Safety-critical Control in Mixed Criticality Embedded Systems

A Study on Mixed Criticality in the Automotive Industry

Master Thesis
Royal Institute of Technology
Stockholm, Sweden

Emil Hjelm
emilhje@kth.se

August 20, 2017

Master Thesis MMK2017:Z MDAZZZ

Safety-critical Control in Mixed Criticality Embedded Systems

Emil Hjelm

Approved: (datum)	Examiner: Martin Törngren	Supervisor: Bengt Eriksson
	Commissioner: Alten Sverige AB	Contact person: Detlef Scholle

Abstract

Modern automotive systems contain a large number of Electronic Control Units, each controlling a specific system of a specific criticality level. To increase efficiency it is desired to combine multiple applications into fewer ECUs, leading to mixed criticality embedded systems. The assurance of safety critical applications not being affected by non-critical applications on the same system is crucial.

A system for vehicle platooning is implemented on a platform hosting systems of mixed criticality where safety-critical systems are separated from non-critical systems via the hypervisor SafeG. The hypervisor added an overhead of 0.6% and increased useful system utilization from 0.005% to potentially 99.4%.

The hypervisor showed good isolation properties and the non-critical systems could fail without affecting the safety-critical systems.

Examensarbete MMK2017:Z MDAZZZ

Säkerhetskritisk kontroll i blandkritiska inbyggda system

Emil Hjelm

Godkänt: (datum)	Examinator: Martin Törngren	Handledare: Bengt Eriksson
	Uppdragsgivare: Alten Sverige AB	Kontaktperson: Detlef Scholle

Sammanfattning

Dagens bilar har många små datorer som var och en kontrollerar enskilda delsystem av olika säkerhetskritiska nivåer. För att öka resurseffektiviteten är det önskvärt att kombinera de olika applikationerna på färre datorer, vilket skulle leda till blandkritiska inbyggda system. I blandkritiska system är det viktigt att icke-kritiska funktioner inte kan påverka säkerhetskritiska funktioner.

I detta arbete utvecklas och implementeras ett system för kolonn-körning (platooning) på en plattform som huserar funktioner av olika säkerhetskritisk grad. De olika funktionerna separeras i en säkerhetskritisk och en icke-kritisk del via SafeG. SafeG introducerade en overhead på 0.6% och ökade nyttiga utnyttjandegraden från 0.005% till potentiellt 99.4%.

Isoleringsmekanismen visade goda isoleringsegenskaper och de icke-kritiska applikationerna kunde haverera utan att påverka de säkerhetskritiska applikationerna.

Preface

This master thesis is one part of a larger project with four other master thesis students at Alten Sweden. I first and foremost want to thank each and every one of this group: Daniel Roshanghias, Erik Lerander, Hanna Hellman and Sanel Ferhatovic. I also want to thank Youssef Zaki, whose work this thesis is largely based on. Lastly I want to thank Detlef Scholle and Alten for the opportunity to conduct my master thesis at Alten.

Emil Hjelm
Stockholm

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Description
ECU	Electronic Control Unit
MCS	Mixed Criticality System
EMC^2	Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments
RTOS	Real-Time Operating System
GPOS	General Purpose Operating System
FPGA	Field Programmable Gate Array
ASIC	Application Specific Integrated Circuit
SIL	Safety Integrity Level
ASIL	Automotive Safety Integrity Level
DAL	Development Assurance Level
VMM	Virtual Machine Monitor
EMC^{2}DP	EMC^2 Development Platform
RM	Rate Monotonic
DM	Deadline Monotonic
FP	Fixed Priority
EDF	Earliest Deadline First
AUTOSAR	AUTomotive Open System ARchitecture
VMM	Virtual Machine Monitor
OS	Operative System
SafeCOP	Safe Cooperating Cyber-Physical Systems
FIFO	First In First Out
RR	Round Robin
CrMPO	Criticality Monotonic Priority Ordering
SMC	Static Mixed Criticality
AMC	Adaptive Mixed Criticality
WCET	Worst Case Execution Time
BCET	Best Case Execution Time
IRQ	Interrupt Request
FIQ	Fast Interrupt Request
VHDL	Very High Speed Integrated Circuit Hardware Descriptive Language
SoC	System on Chip
PS	Processing System
PL	Programmable Logic
FSBL	First Stage Boot Loader
SSBL	Second Stage Boot Loader
BSP	Board Support Package
CC	Cruise Control

Chapter 1

Introduction

This chapter will introduce the subject of mixed criticality embedded systems and the EU project "EMC^2" to the reader.

1.1 Background

Today, modern automotive systems contain approximately 70-100 Electric Control Units (ECUs) [27]. Each ECU controls a subsystem of a specific criticality level such as safety-critical anti-lock brake system, or non-critical entertainment systems [41]. Having the ECUs isolated ensures that the numerous critical and non-critical applications do not interfere with each other, making it relatively easy to certify an individual ECU. However, this approach leads to an inefficient use of system resources and expensive system implementation [10]. In order to lower the cost of the collective system and increase system efficiency (utilization), applications of different criticality levels can be integrated into a single hardware platform, leading to a Mixed Criticality System (MCS). However, this approach increases system complexity, and hinders the certification of safety-critical systems [48]. To enable design, test, and certification of MCS, spatial and temporal partitioning can be used in the architecture of the system.

Protecting the integrity of a component from the faults of another is desired in all systems hosting multiple applications. However, it is of higher significance if the different applications have different criticality levels. Without such protection all components on the same system would need to be engineered to the standards of the highest criticality level, potentially massively increasing development costs [10].

With cars being increasingly connected to the internet, security poses another concern for MCS. If one part of a system is easier to attack, the entire system could be compromised if proper isolation is not in place.

The EU project "Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments" (EMC^2) was founded in order to "find solutions for dynamic adaptability in open systems, provide handling of mixed criticality applications under real-time conditions, scalability and utmost flexibility, full scale deployment and management of integrated tool chains, through the entire lifecycle" [41].

1.1.1 Definition of safety-critical systems

The term "safety-critical system" has many definitions, most quite similar. Most definitions relate to systems with the potential to harm humans if the system malfunctions. According to [15] it is defined as "A system in which any failure or design error has the potential to lead to loss of life." Further, [12] defines safety-critical systems as "A computer, electronic or electromechanical system whose failure may cause injury or death to human beings." A Wikipedia article, [44], defines a safety-critical system (or "life-critical system") as a system whose failure or malfunction may result in one (or more) of the following outcomes:

- death or serious injury to people
- loss or severe damage to equipment/property
- environmental harm

In this thesis, a safety-critical system will be defined as "a system whose failure may cause injury or death to human beings."

1.1.2 Different levels of criticality

Different names of levels of criticality are typically Safety Integrity Level (SIL), Automotive Safety Integrity Level (ASIL) and Development Assurance Level (DAL). The IEC 61508 standard [20] defines four different levels and the ISO 26262 standard [23] and the DO-178C standard [14] define five different levels each. These levels range from low or no hazard up to life-threatening or fatal in the event of a malfunction requiring the highest level of assurance that the dependent safety goals are sufficient and have been achieved.

The number of criticality levels in the implementation part of this thesis will be restricted to two: "safety-critical" and "non-critical". This is due to the constraints presented in section 1.1.3 and 1.5.

1.1.3 Alten Mixed Criticality System

As a part of the EMC^2 project, Alten has developed a system for handling applications of mixed criticality on the same piece of hardware, Alten MCS. The MCS developed at Alten has been implemented on two development boards, the Zedboard (Zynq Evaluation and Development board) and the EMC^2 Development Platform, or EMC^{2}DP. Both boards have a Xilinx Zynq-7000 SoC [47]. Alten MCS employs two operating systems to handle applications of different criticality. A General Purpose Operating System (GPOS) for non-critical applications and a Real-Time Operating System (RTOS) for safety-critical applications. A hypervisor is used to alternate between the two. For more information, see chapter 3.

1.1.4 Platooning

Platooning is the concept of several vehicles driving coordinated with short intermediate distances in order to reduce air friction for the collective platoon. Platooning puts high requirements on the control systems in the platooning vehicles, since failure of these systems could have fatal consequences. Some expected advantages of platooning include increased fuel and traffic efficiency, safety and driver comfort [9].

1.2 Problem statement

An ideal MCS ensures partitioning between different criticality levels while still sharing resources efficiently. This leads to the underlying research question:

- "How, in a disciplined way, to reconcile the conflicting requirements of partitioning for safety assurance and sharing for efficient resource usage?" [10]

The MCS developed at Alten switches Operative System (OS) via a hypervisor to enable partitioning between safety-critical and non-critical applications, which takes about 2 μs. This means that at higher switching frequencies the mode switch takes up more processor time, potentially making up

almost all processor time at very high frequencies.

To evaluate the performance of the system, a distance keeping control algorithm for platooning will be implemented on it. A demonstrator will be constructed in the form of a RC car capable of following a vehicle in front of it at a specified distance. If the lead car exceeds a predefined maximum speed or deviates from the road, the following car should not exceed the maximum speed. It should be verified that no matter the computational load and eventual crashes of the Linux based non-critical system, the distance keeping algorithm on the RTOS should never crash.

This problem leads to the research question:

- Is virtualization an efficient approach when trying to reconcile the conflicting requirements of partitioning for safety assurance and sharing for efficient resource usage when implementing a safety-critical control system?

This thesis will attempt at answering this question by doing a state of the art study in the research topic, implement a safety-critical system on a mixed criticality platform and evaluate the results.

1.3 Purpose

Reducing the amount of computers in automotive systems would have many effects. Manufacturing costs would decrease and with fewer physical components maintenance costs would also decrease. However, the system complexity would increase and thereby increasing time and cost to design the system.

SafeCOP (Safe Cooperating Cyber-Physical Systems) is an European project that targets cyberphysical systems-of-systems whose safe cooperation relies on wireless communication [33]. SafeCOPs Use Case 3 (UC3) regarding "Vehicle control loss warning" together with the EMC2 goals tie well in with the problem statement and use case described in 1.2.

1.4 Goals

In this project there are both team goals and individual goals that do not always necessarily align with each other.

1.4.1 Team goal

The team consists of five master thesis students. The students areas of work are: control theory and system modeling, data aggregation, safety-critical communication in MCS, lane detection and finally safety-critical control in MCS. Together the team will build a vehicle capable of following a vehicle ahead of it while keeping inside road markers.

1.4.2 Individual goal

Verify quantitatively the performance of safety-critical distance keeping controller, see section 1.6. Solve the problems described in section 1.2.

1.5 Scope

The work of this thesis and the implementation on the demonstrator will build upon the work of Youssef Zaki [48].

The embedded computer is constrained to the Xilinx Zynq-7000 SoC [47]. The hypervisor used is constrained to SafeG [39] and the RTOS is constrained to FMP [38].

1.6 Research design

The plan is to conduct a confirmatory investigation using quantitative data/operations with a deductive approach. This is a quantitative research method. [19].

1.7 Ethical considerations

When designing a MCS it is crucial to ensure that errors made by a lower criticality application cannot propagate to higher criticality applications, as this could have catastrophic consequences. Because of this the requirement of partitioning must have higher priority than the need of sharing resources.

Chapter 2

State of the art

This chapter will go through relevant articles and research on the subject of mixed criticality systems, vehicle platooning and safety standards in the automotive industry.

2.1 Mixed criticality systems

A MCS is achieved by letting applications of different criticality share resources. These resources could be the processor, memory, peripherals, input/output ports etc. The most explored area is sharing the CPU between multiple criticality levels [10]. The benefit of combining previously distributed systems into a MCS is higher resource efficiency, which potentially leads to economical benefits.

2.1.1 Economical benefits of MCS

Potential benefits with pursuing MCS as opposed to distributed systems are reduced physical space required, reduced weight, reduced heat generation, reduced power consumption and reduced production costs [10]. This would all ultimately lead to economical benefits.

