Mario Saric, BSc
Matr. Nr. 01428305

Simulation-based Testing of Failsafe Industrial Peripheral Modules

Diploma Thesis
Master's degree programme: Energy Systems and Automation Technology

submitted to

Vienna University of Technology

Supervisor
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Thilo Sauter

Institute of Computer Technology

Vienna, March 2019

Eidesstattliche Erklärung

Acknowledgment

This thesis was written in SIEMENS AG Development House in Graz, which is why I would like express my gratitude to Herbert Tanner for this opportunity and all the resources and patience I got during the writing process. A very special thanks to my colleague Joachim Wahrbichler for the generous technical support and knowledge he shared with me. I would also like to thank my colleague, Nermin Kajtazović, for his support, suggestions and discussions during this time. By all means, I would like to thank my supervisor Prof. Thilo Sauter for his professional and academic support.

Finally, I can not express how much I am grateful to my family: my father Ivan, mother Jadranka and sisters Magdalena and Marijana. It is for their support and sacrifice that I was able to do my studies in Croatia and Austria. They were always there for me, day or night. Thank you!

Abstract

Nowadays, programmable logic controllers (PLCs) are a common technology used for automating industry processes and plants. Their scope of application ranges from simple emergency stop systems to more advanced systems, such as those used in hydro power plants for monitoring turbine rotation. Some of these PLCs and their corresponding peripheral I/O devices are used in the so-called "critical" environments, where in case of failures such systems may pose harm to humans or cause damage to the equipment. Thus, these so-called safety-critical systems must be developed rigorously with a high degree of quality assurance. If such system detects a failure in hardware or software it automatically goes into safe state, usually by de-energizing the outputs of the peripheral I/O modules. To ensure functional safety, the modules are developed in accordance with the IEC 61508 standard. For the software part, the use of the V-model is highly recommended by the IEC 61508. One of the common issues in terms of verification is a large gap between module/unit tests, which are normally performed in software for each software module separately, and hardware/software integration tests, which are performed when the system is integrated and functional. In order to bridge this gap, intermediate levels of integration are required between these two test phases. Simulation-based hardware-software testing is therefore presented in this thesis. Except for a host machine, this approach does not require any hardware. Instead, simulation-based testing is conducted using simulated hardware and an instruction set simulator. The simulation-based components are integrated into the legacy test environment. The tests are compiled and run redundantly on two different tool-chains, resulting in a test report and a coverage report. The new method is applied in a specific use-case: a traditional hardware-software integration test case is implemented in the simulation. The results show that it is possible to execute traditional integration tests without the hardware prototype and obtain a test coverage overview in addition. Consequently, the new method proves that it can be used as a supplement to traditional integration tests.

Kurzfassung

Speicherprogrammierbare Steuerungen (SPS) sind heutzutage eine weit verbreitete Technologie in der industriellen Prozessautomatisierung. Ihr Anwendungsgebiet reicht von simplen Not-Aus Systemen bis hin zu sehr komplexen Anlagen, wie z.B. die Überwachung der Turbinen in einem Wasserkraftwerk. Manche dieser SPS (samt den zugehörigen E/A Peripheriemodulen) werden in sogenannten „kritischen“ Umgebungen eingesetzt, wo der Ausfall dieser Systeme eine Gefahr für Menschen bedeutet sowie ökonomische anrichten kann. Aus diesem Grund besitzen sogenannte sicherheitskritischen Systeme eine sehr rigorose Entwicklung und hohe Qualitätsansprüche. Wenn das System einen Fehler in Hardware und/oder Software detektiert, so schaltet es automatisch in den sicheren Zustand, der üblicherweise das Abschalten der Peripherieausgänge zur Folge hat. Um funktionale Sicherheit zu gewährleisten werden die Module gemäß IEC 61508 Standard entwickelt. Für die Software empfiehlt der Standard die Verwendung des V-Modells. Eines der Probleme bei der Verifikation ist, dass eine große Lücke zwischen Modultests, welche in Software ausgeführt werden und jede Komponente einzeln betrachten, und den Hardware/Software-Integrationstests welche im Systemverband getestet werden besteht. Um diese Lücke zu schließen sind Zwischenstufen im Test nötig. Aus diesem Grund wird eine simulationsbasierte Hardware-Software Testmethode in dieser Arbeit vorgestellt. Dieser Ansatz benötigt außer einem Hostsystem keine zusätzliche Hardware. Stattdessen wird beim Testen nur simulierte Hardware mit einem Befehlssatz-Simulator eingesetzt. Die simulationsbasierten Komponenten wurden in die bestehende Testumgebung integriert. Die Tests werden übersetzt und laufen redundant auf zwei verschiedenen Tool-chains und resultieren in einem Testbericht und einem Codeabdeckungsbericht. Die neue Methode wurde an einem bestehenden Use-Case; einem gewöhnlichen HW/SW-Integrationstest angewendet. Das Ergebnis zeigte, dass es möglich ist gewöhnliche Integrationstestfälle ohne Hardwareprototyp auszuführen und zusätzlich eine Übersicht über die Codeabdeckung zu erhalten. In Folge dessen wird die neue Methode ergänzend zu den bestehenden Integrationstests verwendet.

