Reliable Software Development aided by QEMU Simulation

Rui Almeida

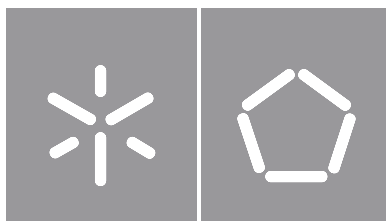
UMinho | 2020

Universidade do Minho
Escola de Engenharia

Rui Jorge Mendes Almeida

Reliable Software Development aided by QEMU Simulation

novembro de 2020



Universidade do Minho
Escola de Engenharia

Rui Jorge Mendes Almeida

Reliable Software Development aided by QEMU Simulation

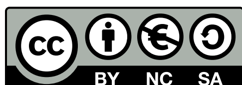
Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do(a)
Professor Doutor Jorge Cabral

novembro de 2020

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.
Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.
Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.



Atribuição-NãoComercial-CompartilhaIgual

CC BY-NC-SA

https://creativecommons.org/licenses/by-nc-sa/4.0/

Agradecimentos

"Em primeiro lugar, gostaria de expressar a minha gratidão aos meus pais por todo o suporte emocional e financeiro que me permitiram frequentar e agora finalizar um curso superior. Por todo o esforço que me permitiu chegar aqui, fico para sempre agradecido.

Agradeço ao meu orientador, doutor Jorge Cabral, por todo o conhecimento transmitido durante este projeto. Um especial agradecimento aos Engenheiros Nelson Naia e Luís Novais por todo o suporte neste esforço de engenharia e por por todo o apoio mental e moral durante este percurso. Ainda no âmbito académico, agradeço ao João Carvalho pelo apoio nas áreas de conhecimento mais "exóticas".

Aos colegas de laboratório do *Embedded Systems Research Group*, Pedro Lobo e Luís Vale, que me acompanharam desde o início desta jornada, agradeço pelo suporte e por todos os momentos de descontração.

Por fim, agradeço à malta da *Phoenix* pela companhia e toda a alegria proporcionada nos momentos de descompressão.

A todos que me ajudaram neste percurso, o meu maior obrigado."

This work is supported by: European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project n° 037902; Funding Reference: POCI-01-0247-FEDER-037902].

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Desenvolvimento de Software Confiável assistido por Simulação em QEMU

Sistemas altamente confiáveis asseguram uma baixa probabilidade de falha por meio de redundância, a qual garante a funcionalidade do sistema replicando componentes ou módulos. Estes módulos interagem entre eles, tomando decisões sobre o estado do sistema e, por esse motivo, tanto os mecanismos de redundância como as interações entre módulos devem ser validados para garantir a gestão correta de redundância. A utilização de um ambiente de co-simulação que consegue replicar todos os módulos e a comunicação entre eles permite validar tais interações antes do *deployment*, pois não está pendente de nenhum recurso de hardware. Além disso, a adoção da co-simulação permite um desenvolvimento mais rápido ao mesmo tempo em que, regra geral, auxilia na detecção de problemas no início do ciclo de desenvolvimento, evitando possíveis problemas que se manifestam tarde no ciclo de desenvolvimento. O uso de simulação também habilita estimações de confiabilidade do sistema, garantindo que as métricas de confiabilidade sejam cumpridas ao longo do ciclo de desenvolvimento e prevenindo reiterações tardias. Embora essas sejam grandes vantagens, elas acarretam um desafio de simulação, uma vez que a maioria dos simuladores não contemplam cenários de redundância.

O objetivo desta dissertação é auxiliar no desenvolvimento de sistemas confiáveis, adotando uma abordagem de simulação e estendendo as funcionalidades do simulador para cobrir o caso de uso de redundância. Usando QEMU (Quick Emulator) para emular o comportamento do sistema, três extensões foram conceptualizadas e desenvolvidas para permitir a validação correta de sistemas redundante e estimativas de confiabilidade por meio de simulação. O ambiente de simulação resultante auxiliou no desenvolvimento de um estudo de caso que se encaixa no conceito *Steer by Wire*. O sistema desenvolvido resultou numa configuração tolerante a falhas com características de redundância homogênea. A partir do uso das extensões, o software do sistema resultante pôde ser validado antes de qualquer implantação de hardware e permitiu ober uma estimativa do tempo antes da falha do sistema.

Palavras-chave: co-simulação, design e estimação de confiabilidade, QEMU, redundância

Abstract

Reliable Software Development aided by QEMU Simulation

Highly reliable systems guarantee low system failure probability during its operational lifetime with the help of redundancy, which ensures system functionalities by replicating components or modules. Such modules interact with each other allowing to make decisions about the system state, and for that reason both the redundancy mechanisms and interactions between modules need to be validated to ensure correct redundancy management. The usage of a co-simulation environment that can replicate all the modules and communications between them allows to validate interactions before deployment, since it is not bound to any hardware resource. Additionally, the adoption of co-simulation allows for faster development while assisting on problem detection early on the development cycle, avoiding possible late design problems. The usage of simulation also enables early system reliability evaluations, ensuring that reliability metrics are fulfilled throughout the development cycle and preventing design reiterations later on the development cycle. Although these are great advantages, it brings a simulation challenge since most full development board simulators do not contemplate such redundancy scenarios on their tools.

The aim of this dissertation is to assist reliable system development by adopting a simulation approach and extending simulator functionalities to cover the redundancy use case. Using QEMU (Quick Emulator) as the simulation tool to emulate system behaviour, three extensions were conceptualized and developed to cover features to allow for both correct redundant system validation and reliability estimations, supported by fault injection, through simulation.

The resulting simulation environment assisted the development of a case study that fits under the Steer by Wire concept. The developed system resulted in a fault tolerant configuration with homogeneous redundancy characteristics. From the usage of the extensions, the resulting system software could be validated for both its algorithms and redundancy management before any hardware deployment, and it allowed for an early time to failure estimation during the design phase.

Keywords: reliability design and estimation, co-simulation, QEMU, redundancy

Table of Contents

1 Introduction **1**

1.1 Motivation and Objectives . . . 2

1.2 Dissertation Structure . . . 3

2 State of the art **4**

2.1 Embedded Systems . . . 4

2.1.1 Definition . . . 5

2.1.2 Embedded Development . . . 5

2.1.3 Development Flow . . . 7

2.2 Operating Systems . . . 8

2.2.1 Concurrency . . . 9

2.2.2 Inter-Process Communication . . . 13

2.3 Reliability-oriented Systems . . . 14

2.3.1 Fault, Error, and Failure . . . 14

2.3.2 Dependability . . . 16

2.3.3 Reliability Metrics . . . 17

2.3.4 Failure Distribution . . . 18

2.3.5 Concepts of Redundancy and Fault Tolerance . . . 19

Redundancy . . . 20

Fault Tolerant Architectures . . . 23

2.3.6 Reliability System Development . . . 24

2.3.7 Reliable Software Development . . . 27

AUTOSAR . . . 28

2.4 Monte Carlo Simulation . . . 29

2.4.1 Bagging and Boosting . . . 31

2.5 Fault Injection . . . 33

2.6 Embedded System Simulation . . . 36
2.6.1 X-in-the-Loop . . . 37
2.6.2 Full System Hardware Simulation . . . 39
2.6.3 Hardware Simulation with Host Software . . . 39
2.6.4 Full System Software Simulation . . . 40
2.7 Co-simulation . . . 41
2.7.1 Synchronization . . . 44
2.7.2 Functional Mock-up Interface . . . 46
FMI for Model Exchange . . . 47
FMU for Co-Simulation . . . 48
2.7.3 FMI usage in the industry . . . 50
2.8 QEMU . . . 50
2.8.1 Binary Translation . . . 51
2.8.2 Deadlines and Emulation Time . . . 54
2.8.3 Translation Block Execution . . . 54
2.8.4 Device Model . . . 55
2.8.5 Using QEMU on research . . . 57
2.9 Summary . . . 57

3 Simulation Extensions for Reliability Development **58**
3.1 Synchronization between redundant subsystems . . . 60
3.1.1 Synchronization Process . . . 61
3.1.2 QEMU's Internal Synchronization . . . 62
3.2 Shared Bus Extension . . . 66
3.2.1 Extension Overview . . . 67
3.2.2 Extension Operations . . . 68
3.2.3 Extension Interface . . . 71
3.3 Fault Injection Extension . . . 74
3.3.1 Fault Injection Components . . . 76
3.3.2 Fault Types . . . 77
3.3.3 Fault Description XML file . . . 80
3.3.4 Fault Injection Coordinator . . . 81

3.3.5 Fault Injection Interface . . . 83
3.4 Summary . . . 87

4 Case Study **88**
4.1 Steering Angle Sensor . . . 88
4.1.1 Application Modeling . . . 91
4.1.2 Architecture Modelling . . . 91
4.1.3 Platform Decision . . . 96
4.1.4 Software Architecture . . . 96
SAS Application . . . 100
4.2 Application simulation using QEMU . . . 102
4.2.1 Target machine on QEMU . . . 102
4.2.2 Validation of the SAS application . . . 106
4.3 Reliability Estimation . . . 109
4.3.1 Fault Modeling . . . 110
4.3.2 Simulation Environment Wrapper . . . 111
4.4 Simulation Results . . . 112
4.5 Deployment on the hardware platform . . . 117

5 Conclusion **118**
5.1 Future Work . . . 119

A State of the Art Help Material **128**
A.1 Relationships between Reliability Functions . . . 128
A.2 AUTOSAR Coding Guidelines Example . . . 129
A.3 Embedded development Flow . . . 130

B Developed Simulation Extensions Diagrams **131**
B.1 Shared Bus Diagrams . . . 131
B.2 Fault Injection Diagrams . . . 132
B.2.1 Fault Injection Functions Flowcharts . . . 133

C Steering Angle Sensor Development **137**

C.1 Application Validation Diagram 137

C.2 State Machine Diagram 138

C.3 MCAL Module Diagrams 139

C.3.1 Class Diagrams 139

C.3.2 Use Case Diagrams 144

C.3.3 Sequence Diagrams 147

C.4 Software Flowcharts 152

D QEMU Machine Additional Material **159**

D.1 Addition of command line arguments 159

D.2 S32K116 Machine Class Diagram 160

D.3 Peripheral Example (ADC) 162

D.4 S32K116 Machine 165

D.5 Makefile changes 175

D.6 Full Command Line Arguments 176

E Monte Carlo Simulations **177**

E.1 Component Mean Time Between Failure Values 177

E.2 Fault Coordinator Python Script 177

List of Figures

2.1 Embedded platform block diagram 6
2.2 Embedded systems development flow 8
2.3 Layers of an OS 9
2.4 Concurrency between tasks 10
2.5 Process and thread memory block diagram 11
2.6 Virtual memory overview 12
2.7 Exchanging data between sockets 13
2.8 Fault, error and failure diagram 15
2.9 Reliability probability function distribution diagram 17
2.10 Example of a failure distribution of a hardware component [1] 19
2.11 Duplication with comparison 21
2.12 Triple Module Redundancy 21
2.13 Architecture of B777 Flight Control Computer 24
2.14 Example of development cycle of a reliable system 25
2.15 Design phase of the development cycle of a reliable system 25
2.16 Verification phase of the development cycle of a reliable system 26
2.17 AUTOSAR layered architecture 29
2.18 Monte Carlo method process 30
2.19 Bagging method diagram 32
2.20 Boosting method diagram 33
2.21 Typical fault injector architecture 36
2.22 X-in-the-loop testing applied on reliability-aware development cycle 37
2.23 Simulation of a system using an hardware simulator 39
2.24 Hardware simulator with software on host 40
2.25 Full system software simulation 41
2.26 Domains of a complex system 42

2.27 Simulation of a single domain using local cache 43
2.28 Exemplification of a causality error 45
2.29 FMU instance interface 46
2.30 FMI simulation type support 47
2.31 FMU for Model Exchange interface 47
2.32 Example of three connected FMU instances 48
2.33 Co-simulation with generated code on a single computer 48
2.34 Co-simulation with tool coupling on a single computer 48
2.35 Distributed co-simulation infrastructure 49
2.36 Simulation control through master-slave interface 49
2.37 FMU for Co-simulation interface 50
2.38 QEMU translation overview 52
2.39 Dynamic translation diagram 53
2.40 Translation Block chaining diagram 53
2.41 Deterministic translation block execution 55
2.42 Full system emulation overview diagram 56

3.1 Redundant architecture diagram 59
3.2 Redundant architecture conceptualized on QEMU diagram 60
3.3 Synchronization timely sequence diagram 61
3.4 Co-simulation environment example 62
3.5 Synchronization of QEMU instances sequence diagram 63
3.6 Synchronization process flowchart 63
3.7 Translation block execution thread simplified flowchart 64
3.8 Budgeted translation block execution flowchart 65
3.9 Example of a co-simulation environment with the Shared Bus extension 66
3.10 Shared Bus extension conceptualization diagram 68
3.11 Communication between Shared Bus socket pairs 69
3.12 QEMU connection to the Shared Bus 70
3.13 Write operation on the Shared Bus 70
3.14 Synchronous read operation on the Shared Bus 71
3.15 Shared Bus node interface diagram 72

3.16 Shared bus extension thread flowchart 73
3.17 Usage of multiple Shared Bus instances by multiple peripherals 74
3.18 Fault injection framework from Andrea Höller PhD thesis 75
3.19 Fault injection components 76
3.20 Injection of fault on instruction decoding 78
3.21 Faults during memory read and write operations 79
3.22 Injection of faults blocking peripheral access 80
3.23 Fault request interactions sequence diagram 82
3.24 Fault Coordinator flowchart 83
3.25 Fault injection initialization flowchart 85
3.26 Fault injection flowchart 86

4.1 SAS architecture concept block diagram 89
4.2 SAS angle messages timely diagram 90
4.3 SAS Fail Degraded angle messages timely diagram 90
4.4 System tasks modelling through threaded execution 91
4.5 SAS system architecture diagram 92
4.6 SAS state machine section (1) 93
4.7 SAS state machine section (2) 93
4.8 SAS state machine section (3) 94
4.9 SAS state machine section (4) 94
4.10 SAS state machine section (5) 95
4.11 SAS Fail Degraded state machine 95
4.12 SAS software stack 96
4.13 MCAL modules developed 97
4.14 Section of the ADC module interface diagram 99
4.15 ADC module use cases diagram 99
4.16 ADC module sequence diagram 100
4.17 Example usage of the MCAL ADC module during Reset state 101
4.18 QEMU S32K116 machine peripherals 102
4.19 S32K116 machine and peripherals class diagram example 103
4.20 Co-simulation environment diagram 107

4.21 Co-simulation environment running the SAS application (1) 107
4.22 Co-simulation environment running the SAS application (2) 108
4.23 Subsystem blocks susceptible to faults 109
4.24 Parameterized failure probability density curves 110
4.25 Co-simulation environment used for simulations 111
4.26 Histogram and probability density function of the simulation results 113
4.27 Histogram and cumulative distribution function of the simulation results 113
4.28 Distribution of the probability of wrong data output by the system 114
4.29 Distribution of the mean time to failure after applying bagging method 114
4.30 Distribution of the mean time to failure after applying boosting method 116
4.31 Distribution of the time before wrong system output after applying boosting method . . . 116
4.32 Fault occurrence before system failure 117

A.1 Relation between the Failure rate density, reliability and hazard rate functions 128
A.2 Embedded systems development flow (detailed) 130

B.1 Shared Bus process flowchart 131
B.2 Change on QEMU for the Fault Injection extension 132
B.3 Memory fault injector function flowchart 133
B.4 Usage of the memory fault injection function on flatview_read and flatview_write functions 134
B.5 Peripheral block fault functions flowchart 135
B.6 Changes to arm_tr_translate_insn function 136

C.1 Class diagram of the software-only SAS validation 137
C.2 SAS State Machine diagram 138
C.3 Class diagram of the WDG MCAL module 139
C.4 Class diagram of the ADC MCAL module 140
C.5 Class diagram of the PORT MCAL module 141
C.6 Class diagram of the MCU MCAL module 142
C.7 Class diagram of the GPT MCAL module 143
C.8 Use Case diagram of the WDG MCAL module 144
C.9 Use Case diagram of the ADC MCAL module 144
C.10 Use Case diagram of the GPT MCAL module 145

C.11 Use Case diagram of the MCU MCAL module (1) 145
C.12 Use Case diagram of the MCU MCAL module (2) 146
C.13 Sequence diagram of the PORT MCAL module 147
C.14 Sequence diagram of the ADC MCAL module (1) 147
C.15 Sequence diagram of the ADC MCAL module (2) 148
C.16 Sequence diagram of the GPT MCAL module (1) 149
C.17 Sequence diagram of the GPT MCAL module (2) 149
C.18 Sequence diagram of the ADC MCAL module (3) 150
C.19 Sequence diagram of the MCU MCAL module 150
C.20 Sequence diagram of the WDG MCAL module 151
C.21 Flowchart of the microcontroller startup sequence 152
C.22 SAS Error State flowchart 153
C.23 SAS Race Condition State flowchart 153
C.24 SAS Reset State flowchart 154
C.25 SAS Angle Calculation flowchart 155
C.26 SAS Redundant Module Check State flowchart 156
C.27 SAS Angle Sampling State flowchart 157
C.28 SAS Transmission State flowchart 158

D.1 Memory Operations functions interfaces 160
D.2 S32K116 QEMU Machine class diagram 161

E.1 Fault Coordinator sequence diagram 177
E.2 Fault Coordinator flowchart 178

List of Tables

2.1 Summary of main advantages and disadvantages of fault injection techniques. 35

2.2 Types of X-in-the-Loop testing 38

3.1 Details about fault locations and fault modes supported by FIES 76

4.1 Results of each iteration of the Boosting method 115

B.1 Communication commands between QEMU and the Shared Bus. 132

E.1 Component Mean Time Between Failure Values 177

List of Abbreviations and Acronyms

ADC Analog to Digital Converter

API Application Program Interface

ARM Advanced RISC Machine

ASIL Automotive Safety Integrity Level

BSW Basic Software

CAN Controller Area Network

CPU Central Processing Unit

DMR Dual-Modular Redundancy

DWC Duplication With Comparison

ECC Error Correction Code

ECU Electronic Control Unit

ESRG Embedded Systems Research Group

FI Fault Injection

FIES Fault Injection framework for the Evaluation of Software-based fault tolerance

FMI Functional Mockup Interface

FMU Functional Mockup Unit

FPGA Field Programmable Gate Array

FSM Finite State Machine

GHDL GNU Hardware Description Languague

GPIO General Purpose Input/Output

GPT General Purpose Timer(s)

HiL Hardware-in-the-Loop

IRQ Interrupt Request

IPC Inter-Process Communication

I/O Input/Output

MCAL Microcontroller Abstraction Layer

MCU Microcontroller Unit

MiL Model-in-the-Loop

MISRA Motor Industry Software Reliability Association

MTBF Mean Time Between Failures

MTTF Mean Time To Failure

MTTR Mean Time to Repair

NMR N-Modular Redundancy

OS Operating System

PC Program Counter

PiL Processor-in-the-Loop

QEMU Quick EMUlator

RAM Random Access Memory

RTE Runtime Environment

SAS Steering Angle Sensor

SbW Steer-by-Wire

SiL Software-in-the-Loop

SoC System-on-Chip

SPI Serial Peripheral Interface

TB Translation Block

TCG Tiny Code Generator

TCP/IP Transmission Control Protocol/Internet Protocol

TMR Triple Modular Redundancy

UART Universal Asynchronous Receiver Transmitter

XML eXtensible Markup Languague

Chapter 1

Introduction

The world is undergoing a day by day technological growth, with an increasing tendency for the usage of digital systems to perform everyday tasks. With the fast growth and ubiquity of technology in human life, there is a also a growing need for better and more robust digital system solutions. Most of these digital solutions come in the form of embedded systems.

Embedded systems cover applications ranging from General Purpose systems, such as household electronics, to Safety Critical systems such as flight control and nuclear control [2]. The development of Safety Critical applications requires particular attention since system failures can incur on loss of money or possibly human lives. The possibility of disasters under such type of applications brought system reliability concepts to the foreground. Reliability is a system metric that is directly related to the system failure probability, meaning that highly reliable systems present the lowest probabilities of failure during its operational lifetime. High reliability systems are mostly found in the fields of avionics, life support, and more recently in the automotive sector.

One way to increase system reliability is to replicate its components, allowing the system to achieve an higher time before failure, consequently reducing its failure probability. This replication technique is known as redundancy, and it can be implemented on both software and hardware. Typically, redundant components interact with each other managing and deciding about the operation state of the system. These interactions are mostly only validated when testing directly on hardware late on the development, greatly affecting time spent on debugging and possibly going back to early development phases to correct design or implementation bugs. An increasing system complexity caused by the addition of redundancy mechanisms aggravates the consequences of late validation. A way to validate despite interactions is through the use of full system simulation but most simulators do not contemplate the use case of redundancy, making it harder to retrieve meaningful results.

Although redundancy improves reliability, there is no guarantee that the system meets reliability requirements only by using this technique. Estimations of reliability throughout the development cycle provide an overview on how the reliability metrics are being met, avoiding possible later and costly development reiterations. These estimations can be done with the help of simulation-based methods, allowing to have feedback early on the development cycle.

Adopting a simulation environment contemplating redundancy as a use case, would bring a lot of advantages to the reliability development cycle, such as easier validation of redundancy interactions and a consequent ability to perform reliability estimations. Also, the development process would be speeded up, and also resource independent, since the system could be developed in its entirety in a host platform without being bound to a physical target platform.

1.1 Motivation and Objectives

The widening usage of embedded systems on safety critical applications consequently leads to an increase in the demand for reliable systems. As reliability development is a large covering area, there are plenty of research possibilities. The opportunity to study reliability comes from a new research team under the Embedded System Research Group (ESRG) that is tackling reliability in an research approach. With that in mind, this type of work pretends to take pioneer steps into maturing the research knowledge of the reliability topic. Another interesting point is that no reliability development cycle is written in stone, meaning that new research on this topic may help to achieve a better and unified reliability development flow.

The work to be developed on this dissertation aims to provide simulation extensions that aid reliability systems development, focusing on the redundancy aspect and on reliability estimations early on development cycle. With that in mind, one of the goals is to take a pragmatic step into a simulation environment that allows to simulate redundant systems and their interactions without the need for physical hardware. Furthermore, a second goal aims to extend such simulation environment to allow early estimation of reliability metrics during development.

1.2 Dissertation Structure

The document presents the development of simulation extensions that aid reliability development, along with a case study that contemplates reliability characteristics. It contains six chapters, which will be briefly described next.

Chapter two presents the concepts and methodologies used during the progress of this dissertation. It starts by introducing concepts about embedded systems and its development cycle, along with operating systems and its concurrency mechanisms. Afterwards, it presents notions about reliable systems, their metrics and development flow, alongside the concept of redundancy and fault tolerant architectures. Furthermore it is also presented techniques to evaluate and estimate system reliability. After reliability, the advantages and techniques of embedded simulation are explored with a special focus on co-simulation. Finally, a special attention is given to QEMU, uncovering its features, namely in embedded platform emulation.

Chapter three presents and describes the work developed to support reliability development through simulation extensions. It features the develop extensions explained in a functionality oriented approach, providing insight into how development was made and the needed information for developers to use the extensions. Examples and usage guidelines are given, helping the reader to understand how to use the provided work as a developer.

Chapter four contains both case-study's development and its usage to validate the simulation extensions. It shows the steps taken to develop the case-study and the results of each of the development phases. The case study chosen was based on a redundant architecture, presenting all the characteristics needed to validate the developed extensions.

The last chapter discusses results and concludes about the work developed during the thesis. It also presents the current limitations of the extensions developed as well as the future improvements that may help their usability and performance.

Chapter 2

State of the art

This chapter aims to contextualize the dissertation regarding the technological landscape as of the current year, and give a broad view on the state of the art concerning the technologies and theoretical basis on which it was conceived, hopefully clarifying the nature of its conception as a logical step to make a contribution in the literature.

Since this dissertation is embedded system oriented, a brief contextualization about the concepts and development techniques regarding embedded systems is made. In the embedded world, safety and reliability are important characteristics that need to be taken into account during the development cycle. Thus an introduction about reliability oriented systems follows the embedded development contextualization. Later, simulation and co-simulation topics are approached since simulation is an important technique that is present during the whole development cycle. There are several simulation tools, but in this dissertation QEMU will have the spotlight, therefore, an overview about this tool is made, finishing the chapter.

2.1 Embedded Systems

Currently, embedded systems can be found everywhere in our every day lives, ranging from consumer electronics and electrical appliances to office automation, industrial automation, military defense systems, transportation systems, aerospace systems, medical systems and so forth. At home, they come in all sizes and formats, from modern washing machines to a simple MP3 player, to more elaborate systems like an ADSL router or printers. Embedded systems became so intrinsic in our daily lives that we are mostly unaware of their presence.

2.1.1 Definition

An embedded system is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a specific function [3]. Although this seems a correct definition of an "embedded system", the definition is hard to pin down, since embedded systems are constantly evolving with advances in technology.

An embedded system has a specific purpose and is designed and optimized for that purpose. Its development is application-oriented, so engineers developing these types of systems should design them with the least amount of resources to perform the proposed task in order to optimize variables such as cost, energy consumption, weight, and performance. However, as an embedded system is always subject to requirements and restrictions, a balance and management of the resources used is always required. Some common features of embedded systems include:

- Containing (a generic type) processing unit: microprocessor, microcontroller, SoC (System On Chip), etc;
- Typically designed and configured with hardware strictly necessary to perform a specific task, or a restricted set of tasks;
- May have restrictions on energy consumption, being powered by batteries, or even a combination of alternative forms of energy, such as renewable wind and solar;
- Most of these systems are used in stand-alone applications, i.e. without any human intervention.

Compared to traditional computer systems, an embedded system has severe hardware constraints: the processing unit is strictly necessary for the tasks it has been designed for, it has a small amount of memory, and there may not even be a graphical interface.

2.1.2 Embedded Development

Usually, an embedded system's hardware and respective boards are designed and developed alongside the software, being tailored and designed specifically for the target scenario. However, mixing software with board development may not be a very good idea, as it may be hard to trace system faults and differentiate them as software bugs or board malfunctions.

To easily develop and debug software, development platforms are used as a more practical way of prototyping embedded systems. Embedded development platforms are often designed by the processing chip's manufacturers, containing a general set of components and interfaces that complement the chip's functionality, such as memories, communication connectors (e.g Ethernet, Controller Area Network (CAN)), Local Interconnect Network (LIN), and General Purpose Input/Output (GPIO), which combined produce a platform to easily prototype embedded systems. Figure 2.1 presents a block diagram of a possible development board, designed around a SoC.

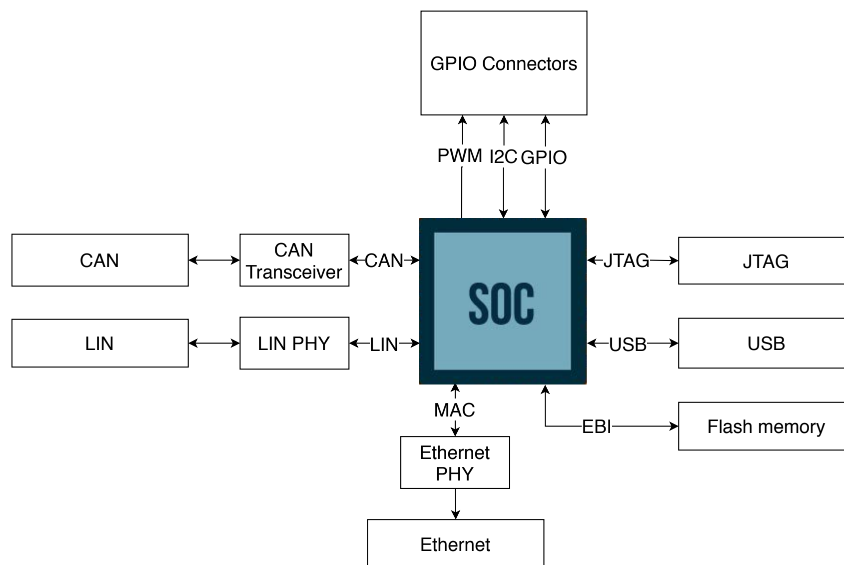


Figure 2.1: Embedded platform block diagram

Development on these platforms can be made using a host-based approach i.e, developing the software on a host desktop system and later deploying it on the target embedded platform [4]. This plays a decisive role in the correctness of the software to deploy, as it facilitates software defect detection.

To be able to do all of this, a toolchain is needed for the target architecture. This toolchain is usually provided by the development board manufacturer. A toolchain comprises programming tools needed to effectively use the target platform, such as compilers, linkers, debuggers, loaders, and other utilities. The act of compiling software for an architecture that's different from the one where the compilation is being done is called *cross-compiling*.

After a prototype is developed and ready for production, the system may be integrated into a final board that is adjusted for the application's purpose and possibly leaving out elements present in the development platform that were not used.

2.1.3 Development Flow

The development of a new embedded system project goes through several phases until a final prototype is ready. Whilst no embedded development flow is written in stone, iterating through certain phases of development can contribute for not only for a better end product but also has positive influence on the development time. The development cycle that will be presented next is internally adopted by the Embedded Systems Research Group and its phases are presented in figure 2.2.

On project start, during the Requirements phase, both the functional and non-functional requirements are gathered, along with the possible restrictions imposed. Upon having all requirements well-defined, the system application (or its purpose), is conceptualized into systems tasks. By specifying possible system tasks, one can have a better overview on both how the system will fulfill its requirements and on the possible needed processing. Such conceptualization can be made on software by having a threaded execution of the possible system tasks, allowing early validation of models or algorithms used on the system.

After application conceptualization, an architecture that satisfies system requirements is designed, contemplating the blocks that allow the system to meet such requirements, e.g. an ADC, a microcontroller, etc. The resulting architecture is further validated by simulating its behaviour, which can be expressed as Finite State Machines. A way to validate is through a software-only approach, where the system state machines are transposed on software and both the states and the transitions are checked for their correctness. This may also allow for software reuse.

As the modelling and validation finishes, decisions about the platform and resources are made. In this phase there is a *trade-off* between project cost and development time when selecting the target platform and resources. Such *trade-off* decides on the the usage of off-the-shelf products or the development of a new platform specially designed for the requirements. Depending on this decisions, the resources are also allocated on software and hardware, providing the basis to develop both software and hardware architectures. Development of both these architectures is decoupled, meaning that development of both software and hardware is made in parallel. Under hardware, development comprises of the definition of the architecture, the selection of the components and the design of both circuits and layout. Validation of the design effort is made through simulation of both components and circuits. Under the software development, an architecture is defined, along with all the interfaces, classes and modules that will support the system application. Validation can be made not only through full system simulation but also through a software-only approach where all the hardware is abstracted and hardware calls are substituted by *host*

mechanisms. The software-only approach is typically made before full system simulation, since software-only occurs on only *host* platform while full system simulation occurs on an emulated *target* platform. Both the validated hardware and software architectures are later deployed into a single platform, which is finally tested to validate requirement fulfillment.

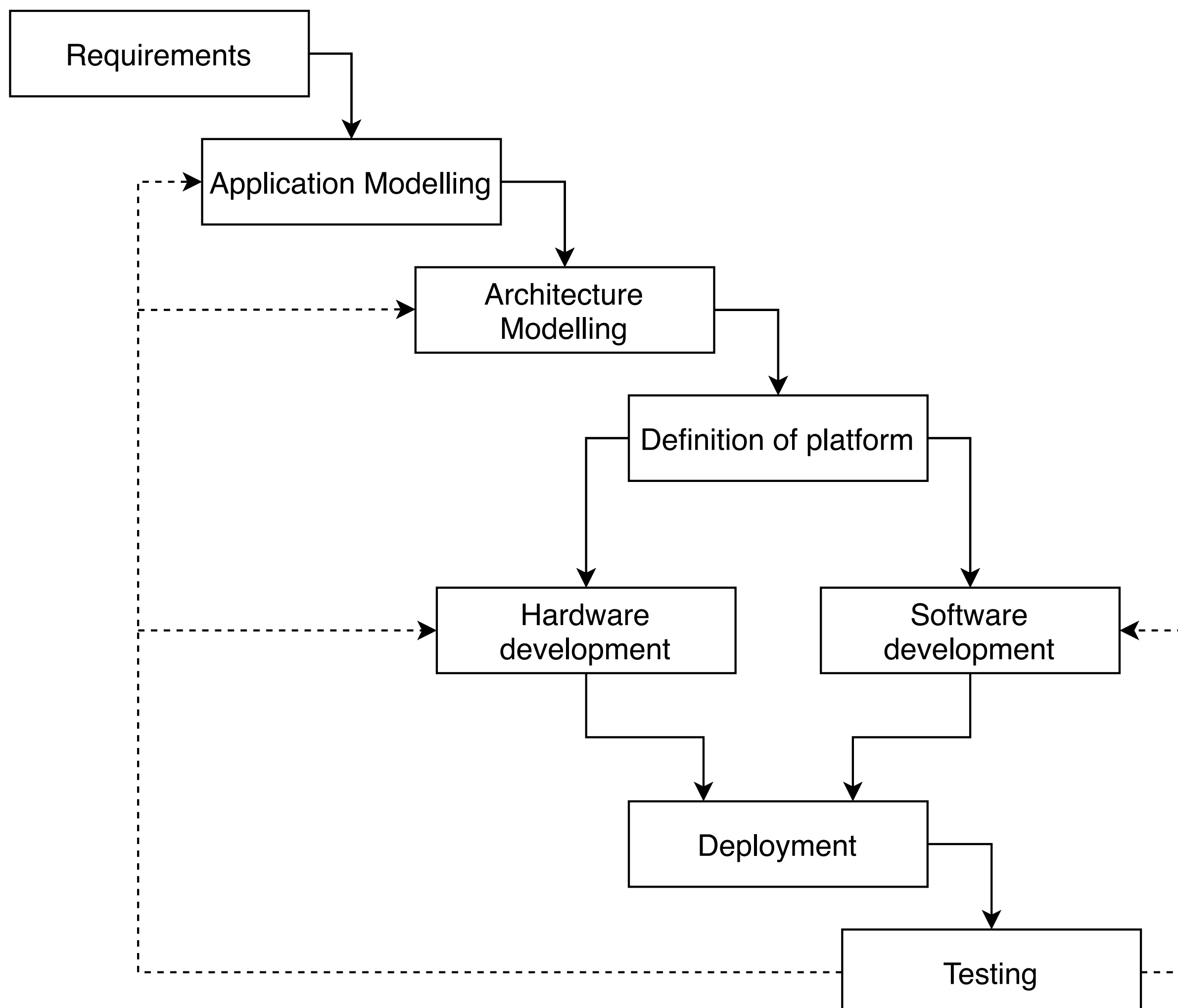


Figure 2.2: Embedded systems development flow

2.2 Operating Systems

An operating system is the layer of software that manages a computer's resources for its users and their applications [5]. This software layer is invisible to the high level developer, controlling embedded devices such as toasters, gaming systems, and the many computers inside modern automobiles and airplanes. Operating systems are also an essential component of more general-purpose systems such

as smartphones, desktop computers, and servers. They are responsible for facilitating the execution of programs (even running many at the same time), allowing programs to share memory and managing interactions between devices.

The core of an operating system is the kernel. It manages many of the fundamental details that an operating system needs to deal with, including memory, concurrency, scheduling, and I/O events. In general, the kernel is the part of the operating system that interacts directly with the hardware; it presents an abstracted interface to the rest of the operating system components and user applications.

A typical operating system is layered and implements at least two execution spaces: the kernel space and the user space. An overview of the common layers of an OS is shown in figure 2.3. The two outter layers, user applications and user services, compose the user space while the kernel layer composes the kernel space. All code that accesses memory, I/O or any hardware resource, runs on the kernel space, interfacing with the upper layers through system calls. User services and user applications run in user space and use the interfaces provided by the kernel to access specific hardware resources. These two layers run all user programs from shells and date services, to code editors and graphical interfaces.

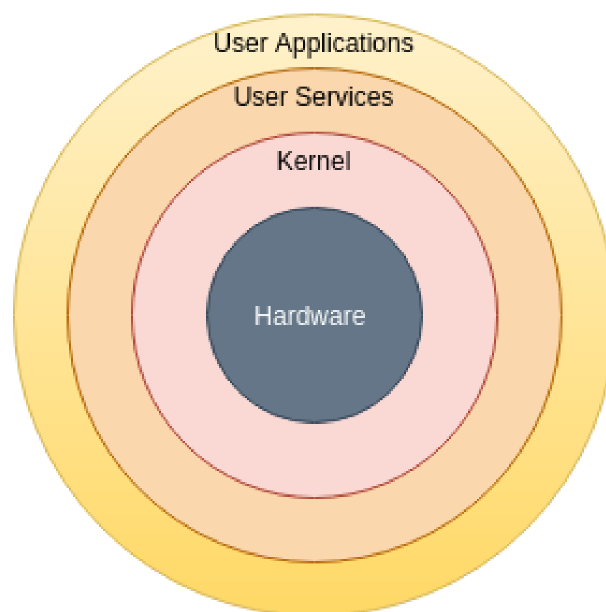


Figure 2.3: Layers of an OS

2.2.1 Concurrency

In real-world systems many things are happening simultaneously and must be addressed within time constraints, thus systems must be reactive. They must respond to external events which may occur at

somewhat random times and in random order. This can be complex since microprocessors are only capable of executing one instruction at a time, and if an external event triggers a processing task, it should mean that the processor would be unavailable for any other external events. Thus a way to avoid processor block on a specific task is to use a technique called virtual parallelism, which shares multiple task execution to achieve the illusion of multiple tasks running concurrently. This technique can dramatically speed things up, for example by preventing one task from blocking another while waiting for I/O.

A single program can contain multiple tasks. These tasks are executed, in a single processor, only one at a time by timely switching the currently executing task. An example is presented on figure 2.4. The example program contains three tasks, where each of them is assigned a timeslot in processor execution time, and all of them run during that time. A high-level view of the execution shows the tasks running concurrently, but, in reality, only one task runs at a time.

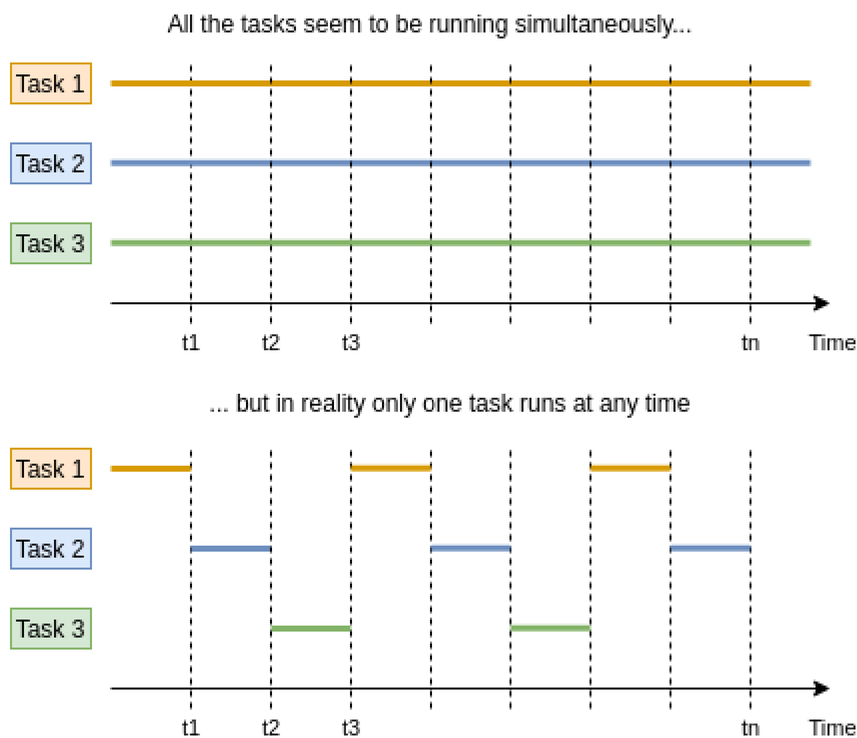


Figure 2.4: Concurrency between tasks

The time tasks spend running is managed by the OS, more specifically the scheduler. The scheduler decides which task depending on the type of task switching algorithm. There are two main groups of switching algorithms: cooperative scheduling and preemptive scheduling. The first one assumes that the tasks are the ones that give up execution for other tasks. An obvious downside is that poorly implemented tasks may compromise system execution. On the other hand, preemptive scheduling implements a timer,

managed by the OS, that interrupts task execution and devices what task to run next. Comparatively to the latter, the level of abstraction of this scheduling mechanism guarantees that poorly written tasks do not block the processor.

Within the OS, a running program is called a process and the multiple tasks that run within the process are called threads. Threads share the same memory region, as they all belong to the same process, however, different processes have different memory regions. As shown in figure 2.5, processes are assigned independent memory regions and the threads within the process share the process memory between them. A combination of registers and stack represents the execution context. A single-threaded process only contains a single stack and a set of registers, while a multi-threaded process contains a stack and registers proportionally to the number of threads. This set of stack and registers is needed to have context switching between threads.

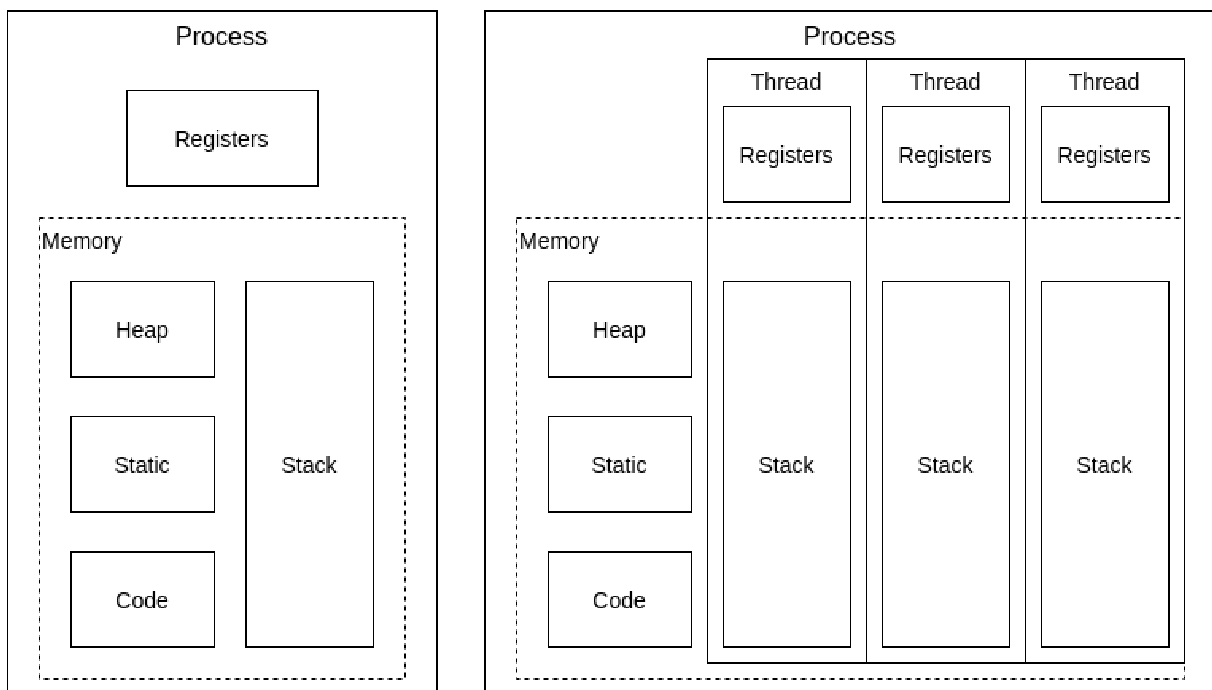


Figure 2.5: Process and thread memory block diagram

A challenge that arises from having multiple processes running is memory management. As multiple processes are running, one process could take up the same memory region or read a memory space that is currently occupied by another process. This overlap would come at a high cost since the development of the process would need to consider this, forcing processes to be responsible for managing their own memory space and avoiding overlapping any other region. The Virtual Memory technique relieves

processes of this responsibility by letting any process assume that all system's memory is his. The memory addresses used in the process are interpreted by the OS as virtual addresses and then translated to physical addresses.

Usually, physical memory organized in page frames. For each process, the OS assigns a memory map that associates a virtual page to a physical page. The memory maps provide different virtual memory mappings for different processes, as they may use identical virtual addresses. As shown in figure 2.6, the memory map contains entries that are the translation between the virtual memory and physical memory.

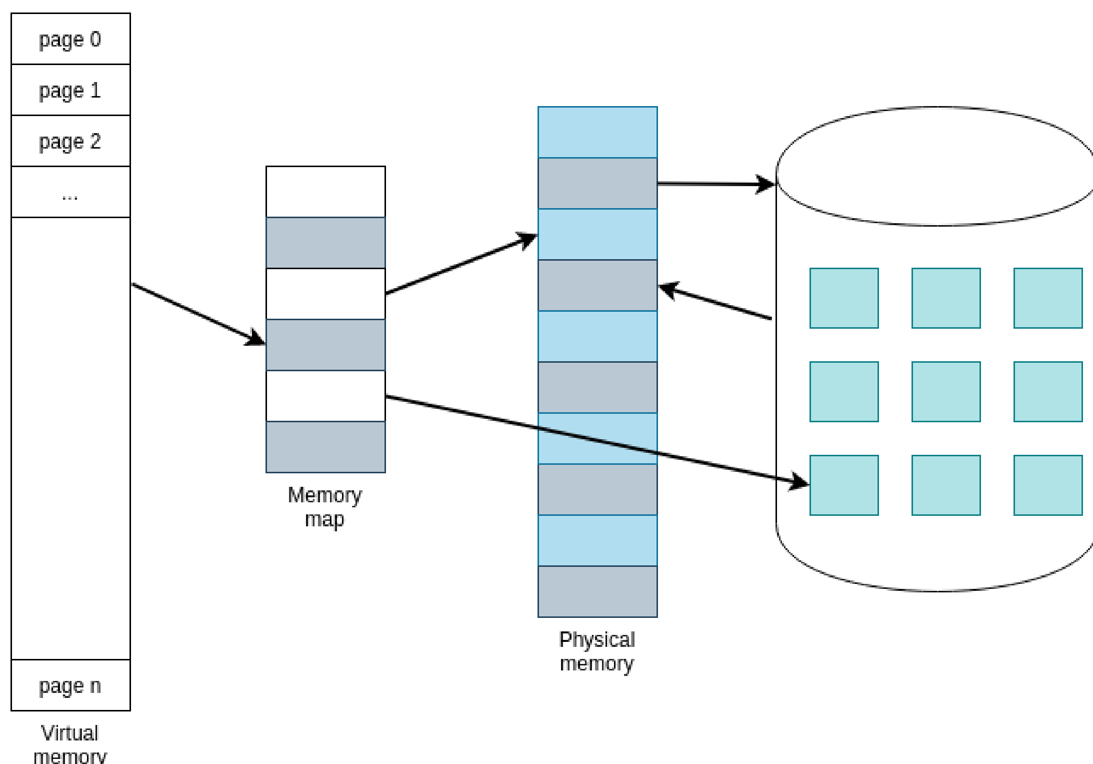


Figure 2.6: Virtual memory overview

Virtual memory can be extended with secondary storage such as hard drives, SSD's and SD cards, with the latter being the most common choice in embedded systems. Memory page frames can be swapped between the system's main memory and secondary storage, moving in and out of the main memory according to the process execution needs. In this case, the page tables must keep additional information for each page, usually in the form of bits like the present bit and the dirty bit representing their states in the secondary storage memory space. The present bit indicates what pages are currently in physical memory or on disk and can indicate how to treat these different pages, i.e., whether to load a page from disk and page another page in physical memory. The dirty bit allows for performance optimization by keeping track of which pages have been modified and must be written to disk before they are replaced.

2.2.2 Inter-Process Communication

Another challenge of designing concurrent systems arises because of the interactions which happen between concurrent activities. Although processes run independently, they may need to share data between them.

An independent process is, typically, not affected by the execution of other processes, but one process can depend on the computation of another process. Interprocess communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both shared memory and data transfer.

Shared memory allows processes to exchange information by placing it in a region of memory shared between the processes. A process can make data available to other processes by placing it in the shared memory region. Because communication does not require system calls, shared memory can provide high-speed communication.

On the other hand, data transfer uses the notion of writing and reading. In order to communicate, one process writes data to the IPC facility, and another process reads it. Examples of this type of IPC are Sockets (figure 2.7), Pipes, and Message Queues. All these types of IPC use file descriptors, although message queues provide a notification facility that can send signals to a process. Regarding direction, sockets are bidirectional, meaning that signal socket instances provide writing and reading capabilities under the same entity, while pipes and message queues are unidirectional, providing only reading or writing capabilities on the same entity.

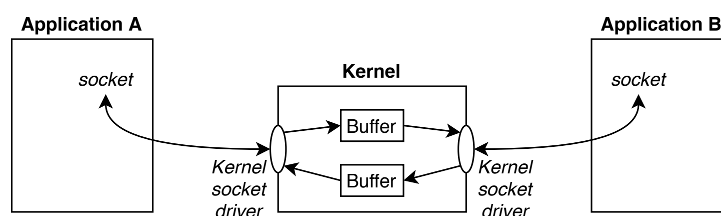


Figure 2.7: Exchanging data between sockets

Concurrent systems also require synchronization between processes or threads. The synchronization facilities allow processes or threads to coordinate their actions, avoiding things such as simultaneously updating a shared memory region or the same part of a file. Without synchronization, such updates could cause an application to produce incorrect results. Different synchronization facilities provide optimal

solutions for different problems. The UNIX system provides the following facilities: Semaphores, Mutexes and Condition Variables, and File Locks.

Mutexes and Condition Variables are blocking mechanisms used with POSIX threads (which is the UNIX interface for threads) that guarantee exclusiveness to a resource, i.e., in a given instance only one thread can perform read or write operations on the said resource. The condition variable signals the end of an operation on the resource.

A semaphore is signaling mechanism that keeps track of the number of accesses permitted to the resource. The number possible concurrent acesses is tracked by an integer variable that accessing threads either decrement or increment as they request or give up a execlusive access to the resource. Similarly to the mutex, if the resource is being used, the requesting thread blocks until semaphore count become greater than one.

Lastly, File Locks are explicitly designed to coordinate the actions of multiple processes operating on the same file. They can also be used to coordinate access to other shared resources. File locks come in two flavors: read locks and write locks. Any number of processes can hold a read lock on the same file. However, when one process holds a write lock on a file, other processes are prevented from holding either read or write locks on that file.

2.3 Reliability-oriented Systems

The growing complexity of equipment and systems, as well as the rapidly increasing cost incurred by loss of operation due to system failures, have brought the aspects of system reliability to the foreground [2]. Reliable systems are systems that have low failure probability during their operational lifetime, providing the intended functionality even in adverse environments. The application spectrum of reliable systems ranges from even simple general purpose commercial systems up to critical ones that typically represent human hazards upon failure. The typical approach for developing reliable systems is to apply techniques that allow the system to tolerate faults, on both hardware and software. In order to understand fault tolerance concept, a definition of fault, error, and failure is of high importance.

2.3.1 Fault, Error, and Failure

A **fault** is an abnormal condition that can cause an element or item to fail. This is the possible cause of an error. An **error** refers to the offset between a computed, observed, or measured value and the true

theoretical value. Errors can propagate within a component or system into failures. A **failure** refers to the termination of the ability of a system or component to perform a function as required. The diagram on figure 2.8 shows how a fault on a component could evolve to a failure on the target system. A fault in a component can provoke an error followed by failure, which in turn provokes a fault in the target system that can evolve to a failure. In short, fault is a defect, an error is an abnormal state, and a failure is an event to avoid.

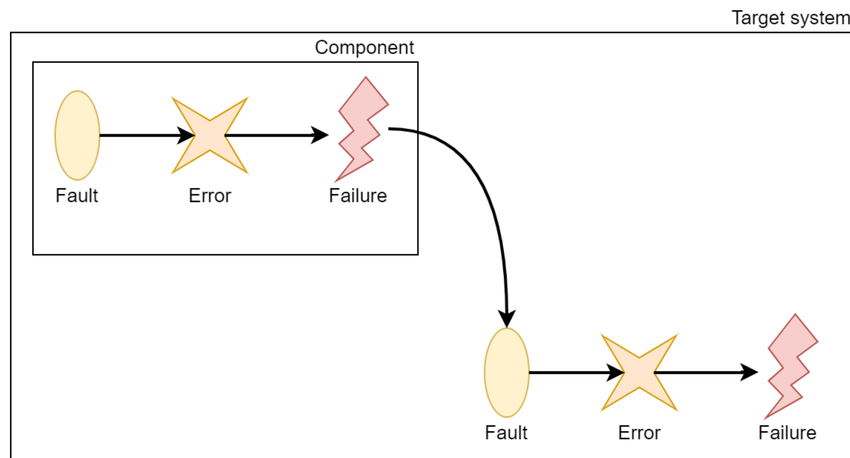


Figure 2.8: Fault, error and failure diagram

Faults can be classified according to many criteria. Regarding domain, they can be classified as hardware or software faults. In terms of their causes, they can be divided into three groups [6]: (1) design faults that include all the faults occurring during development of both software and hardware; (2) physical faults that include all faults that affect hardware; and (3) interaction faults that include all external faults, such as environmental induced. Since software presents, typically, a higher complexity than hardware, design faults are more prone to happen in software. This difference is explained by the fact that hardware machines have usually a smaller number of internal states than software programs [7]. The hardware faults can be due to a physical or an interaction fault and indirectly interfere with the data or the program in execution. Hardware faults are classified according to their duration as permanent or transient [8]. Permanent faults remain in the system until they are removed and are caused by physical defects of the hardware; transient faults appear and disappear with no explicit intervention from the system. Unlike permanent faults, transient faults can be tolerated and mitigated as they may not compromise the correct operation of the system [9].

2.3.2 Dependability

Systems that aim to avoid all types of failures by preventing faults and recovering from them, typically have dependability requirements. The definition of dependability is the ability to avoid service failures that are more frequent and more severe than acceptable [6]. In other words, dependability is the justifiable amount of trust, one can put in the system to deliver the correct service. This measure is of most importance since all the design effort to avoid faults is put into improving system dependability. The dependability concept consists of the following attributes [10]:

- **Availability** - readiness for correct service.
- **Reliability** - probability of continuity of the correct service.
- **Safety** - absence of catastrophic consequences on the user(s) and the environment. For many systems, high reliability and safety is a constraint, such as flight control systems [11].
- **Integrity** - absence of improper system state alterations.
- **Maintainability** - the ability to undergo repairs and modifications.
- **Confidentiality** - absence of unauthorized disclosure of information.

Within dependable system, there is a large spectrum of type of systems that follow, in different weights, some of these attributes. Such systems can range from various applications such as [12]: (1) General Purpose Commercial Systems, (2) High Availability, (3) Long Life and (4) Critical Systems. General Purpose systems are the least demanding system regarding dependability attributes, since typically, they are not very complex. High availability systems demand of a very high probability that the system will be ready to provide the intended service when required, such as transaction processing systems. Long Life systems require that it operates as intended for a long time without maintenance. This includes satellites and other aerospace systems. Lastly, critical systems require a high degree of reliability and safety. This category includes safety-critical systems, in which a failure can cause loss of lives, and mission-critical systems, in which a failure can cause damage in equipment, or the loss of efforts and the mission failure. Such systems are flight control systems, nuclear plants, X*-by-wire systems (Fly, Steer, Brake...).

The main focus, under the dissertation context, will be the Reliability attributes regarding Critical Systems. As such, the next sections will tackle reliability engineering and how it applies to, not only but mostly, critical systems.

Reliability engineering is the engineering branch that aims to develop complex systems that are resistant to faults by applying techniques to prevent or to reduce their likelihood or frequency [13]. As previously mentioned, reliability is the ability to provide correct service even in adverse environments. This is not an absolute attribute, meaning that a system or component will not always, during its lifetime, be able provide correct service. One example of such are any hardware component, which wear out during their lifetime, resulting on increasing possible failure and loss of ability to provide correct service. That being said, reliability can be seen as a probabilistic function, which provides the probability, over time, of providing service correctness [2].

2.3.3 Reliability Metrics

The reliability function is given by a cumulative distribution function:

$$R(t) = 1 - Q(t) = 1 - \int_0^t f(t)dt = \int_t^\infty f(t)dt$$

where Q(t) is the unreliability function, which defines the probability of failure by a certain time. Subtracting this probability from 1 gives the reliability function. This function gives the probability of success of a component or system to accomplish its service. The diagram on figure 2.9 presents an example of such probability.

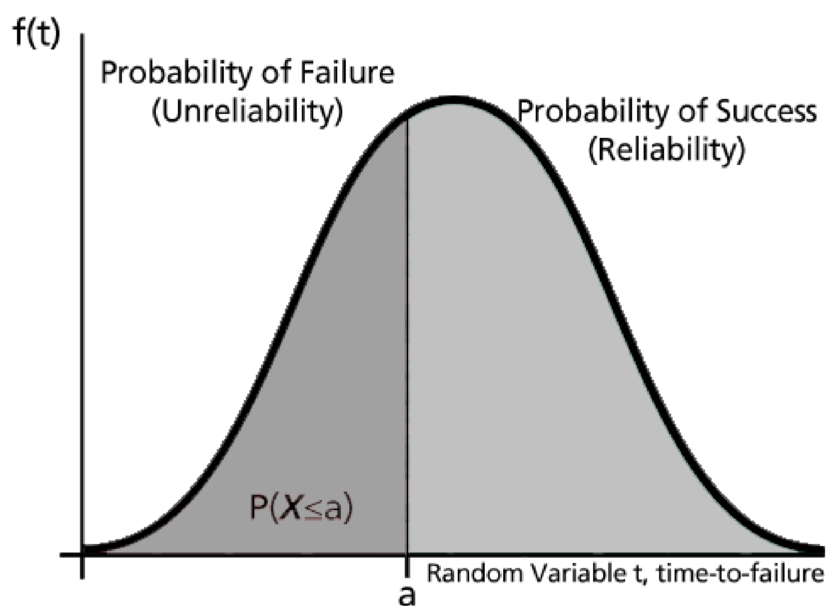


Figure 2.9: Reliability probability function distribution diagram [14]

The failure rate metric plays an important role in reliability analysis as an awareness of component reliability. This metric determines of the number of failures occurring per unit time. The function, named hazard rate, that characterizes such metric as an instantaneous value is given as:

$$h(t) = \frac{f(t)}{R(t)}$$

where, R(t) is the reliability function and f(t) is the probability of failure distribution. The diagram on appendix A.1 shows the relation between the previous mentioned functions.

Another expression that is always part of reliability is Mean Time To Failure (MTTF) used for non-repairable systems, which are systems that have their service terminated after any failure. The Mean Time Between Failures (MTBF), is used if the system recovers to the same state after each failure. MTBF values must be computed with different reliability distributions for different time periods between failures. By using the mathematical expectation theorem, MTBF can be expressed as:

$$MTBF(t) = \int_0^{\infty} t \times f(t)dt$$

where, t is the time in hours and f(t) is the failure probability distribution.

When the hazard rate is constant (λ), the MTBF can be given as the inverse of the failure rate, being $MTBF = \frac{1}{\lambda}$. A typical figure of this value for safety-critical systems is 10^{-9} failures per hour [10].

2.3.4 Failure Distribution

Component failure can be expressed in a probability distribution function or as a constant failure rate, as was previously stated. The typical time versus failure rate curve for components is known has the "bathtub curve", which is presented on figure 2.10. This representation has proven to be particularly appropriate for electronic equipment and systems and is widely accepted in the reliability community [15]. The characteristic pattern for the curve is a period of decreasing failure rate (DFR), or infant mortality, followed by a period of constant failure rate (CFR), or normal life, followed by a period of increasing failure rate (IFR), or wear out.

In the period of infant mortality the high failure rate is the result of poor design, the use of substandard components, or lack of adequate controls in the manufacturing process. During normal life, the failure rate remains constant, which can be seen as useful operating life. Finally, the wear out period has high failure rate as a result of equipment deterioration due to age or use.

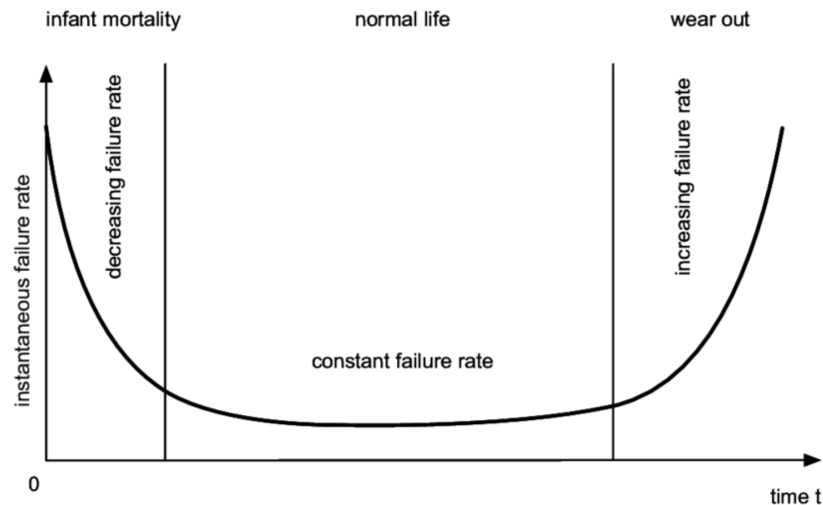


Figure 2.10: Example of a failure distribution of a hardware component [1]

The "bathtub" failure curve gives a good insight into the life cycle reliability performance of a system. Depending on the physical meaning, the random quantities obtained can have different probability distributions laws (exponential, normal, Weibull, gamma, Rayleigh, etc.). Over the infant mortality period of operation, the bathtub curve can be represented by gamma and/or Weibull laws [16]; over the normal period of operation, by the exponential distribution; over the wear out period of operation, by gamma and normal distributions. Thus, most component failure patterns involve a superposition of different distribution laws.

