underscore. In case of 1:1 communication, it shall be omitted.
IocWrite_<IocId>[_<SenderId>]	<Data>	Std_ReturnType	Performs an "explicit" sender-receiver transmission of data elements with "data" semantic for a unidirectional 1:1 or N:1 communication between OS-Applications located on the same or

Name	Arguments Type	Return Type	Description
			on different cores. <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication. <SenderId> is used only in N:1 communication. Together with <IocId>, it uniquely identifies the sender. It is separated from <IocId> with an underscore. In case of 1:1 communication, it shall be omitted.
IocSendGroup_<IocId>	<Data1> <Data2> ...	Std_ReturnType	Performs an "explicit" sender-receiver transmission of data elements with "event" semantic for a unidirectional 1:1 communication between OS-Applications located on the same or on different cores. This API involves a group of data elements which values are specified in parameter. <IocId> is a unique identifier that references a unidirectional 1:1 communication involving many data elements.
IocWriteGroup_<IocId>	<Data1> <Data2> ...	Std_ReturnType	Performs an "explicit" sender-receiver transmission of data elements with "data" semantic for a unidirectional 1:1 communication between OS-Applications located on the same or on different cores. This API involves a group of data elements which values are specified in parameter. <IocId> is a unique identifier that references a unidirectional 1:1 communication involving many data elements.

TABLE 10: OS API IOC Functions

4.1.5.3 IOC Expected Interfaces

The IOC of the OS module requires the following interfaces from other BSWMs. The table only contains an overview of the optional interfaces since there are no mandatory interfaces. The functionality specification of each function is detail in terms of software specification requirements. For further detail refer to section 8.6 of [AD-2].

Name	Arguments Type	Return Type	Description
<ReceiverPullCB>	Void	Void	This callback function can be configured for the receiver of a communication. If configured, IOC calls this callback on the receiving core for each data reception. <ReceiverPullCB> is the callback function name configured by the receiver in the OsIocReceiverPullCB attribute to be called on data reception."

TABLE 11: OS API IOC Expected Interfaces

4.1.6 HOOK FUNCTIONS

Hook functions are called by the operating system if specific conditions are met. They are provided by the user. The table only contains an overview of the function. The functionality specification of each function is detail in terms of software specification requirements. For further detail refer to section 8.7 of [AD-2].

Name	Arguments Type	Return Type	Description
ProtectionHook	StatusType	ProtectionReturnType	The protection hook is always called if a serious error occurs. E.g. exceeding the worst case execution time or violating against the memory protection.
StartupHook_<App>	Void	Void	The application specific startup hook is called during the start of the OS (after the user has started the OS via StartOS()).
ErrorHook_<App>	StatusType	Void	The application specific error hook is called whenever a Task or Category 2 ISR which belongs to the OS-Application causes an error.
ShutdownHook_<App>	StatusType	Void	The application specific error hook is called whenever a Task or Category 2 ISR which belongs to the OS-Application causes an error.

TABLE 12: OS API Hook Functions

4.1.7 SERVICE INTERFACES

The OS module defines the following Client-Server Service Interfaces.

4.1.7.1 OS Service

The table only contains an overview of the operation. For further detail refer to section 8.8.1.1 of [AD-2].

Name	Arguments Type	Return Type	Description
GetCounterValue	Void	TimeInMicrosecondsType	This service reads the current count value of a counter (returning either the hardware timer ticks if counter is driven by hardware or the software ticks when user drives counter).
GetElapsedValue	TimeInMicrosecondsType	TimeInMicrosecondsType	This service gets the number of ticks between the current tick value and a previously read tick value.

TABLE 13: OS API Services Operations

4.1.7.2 Implementation Data Types

The table only contains an overview of the data type. For further detail refer to section 8.8.1.2 of [AD-2].

Name	Type	Constants	Description
TimeInMicrosecondsType	Uint64	None	None

TABLE 14: OS API Services Data Types

4.2. OS/RTE INTERACTION

This section highlights the relations of the AUTOSAR OS module with the RTE. It starts with the analysis of chapter 4 of [AD-3].