Contents

Abstract **iv**

Abbreviations **xi**

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Thesis Structure . . . 4

2. Failsafe Programmable Logic Controllers **5**
- 2.1. SIMATIC PLC System . . . 5
- 2.2. Failsafe PLC Peripheral Module . . . 6
- 2.3. PROFIsafe . . . 9

3. Testing Embedded Systems **12**
- 3.1. Safety V-Model Life-cycle . . . 13
- 3.2. Testing Techniques . . . 15
  - 3.2.1. Black-box Testing . . . 15
  - 3.2.2. White-box Testing . . . 17
- 3.3. Test Levels . . . 20
  - 3.3.1. Unit Test . . . 20
  - 3.3.2. SW Integration Test . . . 20
  - 3.3.3. HW/SW Integration Test . . . 22
  - 3.3.4. Validation Testing . . . 22
- 3.4. Simulation Techniques in Testing . . . 23
- 3.5. Conclusion . . . 24

4. Modeling Embedded Systems **26**
- 4.1. Abstraction Levels . . . 26
- 4.2. Abstraction Pyramid . . . 27
- 4.3. System Level Modeling . . . 29
  - 4.3.1. Single-Language Approach . . . 29

4.3.2. Multi-Language Approach . . . 30
4.4. Existing System-Level Modeling Languages . . . 30
4.4.1. SpecC . . . 31
4.4.2. SystemC . . . 31
4.4.3. C/C++ Modeling . . . 33
4.5. System-Level Modeling Platforms . . . 34
4.5.1. Commercial Tools . . . 34
4.5.2. OVPSim . . . 34
4.5.3. QEMU . . . 35
4.6. Modeling Timing Behavior . . . 35
4.7. Conclusion . . . 36

5. System Design 39
5.1. Simulated Hardware . . . 39
5.1.1. Generic I/O Pin Model . . . 39
5.1.2. Generic Register Model . . . 42
5.1.3. ARM Cortex M3 . . . 42
5.1.4. STM32F2xx Peripherals . . . 42
5.1.5. Timer . . . 43
5.1.6. Other Hardware . . . 43
5.2. Firmware . . . 46
5.3. Middleware . . . 48
5.3.1. F-CPU and PROFIsafe Simulation . . . 48
5.3.2. Simulation Event Manager . . . 48
5.4. Test Environment . . . 50

6. Implementation 54
6.1. PROFIsafe Address Assignment . . . 54
6.1.1. Manual Setting Using a Hardware Switch . . . 55
6.1.2. Address Assignment via Engineering Station . . . 55
6.1.3. HW-SW Integration Test Cases . . . 56
6.2. Virtual Address Assignment . . . 61
6.3. Improving Test Coverage . . . 64

7. Results and Evaluation 66
7.1. Test Coverage . . . 66
7.2. Test Resource Savings . . . 68

7.3. Discussion 68

8. Conclusion **70**

8.1. Summary 70

8.2. Future Work 71

Bibliography **72**

Appendix **78**

A. STM32F2xx Block Diagram **79**

List of Figures

1.1. Typical V-model [3] 2
1.2. Traditional development (top) vs. development using a simulation platform (bottom) [5] 3

2.1. SIMATIC PLC system: S7-1500 CPU with ET 200SP (Image courtesy of SIEMENS) 6
2.2. PLC program execution 7
2.3. Generic failsafe I/O peripheral module block diagram 8
2.4. 1oo2D architecture [7] 8
2.5. PROFIsafe "black channel" concept [8] 10
2.6. PROFIsafe layer structure [8] 10
2.7. PROFIsafe message format [8] 11

3.1. Profit vs. verification effort in software testing [9] 13
3.2. Cost vs. development time in software testing [10] 13
3.3. Safety V-model [4] 14
3.4. Black-box testing 16
3.5. Equivalence class partition 16
3.6. Boundary value analysis 17
3.7. Integration test strategy example [15] 21
3.8. Top down integration strategy [15] 21
3.9. Bottom up integration strategy [15] 22
3.10. Simulation techniques in the development of modern ECUs [17] 24
3.11. SiL simulation diagram 25

4.1. Y-chart diagram [19] 27
4.2. Abstraction pyramid [20] 28
4.3. Single-language approach [21] 29
4.4. Multi-language approach [21] 30
4.5. Basic structure of a SpecC model [23] 32
4.6. SystemC language architecture [26] 33