2.3.5 Concepts of Redundancy and Fault Tolerance

Alongside the system functional requirements that may impact the techniques and components used during development, reliable systems also present reliability requirements that must be met. Design of such system focus on augmenting system reliability by applying practices that allow for longer time to system failure. An example of practice regarding hardware design, is the selection of components that have longer durability, i.e. lower failure rate, which can improve the overall system time to failure.

The most common practice to achieve system reliability is the usage of fault tolerance techniques, which guarantee system functionality even in the presence of faults. These techniques are largely supported by the use of redundancy.

Redundancy

Redundancy is providing functional capabilities that would be unnecessary in a fault-free environment. Redundancy technique rests on having extra components designed to have the same functionality as the original ones. By adding these redundant components, or replicas, it is ensured that if some part of the system fails, a redundant component resumes the functionality of the faulty one. This way the system maintains correct service delivery [8].

There are two kinds of redundancy: spatial and computation. Spatial redundancy provides additional components, functions, or data items to mask faults that may happen on the original components. Space redundancy is further classified into hardware, software, and information redundancy, depending on the type of redundant resources added to the system. In computation redundancy the computation or data transmission is repeated and the result is compared to a stored copy of the previous result. All these types of redundancy will be explored in the next sections.

Hardware Redundancy

Hardware redundancy is when two or more physical copies of the hardware component are used. These hardware components perform some of the functions already provided by the original system. Depending on how the redundant components actuate on the system, they can be classified as passive, active or hybrid. Passive components mask the fault that occurs without requiring any action from the system. This type of components guarantees that only the correct value is passed to the system. Active redundancy requires a fault to be detected before it can be recovered from. After the detection of the fault, the actions of location, containment and recovery are performed to remove the faulty component from the system. Active techniques require that a system is stopped and reconfigured to tolerate faults. Hybrid redundancy combines passive and active approaches. Fault masking is used to prevent the generation of erroneous results and fault detection is used to detect or reconfigure a faulty component.

A natural evolution of active hardware redundancy consists of having two or more component replicas operating in parallel. Duplication with comparison (DWC) is a common solution [8], where two processing units execute the same task at the same time and their results are compared by a checker module (figure 2.11). Depending on the application, the duplicated modules can be processors, memories, data buses or even computational subsystems. This technique allows error detection but does not correct error by itself.

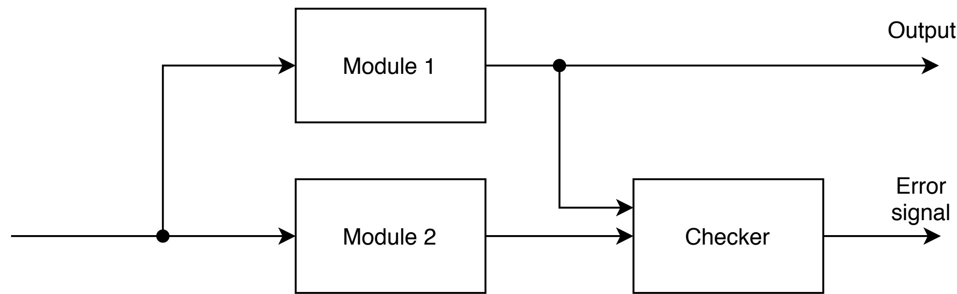


Figure 2.11: Duplication with comparison

Under the passive redundancy techniques, N-modular redundancy (NMR) (figure 2.12) allows both error detection and error correction at an extra cost of hardware area and power. NMR uses voting entities that dictate the correct output from the redundant modules. The main problem with this technique is that, although it can mask N module faults, if a fault happens on the majority of the modules, the voter would produce an erogenous result [8]. Figure 2.12 presents an example of NMR, particularly the triple module redundancy configuration which uses three identical modules, performing identical operations, with a majority voter determining the output.

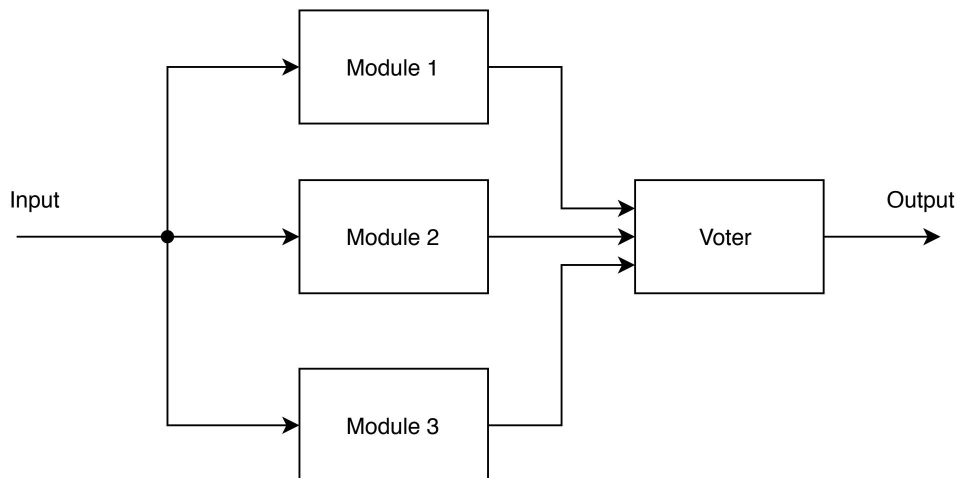


Figure 2.12: Triple Module Redundancy

Software Redundancy

Software redundancy can be divided into two groups: single-version and multi-version software techniques. Single version aims to improve the fault tolerance of a software component by adding to it mechanisms for fault detection, containment, and recovery. Multiple version uses redundant software components which are developed following design diversity rules. This mechanisms that mitigate faults can be

applied to different software layers and software elements [17], such as at the operating system level, the application level, the process level, object level and function/method level.

Single-version techniques are based on redundancy applied to a single software version to detect and recover from faults. Such techniques include error detection, exception handling, checkpoint and restart, process pairs and data diversity.

Multi-version techniques, also called design diversity approach, concerns developing two or more versions of the same software component. This mirrored software components can be executed either in sequence or in parallel and can be used as alternatives (containing different error detection methods), in pairs (different means of error detection by replication), or in larger groups (to enable masking through voting). The thought behind using multiple versions is expecting that components built differently (i.e., different designers, different algorithms, different design tools) should fail differently [18]. Thus, if one software component fails to output the correct value, there is at least one more alternative version that may be able to provide a correct output. Multi-version techniques include Recovery Blocks, N-Version Programming and N-Self Checking Programming.

Computation Redundancy

Computation redundancy involves repeating the computation or data transmission two or more times and comparing results with previously stored copies. The outputs are verified if they match, and, if they do not match, one can assume that an error occurred. This type of redundancy is effective mainly against transient faults. Because the majority of hardware faults are transient, it is unlikely that the separate executions will experience the same fault. Computation redundancy can thus be used to detect transient faults in situations in which such faults may otherwise go undetected. There are two ways to apply this type of redundancy: (1) by having the same computation unit repeat the specific computation in a different point in time during program execution; (2) have a second computation unit perform the same computation at the same time. Comparing both methods, the first one has much lower hardware and software overhead but suffers a high-performance penalty, while the second one uses the additional hardware in favor of performance. The main problem with computation redundancy is the assumption that the data required to repeat a computation is available in the system [19]. Since a transient fault may cause system failure, the computation may be difficult or not possible to repeat. An example of a computation redundancy technique is Lockstep [20], where two processors run, at the same time, the same software and provide feedback when the computations mismatch between processors.

Information Redundancy

Information redundancy adds some redundancy to the original data to tolerate errors. The most common form of information redundancy is coding, which adds check bits to the data, allowing to verify the correctness of the data before using it and, in some cases, even allowing the correction of the erroneous data bits [21]. There are different forms of information redundancies such as parity codes, checksum, linear codes and cyclic codes. One information redundancy form that is widely used in applications for harsh environments is the error-correcting code (ECC). This technique protects memory modules from radiation that may cause bit-flips. Usually, the ECC maintains a stored data immune to single-bit errors, ensuring that the read data is the same that was previously written.

Fault Tolerant Architectures

It is worthing noting that although redundancy is required to achieve reliability, it is not sufficient to just put a group of components together in a "fault tolerant" configuration. How redundancy is managed is as important as the redundancy itself in order to contribute for higher reliability [17]. For this reason, both the software and hardware architectures must support the redundant mechanisms. Such architectures are largely applied on avionics [17][22], life support [23], aerospace [22][23], and, lately, on the automotive sector [24].

The usage of this type of architectures is connected with an increase in cost and/or complexity as well as synchronization problems [25]. This is the main reason why both hardware and software architectures must manage redundancy well. A redundant architecture can have several modules with the same functionality. These modules can communicate with each other and make decisions depending on the exchanged data. This implies that redundant modules present at least on channel of communication between them. Lack of solid synchronization mechanisms can disrupt the interactions between redundant systems, defeating the purpose of redundancy.

One example of such architecture is the flight control computer of the Boeing 777 (figure 2.13). Redundancy is present at both the computation modules and data buses. The computation modules (or channels) are tripled and connected to each one of the three data buses, despite each module only outputting data to its specific lane but reading from all three. This setup enables the channels to communicate with each other without the possibility of one bad channel interrupting all the communications. Internally, each computation module contains three processing units, in a command-monitor-standby arrangement, where one unit writes to the bus while the others monitor its operations. The processing units present

different processors architectures and different application software between them. Differences on architecture can be regarding processor architecture and different component manufacturers, while on software can be differences on computation algorithms or even compiler technology used. These internal modules communicate between them, allowing for sanity checks and rapid reconfiguration of a processing unit in case of failure [17]. When one processing unit is declared bad, it is taken offline and one of the spare processing units assume its functionality.

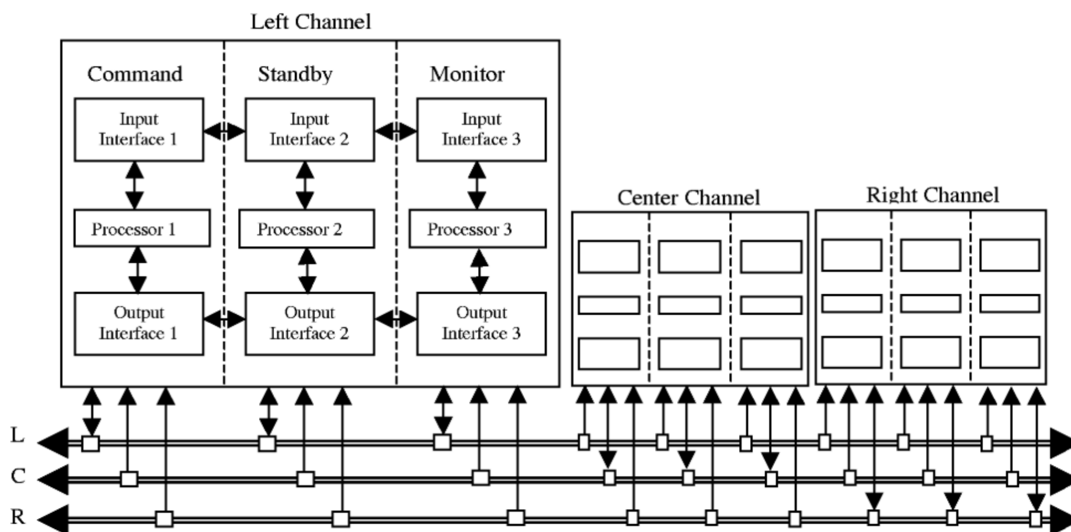


Figure 2.13: Architecture of B777 Flight Control Computer [17]

2.3.6 Reliability System Development

Reliable systems do not have a determined development method nor a procedure to follow in order to reach a certain level of reliability. Instead, there are guidelines provided by some authors, such as [15], that focus on the techniques that aid reliability design, prediction and testing. To assist reliabble system development, the ESRG adopted the development flow presented in figure 2.14. It contemplates five development phases, supported by development methodologies and phases from embedded development flow.

Similar to the embedded development flow, the reliability flows starts by gathering requirements and conceptualizing the system application. After this, the design phase is where the system architecture is modelled, along with the development of both hardware and software architectures. This phase is very similar to the embedded system flow as shown on figure 2.15.

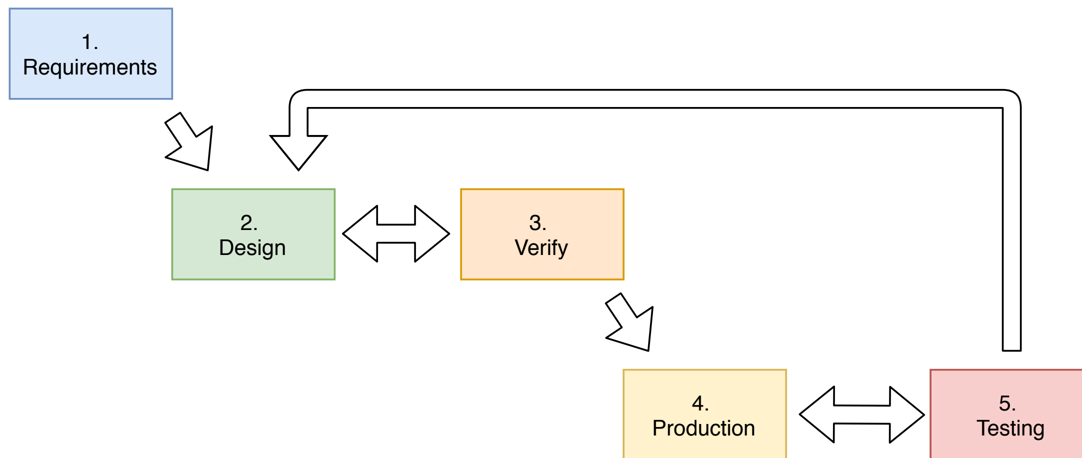


Figure 2.14: Example of development cycle of a reliable system

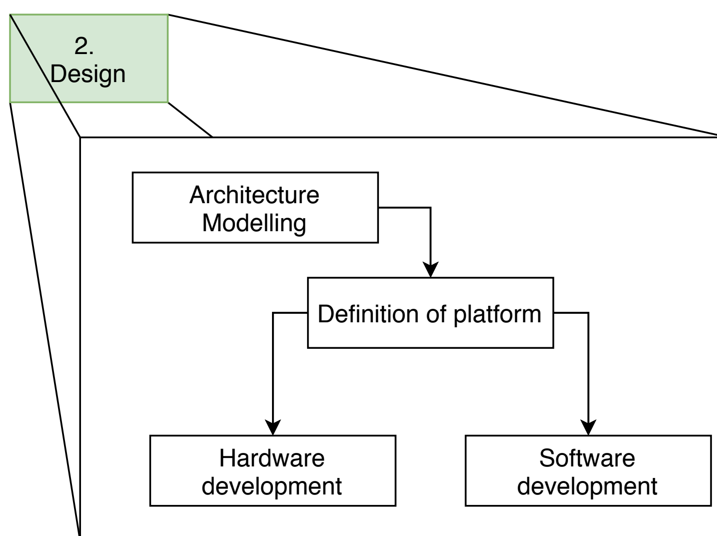


Figure 2.15: Design phase of the development cycle of a reliable system

Depending on the requirements, the system must meet specific reliability metrics, and, for that purpose, fault tolerance techniques such as redundancy are applied to augment overall system reliability. It is mainly during the design phase that these techniques are applied, both on hardware and software. When a system design, considering both the resulting software and hardware architectures, is assumed to meet reliability requirements, it goes through verification to have an insight about its reliability metrics. Only after architecture design, on the verification phase, is that the resulting architectures are checked if they fit the reliability requirements.

Since up until the verification phase the development of both hardware and software is decoupled, the estimation of the reliability metrics should also be decoupled. For this reason, **simulation** comes as a tool to provide an **estimation** of the reliability metrics of both the software and hardware architectures

[13]. Using simulation on reliability development carries the main advantage of providing an early reliability metric estimation, allowing design reiteration without additional costs later on the development cycle. The estimations, done on hardware and software, can be supported by methods such as Monte Carlo (section 2.4) and techniques such as Fault Injection (section 2.5).

The resulting simulation-based estimations are checked with the initial reliability requirements and the violation of the these requirements causes new design iterations, until a good enough solution is found. Upon having both an hardware and a software architecture meet the reliability requirements, a new estimation is made contemplating the results of both architectures. Once again, if this estimation does not match requirements, the system must go through another design iteration. Such reiterations can make the development go back to as early as the first steps of the Design phase, in order to find a better solution that fits all reliability requirements, as shown in figure 2.16.

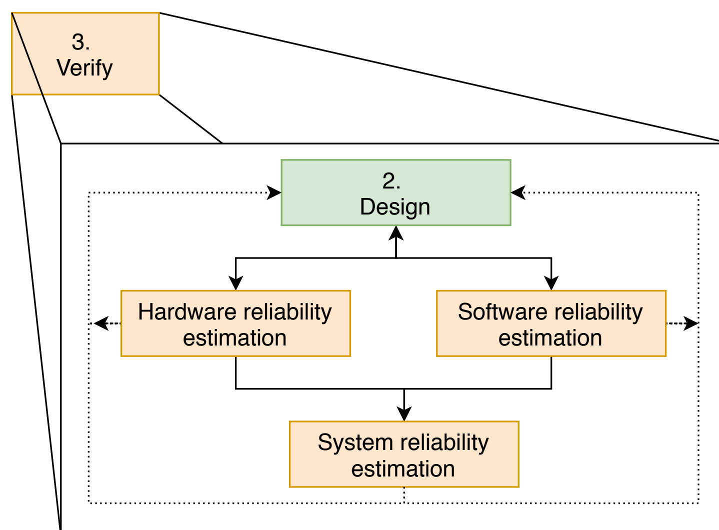


Figure 2.16: Verification phase of the development cycle of a reliable system

Upon having good reliability estimates, the hardware can be produced and the final architecture can be put into test. As simulations are not 100% accurate and there are hardware manufacturing factors that affect the quality of the components, testing aims to provide not only a more accurate reliability metric but also validation of the functional requirements. The typical tests made are *accelerated life tests* which use stress factors such as temperature and vibration to provide an early system failure rate. Once again, the reliability metrics upon testing are checked with the initial requirements, and if they do not meet them, the prototype goes once again into production, if the manufactured component does not match specification, or even back into design, if the real prototype happens to not meet reliability requirements

even with flawless production quality. Development ends when the final prototype is able to perform all the functional requirements and address the reliability requirements.

2.3.7 Reliable Software Development

Software reliability is affected not only by the fault tolerance mechanisms implemented, but also by the quality of the software itself. Unlike hardware, which has associated reliability metrics such as failure-rate and mean time between failures from manufacturers, software does not present such predefined metrics. It differs from hardware reliability in that software reliability reflects the design perfection, rather than manufacturing perfection.

Software reliability is hard to achieve, due to the high complexity that software tends to have. Highly complex system, with software, are harder to reach a certain reliability level, but even in this case, complexity tends to still be pushed into the software layer, resulting on a rapid growth of system size. While the complexity of software is inversely related to software reliability [26], it is directly related to other important factors in software quality and functionality. Emphasizing these features will tend to add more complexity to software.

Increasing complexity leaves the system more prone to design errors that may resonate on errors during development. Software failures may be due to errors, ambiguities, specification misinterpretation, incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems [13]. To mitigate these type of issues, software should adhere to stricter standards and developers should adopt a meticulous approach to software development [27].

While no software development process or certification standard can guarantee software reliability, attention to and prevention of typical software errors and application weaknesses can significantly reduce possible development errors and help developers better understand potential risks. Examples of practices that can help with the latter are using coding guidelines and consistent programming rules, so it can be easier to read, verify and test the software. Also, using pre-defined interfaces giving possibilities for reusability and reused components seem to strengthen the quality. This is also the case with standardization, whether it concerns components or layers. When a standard becomes well-known to most developers it becomes easier to work with, causes fewer mistakes, and for this reason reduces the complexity and increases the reliability.

One of the programming standards that aims toward a more reliable development is MISRA-C. The first draft on this standard came public in 1997 by the Motor Industry Software Reliability Association, and

proposed a set of software development guidelines that promote safety, reliability, ease of maintenance, and portability for safety-critical systems. At the moment, MISRA-C has evolved to its third-generation guideline (MISRA-C:2012), and has come to be widely adopted as practice that improves the reliability, maintainability and portability of all software, not just safety critical. The guidelines range from simple usage of brackets for easier code readings to prohibition of language keywords, such as the *goto* (from C language).

AUTOSAR

AUTOSAR (AUTomotive Open Software Architecture) is an example of a standard that not only dictates a software architecture, but also development methodologies and templates, conformance test suites and application interfaces. The technical goal of this standard is to achieve scalability, portability, modularity and ease of complexity management of automotive systems.

Regarding the architecture, its layered layout offers the mechanisms needed to allow software and hardware independence. It distinguishes between three main software layers which run on a Microcontroller (MCU): Application Software, Runtime Environment (RTE) and Basic Software (BSW). The diagram on figure 2.17 presents the layered architecture. The Basic Software layer contains all the device drivers and all the abstraction for the microcontroller and the Electronic Control Unit (ECU) of the vehicle. The RTE abstracts the application layer from the basic software and organizes the data and information exchange between them. The application layer above the RTE contains the application specific software components, which are completely ECU-independent and work without specific knowledge of the used hardware.

The architecture's definition, internals and development guidelines are integrated with MISRA-C. These guidelines, made public by the AUTOSAR association, provide the developer the needed information to implement the correct interfaces and behaviour of the modules, alongside with the needed MISRA rules to follow upon coding. A few examples of rules and guidelines, particularly affecting the MCAL layer, are presented on appendix A.

The standardization and MISRA rules adoption are two reasons that this architecture is considered to promote reliable system development. Although this does not seem a direct way to augment system reliability, the usage of such architecture and guidelines provides a good foundation for reliable software development. This is backed by the conclusions made in [28], which admits that not only the system complexity is more controllable due to the use of standardized and well defined interfaces, specifications,

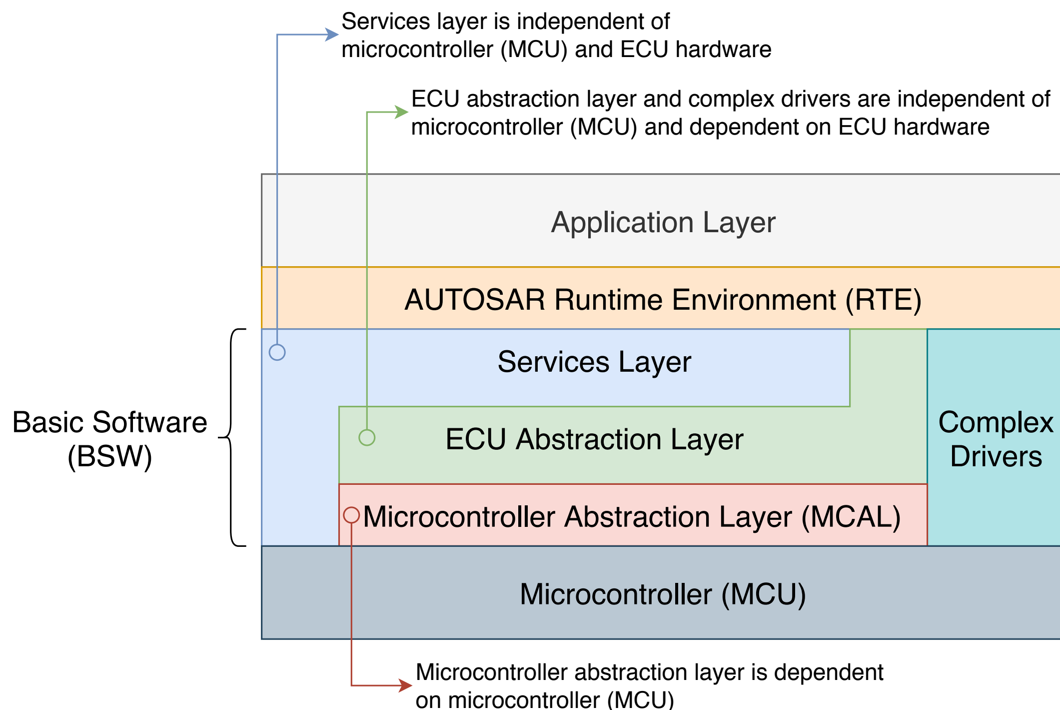


Figure 2.17: AUTOSAR layered architecture

and processes, but also that the reliability is increased as defined specifications give less human erroneous interference.

2.4 Monte Carlo Simulation

Although reliability metrics can be calculated by traditional methods, complex systems with large number of different components makes calculation impractical. Some techniques and methods can be used to synthetize a prediction of such metrics, avoiding the inevitable hard and time consuming work that normally would come with traditional methods. The Monte Carlo method fits the available techniques by providing numerical estimation of an unknown parameter or metric by the mean of repeated sampling.

Monte Carlo methods may vary, but tend to follow a process which starts by the definition of a domain of possible inputs, followed by generation of random inputs from a probability distribution over the domain. After creating the input sets, a computation is made using the specified inputs and the results are aggregated for later analysis. The diagram on figure 2.18 shows the process taken when performing this method. On the example, suppose the function whose probability of success to be estimated is $y = y(X_A, X_B, X_C)$, and that X_A, X_B, X_C are the chosen variable domains whose distributions are $P(X_A), P(X_B), P(X_C)$. The procedure is to pick a set of X's randomly from the distributions, calculate

y for that set, and store that value. This is repeated many times until enough values of *y* are obtained to create a value distribution. In order to have signification results, the process of generating random inputs, computing and getting results is repeated a large number of times, this is the reason why this process is usually made in simulation environment.

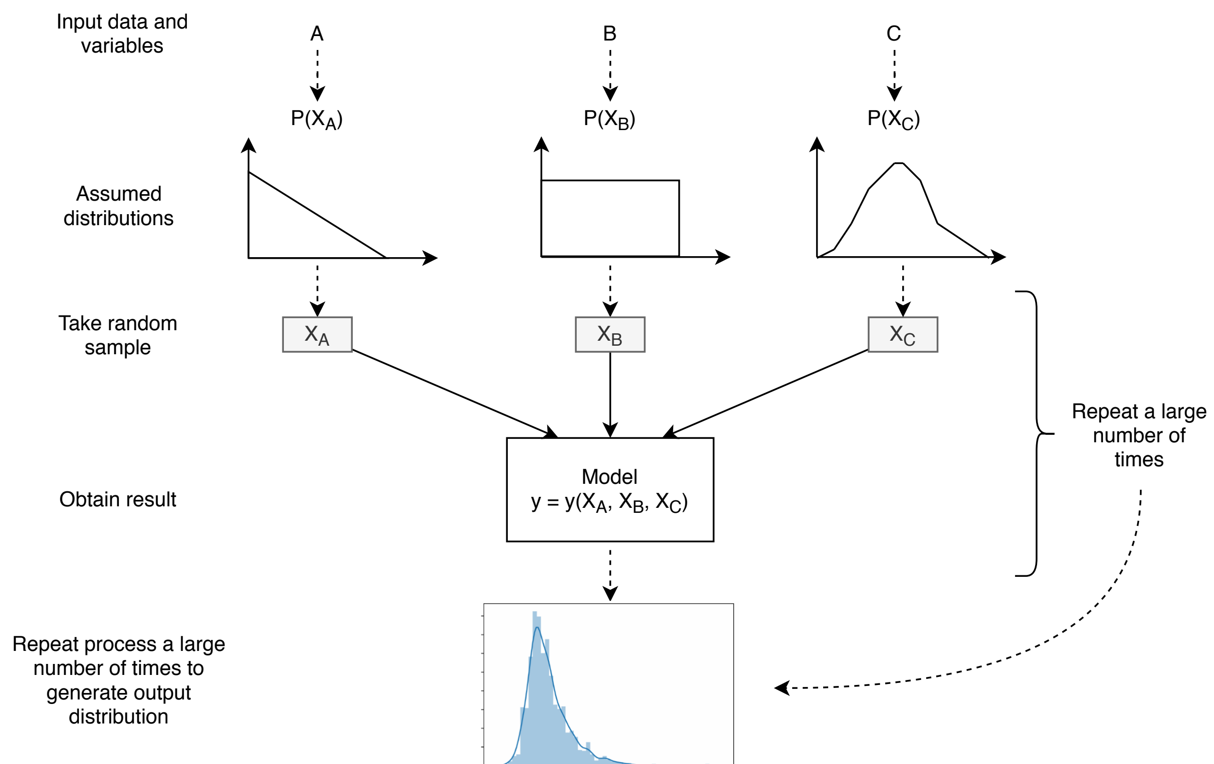


Figure 2.18: Monte Carlo method process

In principle, this method can solve any problem that has a probabilistic interpretation. The underlying principle is the law of large numbers, which states that the larger the sample the more certainly the sample mean will be a good estimate of the population mean [15].

This method is applied is several areas from physical sciences to finance and business, but the focus here is the appliance on reliability engineering. In this area, Monte Carlo method is used to compute system-level response given the component-level response. By feeding inputs based on probability distribution to the system, the system probability of success can be predicted from the individual component probabilities [15]. For example, a system failure rate prediction can be synthetized by having the inputs follow a probability dictated by a failure rate distribution, such as the one presented on 2.10. Under reliability engineering, this method can also be combined with fault injection techniques, in order to gather insight about system-level behaviour, as will be seen on section 2.5.

An example of a Monte Carlo simulation in reliability engineering context to evaluate system failure probability, consists in running a large number of repetitive trials and changing component states according to a probability distribution. On each one of these trials, there are time steps which represent advancement of system lifetime. On each of these steps, component states are changed by generating random numbers and comparing them to the components failure distribution. If the random value is lower than the component failure rate at the given time, component state is changed, otherwise no change is made. The process is done until system failure, and the resulting time step is stored. As the trials are done several times, the time step results of each trial will create a system failure distribution, which approximates to the real system failure rate, with an uncertainty.

The number of trials done during the simulation can vary as the number required to have a good estimation is not well defined. These trials can take considerable computer time that may not fit under the development time budget. If time is scarce, such method may not fit during project lifetime. A solution to such problem is to only perform a limited number of trials, then apply ensemble techniques to have better results with less available data. These techniques aim to combine resulting data from multiple models instead of using a single one. Performance is increased since this techniques help minimize bias and variance of the data [29]. Examples of ensemble techniques are Bagging and Boosting.

2.4.1 Bagging and Boosting

Both Bagging (Bootstrap Aggregating) [30] and Boosting [31] are ensemble methods that use multiple weak models to build a strong one. These ensemble techniques have the advantage to alleviate the small sample size problem by averaging and incorporating over multiple models to reduce the potential for overfitting the initial data. Each model is fed a different set of data, which is dictated by either *bootstraping* the original data or by attributing weights to each sample. By having multiple models with different data, there is a better overview over the classical bias/variance tradeoff, which may by hard to reach with only a single model.

Regarding Bagging, input data is drawn by the *bootstraping* method. This method consists in generating sets of size B from an initial set of data N, by randomly drawing with replacement B samples. This means that in new sets, due to the fact that the original set is randomly sampled, some observations may be repeated. After creating the input sets, each of them is fed to a different model, being the number of models equal to the number of bootstrapped sets. The result of each model is than averaged for a final prediction result [32]. A diagram of the process is presented on figure 2.19, where the blue circles

are the samples chosen as model inputs, after bootstrapping. This techniques mainly aims to reduce the variance of the results.

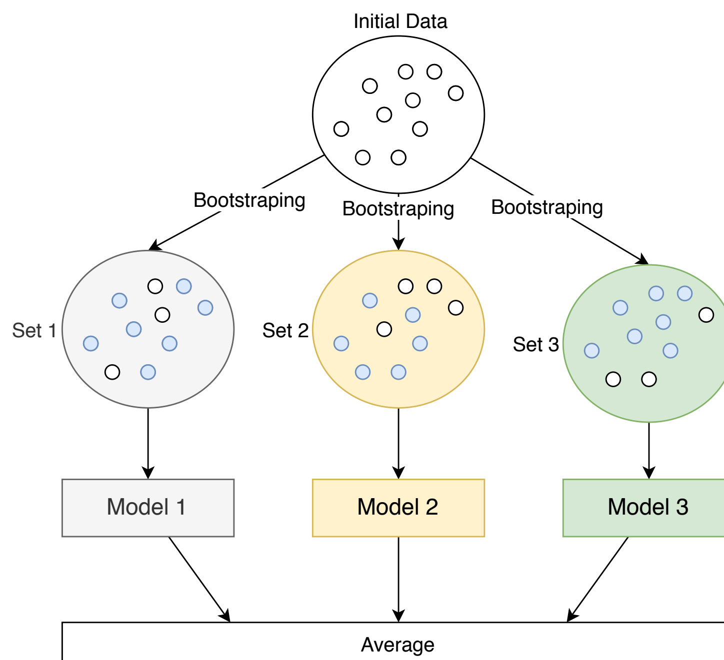


Figure 2.19: Bagging method diagram

Contrarily to Bagging, where each model runs independently and then the outputs are aggregated at the end without preference to any model, in Boosting each model that runs dictates what data the next model will focus on. In this technique a weight distribution is maintained over the samples, and it is adjusted at each iteration. The adjusted weights are chosen so that higher values are associated to previously lower performance estimation [32]. Upon having model results, a second set of weights is assigned to each model, in order to make a weighted average of each of the model results. Similarly to Bagging, data variance is also decreased but Boosting has a better bias error reduction. A diagram of this process is presented on figure 2.20.

The main difference between these methods resides on how the samples are drawn from the original set. In the case of Bagging, any element has the same probability to appear in a new data set. However, for Boosting the observations are weighted and therefore some of them will take more importance in the input data.

One of the most common boosting algorithms was proposed in 1995 by Freund and Schapire [31] and its called Adaptive Boosting, or AdaBoost for short. It was originally designed for classification problems, but its usage can be extended to regression as well as other statistical problems. This algorithm is described next.

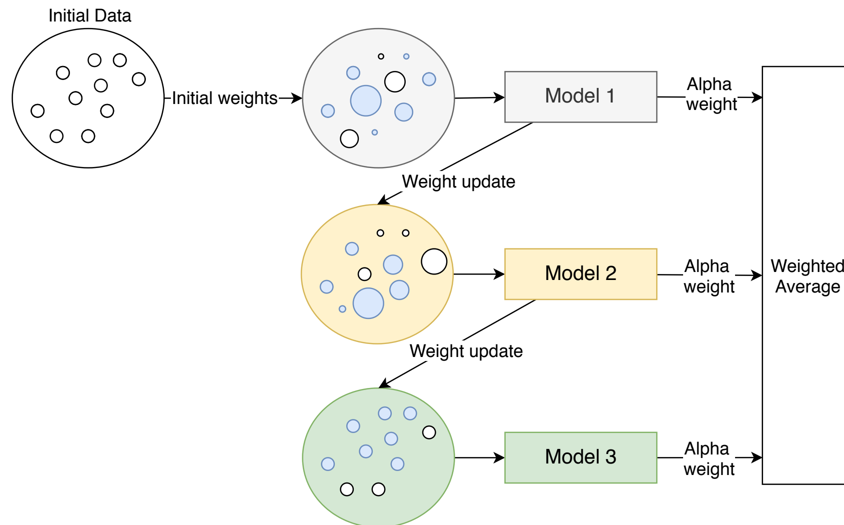


Figure 2.20: Boosting method diagram

From a data distribution of $\{X_i, y_i\}$ where X is the model input data and y the hypothesis result of the input data:

1. Initialize the observation weights $w_i = 1/N, i = 1, 2, ..., N$.

2. For $m = 1$ to M:

3. (a) Fit a model $G(x)$ with the input data using weights w_i.

   (b) Compute

$$err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$$

   where $I(y_i \neq G_m(x_i))$ is the model result compared with the hypothesis result.

   (c) Compute $\alpha_m = log((1 - err_m)/err_m)$.

   (d) Set $w_i \leftarrow w_i exp(\alpha_m I(y_i \neq G_m(x_i)), i = 1, 2, ..., N$.

4. Output the weighted average of $G(x)$

2.5 Fault Injection

The goal of fault injection is to provoke (inject) faults, or stimulus, that are as close as possible to real faults that could occur on real hardware. This technique can be used to stimulate systems in order to determine whether the system response matches its fault-tolerance specifications. Normally, faults are

injected in perfectly chosen system states and points, previously determined by an initial system analysis, in order to gather knowledge about system behavior under controlled conditions. Performing this type of test yields several goals [33]:

- An evaluation of the efficacy of the fault tolerance mechanisms included into the target system and thus a feedback for their correctness (e.g., for removing designs faults in the fault tolerance mechanisms);
- Estimating the failure coverage and latency (i. e timing) of fault tolerant mechanisms;
- Identifying weak links in the design, such as single point failures of the system within which a single fault could lead to severe consequences;
- Studying the system's behavior in the presence of faults, for example propagation of fault effects between system components or the degree of fault isolation and determining the coverage of a given set of tests;
- Evaluating system reliability, in which safety-critical systems are tested with fault-injection.

Four main types of fault injection techniques exist: hardware-based implemented, simulation-based, software implemented and hybrid. Hardware-based is done at physical level, disturbing the hardware with parameters of the environment (heavy ion radiation, electromagnetic interferences, etc.), injecting voltage sags on the power rails of the hardware (power supply disturbances), laser fault injection or modifying the value of the pins of the circuit [33]. Software-based fault injection consists on reproducing, at software level, the errors that would have been produced upon occurring faults in the hardware. Simulation-based consists in injecting fault at high-level models (e.g. VHDL or Verilog models). Hybrid techniques mix software implemented fault injection and hardware monitoring. Each of these techniques have advantages and drawbacks. For the sake of having an overview of each technique, a compilation of advantages and drawbacks of each technique is presented on table 2.1.

A fault injection environment typically consists of several components such as fault injector, fault library, controller, data collector and data analyzer [33]. The fault injector evokes faults into the target system, as specified in the fault library. The fault library stores fault specifications such as fault type, locations and time. The controller runs on the injector host injecting faults and controlling the experiment by using a hook at the target system. In order to verify that the injection really caused a fault and to record other data, a readout collector sends measurements to the data collector. The data analyzer performs

Table 2.1: Summary of main advantages and disadvantages of fault injection techniques.

Types	Advantages	Disadvantages
Hardware-based	• Can access locations that is hard to be accessed by other means. • High time-resolution for triggering and monitoring. • Well suited for the low-level fault models. • Experiments are fast. • No model development or validation required.	• High risk of damage to hardware. • Low portability and observability. • Limited set of injection points and limited set of injectable faults. • Requires special-purpose hardware in order to perform some experiments.
Software-based	• Can target applications and operating systems. • Experiments can be run in near real-time. • Does not require any special-purpose hardware. • No model development or validation required.	• Limited set of injection instants. • It cannot inject faults into locations that are inaccessible to software. • Does require a modification of the source code. • Limited observability and controllability.
Simulation-based	• Can support all system abstraction levels. • Full control of both fault models and injection mechanisms. • Low cost computer automation. • Does not require any special-purpose hardware. • Maximum amount of observability and controllability.	• Large development efforts. • Time consuming (experiment length). • Model is not readily available. • Accuracy of the results depends on the goodness of the model used. • Model may not include design faults that may be present in the real hardware.

data processing and analysis. A block representation of fault injection components is presented in figure 2.21.

Fault injection has been already been explored on multiple system levels, from software level throughout RTL level. In [34] fault injection was used to emulate software faults, in order to have an overview on how injected faults compare with real software faults. Regarding reliability evaluation, the authors of [35] and [36], used this technique to evaluate robustness of digital circuits against faults, at RTL level. Also, in [37] and [38], frameworks were presented that allowed to perform fault injection, on VHDL designs and on software running on emulated hardware, respectively. The latter is rather important since the methodologies and work developed highly contributed for the work done on this dissertation.

As mentioned before, fault injection has been considered very useful to evaluate reliability of a system. By injecting faults into an operational system, it can provide information about the failure process, which means that metrics such as mean time between failures (MTBF) can be taken from fault injection results.

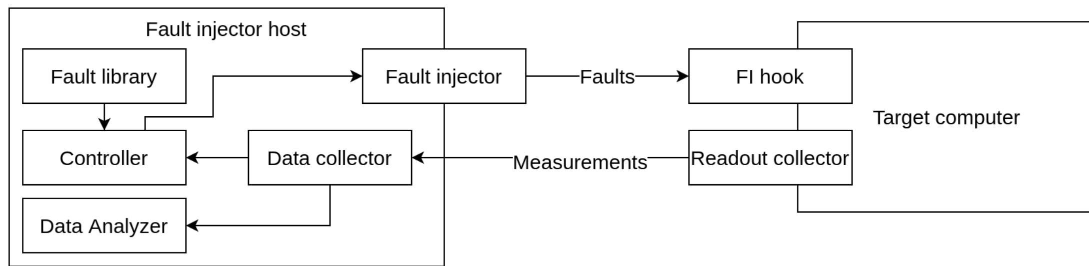


Figure 2.21: Typical fault injector architecture

2.6 Embedded System Simulation

Much like in every development project, the engineering effort needs to be validated, and embedded development is no exception. Although, testing embedded systems, mostly embedded software, is different than testing conventional systems since this type of systems have a close integration of hardware and software.

As mentioned early on this chapter, embedded systems are tightly coupled, meaning a close integration of hardware and software. A single embedded system can contain user designed specialized hardware, a processing unit used to control that hardware and to process and retransmit the acquired data, and software running on that processor. Testing of all listed parts separately is difficult, or even impossible [39].

An approach to test this type of systems is to test directly on the developed hardware but it presents several downsides. One of them is the chance of damaging the hardware, increasing costs of new hardware or even redesign costs. Furthermore, hardware dependency and the fact that the embedded software is often developed in parallel with the hardware, may lead to stall in testing since there is no physical hardware to test the software.

Another reason is that it may be difficult to distinguish software bugs from hardware bugs as the modules may not been previously tested independently. Also, defects are harder to reproduce in embedded systems, so embedded testing process needs to gather as much information as possible, in order find the root of the defect. Combined with the very limited debug capabilities of embedded products, that gives testing another challenge.

Such challenges have led to wide development of adoption of various simulation-based development and testing approaches in the embedded software industry called X-in-the-Loop [40].

2.6.1 X-in-the-Loop

X-in-the-Loop (XiL) is an integrated in-loop method where X refers to the unit under test, which can be a model (model in the loop, MiL), software (software in the loop, SiL), processor (processor in the loop, PiL) and hardware (hardware in the loop, HiL). These methods have gained acceptance due to the increased adoption of model–based development (MBD) industry, especially in the automotive domain [41]. They are widely used on V cycle development and each phase uses a particular method as shown on figure 2.22.

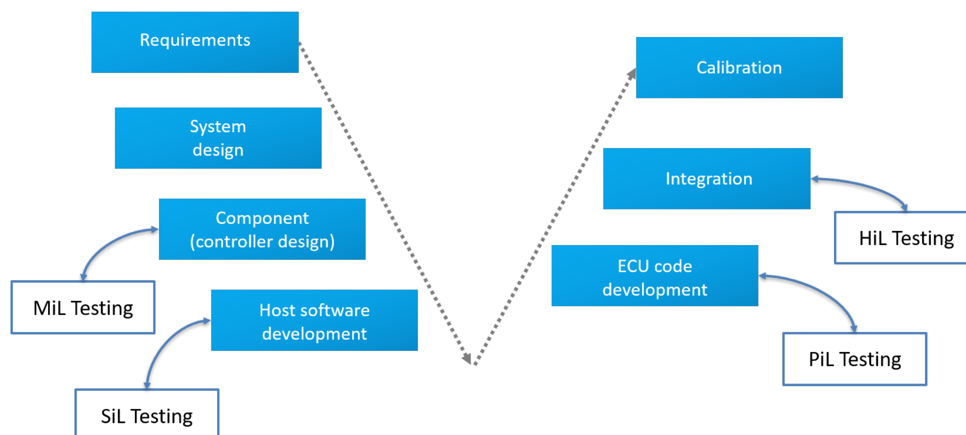


Figure 2.22: X-in-the-loop testing applied on reliability-aware development cycle (adapted from [41])

Model-in-the-Loop happens early on the development phase. It aims to validate the interactions of the system to develop with the environment, i.e, the environment stimuli (such as signals) effects on the system. It abstracts system-specific behaviour in order to correctly validate the model to be used. A simple example are Finite State Machines, in which the model is the machine itself and the state transitions are the responses to the environment.

Upon model verification, the software-in-the-loop phase aims to test the software algorithms that will run on the target platform. The testing is made on a host platform which usually greatly differs from the target platform. This process can be fully abstracted from the target hardware since test is made regarding the correctness of the algorithm. On this phase, all hardware-specific functionalities (e.g. hardware-accesses) are substituted by host-specific mechanisms that allow to emulate read/writes from the real hardware, e.g. offline caching.

The Processor-in-the-Loop phase uses an emulated target platform to execute the algorithm. In PiL,

the target processor, both characteristics and behaviour, is emulated in a simulation environment, allowing to achieve more realistic simulation results. The simulation environment is still integrated in the host platform, but the algorithm is executed as it was already on the target processor. The simulation environment can be seen as layered where: the outer layer is the host PC and the operating system, the next layer is the simulation environment, e.g. KEIL Simulator by Arm, and the last ones are the emulated processor and the algorithm running on it. A good simulation environment can emulate the processor in such way that one can not tell the difference between running the application by emulation or on real hardware.

Lastly, in Hardware-in-the-Loop, the target hardware running the algorithm is connected to a real-time simulation that simulates real signals. It uses the hardware platform's IO ports to interface with the simulation environment. This type of testing is particularly beneficial since the tests can be made fearlessly and comprehensively without risk to a physical, costly system.

For a better understanding of each X-in-the-loop method, table 2.2 presents the entities that are tested on each phase and the interfaces used for each test method.

Table 2.2: Types of X-in-the-Loop testing

Types	Entity Under Test	Test Interfaces
Model-in-the-Loop (MiL)	System model	Behaviour and events of the model e.g: FSM transitions
Software-in-the-Loop (SiL)	Control software (e.g., C code)	Methods, procedures, parameters, variables
Processor-in-the-Loop (PiL)	Binary code on a host machine emulating the target processor	Register values and memory contents of the emulator
Hardware-in-the-Loop (HiL)	Binary code on the target architecture	I/O pins of the target microcontroller or board

Simulating a system has always carried the advantage of increased insight and flexibility, at a cost in execution speed and timing fidelity comparatively to the real machine. However, this has not always been the practice, since the use of simulation technology for large-scale embedded systems software development and testing has been relatively limited up until a decade ago [42].

In this context, focus will be made on two types of simulation: instruction-accurate, which mimic the behavior of a microprocessor, and cycle-accurate, which are optimized for precise simulation of hardware components. The first type of simulators allow to test the user software, while providing functional simulation of the hardware by creating the emulated platform. QEMU [43] is a simulator that fits this category, allowing both instruction-accurate simulation of the software and hardware behaviour emulation. The second type of simulators offer cycle-accurate simulations that accurately simulate hardware behavior.

This kind of simulation is generally used to validate hardware components themselves. Under this type of simulators, GHDL, Icarus Verilog and ModelSim are examples of hardware simulators which provide cycle-accurate Register Transfer Level (RTL) simulation.

2.6.2 Full System Hardware Simulation

A simple approach to simulate the whole system, including embedded system and newly developed hardware, is to use an hardware simulator. Although it can be done, the performance is usually very poor [39], as the hardware level simulators are not suited for simulation of such complex systems as embedded systems. In this type of simulation, the states of all logic gates and registers are simulated, which is not needed to provided the needed accuracy to run software. A solution for this slowdown would be to avoid software simulation in hardware simulators altogether, running software parts externally of hardware simulation. Figure 2.14 presents a diagram of full-system hardware simulation on a development host.

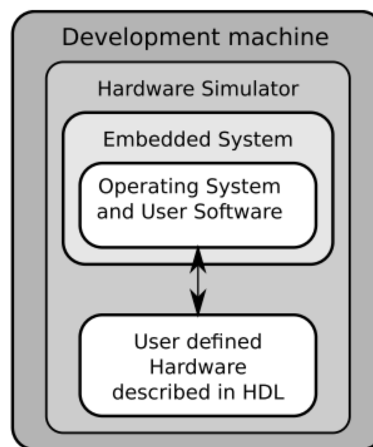


Figure 2.23: Simulation of a system using an hardware simulator [39]

2.6.3 Hardware Simulation with Host Software

A different approach is to use an hardware simulator to emulate the user designed hardware, while the user software runs directly on the development machine. The diagram on figure 2.24 shows a hardware simulator being provided with stimuli from software that is being executed on the host development machine.

The software application that runs on this simulator is compiled for the host development machine, replacing device driver system calls (or hardware register access) with Application Program Interface (API)

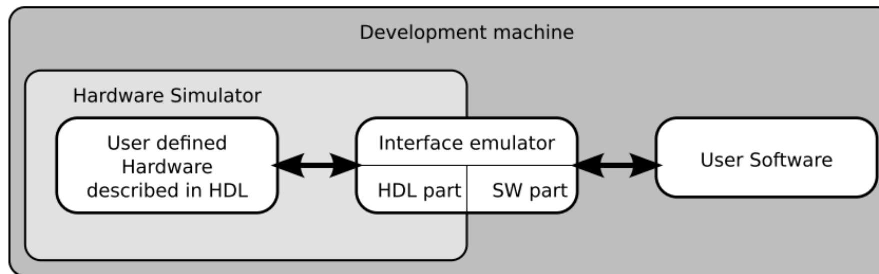


Figure 2.24: Hardware simulator with software on host [39]

calls that allow to communicate with the hardware simulator, emulating system bus transactions. The interface must also be implemented on the hardware simulator side, by providing access to the software calls. This approach is significantly better than full system hardware simulation, as native performances are achieved for software with hardware simulation only requiring hardware that is relevant to the scenario. However, in most cases, the development includes testing of both hardware and low level software components such as device drivers, performance of data transfers and so forth. To emulate these elements, software simulation must be combined with hardware simulation. The act of simulation such different domains such as software and hardware is called co-simulation, which will be discussed in great detail in section 2.7.

2.6.4 Full System Software Simulation

On full system simulation approach, the hardware is emulated on the development machine, while running the user software. Figure 2.25 presents a diagram of full system software simulation. In this approach, hardware modules are functionally emulated and integrated into software simulation. Depending on the simulator used, the hardware modules may or may not be available for the target machine, meaning that the simulator chosen must meet the application scenario or allow to extend the list of supported hardware modules, like QEMU. Having the hardware model behavior emulated is a very useful feature for design space exploration early in the project [44], allowing validation before any commitment to an hardware component or HDL implementation. Furthermore, this does not only allow not only for a flexibility that suits design space exploration very well, but also concurrent development given that software design teams may start device driver development concurrently with hardware design teams. For such reasons, this approach will be used in this dissertation.

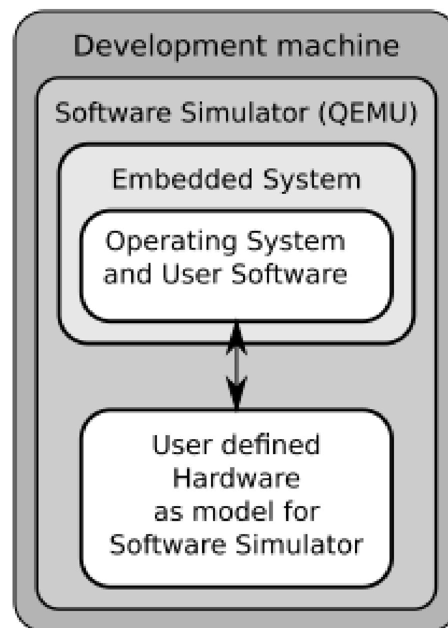


Figure 2.25: Full system software simulation [39]

2.7 Co-simulation

Truly complex engineered systems that integrate physical, software and network aspects, are currently used in several development areas [45]. Due to time-to-market pressure, the development of these systems has to be concurrent and distributed, that is, divided between different teams and/or external suppliers, each in their own domain and each with their own tools. Each participant develops a partial solution within the domain that they have the skillset for, being software, analog circuits, FPGA's or mechanical parts. Figure 2.26 represents the domains that typically integrate the development of an embedded system.

The teams that tackle each domain under a complex system work on different abstraction levels, whereby each team, use simulators that work within the abstraction level necessary for their domain [45]. This means that, for example, teams that work on the software domain, simulate source code execution, while teams that work on the hardware level, run simulations at analog circuitry level. By using simulators with only the necessary functionality for the domain in question, simulations can be fast enough to achieve near real-time execution.

Typically, within a complex system, models developed in different domains are independently validated, meaning that no real interactions exist between them. Although testing is independent, the models need information from other domains to have meaningful simulation results. In order to validate the model

Embedded Development Domains

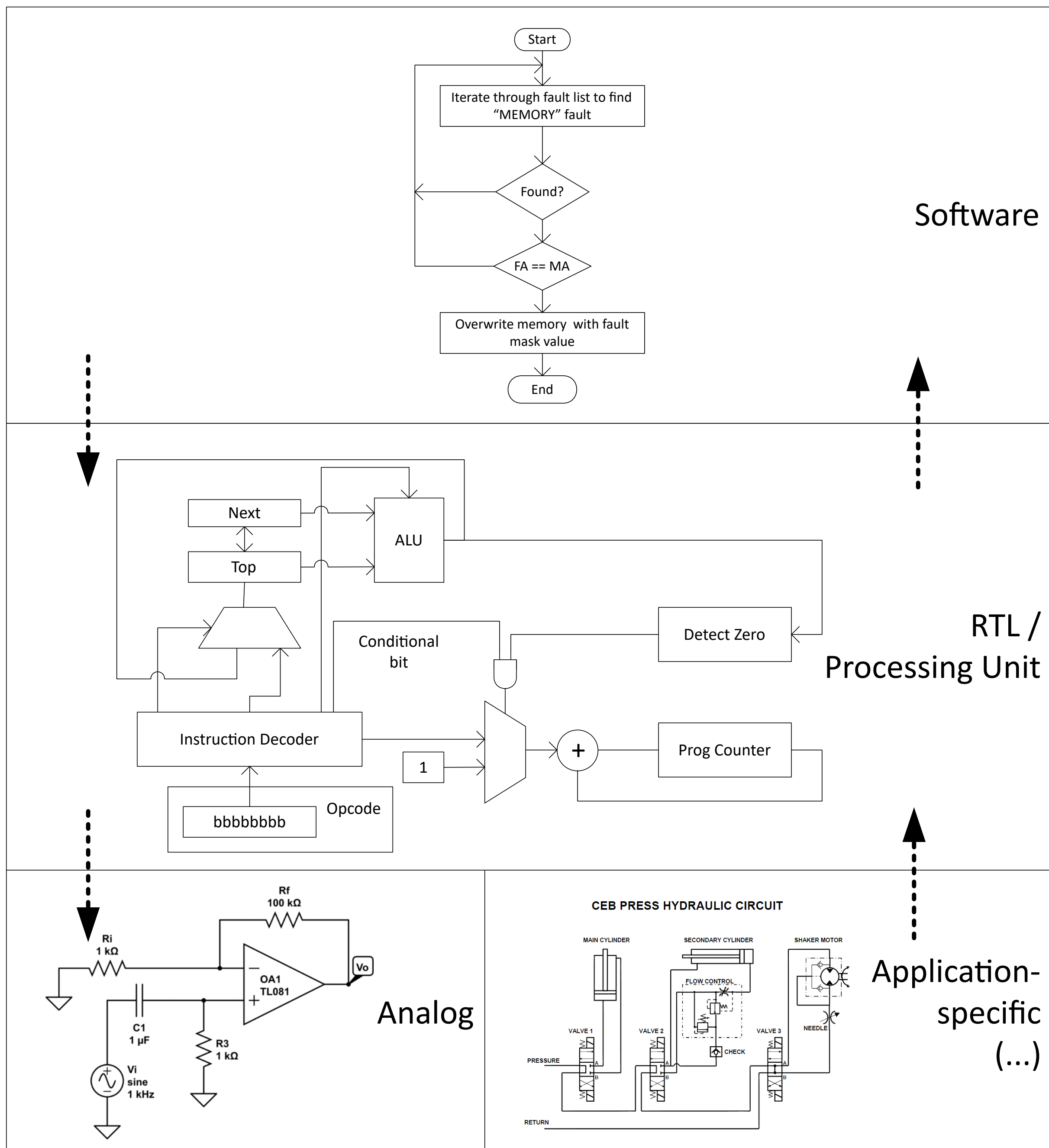


Figure 2.26: Domains of a complex system

behaviour, static stimuli are used with the help of local caches, which store typical stimuli that occur on the target scenario. Figure 2.27 presents a diagram that represents a single domain simulation using a local cache to retrieve information that otherwise would come from a different domain.

This type of testing is enough for validating the behaviour of a single model, but as the domain interactions were not tested nor validated, upon integration of all models, system interactions may be erratic.

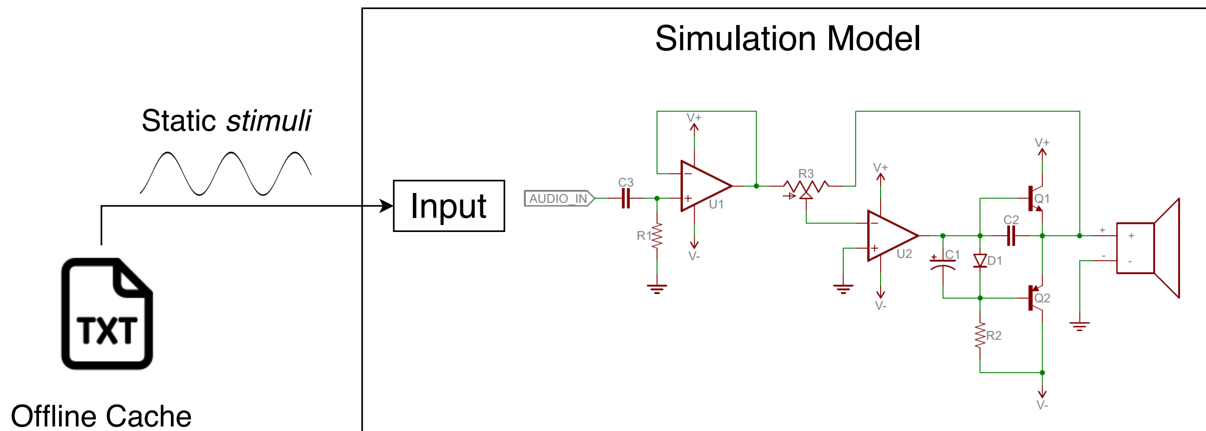


Figure 2.27: Simulation of a single domain using local cache

This is due to the fact that, as the interactions were not really tested, there may be failures regarding domain interactions that can not be observable on isolated domain simulations. For this reason, there is the need to validate all interactions when deploying the system, avoiding additional development costs. On this context, **co-simulation** is a technique to simulate several domains and the interactions between them. It consists on enabling global simulation of a complex system through composition and interfacing of simulators from different domains. This technique has already been applied in many different engineering domains such as Automotive [46][47][48], Robotics [49], HVAC [50][51], and Integrated Circuit and SoC Designs [52].

Several simulators allow extending their functionalities through the usage of dynamic libraries with callback APIs. Such libraries allow simulation tools to exchange information, such as hardware access information and simulation time. An example usage of such libraries is SimCoupler [53], which allows PSIM simulations to interface with Simulink. Extensions can be developed to implement real-time interactions between simulators from different domains, allowing easier validation of interactions across domains. However, if no API's are provided, the simulator can also be extended to allow such interactions, as long as it is open-source.

Although this may seem an excellent way to validate a complex system, the act of simulating in different domains makes interactions difficult due to the different temporal abstraction levels. Since different domains simulators run on different abstraction levels, the time granularity may be different across simulators, which means, at a given *wall-clock* time, the simulators may all have different simulation times, with different execution advancements. This presents a synchronization problem since simulations are independent and interactions should be correctly timed for both simulations.

Before going into detail about simulation synchronization, the concepts behind simulation and types of

simulation should be clearly defined in order to get a better understanding of the next section. In the simulation world, two separate classes of simulation exist: discrete and continuous. In discrete simulations, changes in the state of the system take place at discrete points in time and are instantaneous, whereas in continuous simulations, changes in state occur continuously in time [54]. For this next section, simulation characteristics and assumptions are all borrowed from the discrete simulation domain since the level of abstraction of this type of simulations is enough to represent digital world functionality.

Simulations can be further divided into time- and event-driven regarding the sequence of values of the simulation time. The time of a simulation is an abstraction of real time (or *wall-clock* time), which may not behave as real time does, that is, it may not change at a fixed rate relative to the host computer real time or even be monotonic. In time-driven simulation, the sequence of simulation time values is an increasing arithmetic sequence, which means time increments are constant. In event-driven, the simulation is also monotonic and non-decreasing, but it is not an arithmetic sequence; the sequence values represent times at which the state of the system changes, which such state changes are called "events".

2.7.1 Synchronization

Interactions in different simulation times can result in causality errors, where information sent between simulations is received out of local computation order. An example of a such error can be explained with a simulation of a warehouse, where transport robots that carry boxes are simulated (adapted from [55]). A robot's simulation is triggered by an incoming message which tells him to pick up a box that is then transported to a storage area. During transportation, the robot gets a second message by another robot, being warned to change his route to prevent collisions. If the simulation of the first robot is much faster than the simulation of the second robot, the second message is received, when the first robot has already reached its destination, whereas it should have been received while driving. Such errors can render a simulation useless since the results do not reflect to correct behaviour of the system in question. Hence, all simulation tools must be synchronized.

Figure 2.28 exemplifies, visually, a causality error where two simulations execute simultaneously and exchange messages on specific time events. When simulation 1 sends a message to simulation 2, simulation 1 has already advanced as no messages were received up until the current time. Consequently, the execution of simulation 1 since that time produces erreneously results.