4.2.1 ARCHITECTURAL CONCEPTS

In 4.1.3.3 the concept of Runnable is introduced. A Runnable, also referred as RunnableEntity, is the smallest code fragment of AUTOSAR SWCs and BSWMs that implement AUTOSAR Interfaces. In most cases a SWC is composed by several Runnables. Each Runnable is executed in the context of an OS Task and triggered by RTEEvents. These definitions are detailed by [SWS_Rte_02202], [SWS_Rte_02203], [SWS_Rte_02204], [SWS_Rte_07347] and [SWS_Rte_02207].

In sections 4.1.4.3 and 4.1.4.4 examples are provided demonstrating how single or multiple instantiation of a Runnable may be achieved in the context of an OS Task.

Section 4.1.8.3 refers that the RTE Generator shall reject configurations where a BswEvent that can trigger a Schedulable Entity is not mapped to an OS Task - [SWS_Rte_07516]. It is referred that the execution order of Schedulable Entities within one OS Task shall be respected - [SWS_Rte_07517]. RTE shall also support the execution sequences of Runnable Entities within the same OS Task in an arbitrarily configurable order - [SWS_Rte_07518].

4.2.2 RTE AND BSW IMPLEMENTATION

Section 4.2 refers o the specific implementation aspects of the AUTOSAR RTE and Basic Software Scheduler. It performs the mapping of Runnable and other logical conceptual entities to technical architectures, like the AUTOSAR OS. Aspects regarding the interaction of RTE with the BSW Scheduler with OS and COM modules also referred in this section.

In section 4.2.1 it is referred that due to the standardized interface, the AUTOSAR OS and COM modules and respective services are invisible to the elements of the SWC layer. It is also referred that the OS and COM modules are used by the RTE in order to achieve a functionality requested by the SWCs. Specifically, AUTOSAR OS is used by the RTE to route a signal over core and partition boundaries and to properly schedule the single Runnables, so that the RTE generates task-bodies which contain calls to appropriate Runnables. RTE shall also use the available means to convert interrupts to notifications in a task context, or to guarantee data consistency.

Each RTE implementation uses a specific implementation of AUTOSAR OS module. The BSW Scheduler offers scheduling services to integrate BSWMs to modules of all layers. It uses the AUTOSAR OS in order to provide such services. [SWS_Rte_02250], [SWS_Rte_07519], [SWS_Rte_02251] and [SWS_Rte_02254] refer to the usage of AUTOSAR OS module by the RTE and BSW Scheduler.

Section 4.2.2 describes the interaction between RTE, BSW Scheduler and AUTOSAR OS module, realized over the standardized interface of the OS module (API). The OS is statically configured by the ECU configuration. The RTE Generator may be allowed to create tasks and other OS objects required by the runtime environment. The RTE Generator is responsible for generating the OS task bodies that contain the calls to the Runnable Entities and BSW Schedulable Entities. The latter are themselves OS independent and cannot use OS service calls. Such calls must be encapsulated via the standardized RTE API, respectively, the BSW Scheduler API.

The following OS objects and services are used by the RTE or by the BSW Scheduler:

- Tasks;
- OS Applications;
- Events;
- Resources;
- Interrupt Processing;
- Alarms;
- Schedule Tables;

For further detail of the previously mentioned objects refer to section 4.2.2.1 of [AD-3].

In Section 4.2.2.2 it is referred that the configure of the BSW Scheduler allows the mapping of Bsw Schedulable Entities to both basic and extended tasks.

Section 4.2.2.3 regards the runnable entities and the respective task-mapping aspects. Runnable entities are referred as the schedulable parts of the SWCs, with the exception of reentrant server runnables, which are invoked via direct function calls. Runnable entities must be mapped to tasks. The mapping must be specified in the ECU Configuration Description, in order to serve as an input to the RTE Generator.

Runnable entities are normally activated by the occurrence of an RTEEvent. For further details refer to section 4.2.2.4. It is referred in this section that the OS is used along with the RTE to implement Runnable entities waitpoints.

Section 4.2.2.6 refers to the mapping of runnable entities and BSW schedulable entities to tasks. It is referred that the RTE is responsible for constructing the task bodies. RTE Configurator uses ECU Configuration Description information, like the parts regarding the mapping of runnable entities to OS tasks. The RTE Configurator expects RTE and BSW Scheduler used OS objects to be available (Tasks, Events, Alarms, and others).