4.7. Level of abstraction vs. performance in modeling timing behavior [46] . . . 37

5.1. Complete system overview . . . 40
5.2. Pins class diagram . . . 41
5.3. Generic register class diagram . . . 42
5.4. Modified device header file: a) Physical processor accessing hardware perihperals b) Accessing modeled peripherals using the modified header file . . . 44
5.5. Timer block diagram [50] . . . 45
5.6. Timer class diagram . . . 45
5.7. Firmware architecture . . . 47
5.8. Simulated bus interface . . . 49
5.9. Code instrumentation in module controller . . . 50
5.10. Simulation event manager time sequence diagram . . . 51
5.11. Test environment configuration . . . 52

6.1. Setting the failsafe address using a hardware switch . . . 55
6.2. Implementation configuration in TIA Portal . . . 56
6.3. Step 1 of the address assignment in TIA Portal . . . 57
6.4. Step 2 of the address assignment in TIA Portal . . . 58
6.5. Step 3 of the address assignment in TIA Portal . . . 58
6.6. TIA Portal address assignment time sequence diagram . . . 59
6.7. Activity diagram for the simulation-based address assignment . 62
6.8. Simulation-based address assignment time sequence diagram . . 63
6.9. Additional test cases derived from data records . . . 65

Abbreviations

1oo2D	One-Out-Of-Two-Diagnostic
ADC	Analog-to-Digial Converter
BCET	Best Case Execution Time
CRC	Cyclic Redundancy Check
DIP	Dual-In-line-Package
DUT	Device Under Test
ECU	Engine Control Unit
ESL	Electronic System Level
F-CPU	Failsafe CPU
FMEA	Failure Mode and Effects Analysis
HDL	Hardware Description Language
HiL	Hardware-in-the-Loop
I2C	Inter-Integrated Circuit
IP	Intellectual Property
ISS	Instruction Set Simulator
MiL	Model-in-the-Loop
OVP	Open Virtual Platforms
PiL	Processor-in-the-Loop

Abbreviations

PL	Performance Level
PLC	Programmable Logic Controller
SIL	Safety Integrity Level
SiL	Simulation-in-the-Loop
SLDL	System-Level Design Language
TLM	Transaction-Level Modeling
WCET	Worst Case Execution Time

1. Introduction

Nowadays, programmable logic controllers (PLCs) are the most widely used industrial automation technology [1]. They are used for production and process control in various industrial systems. Their application ranges from simple automation tasks such as conveyor systems to complex control systems such as nuclear plants. The latter is considered a safety-critical environment. Failure in such environments can cause damage to the property or environment, which in turn, either directly or indirectly, leads to physical injury or damage to the human health [2]. Moreover, the failure can also induce serious economic consequences. The root of the failure can be caused by any functional part of the system, including the programmable logic controllers. Given the fact, that the PLCs are becoming more and more complex in terms of hardware and software, a number of guidelines and standards have been introduced lately, in order to ensure functional safety and make them as reliable as possible. In the context of industrial application, the international standard IEC 61508 has emerged [2]. IEC 61508 covers the system development of failsafe programmable electronic systems. It has seven parts that are focused on different development aspects and features a set of methods and recommendations to support the development and ensure functional safety.

1.1. Motivation

This thesis focuses on the software part of the development of failsafe PLC peripheral modules. This is covered in the Part 3 of the IEC 61508 [4]. The software design is largely based on the use of the V-model approach. A typical V-model is shown in the Figure 1.1. The principle of the V-model is a top-down approach for development and testing. Each development level has a corresponding test level. The focus of this thesis is further narrowed to the right side of the V-model: testing and verification. Two major problems arise from testing the software with using the V-model approach. One is that software

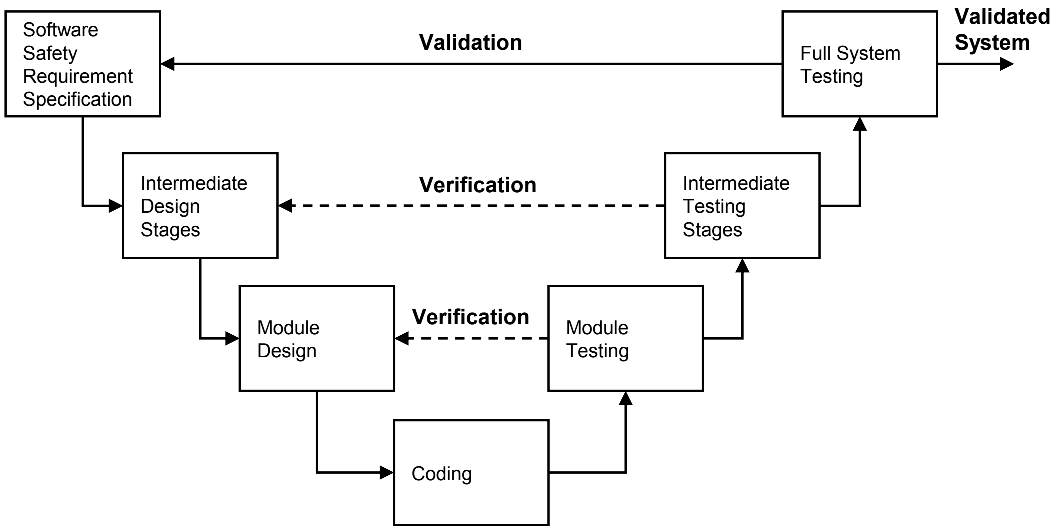


Figure 1.1.: Typical V-model [3]