Potential downsides are increased complexity which could lead to higher system design costs. Building applications on the same platform to share resources could require engineering teams to work more closely together, potentially leading to administrative difficulties and costs. This needs to be investigated and could vary from industry to industry. To combat the potential downsides, the EMC^2 project aims at creating platforms for easier development of MCS.

The EMC^2 project lists several goals [42]:

- Reduce the cost of the system design by 15%
- Reduce the effort and time required for re-validation and re-certification of systems after making changes by 15%
- Manage a complexity increase of 25% with 10% effort reduction
- Achieve cross-sectorial reusability of Embedded Systems devices and architecture platforms that will be developed using the ARTEMIS JU results.

2.1.2 Security concerns

Today, cars are increasingly becoming networked IT entities. With increased connection, security threats are also increasing, and the most visible gateway for hacking attempts in cars is through the infotainment systems. This gives additional incentive to ensure proper isolation in MCS.

2.1.3 Sharing processor

To deal with many different tasks needing processor time, different schedulers can be used to appropriately distribute processor time among the tasks.

2.1.3.1 Conventional scheduling

The simplest scheduling algorithm is First In First Out (FIFO). As the name suggests, the first task to enter the queue is also executed first, and executes until it is finished [4].

Another simple scheduling algorithm is Fixed Priority (FP). Each task is assigned a priority level, and the task with the highest priority in the queue is allowed to execute [29].

The scheduling algorithm Rate Monotonic (RM) assigns a priority level to a task depending on its frequency. Higher frequency leads to higher priority. The algorithm was proven optimal under the assumption that the deadline of every task coincided with its rate [29].

Deadline Monotonic (DM) assigns a priority level to a task depending on its deadline. Shorter deadline leads to higher priority [29].

Earliest Deadline First (EDF) is a dynamic scheduling algorithm that executes the task with the least amount of time left until its deadline.

Round Robin (RR) scheduling lets each task in the job queue execute for a predefined amount of time (called a time quanta), and if a task does not complete it is preempted and placed back in the queue waiting for its next time slot [25].

2.1.3.2 Mixed criticality scheduling

The area of sharing the processor in MCS was first explored by Steve Vestal [40] in 2007. His paper showed that neither Rate Monotonic (RM) nor Deadline Monotonic (DM) priority assignment was optimal for MCS.
In 2008 Baruah and Vestal [8] showed that EDF (Earliest Deadline First) does not dominate FP when criticality levels are introduced, and that there are feasible systems that cannot be scheduled by EDF.

One MCS scheduling algorithm is Criticality Monotonic Priority Ordering (CrMPO). Tasks are assigned priorities first according to criticality (highest criticality first) and then according to deadline (shortest deadline first). Static Mixed Criticality with no run-time monitoring (SMC-NO) is the scheduler that was Vestal's original approach [40]. Another scheduler is SMC with run-time monitoring (abbreviated only as SMC). Yet another scheduling algorithm is Adaptive Mixed Criticality (AMC), described Baruah, Burns and Davis [7].

To evaluate the performance of the different scheduling algorithms Baruah, Burns and Davis [7] tested the scheduling algorithms AMC, SMC and CrMPO for scheduling sporadic tasks of a taskset of 20 tasks where on average 50% where of high criticality and 50% where of low criticality. The tasks of high criticality where allowed an execution time that was twice its low criticality execution time. The comparison of the performance of the schedulers can be seen in Figure 2.1. In the graph the UB-H&L line bounds the maximum possible number of schedulable task sets.

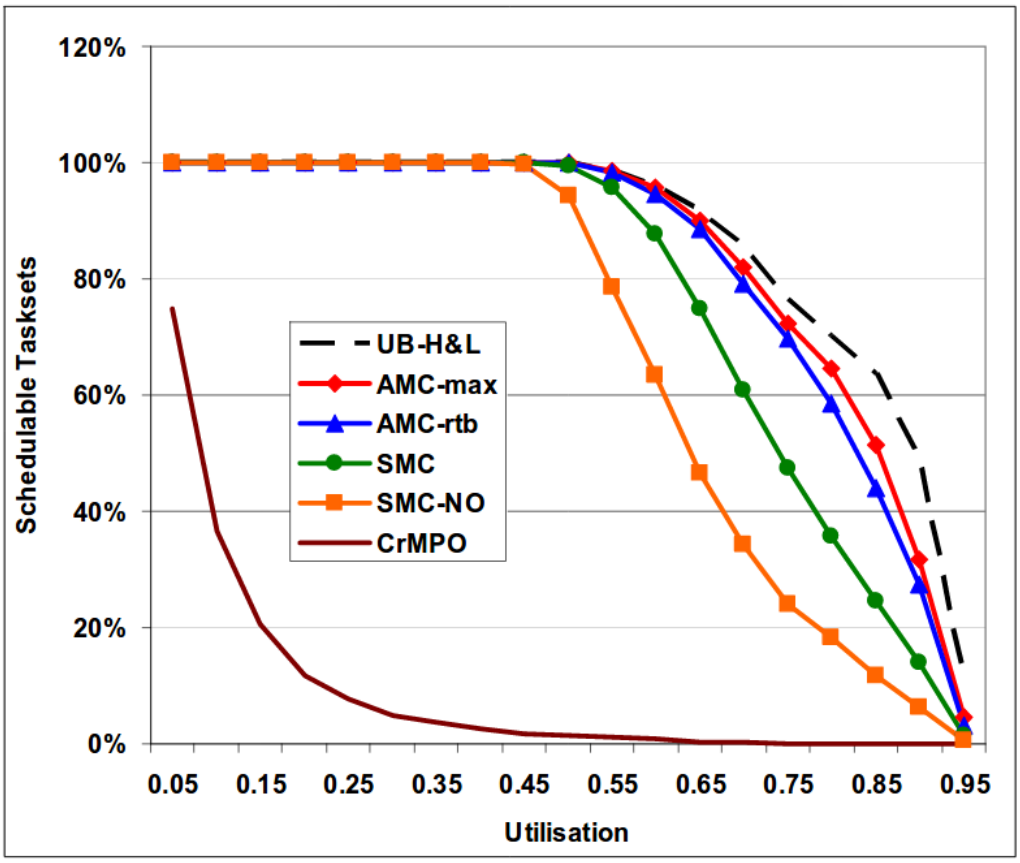


Figure 2.1: Percentage of schedulable tasks. [7]

For a more complete review of work done on MCSs with a shared processor, see the paper by Burns [10].

2.1.3.3 Execution times

A key concept when designing a real time system is the worst case execution time of each task, or WCET. This is the longest amount of time a task will take to execute on the specific hardware. Generally, it is very rare for a task to execute for its full WCET, but is typically executed for its median execution time, see figure 2.2.

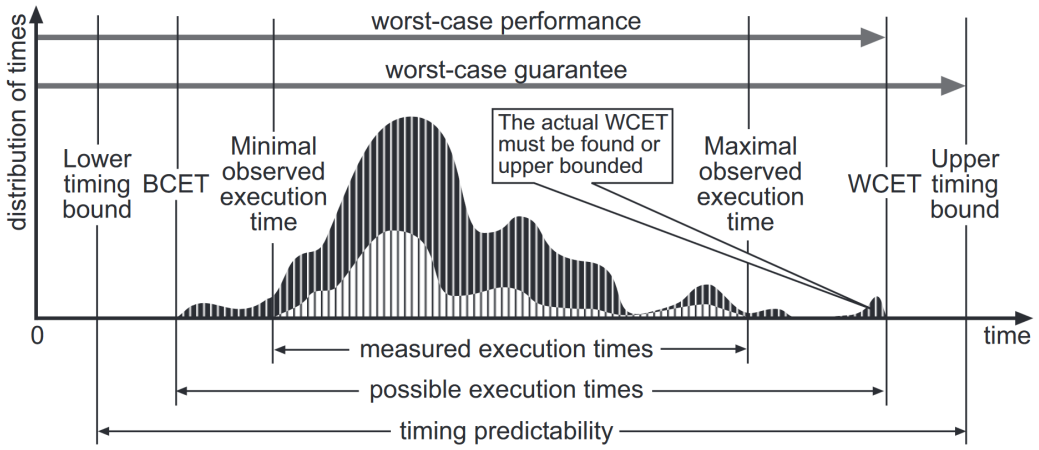


Figure 2.2: WCET illustrated. [45]

In a GPOS, scheduling is optimized for throughput. This means that shorter and therefore sometimes missed deadlines are allowed in exchange for higher rate of tasks. The scheduling in a RTOS however is designed so that every task is guaranteed to execute until it is finished. To be able to guarantee this, the WCET of each task must be known. There are two approaches to learn the WCET of a task, static and dynamic methods [45].

Static based methods involve analysis of code, counting assembler instructions for each part of the code and combining these to get a maximum amount of instructions, where each instruction has a known execution time [45].

Dynamic based methods are a bit more straight-forward, you simply let the code execute many times and measure the time each execution of the code takes. This can be measured by for example setting an output pin high at the start of the task and then setting it low at the end, and have an external clock measure the time the pin is high [45]. The dynamic approach is the most common method in most parts of industry [45].

2.2 Standards

To ensure safe and secure practices in industries, safety standards are becoming more regulated. Different standards address different areas. Below, the two safety standards IEC 61508 and ISO 26262 will be described.

2.2.1 IEC 61508

IEC 61508 [20] is intended to be a basic functional safety standard for electrical and electronic systems applicable to all kinds of industry. It defines four different safety integrity levels, SIL 1 being the least dependable up to SIL 4 which is the most dependable level.

2.2.2 ISO 26262

ISO 26262 [23] is a functional safety standard derived from IEC 61508. It addresses the needs for an automotive-specific international standard that focuses on safety critical components.

2.2.2.1 ASILs

ISO 26262 describes five different Automotive Safety Integrity Levels (ASIL) relating to hazard and risk. Ranked from lowest (no) hazard to highest hazard, these levels are: QM, A, B, C and D. A function is assigned an ASIL depending on the severity if the function fails, the probability that the function fails and the controllability of the function, see table 2.1.

Severity	**Probability**	**Controllability**		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Table 2.1: ASIL as a function of severity, probability and controllability.

The various integrity levels can be translated into integers (ASIL $QM = 0$; $A = 1$; $B = 2$; $C = 3$ and $D = 4$). If a hazard requires several components to fail, the added ASIL of these components is used to determine if there is an violation, assuming the components faults are statistically independent

of each other. For example, a safety level ASIL B can be met by two independent components which each individually only meet ASIL A (and thus effectively $A + A = B$). [6]

Developing functions to standards of different ASIL levels have different costs. The different ASILs can relate to cost according to various cost heuristics, see table 2.2.

Cost Heuristic	**QM**	**A**	**B**	**C**	**D**
Linear	0	10	20	30	40
Logarithmic	0	10	100	1000	10000
Experimental-I [6]	0	10	20	40	50
Experimental-II [6]	0	20	30	45	55

Table 2.2: ASIL cost heuristics.

2.2.3 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a partnership between several automotive interested parties. The partnership aims at establishing and standardizing the software architecture for automotive ECUs, excluding infotainment [5]. Contributors include BMW, Daimler, Ford, GM, Toyota and Volkswagen.

2.3 Hypervisors

A hypervisor (or virtual machine monitor) is a piece of software capable of running several operating systems on the same hardware system. The hypervisor can be seen as an interface between the operative systems and the hardware of a system. In order to facilitate for multiple operative systems of different criticality an appropriate hypervisor should be used. A system will only be as secure as its hypervisor, so it is important that the hypervisor has been engineered to at least the same standards as the most safetly-critical functions that will be implemented on the system. Other things to consider is that some hypervisors can accommodate multiple instances of operative systems, and some only facilitate for two. A description of a few available open-source hypervisors will follow.