To deal with synchronization between simulations, there are some algorithms to mitigate or even eliminate any causality error. Algorithms generally fall into two major classes of synchronization [56]:

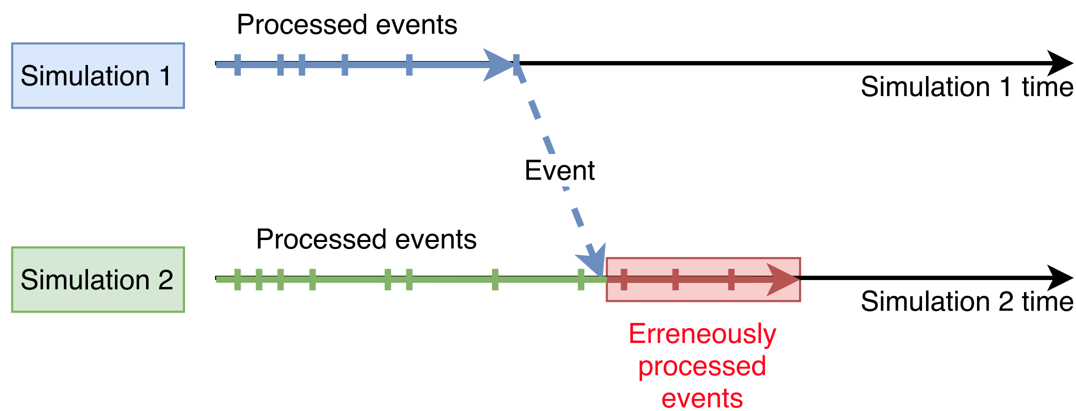


Figure 2.28: Exemplification of a causality error

conservative, which strictly avoid causality errors; and optimistic, which allow errors and recovers from them.

Conservative synchronization is based on the work of Chandy and Misra [57], in which events are processed in sequential chronological order and simulations communicate through time-stamped messages. In execution terms, these mechanisms assure that no message is delayed or received too late, therefore guaranteeing that all messages are attended on time. To do so, the simulations are blocked from further processing until the next message can be safely sent and received on both simulations. The main issue of any conservative simulation is determining how much can a simulation can execute to avoid any causality error. This issue was explored by Ayani [58], who presented a method to identify time-windows in which simulations can safely process events without risking possible causality errors. Altough these type of mechanism eliminate these type of errors, blocking simulations from executing can slow down the speed of the simulation.

On the other hand, optimistic synchronization algorithms allow causality errors to happen and have the ability to detect them. If a causality error is detected, the simulation has to be rolled back, meaning that all preceding simulation results have to be undone until the causality error is resolved. Before the occurrence of a causality error, the simulations are not synchronized and run independently of each other, therefore only being synchronized when causality error occurs. For realizing an optimistic synchronization, all previous states of the simulation have to be saved, as they are needed again in case of a roll back. Additionally a mechanism to call back previous states has to be installed, as in case of a causality error those states can also be wrong. One of the best known optimistic synchronization algorithms is the “Time Warp algorithm” [59]. This algorithm contains two main parts: the local control mechanism that ensures that events are executed and messages received in correct order, and the global control mechanism that

manages memory space, flow control, I/O, and error handling. One big disadvatange of this algorithm is that a single message can cause a chain of rollbacks of every object in the simulation, which comes a big slowdown problem, specially in large simulations.

In order to implement such algorithms and functionalities, the simulation tools, as said before, must have interfaces to allow share of information between them. An example of such tools is QEMU, which is a open-source emulator that allows the developer to modify its internal source code to add new features. As this is the tool used in this dissertation, an overview about it is made later in this chapter.

2.7.2 Functional Mock-up Interface

The Functional Mockup Interface (FMI) is a tool independent standard to support the exchange of dynamic models and co-simulation using a combination of XML files, binaries (shared objects or DLLs) and C code [60]. The first version, FMI 1.0, was published in 2010 and its goal was to support the exchange of simulation models between suppliers and OEMs even if a large variety of different tools are used. As of today, the FMI is currently on version 2.0 [60] and is supported by over 100 simulation tools.

The FMI defines an interface to be implemented by an executable called an FMU (Functional Mock-up Unit) (figure 2.29) [60]. The FMI functions are used (called) by a simulation environment to create one or more instances of the FMU and to simulate them, typically together with other models. An FMU contains an eXtensible Markup Languague (XML) file with description of the interface, and source code or dynamic library which implements the interface. Source code is used if target platform independence is desired, however the latter solution is used whenever the suppliers want to hide the source code to secure the contained know-how [61].

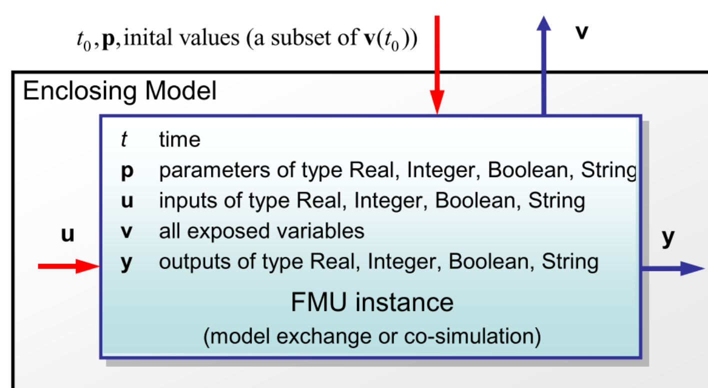
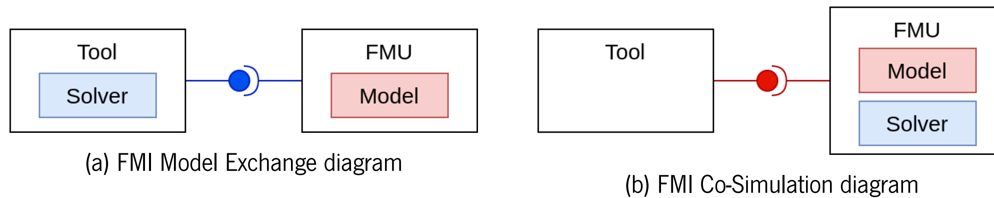


Figure 2.29: FMU instance interface

Depending on the type of simulation, an FMU may either have its own solvers (FMI for Co-Simulation) or require the simulation environment to perform numerical integration (FMI for Model Exchange). The diagram on figure 2.30 presents the two use cases for the different types of simulations supported by the FMI.



(a) FMI Model Exchange diagram

(b) FMI Co-Simulation diagram

Figure 2.30: FMI simulation type support

FMI for Model Exchange

The Model Exchange interface is used to describe models of dynamic systems, i.e., models defined by differential, algebraic and discrete equations and to provide an interface to evaluate these equations as needed in different simulation environments, as well as in embedded control systems, with explicit or implicit integrators and fixed or variable step-size [60]. The resulting FMU from a model description is an input/output block that contains only the implementation of the model while the solver belongs to the simulation tool, as shown on figure 2.31. The solver sets the FMU internal state, asks for the state derivatives, and determines the step size and how to compute the state at the next time step.

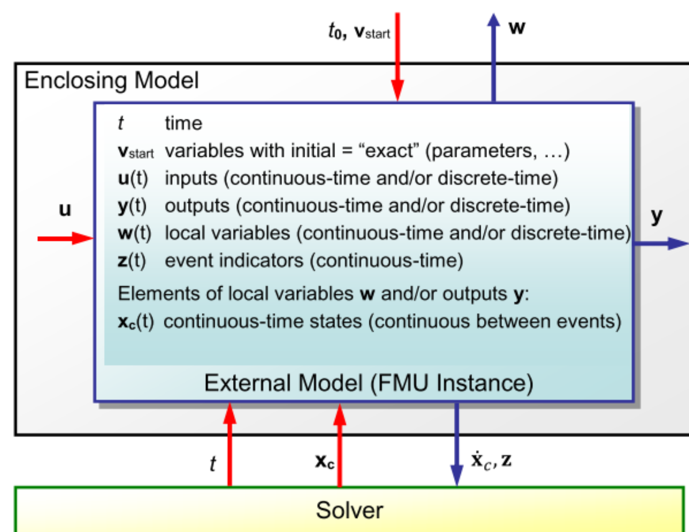


Figure 2.31: FMU for Model Exchange interface

Many FMU instances can be connected to create a larger model, as shown on figure 2.32. They are connected hierarchically through their input and output variables.

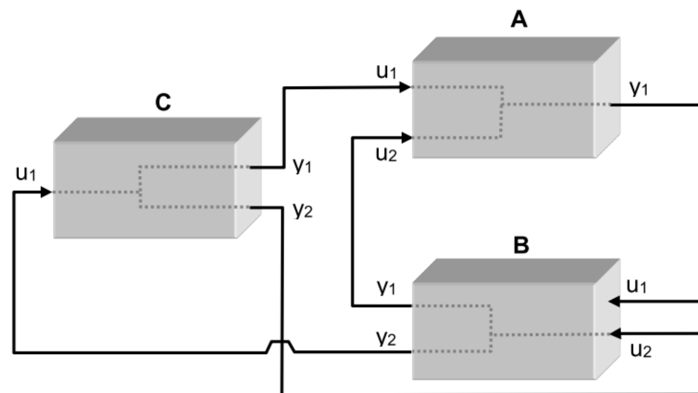


Figure 2.32: Example of three connected FMU instances

FMU for Co-Simulation

FMI for Co-Simulation is designed both for the coupling of simulation tools (simulator coupling, tool coupling), and coupling with subsystem models, which have been exported by their simulators together with its solvers as runnable code. These scenarios are presented on figures 2.33, 2.34 and 2.35, where the FMI interfaces have a coupling job on both simulation tools and distributed co-simulation.

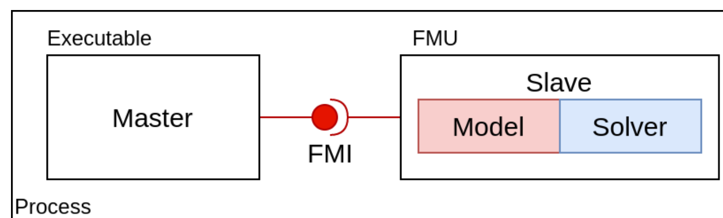


Figure 2.33: Co-simulation with generated code on a single computer

In tool coupling (figure 2.34), FMU implementation wraps the FMU function calls to API calls which are provided by the simulation tool. Additionally to the FMU the simulation tool is needed to run a solver.

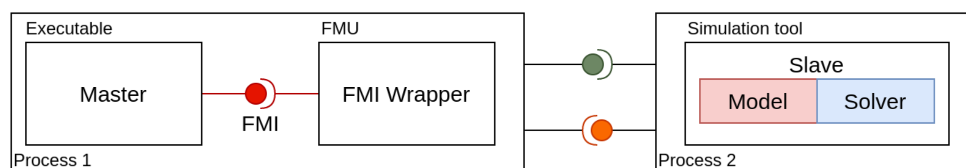


Figure 2.34: Co-simulation with tool coupling on a single computer

In its most general form, a tool coupling based co-simulation is implemented on distributed hardware with subsystems being handled by different computers with different OS (cluster computer, computer farm, computers at different locations). The data exchange and communication between the subsystems is typically done using one of the network communication technologies (for example, MPI, TCP/IP). The definition of this communication layer is not part of the FMI standard. However, distributed co-simulation scenarios can be implemented using FMI as depicted in figure 2.35.

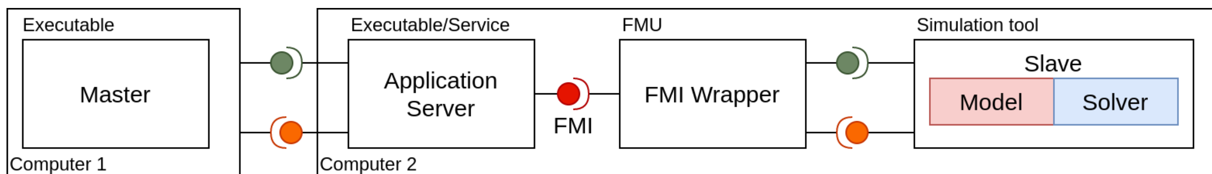


Figure 2.35: Distributed co-simulation infrastructure

Instead of coupling simulation tools directly, it is assumed that all communication is handled via a master. The master plays an essential role in controlling the coupled simulation. Besides distribution of communication data, the master analyses the connection graph, chooses a suitable simulation algorithm and controls the simulation according to that algorithm. The slaves are the simulation tools, which are prepared to simulate their model. The slaves are able to communicate data, execute control commands and return status information [61]. The diagram on figure 2.36 shows how all simulations tools communicate with each other by interfacing with the master.

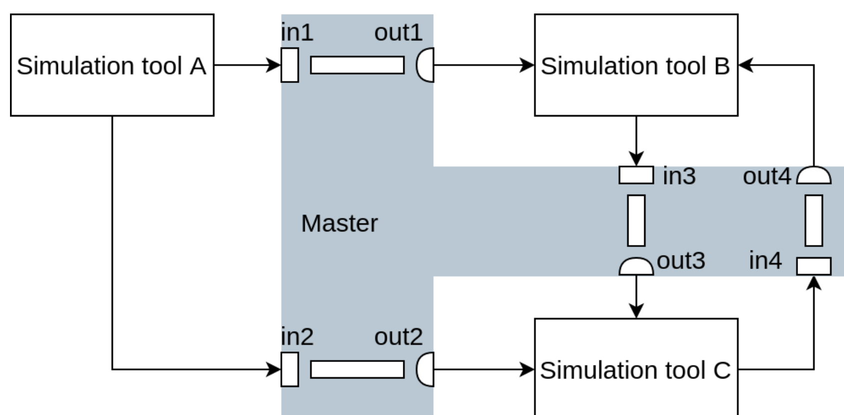


Figure 2.36: Simulation control through master-slave interface

In co-simulation stand alone, an FMU contains not only a model, but also solver code exported by another simulation tool to solve the model during simulation. Figure 2.37 represents a co-simulation slave FMU, which contains both model and solver.

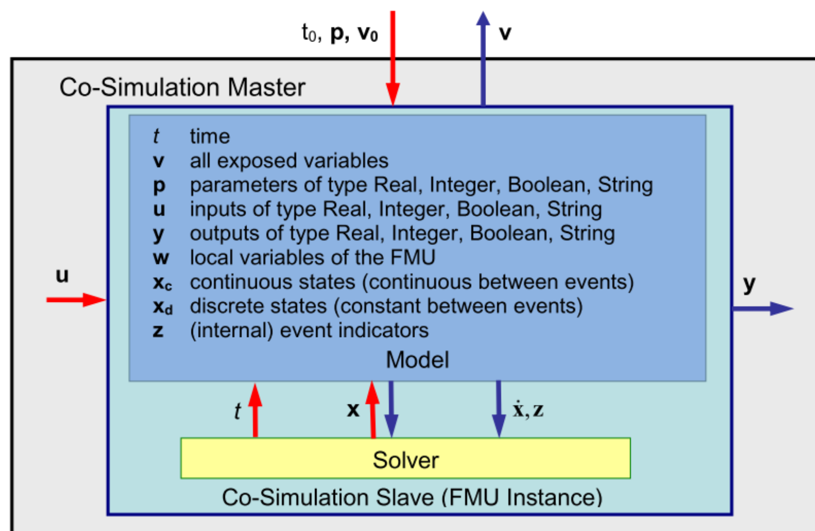


Figure 2.37: FMU for Co-simulation interface

2.7.3 FMI usage in the industry

FMI was used, and still is, in industrial and scientific projects by several companies and research institutions. The gearbox projects for Mercedes-Benz passenger cars used FMI in software-in-the-loop simulations [62]. In mechatronic gearshift simulations for commercial vehicles at Daimler AG, the standard was utilized twice by having powertrain software simulated in SimulationX and the multibody system in Simpack [46]. At IFP Energies Nouvelles, FMI for Model Exchange is used to parallelize the execution of complex internal combustion engine models in the tool xMOD [63]. Their use is mainly intended to validate engine controls with the help of hardware in the loop simulations. In [64], an algorithm was implemented in Python for derivative-free optimization implemented in Python and applied to parameter optimization of FMUs. The optimization algorithm is applied to a Volvo truck engine to identify model parameters based on measurement data from a test cycle. In [65] the FMI based co-simulation master from Fraunhofer is used to develop, implement and test sophisticated algorithms for the co-simulation of FMUs generated by Dymola.

2.8 QEMU

QEMU (which stands for Quick Emulator) is an open-source machine emulator and virtualizer. It uses a portable dynamic translator, which translates target binary code to host binary code. QEMU is very versatile and can run x86, x86-64 and PowerPC systems, and it can emulate x86, x86-64, ARM, SPARC,

PowerPC and MIPS architectures. For most of these, it can be run in two ways: full-system emulation and user-mode emulation. User mode emulation allows QEMU to launch processes compiled for one CPU on another CPU, translating system calls on the fly. The full system emulation capability emulates a full system, including a processor, memories, interrupt controllers and peripherals. Given the scope of the dissertation, QEMU will be used in full system emulation, while user mode emulation will be ignored. This latter feature is very useful in the embedded context, since it enables development, debugging and testing without the physical target hardware.

Being an open-source software, its source code can be changed in order to edit its features or even add new ones. For this reason, one of the main focus of this dissertation is using QEMU as a simulation tool, with efforts to extend its features to support co-simulation using full-system software simulation.

Currently, QEMU is poorly documented, which means that analyzing the source code is required to understand QEMU's internal architecture. Documentation is sparse and almost non-existent, and source code comments are frequently outdated. The main communication channels between QEMU developers are QEMU's development mailing list, which is used to search for insights in emails and conversations between developers, or the QEMU IRC channel which provides a live-chat between developers. Given the low amount of available QEMU literature and information, it is relevant to provide an insight into its internal architecture, with special focus on functional hardware emulation, binary translation and emulation time, in order to understand the work developed in this dissertation.

2.8.1 Binary Translation

QEMU uses dynamic binary translation to execute target code on the host platform. The translation is done by a module called the Tiny Code Generator or TCG for short. This module is the core binary translation engine and works by translating each disassembled guest instruction into a sequence of host instructions. A simplified overview of QEMU instruction translation is presented on figure 2.38. It starts by disassembling the target binary code and generating an intermediate representation of it. Then, the generated intermediate code representation is transformed into host instructions with the help of the Tiny Code Generator (TCG).

Translation is made on-the-fly, during runtime. Instead of translating every single instruction, guest code is split into chunks of instructions called "translation blocks". This block is similar to a basic block of instructions but it always executed as whole, meaning that there are no jumps in the middle of the block. Translation blocks are then cached into a translation cache, that is used to speed up execution of

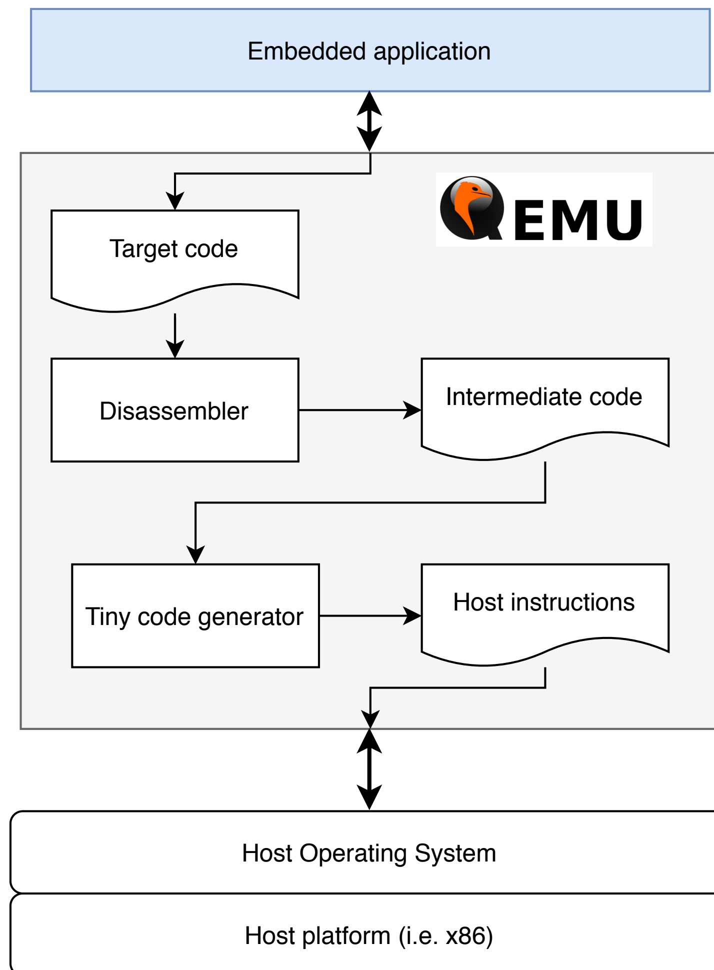


Figure 2.38: QEMU translation overview (adapted from [66])

similar code blocks, and the program is mapped into a lookup table. The length of the block is established upon translation up until the nearest jump or an instruction that changes the CPU state in a way that can not be deduced at translation time, this being the reason the block will always execute as a whole. If the flow forces a conditional jump, the virtual Program Counter (PC) takes an address of an already cached translation block. This mechanism speeds up execution since there is no target code translation overhead. For this reason, cached blocks are indexed using their guest virtual addresses, so they can be found easily using the virtual PC value, and are purged every time the cache fills up [67]. The translation block is executed when it is translated and already on the cache. If that is not the case, a new block must be prepared for execution. The diagram on figure 2.39 presents the dynamic translation process.

One other mechanism that QEMU employs on binary translation is TB chaining. Normally, the execution of every translation block is surrounded by the execution of special code blocks: the prologue, which initializes the processor for generated host code and jumps to the code block; and the epilogue,

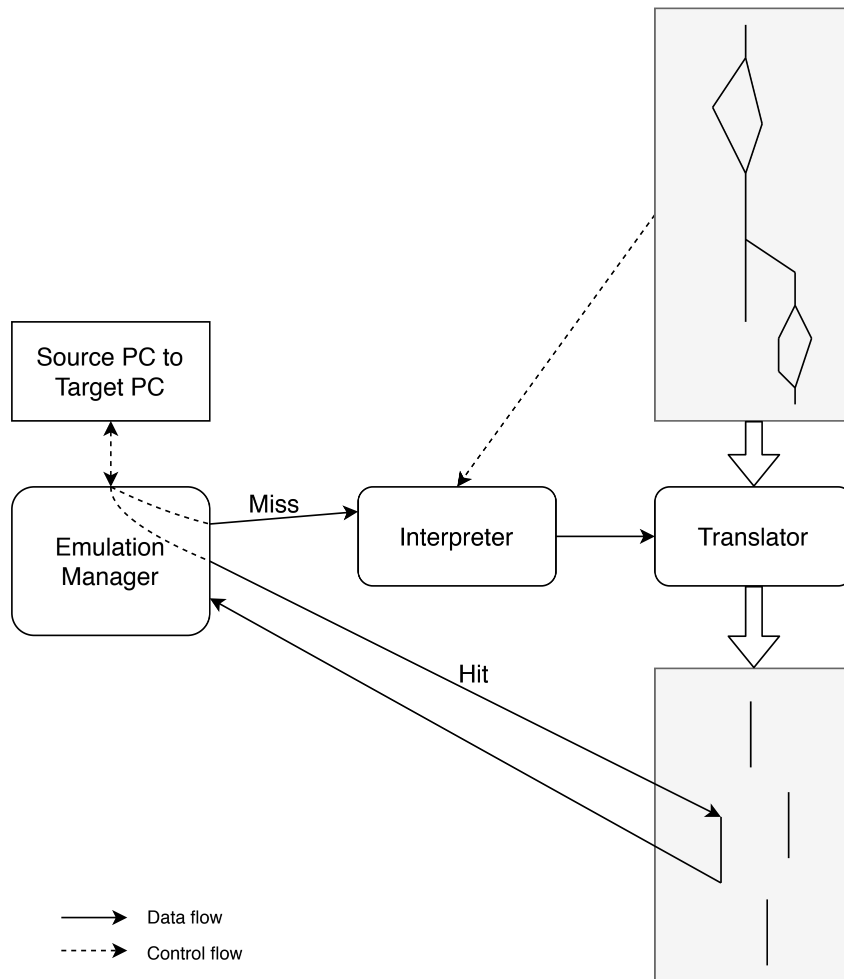


Figure 2.39: Dynamic translation diagram (adapted from [67])

that restores normal state and returns to the main loop. However, returning to the main loop after each block adds signification overhead. To reduce this overhead, the TB's are chained in order to jump directly to an already translated block instead of jumping to the epilogue. The diagram on figure 2.40 presents a simplification of the TB chaining process.

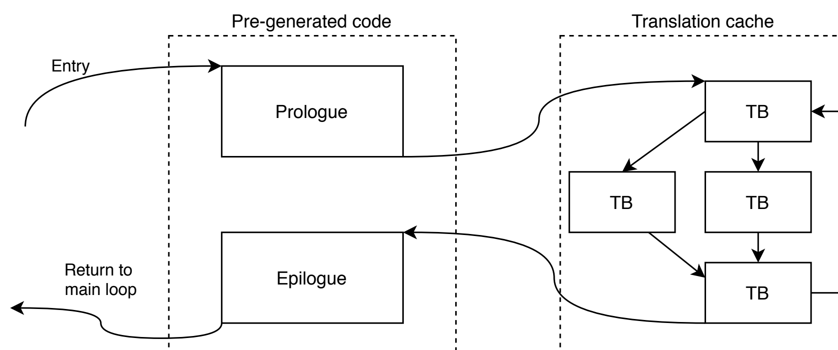


Figure 2.40: Translation Block chaining diagram (adapted from [67])

2.8.2 Deadlines and Emulation Time

QEMU's internal timers, called deadlines, provide a mean of calling a given routine (a callback) after a time interval has elapsed, passing an opaque pointer to the routine. This is particularly important for emulated devices such as timers, that need to be aware of the execution time. QEMU offers three clock sources: Realtime, Host and Virtual clocks. The Realtime clock runs even when the emulation is stopped, with a resolution of 1000Hz. The Host clock runs even when the emulation is stopped, but is sensitive to time changes of the host clock. Lastly, the Virtual clock only runs when the emulation is running and has a high resolution. Most emulated devices use this clock as the clock source in their behaviour.

The emulation time is a particular tricky feature on QEMU since emulation is made as fast as the host platform can run it. This differs from a physical guest platform since, typically, the physical guest has a lower clock than the host platform. This means that the time spent on code execution may not correspond to a real execution time, even when using the Virtual clock. This results on a non-deterministic execution. To mitigate this issue, newer versions of QEMU implement the 'icount' parameter.

The 'icount' parameter affects the increment of the Virtual clock and when not used it does not match real time and its advancement speed depends on how fast host CPU runs guest instructions. When 'icount' is not specified virtual time ticks synchronously with real time. With this parameter, the Virtual clock assumes that one guest instruction counter tick equals 2^N nanoseconds, being N the user specified on the 'icount' parameter, as shown below.

```
1 guest instruction counter tick = 1 emulated nano second << N
```

2.8.3 Translation Block Execution

The target code executes in a fetch-decode-execute fashion, by continuously creating and executing translation blocks. This process occurs on the qemu_tcg_rr_cpu_thread_fn thread for all architectures. The thread is responsible to create translation blocks from target code, if no translation blocks exist in cache, execute the generated translation block and increment the total emulation time and the deadlines elapsed time. As translation blocks execute atomically, the time increment is proportional to the size of the translation block, i.e. the number of instructions executed. The 'icount' parameter takes great importance here, since its usage forces a straight conversion from number of instructions executed to nanoseconds, providing deterministic translation block execution.

The size of the translation blocks is not only dictated by the existence of a jump instruction but also by the next occurring deadline. This means that, before the creation of a translation block, the next deadline to finish is checked, and the translation block size is affected by the remaining time until the deadline finishes. The diagram on figure 2.41 expresses the creation and execution of translation block, when deadlines are running. It is worth noting that, during a deadline, several translation blocks can be executed, as the target code may contain jump instructions. That being said, the translation blocks always run between deadlines and the existence of several deadlines implies execution slowdown since the translation block cache needs to be updated more frequently.

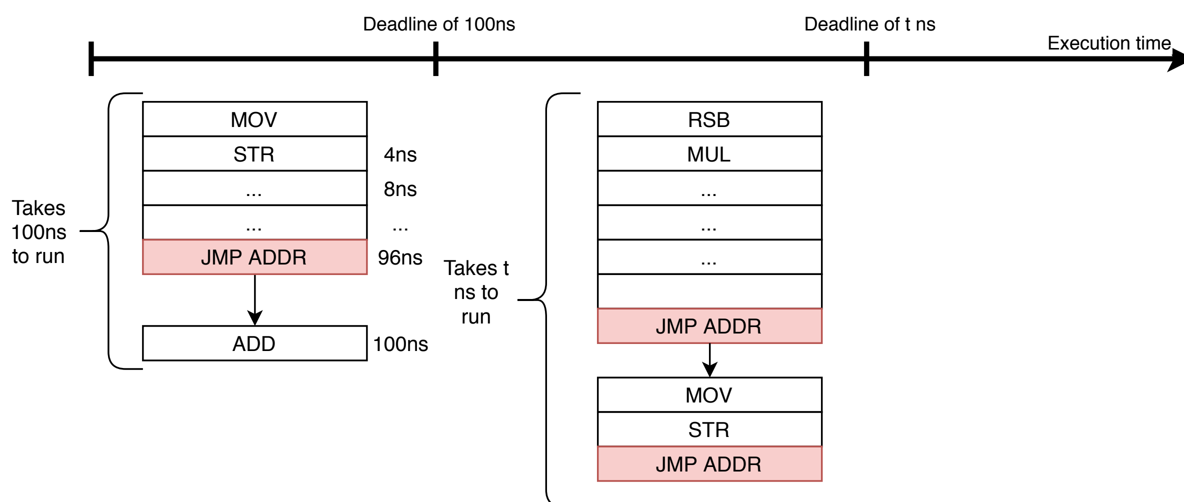


Figure 2.41: Deterministic translation block execution

2.8.4 Device Model

As previously mentioned, QEMU can perform full system emulation with emulated hardware devices. The emulated devices are connected to the QEMU System Bus which is the entity responsible for the interface between all the emulated devices. Figure 2.42 presents a simplification of a full system emulated with multiple emulated devices and an embedded application.

QEMU handles peripheral hardware access by calling a set of functions and routines that emulate hardware behaviour. Whenever a read or write operation is made, the functions that emulate the device read or write transactions are called. The function calls are responsibility of the QEMU system bus, which keeps a list of devices and their respective address. Although the behaviour is emulated, latencies specific to writes and reads on the hardware cannot be emulated.

The creation of an emulated device is made by instantiating an object class and attaching it to the system bus. This way, one device model can be instantiated multiple times into multiple emulated devices, as long as they have different addresses. For example, a development board may have multiple instances of the same ADC device model mapped into different addresses. They share the same behaviour but are independent devices. Most of the device models inherit the classes SysBusDevice, MemoryRegion and the IRQ interface.

The SysBusDevice class represents the device mapped to the system bus. It contains functions to bind the device memory region to a specific address space, assign a IRQ to the device and other helper functions for creating devices. The MemoryRegion class contains functions to register the device to a specific memory region. Each device is responsible for transaction behavior implementation, and as such, each SysBusDevice is associated with a function that implements read operations, and a function that implements write operations. These functions are registered as callbacks in MemoryRegionOps structure present in the MemoryRegion class. The IRQ interface provides API's to assert and handle interrupts. The IRQs are connected to the interrupt controllers or CPUs as input and to the emulated devices as outputs, in order to generate the corresponding hardware interrupt.

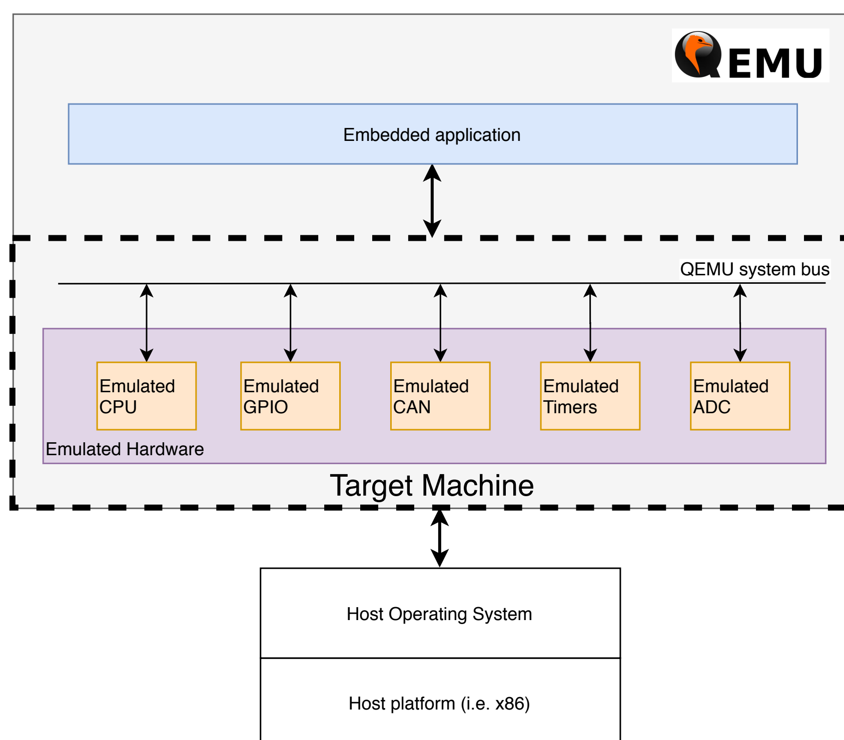


Figure 2.42: Full system emulation overview diagram

2.8.5 Using QEMU on research

QEMU has already been used for several research purposes. One of the topics where it is frequently used is fault injection. The authors of [68], [38] and [69] proposed frameworks for fault injection, that allow to inject faults at the CPU and memory levels. The last two are more complex, since both allow to inject transient and permanent fault at specific times and durations, while the first only implemented permanent faults. Furthermore, an effort on binary mutation was made by [70], which used QEMU's translation to apply mutations to application binary code for testing purposes.

Regarding co-simulation, in [44] QEMU's functionalities were extended to allow for both dynamic instantiation of peripherals and interfacing with external hardware accelerated devices. This allowed for hardware devices that are modelled externally in other simulation tools to still be able to interact with QEMU instances. Similarly, the authors of [71] developed an interface between hardware-accelerated systems and QEMU, with the purpose of validating GPU designs.

Alongside the mentioned research, there are many others that used QEMU to develop and validate their applications. The ability to make changes to its source code makes it a very versatile tool when it comes to complex systems. This versatility is particularly useful on co-simulation, since bridges between simulation tools can be easily implemented allowing for information exchange between tools. This is the main reason QEMU was chosen as the foundation for the work developed on this dissertation.

2.9 Summary

The goal of this chapter was to describe the methodologies and theoretical concepts that aided the development of this thesis. It started by approaching embedded systems and the development cycle adopted. Following this topic, an overview about reliability oriented systems was given, featuring reliability metrics, the Monte Carlo method for reliability estimation and the Fault Injection technique to support such estimations. Futhermore, simulation under embedded development was tackled, with special attention to co-simulation and the synchronization issues that arise from it. Finally, QEMU capabilities and internal mechanisms were highlighted, mainly regarding its binary execution method, device model topology, deadline mechanisms and translation block execution.

Chapter 3

Simulation Extensions for Reliability Development

Simulation is a very important part of resilient embedded systems development, as reliability estimations by means of simulation provide a good overview of reliability related metrics early in the design phase. By using these metrics for early assessment, designs can be reiterated, and fault-tolerant mechanisms may be validated, refined or even expanded upon with new approaches if the design is not robust enough for the desired operation conditions.

Typical embedded full system emulators that allow software simulation are very useful to validate entire software stacks before deploying them on a physical target. However, in reliable embedded development, most redundant processing architectures require some for of interaction and synchronization in order to manage and coordinate redundant modules. Validation of software layers that manage redundancy is often done on physical prototypes, as they are not easy to be simulated even on full development board simulators, given that co-simulation scenarios with multiple processing boards are not often contemplated in these tools.

To improve software development cycles in redundant architecture scenarios, QEMU was extended to integrate functionalities that allow validation of these software stacks and evaluation of fault tolerant software, with development of a practical case study scenario also being done to stimulate simulation environment solutions. This chapter describes the simulation extensions that were developed with the main goals of multi-modular processing system software validation and reliability estimation with fault injection.

With all this in mind, three extensions were developed. The Synchronization extension aiming to mitigate causality errors during co-simulation. The Shared Bus extension that allows for redundant modules to communicate with each other. The Fault Injection extension which enables reliability estimation

capabilities, by providing mechanisms to inject faulty stimuli to system components.

A simple example of redundant architectures is presented on figure 3.1. The redundant subsystems use the same inputs to compute a value which contributes to the system output. The computed values should be equal across subsystems if all of them do the same operations and receive the same inputs. This happens when the architecture presents subsystems that are homogeneous. Such behaviour differs from heterogenous architectures which can have different inputs and outputs between subsystems. Alongside data output and computation, redundant subsystems may also be connected between them for N connections depending on the number of redundant subsystems present.

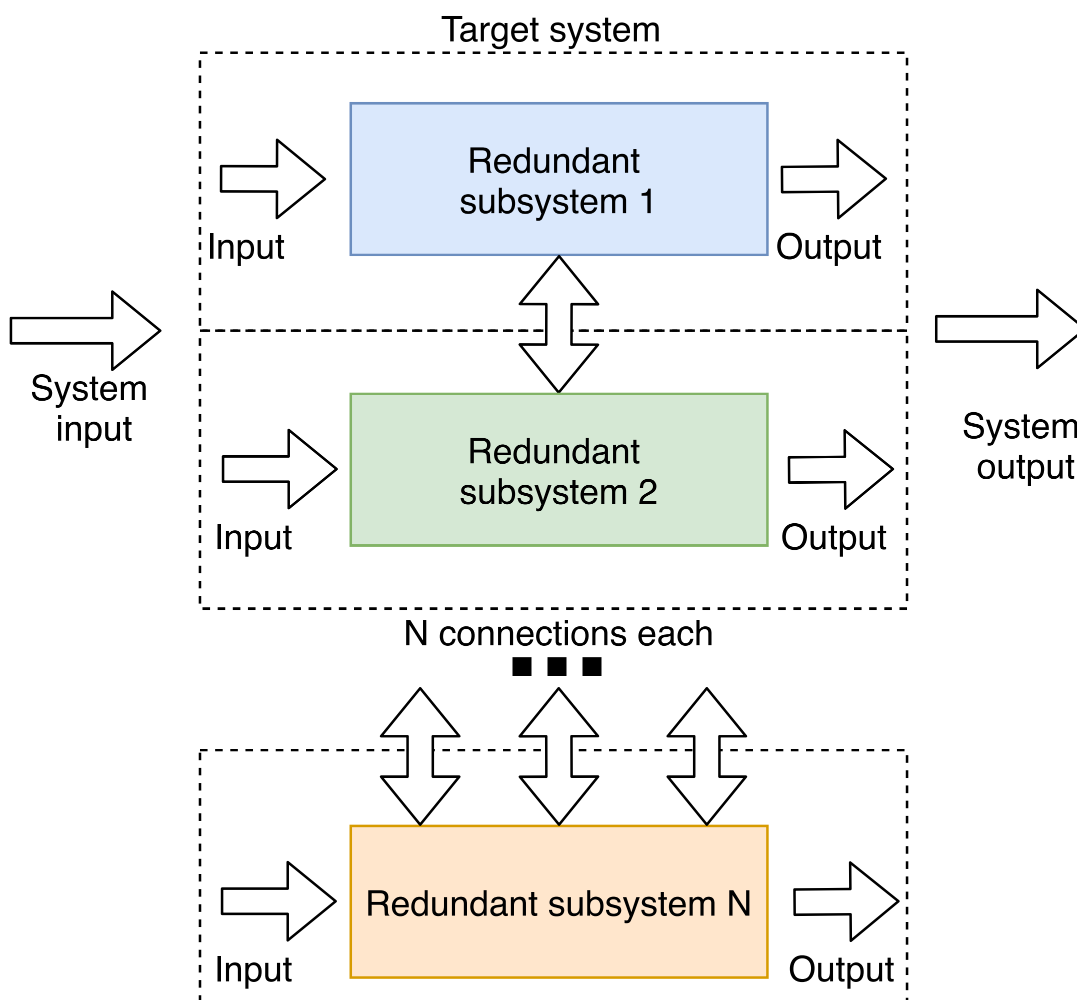


Figure 3.1: Redundant architecture diagram

In this scenario, a target system can have multiple redundant subsystems, each contributing for the output. The redundant modules have independent hardware and computations are made within each subsystem processor. Each redundant module is conceptualized as a QEMU instance (or simulation),

running all the software stack and emulating all the hardware that composes the subsystem. The diagram on figure 3.2 presents the conceptualization of the redundant modules as QEMU instances.

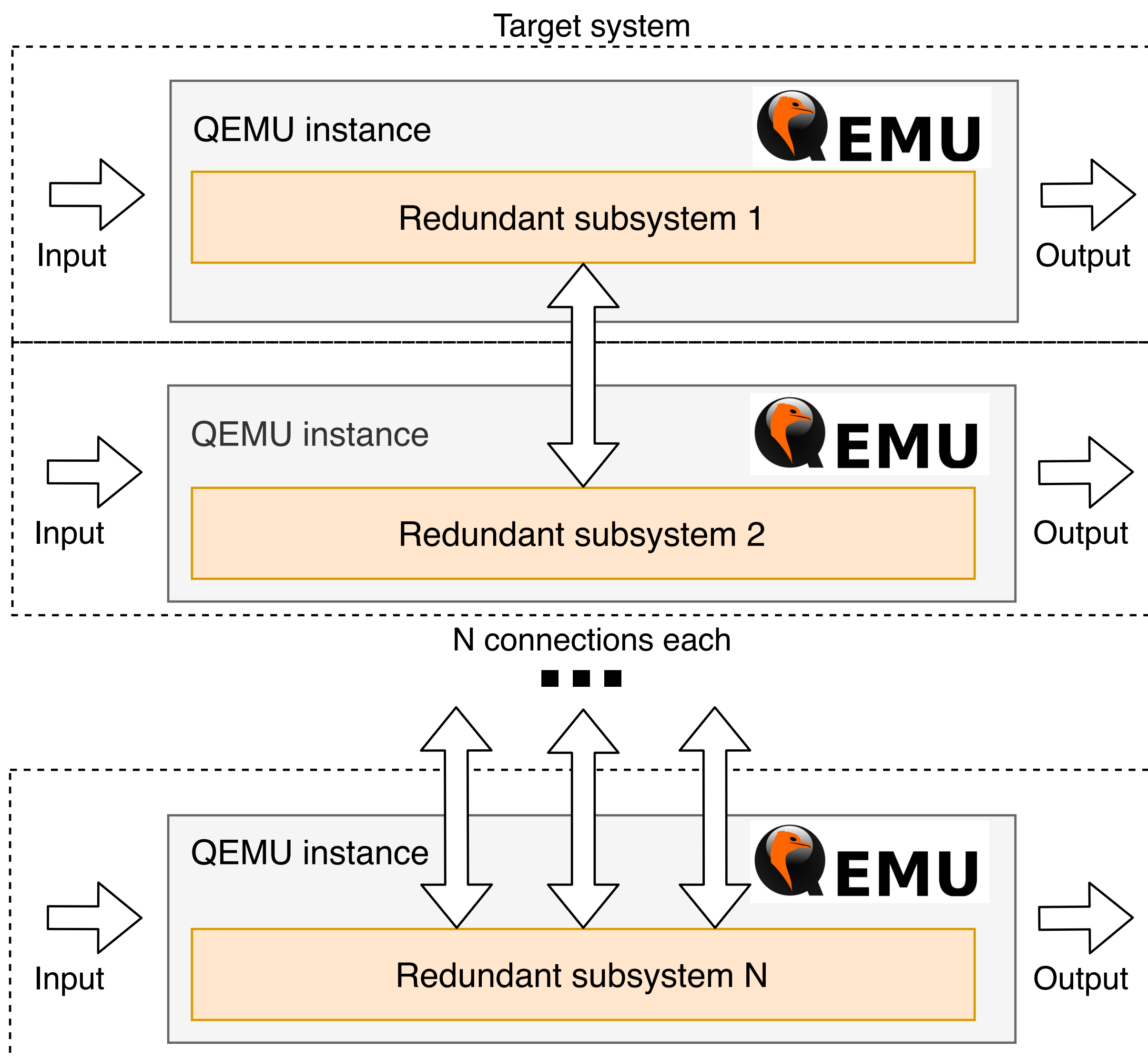


Figure 3.2: Redundant architecture conceptualized on QEMU diagram

As the instances are independent from each other, there must be guarantee that causality errors do not happen. That being said, a synchronization mechanism was implemented to mitigate any simulation synchronization issues between redundant subsystem simulations.

3.1 Synchronization between redundant subsystems

The synchronization method between simulations guarantees that no causality errors happen. This type of errors are avoided by ensuring that up until any interaction, the total simulation time between

simulation instances is equal. The diagram on figure 3.3 shows a conservative synchronization method based on a time budget concept. Both instances are allowed to run for a specific time budget, and, at the end of each budget, running instances wait for the remaining instances to finish operations, synchronizing their global simulation time. The value of the time budget is application-specific since different applications have different deadlines up until causality errors can occur.

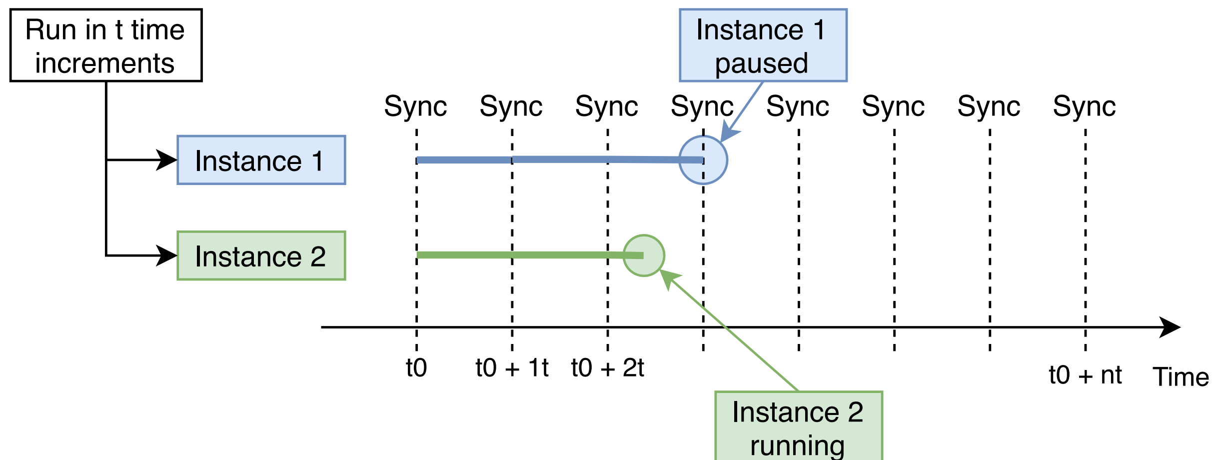


Figure 3.3: Synchronization timely sequence diagram

For this purpose, a process was created, which handles synchronization between simulations using the time budget concept. The diagram on figure 3.4 shows an example of a co-simulation environment with redundant subsystems running on QEMU instances being controlled by the synchronization process. This simulation environment is scalable for multiple redundant module simulations or even other external domain integrations, given that the synchronization process is not limited to QEMU connections, allowing interconnection with any simulation tool in scenarios where other simulation domains are desirable. All simulation tools and QEMU instances communicate with the synchronization process and obey its requests to either stop or resume their simulations. The user is responsible to choose the time budget for synchronization purposes in each of the simulation tools, independently. For this purpose, each simulation tools must have a mechanism to allow the simulation to run for a specified time.

3.1.1 Synchronization Process

The simulation instances communicate with the synchronization process through TCP/IP sockets. The process creates a master server socket and listens to incoming connections and synchronization requests. When a QEMU instance connects to the server socket, the connection is saved on a list of sockets in order to keep track of the number of simulations that need synchronization. When an "out of budget" message

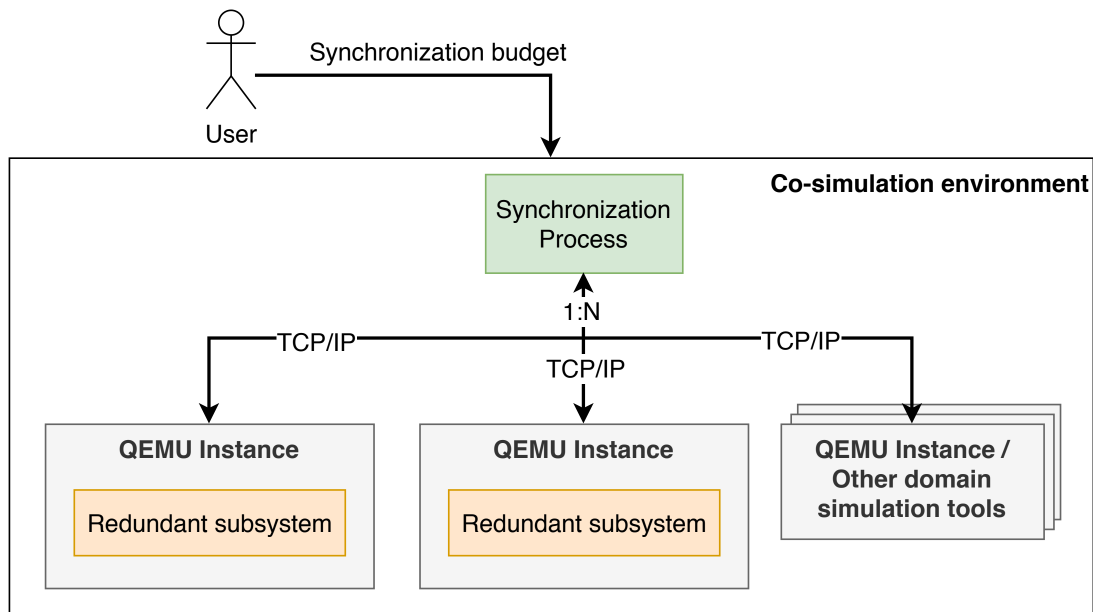


Figure 3.4: Co-simulation environment example

is received, meaning that the simulation reached the end of its execution budget, the process checks if all other connected simulations have already reached the same simulation time. If that is the case, a resume message is sent to every simulation, allowing them to resume execution. If any simulation did not reach the same execution time, all other simulations are blocked from further execution until all reach the same time. An interaction between the process and a simulation tool is shown on the sequence diagram on figure 3.5, where one QEMU instance finishes budget and blocks simulation execution until a resume message is received.

During runtime, the process always listens for new simulation connections, making it scalable for multiple simulations to synchronize. The flowchart on figure 3.6 presents the fully working principle of the synchronization process.

3.1.2 QEMU's Internal Synchronization

QEMU instances establish connection to the synchronization process during the initialization sequence, before machine emulation starts. During this initialization, QEMU reads the arguments provided by the '-sync' command line flag, which was added to the supported command line arguments, to both provide the correct synchronization process server port and the time budget, in nanoseconds, to be used for synchronization. An usage example of this argument is presented on the following command line snippet:

```
$ qemu-system-arm [flags] -sync <time budget>,<server port>
```

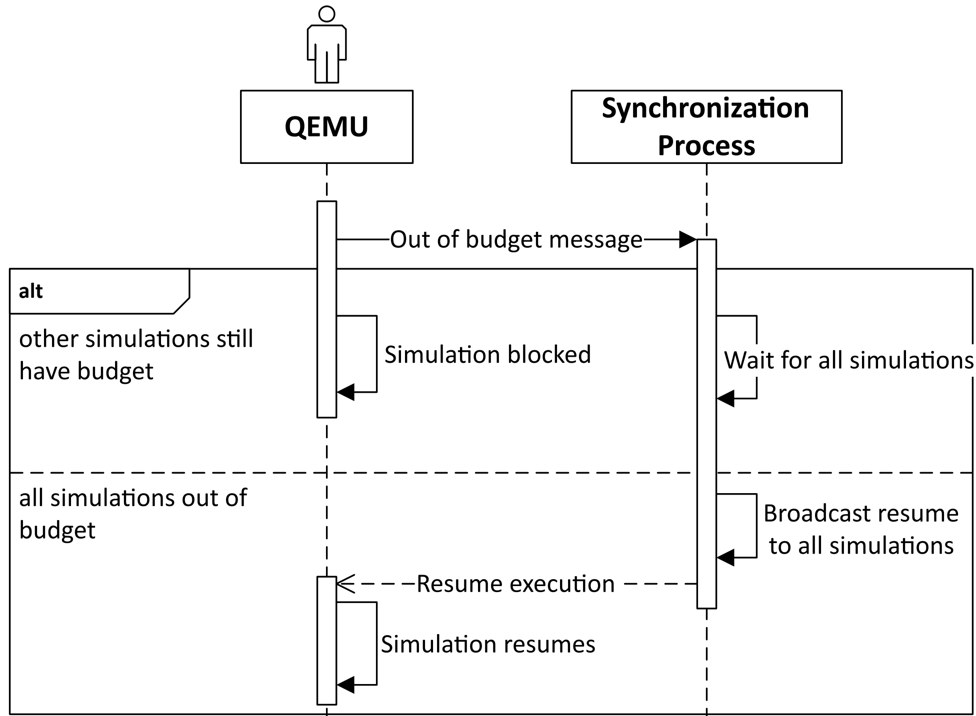


Figure 3.5: Synchronization of QEMU instances sequence diagram

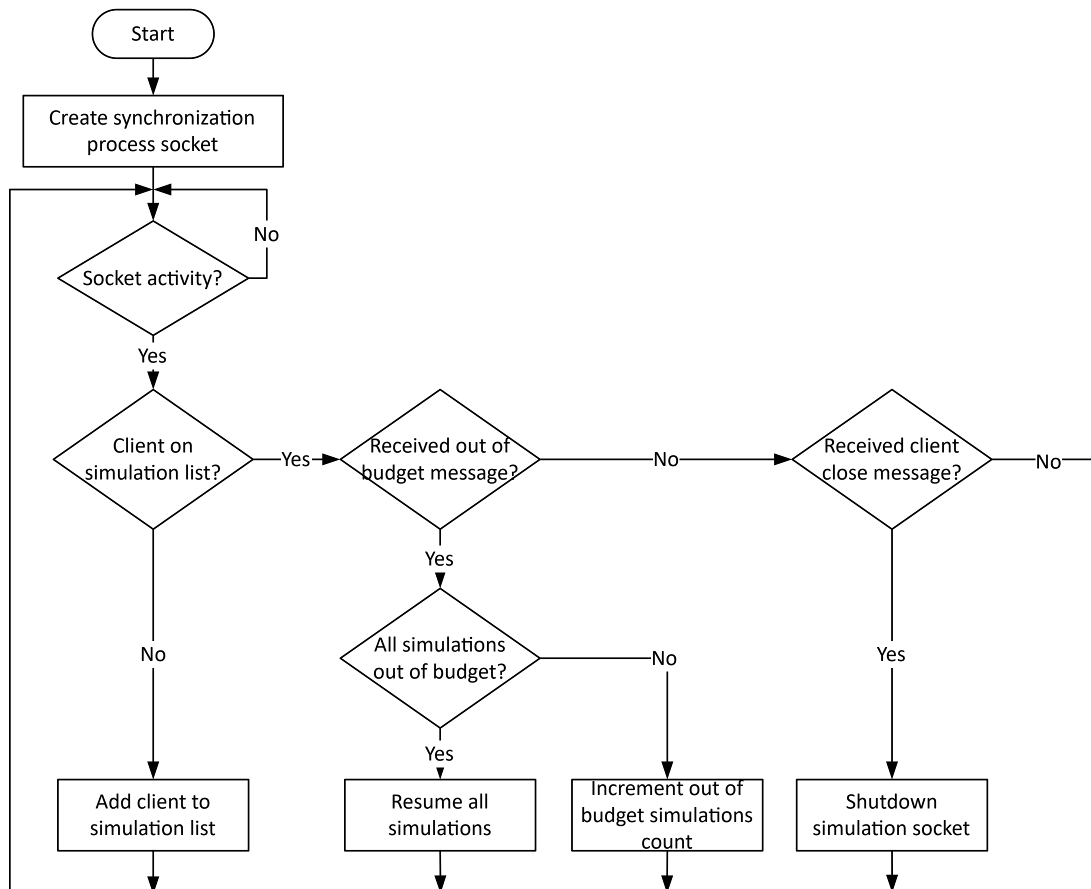


Figure 3.6: Synchronization process flowchart

During QEMU emulation, simulation time advancements occur as translation blocks are executed. This occurs on the translation thread, which fetches (or creates if not already in cache) a translation block, executes it and increments the simulation time according to instructions executed. A simple flowchart of execution is presented on figure 3.7. With that in mind, synchronization is made by monitoring the translation block execution and blocking further execution upon reaching a time budget.

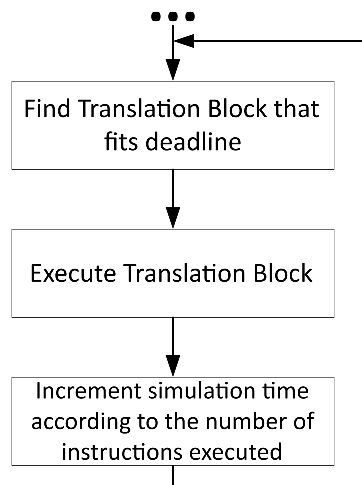


Figure 3.7: Translation block execution thread simplified flowchart

For this purpose, the thread algorithm was modified to take into consideration the time budget. Everytime a new translation block is executed, the current simulation time is compared with the remaining execution time budget. Both these values, simulation time and remaining budget, are updated when a translation block finishes execution. Before getting a translation block, there is a check if the is any more execution budget left. If so, translation blocks keep executing. If the budget is expired, execution reaches a synchronization point and waits for feedback after sending an "Out of budget message" to the synchronization process. Upon receiving feedback, the budget is reset and a new deadline is created according to that same budget. The new deadline is set to the current simulation time plus the time budget, preventing the generation of translation blocks that would otherwise atomically execute for a longer time than the time budget. The flowchart on figure 3.8 shows the budgeted execution of translation blocks that was previously described. The yellow blocks represent the added steps to the translation block execution thread.

Time budget granularity depends on the time per instruction, which is directly related to the 'icount' parameter chosen for the simulation. The lowest number possible for the time budget is the time to execute one instruction, meaning that, in this case, synchronization would occur at instruction level.

Upon having synchronization between instances, communication with other tools or between QEMU instances is possible without risking causality errors.

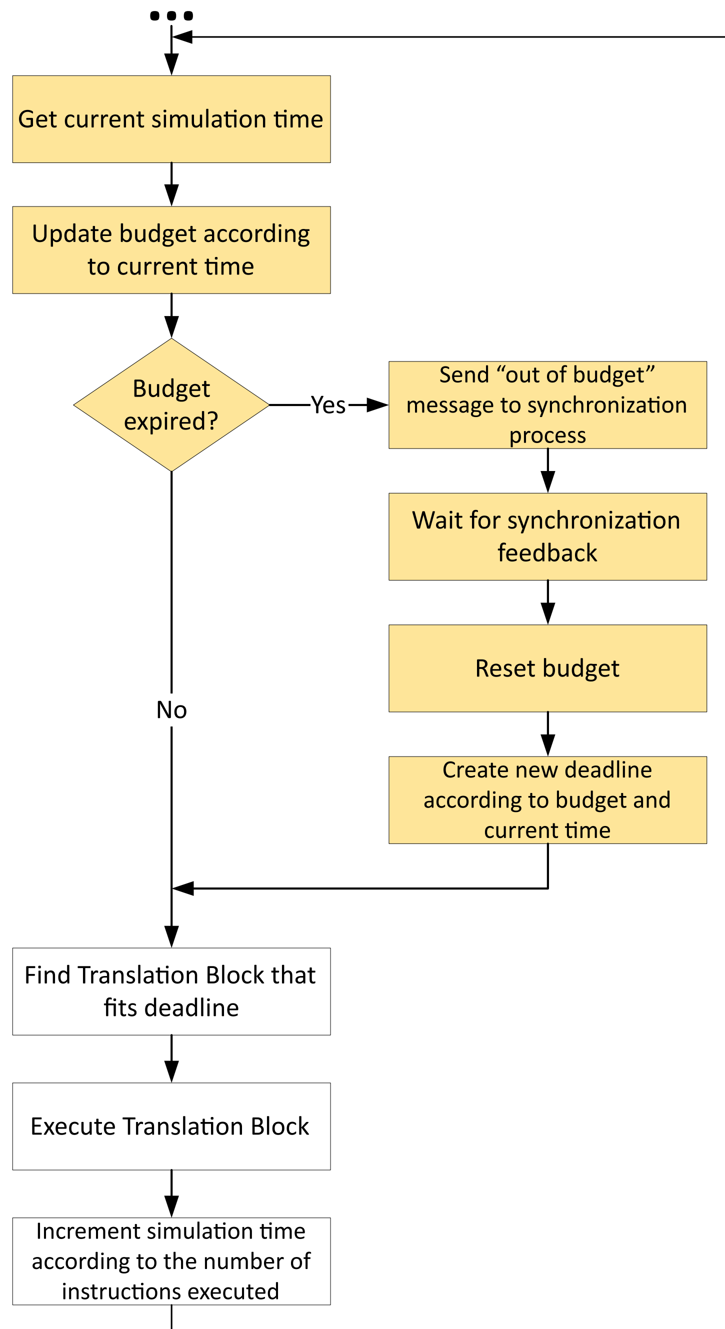


Figure 3.8: Budgeted translation block execution flowchart

3.2 Shared Bus Extension

Under a redundant architecture, there are often communication channels between subsystems. These communication channels are typically associated with communication peripherals such as LPUART or SPI modules. The Shared Bus extension allows for interactions between communication peripherals on different redundant subsystems.

Alongside the usage for redundant architectures, the extension provides a way for other simulation tools to interact with each other. This allows for different simulation approaches such as Hardware-in-the-Loop to be easily integrated in the development cycle. Figure 3.9 presents a further detailed co-simulation environment with the addition of the a Shared Bus extension, allowing communication between simulation tools.