Scenarios regarding the mapping of Runnable Entities to tasks are presented in section 4.2.2.6.1. For each scenario, the usage of basic and extended tasks for runnable entities mapping is discussed.

Section 4.2.2.7 regards the issue on monitoring a runnable execution time. Examples are provided for possible solutions:

- Without using OS Events;
- Using OS Events;
- Monitoring of groups of runnables.

In section 4.2.2.9 it is referred how timing event activated runnable entities can be synchronized. It is referred that the timing events are implemented using OS alarms and OS schedule table expiry points. The latter proper functioning requires OS schedule tables and OS counter. Several possibilities are presented regarding the usage of these OS objects, regarding timing event activated runnable entities synchronization. For further detail on the subject refer to [SWS_Rte_07804] and [SWS_Rte_07805].

Section 4.2.2.10 refers to background event activated runnable entities and BSW schedulable entities. A background event is identified as a RTE/BSW event that performs background activities in runnable or BSW schedulable entities. It is similar to a timing event, but it has no fixed time period and normally is activated with the lowest priority. Two possible ways of performing background activation are provided, both involving the usage of OS tasks.

Init event activated runnable entities are described in section 4.2.2.11. An init event activates a runnable entity for initialization purpose in case of RTE or partition restart. The init event can be mapped to an OS task. For further detail refer to [SWS_Rte_06761] and [SWS_Rte_06762].

In section 4.2.3.2 it is mentioned that the RTE and the BSW Scheduler shall respect the activation offset of runnable entities and BSW schedulable entities, respectively, which each one is mapped within one OS task. These are referred in the requirements [SWS_Rte_07000] and [SWS_Rte_07520].

In section 4.2.3.3 of [AD-3], the requirement [SWS_Rte_08054] refers that the RTE shall collect the activating RTE events if the “provide activating” RTE event feature is enabled. In the context of the OS task, the executable entity is mapped to in an activation vector at the corresponding bit position as defined in [SWS_Rte_08053] requirement.

Some mechanisms to guarantee data consistency in RTE are related directly to the OS. In section 4.2.5.4 these mechanisms are discussed with detail. One of them is the usage of OS resources that is not so effective blocking higher priority tasks than interrupt blocking strategy but might consume more resources. The choice of this mechanism may vary depending on the number of cores and/or the number of the available resources. Another mechanism using OS features is the task blocking strategy that consists on assigning same priorities to affected tasks, assigning same internal OS resource to affected tasks or configuring the tasks to be non-preemptive. To use some of these mechanisms parameter RteExclusiveAreaImplMechanism must be configured with one of the values present in section 4.2.5.5.1 of [AD-3].

4.2.3 COMMUNICATION PARADIGMS

In section 4.3.1.5.1 of [AD-3], and regarding the requirement [SWS_Rte_07062], it is referred that it is only supported to group Implicit Read Accesses or Implicit Write Accesses of runnable entities executed in the same OS Task, meaning that a Coherent Implicit Data Accesses results in a task local buffer. In the same section of [AD-3], it is also said that it is ensured that the data produced by one runnable entity is propagated before runnable entities assigned to other OS tasks are activated due to task scheduling caused by the explicit schedule point. This is according to the requirements [SWS_Rte_07020] and [SWS_Rte_07021]. In addition, it is also referred that Consistency Needs are used for the correct configuration of the RTE and OS with respect to preemption, RteEventToTaskMapping and RteImplicitCommunication. More information about the handling of Consistency Needs can be found in the constraint [constr_9001].

The IOC channels configurations must be provided as inputs for the RTE Generator if strictConfigurationCheck is to true, and is provided by the RTE Generator or RTE Configuration Editor if strictConfigurationCheck is set to false. The IOC channels are configured in the OS Configuration. This is referred in requirement [SWS_Rte_02737] at section 4.3.4.1 of [AD-3].

In section 4.3.4.4 of [AD-3] is referred a summary of OS features that can be used for signalling and control flow across partition boundaries. Those features are:

- activation of tasks;
- start and stop of schedule tables;
- event signaling;
- alarms;
- spin locks.