2.3.1 SafeG

SafeG (short for *Safety Gate*) is a hypervisor developed by the TOPPERS group of Nagoya University in Japan [39]. It can host two operating systems on the same hardware, partitioning them into Trusted and Non-Trusted zones (sometimes called Trusted and Non-Trusted "worlds") via ARM TrustZone [3].

SafeG guarantees both that the memory and temporal execution of tasks in the OS on the secure side is isolated from the OS on the non-secure side [34]. Both memory and time isolation is backed by the ARM TrustZone hardware extensions [34]. RTOS tasks are triggered by FIQ (Fast Interrupt Request), an interrupt with higher priority than IRQ (Interrupt Request), which is allocated to the GPOS [34].

Figure 2.3 depicts the SafeG architecture. It shows a simplified view of a TrustZone enabled ARM processor together with partitioned memory and device IO.

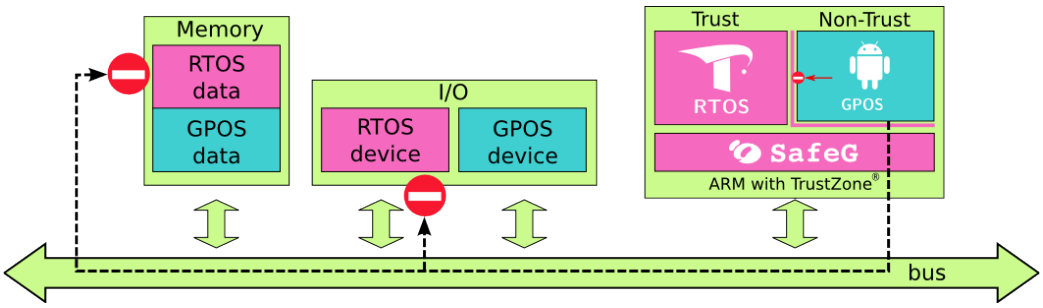


Figure 2.3: SafeG and TrustZone.

The features of SafeG as described on their website are as follows [39]:

- Enables the concurrent execution of RTOS and GPOS on either single-core or multi-core ARM-based platforms.
- Devices and memory regions that are configured as Secure are protected against illegal GPOS accesses.
- Normal world devices can be accessed from both GPOS (Non-Trusted) and RTOS (Trusted) software.
- Real-time requirements are guaranteed in RTOS (Trusted) via the utilization of FIQ and IRQ interrupts, where FIQ interrupts are issued for

RTOS and IRQ interrupts are issued for GPOS. While in the Trusted state, IRQ interrupts are disabled so that GPOS can not disturb the execution of the RTOS. Therefore, GPOS only executes when the RTOS issues the Secure Monitor Call (SMC) instruction, which causes the SafeG monitor to switch from the Trusted world (RTOS) to the Non-Trusted world (GPOS). Furthermore, FIQ interrupts are active during the execution of GPOS, which enables RTOS to retake control of the system. For example, a cyclic execution of RTOS/GPOS can be controlled by an FIQ interrupt of a system timer.

- GPOS does not require any major changes, and can execute with minimal overhead.
- Includes an efficient guest-to-guest communication mechanism (i.e.referred to as SafeG COM).

The RTOS tasks are statically mapped to a processor during compilation, while the GPOS uses all available virtual CPUs in Symmetric Multi-Processor mode.

The SafeG Monitor is the gateway between the GPOS and the RTOS. During the switch operation, it is responsible for saving the state of one zone and loading the state of the other. According to TOPPERS [35], the WCET of switching the OS is just under $2\mu s$, see table 2.3.

Switch path	WCET
Switch from RTOS to GPOS	$1.5\mu s$
Switch from GPOS to RTOS	$1.7\mu s$

Table 2.3: WCET of switching OS. [35]

2.3.2 seL4 microkernel

seL4 is a microkernel developed, maintained and formally verified by NICTA (now the Trustworthy Systems Group at Data61) and owned by General Dynamics C4 Systems [31]. seL4 is formally verified, which implies that its specification is verified mathematically [11]. The kernel follows the "minimality principle", meaning that it "allows features in the kernel only if the required functionality could not be achieved by a user-level implementation" [30]. As a result, the microkernel is small, efficient, and robust.

Scheduling in seL4 is deliberately left underspecified. The present implementation uses a fixed-priority round-robin scheduler [30].

2.3.3 SICS Thin hypervisor

SICS Thin Hypervisor (STH) is a light-weight hypervisor designed for ARM-based devices [13]. STH runs directly on top of the hardware (bare metal), and achieves system virtualization through paravirtualization. STH allows for more than two guests to run on top of the hypervisor, but has no TrustZone support.

The current version of STH supports ARMv5 (926EJ-S) and ARMv7 Cortex-A8 only, but due to its flexible nature and minimal layer of hardware abstraction it is easily migrated to other platform [13]. There is no graphical support and all communication with the kernel is done through the UART.

2.3.4 Sierra visor

SierraVisor is a bare metal universal hypervisor created by Sierraware [37]. It supports paravirtualization, TrustZone virtualization, and hardware assisted virtualization. The SierraVisor Hypervisor supports any ARM11, Cortex-A9, or Cortex-A15 platform, but only Cortex-A15 supports the hardware assisted virtualization option.

2.4 Field Programmable Gate Array

A Field Programmable Gate Array (FPGA) is an array of logic gates with reconfigurable interconnects that can be connected to each other. As the name suggests, it is possible to reprogram the physical function of the FPGA "in the field". This is what distinguishes a FPGA from an Application Specific Integrated Circuit (ASIC), an ASIC can not be reprogrammed after manufacturing.

To program the behavior of the FPGA, Hardware Descriptive Language (HDL) is used. The two most widely used HDLs are Verilog and VHDL (VHSIC Hardware Descriptive Language, where VHSIC is an acronym for Very High Speed Integrated Circuit). Verilog came to existence late 1983. The language was standardized as IEEE standard 1364-1995, and has since been revised and included in IEC/IEEE Behavioural Languages standard 61691 [22]. Work with VHDL formally began in 1981, and in 1987 it was

standardized as IEEE-1076. The active standard for VHDL is also part of the IEC/IEEE Behavioural Languages standard 61691 [21]. Using Google Trends [17] with the search terms "vhdl" and "verilog", one can see that Verilog seems to be the more popular language of the two in the USA, India and South Korea, while VHDL is more used in Europe and South America, see Figure 2.4.

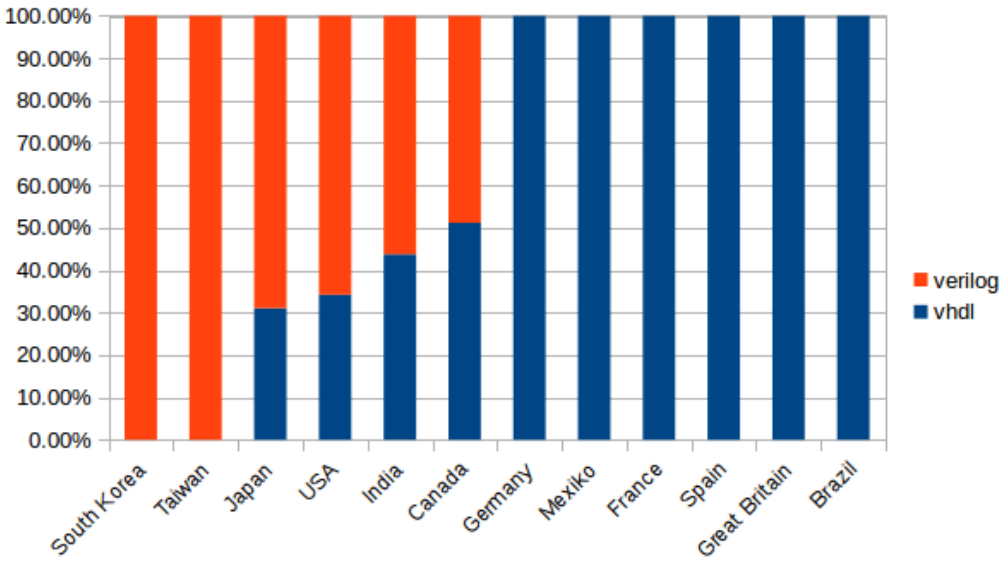


Figure 2.4: Distribution of Google searches for "verilog" vs "vhdl" in the world.

A key difference between code running on a processor and code describing functions in a FPGA is that traditional code is executed in serial, where only one computation can be executed at any one time. In a FPGA however, functions can run in parallel. This makes FPGAs excellent for implementation of hardware functions that require high speed and high frequency that you do not want to take up precious processor time.

Some FPGA circuits have non-volatile memory from where the hardware descriptive language is loaded when the circuit is powered up, this means that its configuration is not lost if its power goes down.

2.5 Platooning

Platooning, road trains or convoy driving is the concept of a chain of vehicles with no physical connection travelling at a given (short) intermediate distance

in order to utilize the reduced air friction behind the vehicle in front. The primary objective for each vehicle with respect to safety is to maintain its distance to the preceding vehicle in the platoon.

2.5.1 Benefits of platooning

Potential benefits of vehicle platooning includes lower fuel consumption, less road space required and more efficient traffic flow.

Using simulations of platooning, Alam, Gattami, and Johansson [1] showed in 2010 that there is a 4.7–7.7% fuel reduction potential in heavy duty vehicle platooning at a set speed of 70 km/h with two identical trucks.

In 2014, Lammert et. al. [26] showed that platooning can result in reduced fuel consumption of up to 5.3% for the lead vehicle and up to 9.7% for following vehicles.

Tests by truck manufacturer Scania have shown that platooning can reduce fuel consumption by up to 12% [36].

2.5.2 Safety requirements for platooning

With shorter distance between vehicles, the margin of error also decreases. This puts high requirements on the systems controlling the speed of the vehicles in the platoon.

In an article by Alam et al. [2], safe sets for heavy duty vehicle platooning are computed to calculate minimum distance between the vehicles in a platoon without endangering safety. Varying parameters are concidered such as mass, air resistance and road gradient since these could cause a difference in braking capabilities between the vehicles. Alam et al. [2] show that if two identical vehicles in a platoon drive with a speed of 90 km/h and keep a minimum distance of 1.2 m, a collision can always be avoided. In the case of communication delays of up to 500 ms, a distance of at least 2 m should be kept.

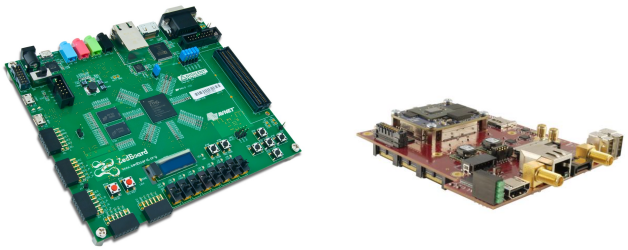
Chapter 3

Alten MCS

This chapter will describe the Alten MCS, its different components and how it is built.

3.1 Overview

The Alten MCS (Alten Mixed Criticality System) is a Mixed Criticality System consisting of both hardware and software components. The general idea is that it should be capable of running two operating systems of different criticality on the same hardware, separated via a hypervisor, where errors from the non-critical OS should not be able to propagate into the safety-critical OS. The Alten MCS has currently been built on two different development boards, the EMC^2 Development Platform (EMC^{2}DP) and the Zynq Evaluation and Development board (Zedboard). Both boards are equipped with a Zynq-7000 System on Chip (SoC). The development boards can be seen in Figure 3.1.