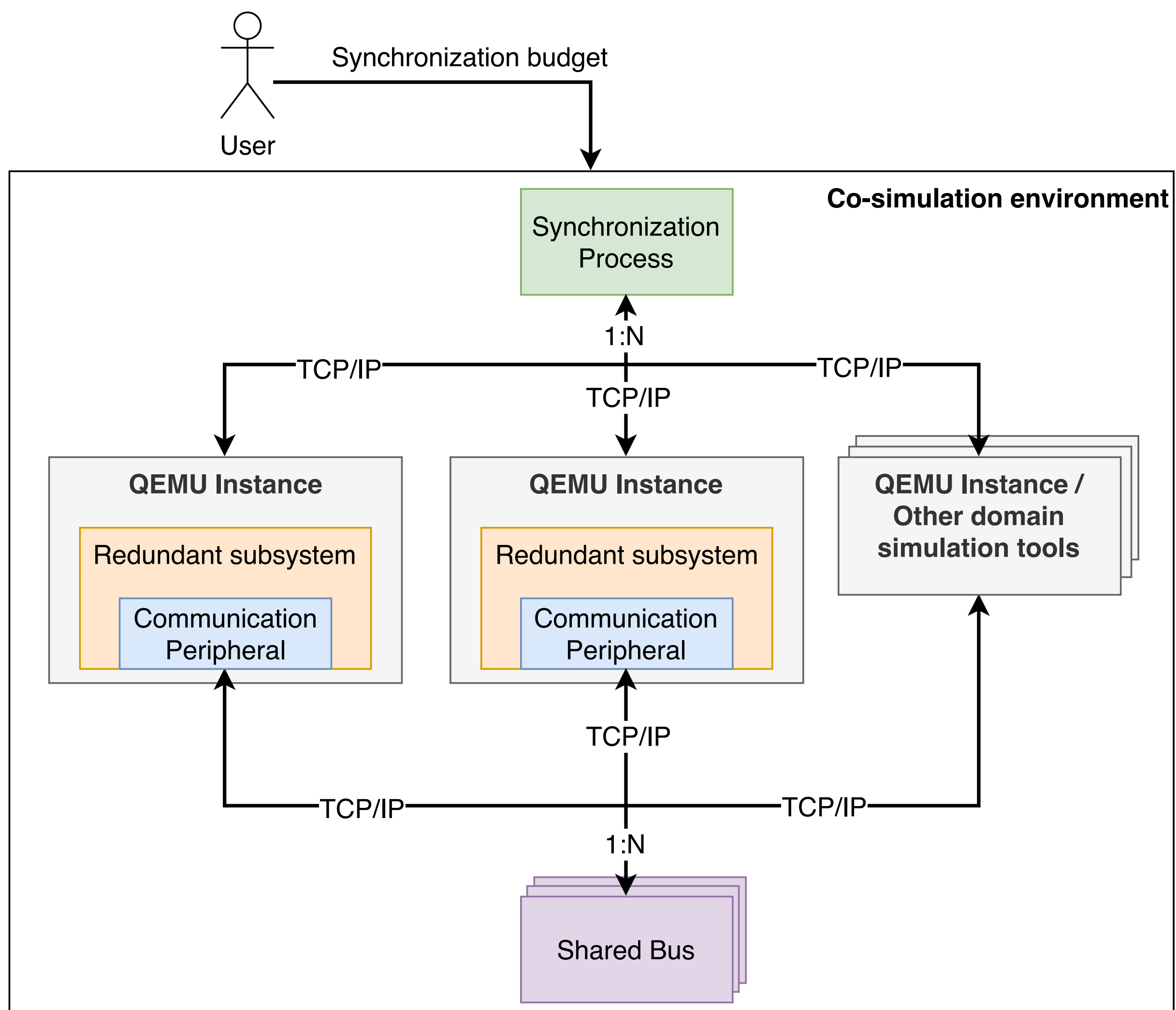


Figure 3.9: Example of a co-simulation environment with the Shared Bus extension

3.2.1 Extension Overview

The extension aims to emulate a data bus and its transactions. It covers the typical R/W operations done by common protocols such as UART, and, at the same time allows for multiple peripherals to connect to it, attending to the bus characteristics of more complex protocols such as CAN or SPI. As such, the developed extension is composed by two parts: (1) An independent process (called Shared Bus) that manages all communication connections to an emulated bus; (2) An interface allowing tools to interact with the emulated bus for read and write operations.

The diagram on figure 3.10 shows the conceptualization of the Shared Bus extension. For exemplification purposes, a CAN bus interfacing with two QEMU instances is used. Along with this interface, the peripherals that facilitate communication are also represented (which allow performing read and write operations on the data bus, much like in every communication protocol). The explanation of the extension will be done using this QEMU communication peripheral as the entity interfacing with the Shared Bus, even though its working principles translate to any other simulation tool.

One thing to notice is that, unlike write operations which are always synchronous, the read operations can be synchronous or asynchronous. What this means is that, on the first case, the peripheral makes a request and blocks execution until that request is finished, while on the second case, the request is made and the data resulting from the request is buffered, preventing execution block. In order to cover the behaviour of many bus kinds, the option of having synchronous writes and both asynchronous and synchronous reads was considered on the implementation, allowing easy adaptation to different kinds of protocols. With that in mind, a pair of sockets in a client-server configuration is used for communication between simulation tools and the emulated bus. On the QEMU side, the peripheral contains two sockets: (1) A client socket which is used for synchronous read/write operations; (2) A server socket which is used for asynchronous reads from the peripheral. More details about this configuration are given later on this section.

On the Shared Bus process side, there are also a pair of sockets that connect to each instance. This process is responsible to receive and save the data, and broadcast it to other simulation tools. It always listens for more connections to the bus, making it scalable for multiple simulation tools to share data between them. For a general overview, the flowchart on figure B.1 presents the working principle of the Shared Bus process. This kind of implementation fully abstracts the communication protocols timings and working principles. It only concerns the behavior of the communication by saving the data and relaying it to other communication peripherals. For this reason, every peripheral implementation needs to consider

invalid data writes or reads to the bus, and check of the integrity of the data received.

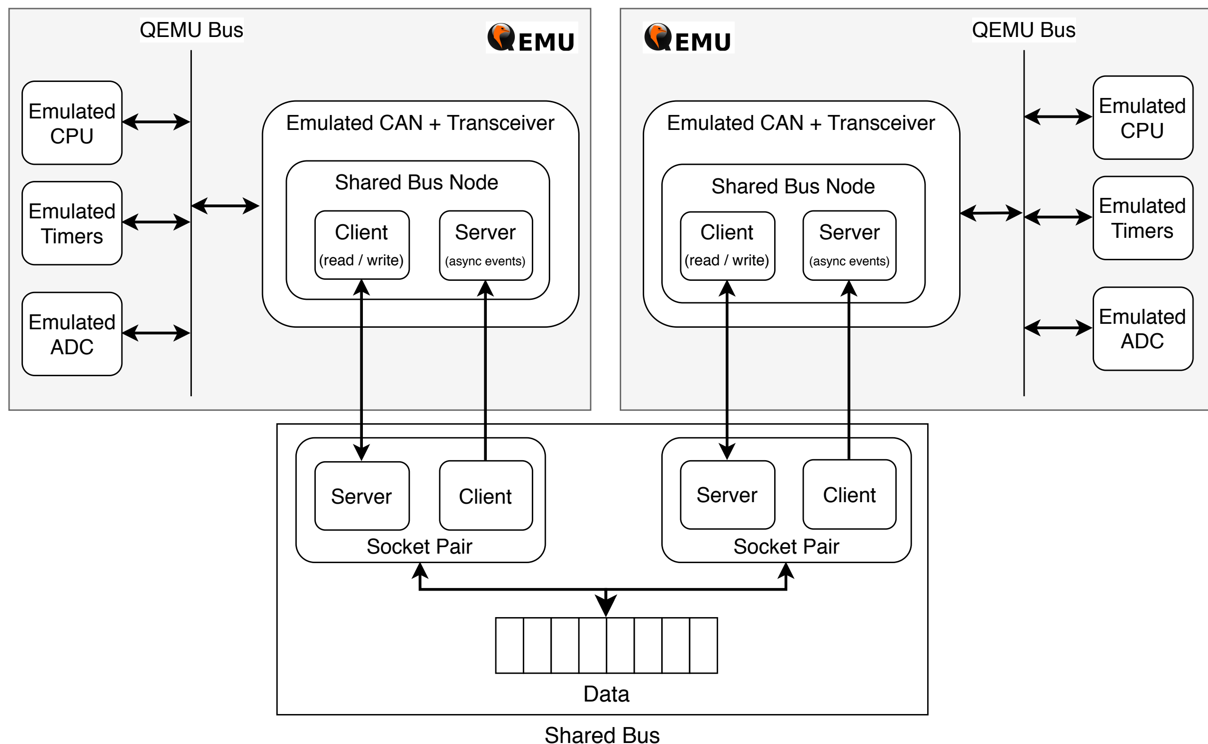


Figure 3.10: Shared Bus extension conceptualization diagram

The communication between QEMU instances and the Shared Bus is made through the socket pairs as seen on figure 3.11. The client-server socket interaction between QEMU and Shared Bus allows for synchronous read and write operations. This client-server communication is bidirectional, meaning that commands and data can be shared between both entities. On the other hand, the server-client interaction between QEMU and Shared Bus is a one way communication for asynchronous reads by the peripheral. On this interaction, the Shared Bus only sends data to the peripheral, leaving the peripheral with the responsibility of all processing and data validation according to its protocol. The commands used for the communication protocol are presented on appendix table B.1.

3.2.2 Extension Operations

In order to interface with the Shared Bus process, QEMU needs to connect as a client to the master server socket of the Shared Bus process and create its own server socket to allow incoming asynchronous data from the bus. After connecting to the Shared Bus, a connect command is sent to signal the process to wait for a server port to connect. QEMU follows by creating a server socket for the asynchronous events and sends its resulting port to the Shared Bus process. The latter connects to it by dynamically

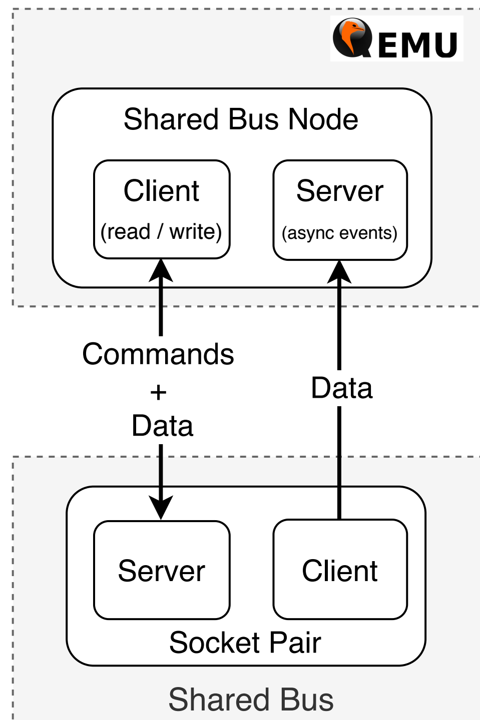


Figure 3.11: Communication between Shared Bus socket pairs

creating a client connection. At this point, both QEMU and the Shared Bus, have a pair of sockets that allow read/write operations and asynchronous data reception/transmission. This initialization process is shown on figure 3.10 as a sequence diagram.

Upon connection of all socket pairs, the peripheral configured to interact with the emulated bus can realize read and write operations on it. The write operation sends data to the Shared Bus which stores it and broadcasts it to the other peripherals connected to the bus. This means that everytime a write on the bus happens, every peripheral receives an asynchronous event and its the peripheral's responsibility to or not to attend to that event. This operation only goes through if the Shared Bus acknowledges the write command, else no data is written on the bus and no broadcast is made. Figure 3.13 presents the sequence diagram relative to this operation.

The read operation is a simple request and response operation for synchronous operations. Asynchronous reads occurs when data is broadcast by the Shared Bus. Figure 3.14 shows the read operation sequence diagram.

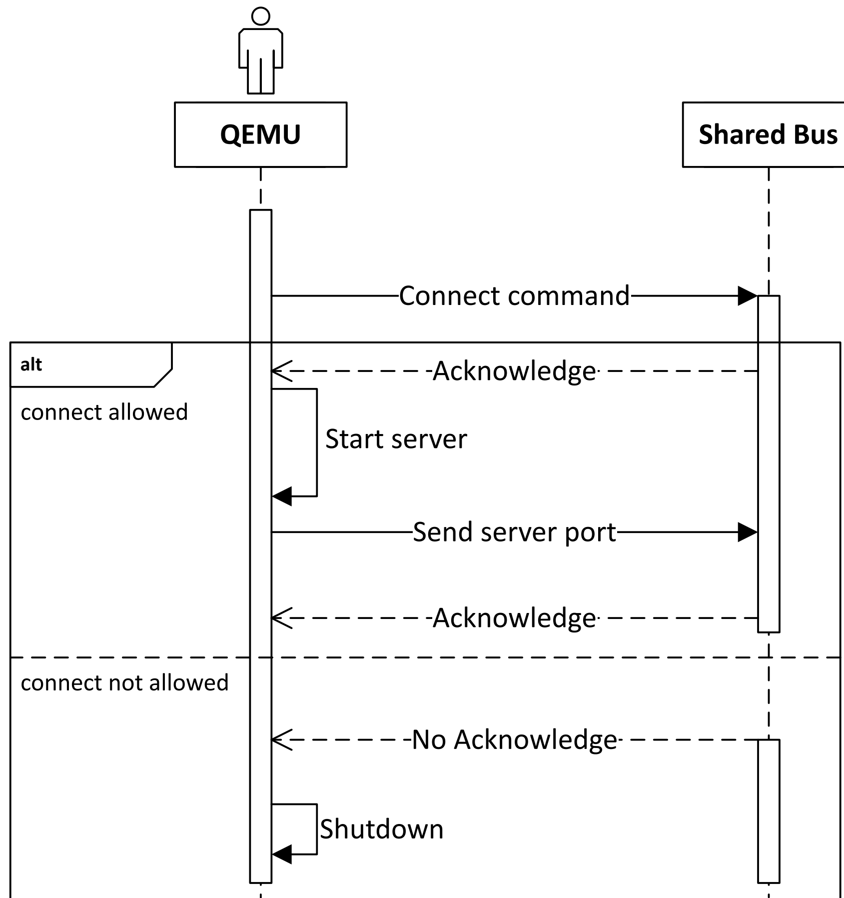


Figure 3.12: QEMU connection to the Shared Bus

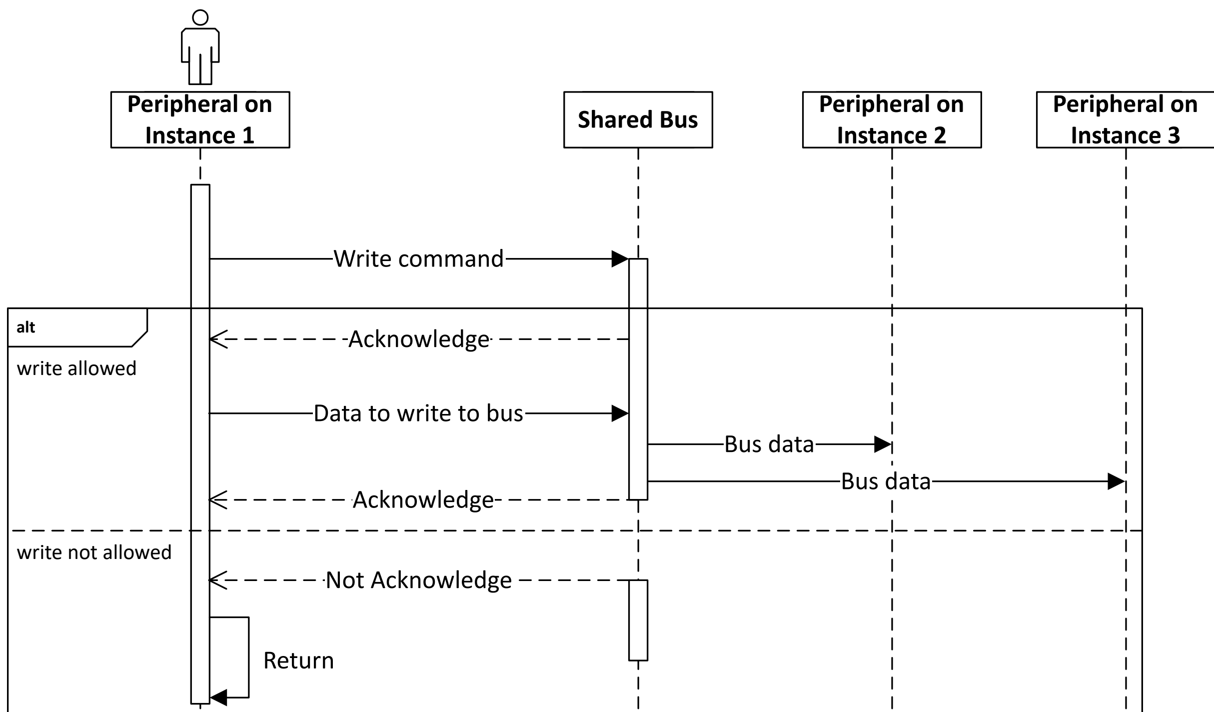


Figure 3.13: Write operation on the Shared Bus

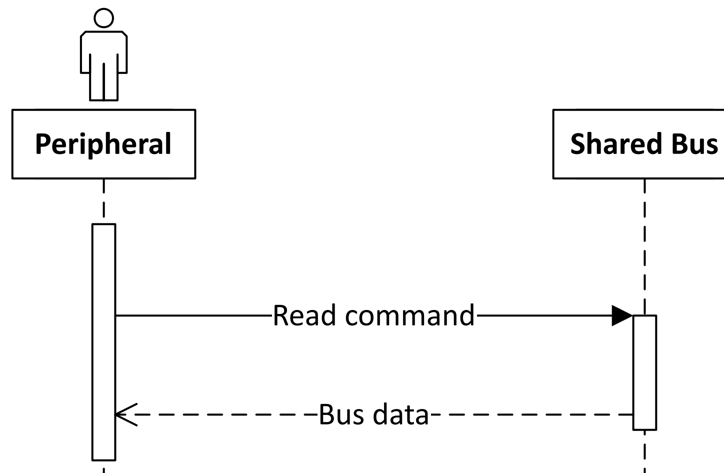


Figure 3.14: Synchronous read operation on the Shared Bus

3.2.3 Extension Interface

The Shared Bus can be used by multiple QEMU communication peripherals of different types. To improve portability and ease of development, the interface used by the peripherals should be abstracted from the peripheral behaviour itself in order to avoid designing a specific interface for each peripheral. For that purpose, an API that allows any QEMU peripheral to interface with the Shared Bus was developed and its specification is presented on figure 3.15. The interface is composed by the following data structures:

- ClientSocket: socket descriptor for synchronous read and write operations;
- ServerSocket: server socket descriptor for asynchronous operations;
- SharedData: temporary storage for the most recent data from the bus;
- DataMutex: mutex for operations on peripheral memory;
- ServerThread: pointer to the asynchronous events thread, which will be discussed in more detail next;
- write_on_shared: API for write operation on the Shared Bus;
- read_from_shared: API for read operation on the Shared Bus.

Alongside extending the interface, a mechanism that allows listening to asynchronous reads needs to be implemented. This mechanism must not block execution since asynchronous events can happen anywhere along execution timeline. To do so, a thread needs to run along main execution, monitoring the

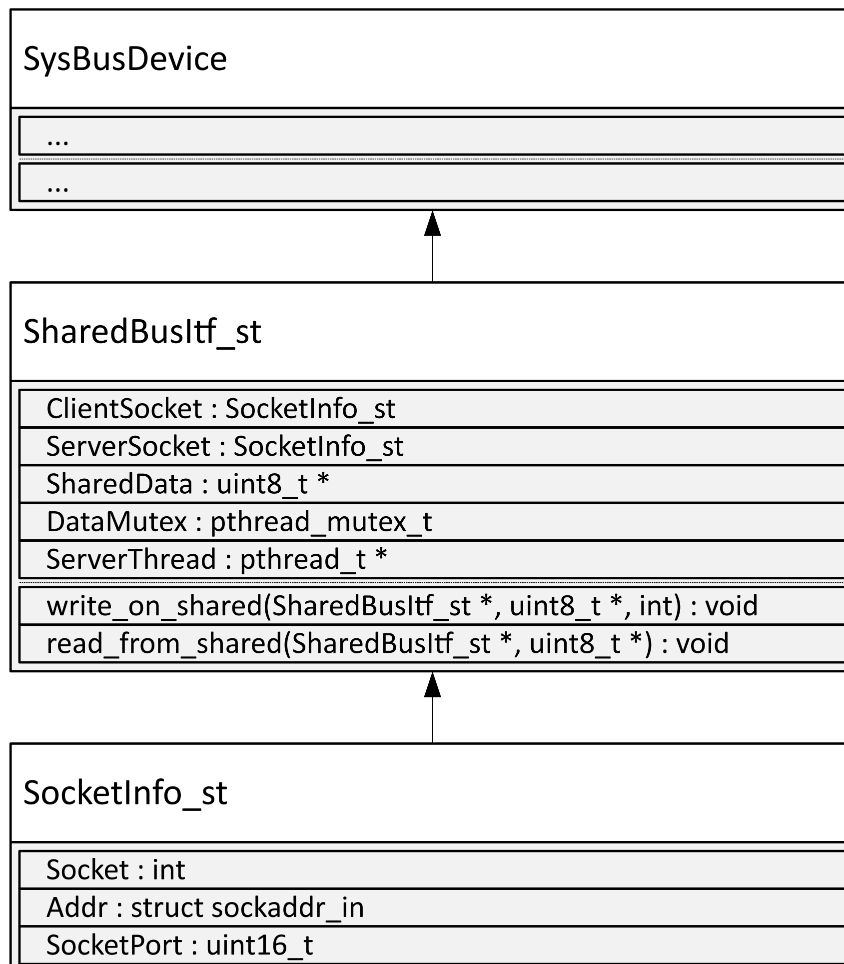


Figure 3.15: Shared Bus node interface diagram

peripheral's server socket for asynchronous data. An implementation of such thread is presented on figure 3.16. It monitors the server socket for any events and handles the data received. The data received is written in the peripheral memory space if the peripheral's configuration allows it so. Since the translation block execution thread can also access peripheral memory space, read and write operations need to be protected by a mutex, to guarantee no concurrent accesses. Beside writing on memory, if the peripheral is configured to interrupt on asynchronous reads, the peripheral specific interrupt is asserted.

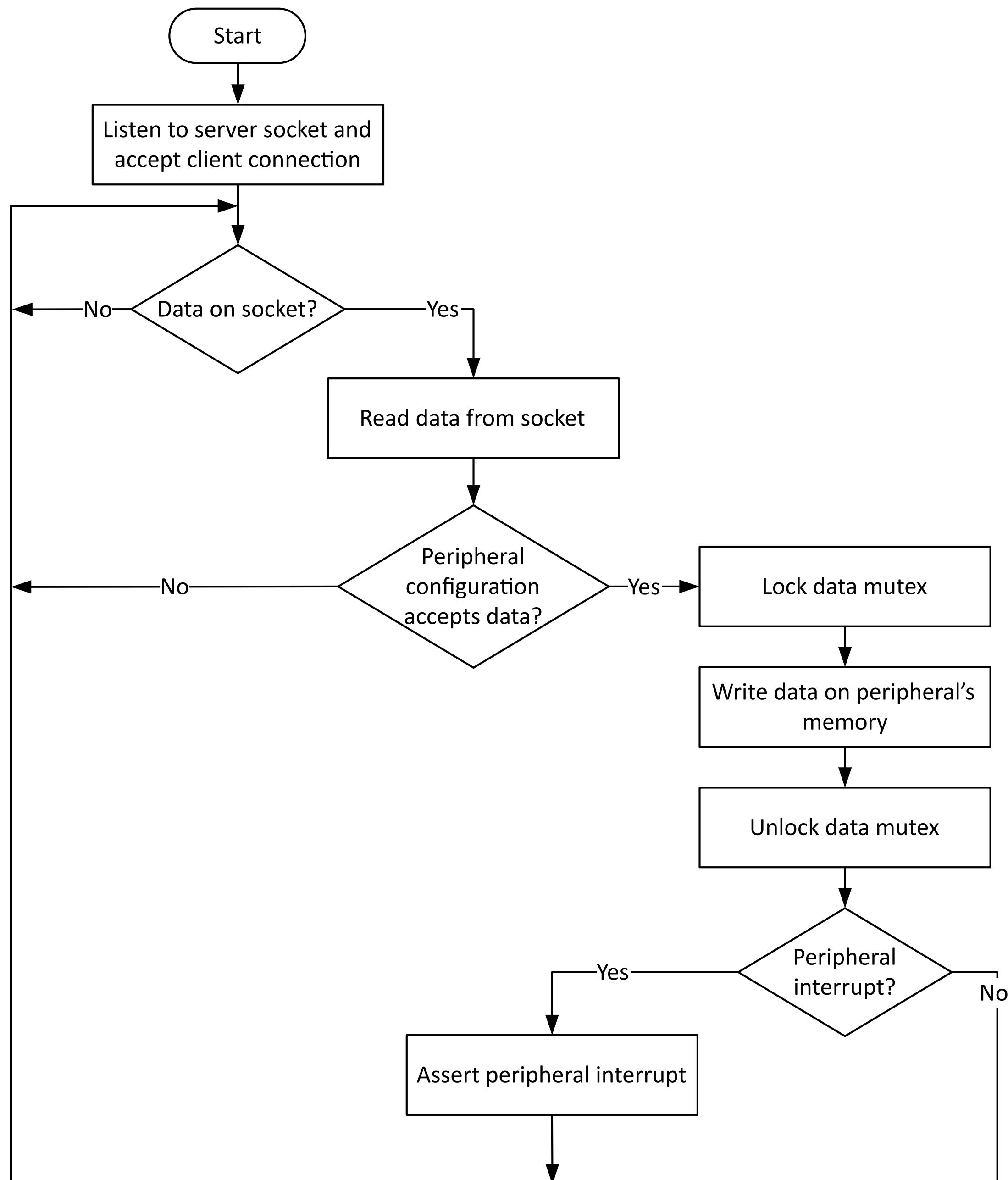


Figure 3.16: Shared bus extension thread flowchart

In order to use the Shared Bus extension, the peripherals within QEMU need to have information about

the available emulated buses to connect to. Each peripheral, depending on the type of communication, is responsible to retrieve the correct master server socket port to initially connect to the Shared Bus. The diagram on figure 3.17 shows an example usage where CAN and GPIO peripherals connect each to an instance of the Shared Bus.

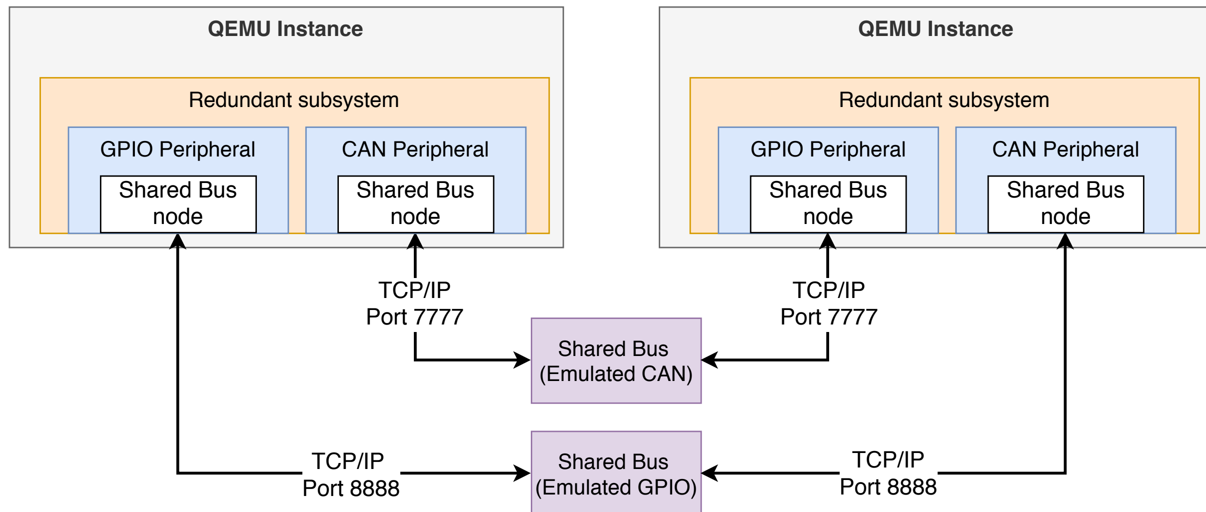


Figure 3.17: Usage of multiple Shared Bus instances by multiple peripherals

For that purpose a command line argument was added to QEMU to signal the available Shared Bus instances. The command arguments are presented on the following command line snippet:

```
$ qemu-system-arm [flags] -sb <bus type>=<server port>,
<bus type>=<server port>,...
```

From figure 3.17, the corresponding command that reflects such situation is:

```
$ qemu-system-arm [flags] -sb can=7777,gpio=8888
```

3.3 Fault Injection Extension

As mentioned on the previous chapter, fault injection can be used to evaluate reliability oriented systems. By providing faulty stimuli, and by gathering information about the running state of the simulation, the behaviour of the system can be evaluated on a software perspective, along with the efectivessness of its fault tolerance mechanisms. Under that perspective, QEMU was extended to allow fault injection capabilities on different system components.

The extension is based on the research of Andrea Höller [72]. On her thesis, a virtualization-based fault injection framework on QEMU (named FIES) was developed, aiming to assess fault tolerance during software development. The framework supports several types of faults, with the ability to be injected on different system components and at different emulation times, providing a wide use case reach. An overview of the Fault Injection framework is presented on figure 3.18.

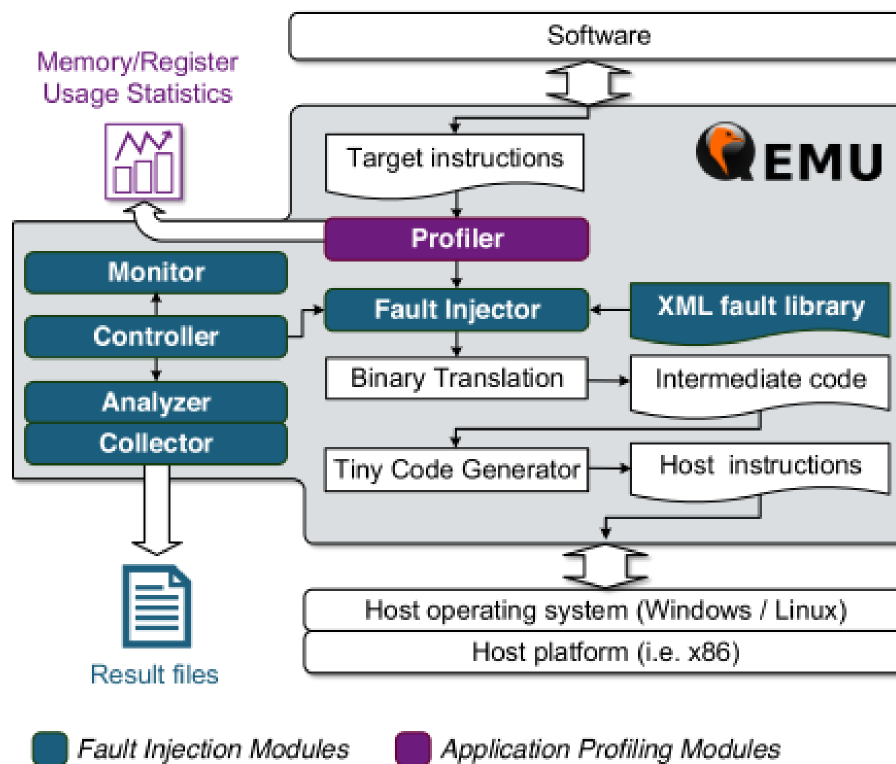


Figure 3.18: Fault injection framework from Andrea Höller PhD thesis [72]

As seen on the figure above, fault injection occurs during the dynamic translation of target code and it is supported by several modules which assist the injection. During translation, all CPU and memory operations are monitored in order to accurately inject faults. The framework injection capabilities include the faults detailed on table 3.1. All these faults can be triggered by time, program counter value or memory access, and all of them can be transient or permanent. Such capabilities provide a good starting point to adapt framework functionalities in order to cover the needs of the case study that will be explored on the next chapter.

The framework was developed for QEMU 1.7 and made public. This version of QEMU is outdated, so the framework was ported into QEMU 4.1. Since a great effort would be needed to port all fault type

Table 3.1: Details about fault locations and fault modes supported by FIES

Location	Fault Name	Fault Description
Memory	Memory cell fault	Change the content of addressed memory cell
Memory	Address decoder fault	Access an incorrect memory address
CPU	Instruction decoder fault	Replace an instruction by another instruction
CPU	CPSR cell fault	Change CPU condition flag value
Register	Register cell fault	Change CPU register value
Register	Register decoder fault	Causes an incorrect register to be addressed

modules to the new version of QEMU, only the required faults for the case-study were selected to be ported. This resulted on the port of the **Memory cell** and **Instruction decoder** fault types.

3.3.1 Fault Injection Components

The fault injection extension follows a similar layout as the one on figure 3.18. It integrates some of the components from the FIES framework and adds an external coordinator, as shown on figure 3.19. The fault injection coordinator is responsible to generate faults, transmit them to the simulation instances and make decisions according to their state, while the components within QEMU are responsible to decide how and where the faults are injected. Each one of the fault injection components will be described next.

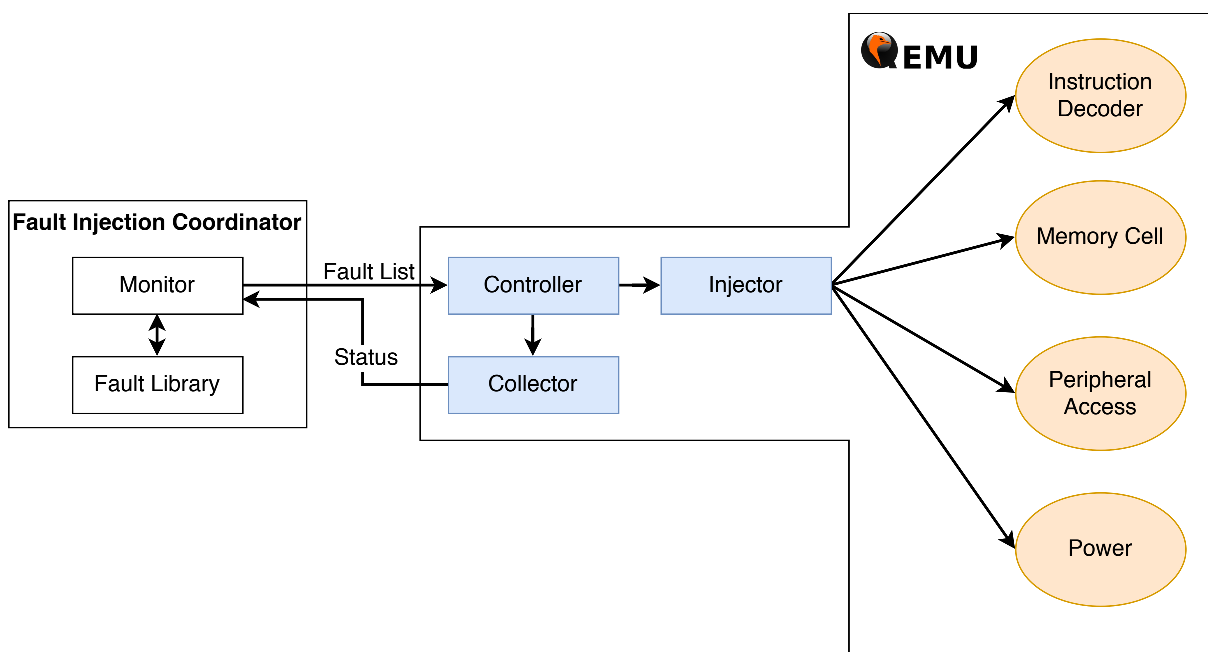


Figure 3.19: Fault injection components

Fault Injection Coordinator This independent entity generates faults and creates a fault list with them. Within the coordinator, the monitor manages incoming simulation connections for fault injection

and controls simulations based on data received from the collector. Also within the coordinator, the fault library generates the definitions of the faults using XML files. The schema of these files will be described later on the section.

Controller This component decides how to inject faults according to the fault list. The faults specified on the fault list remain on the system for a user-specified amount of execution time. Based on this information and the QEMU built-in timer, the controller decides when and where a fault should be triggered or stopped. The controller is parses the fault list, which comes as an XML file from the coordinator.

Injector The injector is the core of the fault injection. It contains functions and methods that allow injection of the different types of faults. Faults can be of four different types which occur on different execution locations: instruction decoder, memory cells, peripheral access and system power. According to the faults present on the fault list received by the controller, the fault specific functions are called.

Collector The collector gathers information about the status of the simulation after any fault is injected. The goal of this component is to gain knowledge on how the system responds to the fault by retrieving the system's internal execution status using monitor variables. This status is sent as feedback to the coordinator monitor, allowing it to make decisions regarding simulation management.

3.3.2 Fault Types

As previously mentioned on the fault injector description, faults can occur on instruction decoding, memory cells, peripheral access or system power. The first two were ported from the FIES framework while the latter ones were added to cover the needs for the case-study. Besides these type of faults, a clock fault type was also added, which is a type of fault addded specially for the case study. Each one of these type of faults will be addressed on the next subsections.

Instruction Decoder

Instruction decoder faults occurs before execution of translation blocks, when the target code is disassembled. This disassembling process happens on the translation block execution thread, specifically on the arm_tr_translate_insn function. This type of fault replaces the current disassembled host instruction by a different one. The before-mentioned type of faults are injected by modifying instruction variables directly before the translation from target to host architecture occurs. At the moment of injection, the

disassembled instruction is overwritten by the specified instruction on the fault definition and is later converted to host instructions and added to a translation block. The diagram on figure 3.20 shows how and where instruction decoding faults occur.

With this type of fault, one can observe two types of fault effects, on system-level and hardware-level. On system level, instruction faults can mean code error or data error, if the instruction performs unintended data manipulation. On hardware-level they can mean a fault on the code segment of system memory, any fault from control flow violation or even faults on the internal CPU instruction decoders or registers.

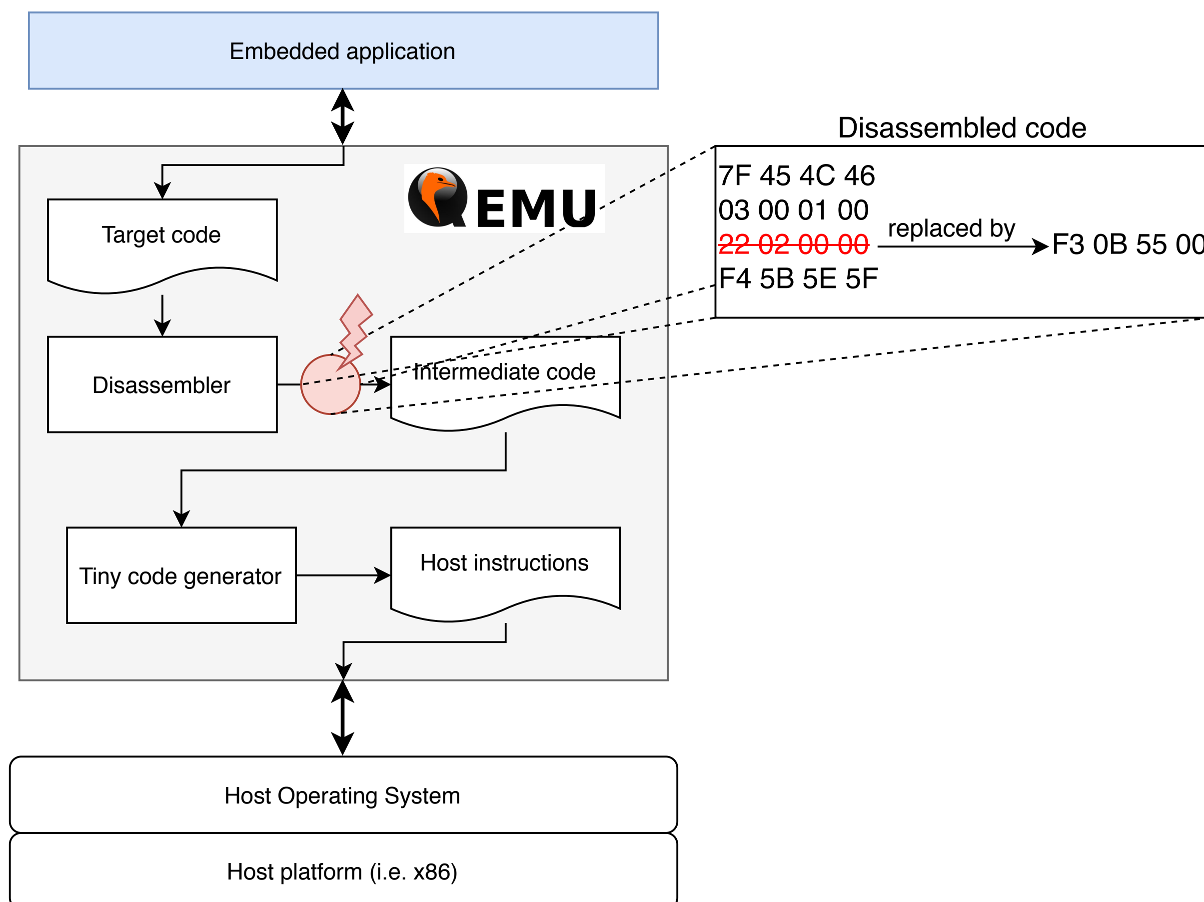


Figure 3.20: Injection of fault on instruction decoding

Memory Cell

Memory faults occur during read and write operations on physically addressable memory. Such operations are monitored and as they are realized, available memory faults are injected. This is done by checking the addresses on the R/W operations on QEMU's Soft-MMU translation functions flatview_write and flatview_read. Regarding write operations, before writing on the designated address, the value defined

in the fault overwrites the value to be written on the memory space. On read operations, the value is not overwritten on memory but the resulting value from the read operation is swapped by the value defined on the fault. Both these possibilities are shown on figure 3.21. Such faults can be used to mimic system-level fault effects such as data errors or hardware-level fault sources such as faults in RAM or data buses, or even faults in the read/write logic of the RAM controllers.

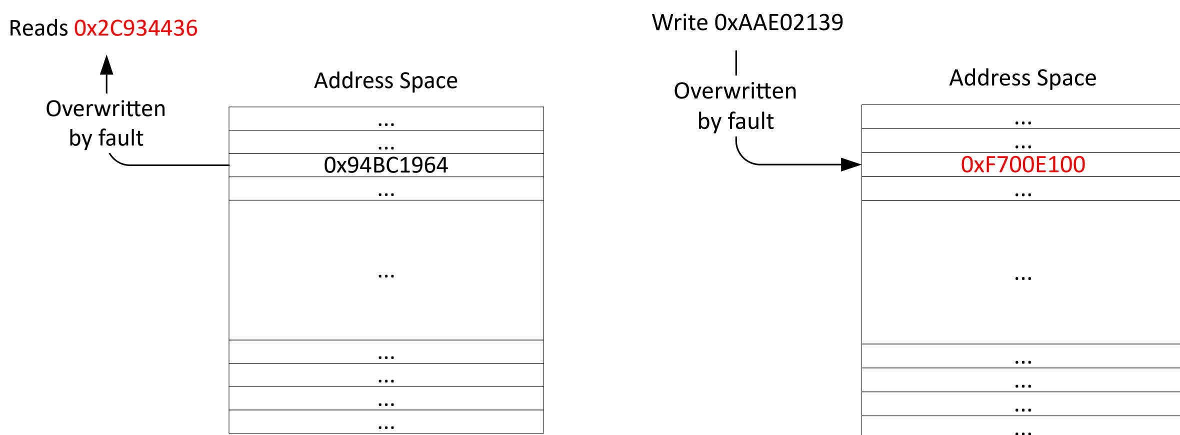


Figure 3.21: Faults during memory read and write operations

Peripheral Access

This type of fault prevents access to peripheral memory regions by the QEMU system bus. When an access to a peripheral's memory region is made, this type of fault blocks calls to the read/write operations functions that emulate peripheral behaviour. Meaning that these type of faults makes the peripheral unavailable for usage. This is done by monitoring memory access during both read and writes on the QEMU system bus (functions memory_region_read_accessor and memory_region_write_accessor) and blocking any access that matches the fault specified address. Figure 3.22 shows a diagram of this type of fault, where interactions between an emulated peripheral and the system bus are blocked, preventing transactions. Peripheral access blocks can be used to emulate real hardware faults on the peripherals, such as component failure by power spikes or electrostatic discharges.

Power

Power faults aim to simulate a power failure on the system. This is done by forcing the QEMU instance to reset the CPU and every peripheral using the native API qemu_system_reset_request. Although this resets the simulation, the total simulation time is not affected, as QEMU keeps track of the simulation

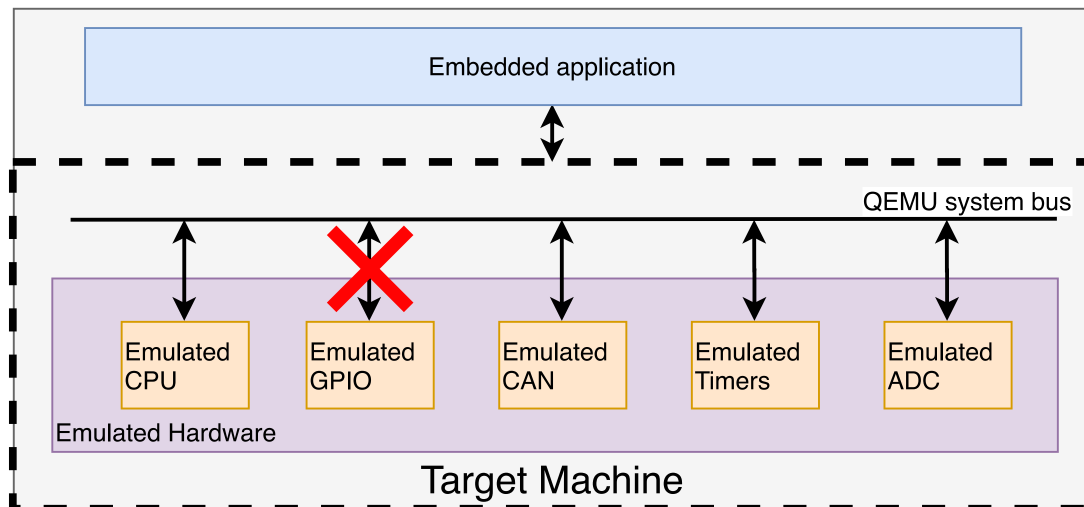


Figure 3.22: Injection of faults blocking peripheral access

time up until shutdown of the instance. This type of fault is particularly important for redundant systems, since it allows to evaluate the system behaviour when a redundant module fails.

Clock

A special case of faults that were added to the extension are clock faults. This was added aiming to emulate clock drift type of situations between redundant subsystems. Since QEMU does not emulate real clock timings (code runs as fast as possible), real clock speed drifts are not possible to represent. With that in mind, and knowing that all simulations obey to the synchronization process, a clock fault means loss of synchronization between a simulation and the synchronization process. This is done by dropping the communication between them and letting the simulation run at its own pace. Although it is not possible to know exactly how much the simulation will be delayed or sped up in relation to other simulations, the outcome is always considered a clock drift since execution speed varies in relation to the wall-clock time.

3.3.3 Fault Description XML file

Fault description and parameters are contained in an XML file, which is written by the coordinator and read by the fault controller. This file contains the fault list, described by the following fields:

- <id> Defines fault ID
- <component> Location of fault: CPU (for instruction decoder), MEMORY, PERIPHERAL, POWER or CLOCK

- <params>

  <address> Memory address to inject memory fault or peripheral address to block operations

  <mask> Bit mask for memory fault or new instruction for instruction decoder fault

An example of an XML file containing faults is presented on the listing below.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <injection>
3   <fault>
4    <id>1</id>
5    <component>CPU</component>
6    <params>
7     <mask>0xE7FE0000</mask>
8    </params>
9   </fault>
10  <fault>
11   <id>2</id>
12   <component>PERIPHERAL</component>
13   <params>
14    <address>0x40048004</address>
15   </params>
16  </fault>
17  <fault>
18   <id>3</id>
19   <component>MEMORY</component>
20   <params>
21    <address>0x20000174</address>
22    <mask>0xFE</mask>
23   </params>
24  </fault>
25 </injection>
```

Listing 3.1: Example of a fault list XML file

3.3.4 Fault Injection Coordinator

The fault injection coordinator is responsible to generate the fault list as an XML file and manage incoming and running simulation connections. The implementation of the coordinator is not generic and

it is the developer burden to implement it in a way it satisfies the simulation needs. This is because the Fault Library and simulation management decisions are case-study specific and depend on what the user wants to observe as simulation result. With that in mind, the current section describes the coordinator that was specifically designed for the case-study that will be addressed in the next chapter.

This coordinator starts by allowing QEMU simulations to connect to it through a TCP/IP socket. Upon connecting, instances can request new fault lists. The fault list is generated according predefined faults already present on the Fault Library. Everytime the simulations request a new fault list, a new list is created from the predefined faults and the XML file is overwritten with this new list. After adding all the faults, the instances are notified that a new fault list is available for reading. The sequence diagram on figure 3.23 shows the interactions between QEMU and the coordinator when a fault list is requested.

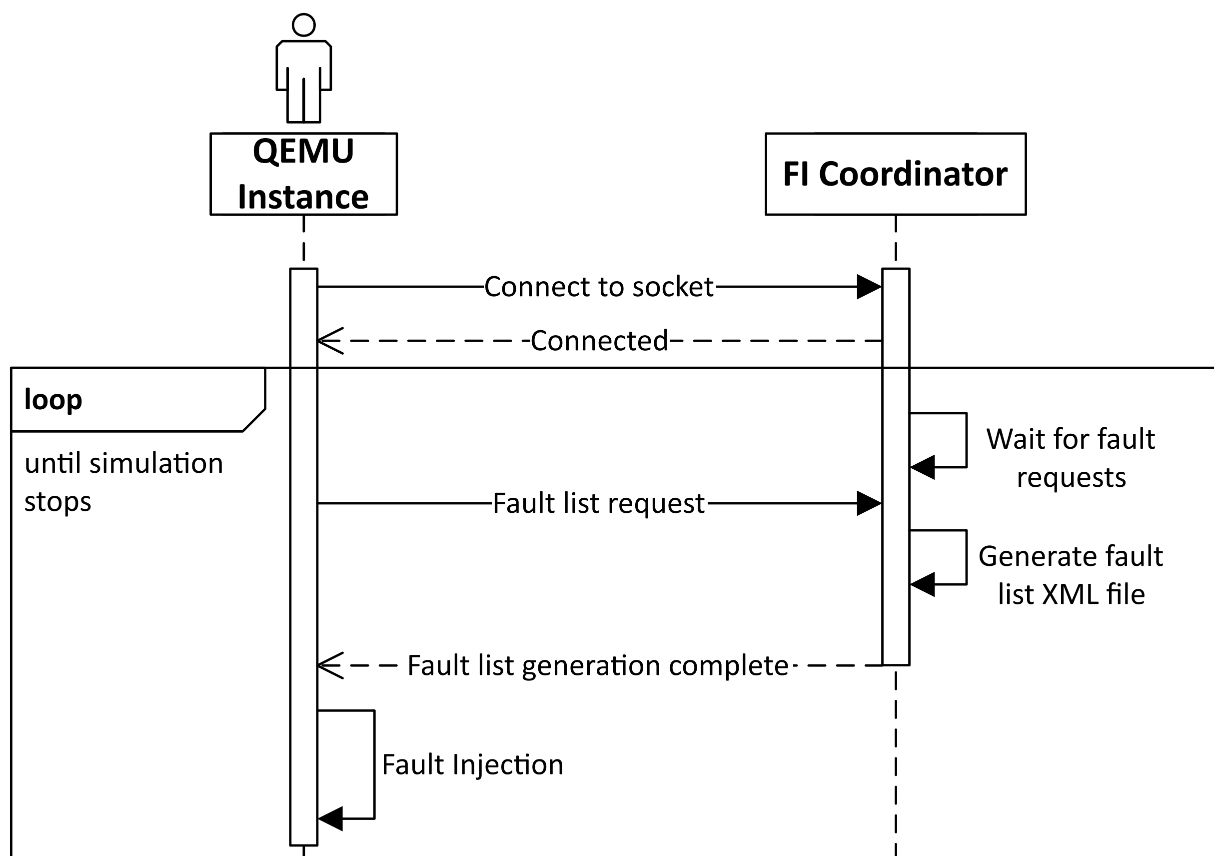


Figure 3.23: Fault request interactions sequence diagram

The data logged by the collector and read by the coordinator, allows to make decisions wether to restart or keep simulations running in order to gather overview the simulation state. For example, if both simulations keep running on an error state, it may be reasonable to restart the simulations to allow another experiment to occur. Everytime a fault request occurs, the coordinator monitor checks the variables logged

by the collector and makes a decision on the running state of the simulation. The flowchart on figure 3.24 presents the working principle of the coordinator, where simulations are reset upon an error condition.

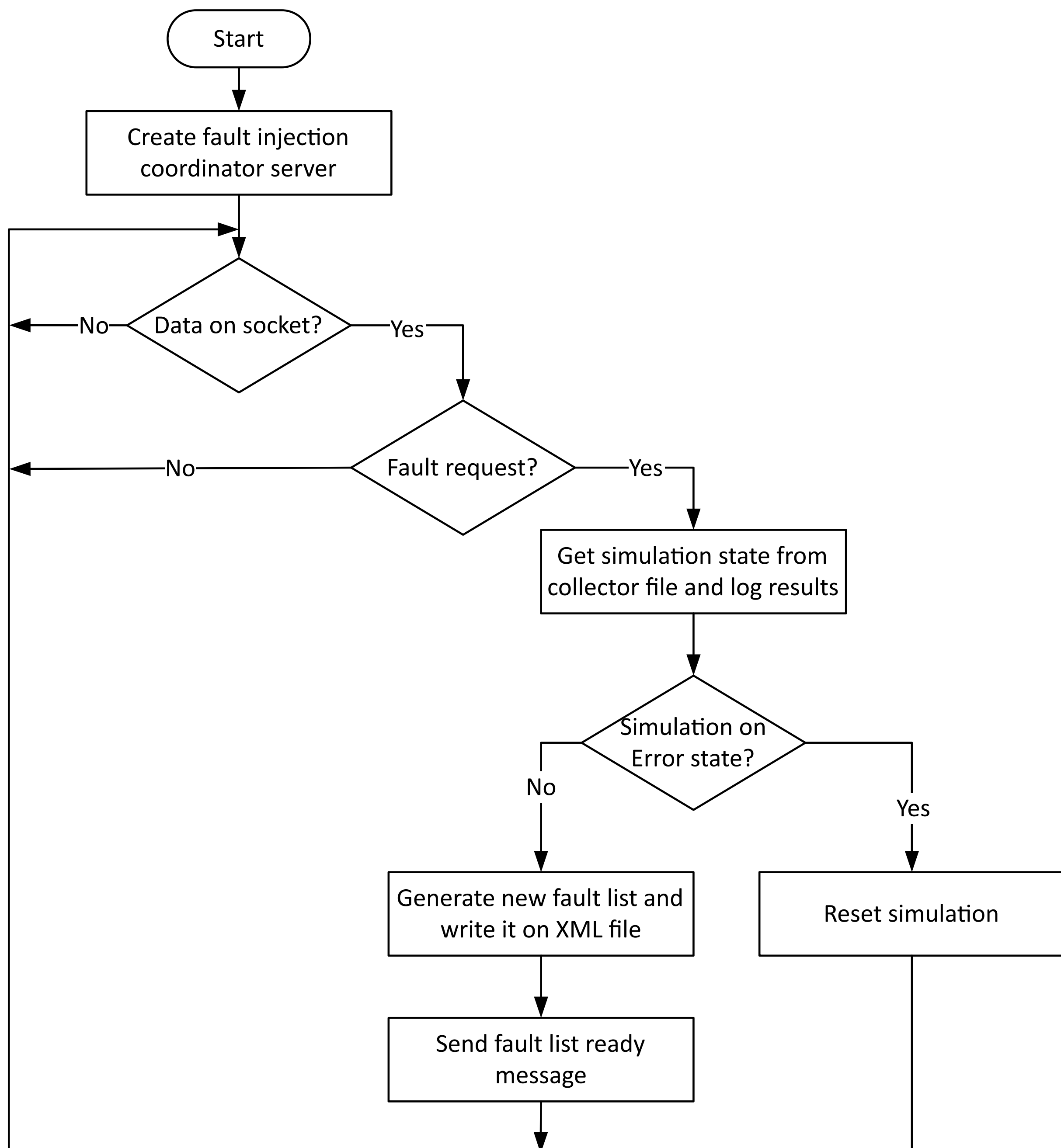


Figure 3.24: Fault Coordinator flowchart

3.3.5 Fault Injection Interface

For all fault injection purposes, a command line argument was added to define the coordinator port, the name of the XML file that contains the library, the start time to inject faults, the fault time period

which faults are requested and a list of monitor variables. An usage example is presented on the following snippet:

```
$ qemu-system-arm [flags] -fi <server port>,<xml filename>,
<start time>,<time period>,<monitor variable list>
```

Before starting emulation, QEMU retrieves fault injection arguments and both connects to the coordinator server and retrieves the addresses of the monitor variables. These addresses are retrieved by using the Linux *readelf* command and getting the assigned addresses of the variables on the application .elf file. After executing this command along with the .elf filename argument, the variable names are searched in the resulting command output. Upon finding them, the corresponding addresses are added to the list of variables to be logged upon fault injection. The flowchart on figure 3.25 shows the initialization process.

During runtime, fault requests are made at a timed period as provided by the user. This is done using QEMU's deadline timers assuring that injection is made at the correct time. At each deadline, monitor variables are logged, providing feedback to the coordinator about the results of the injection. Then a new fault list XML file is requested, read and parsed by the fault controller, in order to retrieve the next faults to inject. For every fault present, the corresponding injection functions and methods are called to allow said fault in the system. Since CPU and memory faults are injected through monitoring execution, these type of faults are not immediately injected upon receiving the fault list. After all faults are injected, or signaled for injection, a new deadline is created with the same time period that the user previously specified. The injection process is presented on the flowchart on figure 3.26.

Before fault lists are requested, the data from the monitor variables are logged. These variables are application specific and user specified at the start of the simulations. To get the data relative to the monitor variables, QEMU's native function cpu_memory_rw_debug is used to retrieve the data from these variables, taking advantage of the soft-MMU capabilities to access application's memory space. The resulting variable values are saved on a file which both QEMU instances and the FI coordinator have access.

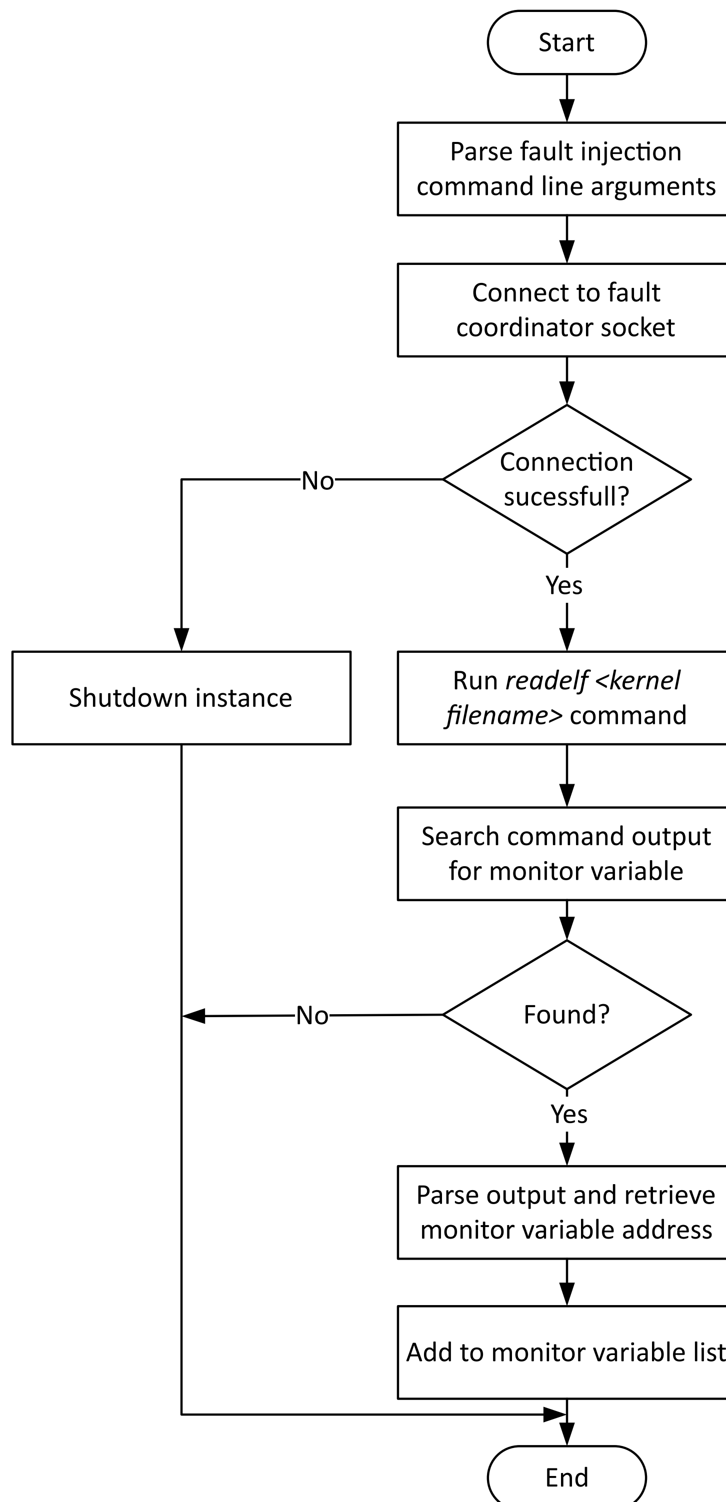


Figure 3.25: Fault injection initialization flowchart

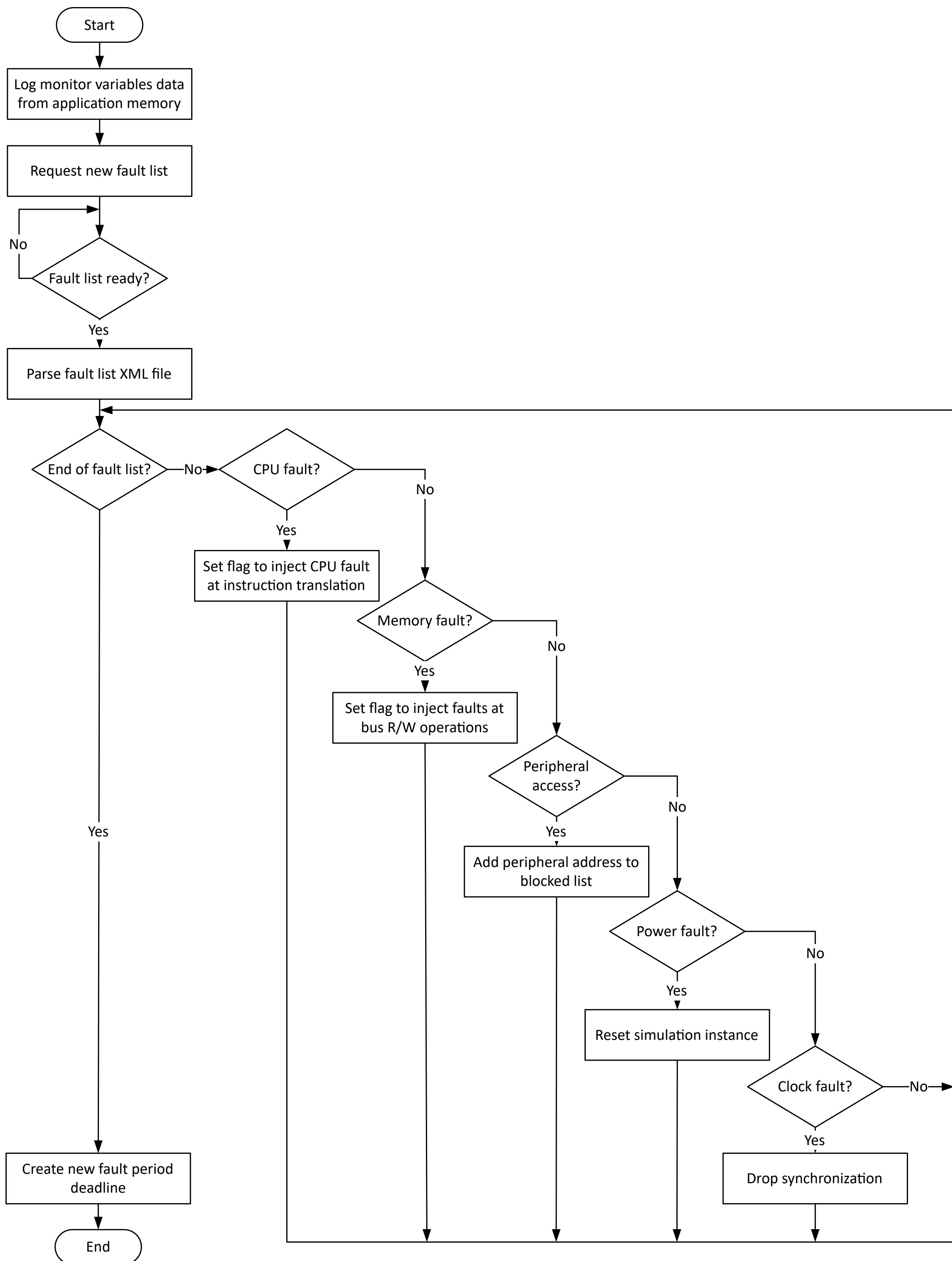


Figure 3.26: Fault injection flowchart

3.4 Summary

This chapter described the extensions that were developed in the context of reliability development. Three extensions were developed, tackling the problems that arise from simulation of multiple processing system, and allowing for reliability estimation through simulation. These extensions were developed to be integrated with QEMU, taking advantage of its full system emulation capabilities.

The Synchronization extension mitigates the synchronization issues that arise from simulating processing systems independently. For that purpose, a time budget execution concept was implemented, avoiding potential causality errors. The Shared Bus extensions allows for different simulations to communicate through their peripherals, enabling interactions between them. This was done by emulating data buses, allowing peripherals to connect to them and make read and write operations on them. Lastly, the Fault Injection extension provides support for reliability estimations, by allowing to inject faults in instruction decoding, memory, peripheral access and power supply. Alongside the description of the extensions and their mechanisms, information about how to use the extensions was also provided, giving developers the needed knowledge to take advantage of them.

Chapter 4

Case Study

The integration of the developed extensions presented on the previous chapter results in a simulation environment that assists not only redundant architectures design and testing, but also evaluation of reliability metrics. To validate such work, a case study that meets redundant architecture characteristics was used. This chapter describes the case study that was chosen and demonstrates the work developed in a practical context. The selected case study is the Steering Angle Sensor (SAS), which is an automotive sensor that fits in a Steer-by-Wire paradigm.

The SAS is a joint effort between University of Minho and Bosch Technology and Development Center. While Bosch dictated the requirements and initial architectural approach, the University team had great influence on both hardware and software decisions. Under the system and product imposed requirements, an iteration of hardware and software was developed by the University team, but in the context of this dissertation, software will be the main focus.

The case study followed the previously proposed development flow for reliable systems, supported by development methodologies that are part of the embedded development flow. The extent of the work done on the case-study covers development phases up to reliability estimation of the system software. All remaining steps that occur in parallel, such as hardware reliability estimation, will not be addressed in the context of this dissertation.

4.1 Steering Angle Sensor

Advances made in the automotive sector over the last two decades are largely associated with the use of Electric/Electronic (E/E) systems instead of mechanical systems [24]. The introduction of the Steer-by-Wire concept fits these advancements as it aims to replace the mechanical linkage between the steering wheel and the motor wheels with an E/E system. Under the Steer-by-Wire, the Steering Angle Sensor (SAS)

is a solution, providing the handwheel angle that is used to steer the vehicle. As such, vehicle stability is directly dependent on the correctness of this information and any discrepancies may lead to an accident.

Due to the fact that this type of system may lead to potential human hazard upon failure, it is classified as the top Automotive Safety Integrity Level (ASIL), known as ASIL-D. This means that it must follow reliability characteristics that allow the system to still operate in case of failure (fail-operational), which, in this case, is mainly done by the use of redundancy.

Regarding the functional requirements, the SAS system must acquire angle data from the sensing elements and send the processed angle via CAN every 10 milliseconds. For this purpose and to meet fail-operational characteristics, the architectural concept that will support this system is presented in figure 4.1.

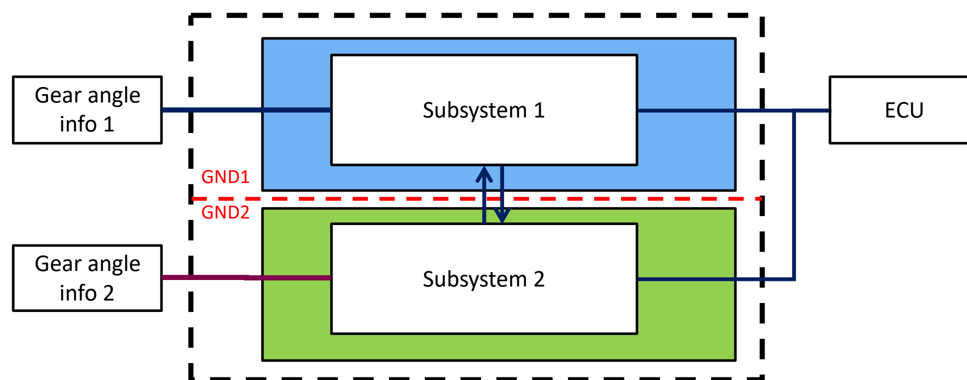


Figure 4.1: SAS architecture concept block diagram

The system is composed by two completely identical subsystems that perform the same operations and run the same software. Both subsystems are isolated from each other, although there is a communication channel between them to exchange information. They both connect to the same output channel, sending angle information through CAN bus. Both subsystems send the angle information every 20 milliseconds, but one of them is 10 milliseconds delayed from the other. This way, the 10 milliseconds output requirement is still accomplished, even when subsystem outputs happen every 20 milliseconds. The diagram in figure 4.2 visually shows the messages sent by the subsystems in a timely manner.

If a subsystem fails, redundancy allows for the system to be operationalby reconfiguring the working subsystem to meet the 10 milliseconds deadline. This mode of operation is named Fail Degraded and it assures that the working module outputs a valid angle message every 10 milliseconds. The diagram in figure 4.3 visually shows the messages sent in a timely manner when one of the subsystems fails.

Typically, development would start by gathering the requirements but since that was already done by Bosch, this step was skipped and development started by modelling the application.

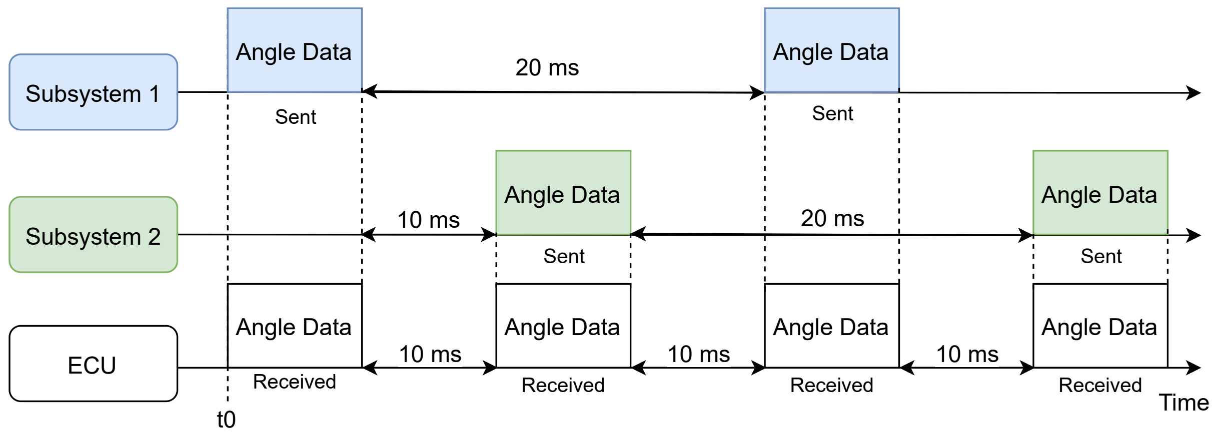


Figure 4.2: SAS angle messages timely diagram

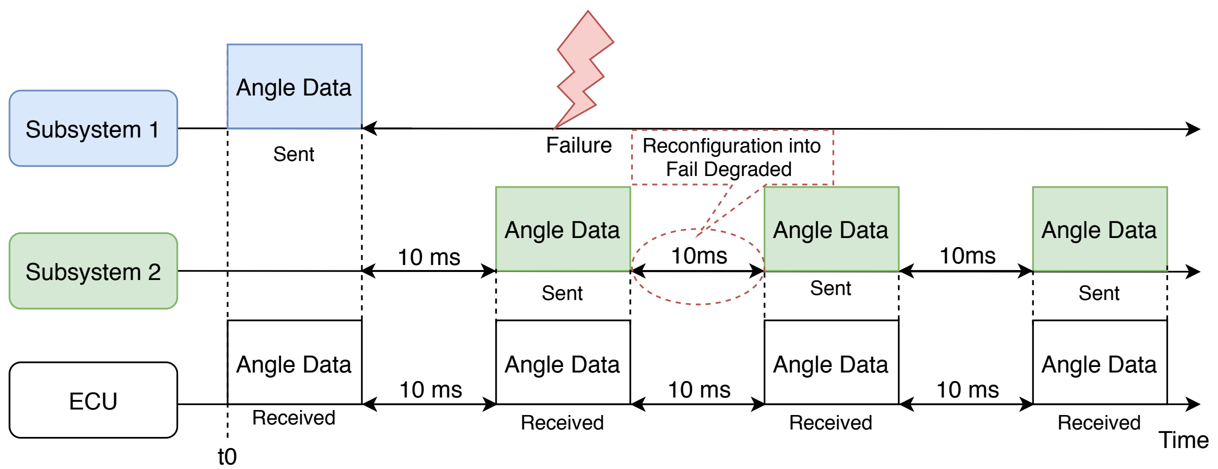


Figure 4.3: SAS Fail Degraded angle messages timely diagram

4.1.1 Application Modeling

The application modelling phase aimed to gather knowledge about the possible system tasks, by only taking into consideration project requirements. Although redundancy is already a requirement, validation in this phase contemplated the system without redundancy. This allows to have an insight about the system behaviour regardless of its fault tolerance mechanisms, and their influence on the system.

This process of validation went through a software-only approach, implementing system tasks as threads running on the *host* computer. The system was summed up in a threaded execution as presented in figure 4.4, supported by the class diagram included in appendix C.1. From the diagram, three main system tasks were identified: (1) a sampling task, which retrieves data from the sensing elements; (2) a calculation task that applies an algorithm to convert the sample data to nominal angle value in degrees; (3) a transmission task responsible to output the data from the system.

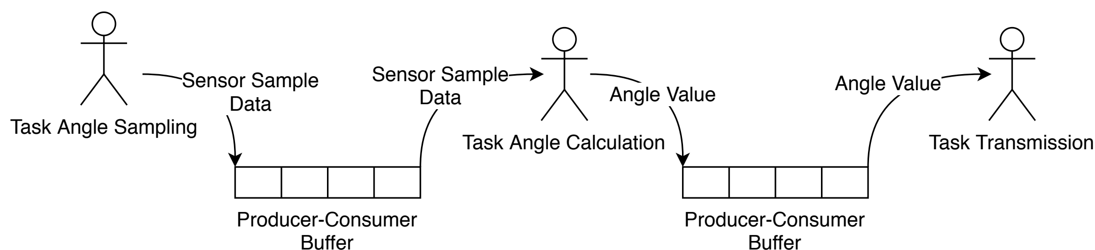


Figure 4.4: System tasks modelling through threaded execution

After this validation, the next step is to design a system architecture that fits the project requirements. This architecture provides the basis for both the software and hardware architectures that will be developed on the design phase.

4.1.2 Architecture Modelling

A system architecture conceived from the concept in figure 4.1 was initially proposed by Bosch, containing the needed components to tackle project requirements. The ESRG team suggested a change on the architecture, which consisted in adopting an isolated communication channel between the subsystems, instead of relying on combinational logic to inform about the operation state of the microcontroller. This opens up the usage of various types of communication protocols, and the possibility to interchange more complex message exchange between systems.

The resulting architecture and its constituent elements is depicted in figure 4.5. Each subsystem is composed by a microcontroller, a CAN transceiver, and an external Watchdog, and can interact with each

other using a communication channel. The subsystems are powered by independent power supplies, avoiding possible common power faults. Each subsystem connects to a sensing element of the same type, providing both subsystems with homogeneous information.

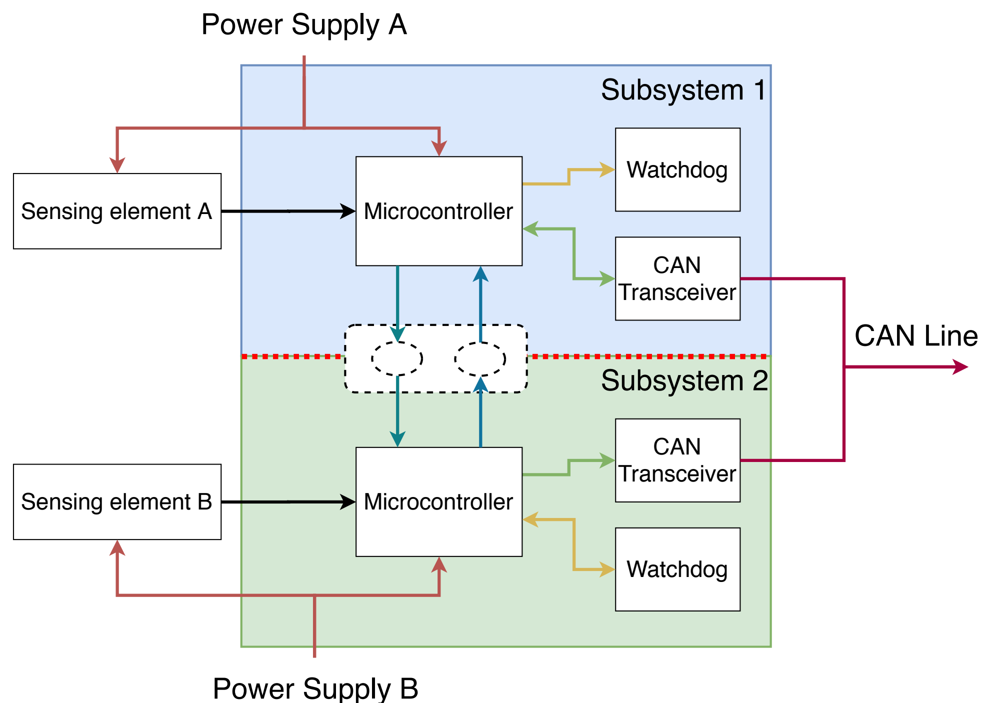


Figure 4.5: SAS system architecture diagram

Following the architecture design and project requirements, system functionalities were designed and described as a state machine, providing an overview of its behaviour and allowing for easier validation. The flow of the system will be described next, followed by sections of the state machine. The state machine describes the behavior of a single redundant module, given that both modules will run the same software. This means that the described behaviour happens on both subsystems, although not at the same time. The full state machine is included in appendix C.2.

Both redundant modules start on a Reset state (figure 4.6). On this state, the microcontroller is initialized, the CAN transceiver is enabled and an initial error check is made. This error check verifies the occurrence of a previous watchdog reset or initializations errors. These errors are assumed to be Boot Time Error Conditions. If no errors are found, both modules jump to a Race Condition state, otherwise they are prevented from booting, entering Error State. The Error State is a state that disables all redundant module functionalities by disabling the CAN transceiver and entering an infinite loop.

The Race Condition (figure 4.7) state solves the race condition that occurs when both redundant modules access the CAN bus resource. In this state, both modules check for the two serial numbers

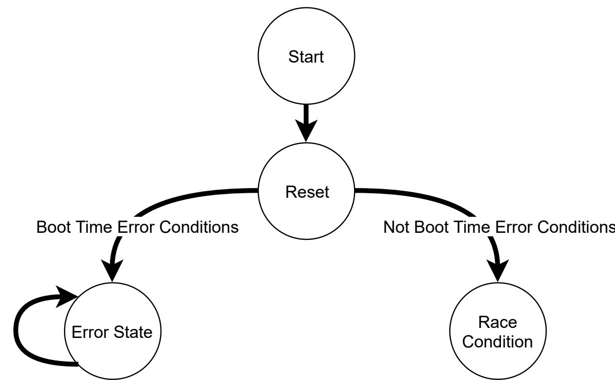


Figure 4.6: SAS state machine section (1)

stored on non-volatile memory, which correspond to the factory issued microcontroller serial numbers of both redundant subsystems. Depending on this number, one of the modules takes the lead and is the first one to transmit an angle value through the CAN bus. The remaining redundant module is delayed by entering the Halt state, while the lead module jumps to the Angle Sampling state. When the delayed module enters the Halt state, the modules no longer are on the same state at the same time. In the Halt state, the delayed module waits for a synchronization pulse through the communication channel, which occurs after 10 milliseconds passed, when the lead module transmits data.

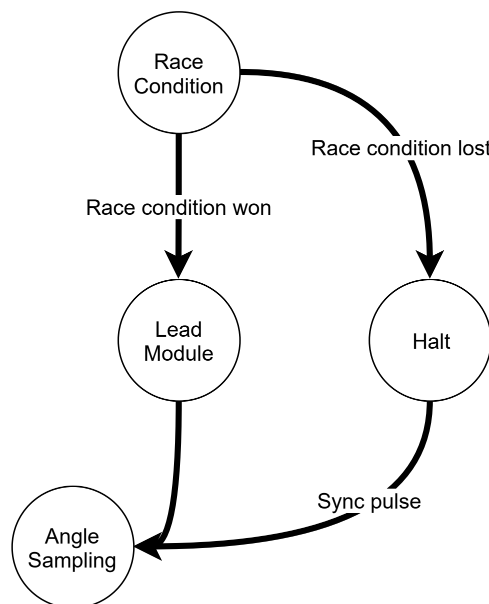


Figure 4.7: SAS state machine section (2)

The Angle Sampling (figure 4.8) state starts both the sampling sequence and a timeout value of 10 milliseconds, and jumps to Angle Calculation when the sensing elements have been sampled by the ADC. On the Angle Calculation state, the nominal steering angle is calculated from the sampled ADC values.

The first time a system goes through this state, the 10 millisecond timeout has no effect and the system goes immediately to the Transmission state (if lead module) or to the Redundant Module Check (if delayed module), accordingly to the previously solved race condition. In this same state (Angle Calculation), there is also a check for a CAN transmission error from a previous CAN transmission, and if that happens, the system goes to Error State.

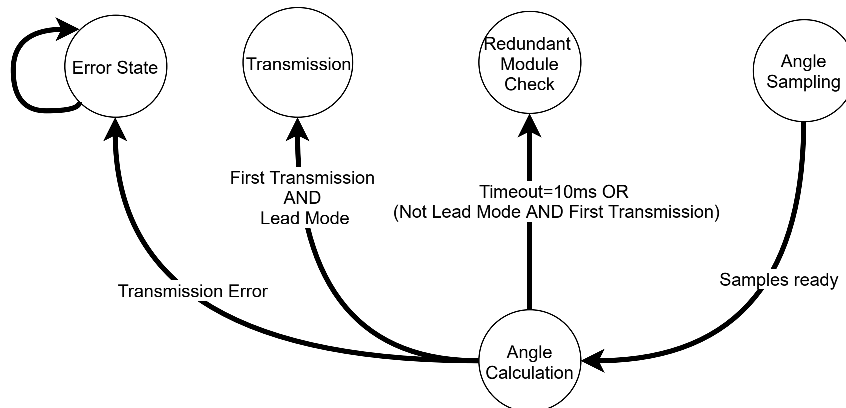


Figure 4.8: SAS state machine section (3)

The Redundant Module Check state (figure 4.9) verifies synchronization between redundant modules by checking if a CAN transmission was made within a time window. If a transmission was made within a time window of 2 milliseconds upon reaching this state, the module assumes correct synchronization with the redundant module and stays in that state for 10ms. After this time, it jumps to the Transmission state. If no transmission was made, the system assumes that there was an error by the redundant module and goes to Fail Degraded state. This state ensures that the system has a correct throughput of the Angle value every 10 milliseconds by both modules.

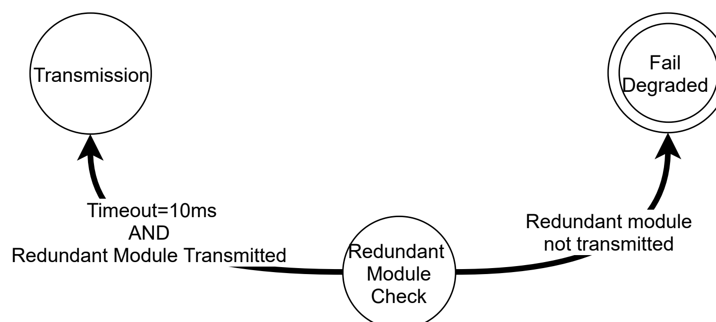


Figure 4.9: SAS state machine section (4)

Upon reaching the Transmission state (figure 4.10), there is a check if the redundant module is transmitting data. If that is the case, it means that an error has happened and the system goes to the

Error State. Otherwise, a synchronization pulse as feedback is sent through the isolated communication channel indicating transmission. If no failure occurred at the moment this feedback is sent, the redundant module is on Redundant Module Check state. Upon successful transmission, the module goes back to the Angle Sampling state.

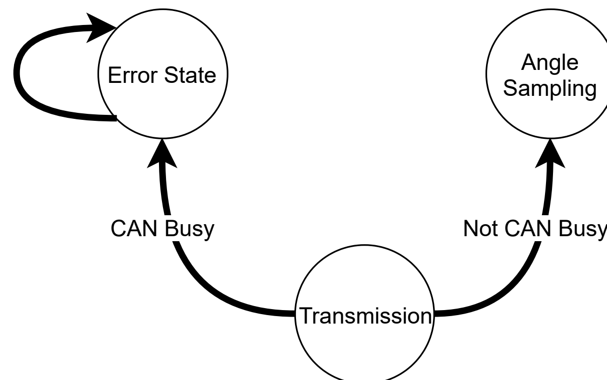


Figure 4.10: SAS state machine section (5)

The Fail Degraded state (figure 4.11) is the state where, upon failure, the working module reconfigures itself and assures a message throughput of 10 milliseconds. Upon entering this state, an indication of fail degraded operation is saved on the non-volatile memory.

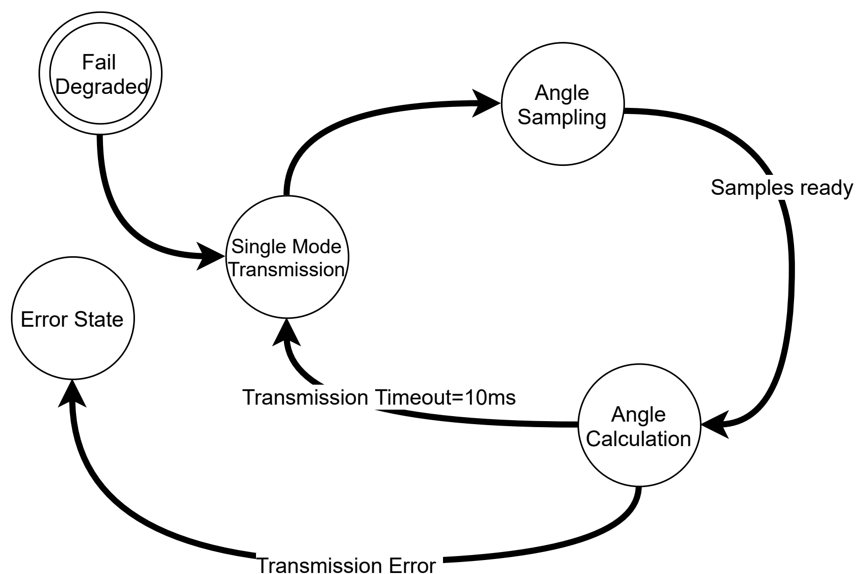


Figure 4.11: SAS Fail Degraded state machine

After system architecture and corresponding state machine completion, validation of the latter was done to check if all the functional requirements were attended. After validation, development was split into the hardware and software architecture design and implementation. As this dissertation is software oriented, the hardware development will not be addressed.

4.1.3 Platform Decision

Platform selection for this project was a Bosch decision. Due to the ASIL-D requirements of the project, the S32K2 architecture by NXP, which is scheduled for AUTOSAR support, was selected. The platform was not available at the start of the project, and to allow for an immediate start, an architecturally compatible platform was adopted, more specifically the S32K1. The lack of AUTOSAR support for this platform led to Bosch deciding to use to use a bare-metal environment for the application. Despite this decision, software was agreed to follow the AUTOSAR API as closely as possible. This would enable easy future application porting for the S32K2 platform in case of de facto full AUTOSAR support, and even module porting in case of partial support availability.

As such, the platform used for the project was the S32K1 from NXP, more specifically, the S32K116 microcontroller. The platform is based on a 32-bit ARM Cortex M0+ machine within a SoC that is specially designed for Automotive applications. The most important platform features for the project, regarding peripherals, are: 12-bit Analog-Digital Converter, CAN modules, and a wide variety of peripherals.

4.1.4 Software Architecture

Based on the agreed requirements and restraints of the platform selection, the developed software architecture is based on two layers: the Microcontroller Abstraction Layer (MCAL) and the Application. The MCAL provides support drivers for the Application to interact with the hardware, while the Application layer runs code relative to the system state machine, as previously shown. The diagram in figure 4.12 presents the software stack used for the SAS. Initially, The ESRG team, only had the responsibility to develop the MCAL, following the AUTOSAR interface specification and guidelines. As the project progressed and due to other factors, the ESRG team also implemented the Application layer.

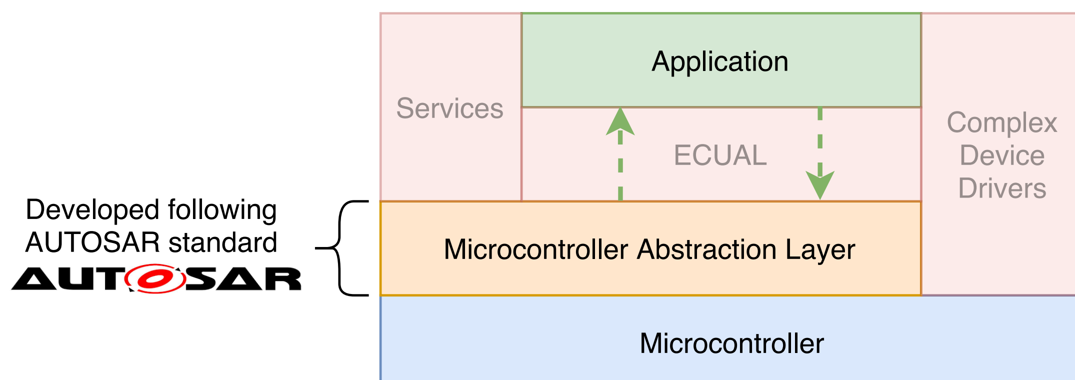


Figure 4.12: SAS software stack

By Bosch decision, the software was set to be developed in C language, using the NXP native toolchain for the chosen platform, which is based on the GNU Arm Embedded Toolchain.

Microcontroller Abstraction Layer

MCAL is the layer that interacts directly with the microcontroller hardware, providing drivers and the needed interfaces for the upper layers to interact with it. Within the MCAL layer specification, there are several modules that operate on microcontroller hardware modules, but not all of them are needed for the case-study. The diagram in figure 4.13 shows the modules that were developed, to cover case-study needs.

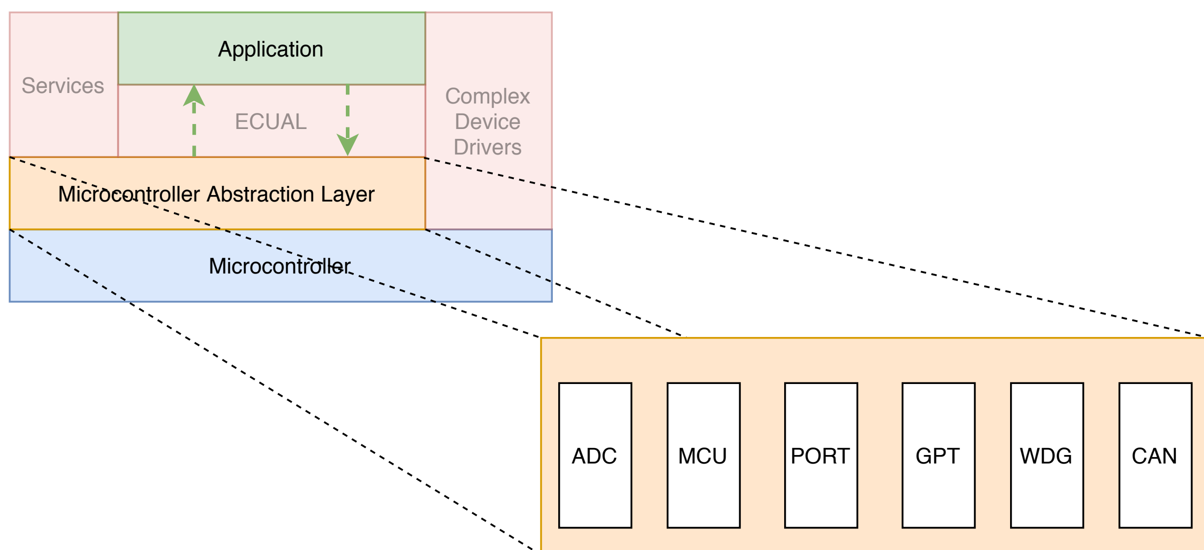


Figure 4.13: MCAL modules developed

The modules follow the specification of the AUTOSAR standard, providing all the necessary functionalities and the correct interfaces. Each module is described as follows:

- ADC - The ADC module initializes and controls the internal Analogue Digital Converter Unit(s) of the microcontroller. It provides services to start and stop conversions, enable and disable trigger sources and notification mechanisms and routines to query the status of a conversion. The conversions can be continuous, being periodically done, or one-shot, and can be triggered by either software or by hardware modules, such as a time peripherals. Additionally, the ADC can be configured as streaming, allowing to save previous conversions in either circular or linear buffers, or single access, discarding previous conversions. Internally, conversions are controlled by state machines, which greatly depend on how the ADC is configured. These state machines are specified on the AUTOSAR standard.

- MCU - The MCU module provides services for basic microcontroller initialization, run state, power down and reset functionalities, and microcontroller specific functions required from other MCAL software modules. This module is responsible for initializing modules such as peripheral clocks, initialize memory regions, set the microcontroller power mode, etc.

- PORT - The PORT module applies to the on-chip ports and port pins, providing services to initialize the whole port structure of the microcontroller. Assignment of pins can be made to peripherals such as analog conversions, PWM output, general purpose I/O, etc.

- GPT - This module initializes and controls all the internal General Purpose Timer(s) (GPT) of the microcontroller. The GPT module provides services to start and stop hardware timers, get timer values and control interrupt triggered notifications. It is assumed that any timer module within the microcontroller, that allows for multiple functionalities, is a general purpose timer, but the number of timer channels controlled by the GPT module depends on the available hardware, implementation and system configuration. In this case, three timers were implemented, resulting in eight independent timer channels.

- WDG - This module provides services for initialization, changing the operation mode and triggering watchdogs. Since there is an external watchdog, this module was developed for both the internal and external watchdogs.

- CAN - The CAN module provides services for initiating transmissions and managing callback functions for notifying events. Since this module was initially meant to be provided by Bosch, only the basic functionalities were developed.

Before starting MCAL development, the diagrams that specify each module were designed as a way to guide further development. The diagrams include class diagrams, use cases and sequence diagrams. The class diagrams, as the one presented in figure 4.14, define the interfaces used on each module, which are specified by AUTOSAR. The full version of the class diagrams developed for each module are shown on appendix C.3.1.

The use case diagrams identify, for each functionality, the required microcontroller hardware peripherals and the dependencies on other software modules. An example of a simplified use case diagram for the ADC module is presented in figure 4.15. The ADC module interacts with other MCAL modules such as the MCU and PORT, in order to initialize itself. It also accesses peripherals or hardware modules such

Adc
CurrentConfig : Adc_ConfigType * CurrentGroup : Adc_GroupType
Adc_Init(const Adc_ConfigType* ConfigPtr) : Std_ReturnType
Adc_DeInit(void) : void
Adc_SetupResultBuffer(Adc_GroupType Group, const Adc_ValueGroupType* DataBufferPtr) : Std_ReturnType
Adc_StartGroupConversion(Adc_GroupType Group) : void
Adc_StopGroupConversion(Adc_GroupType Group) : void
Adc_ReadGroup(Adc_GroupType Group, Adc_ValueGroupType* DataBufferPtr) : Std_ReturnType
Adc_EnableHardwareTrigger(Adc_GroupType Group) : void
Adc_DisableHardwareTrigger(Adc_GroupType Group) : void
Adc_GetGroupStatus(Adc_GroupType Group) : Adc_StatusType
Adc_GetStreamLastPointer(Adc_GroupType Group, Adc_ValueGroupType** PtrToSamplePtr) : Adc_StreamNumSampleType
Adc_Pdb_Init (void) : void
...

Figure 4.14: Section of the ADC module interface diagram

as the PDB and ADC to configure the hardware level registers. Under other interfaces, such as reading or starting conversions, the module only needs to interact with the ADC peripheral. The remaining use case diagrams are included in appendix C.3.2.

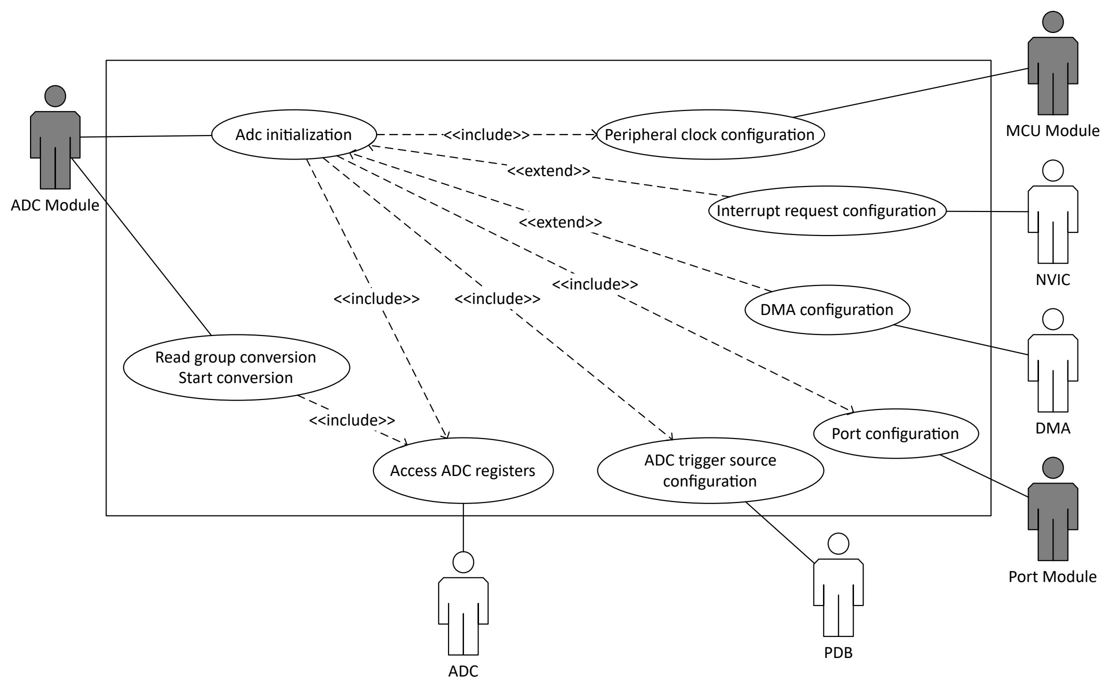


Figure 4.15: ADC module use cases diagram

The sequence diagrams specify the transactions between the modules identified on the use case diagrams. Such diagram joins both the class diagram specification and use case interactions to further clarify how to use the interfaces specified on the class diagram. A simplified example sequence diagram for the ADC module is presented in figure 4.16. The simplified diagram shows only the interactions with

the ADC peripheral and the PDB peripheral, although one can see from the use case diagram that several other interactions occur when a MCAL interface is used. The remaining sequence diagrams are presented on appendix C.3.3.

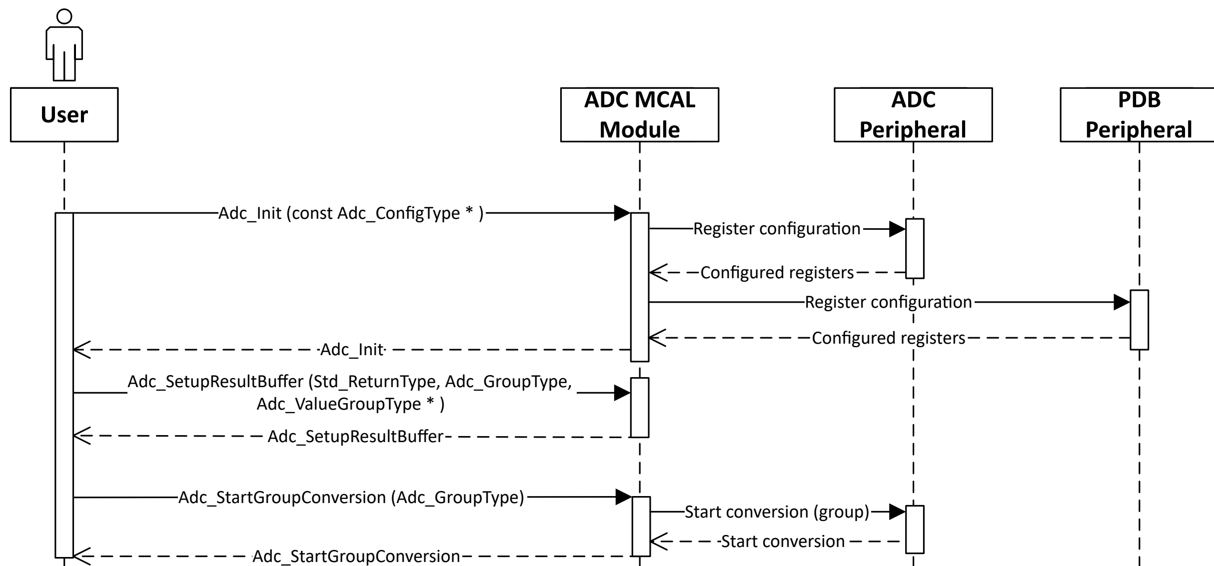


Figure 4.16: ADC module sequence diagram

SAS Application

Implementation of the application layer closely followed the state machine previously described, as the implementation on bare-metal eases a direct transposition from state machine to software. During the implementation, care was taken when mapping the behaviour and actions into MCAL functions, in order to correctly interface the application with the hardware. All the actions mentioned on the state diagrams, such as "initialize microcontroller ADC..." and "start timer", were mapped into MCAL functions such as *ADC_Init* and *Gpt_StartTimer*. Since there are no middle layers between the application and the MCAL, the application is responsible to create the necessary structures to configure all the MCAL modules and call the necessary interfaces to manipulate the hardware. An example of the interface between the application and the MCAL, is presented in figure 4.17 which regards the ADC initialization on the Reset state of the application. In the sequence diagram it is possible to see the initialization of the ADC peripheral using the MCAL interfaces, *Adc_Init* and *Adc_SetupResultBuffer*.

Since the functionalities are already presented on the previous section, the particular implementation will not be explained in-depth. For a more detailed overview of the implementation, each state is presented on the flowcharts include in appendix C.4.

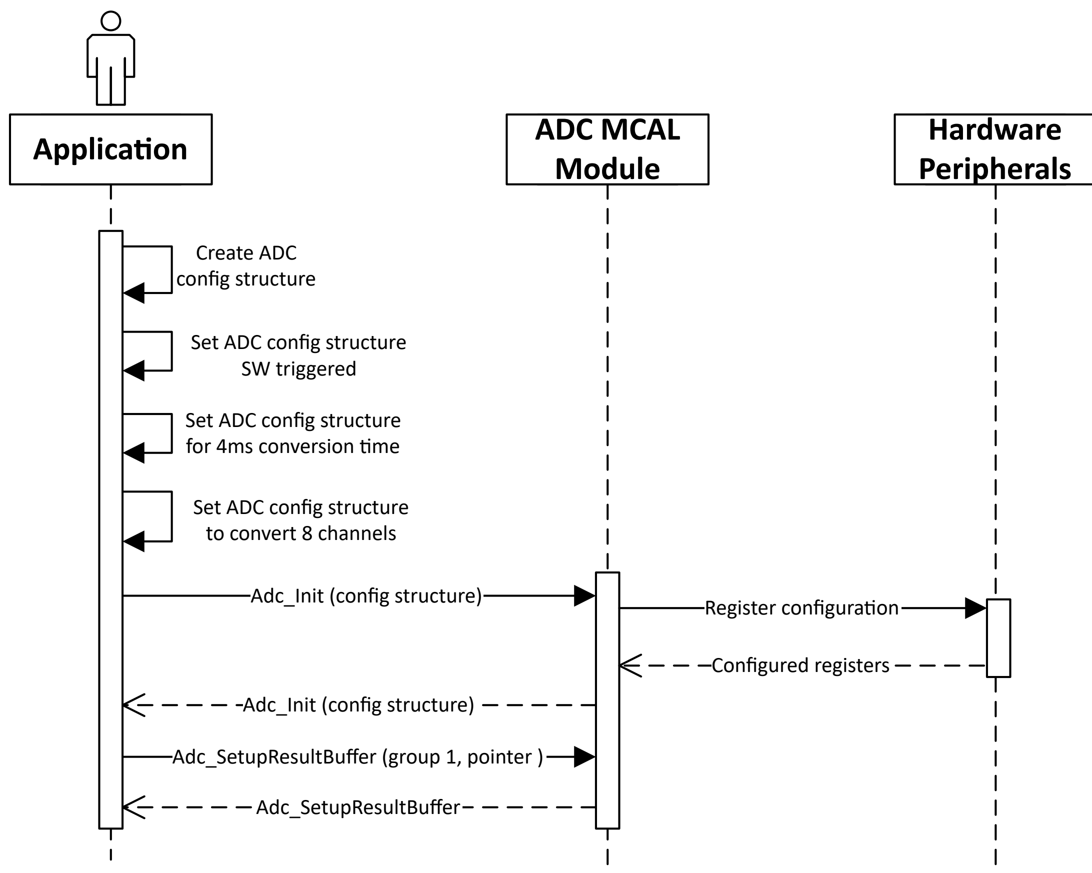


Figure 4.17: Example usage of the MCAL ADC module during Reset state

4.2 Application simulation using QEMU

With software development finished, the next step was to simulate and validate the software, in order to find any implementation errors or possible improvements by design reiterations. For this purpose, the SAS platform was created as a QEMU machine, emulating its real behaviour and allowing for reliability estimations, as will be shown later on the chapter.

4.2.1 Target machine on QEMU

In order to emulate the SAS platform on QEMU, a new machine that reflects the platform's behaviour was created from scratch and added to QEMU. Before any development was made, a gathering of the needed peripherals and hardware modules was made, which resulted on the list presented in figure 4.18. The list contains the peripherals divided by modules and the addresses that will be used to map each in machine memory space.

Uart

Peripheral	Address
LPUART0	40069000
LPUART1	4006A000

Adc

Peripheral	Address
ADC0	4003B000
PDB	40036000

General Purpose Timers

Peripheral	Address
LPIT0	40037000
FTM0	40038000
FTM1	40039000

Mcu

Peripheral	Address
SCG	40064000
PCC	40065000
PMC	4007D000
SMC	4007E000
RCM	4007F000
SIM	40048000

Wdg

Peripheral	Address
WDOG	40052000

Can

Peripheral	Address
FLEXCAN0	40024000

Port

Peripheral	Address
PORTA	40049000
PORTB	4004A000
PORTC	4004B000
PORTD	4004C000
PORTE	4004D000

Gpio

Peripheral	Address
PTA	400FF000
PTB	400FF040
PTC	400FF080
PTD	400FF0C0
PTE	400FF100

Figure 4.18: QEMU S32K116 machine peripherals

Upon gathering all the required *on-chip* peripherals, the machine itself was designed to support all these modules. The diagram in figure 4.19 presents a small section of the class diagram that specifies both

the machine and all the peripherals contained on it. The machine matches the target platform, ARMv7, and contains the attached peripherals as objects. The peripherals contain the respective hardware registers and the needed functions to emulate their behaviour. The full class diagram for the S32K116 machine is presented on appendix D.2. One thing to notice is that the communication peripherals LPUART, CAN and GPIO were extended to interact with the Shared Bus, thus they depend on the interface specified on the last chapter.

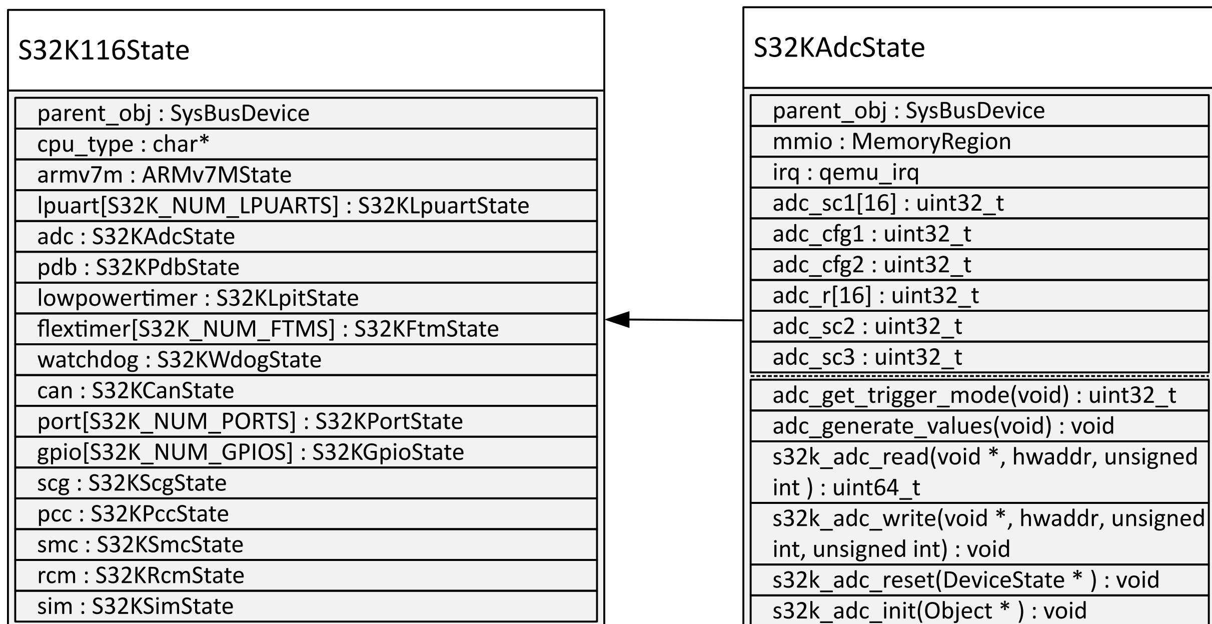


Figure 4.19: S32K116 machine and peripherals class diagram example

Machine development started by implementing the SoC of the target machine without any peripheral. This includes setting the correct processor architecture, which in this case was the ARMv7, create the memory map and configure the IRQ vector. The snippet below shows an extract of the memory mapping and processor assignment for the target machine.

```
static void s32k116_soc_realize(DeviceState *dev_soc, Error **errp){
...
MemoryRegion *system_memory = get_system_memory();
MemoryRegion *sram = g_new(MemoryRegion, 1);
MemoryRegion *flash = g_new(MemoryRegion, 1);
memory_region_init_ram(flash, NULL, "S32K116.flash", FLASH_SIZE, &
    error_fatal);
memory_region_add_subregion(system_memory, FLASH_BASE_ADDRESS, flash);
memory_region_init_ram(sram, NULL, "S32K116.sram", SRAM_SIZE, &
    error_fatal);
```

```
9        memory_region_add_subregion(system_memory, SRAM_BASE_ADDRESS, sram);
10       armv7m = DEVICE(&s->armv7m);
11       qdev_prop_set_uint32(armv7m, "num-irq", 32);
12       ...
```

Listing 4.1: QEMU machine processor configuration snippet

After creating the machine with basic configurations, peripherals were added to it. For every peripheral described on the previously mentioned class diagram, at least one source file and an header file were developed. The header file contains not only the interfaces but also the memory offsets and sizes of the peripheral registers. The source file contains *reset*, *write* and *read* functions that emulate the behaviour when such operations occur, other QEMU specified functions and behaviour help functions. The snippet below provides an example of a *read* function of the developed ADC peripheral.

```
1  static uint64_t s32k_adc_read(void *opaque, hwaddr addr, unsigned int size)
       {
2      S32KAdcState *s = opaque;
3      ...
4      else if(addr >= ADC_R_OFFSET && addr < ADC_R_END){
5          uint32_t index = ((addr - ADC_R_OFFSET) / ADC_REG_OFFSET);
6          s->adc_sc1[index] &= ~ADC_SC1_COCO_MASK;
7          if((s->adc_sc1[index] & ADC_SC1_AIEN_MASK) != 0){
8              qemu_set_irq(s->irq, 0);
9          }
10         return s->adc_r[index];
11     }
12     else{
13         switch (addr) {
14         case ADC_CFG1_OFFSET:
15             return s->adc_cfg1;
16         case ADC_CFG2_OFFSET:
17     ...
18 }
```

These functions are assigned as callbacks in the *MemoryRegionOps* structure which is registered in the correct memory regions on the module initialization (*init* function). This initialization function, which all modules contain, assigns the correct interrupt request to the peripheral, registers the peripheral into the

correct memory region and sets its memory size. The snippet below shows an example of an initialization function. To show all functions put together, the full version of a peripheral is included in appendix D.1.

```
static const MemoryRegionOps s32k_adc_ops = {
    .read = s32k_adc_read,
    .write = s32k_adc_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
};
static void s32k_adc_init(Object *obj)
{
    S32KAdcState *s = S32K_ADC(obj);
    current_device = s;
    sysbus_init_irq(SYS_BUS_DEVICE(obj), &s->irq);
    memory_region_init_io(&s->mmio, obj, &s32k_adc_ops, s, TYPE_S32K_ADC, 0x208);
    sysbus_init_mmio(SYS_BUS_DEVICE(obj), &s->mmio);
}
```

After developing the required peripherals, they were attached to the machine. The attachment procedure is presented on the snippet below. It starts by specifying the bus device that will be added, which in this case is the ADC. Then, the device is registered on the correct memory address. During this step, the memory assignment of the machine has already happened, meaning that MMIO devices can be safely mapped. After registering it on memory, the peripheral's interrupt request is assigned to the correct IRQ table vector position and the device is attached to the QEMU system bus.