The call-trusted-function mechanism of AUTOSAR OS can be used to implement a function call from an untrusted to a trusted partition. The trusted partition of AUTOSAR OS is a partition that has full access to the OS objects of other partitions on the same core, and it contains BSWMs. The trusted functions configurations shall be provided as inputs for the RTE Generator if strictConfigurationCheck is to true, and is provided by the RTE Generator or RTE Configuration Editor if strictConfigurationCheck is set to false. The trusted functions are configured in the OS Configuration. The requirement [SWS_Rte_07606] refers that the OS ensures that the partition of the caller is not terminated or restarted when a trusted function is executed, and the requirement [SWS_Rte_02761] refers that the RTE must use OS call trusted function from a task of an untrusted partition to BSW or to the SWC of another partition if this is not a pure function and only in a partitioned system that supports stop or restart of partitions. These topics are deeply detailed in section 4.3.4.5 of [AD-3].

4.2.4 TRIGGERS

In Trigger Sink section (4.5.1.2) of [AD-3], it is referred by the requirement [SWS_Rte_07213] that the RTE Generator shall support invocation of triggered executable entities via OS tasks.

In section 4.5.1.3 of [AD-3], it is referred by the requirement [SW_Rte_07544] that it is possible to the BSWM trigger source directly takes care about the activation of the OS task to which the ExternalTriggerOccurredEvents of the triggered executable entities are mapped. Then the name of the used OS tasks is annotated by the task attribute. In the requirement [SWS_Rte_07545] it is referred that several OS tasks may be used to implement a trigger and, if the BswTriggerDirectImplementation is defined for a released trigger which swImplPolicy attribute is set to queued, it is part of the trigger source to implement the queue or to use the means of the OS to queue the number of raised triggers.

The queuing of triggers is important to ensure that the number of executions of triggered executable entities is the same of the number of released triggers. To implement those queues of triggers the means of the OS which already can queue the activation requests for a OS task can be used. For further detail on the subject refer to the section 4.5.5 of [AD-3].

Continuing on the triggers subject, in section 4.5.6 it is referred that the manner of the triggered executable entities are activated, synchronous or asynchronous, depends on how the RTEEvents and BSWEvents are mapped to OS tasks. Further details can be found in this section of [AD-3].

4.2.5 RTE AND BSW INITIALIZATION AND FINALIZATION

In section 4.6.1.1 of [AD-3], regarding the requirement [SWS_Rte_07580], it is referred that it has to be ensured that runnable entities are not activated before the RTE is initialized or after the RTE is finalized by the OS task where BSWEvents and RTEEvents are mapped to the RTE Generator.

Regarding the initialization of the RTE, it is referred in section 4.6.1.2 that, only when the OS is available and all BSWMs are initialized, the ECU state manager calls the startup routine Rte_Start of the RTE at the end of startup phase II.

In section 4.6.1.3 of [AD-3] it is referred that the RTE can rely on the OS functionalities to stop or restart an OS application and all the related OS objects. This is mentioned regarding the stop and restart of the RTE subject, more specifically the requirement [SWS_Rte_07604].

4.2.6 DEVELOPMENT ERRORS

The requirement [SWS_Rte_06631] of the section 4.8.1 of [AD-3] refers that the RTE shall use the OS application identifier as the instance ID in order to allow the developer to indentify in which runtime section of the RTE the error occurs.

4.2.7 RTE REFERENCE

The requirement [SWS_Rte_02711] regards memory consistency on a multi-core ECU by referring that RTE should use OS mechanisms to manage shared memory. Further detail could be found in section 5.3.9.2.

In section 5.10.2 of [AD-3] it is referred that the implementation of a specific RTE Generator, System description and ECU configuration needs to use the APIs provided by the OS module. In case of multi-core the RTE may use the IOC module to implement the intercore communication, being this module part of the OS. In both of these cases it is not specified a list of the expected APIs in this section.

In section 5.11.4.3, regarding OS Trace Events, it is referred that these kind of events occur when the generated RTE interacts with the AUTOSAR OS. The OS Trace Events that can occur are:

- Task Activate;
- Task Dispatch;

- Set OS Event;
- Wait OS Event;
- Received OS Event.