(a) Zedboard (b) EMC^{2}DP

Figure 3.1

3.2 Hardware

The Zynq-7000 SoC has a Processing System (PS) consisting of a hardwired application processing unit, memory controller, and peripheral devices. The main processing unit is a dual-core Cortex-A9 ARM processor. Connected to the PS region is a Programmable Logic (PL) region. The PL is based on Xilinx's 7-series FPGA technology.

Both the PS and PL regions have support for ARM TrustZone [3].

3.3 Operative systems

The Alten MCS uses two Operative Systems (OS) to create temporal and spatial separation between safety-critical and non-critical applications using TrustZone. In its current setup the Real-Time Operative System (RTOS) FMP by TOPPERS [38] is used for safety-critical applications. This RTOS follows the uITRON4.0 specification [24], which is a widely used RTOS specification for Japanese embedded systems. For non-critical applications, the General Purpose Operative System (GPOS) Linux kernel 4.4 is used. Instead of Linux another instance of FMP could be used for non-critical applications.

Tasks in FMP are statically allocated to a processor core, defined in a .cfg file. Linux divides its workload across all available processor cores in SMP (Symmetric Multi-Processor) mode.

3.4 Hypervisor

A hypervisor (also known as a Virtual Machine Monitor (VMM)) is used to alternate between the safety-critical (S_OS) and non-critical (NS_OS) OS. The hypervisor used is SafeG [39], also developed by TOPPERS. It switches processor state via a hardware switch. See figure 3.2. The switching takes roughly 2 μs [35].

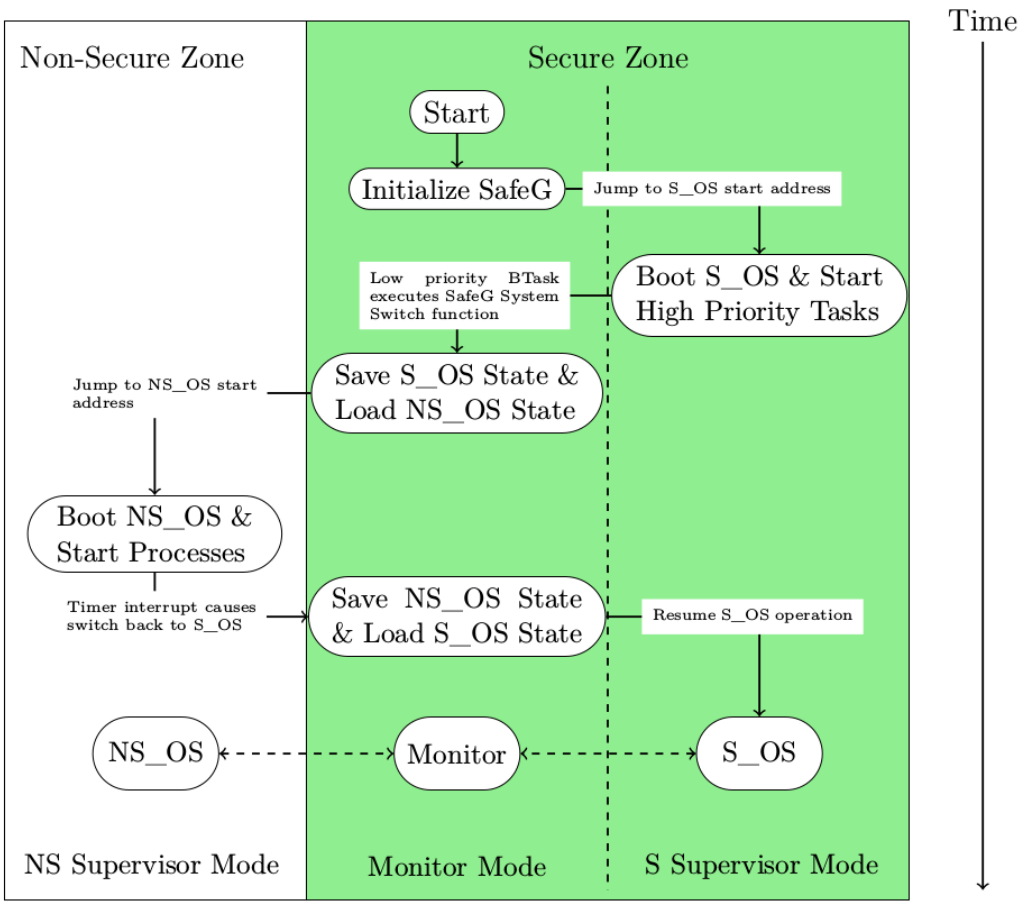


Figure 3.2: Flowchart of the boot sequence of the CPU. [48]

When executing the tasks via the hypervisor, all S_OS tasks are executed according to the scheduling. At the lowest priority is the task that switches processor state. As soon as an S_OS task needs to execute, a FIQ is issued to interrupt the processor and switch mode back to the S_OS.

The time it takes the hypervisor to switch processor state bounds the theoretical maximum frequency a cyclic task can have while the processor still manages to maintain its switching capabilities. The maximum frequency, f_{max}, can be calculated as

$$f_{max} = \lim_{e_s, e_{ns} \to 0} \frac{1}{e_s + e_{ns} + 2e_{switch}} \tag{3.1}$$

where e_s is the execution time of the tasks on the S_OS, e_{ns} is the execution time of the tasks on the NS_OS and e_{switch} is the time required for the mode-

switch.

A more practical limitation is that the periodicity of a cyclic task in FMP is defined by an integer in milliseconds, meaning that the highest frequency for a periodic task is 1 kHz.

A basic overview of the hardware and the software of the system can be seen in Figure 3.3.

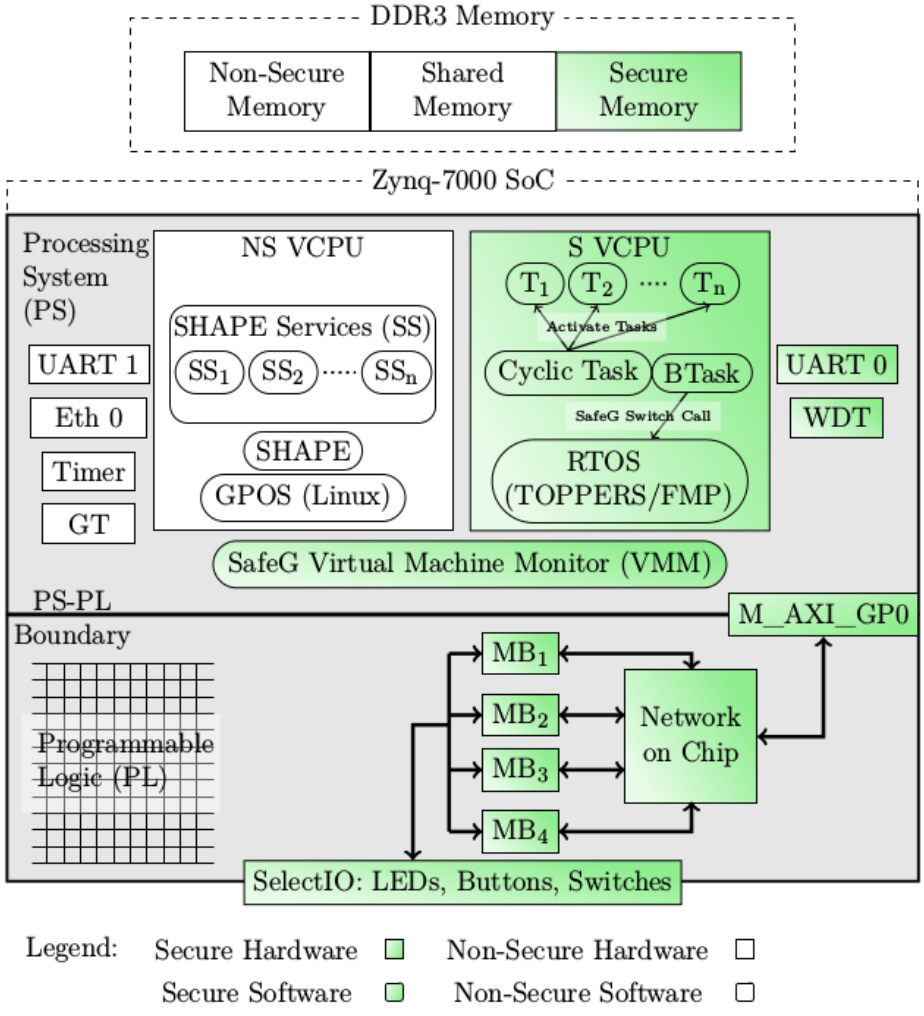


Figure 3.3: Overview of the Alten MCS. [48]

3.5 Build procedure

The MCS is built from many different components. Hardware design, applications, virtualization layer, operative systems, boot loaders etc. This section will describe the build procedure.

Xilinx's software Vivado HLS [46] is used to design the FPGA of the Zynq-7000. In Vivado the different IPs, their interconnect and the Processing System is configured. This results in a bitstream file that is used in Xilinx SDK to generate a Board Support Package (BSP) and the First Stage Boot Loader (FSBL). The BSP contains libraries for functions and variables necessary for the IPs on the FPGA, and the FSBL configures the FPGA correctly. The FSBL is included in a boot file (BOOT.bin) generated in Xilinx SDK. The boot file is loaded to an SD card that is inserted into the development board. When powered up, the Zynq-7000 boots from the SD card and configures the FPGA according to the FSBL and loads the Second Stage Boot Loader (SSBL), u-boot, which also is contained in the boot.bin file. U-boot is a program that can load executables and other system files from a remote server into the DDR3 memory. This is used to load relevant files for the OS into the Zynq, which brings us to the software part of the build procedure. The software is divided into three parts: RTOS, GPOS and monitor. The SafeG monitor is built and compiled using GCC (Gnu Compiler Collection), resulting in monitor.bin. The RTOS FMP is built and compiled from a .cfg file where tasks and handlers are created, a .c file where tasks and handlers are defined and a .h file containing declarations. This results in fmp.bin. For the non-secure side either Linux or another instance of FMP could be used. The process for FMP is the same as with the secure side. For Linux on the non-secure side, the Linux kernel needs to be modified to support SafeG. This results in the files zImage, devicetree.dtb. All of these files are created and stored on a server and are then loaded into the Zynq-7000 using u-boot.

Figure 3.4 provides a summary of the different dependencies for the system and the required flow for building the system.

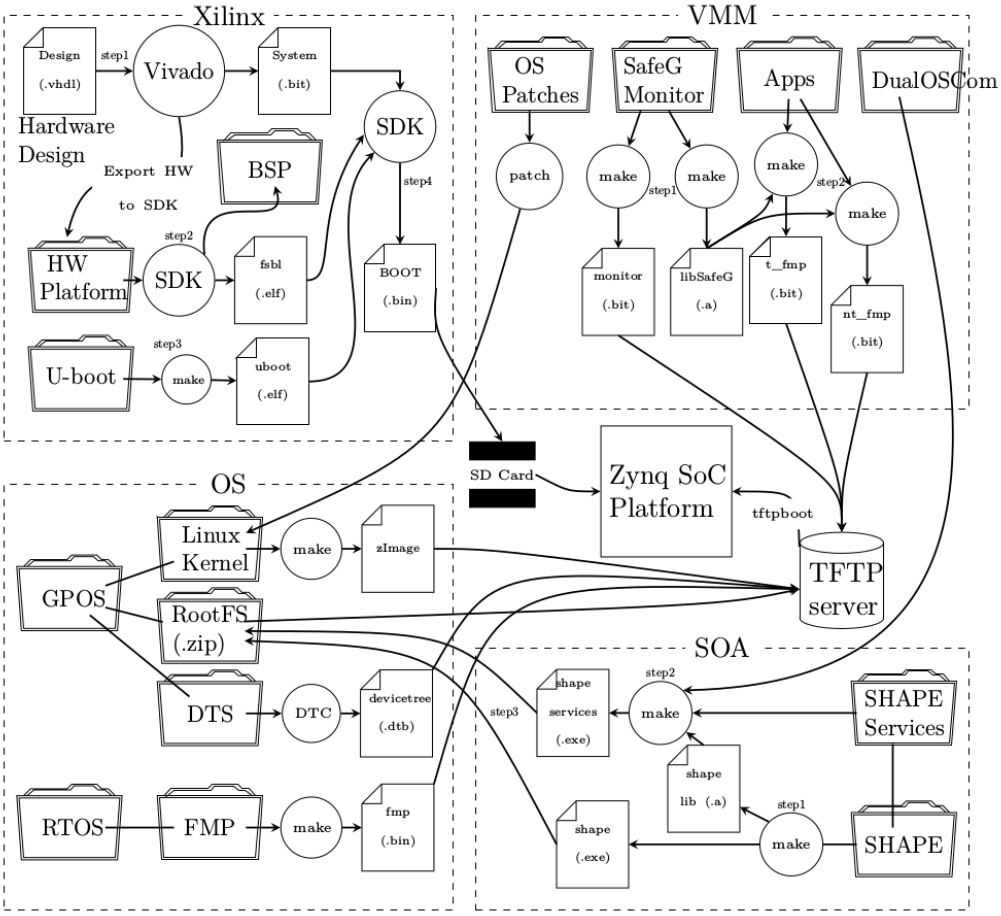


Figure 3.4: System build procedure. [48]

This build procedure setup provides for easy modifications to some small parts of the system without having to rebuild other parts, and the board to which the code should be loaded into only needs to be connected to a network socket and a laptop.