```
    ...
    /* Attach ADC */
    dev = DEVICE(&(s->adc));
    object_property_set_bool(OBJECT(&s->adc), true, "realized", &err);
    if (err != NULL){
        error_propagate(errp, err);
        return;
    }
    busdev = SYS_BUS_DEVICE(dev);
    sysbus_mmio_map(busdev, 0, 0x4003b000);
    sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, 28));
    ...
```

```
13      sysbus_init_child_obj(obj, "adc", &s->adc, sizeof(s->adc),
      TYPE_S32K_ADC);
14      ...
```

The machine with all the peripherals was *wrapped* in a development board, which allowed to add *on-board* devices. This was needed since the SAS's architecture presented an external watchdog. The emulation of *on-board* devices does not follow the same development layout as previously shown, as these devices are not attached to the QEMU bus nor are registered in the machine's memory space. As such, the external watchdog was developed to use QEMU's native deadline timers as a way to emulate real watchdog timed behaviour.

Upon finishing development, the platform was ready to be used on QEMU. For this purpose, a new entry was added to the native supported machines, in order to allow the developed machine to be used. The usage of the machine is shown on the following snippet:

```
$ qemu-system-arm [flags] -machine s32k116evb
```

To add all the peripherals and machines to QEMU, the internal Makefiles needed to be extended in order to compile the new modules. Some of of the changes made are shown on appendix D.5.

4.2.2 Validation of the SAS application

With the completed target machine and the simulation extensions developed on the previous chapter, the SAS application could be correctly emulated. The emulation environment combined two QEMU instances, one for each redundant module, two Shared Bus instances, for CAN data output and GPIO for communication between redundant modules, and one synchronization process, as seen in figure 4.20.

All these entities executed on different terminals under a *Linux* environment, as shown in figure 4.21. On the right side of the figure, the two terminals are the QEMU running instances that emulate each one of the redundant subsystems. The terminals output the monitor variable CurrentState that was chosen on the command line argument. The printed values '5' and '7' correspond to the internal state of the application which, on this context, meant that the program is jumping from Angle Sampling to Redundant Module Check every 10 milliseconds.

On the bottom left side, the synchronization process shows the time increments of both instances. This value matches the user chosen value of 50016 nanoseconds for the time budget, which for the case study is very conservative since the maximum tested time budget was 500000 nanoseconds. On the top

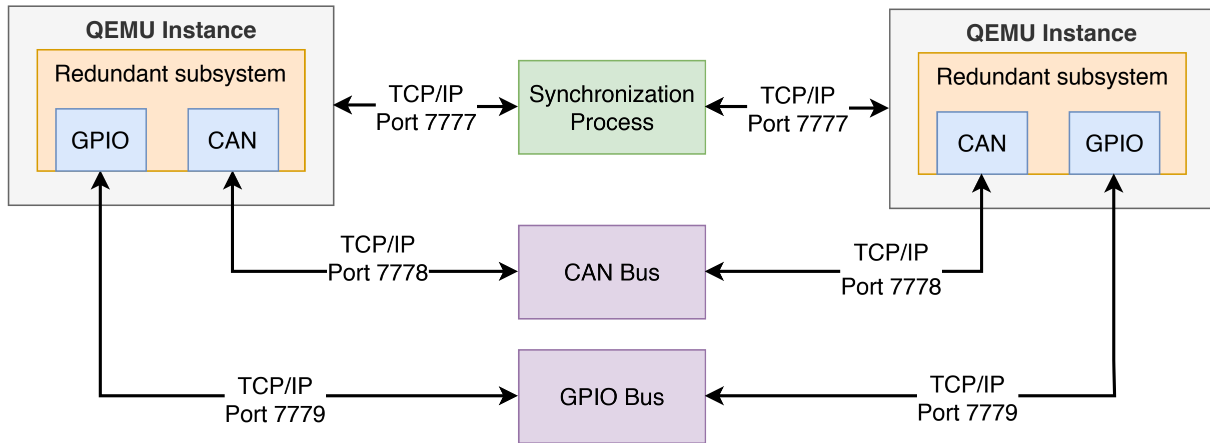


Figure 4.20: Co-simulation environment diagram

left, the CAN bus shows messages that are sent from both redundant subsystems, alongside with the timestamp at which they were sent. As seen in the figure, the messages are sent every 10 milliseconds, validating the initial system requirement. The process that handles the GPIO bus is not shown since it follows similar layout of the CAN bus.

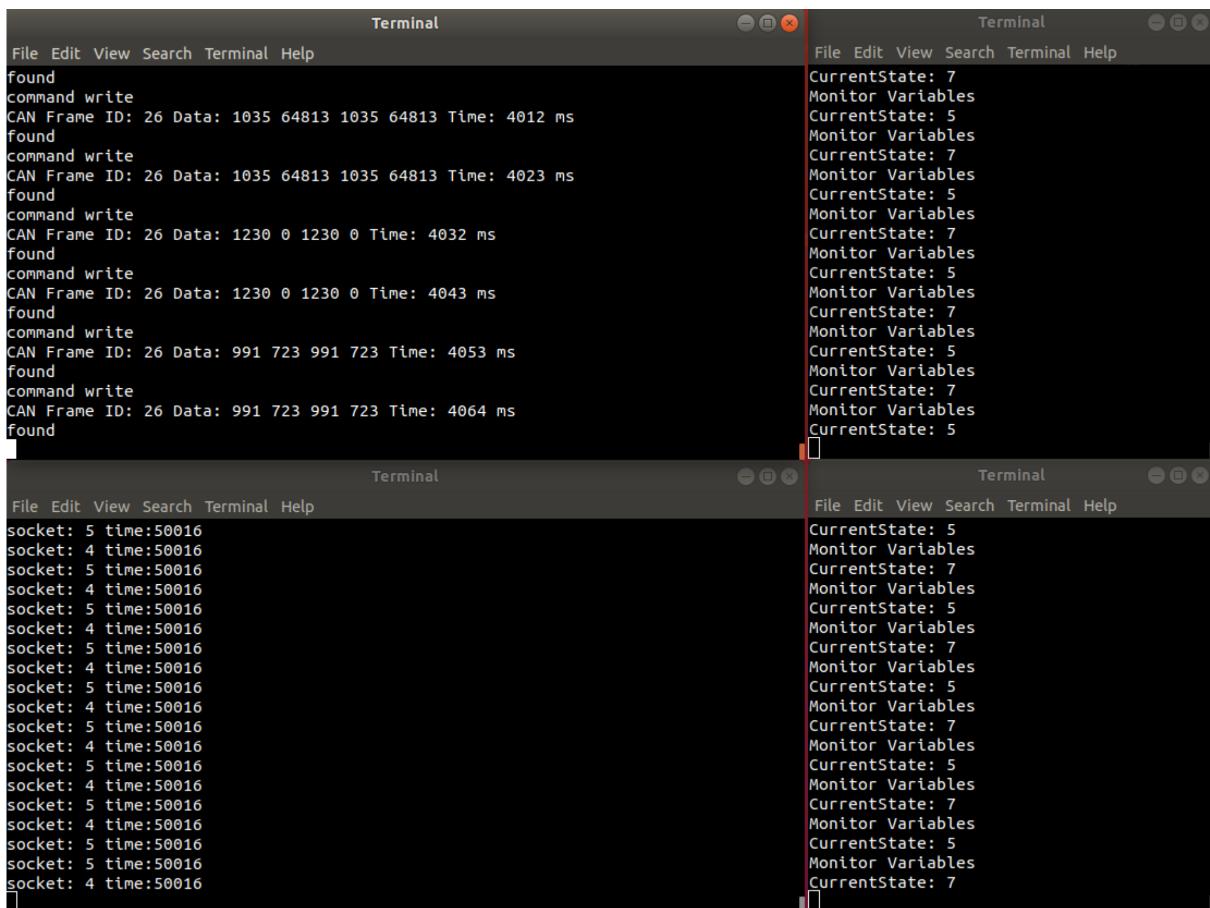


Figure 4.21: Co-simulation environment running the SAS application (1)

As one of the redundant module fails, which in this case was done by shuting down one of the instances, the system correctly enters Fail Degraded operation, as seen in figure 4.22. The internal state maintains a value of '10', meaning Fail Degraded state, and the messages are still sent every 10 milliseconds, as seen on the CAN bus output. At the moment of redundant module failure, the synchronization process disconnects from the module that was shut down since it is not needed for further synchronization.

Figure 4.22: Co-simulation environment running the SAS application (2)

The test regarding redundant module failure was destructive, meaning that, upon failure, the redundant module did not boot again. To validate the system on non-destructive failures, such as a microcontroller reset or clock drifts, the power and clock faults from the Fault Injection extension were used to mimic such behaviours. Upon injecting a power fault, the module that was powered down correctly entered a permanent Reset state and was constantly being reset by the external watchdog, while the other one entered Fail Degraded operation. Upon injecting a clock drift fault, both redundant modules entered Fail Degraded operation, sending messages every 10 milliseconds. Although it does not go against the

initial requirements, this behaviour is an issue to consider on possible design reiterations. To possibly mitigate this issue, the redundant modules should adopt a more complex protocol when it comes to deciding about the operating states.

4.3 Reliability Estimation

Despite the fact that there were no required reliability metrics to achieve, the system still went through a reliability analysis as a proof of concept of simulation-based estimations using the work developed on the previous chapter. For this purpose, the system went through Monte Carlo simulations to get an estimation of its reliability metrics. During each simulation, faults were injected into system components, according to their failure distributions, emulating failures on: system power, microcontroller, CAN communication, clock integrity, isolated communication channel and sensing elements. The faults were generated for each redundant subsystem independently, so they can occur on one redundant subsystem and not on the other. The diagram in figure 4.23 shows the possible faults mapped into system blocks.

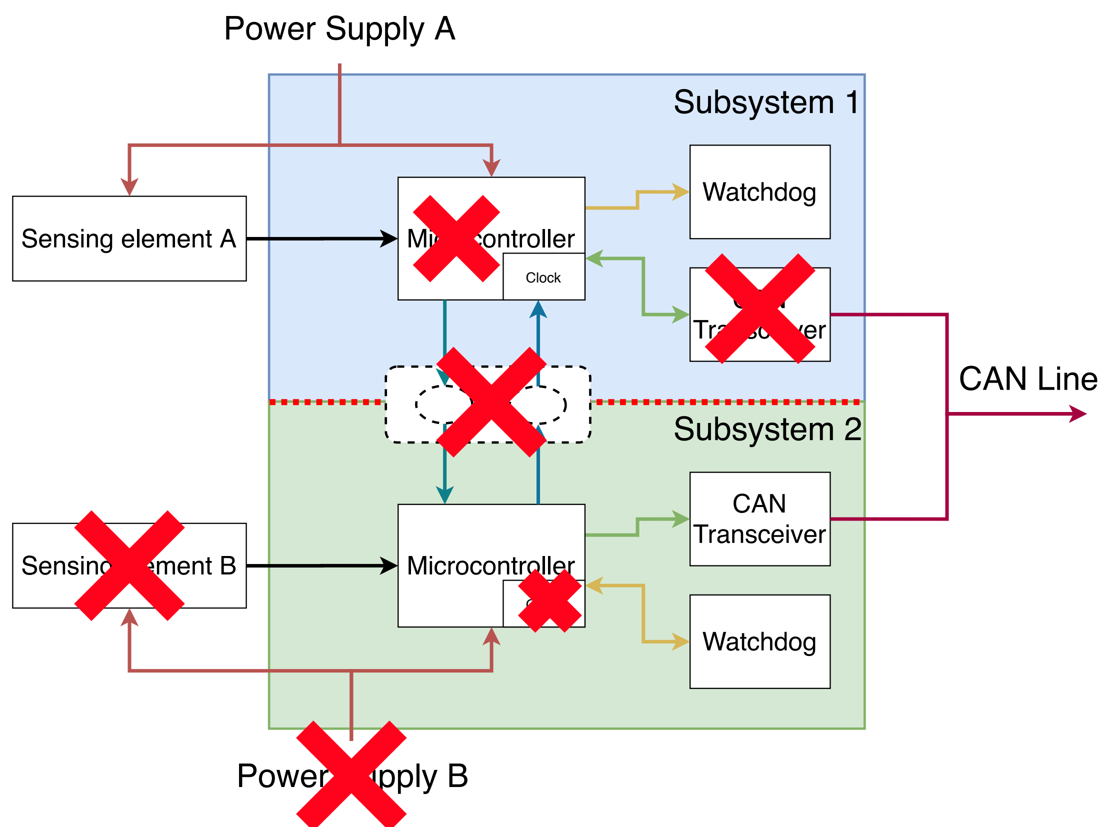


Figure 4.23: Subsystem blocks susceptible to faults

Each simulation trial aimed to execute until system failure or up until a total time of 1500000 hours of component operating time. Since there was not enough data to get an accurate probability of failure distribution of each component, failure rate probability curves were created according to the MTBF values of the components. The curves were parameterized to fit a Weibull distribution, alongside with a cumulative failure rate of 1×10^{-4} in the first 100 hours. Figure 4.24 shows the resulting probability density and cumulative density curves when the component mean time before failure is 2.5×10^5 hours. The MTBF values used for each system block are presented in table E.1.

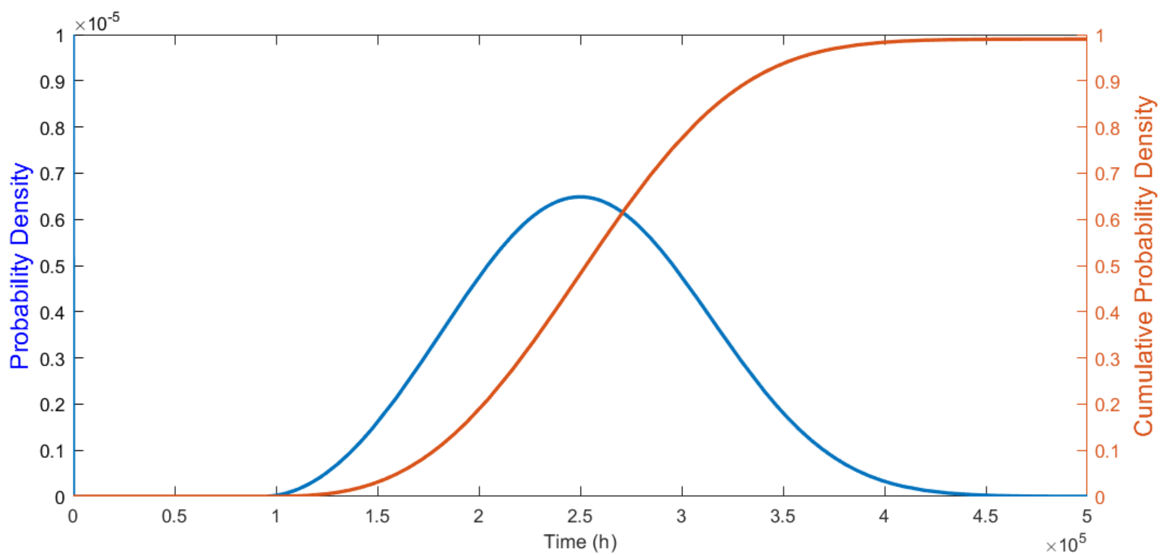


Figure 4.24: Parameterized failure probability density curves

Regarding the simulations, each Monte Carlo step was considered to represent 10 hours of component lifetime. For this purpose, the previously generated curves were integrated into periods of 10 hours, dictating the correct failure probability at each simulation step.

4.3.1 Fault Modeling

To induce failure on different system blocks, different faults were modelled. The possible faults were specified as following:

- Microcontroller - Inject a CPU fault and overwrite the current translated instruction to value 0xE1A0. This new instruction, on ARM Thumb architecture, is an absolute jump instruction to an address out of application range. This renders the microcontroller unable to execute further instructions.
- Power supply - Inject the power specific fault to reset the simulation.

- Clock integrity - Inject the clock specific fault, dropping synchronization between redundant modules.
- CAN communication - Block access to the CAN peripheral, by injecting the peripheral access fault on address 0x40024000.
- Communication Channel - Block access to the GPIO C peripheral which is connected to the communication channel, by injecting the peripheral access fault on address 0x4004B000.
- Sensing Elements - Use the memory fault to overwrite the memory of the ADC sampling result registers with randomly generated values. The target address ranges from 0x4003B048 to 0x4003B068.

4.3.2 Simulation Environment Wrapper

The simulation environment used for fault injection used the same entities as the one used to validate the SAS, alongside the Fault Injection Coordinator, implemented as a Python script, controlling simulation trials. The resulting environment is shown in figure 4.25.

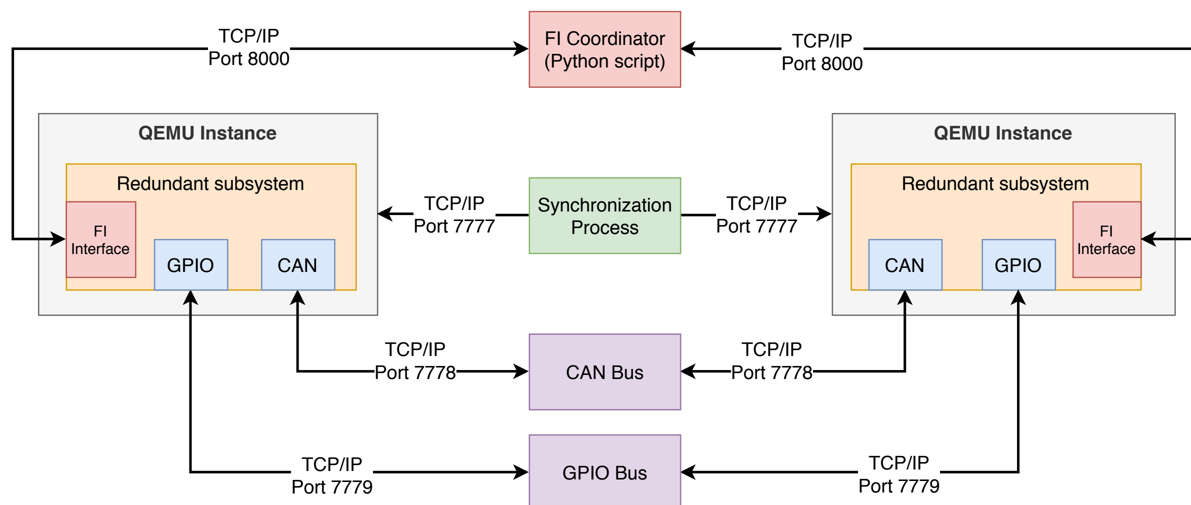


Figure 4.25: Co-simulation environment used for simulations

The script used was an extended version of the Fault Injection Coordinator flowchart that was presented on the previous chapter. At startup, the script retrieved all fault distributions for each component from the parameterized curves. The Fault Library was also created, containing the needed faults for each component.

To ease system behaviour analysis during the simulations, it was assumed that the subsystems operated in three possible states: state 'OK', in which both redundant modules correctly sent the angle

information every 20 milliseconds, with a 10 milliseconds offset; state 'FD', in which the subsystem is in fail degraded operation, and state 'KO', meaning that the subsystem fails to send any angle message. This latter state is assumed to be a system failure state.

During simulation, faults were injected until system failure, which was known by checking the status of both simulations from the monitored variables. If both simulations were 'KO', the current simulation trial is over and the final emulation time was saved, as it was considered system failure. When this happened, a new trial started by restarting simulations. If the system was still operating normally or in fail-degraded, the trial continued by generating new faults until system failure. New faults were created by generating random values during runtime and checking them against the fault probabilities. The resulting simulation states, simulation time and the injected faults were all logged into a text file for later processing. The script flowchart is presented on appendix E.2. Regarding simulation speed, each 10 hour increment of component lifetime (or Monte Carlo step) during the trials took approximately 75ms to complete.

4.4 Simulation Results

The simulations resulted in 181 trials, 10 of which reached the maximum simulation time of 1500000 hours of component lifetime, with the remaining ones resulting in system failure. The resulting distribution of the times before failure from the trials are presented in figure 4.26. From the collected data, the resulting system mean time to failure value is greater than **378797 hours**.

The cumulative distribution function of the resulting data is presented in figure 4.27. This function is equivalent to the unreliability function.

Although the system presented such time before failure, certain faults caused the system to output wrong data while still being in an operational state. This behaviour was caused by the occurrence of sensor faults. In order to get an overview about the time the system outputs wrong data while being operational, the distribution in figure 4.28 shows the data relative to such behaviour. The resulting mean time before wrong data output is **194361 hours**.

Since the amount of simulation data was low comparing with the original plan of 10000 trials, ensemble methods were applied to try to gather a better approximation of the uncertainty of the data. First, the bagging method was applied to the original time before failure data. From the original data, 10000 sets of 20 samples were bootstrapped, resulting on data with a mean value of **379148 hours** and a standard deviation of ± 36302, as seen in figure 4.29.

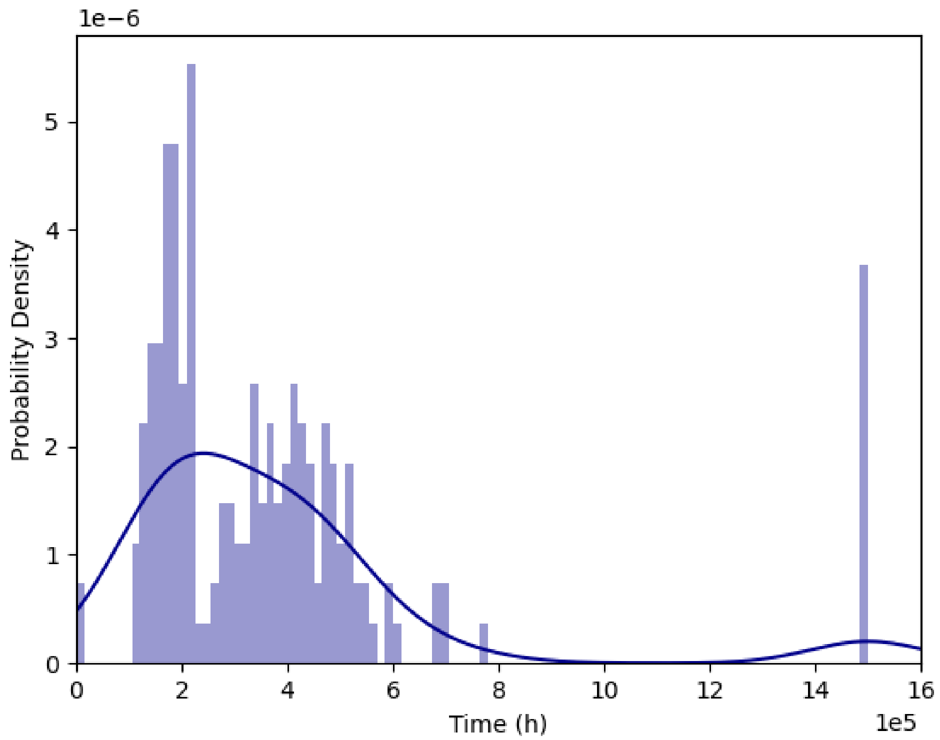


Figure 4.26: Histogram and probability density function of the simulation results

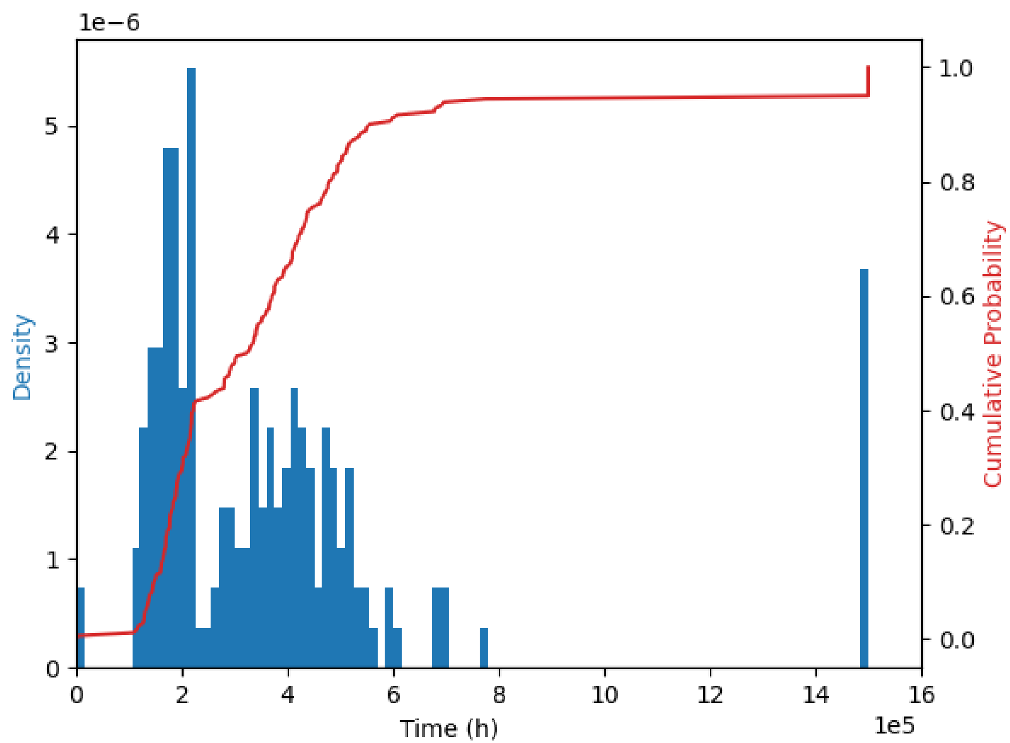


Figure 4.27: Histogram and cumulative distribution function of the simulation results

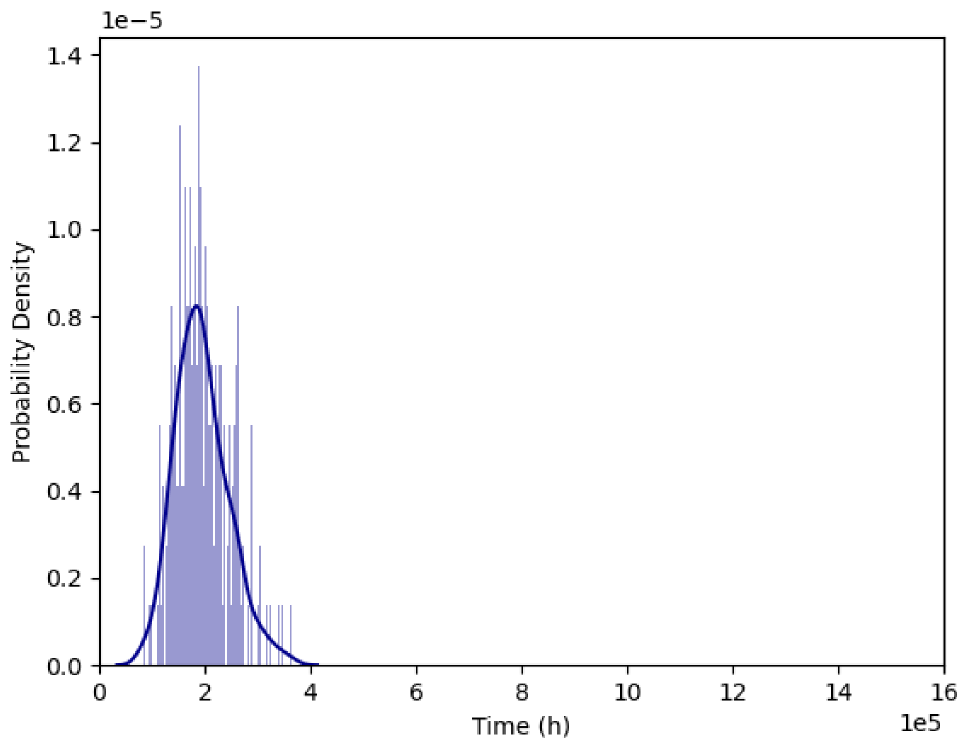


Figure 4.28: Distribution of the probability of wrong data output by the system

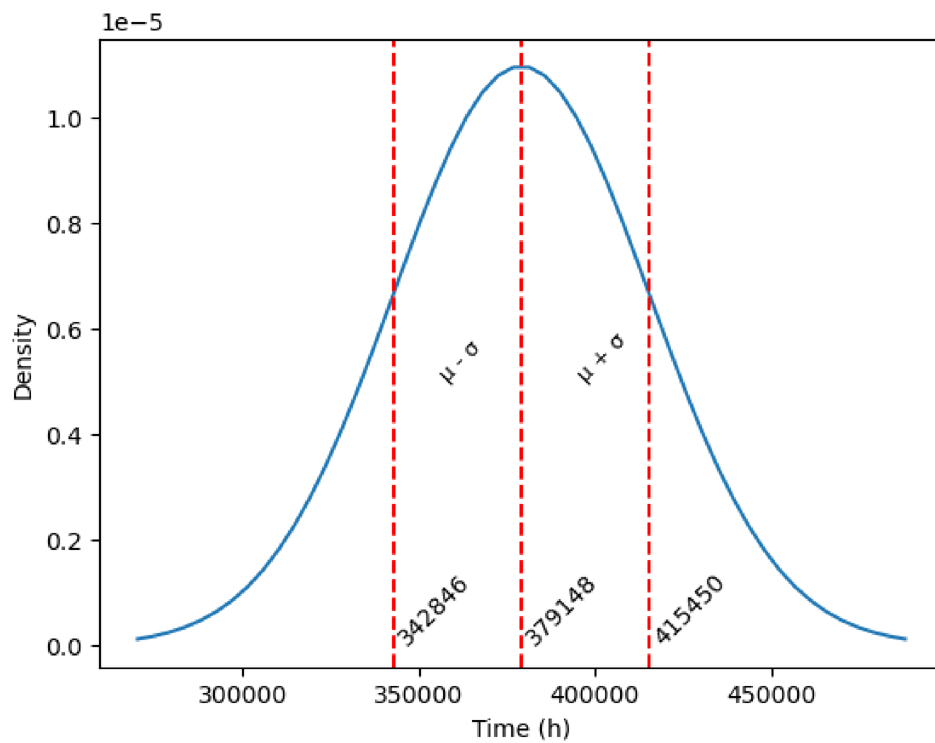


Figure 4.29: Distribution of the mean time to failure after applying bagging method

On the resulting data from bagging, a boosting method was applied. From the bootstrapped sets, each one was assumed to be a different estimation of the mean time to failure. The boosting process followed the algorithm presented on section 2.4.1. The hypothesis was assumed to be equal to the population mean (378797 hours). Consequently, the comparison between the hypothesis and the estimation was expressed into an error calculation where $I(y_i \neq G_m(x_i)) = \frac{x_i - mean}{mean}$. The algorithm ran for 10 iterations and the results of both the weighted mean of each estimation and the alpha weights are presented on table 4.1.

Table 4.1: Results of each iteration of the Boosting method

Iteration	Alpha weight	Mean
1	0.272	357076.826
2	0.109	349373.807
3	0.096	343060.015
4	0.088	337670.933
5	0.082	332963.504
6	0.077	328786.569
7	0.073	325036.849
8	0.070	321639.555
9	0.068	318538.394
10	0.065	315689.836

After all the iterations, each estimation was weight averaged, which resulted on the time before failure distribution in figure 4.30. The distribution presented a mean value of **338910 hours** and a standard deviation of $\pm$14566 hours. Comparing the original data with the Boosting results, it can be seen that the difference between means is around 10%.

Under the same context, the data regarding the times before wrong system output went through the bagging and boosting process. First, bagging was applied, generating 10000 sets of 30 samples resulting on data with a mean value of **194392 hours** with a standard deviation of $\pm$9804 hours. Then, boosting was applied, following the same algorithm and number of iterations used on the time before failure data. The process resulted on a data distribution with a mean value of **184991 hours** with a standard deviation of 4149 hours, as shown in figure 4.31. Comparing with the original data, the difference between means is around 4.8%.

Regarding fault occurrence, the histogram in figure 4.32 shows the amount of faults occurred before system failure. The data shown on the histogram disregards the operational state of the redundant subsystems, meaning that, even if a subsystem was already on a failure state, the occurred faults were still considered. Since there were two redundant subsystems, the maximum possible amount of faults was 12, since 6 types of system block faults could happen in each module. Due to the fact that the system has

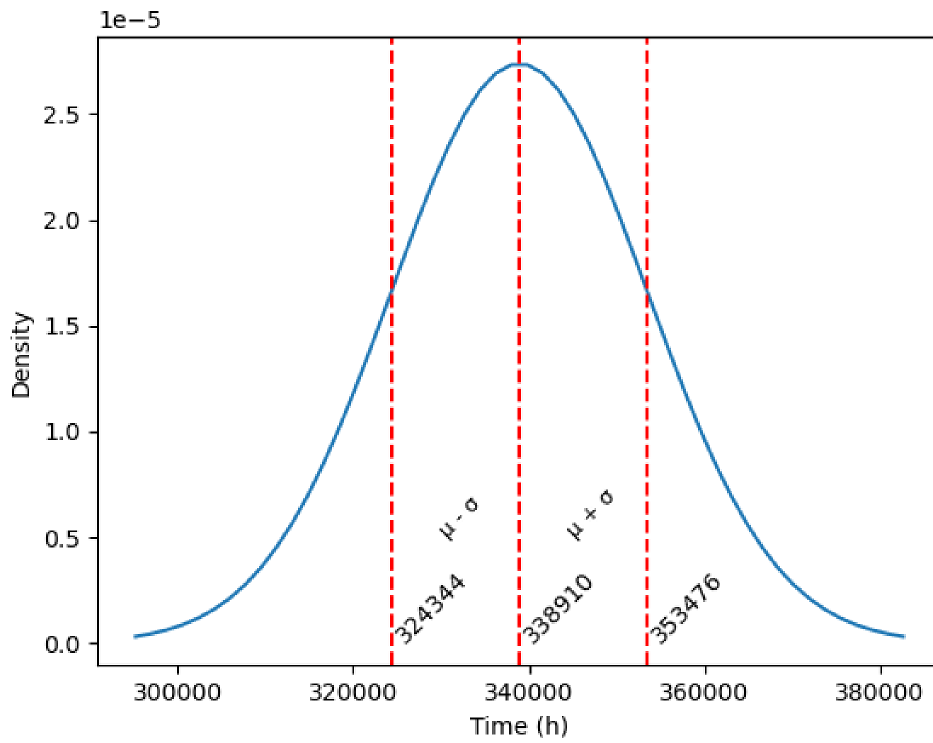


Figure 4.30: Distribution of the mean time to failure after applying boosting method

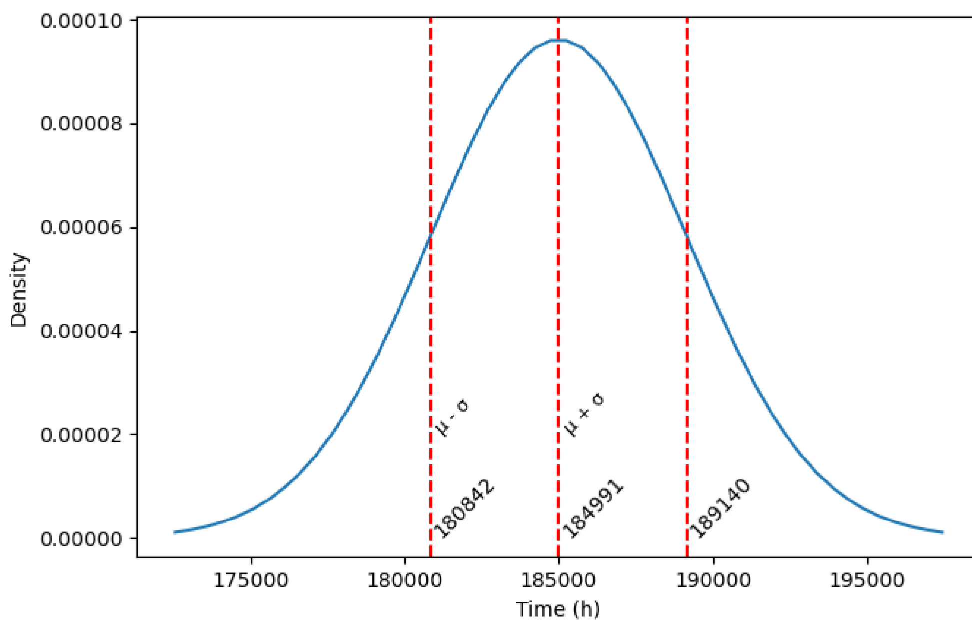


Figure 4.31: Distribution of the time before wrong system output after applying boosting method

components that do not contribute to system failure i.e, faults on such components are not destructive for the system, it tolerated a significant amount of faults before failing.

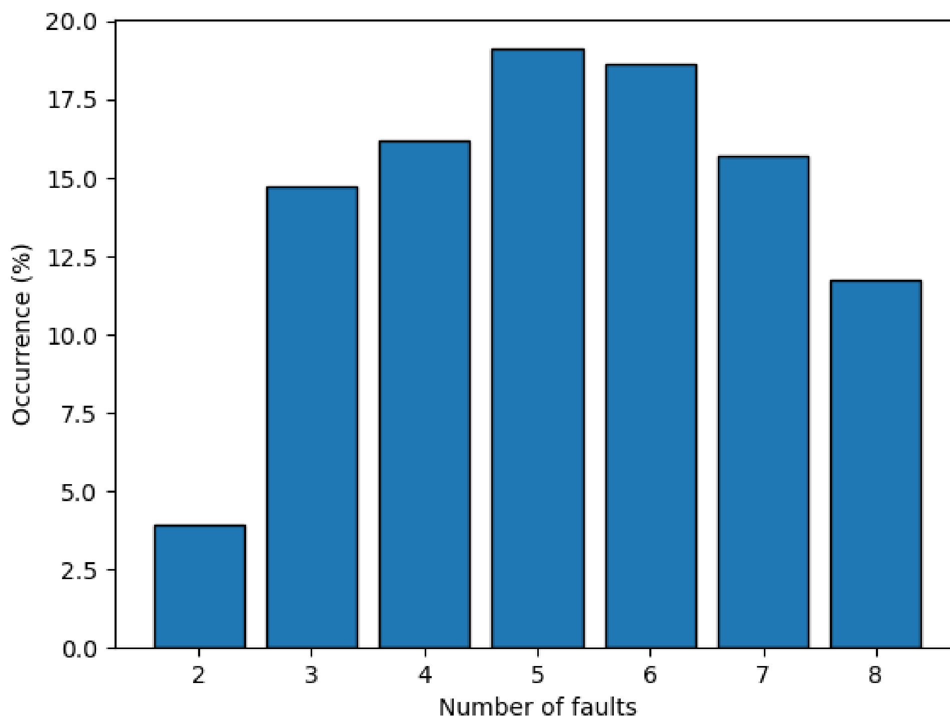


Figure 4.32: Fault occurrence before system failure

4.5 Deployment on the hardware platform

An iteration of the SAS software was deployed on the hardware platform before the simulation-based validation was done. This step was done to allow the accelerated tests to be started as early as possible, supporting another dissertation that was developed at the same time as this one, by another member of the research group. This was an exception made to avoid delaying progression of the mentioned dissertation.

During testing of the deployment, a design bug regarding redundant module synchronization was detected and mitigated on a new version of the design (which is already contemplated on the previously presented system state machine) and on the resulting software. Also, an issue that arises from clock drifts between redundant modules was also observed, which was then confirmed during the simulation. The debugging of these problems took a considerable amount of time and could have been avoided if system simulation was made prior to the deployment.

Chapter 5

Conclusion

By developing this dissertation, different knowledge areas were tackled and skills that will surely transpose into further projects were solidified. Knowledge areas included reliability oriented and embedded system development and design, reliability estimation, full-system emulation through QEMU, adequate software development, *Linux* operating system and its internal mechanisms, network sub-systems and Python scripting.

Regarding the work developed on this dissertation, the simulation extensions assist reliability oriented development, aiming mainly for redundant architectures. Since reliable systems do not have a well defined development flow, the usage of co-simulation environments supported by such extensions aid development from the design phase up until testing. Design iterations and respective software stacks can be validated before any physical prototype is available, reducing the overall development effort and time. The addition of the fault injection capabilities also support early reliability estimations, avoiding possible reiterations later on the development cycle.

The problems that were observed upon deployment of the SAS software on the hardware platform, confirmed that validation through simulation is an advantage to the development cycle. The usage of a solid simulation environment can prevent errors late on the development cycle, reducing time spent on debugging or even avoiding the need for it.

Developed software based on AUTOSAR did not directly affect system reliability by providing fault tolerance mechanisms, but instead gave foundation to the software architecture and helped avoiding typical design bugs. Although it made software design sturdier, the used methodologies are already based on classical software development standards, which are not new and are already widely adopted in numerous other development fields.

In regards to the fault tolerance mechanisms used on the case study, only hardware redundancy was considered, leaving the software to only manage it. Other types of fault tolerance mechanisms could be

implemented on software but such mechanisms were not implemented since, at the time, they were not a priority.

About the simulation results, the system used for the case study presented a mean time to failure of about 338910 hours (39 years). Although the system is operational up until this time, it starts outputting wrong data at a mean time of 184991 hours (21 years). This results from the lack of data sanity checks by the system, leaving the ECU to deal with wrong data outputs.

The usage of the homogeneous redundancy technique on the case-study showed that this technique has limitations when aiming for highly reliable redundant architectures. As system components present the same characteristics, component failures can affect the redundant modules at similar times with the same probability. This means that redundant modules following the same failure distribution would fail at similar times, not taking full advantage of redundancy capabilities.

The project partnership between the University and Bosch provided a better understanding of how projects are tackled in the industry. Such partnership allowed to experience great aspects from both worlds: the knowledge and freedom to experiment from the University, and the product development methodologies and team organization practices from the real industry. These aspects help to pave the way for the entry in the labor market by providing experience on how real industry works.

5.1 Future Work

Concerning possible future work, there are several improvements and additions that can enhance the general performance and usability of the simulation extensions.

Firstly, it is decisive to increase the overall performance of the fault injection mechanisms. This will allow for increased throughput when performing simulation trials. This improvement could be implemented by replacing current fault request mechanism during runtime to an initialization time mechanism, by creating, reading and parsing the fault list during QEMU's initialization.

Secondly, the fault injection capabilities could be extended to allow more types of faults with different characteristics. The step towards this direction could be to port all the remaining fault types of the FIES framework, allowing a wider range of use cases covered by the FI extension.

Thirdly, the synchronization process should be tested with heterogenous architectures. Due to the fact that the case-study only covered a homogeneous architecture, this test is important to validate the usage of this synchronization method on different types of redundant architectures. The test should mostly

focus the internal synchronization mechanism of QEMU, since it is significantly more complex than the synchronization process itself. By validating the usage on both types of architectures, it opens doors for experimentation on different kinds of architectures. Alongside this advantage, this could also allow for a further exploration when it comes to emulating Lockstep processors.