For further detail refer to section 5.11.4.3 of [AD-3].

4.2.8 OS/BSWMS SCHEDULING INTERACTION

In section 6.3.2.3.2 of [AD-3] it is referred, by requirement [SWS_Rte_07511], that in case of BSWMs delivered as source code the definitions of the BSW scheduler API can be optimized during RTE Generation Phase when the mapping of the BSW schedulable entities to OS tasks is known.

The requirement [SWS_Rte_07578], in section 6.5.1 of [AD-3], refers that after the initialization of the OS and before the BSW scheduler is initialized the BSW scheduler shall support calls of SchM_Enter and SchM_Exit. This feature shall also be supported in the context of OS tasks and category 1 and category 2 interrupts, regarding the requirement [SWS_Rte_07579].

In section 6.6.4 of [AD-3], the requirement [SWS_Rte_08059] refers that the RTE shall collect the activating BSW events if the “provide activating” BSW event feature is enabled. In the context of the OS task, the executable entity is mapped to in an activation vector at the corresponding bit position as defined in [SWS_Rte_08058] requirement.

4.2.9 RTE ECU CONFIGURATION

Regarding the mapping of RTEEvents to OS tasks, in section 7.6.1 of [AD-3], it is referred all the RTE parameters and configuration elements that shall be configured properly in order to map AUTOSAR SWCs’ runnable entities to OS tasks. By feature, these parameters and configuration elements are:

- Evaluation and execution order, which consists on the order of runnable entities within the associated OS task;
- Direct function call, which indicates that the respective runnable entity shall be executed in the context of the caller;
- Schedule Points, to allow explicit calls to the OS scheduler in an non-preemptive scheduling setup;
- Time protection support;
- OS interaction, to allow the activation of a OS task and the implementation of a timing event;
- Background activation.

For further details refer to the respective section of [AD-3].

The section 7.6.2 of [AD-3] approaches the configuration items which are closely related to the interaction of the RTE with the OS. In this section it is referred all the used OS objects that the RTE may use to achieve various activation scenarios as also optional configurations of the RTE to support the mapping of modes and OS’ schedule tables and other configurations to the RTE Generator implement a different data consistency mechanism for each exclusive area defined for an AUTOSAR SWC.

4.3. OS/BSWMS INTERACTION

The section 7.1.15 of [AD-6] regards the interrupt service routines implementation. Considering this, the requirement [SWS_BSW_00167] refers that where an ISR is likely to take a long time it should be run as a OS task to not take that long in interrupt context. Regarding the fact of the AUTOSAR architecture does not allow execution in interrupt context on application level, the requirement [SWS_BSW_00182] refers that it shall be restrictions in the transition from ISR to OS task by making this transition take place at the lowest level possible of the BSW. In the case of CAT1 ISR the transition between ISR and OS task shall be at the latest in the RTE, and in the case of CAT2 ISR it shall be at latest in the Microcontroller Abstraction Layer.

In the section 7.1.16 of [AD-6], the requirement [SW_BSW_00326] lists the permissions to usage of OS services for BSWMs:

OS Services	AUTOSAR BSW Modules
Activate Task	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
Terminate Task	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
Chain Task	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
Schedule	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
GetTaskID	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
GetTaskState	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
DisableAllInterrupts	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager
EnableAllInterrupts	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager
SuspendAllInterrupts	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, Microcontroller Abstraction Layer
ResumeAllInterrupts	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, Microcontroller Abstraction Layer
SuspendOSInterrupts	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, Microcontroller Abstraction Layer
ResumeOSInterrupts	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, Microcontroller Abstraction Layer
GetResource	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
ReleaseResource	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
SetEvent	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
ClearEvent	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
WaitEvent	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
GetAlarmBase	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers

OS Services	AUTOSAR BSW Modules
GetAlarm	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
SetRelAlarm	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
SetAbsAlarm	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
CancelAlarm	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
GetActiveApplicationMode	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager
StartOS	ECU State Manager
ShutdownOS	ECU State Manager
GetApplicationID	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
StartScheduleTable	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager
StopScheduleTable	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers ECU State Manager
NextScheduleTable	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager
SyncScheduleTable	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager, Synchronized Time-Base Manager
GetScheduleTableStatus	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager, Synchronized Time-Base Manager
SetScheduleTableAsync	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers ECU State Manager
IncrementCounter	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
GetCounterValue	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager, Microcontroller Abstraction Layer, Synchronized Time-Base Manager, Other BSW Modules
GetElapsedCounterValue	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager, Microcontroller Abstraction Layer, Synchronized Time-Base Manager, Other BSW Modules
TerminateApplication	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
AllowAccess	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers

OS Services	AUTOSAR BSW Modules
GetApplicationState	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers
ControlIdle	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager
GetNumberOfActivatedCores	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager
GetCoreID	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager, Microcontroller Abstraction Layer, Synchronized Time-Base Manager, Other BSW Modules
StartCore	ECU State Manager
StartNonAutosarCore	ECU State Manager
GetSpinlock	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager, Microcontroller Abstraction Layer
ReleaseSpinlock	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager, Microcontroller Abstraction Layer
TryToGetSpinlock	RTE, BSW Scheduler, BSW Mode Manager, Complex Device Drivers, ECU State Manager, Microcontroller Abstraction Layer
ShutdownAllCores	ECU State Manager

The section 8.1.3 of [AD-7] regards all BSW Mode Manager specific functions and theirs definitions. The requirement [SWS_BswM_00118] refers that BswM_Init shall only require the OS and the BSW Scheduler to be initialized before it can be called. The requirements [SWS_BswM_00143] and [SWS_BswM_00144] are related to BswM_TriggerSlaveRTEStop function. They refer that that function shall start a OS task on the slave core which stops the RTE and return the result of the task creation. On the other hand, the requirements [SWS_BswM_00142] and [SWS_BswM_00145] refer that BswMTriggerStartUpPhase2 function shall start a OS task on the slave core which starts the Scheduler Manager and the RTE and return the result of the task creation.

In section 8.1.4.2 of [AD-7] there are specified optional functions for BSW Mode Manager API. One of them is referred by requirement [SWS_BswM_00008] which is ControlIdle. This function allows the caller to select the idle mode action which is performed during idle time of the OS, e.g. if no OS task or ISR is active.

Finally, the section 10.2 of [AD-7] summarizes all containers and configuration parameters of BSW Mode Manager. The configuration parameter [ECUC_BswM_00972] named BswMCoreHaltActivationState, which is part of the container BswMCoreHaltMode ([ECUC_BswM_00970]), represents the Halt mode to be applied to the ECU. The different Halt modes depend on the CPU hardware and the OS implementation, and they are defined as strings in the OS. The configuration parameter [ECUC_BswM_00973] named BswMCoreRef, which is part of the container BswMTriggerSlaveRTEStop ([ECUC_BswM_00919]), represents the reference to the identifier of the slave core that is used as input parameter for the BswM_TriggerStartUpPhase2 and

BswM_TriggerSlaveRTEStop functions and its value shall be synchronized with the OsApplicationCoreAssignment parameter.

Acronyms and symbols

Abbreviation	Meaning
AMP	Asymmetric Multiprocessing
API	Application Programming Interface
CAN	Controller Area Network protocol
CSW	Critical Software
ECU	Electronic Control Unit
EPIT	Enhanced Periodic Interrupt Timer
FIQ	Fast Interrupt Request
FlexCAN	Flexible Controller Area Network
GIC	ARM's Generic Interrupt Controller
HRT	Hard-Real-Time
HW	Hardware
I2C	Inter-Integrated Circuit
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MB	Message Buffer
MSD	Minimum-Set-of-Data
OS	Operating System
RTOS	Real-Time Operating System
Rx	Reception
SMC	Secure Monitor Call
SMP	Symmetric Multiprocessing
SoC	System on Chip
SW	Software
Tx	Transmission
TZ	ARM TrustZone
TZASC	TZ Address Space Controller
UART	Universal Asynchronous Receiver/Transmitter
VM	Virtual Machine