Chapter 4

System design

For the system to accomplish the goals stated in 1.2, several hardware and software functions need to be designed and implemented.

4.1 Operative system functions

This section will describe the functions to be implemented in the RTOS.

4.1.1 Longitudinal control

A controller will be implemented to control the speed of the vehicle and its distance to the vehicle in front of it. The controller will have three different modes, Cruise Control (CC), Adaptive Cruise Control (ACC), and Cooperative Adaptive Cruise Control (CACC). The CACC controller will use speed, distance to preceding vehicle and information communicated from the preceding vehicle as input to calculate a control signal. For an illustration of the controller architecture, see figure 4.1.

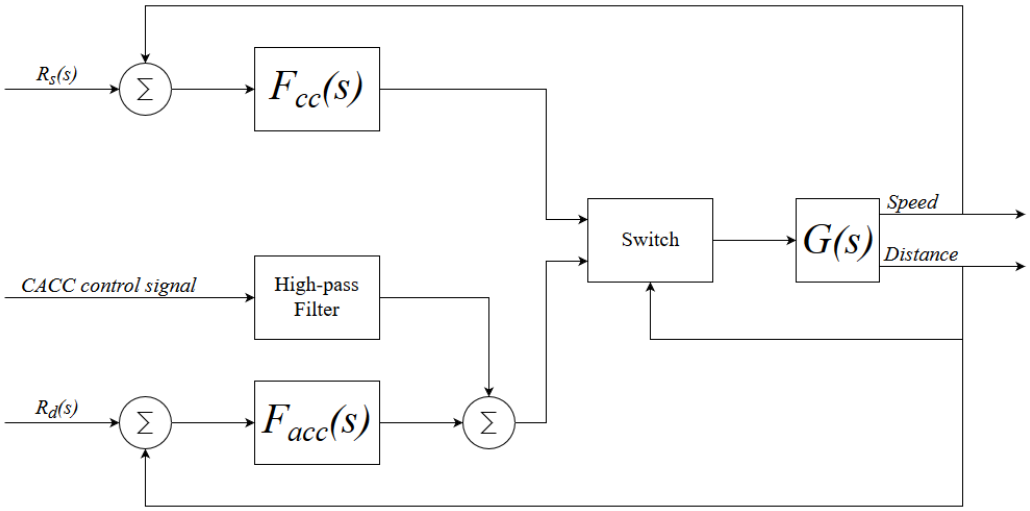


Figure 4.1: The architecture of the designed controller where F_{cc} is a PID controller controlling the speed, F_{acc} is a PID controller controlling the distance and G is the dynamics of the car.

For more information, see the report by Roshanghias [32].

4.1.2 Lateral control

A lane-detecting algorithm will be implemented on a Raspberry Pi that will send data to the Alten MCS. The choice of Raspberry Pi was made because of the amount of open source video processing code available for the platform. For more info, see the report by Ferhatovic [16].

Lateral control will be implemented on the RTOS on the Alten MCS. The function receives the deviation from the center line as calculated by the lane detection algorithm from the Raspberry Pi, and calculates a control signal from this.

4.1.3 Data aggregation

This function will read wheel sensor data and calculate if the road conditions are unsafe for platooning. See the report by Hellman [18].

The function will also serve as a filter for distance measurements and speed measurements for other tasks to use.

4.1.4 Communication

To maintain secure communication between the two vehicles in the platoon a communication protocol will be developed. For more information, see the report by [28].

4.2 Hardware implemented functions

This section will describe the functions to be implemented on the FPGA, also known as IPs.

4.2.1 Speed reading

To read the speed of the vehicle, encoders need to be connected to the wheels. The four encoders will send pulses very quickly and a hardware decoder is needed to process them and send the speed of each wheel to the OS.

4.2.2 Distance reading

To read the distance to the preceding vehicle, a LIDAR (LIght Detection And Ranging) will be used. The LIDAR communicates the distance it is reading via a PWM signal. A hardware function is needed to read the PWM pulse and convert it to a distance.

4.2.3 Communication

WiFi communication is used between the two vehicles in the platoon. To read packages, a WiFi module is used that communicates via UART with the Alten MCS. A MicroBlaze in the FPGA processes the packages and sends the parsed data to the OS via a mailbox. For more information, see the report by Lerander [28].

4.2.4 Actuation

To actuate control signals and thereby control the speed of the motor and the position of the steering servo, Pulse Width Modulation (PWM) is used. Two hardware implemented PWM functions will be needed for this.

4.2.5 Raspberry Pi communication

To receive information from the Raspberry Pi, some form of communication interface needs to be used. One simple solution for this is UART (Universal Asynchronous Receiver Transmitter) serial communication.

4.3 Overview design

An overview of the different functions to be implemented on the Zynq-7000 can be seen in figure 4.2.

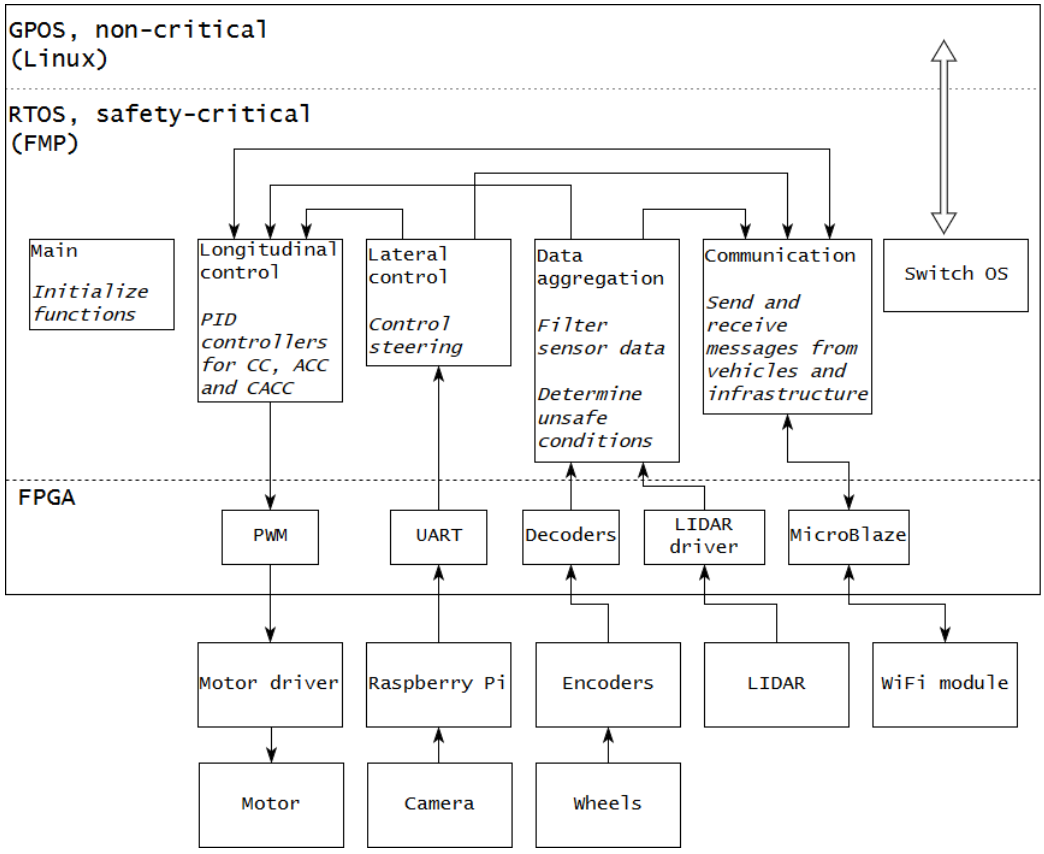


Figure 4.2: Overview of the different functions to be implemented. The arrows indicate information flow between the different functions.

A sequence diagram of the various dependencies and flow of each function can be seen in figure 4.3.

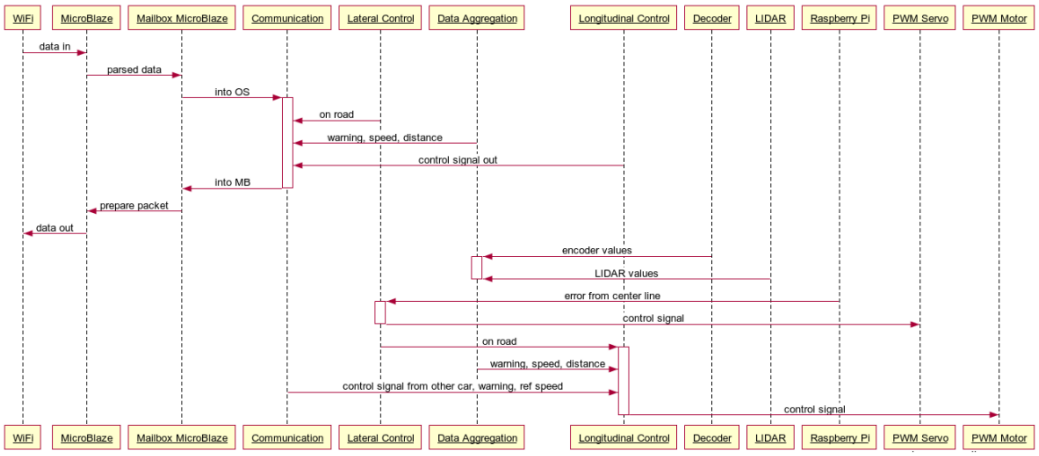


Figure 4.3: Sequence diagram of the various hardware and software functions, their dependencies and flow of information.

Chapter 5

Implementation

This chapter will describe the implementation of the entire system in the demonstrator.

5.1 Platform

To demonstrate the system, the RC car UTOR 8E from BSD racing was chosen. It is a 1/8 scale RC car with a steering servo, brushless DC motor, and most importantly, it was readily available from a local store.

For the demonstrator, it was decided to use the Zedboard on both vehicles for a number of reasons:

- The power supply connector for the EMC^{2}DP would have to be remade to be used in the vehicle.
- Connections on the vehicles fitting the measurements of the Zedboard were made.
- Same BSP for both vehicles.

The car with all of its components can be seen in figure 5.1.