Fourthly, the Shared Bus peripheral interface within QEMU could also be redesigned to use only one thread per Shared Bus instance instead of one per peripheral. This could reduce peripheral overhead and ease of development, although it is not known if this change is trully advantageous. Also, on the current version of the Shared Bus, peripherals always use *localhost* IP address to connect to the instances. To support instances within a different computer or different networks, the IP address should be given on QEMU's argument command line, allowing to have networked communication between peripherals.

Finally, since the extensions developed fit under co-simulation, FMI support could be added to QEMU, adopting the usage of both the Shared Bus and the synchronization extensions under the FMI standard. This would be a significant upgrade on the usability of QEMU with the extensions, since it would allow other domains to easily integrate a co-simulation environment. Such integration could open possibilities for more complex experiments between domains.

Regarding the case-study, besides the imposed requirements by Bosch, there are some improvements and different designs that could be explored. One of the improvements regards the communication protocol used to exchange messages between redundant modules, which is a simple digital feedback. A more complex communication protocol could allow for a better synchronization management and decisions between the redundant modules. Relatively to the system architecture, an heterogenous approach could be explored, by having different types of microcontrollers from different manufacturers and with different characteristics. This would provide better sturdiness when it comes to common mode faults. Finally, as mentioned before, software fault tolerance mechanisms, such as N-Version Programming or Recovery Blocks, were not implemented but, as future work, they could be explored as further improvement to system reliability.

References

[1] Y. Lu, A. A. Miller, R. Hoffmann, and C. W. Johnson, "Towards the automated verification of weibull distributions for system failure rates," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9933 LNCS, no. September, pp. 81–96, 2016.

[2] A. Birolini, *Reliability Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, vol. 34, no. 4. [Online]. Available: http://link.springer.com/10.1007/978-3-662-03792-8

[3] M. Barr and P. O. Reilly, *Programming Embedded Systems in C and C++*. O'Reilly & Associates, Inc. Sebastopol, CA, USA ©1998, 1999.

[4] H. M. Qian and C. Zheng, "A embedded software testing process model," *Proceedings - 2009 International Conference on Computational Intelligence and Software Engineering, CiSE 2009*, 2009.

[5] J. M. Ludden and G. T. Moore, "Principles and practice," *International Journal of Care Pathways*, vol. 2, no. 1, pp. 26–27, 1998.

[6] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 01, no. 1, pp. 11–33, 2004.

[7] B. Randell, "System structure for software fault tolerance," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 437–449, 1975.

[8] E. Dubrova, *Fault-Tolerant Design*. New York, NY: Springer New York, 2013. [Online]. Available: http://link.springer.com/10.1007/978-1-4614-2113-9

[9] V. P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[10] P. Verissimo and L. Rodrigues, *Distributed systems for system architects*, 2001.

[11] E. ZIO, M. FAN, Z. ZENG, and R. KANG, “Application of reliability technologies in civil aviation: Lessons learnt and perspectives,” *Chinese Journal of Aeronautics*, vol. 32, no. 1, pp. 143–158, 2019. [Online]. Available: https://doi.org/10.1016/j.cja.2018.05.014

[12] D. P. Siewiorek, “Architecture of Fault-Tolerant Computers: An Historical Perspective,” *Proceedings of the IEEE*, vol. 79, no. 12, pp. 1710–1734, 1991.

[13] P. D. T. O’Connor and A. Kleyner, *Practical Reliability Engineering*. Chichester, UK: John Wiley & Sons, Ltd, dec 2011, vol. 28, no. 1. [Online]. Available: http://doi.wiley.com/10.1002/9781119961260

[14] ReliaSoft Corporation, “Life Data Analysis Reference,” *Tools to Empower: The Reliability Professional*, pp. 103–154, 2015.

[15] D. of Defense, “MILITARY HANDBOOK Electronic Reliability Design Handbook,” *Mil-Hdbk-338B*, no. 1 October, 1998.

[16] T. I. B□jenescu and M. I. Bâzu, *Reliability of Electronic Components*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. [Online]. Available: http://link.springer.com/10.1007/978-3-642-58505-0

[17] T. Wilfredo, “Software Fault Tolerance: A Tutorial,” 2000.

[18] M. Lyu and X. Cai, “Fault-tolerant software,” in *Wiley Encyclopedia of Computer Science and Engineering*, 2007.

[19] M. Abd-El-Barr, *Design and analysis of reliable and fault-tolerant computer systems*, 2006.

[20] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, “Fault-tolerant platforms for automotive safety-critical applications,” *CASES 2003: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 170–177, 2003.

[21] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*, 1st ed. Elsevier, 2007, no. 1. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780120885251X50007

[22] D. Siewiorek and P. Narasimhan, “Fault-tolerant architectures for space and avionics applications,” *NASA Ames Research*, pp. 1–19, 2005.

[23] P. O. Wieland, “Living together in space: The design and operation of the life support systems on the International Space Station,” *NASA Technical Memorandum*, vol. 1, no. 206956, 1998.

[24] C. Huang, F. Naghdy, H. Du, and H. Huang, “Fault tolerant steer-by-wire systems: An overview,” *Annual Reviews in Control*, vol. 47, pp. 98–111, 2019. [Online]. Available: https://doi.org/10.1016/j.arcontrol.2019.04.001

[25] A. Dasgupta and J. M. Hu, “Hardware reliability,” *Product Reliability, Maintainability, and Supportability Handbook, Second Edition*, pp. 95–140, 2009.

[26] A. Yadav and R. Khan, “Reliability Estimation Framework -Complexity Perspective,” *Computer Science & Information Technology (CS & IT)*, no. December, pp. 45–70, 2012.

[27] J. Pan, “Software Reliability,” 2020-09-05, 1999.

[28] S. Dersten, J. Axelsson, and J. Froberg, “Effect analysis of the introduction of AUTOSAR - A systematic literature review,” *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011*, no. February 2015, pp. 239–246, 2011.

[29] J. Fürnkranz, “Ensemble-Methoden: Lecture Notes,” 2009. [Online]. Available: http://www.ke.tu-darmstadt.de/lehre/archiv/ws0910/mldm

[30] L. Breiman, “Bagging predictors,” *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[31] Y. Freund and R. E. Schapire, “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting,” *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[32] R. Esposito and L. Saitta, “Monte Carlo theory as an explanation of bagging and boosting,” *IJCAI International Joint Conference on Artificial Intelligence*, pp. 499–504, 2003.

[33] H. Ziade, R. Ayoubi, and R. Velazco, “A survey on fault injection techniques,” *International Arab Journal of Information Technology*, vol. 1, no. 2, pp. 171–186, 2004. [Online]. Available: http://www.citemaster.net/get/f18d7caa-7207-11e3-b274-00163e009cc7/04-Hissam.pdf

[34] F. Cerveira, R. Barbosa, M. Mercier, and H. Madeira, “On the Emulation of Vulnerabilities through Software Fault Injection,” *Proceedings - 2017 13th European Dependable Computing Conference, EDCC 2017*, no. 2, pp. 73–78, 2017.

[35] M. Sonza Reorda, L. Sterpone, M. Violante, M. Portela-Garcia, C. Lopez-Ongil, and L. Entrena, "Fault injection-based reliability evaluation of SoPCs," *Proceedings - Eleventh IEEE European Test Symposium, ETS 2006*, vol. 2006, pp. 75–80, 2006.

[36] V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions," *Digest of Papers - 27th Annual International Symposium on Fault-Tolerant Computing, FTCS 1997*, pp. 32–36, 1997.

[37] L. A. Naviner, J. F. Naviner, G. G. Dos Santos, E. C. Marques, and N. M. Paiva, "FIFA: A fault-injection-fault-analysis-based tool for reliability assessment at RTL level," *Microelectronics Reliability*, vol. 51, no. 9-11, pp. 1459–1463, 2011. [Online]. Available: http://dx.doi.org/10.1016/j.microrel.2011.06.017

[38] A. Höller, G. Macher, T. Rauter, J. Iber, and C. Kreiner, "A Virtual Fault Injection Framework for Reliability-Aware Software Development," *Proceedings - 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN-W 2015*, pp. 69–74, 2015.

[39] W. M. Zabołotny, "Development of embedded PC and FPGA based systems with virtual hardware," R. S. Romaniuk, Ed., vol. 8454, oct 2012, p. 84540S. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.981877

[40] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz, "Testing embedded software: A survey of the literature," *Information and Software Technology*, vol. 104, no. July, pp. 14–45, 2018.

[41] J. Niatas, "Virtual Hardware ECU – How to Significantly Increase Your Testing Throughput," 2017, pp. 1–30.

[42] J. Engblom, G. Girard, and B. Werner, "Testing Embedded Software using Simulated Hardware," *Embedded Real-Time Software Conference (ERTS 2006), Toulouse, France*, no. January, pp. 1–9, 2006. [Online]. Available: http://www.engbloms.se/publications/engblom-erts2006.pdf

[43] F. Bellard, "QEMU, a fast and portable dynamic translator," *USENIX 2005 Annual Technical Conference*, pp. 41–46, 2005.

[44] N. Naia, "Real-Time Linux and Hardware Accelerated Systems on QEMU," 2015.

[45] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, “Co-simulation: State of the art,” 2017. [Online]. Available: http://arxiv.org/abs/1702.00686

[46] A. Abel, T. Blochwitz, A. Eichberger, P. Hamann, and U. Rein, “Functional Mock-up Interface in Mechatronic Gearshift Simulation for Commercial Vehicles,” *Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany*, vol. 76, pp. 775–780, 2012.

[47] L. Belmon, “Virtual Integration for hybrid powertrain development, using FMI and Modelica models,” *Proceedings of the 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden*, vol. 96, no. March 2014, pp. 419–426, 2014.

[48] A. Ben Khaled, M. Ben Gaid, N. Pernet, and D. Simon, “Fast multi-core co-simulation of Cyber-Physical Systems: Application to internal combustion engines,” *Simulation Modelling Practice and Theory*, vol. 47, pp. 79–91, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1569190X14000665

[49] X. Zhang and J. F. Broenink, “A structuring mechanism for embedded control systems using co-modelling and co-simulation,” in *SIMULTECH 2012*. SciTePress - Science and and Technology Publications, 2012, pp. 131–136.

[50] W. S. Dols, S. J. Emmerich, and B. J. Polidoro, “Coupling the multizone airflow and contaminant transport software CONTAM with EnergyPlus using co-simulation,” *Building Simulation*, vol. 9, no. 4, pp. 469–479, 2016.

[51] T. Nouidui, M. Wetter, and W. Zuo, “Functional mock-up unit for co-simulation import in EnergyPlus,” *Journal of Building Performance Simulation*, vol. 7, no. 3, pp. 192–202, 2014.

[52] R. Saleh, S.-J. Jou, and A. R. Newton, *Mixed-Mode Simulation and Analog Multilevel Simulation*. Boston, MA: Springer US, 1994. [Online]. Available: http://link.springer.com/10.1007/978-1-4757-5854-2

[53] Powersim, “SimCoupler.” [Online]. Available: https://powersimtech.com/products/psim/simcoupler/

[54] E. M. JK Peacock, JW Wong, “Distributed simulation using a network of microcomputers,” in *THIRD BERKELEY WORKSHOP*, Peacock, Ed., 2011, p. 358.

[55] T. Jung and M. Weyrich, “Synchronization of a “Plug-and-Simulate”-capable Co-Simulation of Internet-of-Things-Components,” vol. 00, no. July, pp. 18–20, 2018.

[56] S. Jafer, Q. Liu, and G. Wainer, “Synchronization methods in parallel and distributed discrete-event simulation,” *Simulation Modelling Practice and Theory*, vol. 30, pp. 54–73, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.simpat.2012.08.003

[57] K. Mani Chandy and J. Misra, “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs,” *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, 1979.

[58] R. Ayani and H. Rajaei, “Parallel simulation using conservative time windows,” in *Proceedings of the 24th conference on Winter simulation - WSC ’92*. New York, New York, USA: ACM Press, 1992, pp. 709–717. [Online]. Available: http://portal.acm.org/citation.cfm?doid=167293.167684

[59] D. R. Jefferson, “Virtual Time,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[60] T. Blockwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel, “Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models,” *Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany*, vol. 76, no. July 2015, pp. 173–184, 2012.

[61] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauss, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf, “The Functional Mockup Interface for Tool independent Exchange of Simulation Models,” *Proceedings from the 8th International Modelica Conference, Technical Univeristy, Dresden, Germany*, vol. 63, no. March, pp. 105–114, 2011.

[62] E. Chrisofakis, A. Junghanns, C. Kehrer, and A. Rink, “Simulation-based development of automotive control software with Modelica,” *Proceedings from the 8th International Modelica Conference, Technical Univeristy, Dresden, Germany*, vol. 63, pp. 1–7, 2011.

[63] A. Ben Khaled, M. Ben Gaid, D. Simon, and G. Font, *Multicore simulation of powertrains using weakly synchronized model partitioning*. IFAC, 2012, vol. 45, no. 30. [Online]. Available: http://dx.doi.org/10.3182/20121023-3-FR-4025.00018

[64] S. Gedda, C. Andersson, J. Åkesson, and S. Diehl, "Derivative-free Parameter Optimization of Functional Mock-up Units," *Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany*, vol. 76, pp. 819–828, 2012.

[65] T. Schierz, M. Arnold, and C. Clauss, "Co-simulation with communication step size control in an FMI compatible master algorithm," *Proceedings of the 9th International MODELICA Conference, September 3-5, 2012, Munich, Germany*, vol. 76, no. 2, pp. 205–214, 2012.

[66] V. Chipounov and G. Candea, "Dynamically Translating x86 to LLVM using QEMU," *Technical Report EPFL-TR-149975*, vol. 86, no. March, 2010.

[67] C. May, "QEMU : Architecture and Internals Lecture for the Embedded Systems Course Quick EMUlator (QEMU)," 2014.

[68] S. Chyłek and M. Goliszewski, "QEMU-based fault injection framework," *Studia Informatica*, vol. 33, no. 4, pp. 25–42, 2012.

[69] Y. Li, P. Xu, and H. Wan, "A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing," pp. 123–127, 2013.

[70] M. Becker, D. Baldin, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "XEMU: An efficient QEMU based binary mutation testing framework for embedded software," *EMSOFT'12 - Proceedings of the 10th ACM International Conference on Embedded Software 2012, Co-located with ESWEEK*, pp. 33–42, 2012.

[71] S. T. Shen, S. Y. Lee, and C. H. Chen, "Full system simulation with QEMU: An approach to multi-view 3D GPU design," *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems*, pp. 3877–3880, 2010.

[72] A. Höller, G. Schönfelder, N. Kajtazovic, T. Rauter, and C. Kreiner, "FIES: A fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems," *Proceedings - International Workshop on Microprocessor Test and Verification*, vol. 2015-April, pp. 105–110, 2015.

Appendix A

State of the Art Help Material

A.1 Relationships between Reliability Functions

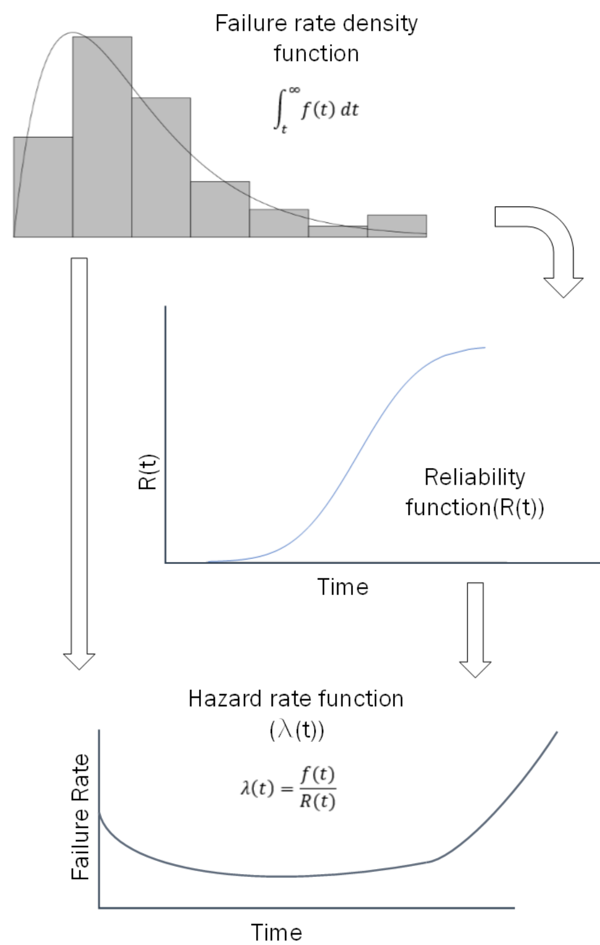


Figure A.1: Relation between the Failure rate density, reliability and hazard rate functions

A.2 AUTOSAR Coding Guidelines Example

Rule M3-3-2 (required, implementation, automated)
If a function has internal linkage then all re-declarations shall include the static storage class specifier.

See MISRA C++ 2008 [6]

Note: Static storage duration class specifier is redundant and does not need to be specified if a function is placed in an unnamed namespace.

6.3.4 Name lookup

Rule M3-4-1 (required, implementation, automated)
An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

See MISRA C++ 2008 [6]

6.3.9 Types

Rule M3-9-1 (required, implementation, automated)
The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

See MISRA C++ 2008 [6]

Rule A3-9-1 (required, implementation, automated)
Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types.

Rationale

The basic numerical types of char, int, short, long are not supposed to be used, specific-length types from <cstdint> header need be used instead.

Fixed width integer types are:

- std::int8_t
- std::int16_t
- std::int32_t
- std::int64_t
- std::uint8_t
- std::uint16_t

A.3 Embedded development Flow

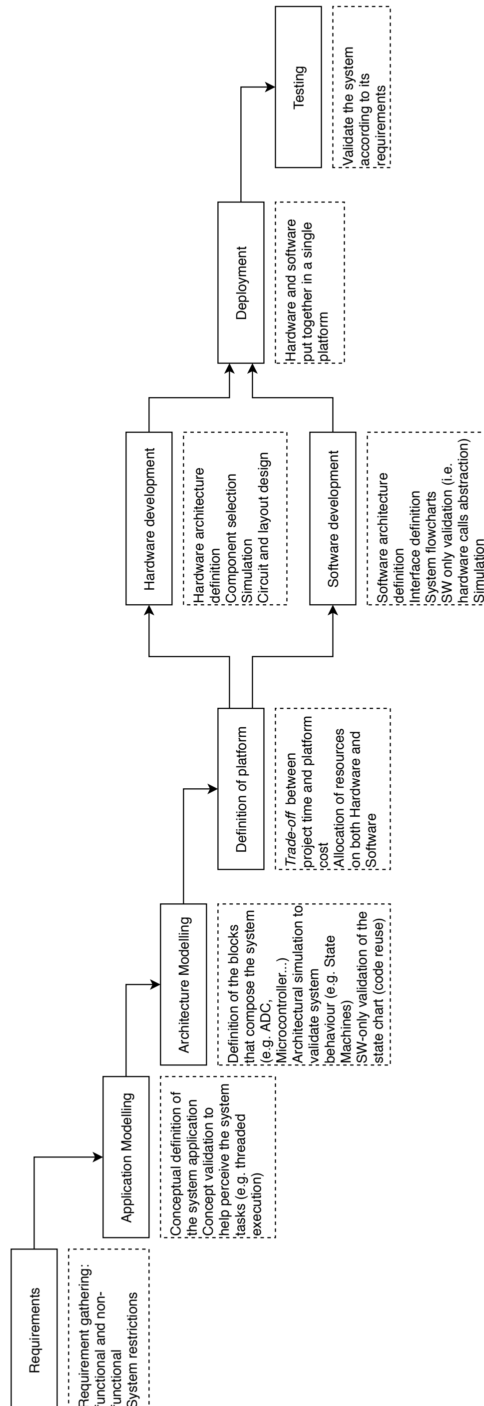


Figure A.2: Embedded systems development flow (detailed)

Appendix B

Developed Simulation Extensions Diagrams

B.1 Shared Bus Diagrams

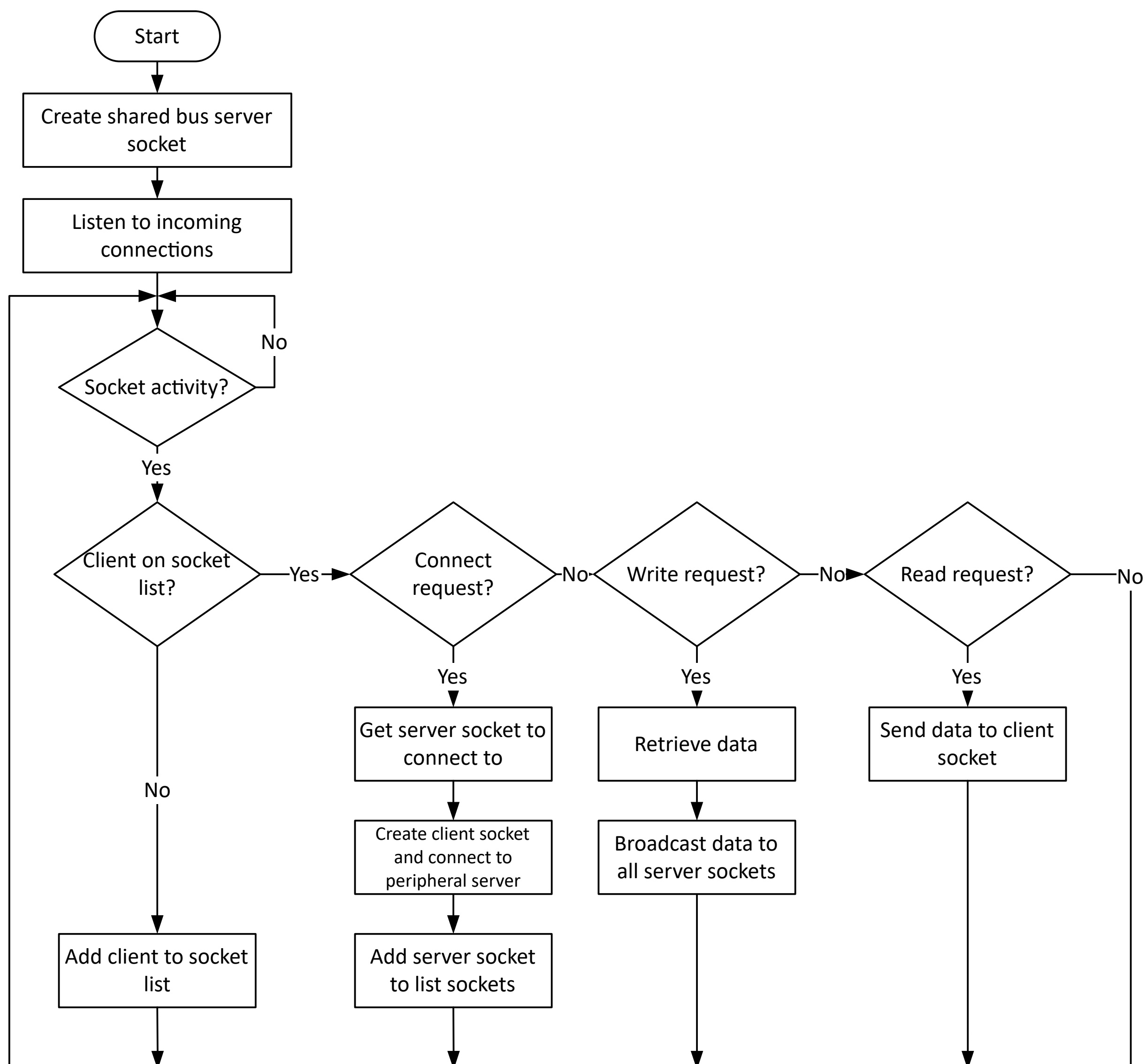


Figure B.1: Shared Bus process flowchart

Table B.1: Communication commands between QEMU and the Shared Bus.

Command	Purpose
SB_ACK	Acknowledge
SB_NACK	No acknowledge
SB_CONNECT	Connect shared bus to peripheral server
SB_WRITE	Write on shared bus
SB_READ	Read from shared bus

B.2 Fault Injection Diagrams

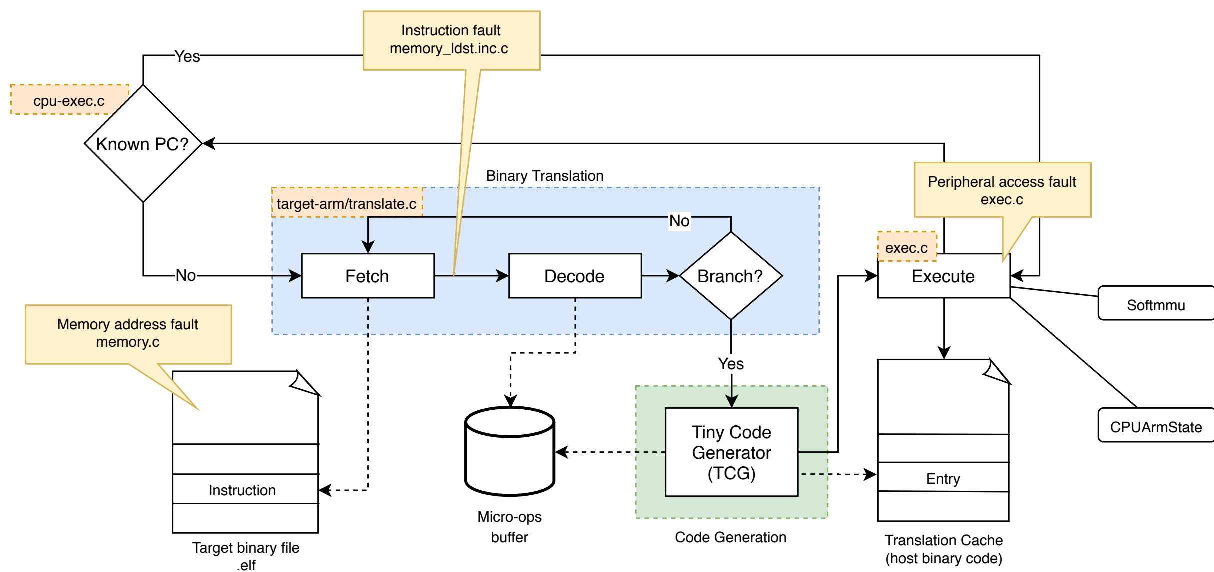


Figure B.2: Change on QEMU for the Fault Injection extension

B.2.1 Fault Injection Functions Flowcharts

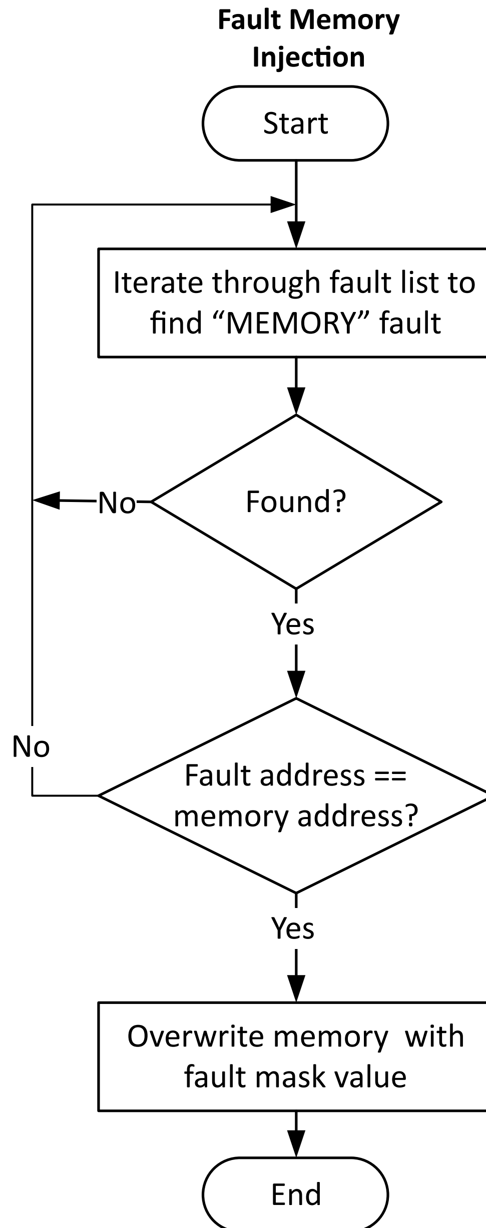


Figure B.3: Memory fault injector function flowchart

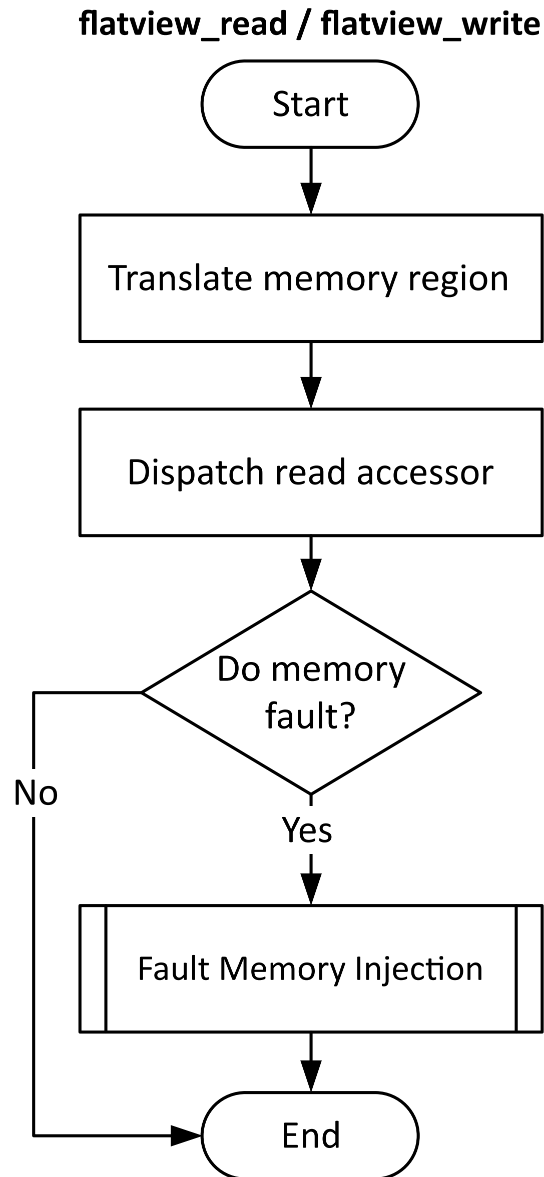


Figure B.4: Usage of the memory fault injection function on flatview_read and flatview_write functions

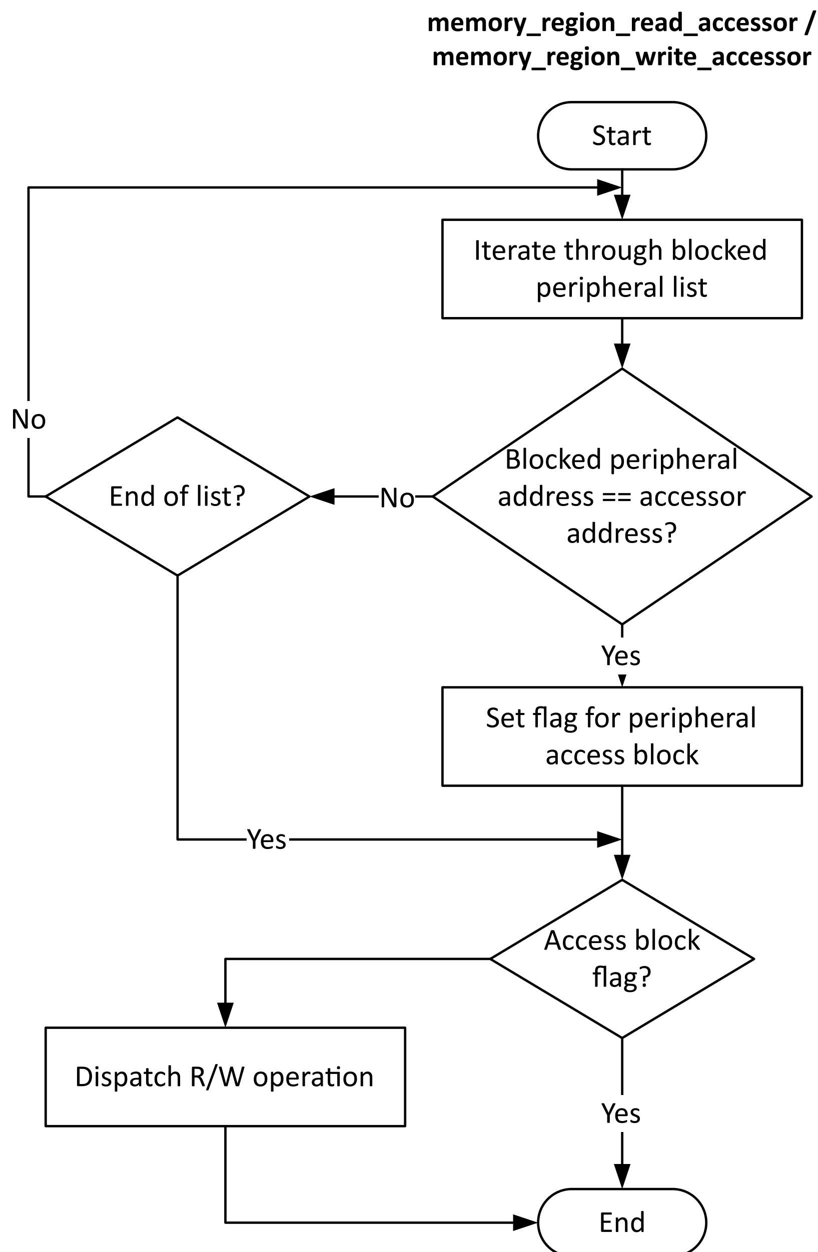


Figure B.5: Peripheral block fault functions flowchart

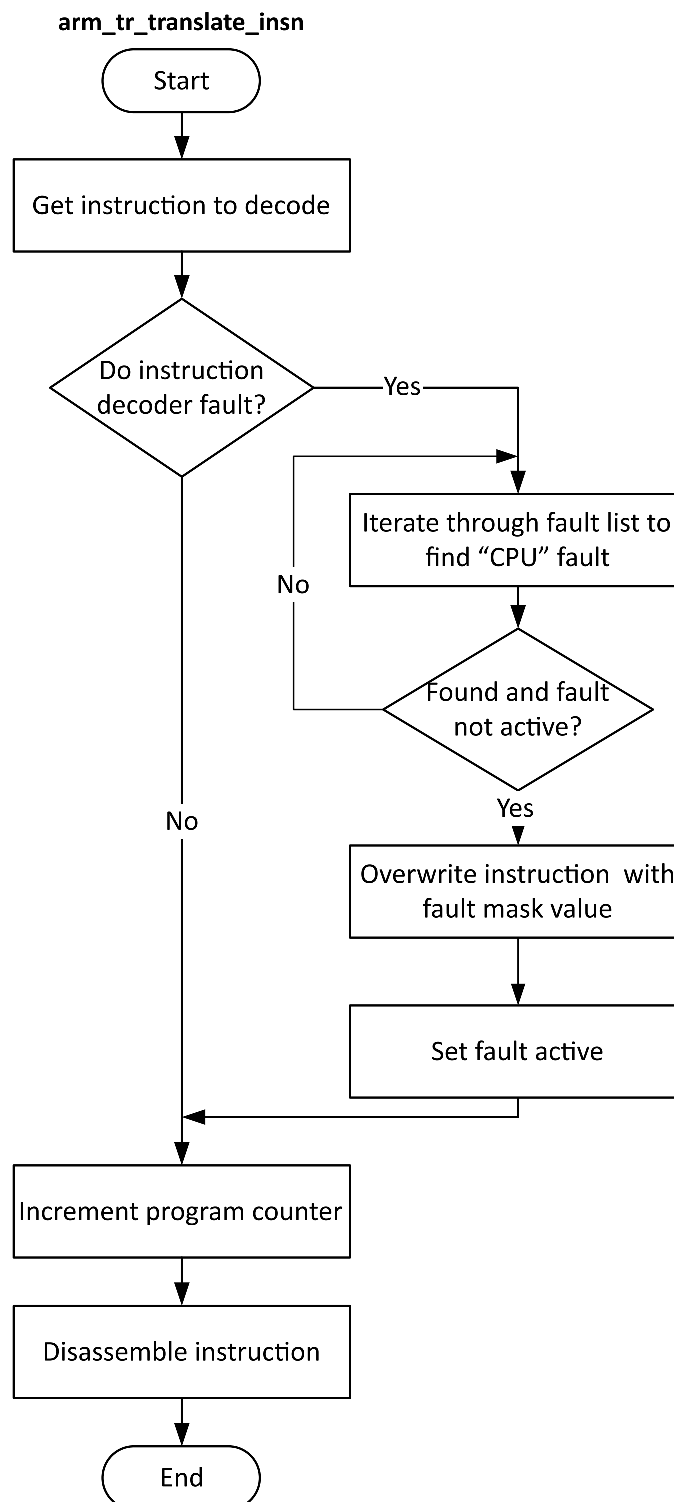


Figure B.6: Changes to arm_tr_translate_insn function

Appendix C

Steering Angle Sensor Development

C.1 Application Validation Diagram

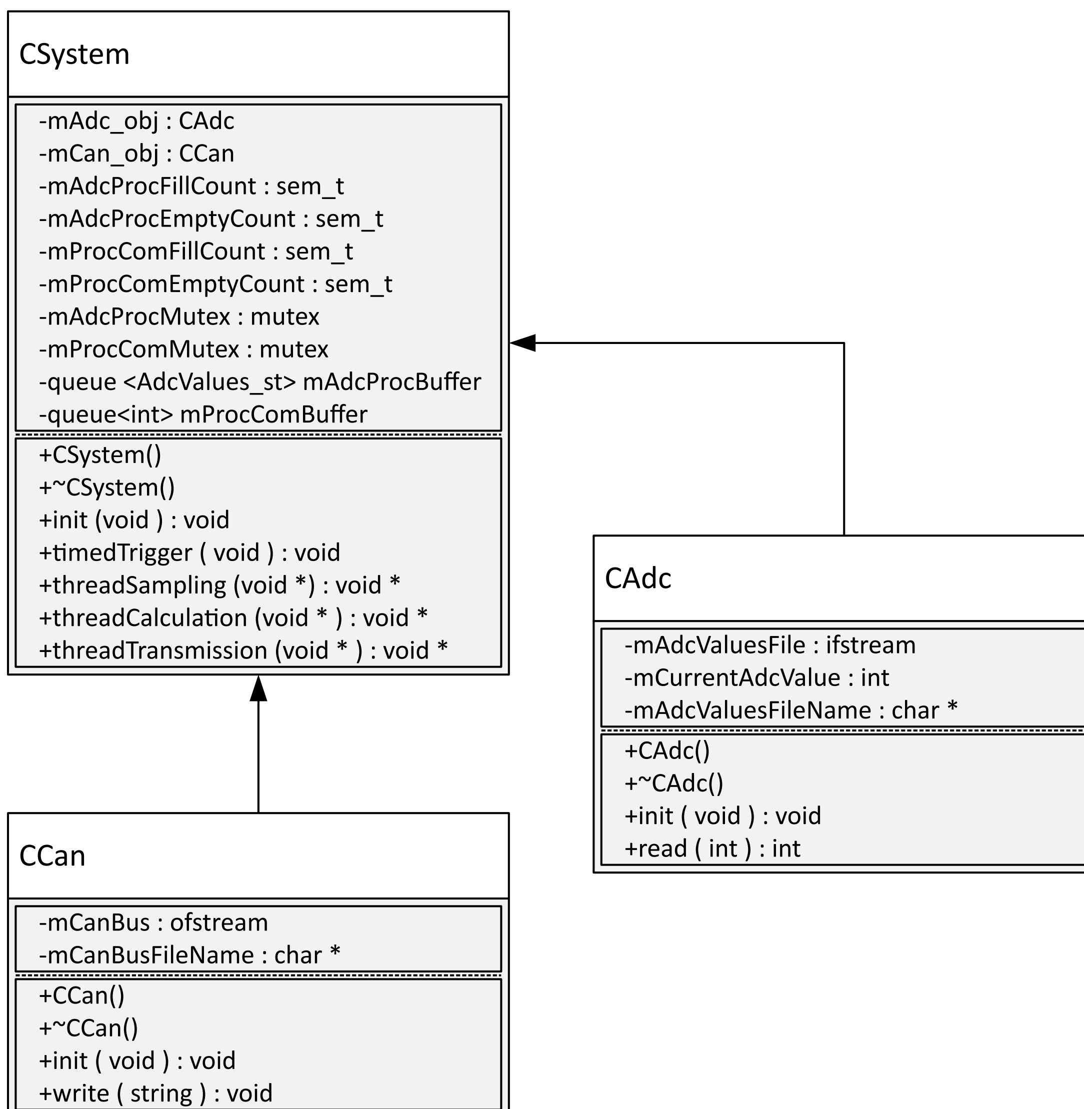


Figure C.1: Class diagram of the software-only SAS validation

C.2 State Machine Diagram

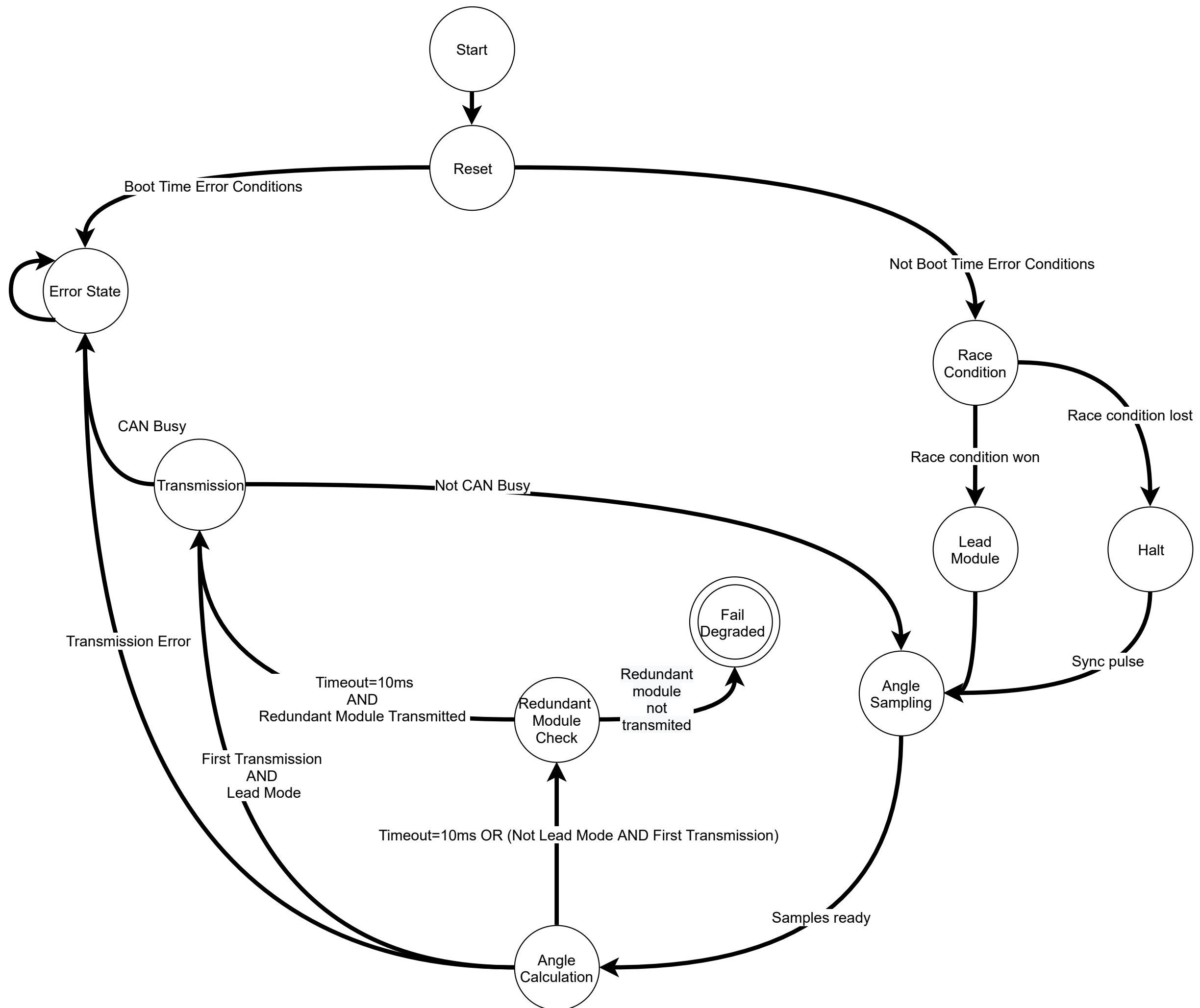


Figure C.2: SAS State Machine diagram

C.3 MCAL Module Diagrams

C.3.1 Class Diagrams

Figure C.3: Class diagram of the WDG MCAL module

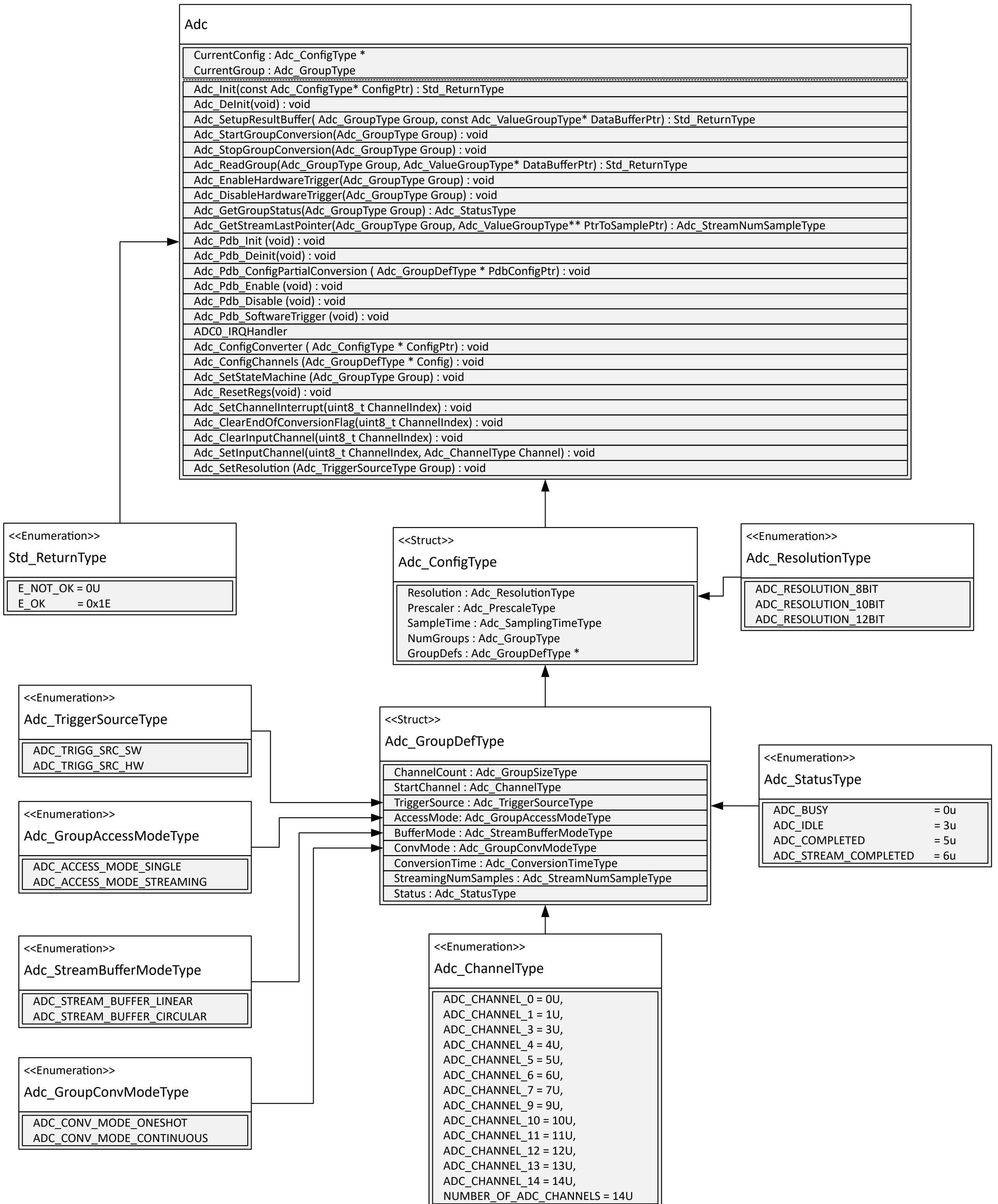


Figure C.4: Class diagram of the ADC MCAL module

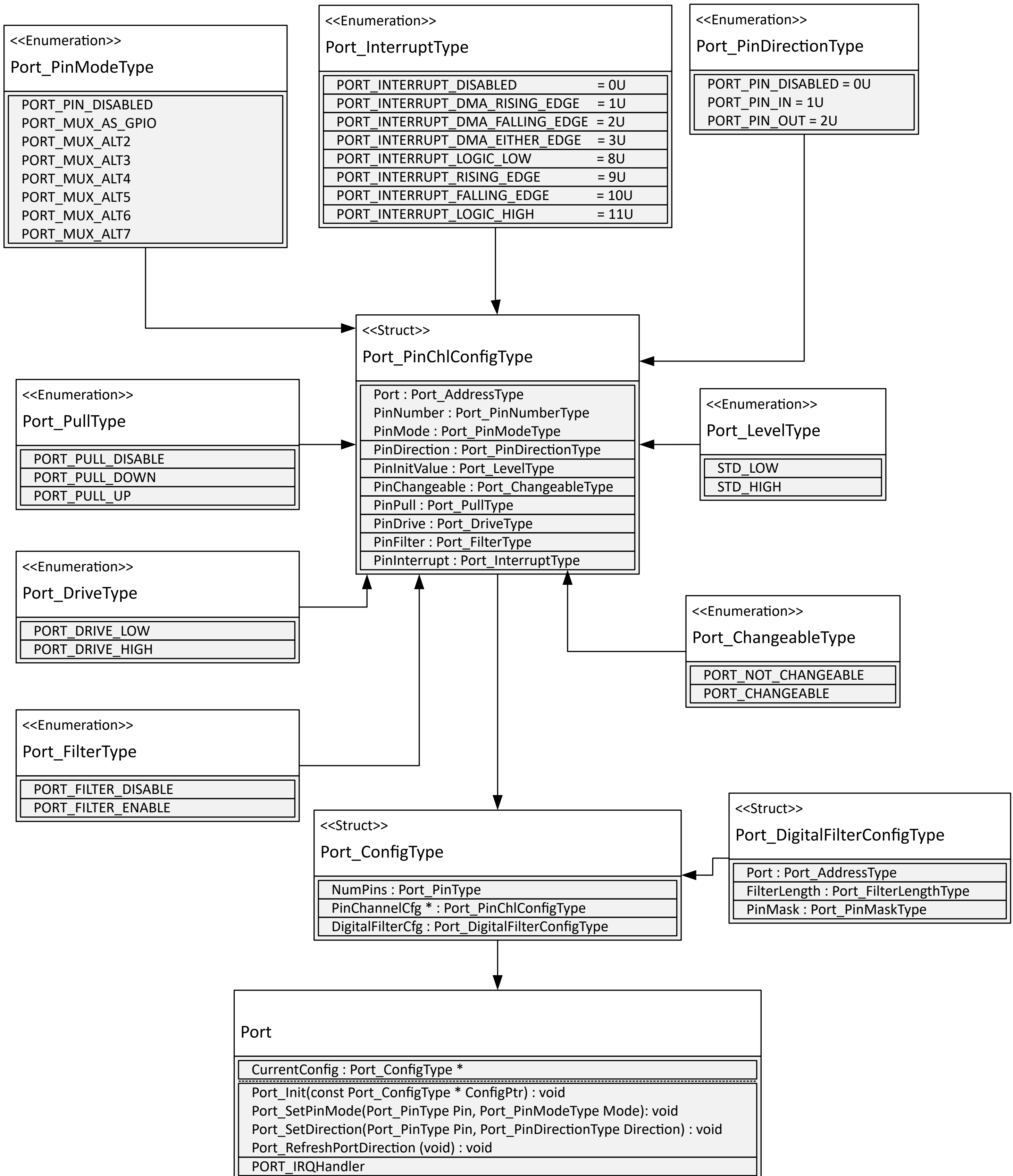


Figure C.5: Class diagram of the PORT MCAL module

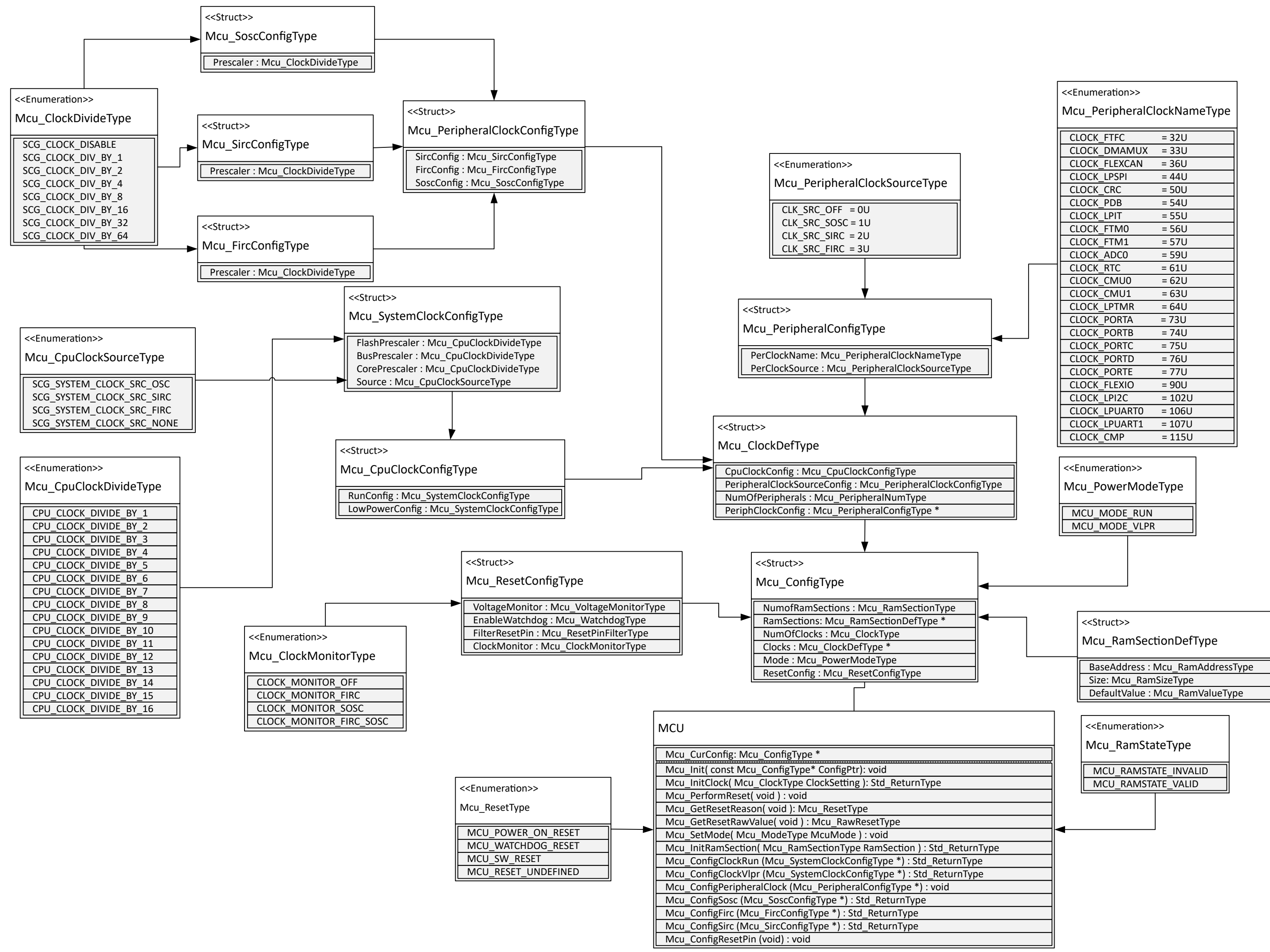


Figure C.6: Class diagram of the MCU MCAL module

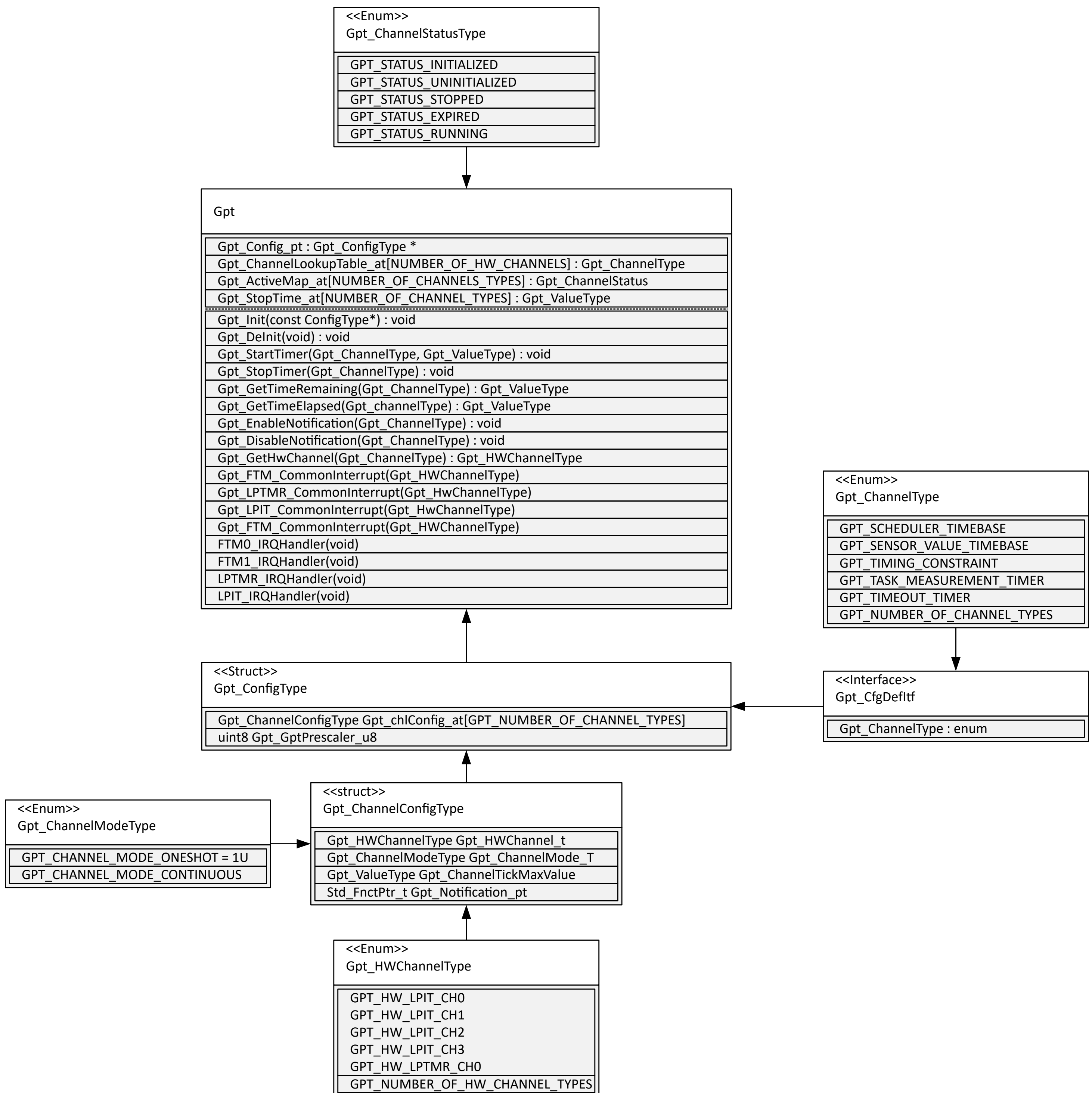


Figure C.7: Class diagram of the GPT MCAL module

C.3.2 Use Case Diagrams

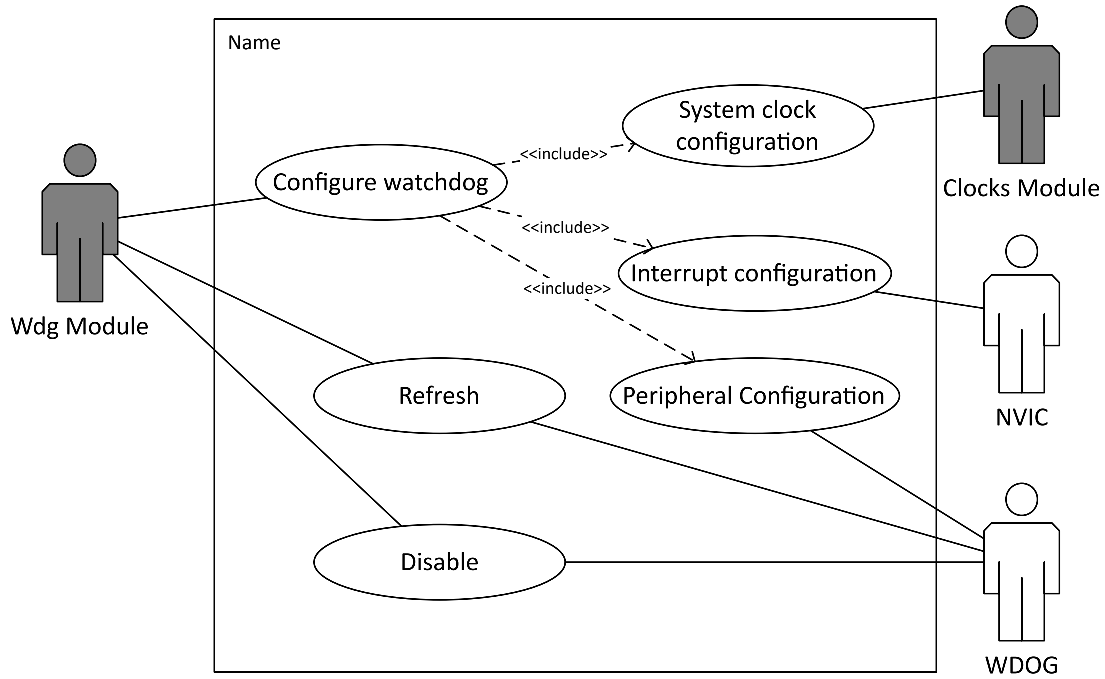


Figure C.8: Use Case diagram of the WDG MCAL module

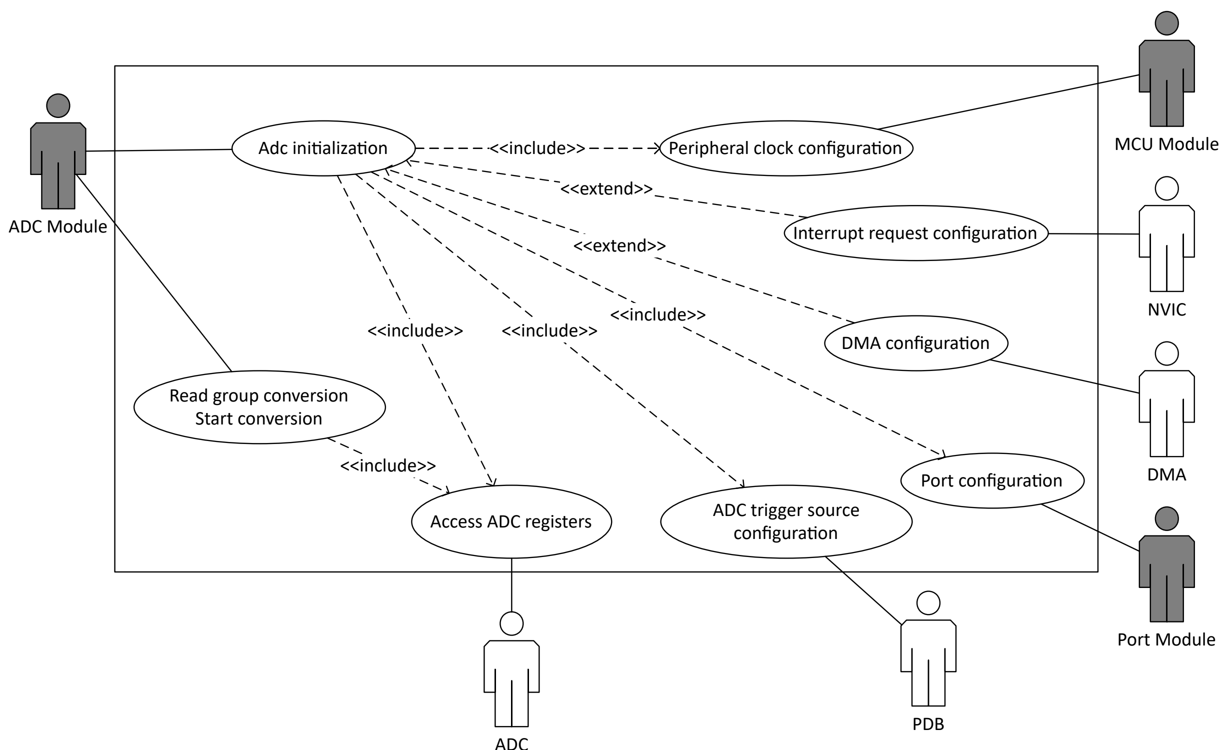


Figure C.9: Use Case diagram of the ADC MCAL module

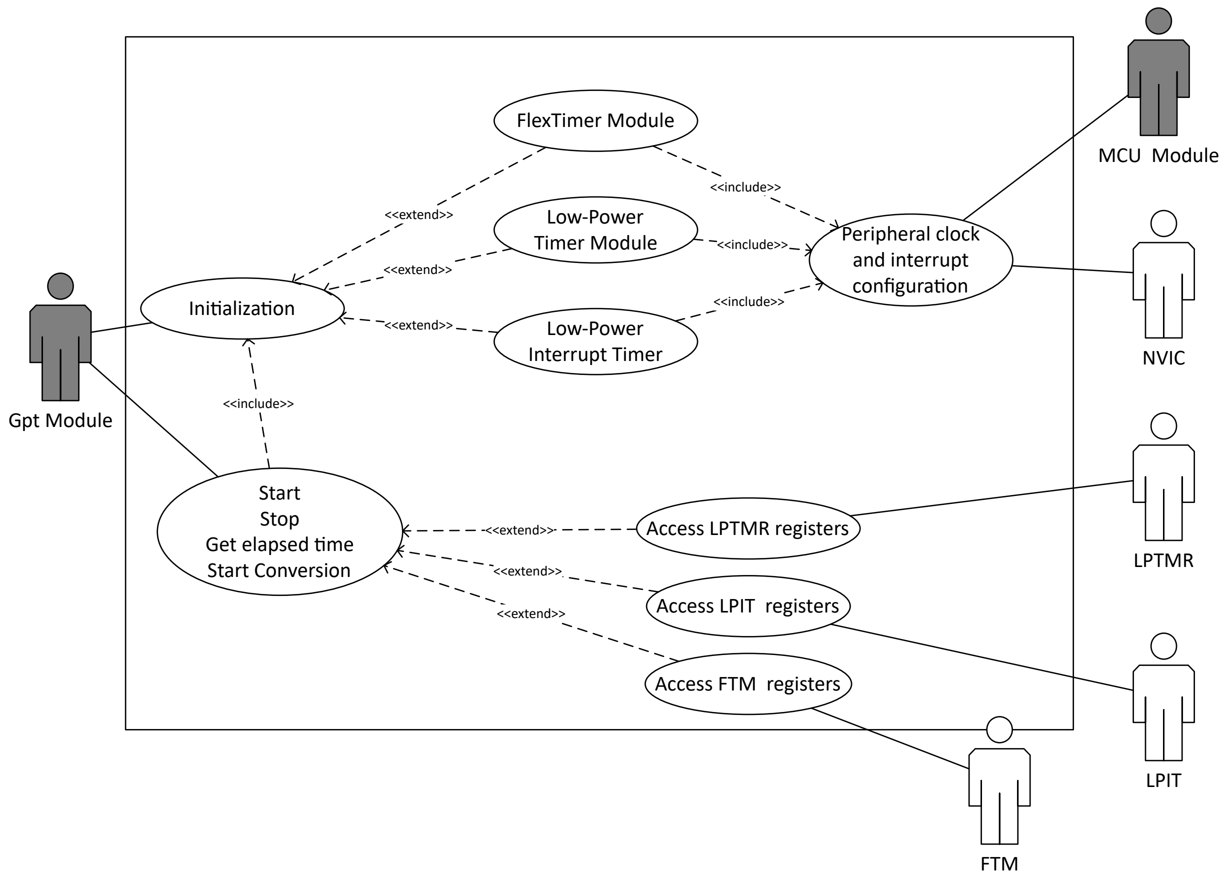


Figure C.10: Use Case diagram of the GPT MCAL module

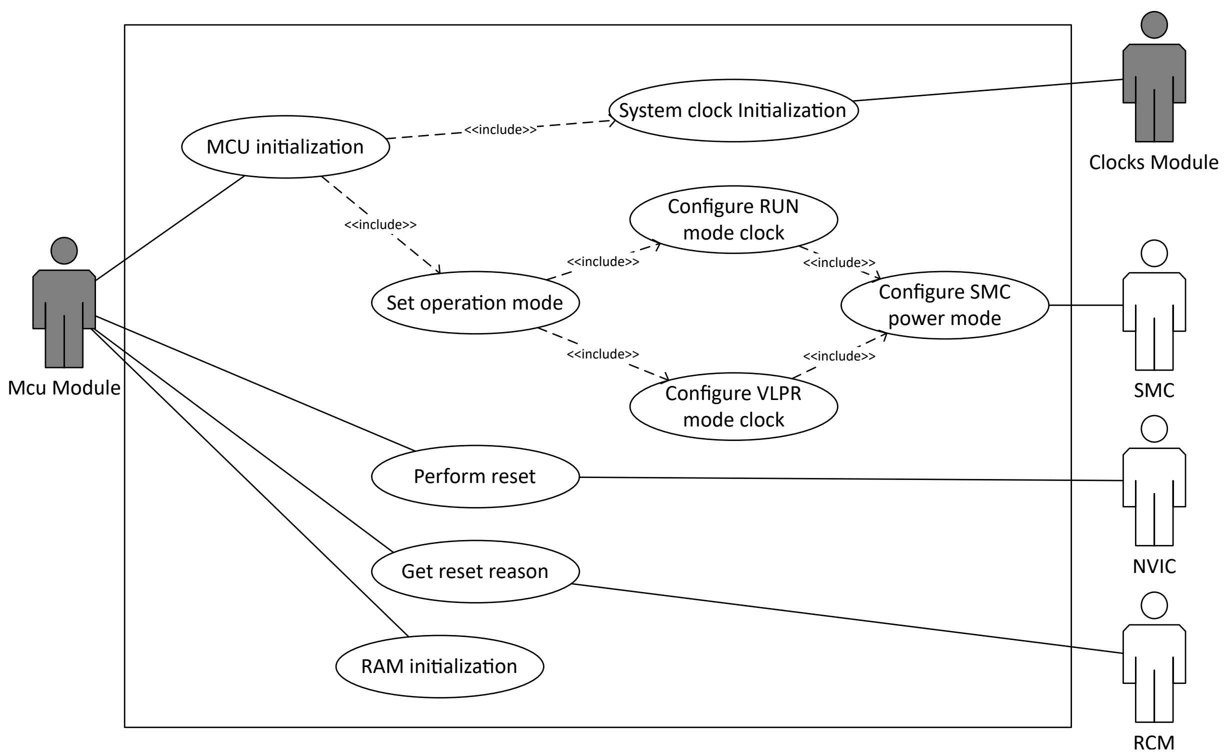


Figure C.11: Use Case diagram of the MCU MCAL module (1)

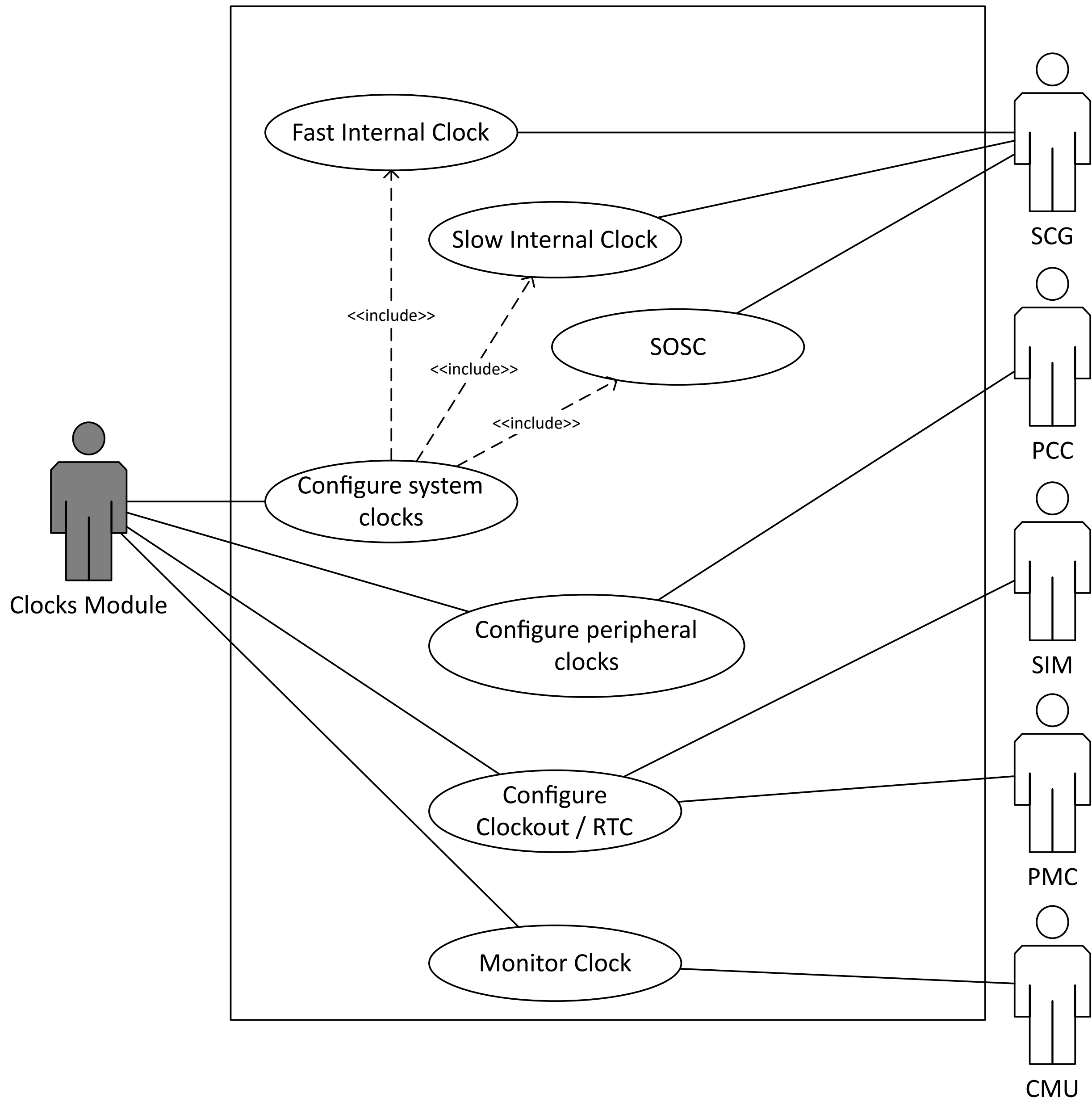


Figure C.12: Use Case diagram of the MCU MCAL module (2)

C.3.3 Sequence Diagrams

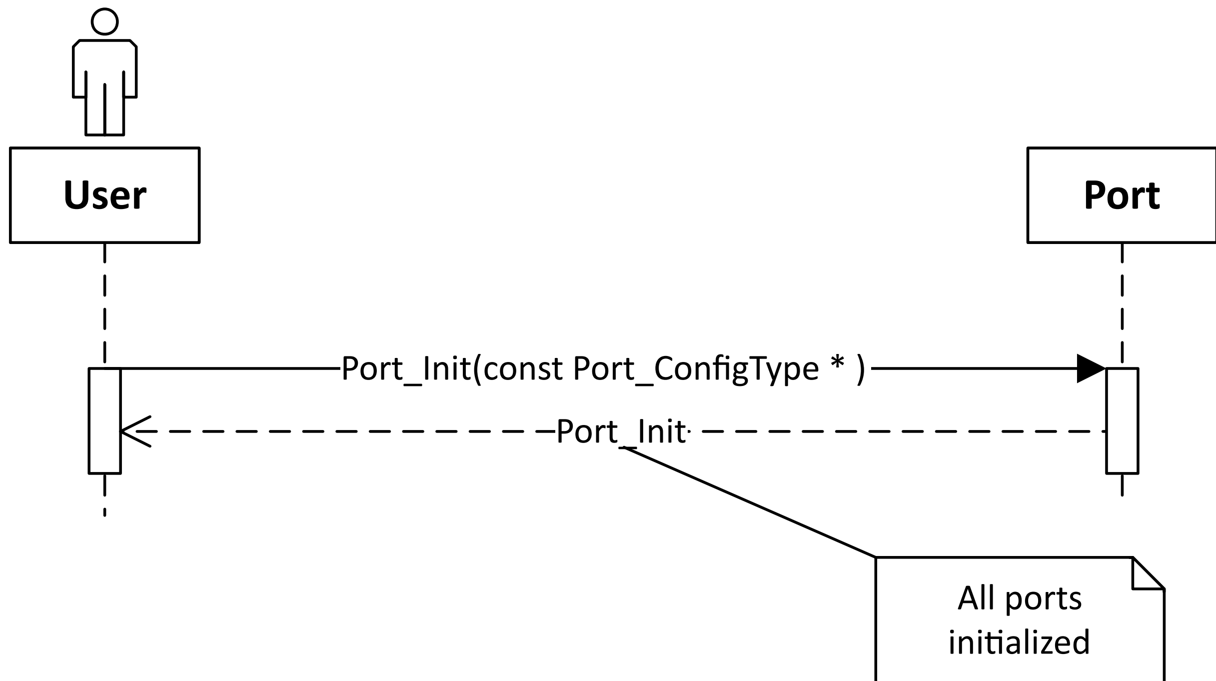


Figure C.13: Sequence diagram of the PORT MCAL module

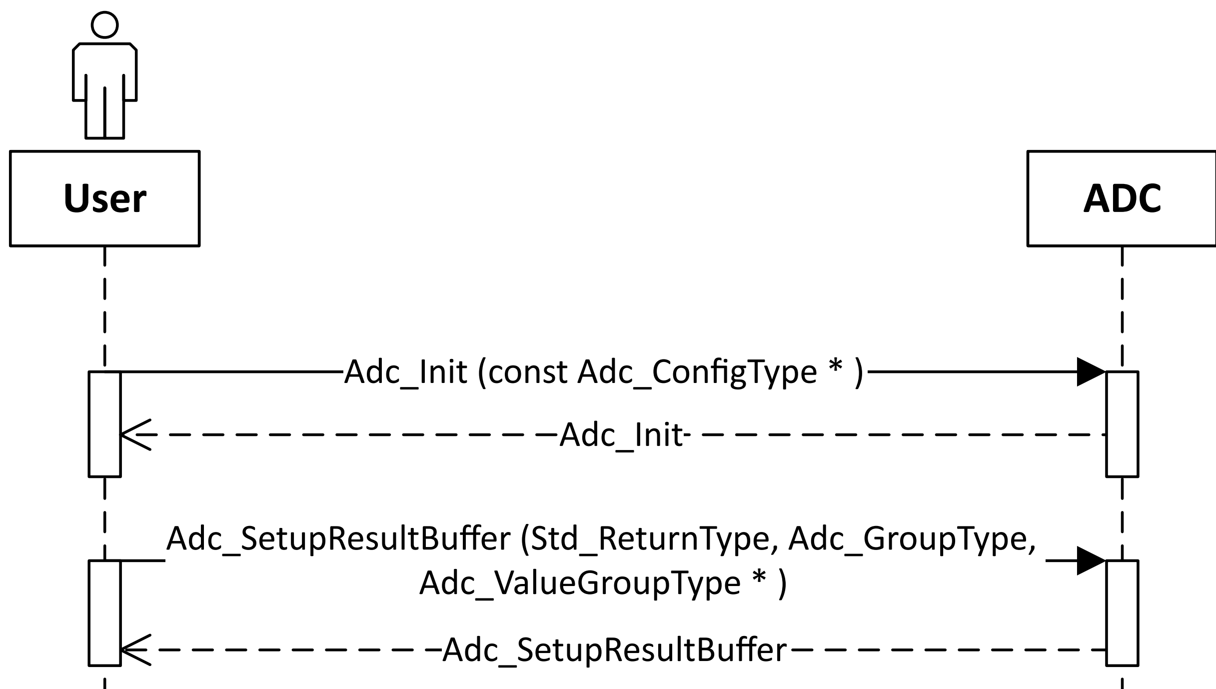


Figure C.14: Sequence diagram of the ADC MCAL module (1)

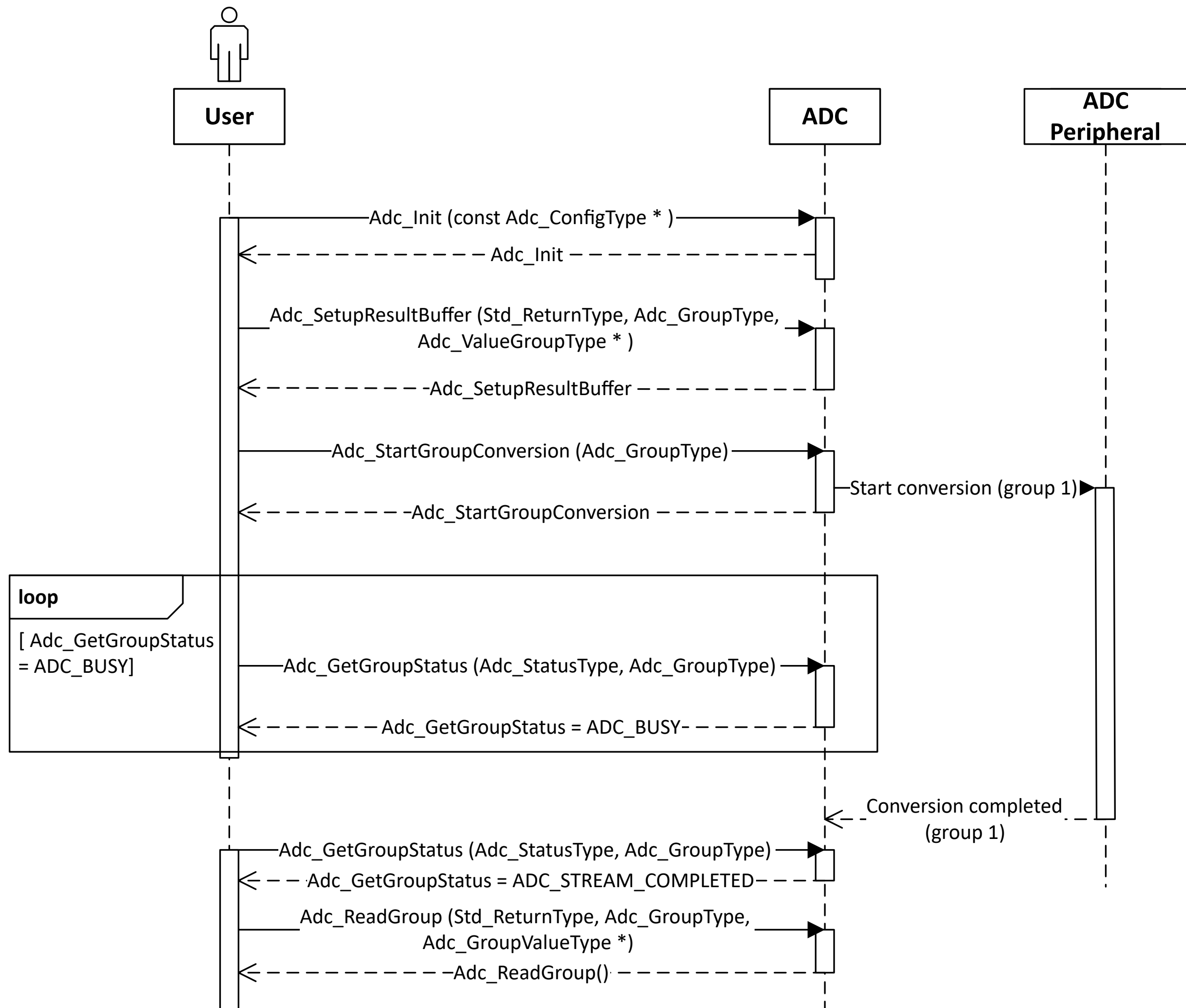


Figure C.15: Sequence diagram of the ADC MCAL module (2)

User
Gpt
Timer peripheral
Gpt_EnableNotification(Gpt_ChannelType)
Gpt_EnableNotification
Gpt_StartTimer(Gpt_ChannelType, Gpt_ValueType)
Start Channel
Gpt_StartTimer
Channel Time Target reached
Gpt Channel Callback
Gpt_GetTimeElapsed(Gpt_ChannelType) : Gpt_ValueType
Read Channel
Channel Count
Gpt_GetTimeElapsed()
Channel Time Target reached
Gpt Channel Callback
Gpt_StopTimer(Gpt_ChannelType)
StopChannel
Gpt_StopTimer()

Figure C.16: Sequence diagram of the GPT MCAL module (1)

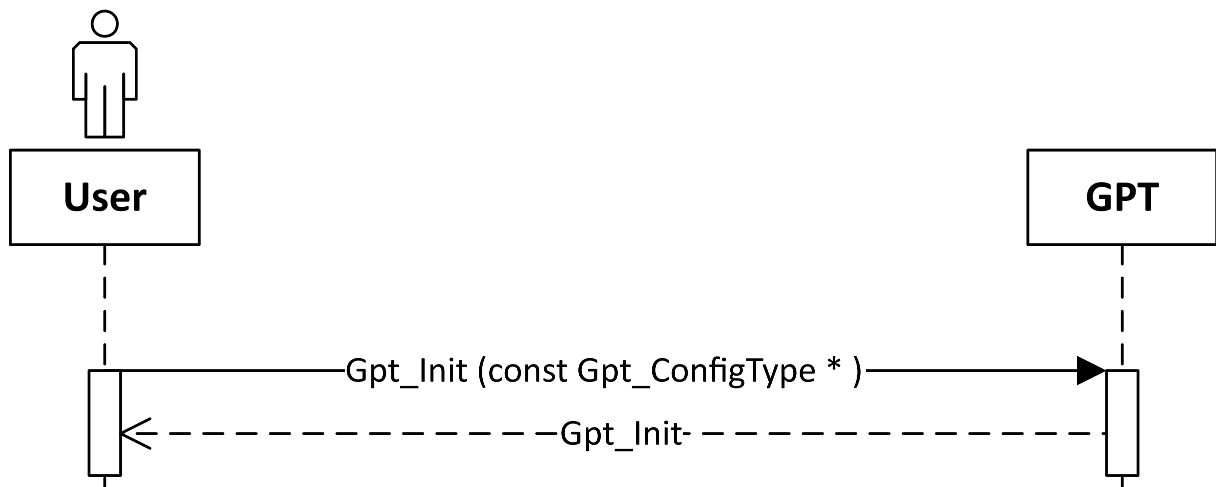


Figure C.17: Sequence diagram of the GPT MCAL module (2)

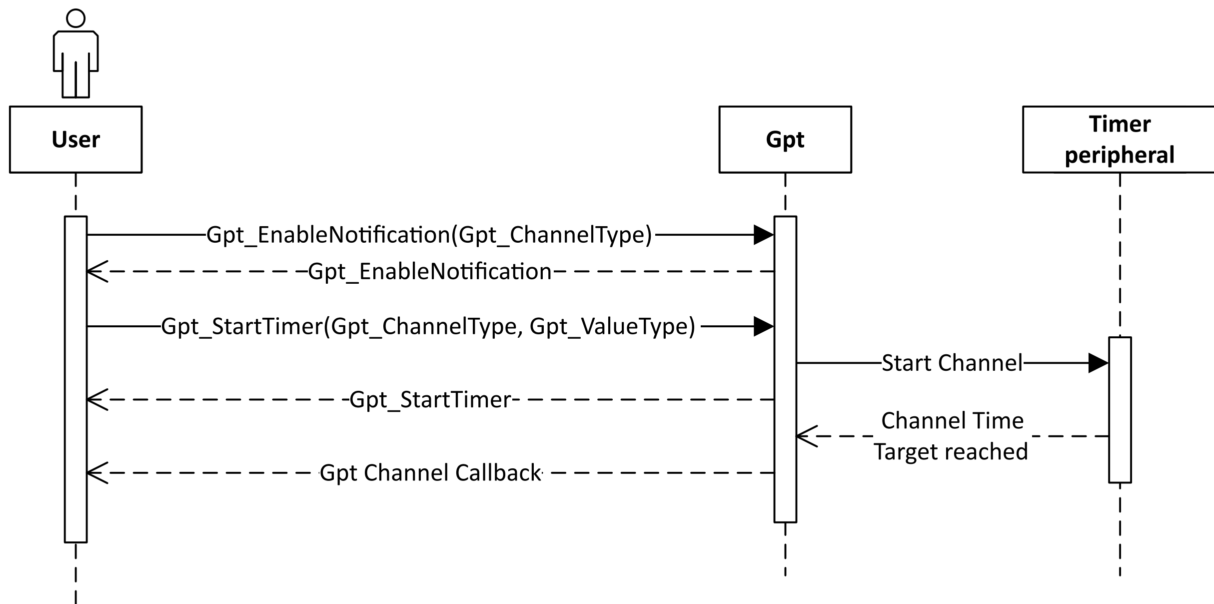


Figure C.18: Sequence diagram of the ADC MCAL module (3)

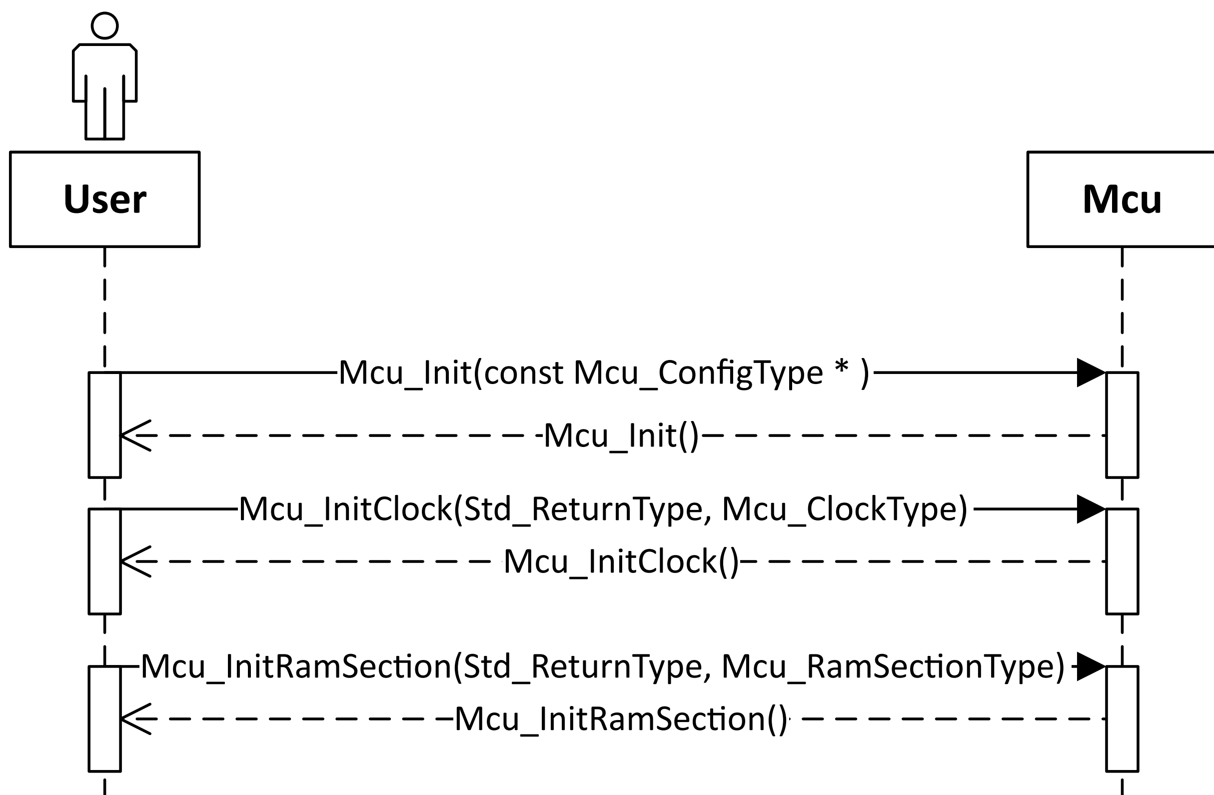


Figure C.19: Sequence diagram of the MCU MCAL module

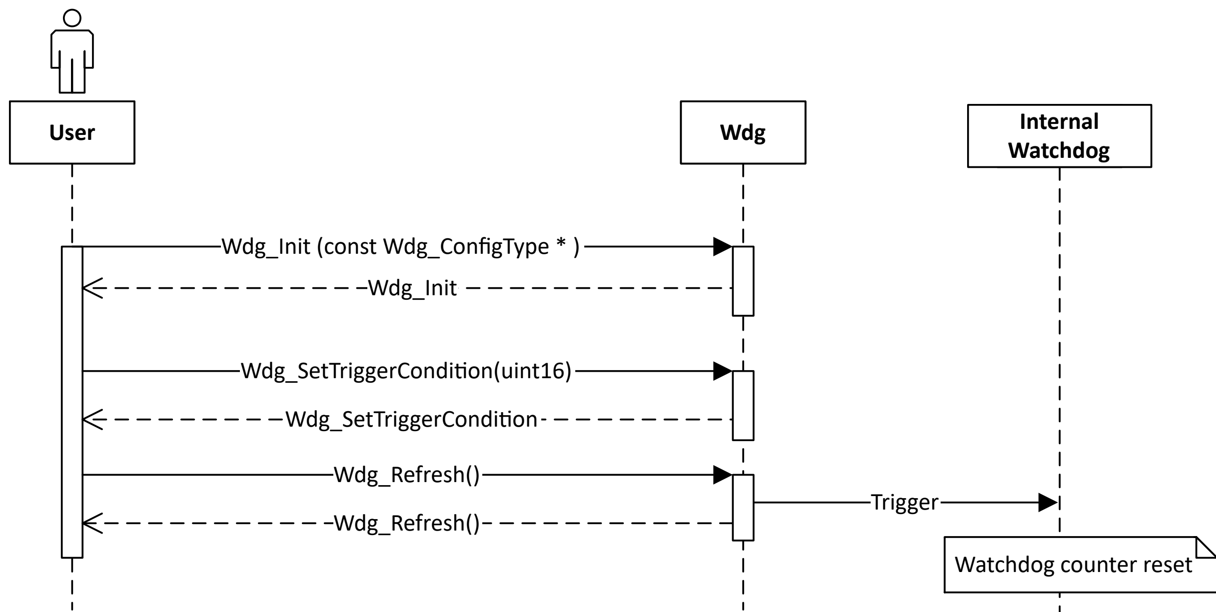


Figure C.20: Sequence diagram of the WDG MCAL module

C.4 Software Flowcharts

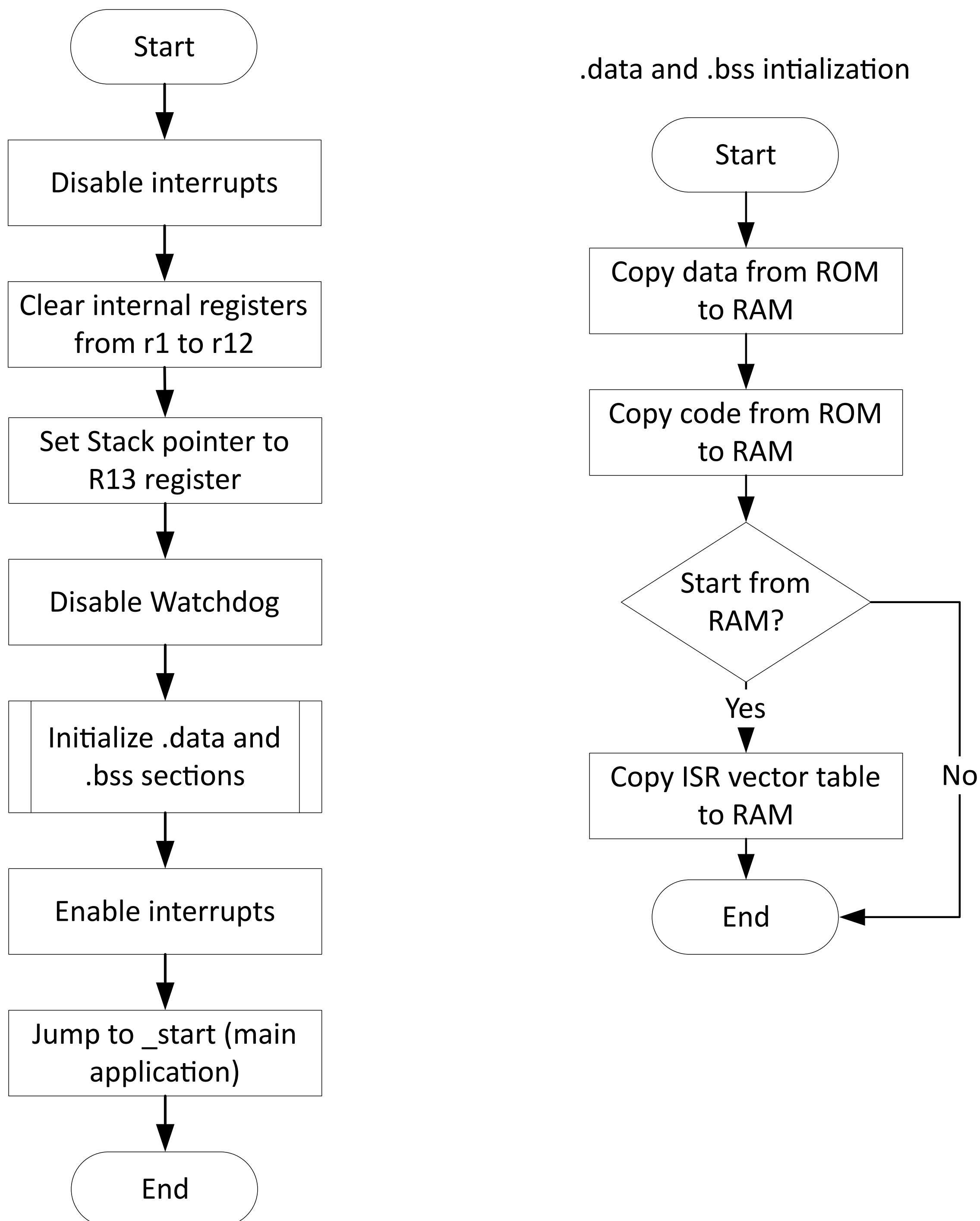


Figure C.21: Flowchart of the microcontroller startup sequence

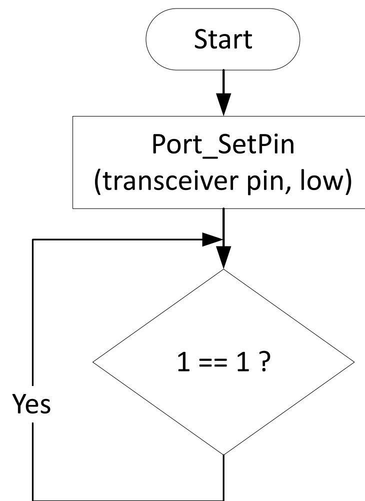


Figure C.22: SAS Error State flowchart

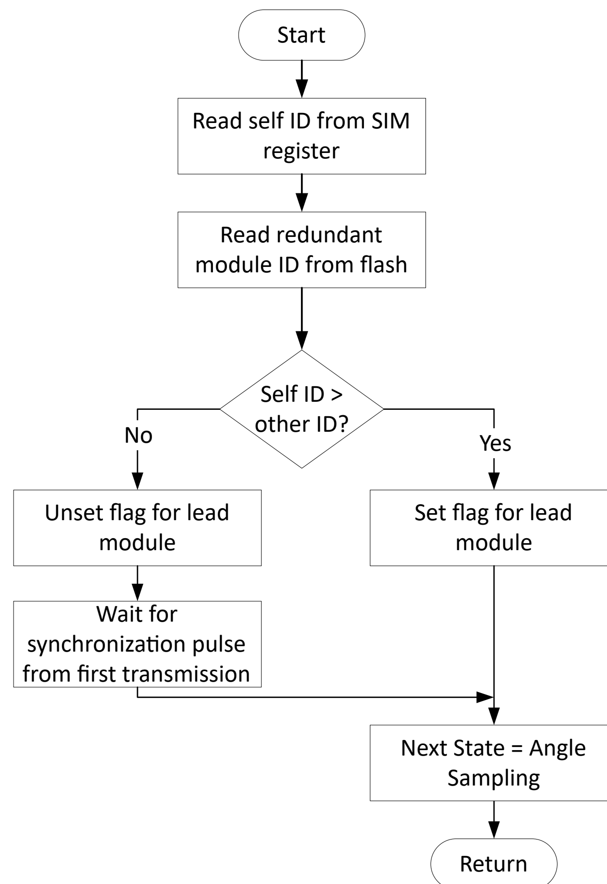


Figure C.23: SAS Race Condition State flowchart

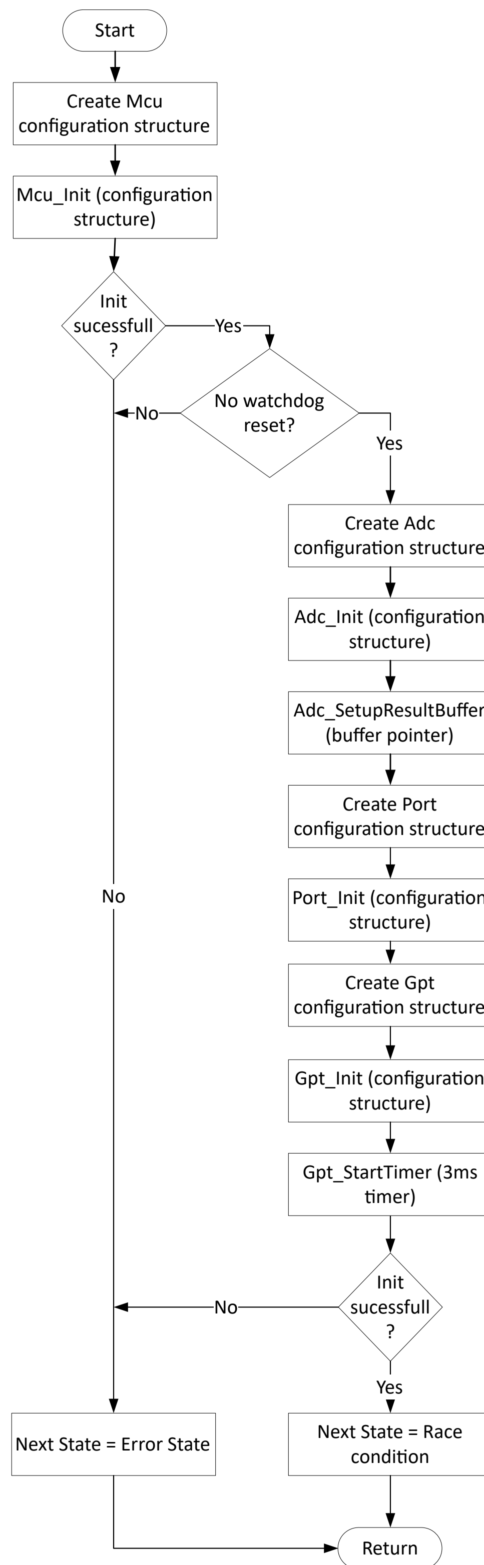


Figure C.24: SAS Reset State flowchart

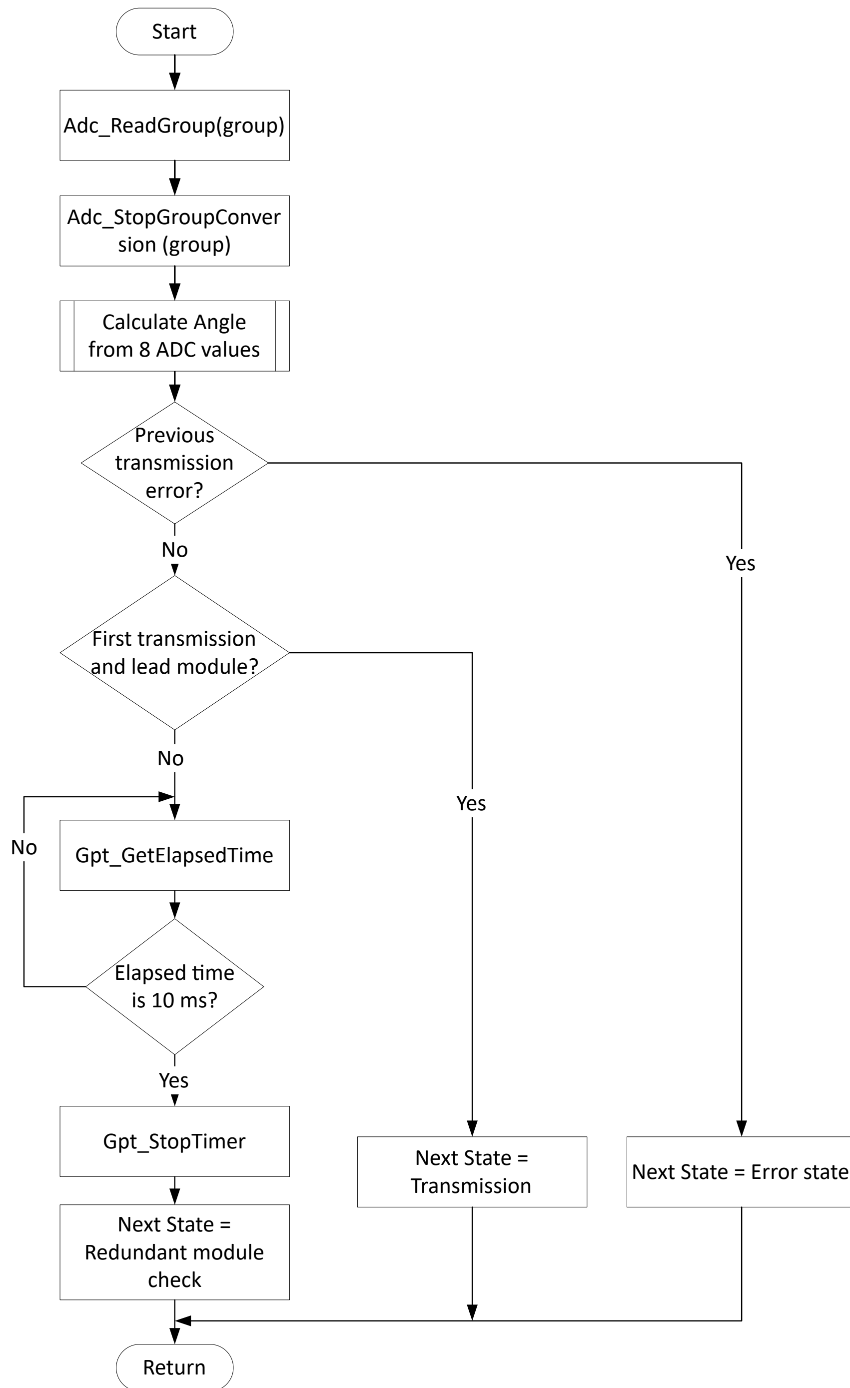


Figure C.25: SAS Angle Calculation flowchart

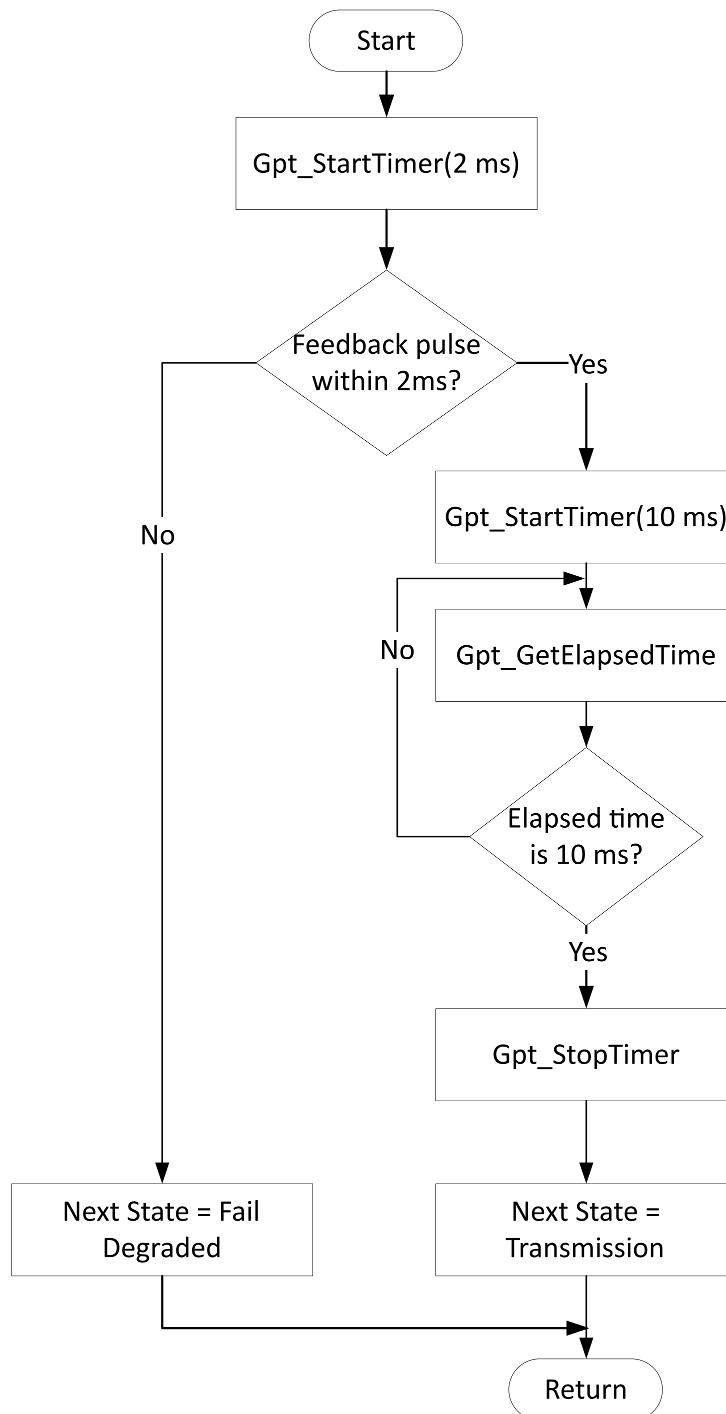


Figure C.26: SAS Redundant Module Check State flowchart

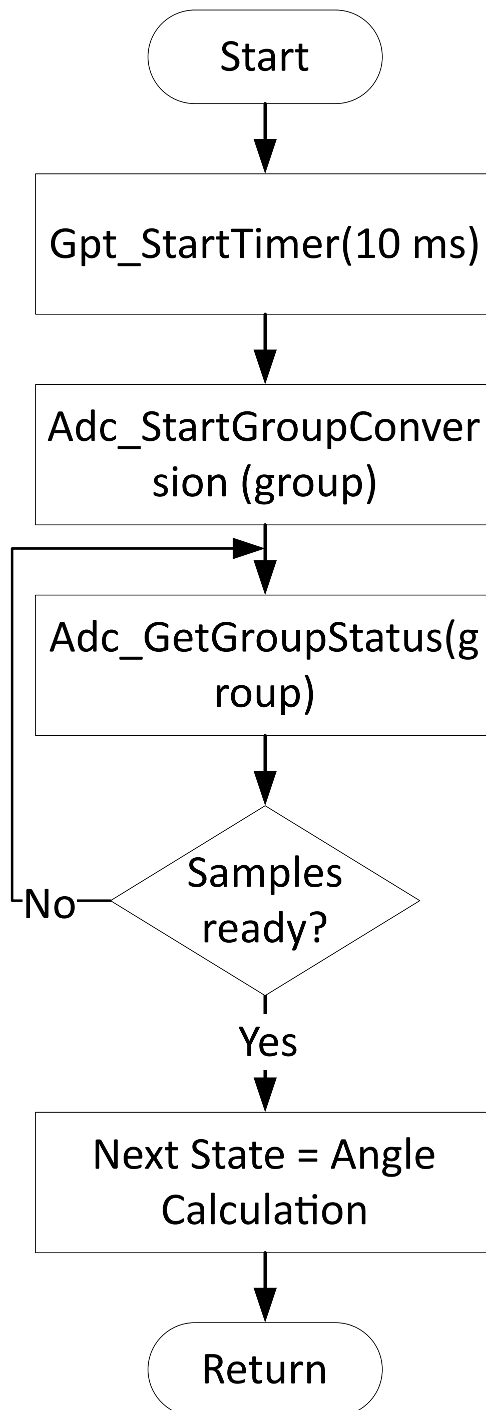


Figure C.27: SAS Angle Sampling State flowchart

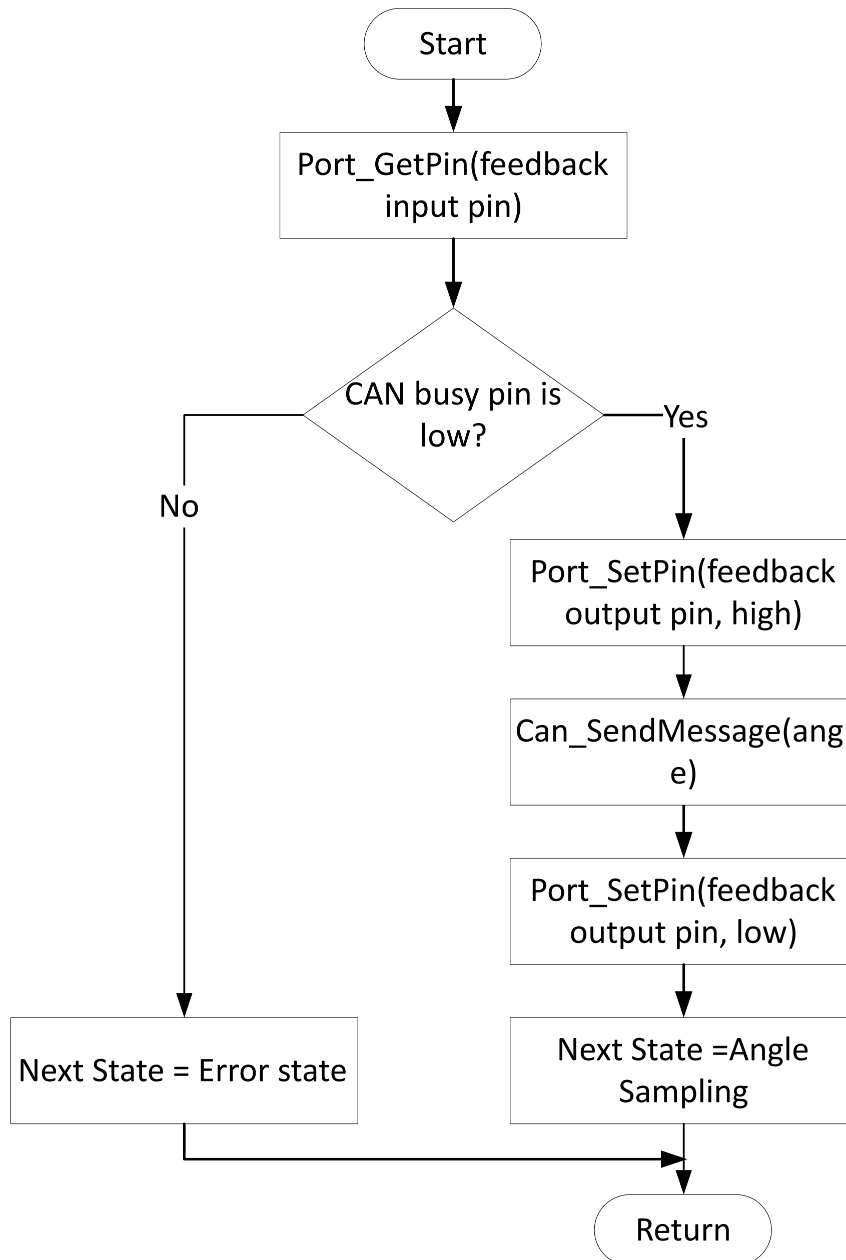


Figure C.28: SAS Transmission State flowchart

Appendix D

QEMU Machine Additional Material

D.1 Addition of command line arguments

To add new command line arguments to QEMU, the file *qemu-options.hx* needs to be modified. Under this file, the new command argument needs to be specified along with the possible parameters that can be used with it. For exemplification purposes the "-fi" command is presented on the snippet below.