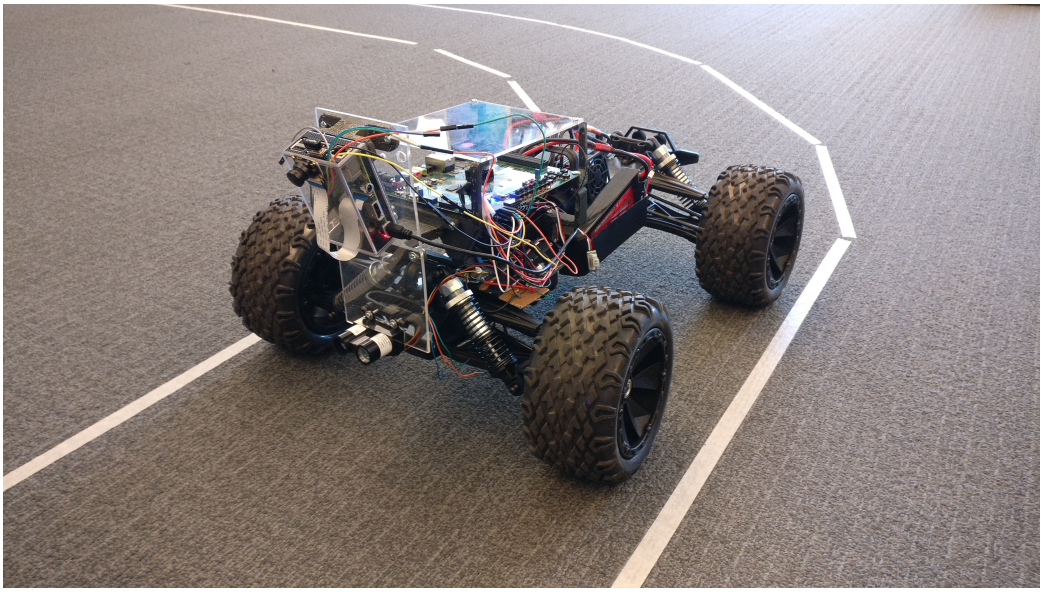


Figure 5.1: One of the two vehicles on the test track. Visible in the picture is the LIDAR in the front, the Raspberry Pi and its camera. The Zedboard can be seen under the covering plastic glass.

5.1.1 Driver

The driver for the motor and steering servo is controlled by a PWM signal with a frequency of 50 Hz. The speed is controlled by the pulse length of the PWM signal. This was tested with the hand-controller to see what PWM signals were received at the end points. Maximum and minimum high time measured was 2 ms and 1 ms, which at 50 Hz corresponds to a duty cycle of 10 and 5 %. Depending on how high frequencies the driver could handle, the frequency could be increased without changing the high time, potentially up until a period of 0.02 seconds would make the maximum signal correspond to a duty cycle of 100%.

This was tested, and gave somewhat expected results. The driver had some disturbances and gave maximum or minimum output sporadically when it shouldn't, with more disturbances at higher frequencies. It was decided that a frequency of 50 Hz would have to do.

5.2 Electrical wiring

The electrical power for all components on the car is sourced from the battery pack: two 7.4V batteries in parallel. Three voltage regulators are connected to the batteries to give power to other components.

- 7.4V to 5V for LIDAR, Encoders and Raspberry Pi.
- 7.4V to 5V for WiFi module.
- 7.4V for DC input voltage for Zedboard. (Here the voltage regulator only acts as a filter.)

The WiFi module was put separately from the other 5V output since it draws too much current to be combined with the other components for one voltage regulator to handle.

5.3 RTOS tasks

This section will describe the task implemented on the RTOS.

5.3.1 Data aggregation

The task `data_aggregation()` is the task with the highest frequency in the system. It executes with a period of 1 ms, which the shortest period allowed in FMP. It reads the speed from each wheel as measured by the decoder, and analyses if the road conditions are unsafe for platooning. It also filters speed and distance measurements through a low-pass filter to produce reliable data for the other tasks to use.

5.3.2 Communication

The task `communication()` reads V2V and I2V messages stored in a buffer on the FPGA. The messages are unpacked and forwarded to their intended destination task. Information from the other tasks are sent the to the FPGA. The task `communication()` is bounded by the period of the larger communication loop and the period of `lateral_control()` and `longitudinal_control()`. It can only send messages with a period of 800 ms. Since the received messages are stored in a buffer on the FPGA, the task only needs to execute before every execution of `lateral_control()` and `longitudinal_control()`. This puts its period at 20 ms.

5.3.3 Lateral control

The task `lateral_control()` receives angle and positional error of the vehicle in relation to the detected lane and computed a control signal via a PID controller. The task also sends lane detection status to `longitudinal_control()`. Due to the limitations by the PWM described earlier the control frequency was capped at 50 Hz, meaning a period of 20 ms.

5.3.4 Longitudinal control

Just like `lateral_control()`, the period of `longitudinal_control()` is 20 ms. The task consists of two PID controllers, one for CC and one for ACC (as described in chapter 4). The controller adds the control signal from the preceding vehicle to its own control signal if secure communication is established and the vehicles are in platoon/CACC mode. It also sends its control signal to the `communication()` task. If the lane detection status received from `lateral_control()` indicates that no lane is detected, `longitudinal_control()` stops the vehicle.

5.4 GPOS

Linux was run on the non-secure side. From the beginning there were ideas of hosting a video feed from the camera on the vehicle, but there was not nearly enough time to implement this. Instead, the Linux was running idle and was only used to navigate the file system to demonstrate that it was in fact up and running.

5.5 Processor scheduling

The above tasks with their required period and WCET as can be seen in chapter 6 resulted in the processor scheduling seen in figure 5.2.

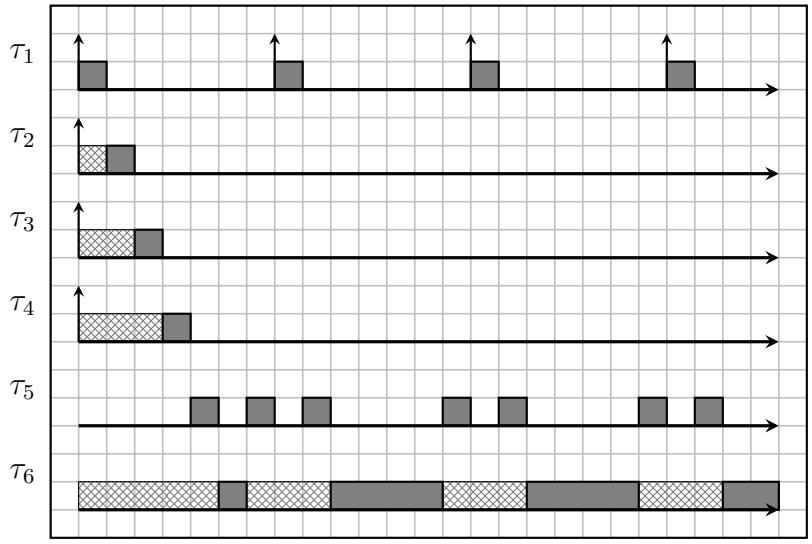


Figure 5.2: Processor scheduling, not to scale.
τ_1: Data aggregation τ_2: Communication τ_3: Lateral control
τ_4: Longitudinal control τ_5: OS switch τ_6: GPOS

5.6 Hardware functions

This section will describe the implementation of the hardware functions, or FPGA IPs. For the entire Vivado block design showing the FPGA IPs and their interconnect, see Appendix A.

5.6.1 Pulse Width Modulation

Two AXI Timer IPs were used to control the PWM signal to the motor and servo. Each AXI Timer has two internal timers, one timer setting the output high at the period specified, and one timer setting the output low after the start of the first timer, effectively creating a PWM signal. The resolution of the timers are 20 ns.

5.6.2 LIDAR

To read the distance to the preceding vehicle, Garmin LIDAR Lite v3 was used. Its technical specifications can be read in table 5.1.

Specification	**Measurement**
Power	5 Vdc nominal 4.5 Vdc min., 5.5 Vdc max.
Current consumption	105 mA idle 135 mA continuous operation
User interface	I2C PWM
Range	40 m
Resolution	±1 cm
Accuracy < 5 m	±2.5cm
Accuracy ≤ 5 m	±10 cm
Repetition rate	50 Hz default 500 Hz max

Table 5.1: Specifications for Garmin LIDAR Lite v3.

As can be seen in table 5.1, there are two interface options for the LIDAR, I2C and PWM. The PWM interface was chosen. A hardware function was implemented on the FPGA to read the pulse width from the PWM signal. The length of the pulse width was counted and written on an AXI bus address, ready for the OS to be read and converted into a distance in cm.

5.6.3 Encoders/Decoders

To read the speed of each wheel, encoders with a resolution of 1024 ppr were used. To convert the signals from each encoder, hardware implemented decoders were written. The decoders read A and B pulses from each encoder and counted the time between each pulse to calculate the rotational speed of the wheels. This value was written on an AXI bus address for the OS to read.

5.6.4 UART

To communicate with the Raspberry Pi, the UART interface was used. The IP UART Lite with a baudrate of 115200 was implemented on the FPGA.

5.6.5 MicroBlaze

To handle the communication protocol, a processor in the form of a MicroBlaze was implemented on the FPGA. For more information, see the report by Lerander [28].

Chapter 6

Results

This chapter will present results from the demonstrator to the reader.

6.1 Execution times

This section covers the execution times of the tasks in the RTOS. Execution times were measured using a hardware timer on the FPGA with a resolution of 20 ns. All times were measured on the EMC^{2}DP.

6.1.1 Data aggregation

Data aggregation is the task with the highest frequency and priority in the system. It executes with a period of 1 ms. Data for its execution times can be seen in table 6.1.

	Time	Clock cycles
WCET	4 μs	200
BCET	3.24 μs	162
Average ET	3.38 μs	169
Median ET	1.72 μs	167
Standard deviation	0.1 μs	5

Table 6.1: Execution times of data aggregation. Sample size: 6351

6.1.2 Communication

Communication is the task with the second highest priority. It executes with a period of 20 ms. Data for its execution times can be seen in table 6.2.

	Time	Clock cycles
WCET	6.32 μs	316
BCET	3 μs	150
Average ET	3.48 μs	174
Median ET	3.02 μs	151
Standard deviation	0.9 μs	48

Table 6.2: Execution times of communication. Sample size: 3851

6.1.3 Lateral control

Lateral control is the task with the third highest priority, and executes with a period of 20 ms. It contains a timeout of 0.1 ms in a while loop as it waits for UART communication from the Raspberry Pi, which greatly affects its WCET. The execution times were measured when there was no communication via UART. Data for the tasks execution times can be seen in table 6.3.

	Time	Clock cycles
WCET	104 μs	5219
BCET	101 μs	5061
Average ET	103 μs	5155
Median ET	104 μs	5200
Standard deviation	1.32 μs	66

Table 6.3: Execution times of lateral control. Sample size: 878

6.1.4 Longitudinal control

Longitudinal control is the task with the lowest priority in the RTOS. It executes with a period of 20 ms. Data for the tasks execution times can be seen in table 6.4.

	Time	Clock cycles
WCET	4.6 μs	230
BCET	3.62 μs	181
Average ET	3.74 μs	187
Median ET	3.7 μs	185
Standard deviation	0.14 μs	7

Table 6.4: Execution times of longitudinal control. Sample size: 4050

6.2 SafeG overhead

To measure the overhead of SafeG monitor, a hardware timer was started on the secure side, immediately followed by a syscall to switch OS. The non-secure side immediately read the value of the timer and printed this value. This procedure was looped many times to gather a reliable set of data points. Pseudo code for this procedure follows below:

Secure side:

loop
  $start\_timer()$
  $safeg\_syscall\_switch()$
end loop

Non-secure side:

loop
  $time \leftarrow get\_time()$
  $reset\_timer()$
  $print(time)$
  $safeg\_syscall\_switch()$
end loop

This resulted in a data set of 103211 execution times, see table 6.5.