```
...
DEF("fi", HAS_ARG, QEMU_OPTION_fi,
"-fi      activates the fault injection experiment\n", QEMU_ARCH_ALL)
STEXI
@item -fi @var{item1}[,...]
@findex -fi
Activates the ability to do fault injection experiments
ETEXI
...
```

After changing this file, the command can be read and attended in the *vl.c* file, as shown on the snippet below.

```
...
case QEMU_OPTION_fi:
    opt_str = (char*)malloc((strlen(optarg)+1) * sizeof(char));
    strcpy(opt_str, optarg);
    sep_str = strtok(opt_str, ",");
    for (param_num = 0; sep_str != NULL; param_num++){
        switch(param_num){
            case 0:
                fi_port = strtol(sep_str, NULL, 10);
                break;
```

```
11                  case 1:
12                      fault_library_name = (char*)malloc((strlen(sep_str) +
    1) * sizeof(char));
13                      strcpy(fault_library_name, sep_str);
14                      error_report("Fault libary name: %s",fault_library_name
    );
15                      break;
16                  case 2:
17                      fi_period =  strtol(sep_str, NULL, 10);
18                      break;
19                      ...
20      ...
```

D.2 S32K116 Machine Class Diagram

Each one of the peripherals also contains *write*, *read*, *init* and *reset* functions, whose interfaces are presented on figure D.1. These functions have the same definition across all peripherals and for that reason, they are not presented on the class diagram to ease readability.

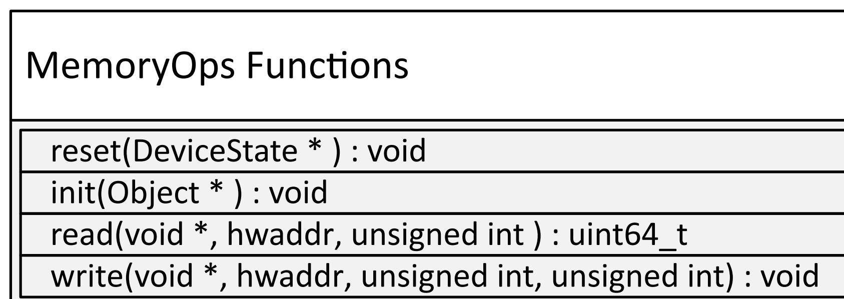


Figure D.1: Memory Operations functions interfaces

The diagrams contain the variables that hold register values and the helper functions used by the functions mentioned above.

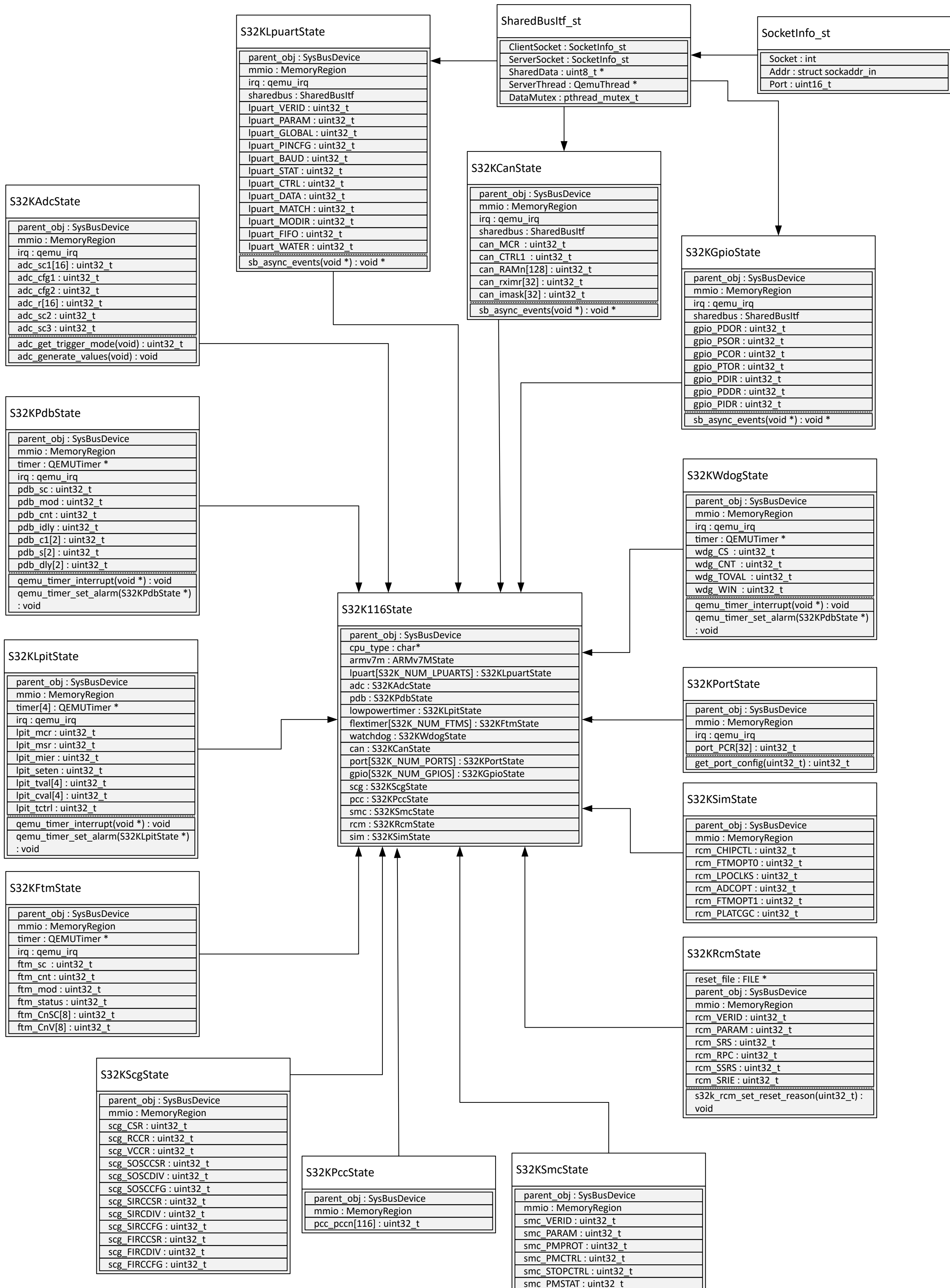


Figure D.2: S32K116 QEMU Machine class diagram

D.3 Peripheral Example (ADC)

```
/*
 * S32K ADC
 */

#include "qemu/osdep.h"
#include "hw/sysbus.h"
#include "hw/hw.h"
#include "qemu/log.h"
#include "qemu/module.h"
#include "hw/adc/s32k_adc.h"
#include <math.h>
#include <time.h>
#include "../../fault-injection-vars.h"
#include "../../adc_values.h"

static S32KAdcState * current_device;

static int adc_values_index = 0;

static void s32k_adc_reset(DeviceState *dev)
{
    S32KAdcState *s = S32K_ADC(dev);
    /* Reset registers */
    for(int i = 0; i < 16; i++)
    {
        s->adc_sc1[i] = 0x0000003F;
        s->adc_r[i] = 0x00000000;
    }
    s->adc_cfg1 = 0x00000000;
    s->adc_cfg2 = 0x0000000C;
    s->adc_sc2 = 0x00000000;
    s->adc_sc3 = 0x00000000;
}
```

```
34
35 static uint64_t s32k_adc_read(void *opaque, hwaddr addr, unsigned int size)
   {
36     S32KAdcState *s = opaque;
37
38     if(addr >= ADC_SC1_OFFSET && addr < ADC_SC1_END){
39         uint32_t index = (addr / ADC_REG_OFFSET);
40         return s->adc_sc1[index];
41     }
42     else if(addr >= ADC_R_OFFSET && addr < ADC_R_END){
43         uint32_t index = ((addr - ADC_R_OFFSET) / ADC_REG_OFFSET);
44         s->adc_sc1[index] &= ~ADC_SC1_COCO_MASK;
45         if((s->adc_sc1[index] & ADC_SC1_AIEN_MASK) != 0){
46             qemu_set_irq(s->irq, 0);
47         }
48         return s->adc_r[index];
49     }
50     else{
51         switch (addr) {
52         case ADC_CFG1_OFFSET:
53             return s->adc_cfg1;
54         case ADC_CFG2_OFFSET:
55             return s->adc_cfg2;
56         case ADC_SC2_OFFSET:
57             return s->adc_sc2;
58         case ADC_SC3_OFFSET:
59             return s->adc_sc3;
60         default:
61         }
62     }
63     return 0;
64 }
65
66 static void s32k_adc_write(void *opaque, hwaddr addr, uint64_t val64,
   unsigned int size){
67     S32KAdcState *s = opaque;
68     uint32_t value = (uint32_t) val64;
```

```
69
70      if(addr >= ADC_SC1_OFFSET && addr < ADC_SC1_END){
71          uint32_t index = (addr / ADC_REG_OFFSET);
72          s->adc_sc1[index] = value;
73          return;
74      }
75      else{
76          switch (addr) {
77          case ADC_CFG1_OFFSET:
78              s->adc_cfg1 = value;
79              return;
80          case ADC_CFG2_OFFSET:
81              s->adc_cfg2 = value;
82              return;
83          case ADC_SC2_OFFSET:
84              s->adc_sc2 = value;
85              return;
86          case ADC_SC3_OFFSET:
87              s->adc_sc3 = value;
88              if((s->adc_sc3 & ADC_SC3_CAL_MASK) != 0){
89                  s->adc_sc1[0] |= ADC_SC1_COCO_MASK; /* Set COCO flag */
90              }
91              return;
92          default:
93          }
94      }
95      return;
96  }
97
98  static const MemoryRegionOps s32k_adc_ops = {
99      .read = s32k_adc_read,
100     .write = s32k_adc_write,
101     .endianness = DEVICE_NATIVE_ENDIAN,
102 };
103
104 static void s32k_adc_init(Object *obj)
105 {
```

```
106     S32KAdcState *s = S32K_ADC(obj);
107     current_device = s;
108     sysbus_init_irq(SYS_BUS_DEVICE(obj), &s->irq);
109     memory_region_init_io(&s->mmio, obj, &s32k_adc_ops, s, TYPE_S32K_ADC, 0
    x208);
110     sysbus_init_mmio(SYS_BUS_DEVICE(obj), &s->mmio);
111 }
112
113 static void s32k_adc_class_init(ObjectClass *klass, void *data)
114 {
115     DeviceClass *dc = DEVICE_CLASS(klass);
116     dc->reset = s32k_adc_reset;
117 }
118
119 static const TypeInfo s32k_adc_info = {
120     .name          = TYPE_S32K_ADC,
121     .parent        = TYPE_SYS_BUS_DEVICE,
122     .instance_size = sizeof(S32KAdcState),
123     .instance_init = s32k_adc_init,
124     .class_init    = s32k_adc_class_init,
125 };
126
127 static void s32k_adc_register_types(void)
128 {
129     type_register_static(&s32k_adc_info);
130 }
131
132 type_init(s32k_adc_register_types)
```

Listing D.1: Emulated ADC peripheral example

D.4 S32K116 Machine

```
1 /*
2  * S32K116 SoC
3  */
4
5 #include "qemu/osdep.h"
```

```
6  #include "qapi/error.h"
7  #include "qemu/module.h"
8  #include "hw/arm/boot.h"
9  #include "hw/misc/unimp.h"
10 #include "exec/address-spaces.h"
11 #include "hw/arm/s32k116_soc.h"
12 
13 /* Peripheral address */
14 static const uint32_t lpuart_addr[S32K_NUM_LPUARTS] = { 0x4006A000, 0
     x4006B000 };
15 static const uint32_t adc_addr = 0x4003B000;
16 static const uint32_t pdb_addr = 0x40036000;
17 static const uint32_t lpit_addr = 0x40037000;
18 static const uint32_t ftm_addr[S32K_NUM_FTMS] = { 0x40038000, 0x40039000 };
19 static const uint32_t scg_addr = 0x40064000;
20 static const uint32_t pcc_addr = 0x40065000;
21 static const uint32_t pmc_addr = 0x4007D000;
22 static const uint32_t smc_addr = 0x4007E000;
23 static const uint32_t rcm_addr = 0x4007F000;
24 static const uint32_t sim_addr = 0x40048000;
25 static const uint32_t wdg_addr = 0x40052000;
26 static const uint32_t can_addr = 0x40024000;
27 static const uint32_t port_addr[S32K_NUM_PORTS] = {0x40049000, 0x4004A000,
     0x4004B000, 0x4004C000, 0x4004D000 };
28 static const uint32_t gpio_addr[S32K_NUM_GPIOS] = {0x400FF000, 0x400FF040,
     0x400FF080, 0x400FF0C0, 0x400FF100 };
29 
30 /* Peripheral IRQS */
31 static const int lpuart_irq[S32K_NUM_LPUARTS] = {31, 30};
32 static const int adc_irq = 28;
33 static const int pdb_irq = 19;
34 static const int lpit_irq = 20;
35 static const int ftm_irq[S32K_NUM_FTMS] = {12, 15};
36 static const int wdg_irq = 22;
37 static const int can_irq = 10;
38 static const int port_irq = 9;      /* Connect interrupt to all ports/gpios
      */
```

```
39 static const int scg_irq = 21;
40
41 static void s32k116_soc_initfn(Object *obj){
42     S32K116State *s = S32K116_SOC(obj);
43     int i;
44     sysbus_init_child_obj(obj, "armv6m", &s->armv7m, sizeof(s->armv7m),
   TYPE_ARMV7M);
45     for (i = 0; i < S32K_NUM_LPUARTS; i++){
46         sysbus_init_child_obj(obj, "lpuart[*]", &s->lpuart[i], sizeof(s->
   lpuart[i]), TYPE_S32K_LPUART);
47     }
48     sysbus_init_child_obj(obj, "adc", &s->adc, sizeof(s->adc),
   TYPE_S32K_ADC);
49     sysbus_init_child_obj(obj, "smc", &s->smc, sizeof(s->smc),
   TYPE_S32K_SMC);
50     sysbus_init_child_obj(obj, "sim", &s->sim, sizeof(s->sim),
   TYPE_S32K_SIM);
51     sysbus_init_child_obj(obj, "rcm", &s->rcm, sizeof(s->rcm),
   TYPE_S32K_RCM);
52     sysbus_init_child_obj(obj, "pcc", &s->pcc, sizeof(s->pcc),
   TYPE_S32K_PCC);
53     sysbus_init_child_obj(obj, "scg", &s->scg, sizeof(s->scg),
   TYPE_S32K_SCG);
54     sysbus_init_child_obj(obj, "wdg", &s->wdg, sizeof(s->wdg),
   TYPE_S32K_WDG);
55     sysbus_init_child_obj(obj, "pdb", &s->pdb, sizeof(s->pdb),
   TYPE_S32K_PDB);
56     sysbus_init_child_obj(obj, "lpit", &s->lpit, sizeof(s->lpit),
   TYPE_S32K_LPIT);
57     sysbus_init_child_obj(obj, "ftm0", &s->ftm0, sizeof(s->ftm0),
   TYPE_S32K_FTM);
58     sysbus_init_child_obj(obj, "can", &s->can, sizeof(s->can),
   TYPE_S32K_CAN);
59     s->port_irqs = OR_IRQ(object_new(TYPE_OR_IRQ));
60     for (i = 0; i < S32K_NUM_PORTS; i++){
61         sysbus_init_child_obj(obj, "port[*]", &s->port[i], sizeof(s->port[i
   ]), TYPE_S32K_PORT);
```

```
62      }
63      sysbus_init_child_obj(obj, "pta", &s->pta, sizeof(s->pta),
    TYPE_S32K_GPIO_A);
64      sysbus_init_child_obj(obj, "ptb", &s->ptb, sizeof(s->ptb),
    TYPE_S32K_GPIO);
65      sysbus_init_child_obj(obj, "ptc", &s->ptc, sizeof(s->ptc),
    TYPE_S32K_GPIO_SB);
66      sysbus_init_child_obj(obj, "ptd", &s->ptd, sizeof(s->ptd),
    TYPE_S32K_GPIO);
67      sysbus_init_child_obj(obj, "pte", &s->pte, sizeof(s->pte),
    TYPE_S32K_GPIO);
68  }
69
70  static void s32k116_soc_realize(DeviceState *dev_soc, Error **errp){
71      S32K116State *s = S32K116_SOC(dev_soc);
72      DeviceState *dev, *armv7m;
73      SysBusDevice *busdev;
74      Error *err = NULL;
75      int i;
76      MemoryRegion *system_memory = get_system_memory();
77      MemoryRegion *sram = g_new(MemoryRegion, 1);
78      MemoryRegion *flash = g_new(MemoryRegion, 1);
79      MemoryRegion *flash_alias = g_new(MemoryRegion, 1);
80      memory_region_init_ram(flash, NULL, "S32K116.flash", FLASH_SIZE, &
    error_fatal);
81      memory_region_init_alias(flash_alias, NULL, "S32K116.flash.alias",
    flash, 0, FLASH_SIZE);
82      memory_region_set_readonly(flash, true);
83      memory_region_set_readonly(flash_alias, true);
84      memory_region_add_subregion(system_memory, FLASH_BASE_ADDRESS, flash);
85      memory_region_add_subregion(system_memory, 0, flash_alias);
86      memory_region_init_ram(sram, NULL, "S32K116.sram", SRAM_SIZE, &
    error_fatal);
87      memory_region_add_subregion(system_memory, SRAM_BASE_ADDRESS, sram);
88      armv7m = DEVICE(&s->armv7m);
89      qdev_prop_set_uint32(armv7m, "num-irq", 32);
90      qdev_prop_set_string(armv7m, "cpu-type", s->cpu_type);
```

```
91     qdev_prop_set_bit(armv7m, "enable-bitband", true);
92     object_property_set_link(OBJECT(&s->armv7m), OBJECT(get_system_memory()
    ), "memory", &error_abort);
93     object_property_set_bool(OBJECT(&s->armv7m), true, "realized", &err);
94     if (err != NULL) {
95         error_propagate(errp, err);
96         return;
97     }
98     /* Attach LPUART */
99     for (i = 0; i < S32K_NUM_LPUARTS; i++) {
100        dev = DEVICE(&(s->lpuart[i]));
101        qdev_prop_set_chr(dev, "chardev", serial_hd(i));
102        object_property_set_bool(OBJECT(&s->lpuart[i]), true, "realized", &
    err);
103        if (err != NULL) {
104            error_propagate(errp, err);
105            return;
106        }
107        busdev = SYS_BUS_DEVICE(dev);
108        sysbus_mmio_map(busdev, 0, lpuart_addr[i]);
109        sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, lpuart_irq[i
    ]));
110    }
111    /* Attach ADC */
112    dev = DEVICE(&(s->adc));
113    object_property_set_bool(OBJECT(&s->adc), true, "realized", &err);
114    if (err != NULL) {
115        error_propagate(errp, err);
116        return;
117    }
118    busdev = SYS_BUS_DEVICE(dev);
119    sysbus_mmio_map(busdev, 0, adc_addr);
120    sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, adc_irq));
121    /* Attach SMC */
122    dev = DEVICE(&s->smc);
123    object_property_set_bool(OBJECT(&s->smc), true, "realized", &err);
124    if (err != NULL) {
```

```
125         error_propagate(errp, err);
126         return;
127     }
128     busdev = SYS_BUS_DEVICE(dev);
129     sysbus_mmio_map(busdev, 0, smc_addr);
130     /* Attach SIM */
131     dev = DEVICE(&s->sim);
132     object_property_set_bool(OBJECT(&s->sim), true, "realized", &err);
133     if (err != NULL) {
134         error_propagate(errp, err);
135         return;
136     }
137     busdev = SYS_BUS_DEVICE(dev);
138     sysbus_mmio_map(busdev, 0, sim_addr);
139     /* Attach RCM */
140     dev = DEVICE(&s->rcm);
141     object_property_set_bool(OBJECT(&s->rcm), true, "realized", &err);
142     if (err != NULL) {
143         error_propagate(errp, err);
144         return;
145     }
146     busdev = SYS_BUS_DEVICE(dev);
147     sysbus_mmio_map(busdev, 0, rcm_addr);
148     /* Attach PCC */
149     dev = DEVICE(&s->pcc);
150     object_property_set_bool(OBJECT(&s->pcc), true, "realized", &err);
151     if (err != NULL) {
152         error_propagate(errp, err);
153         return;
154     }
155     busdev = SYS_BUS_DEVICE(dev);
156     sysbus_mmio_map(busdev, 0, pcc_addr);
157     /* Attach SCG */
158     dev = DEVICE(&s->scg);
159     object_property_set_bool(OBJECT(&s->scg), true, "realized", &err);
160     if (err != NULL) {
161         error_propagate(errp, err);
```

```
162            return;
163        }
164        busdev = SYS_BUS_DEVICE(dev);
165        sysbus_mmio_map(busdev, 0, scg_addr);
166        sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, scg_irq));
167        /* Attach WDG */
168        dev = DEVICE(&s->wdg);
169        object_property_set_bool(OBJECT(&s->wdg), true, "realized", &err);
170        if (err != NULL) {
171            error_propagate(errp, err);
172            return;
173        }
174        busdev = SYS_BUS_DEVICE(dev);
175        sysbus_mmio_map(busdev, 0, wdg_addr);
176        sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, wdg_irq));
177        /* Attach PDB */
178        dev = DEVICE(&s->pdb);
179        object_property_set_bool(OBJECT(&s->pdb), true, "realized", &err);
180        if (err != NULL) {
181            error_propagate(errp, err);
182            return;
183        }
184        busdev = SYS_BUS_DEVICE(dev);
185        sysbus_mmio_map(busdev, 0, pdb_addr);
186        sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, pdb_irq));
187        /* Attach LPIT */
188        dev = DEVICE(&s->lpit);
189        object_property_set_bool(OBJECT(&s->lpit), true, "realized", &err);
190        if (err != NULL) {
191            error_propagate(errp, err);
192            return;
193        }
194        busdev = SYS_BUS_DEVICE(dev);
195        sysbus_mmio_map(busdev, 0, lpit_addr);
196        sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, lpit_irq));
197        /* Attach FTM */
198        dev = DEVICE(&s->ftm0);
```

```
199     object_property_set_bool(OBJECT(&s->ftm0), true, "realized", &err);
200     if (err != NULL) {
201         error_propagate(errp, err);
202         return;
203     }
204     busdev = SYS_BUS_DEVICE(dev);
205     sysbus_mmio_map(busdev, 0, ftm_addr[0]);
206     sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, ftm_irq[0]));
207     /* Attach CAN */
208     dev = DEVICE(&s->can);
209     object_property_set_bool(OBJECT(&s->can), true, "realized", &err);
210     if (err != NULL) {
211         error_propagate(errp, err);
212         return;
213     }
214     busdev = SYS_BUS_DEVICE(dev);
215     sysbus_mmio_map(busdev, 0, can_addr);
216     sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(armv7m, can_irq));
217     /* Connect PORT interrupts */
218     object_property_set_int(OBJECT(s->port_irqs), S32K_NUM_PORTS,
219                             "num-lines", &err);
220     object_property_set_bool(OBJECT(s->port_irqs), true, "realized", &err);
221     if (err != NULL) {
222         error_propagate(errp, err);
223         return;
224     }
225     qdev_connect_gpio_out(DEVICE(s->port_irqs), 0,
226                           qdev_get_gpio_in(armv7m, port_irq));
227     /* Attach PORTS */
228     for (i = 0; i < S32K_NUM_PORTS; i++) {
229         dev = DEVICE(&(s->port[i]));
230         object_property_set_bool(OBJECT(&s->port[i]), true, "realized", &
    err);
231         if (err != NULL) {
232             error_propagate(errp, err);
233             return;
234         }
```

```
235         busdev = SYS_BUS_DEVICE(dev);
236         sysbus_mmio_map(busdev, 0, port_addr[i]);
237     }
238     /* Attach GPIOS */
239     dev = DEVICE(&s->pta);
240     object_property_set_bool(OBJECT(&s->pta), true, "realized", &err);
241     if (err != NULL) {
242         error_propagate(errp, err);
243         return;
244     }
245     busdev = SYS_BUS_DEVICE(dev);
246     sysbus_mmio_map(busdev, 0, gpio_addr[0]);
247     sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(DEVICE(s->port_irqs), 0)
    );
248     // PTB
249     dev = DEVICE(&s->ptb);
250     object_property_set_bool(OBJECT(&s->ptb), true, "realized", &err);
251     if (err != NULL) {
252         error_propagate(errp, err);
253         return;
254     }
255     busdev = SYS_BUS_DEVICE(dev);
256     sysbus_mmio_map(busdev, 0, gpio_addr[1]);
257     sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(DEVICE(s->port_irqs), 1)
    );
258     // PTC
259     dev = DEVICE(&s->ptc);
260     object_property_set_bool(OBJECT(&s->ptc), true, "realized", &err);
261     if (err != NULL) {
262         error_propagate(errp, err);
263         return;
264     }
265     busdev = SYS_BUS_DEVICE(dev);
266     sysbus_mmio_map(busdev, 0, gpio_addr[2]);
267     sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(DEVICE(s->port_irqs), 2)
    );
268     // PTD
```

```
269     dev = DEVICE(&s->ptd);
270     object_property_set_bool(OBJECT(&s->ptd), true, "realized", &err);
271     if (err != NULL) {
272         error_propagate(errp, err);
273         return;
274     }
275     busdev = SYS_BUS_DEVICE(dev);
276     sysbus_mmio_map(busdev, 0, gpio_addr[3]);
277     sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(DEVICE(s->port_irqs), 3)
    );
278     // PTE
279     dev = DEVICE(&s->pte);
280     object_property_set_bool(OBJECT(&s->pte), true, "realized", &err);
281     if (err != NULL) {
282         error_propagate(errp, err);
283         return;
284     }
285     busdev = SYS_BUS_DEVICE(dev);
286     sysbus_mmio_map(busdev, 0, gpio_addr[4]);
287     sysbus_connect_irq(busdev, 0, qdev_get_gpio_in(DEVICE(s->port_irqs), 4)
    );
288 }
289
290 static Property s32k116_soc_properties[] = {
291     DEFINE_PROP_STRING("cpu-type", S32K116State, cpu_type),
292     DEFINE_PROP_END_OF_LIST(),
293 };
294
295 static void s32k116_soc_class_init(ObjectClass *klass, void *data)
296 {
297     DeviceClass *dc = DEVICE_CLASS(klass);
298
299     dc->realize = s32k116_soc_realize;
300     dc->props = s32k116_soc_properties;
301 }
302
303 static const TypeInfo s32k116_soc_info = {
```

```
304     .name          = TYPE_S32K116_SOC,
305     .parent        = TYPE_SYS_BUS_DEVICE,
306     .instance_size = sizeof(S32K116State),
307     .instance_init = s32k116_soc_initfn,
308     .class_init    = s32k116_soc_class_init,
309 };
310
311 static void s32k116_soc_types(void)
312 {
313     type_register_static(&s32k116_soc_info);
314 }
315
316 type_init(s32k116_soc_types)
```

Listing D.2: S32K116 Machine

D.5 Makefile changes

Under the ARM hardware Makefile:

```
1     ...
2     obj-$(CONFIG_NETDUINO2) += netduino2.o
3     obj-y += s32k116evb.o
4     obj-$(CONFIG_NSERIES) += nseries.o
5     ...
6     obj-y += s32k116_soc.o
```

Under the ARM devices Makefile:

```
1     # ARM devices
2     ...
3     common-obj-$(CONFIG_ARM11SCU) += arm11scu.o
4     obj-y += s32k_smc.o
5     obj-y += s32k_sim.o
6     obj-y += s32k_rcm.o
7     obj-y += s32k_pcc.o
8     obj-y += s32k_scg.o
9     obj-y += s32k_port.o
10    obj-y += s32k_gpio.o
11    obj-y += s32k_can.o
```

```
12    obj-y += s32k_wdg.o
13    obj-y += s32k_adc.o
14    obj-y += s32k_lpuart.o
15    obj-y += s32k_pdb.o
16    obj-y += s32k_lpit.o
17    obj-y += s32k_ftm.o
18    ...
```

D.6 Full Command Line Arguments

```
qemu-system-arm -M s32k116evb -kernel SAS.elf \
-nodefaults \
-serial stdio \
-nographic \
-D qemu-log-sas.log \
-icount 5 \
-sb can=8888,gpio=8887 \
-sync 7777,50016 \
-fi 7776,fault_lib_1.xml,10000000,10000000,CurrentState \
-simfile sim_file_b1.txt \
-watchdogfile wdg_b1.txt
```

Appendix E

Monte Carlo Simulations

E.1 Component Mean Time Between Failure Values

Table E.1: Component Mean Time Between Failure Values

Component	MTBF (hours)
Microcontroller	4.2×10^5
CAN	1.7×10^5
Digital Isolator	2.5×10^5
Clock	1.6×10^5
Power	5.3×10^5
Sensor	2×10^5

E.2 Fault Coordinator Python Script

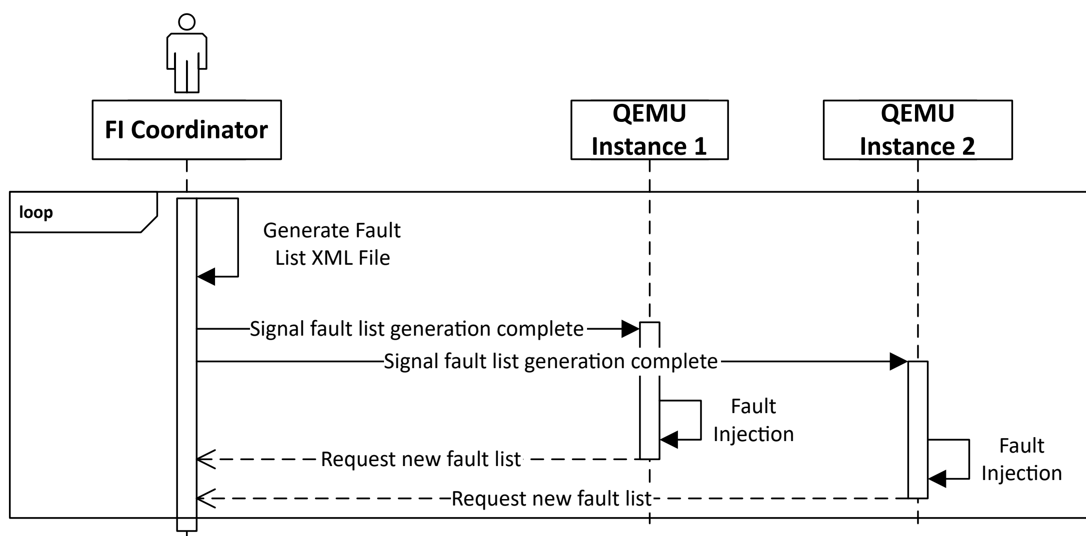


Figure E.1: Fault Coordinator sequence diagram

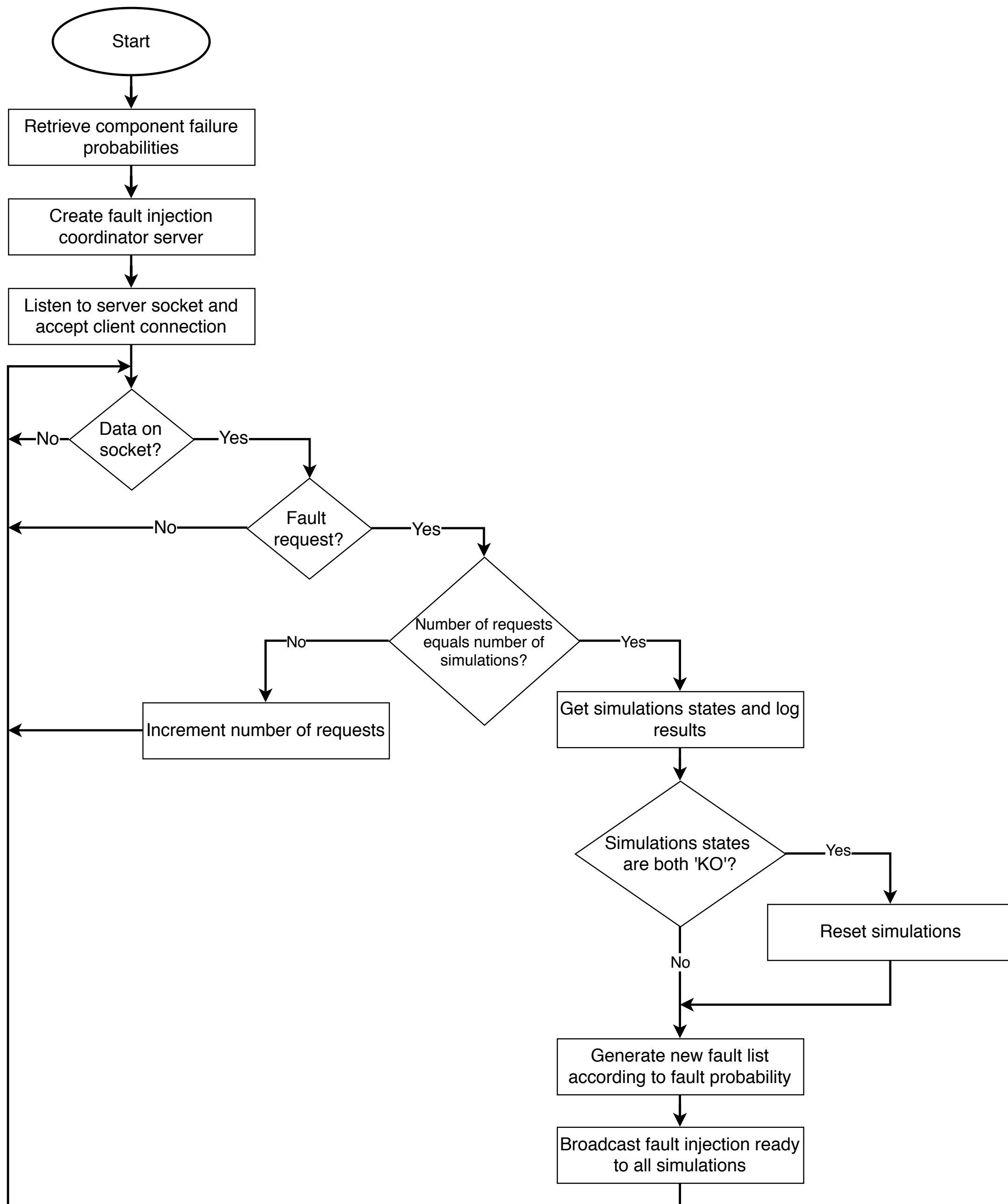


Figure E.2: Fault Coordinator flowchart