	Time	Clock cycles
WCET	3.06 μs	153
BCET	0.96 μs	48
Average ET	1.63 μs	81
Median ET	1.72 μs	86
Standard deviation	0.258 μs	13

Table 6.5: Execution times of OS switch. Sample size: 103211

The measured execution times for the OS switch are also visualized in figure 6.1.

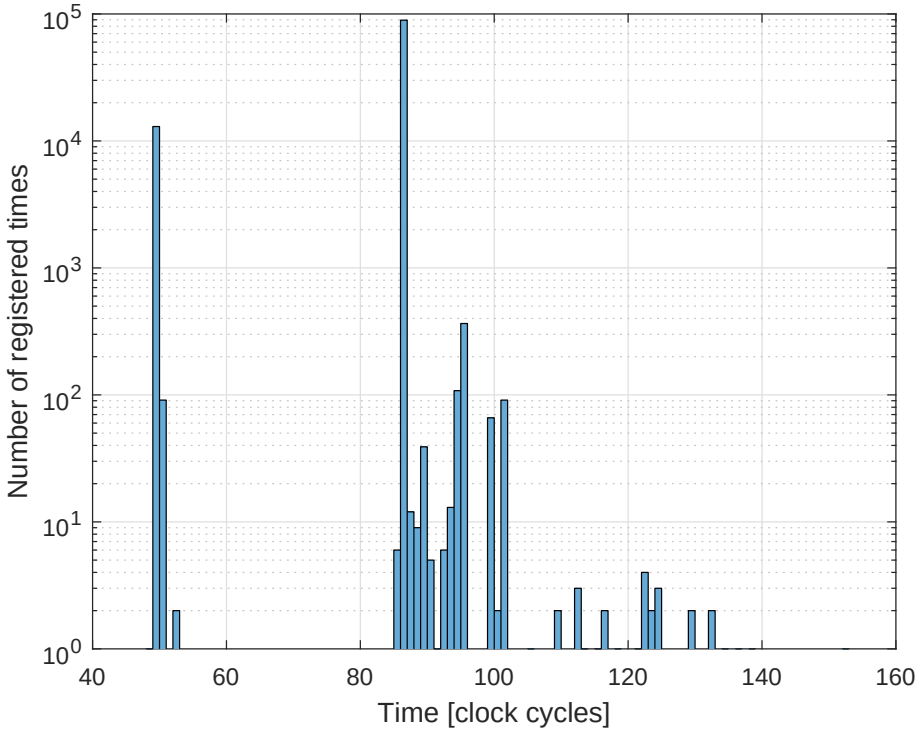


Figure 6.1: Visualization of execution times for the OS switch. Note the logarithmic y-axis.

This results in an overhead of 0.6% for the entire system. The potential useful CPU utilization is 99.4%, providing that the GPOS uses all of its allocated CPU time. Using only the RTOS without the hypervisor the CPU utilization would be 0.005%.

Theoretically, if FMP would allow for shorter periods for a task, the maximum overhead for this system would be 24%.

6.3 Hypervisor robustness

To test the robustness of the hypervisor, a fork bomb was made on the GPOS. A fork bomb is a function that calls itself two times. This quickly depletes the system of its resources, causing it to crash eventually [43]. The test resulted in the GPOS crashing and becoming non-responsive while the RTOS managed to maintain all its functionality.

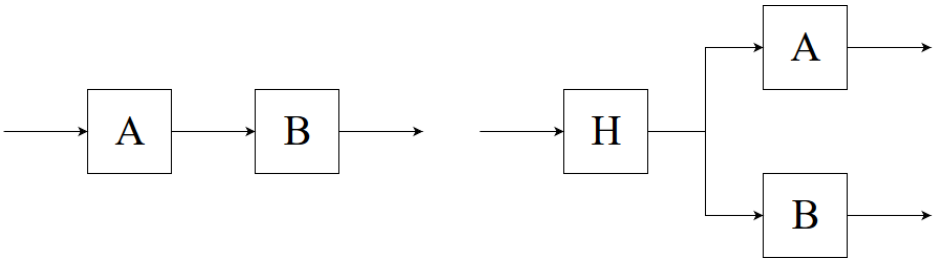
Chapter 7

Discussion

Discussion about the results produced by the thesis.

7.1 Robustness

It is important to know that a hypervisor can never increase robustness of a single OS, only make it more isolated from errors from another OS. Potentially, the hypervisor itself can become the source of an error causing the collective system to fail, see figure 7.1.



(a) Integrity of A is dependent on integrity of B.

(b) Integrity of A is isolated from B, but is instead dependent on H.

Figure 7.1

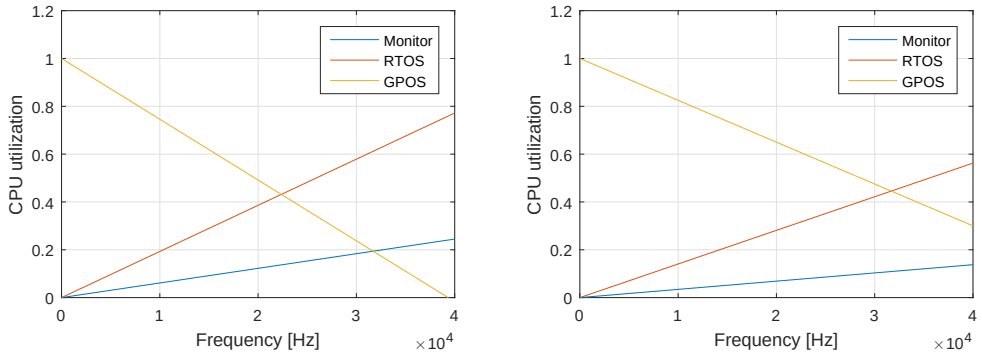
Keeping this in mind is important when discussing the robustness of a MCS.

7.2 Information retrieval

The hardware timer used when measuring execution times requires very little processor time, but still some. This adds an ever so slight error to the execution times. The data was deemed to be of good enough quality anyway.

7.3 Virtualization resource gain vs resource loss

As described earlier, higher frequencies of RTOS tasks leads to more overhead. Disregarding the maximum frequency of 1 kHz for tasks in FMP, the maximum frequency for the functions in the implemented system would be just under 40 kHz, leading to an overhead of 24% for the hypervisor. This gives the GPOS no time at all for the case when every other task executes for its WCET. However, assuming every task executes for its median execution time, the GPOS can execute for 31% of the time at 40 kHz. This gives a very good combination of throughput for the entire system and real-time guarantees for the safety-critical tasks at the cost of a maximum overhead of 24%. This is visualized in figure 7.2.



(a) Worst case execution time

(b) Median execution time

Figure 7.2: CPU utilization of monitor, RTOS and GPOS. Note that for the WCET of tasks, 100% CPU utilization by the monitor and RTOS is reached just before 40 kHz.

7.4 Conclusion

In the introduction of this thesis, the following research question was presented:

- Is virtualization an efficient approach when trying to reconcile the conflicting requirements of partitioning for safety assurance and sharing for efficient resource usage when implementing a safety-critical control system?

After the implementation, it can be clearly seen that virtualization can be a very efficient and secure approach for reconciling the conflicting requirements of partitioning and sharing. However, finding out if it is efficient in the industry as a whole requires additional research in technical aspects as well as other aspects such as management and financial related ones. These are presented in chapter 8.

Chapter 8

Future work

This chapter will contain thoughts and ideas for future work building on this thesis or in the area of MCS in general.

8.1 MCS using virtualization

An interesting continuation on this thesis would be to facilitate for more than two different criticality levels. For example, the FPGA could host applications of a third criticality level. To accommodate for more criticality levels on the processor, another hypervisor would need to be used.

Another interesting continuation would be to examine different scheduling methods, such as AMC described in chapter 2.

8.2 MCS using other means of partitioning

The most explored area regarding mixed criticality research is sharing processor between different criticalities. Something that is not yet as explored is sharing other resources such as memory. It would be interesting to examine the limitations for other configurations of MCS.

8.3 Amount of criticality levels

Research should be done to investigate how many different levels of criticality, n, to facilitate for on MCS in different industries. In the automotive for example, n should be between 1 and 5 since ISO26262 defines 5 different ASILs. If the applications in a car are spread uniformly across all criticality

levels it might be of higher interest to have n closer to 5. Similarly, if the applications are heavily concentrated on a certain criticality level, n probably should be closer to 2.

8.4 Economical benefits for pursuing MCS

It is not clear how much the potential economical benefit would be from pursuing MCS. The economical impacts of MCS might be different in different industries. It must be calculated more exactly how large the potential benefits would be to gauge the need for pursuing MCS.

8.5 Small fixes

This section will contain fixes that should be relatively easy to complete, but were omitted due to timing limitations.

- Utilize dual core for RTOS: In the demonstrator, the second core in the CPU was never utilized. Only the GPOS used both cores. Using the second core in the RTOS makes processor scheduling more complex.
- Data aggregation filtering in the FPGA: Moving data aggregation filtering to the FPGA would mean that the monitor overhead would reduce by a factor of 20 (even though it already is very low).
- Electrical wiring: In the demonstrator there is currently no circuit board for the electronics. Implementing this would increase the physical robustness of the system greatly.
- Utilizing the GPOS for something useful: As mentioned earlier, there were plans on hosting a video stream from the camera on the vehicle on the Linux. This was scrapped due to lack of time. It would serve great purpose for the demonstration of the system to have a visible application running on Linux.

Bibliography

[1] Assad Alam, Ather Gattami, and Karl H. Johansson. An experimental study on the fuel reduction potential of heavy duty vehicle platooning. In *13th International IEEE Conference on Intelligent Transportation Systems*, pages 306–311, September 2010.

[2] Assad Alam, Ather Gattami, Karl H. Johansson, and Claire J. Tomlin. Guaranteeing safety for heavy duty vehicle platooning: Safe set computations and experimental evaluations. *Control Engineering Practice*, November 2013.

[3] ARM Ltd. ARM TrustZone, February 2017. `https://www.arm.com/products/security-on-arm/trustzone`.

[4] Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, February 2015.

[5] AUTOSAR. AUTOSAR Homepage, February 2017. `http://www.autosar.org/`.

[6] Luís Silva Azevedo, David Parker, Yiannis Papadopoulos, Martin Walker, Ioannis Sorokos, and Rui Esteves Araújo. *Exploring the Impact of Different Cost Heuristics in the Allocation of Safety Integrity Levels*, pages 70–81. Springer International Publishing, Cham, 2014.

[7] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43, Nov 2011.

[8] Sanjoy Baruah and Steve Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications, 2008.

[9] Carl Bergenhem, Henrik Pettersson, Erik Coelingh, Cristofer Englund, Steven Shladover, and Sadayuki Tsugawa. Overview of platooning systems. In *Proceedings of the 19th ITS World Congress*, Vienna, Austria, October 2012.

[10] Alan Burns and Robert I. Davis. Mixed criticality systems - a review. Available online: https://www-users.cs.york.ac.uk/burns/review.pdf, July 2016.

[11] Data61. The sel4 microkernel, May 2017. http://sel4.systems/Info/Docs/GD-NICTA-whitepaper.pdf.

[12] Dictionary.com. safety-critical system, January 2017. http://www.dictionary.com/browse/safety-critical-system.

[13] Viktor Do. *SICS Thin Hypervisor Reference Manual Version 0.4*, April 2013.

[14] Software considerations in airborne systems and equipment certification. Standard, Radio Technical Commission for Aeronautics, Washington, DC, USA, December 2011.

[15] Encyclopedia.com. safety-critical system, January 2017. http://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/safety-critical-system.

[16] Sanel Ferhatovic. Comparative study on road and lane detection in mixed criticality embedded systems. Master's thesis, KTH, Royal Institute of Technology, Sweden, June 2017.

[17] Google. Google trends. https://trends.google.com/trends/.

[18] Hanna Hellman. Data aggregation in time sensitive multi-sensor systems. Master's thesis, KTH, Royal Institute of Technology, Sweden, June 2017.

[19] Anne Håkansson. Portal of research methods and methodologies for research projects and degree projects, July 2013.

[20] Functional safety of electrical/electronic/programmable electronic safety-related systems – parts 1 to 7. Standard, International Electrotechnical Commission, Geneva, CH, April 2010.

[21] IEC/IEEE International Standard - Behavioural languages - Part 1-1: VHDL Language Reference Manual. Standard, IEEE, May 2011.

[22] IEC/IEEE Behavioural Languages - Part 4: Verilog Hardware Description Language (Adoption of IEEE Std 1364-2001). Standard, IEEE, 2004.

[23] Road vehicles – functional safety – part 9: Automotive safety integrity level (asil)-oriented and safety-oriented analyses. Standard, International Organization for Standardization, Geneva, CH, November 2011.

[24] ITRON Committee, TRON Association. uITRON4.0 Specification Ver. 4.00.00. `http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf`.

[25] Leonard Kleinrock. Analysis of a time-shared processor. *Naval Research Logistics Quarterly, 11:1*, March 1964.

[26] Michael P. Lammert, Adam Duran, Jeremy Diez, Kevin Burton, and Alex Nicholson. Effect of platooning on fuel consumption of class 8 vehicles over a range of speeds, following distances, and mass. *SAE Int. J. Commer. Veh.*, 7:626–639, 09 2014.

[27] Max Lemke et al. Mixed criticality systems. report from the workshop on mixed criticality systems, February 2012.

[28] Erik Lerander. Communication between mixed criticality systems. Master's thesis, KTH, Royal Institute of Technology, Sweden, June 2017.

[29] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM Volume 20 Issue 1*, pages 46–61, January 1973.

[30] Anna Lyons and Gernot Heiser. Mixed-criticality support in a high-assurance, general-purpose microkernel. 2014. `https://www-users.cs.york.ac.uk/~robdavis/wmc2014/2.pdf`.

[31] Daniel Potts, Rene Bourquin, Leslie Andresen, June Andronick, Gerwin Klein, and Gernot Heiser. Mathematically verified software kernels: Raising the bar for high assurance implementations, July 2014. `http://sel4.systems/`.

[32] Daniel Roshanghias. Evaluation and implementation of a cooperative adaptive cruise controller in a safety critical system. Master's thesis, KTH, Royal Institute of Technology, Sweden, June 2017.

[33] SafeCOP. SafeCOP - part B, October 2016.

[34] Daniel Sangorrin Lopez. Advanced integration techniques for highly reliable dual-os embedded systems, July 2012.

[35] Daniel Sangorrín, Shinya Honda, and Hiroaki Takada. Dual operating system architecture for real-time embedded systems, Jul 2010. http://www.artist-embedded.org/docs/Events/2010/OSPERT/slides/1-1_NU_OSPERT2010.pdf.

[36] Scania. Platooning saves up to 12 percent fuel, December 2015.

[37] Sierraware. Sierravisor, May 2017. https://www.sierraware.com/arm_hypervisor.html.

[38] TOPPERS Project, Inc. Introduction to the TOPPERS/FMP kernel, February 2017. https://www.toppers.jp/en/fmp-kernel.html.

[39] TOPPERS Project, Inc. TOPPERS SafeG, February 2017. https://www.toppers.jp/en/safeg.html.

[40] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, 2007.

[41] W. Weber, A. Hoess, F. Oppenheimer, B. Koppenhöefer, B. Vissers, and B. Nordmoen. EMC2 a Platform Project on Embedded Microcontrollers in Applications of Mobility, Industry and the Internet of Things. In *2015 Euromicro Conference on Digital System Design*, pages 125–130, August 2015.

[42] Werner Weber. A Platform Project on Embedded Microcontrollers in Applications of Mobility, Industry and the Internet of Things, Mars 2015. https://artemis-ia.eu/publication/download/1131.pdf.

[43] Wikipedia. Fork bomb, June 2017. https://en.wikipedia.org/wiki/Fork_bomb.

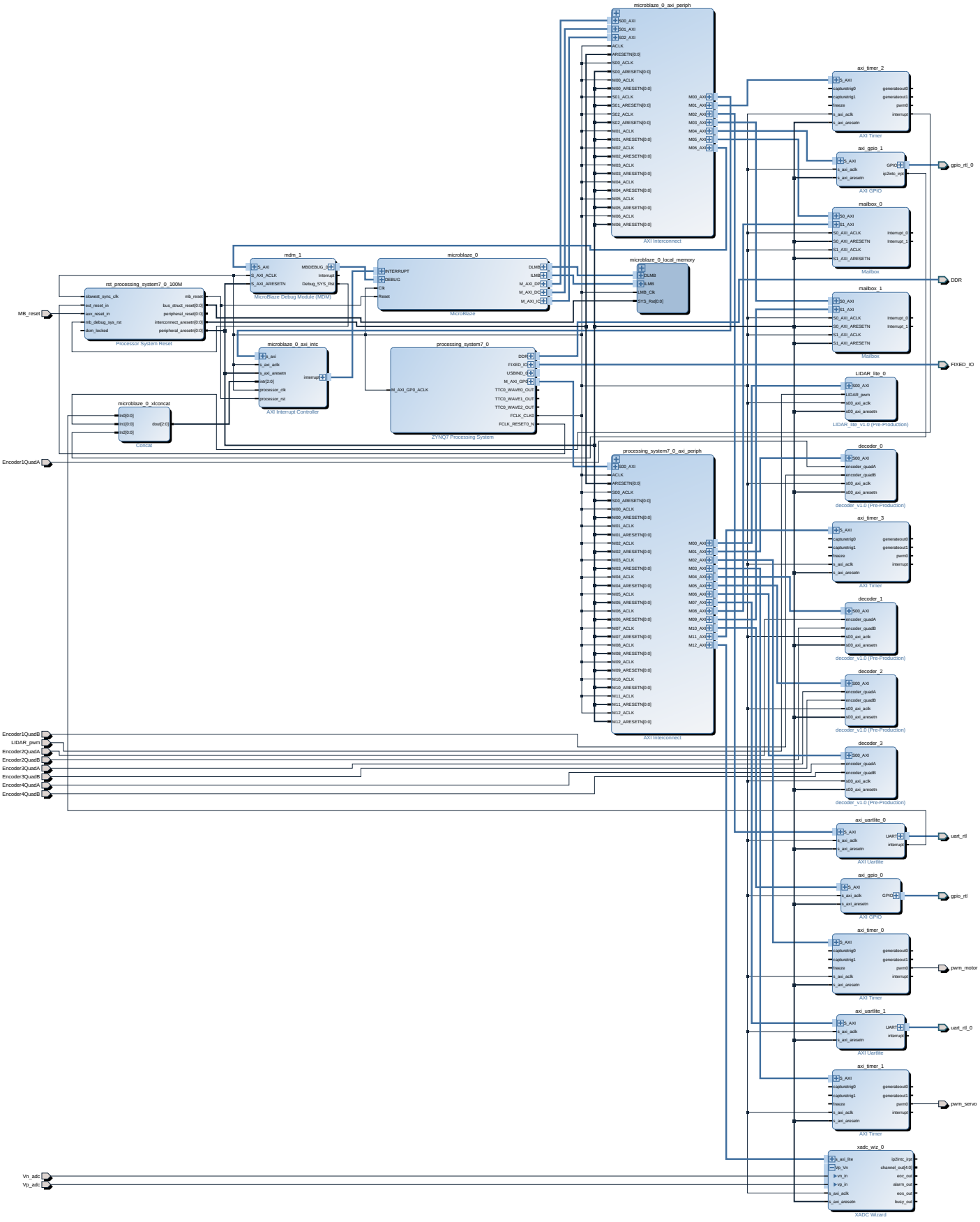
[44] Wikipedia.com. Life-critical system, January 2017. https://en.wikipedia.org/wiki/Life-critical_system.

[45] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing System.*

[46] Xilinx Inc. Vivado Design Suite, March 2017. https://www.xilinx.com/products/design-tools/vivado.html.

[47] Xilinx Inc. Zynq-7000 All Programmable SoC, February 2017. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[48] Youssef Zaki. An embedded multi-core platform for mixed-criticality systems: Study and analysis of virtualization techniques. Master's thesis, KTH, School of Information and Communication Technology (ICT), 2016.


microblaze_0_axi_periph
S00_AXI
S01_AXI
S02_AXI
ACLK
ARESETN[0:0]
M00_AXI
M01_AXI
M02_AXI
M03_AXI
M04_AXI
M05_AXI
M06_AXI
AXI Interconnect
axi_timer_2
S_AXI
capturetrig0
capturetrig1
freeze
s_axi_aclk
s_axi_aresetn
generateout0
generateout1
pwm0
interrupt
AXI Timer
axi_gpio_1
GPIO
ip2intc_irpt
AXI GPIO
gpio_rtl_0
mailbox_0
S0_AXI
S1_AXI
S0_AXI_ACLK
S0_AXI_ARESETN
S1_AXI_ACLK
S1_AXI_ARESETN
Interrupt_0
Interrupt_1
Mailbox
DDR
mailbox_1
FIXED_IO
LIDAR_lite_0
S00_AXI
LIDAR_pwm
s00_axi_aclk
s00_axi_aresetn
LIDAR_lite_v1.0 (Pre-Production)
mdm_1
S_AXI
S_AXI_ACLK
S_AXI_ARESETN
MBDEBUG_0
Interrupt
Debug_SYS_Rst
MicroBlaze Debug Module (MDM)
microblaze_0
INTERRUPT
DEBUG
Clk
Reset
DLMB
ILMB
M_AXI_DP
M_AXI_DC
M_AXI_IC
MicroBlaze
microblaze_0_local_memory
DLMB
ILMB
LMB_Clk
SYS_Rst[0:0]
rst_processing_system7_0_100M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
MB_reset
microblaze_0_axi_intc
s_axi
s_axi_aclk
s_axi_aresetn
intr[2:0]
processor_clk
processor_rst
interrupt
AXI Interrupt Controller
processing_system7_0
DDR
FIXED_IO
USBIND_0
M_AXI_GP0
TTC0_WAVE0_OUT
TTC0_WAVE1_OUT
TTC0_WAVE2_OUT
FCLK_CLK0
FCLK_RESET0_N
M_AXI_GP0_ACLK
ZYNQ7 Processing System
microblaze_0_xlconcat
In0[0:0]
In1[0:0]
In2[0:0]
dout[2:0]
Concat
Encoder1QuadA
processing_system7_0_axi_periph
M07_AXI
M08_AXI
M09_AXI
M10_AXI
M11_AXI
M12_AXI
decoder_0
encoder_quadA
encoder_quadB
decoder_v1.0 (Pre-Production)
axi_timer_3
decoder_1
decoder_2
decoder_3
Encoder1QuadB
LIDAR_pwm
Encoder2QuadA
Encoder2QuadB
Encoder3QuadA
Encoder3QuadB
Encoder4QuadA
Encoder4QuadB
axi_uartlite_0
UART
axi_uartlite_0
uart_rtl
axi_gpio_0
gpio_rtl
axi_timer_0
pwm_motor
axi_uartlite_1
uart_rtl_0
axi_timer_1
pwm_servo
xadc_wiz_0
s_axi_lite
vp_Vn
vn_in
vp_in
ip2intc_irpt
channel_out[4:0]
eoc_out
alarm_out
eos_out
busy_out
XADC Wizard
Vn_adc
Vp_adc