testing heavily depends on hardware architecture and the other is that there is a large gap between module/unit tests, which are normally conducted in software for each software module in isolation, and hardware/software integration tests, which are conducted when a system is integrated. To be more appropriate, an intermediate levels of integration are required between these two test phases, since on the one side, unit testing can help to identify the potential systematic faults with reasonably good coverage in isolated software modules, and on the other side, hardware/software integration allows to verify functions of the integrated system, but lacks in achieving high test coverage.

Hardware-dependent Software Testing

Software development can be conducted as detached from the hardware only to some extent. It is feasible to develop and test software components with little or no interaction with the hardware. However, when developing software modules which closely interact with the hardware (e.g., drivers), a hardware prototype is essential. Ideally, one would be completely independent from hardware development and have an executable software without the need of hardware. A potential solution to this would be to introduce virtual hardware components by introducing a simulation-based platform. With this new platform, the software developer is not only able to execute hardware-dependent test cases before

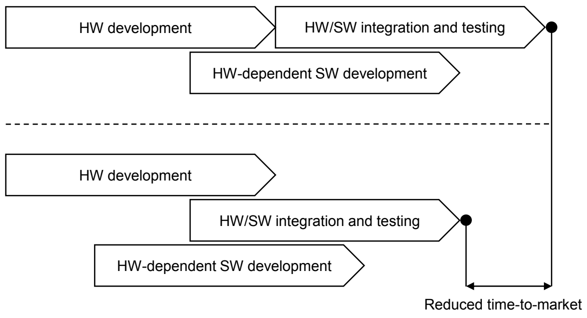

HW development
HW/SW integration and testing
HW-dependent SW development
HW development
HW/SW integration and testing
HW-dependent SW development
Reduced time-to-market

Figure 1.2.: Traditional development (top) vs. development using a simulation platform (bottom) [5]

actually getting the hardware, but also to use it to develop and test new functionalities. This leads to a reduced time-to-market, as illustrated in Figure 1.2.

Integration Testing Gap

In the most optimal scenario, the software should be fairly tested. However, there are two problems with software testing. First, it is difficult to determine what it means that the system is completely tested. Second, even if the exact determination would be possible, there is still only a limited number of resources for testing. IEC 61508 partially handled that problem and introduced a formal metric to measure to which degree the system has been tested. This metric is called test coverage. Ideally, 100% of statements, methods and branches shall be covered. If that is not possible, a valid explanation has to be supplemented. Furthermore, the statement about the code coverage is relatively easy to generate for isolated software units and software integration tests, since there is a transparent overview over the software architecture at these levels. However, when moving to hardware-software integration test, the source code is no longer available and thus, there is no feedback from the software part. Hardware-software integration tests are used for functional verification but they lack an overview about the code coverage. This thesis introduces a simulation-based hardware-software integration test approach, which provides a code-coverage overview for traditional hardware-software integrations tests.

Thesis Contributions

To tackle the problems above, this thesis will focus on introducing a simulation-based hardware-software testing platform. In accordance with the IEC 61508 and based on the traditional software development using a V-model approach, a simulation-based testing approach is introduced for testing failsafe peripheral modules. The platform will not have any hardware components. The goal is to execute traditional hardware-software integration test cases on the simulation platform.

1.2. Thesis Structure

The thesis is organized in eight chapters. After this introductory chapter, subsequent three chapters deal with theoretical background, literature review and related work. Chapter 2 introduces the SIEMENS SIMATIC PLC system to the reader. The presented peripheral module is a safety-critical embedded system. In order to get the target system certified, it has to be comprehensively tested. Chapter 3 gives a brief overview of testing methods used for embedded systems in general, as well as those specifically used for the target system. One of the testing techniques applied is also software simulation. To implement a simulation in the testing life-cycle, a software model of the embedded system is necessary. Chapter 4 focuses on modeling an embedded system on a system level. System-modeling state-of-the art is presented and backed up with the related work. After providing theoretical fundamentals, Chapter 5 describes the final system. In Chapter 6, the system is implemented in a specific use-case. The results of the implementation and detailed evaluation of the system are presented in the Chapter 7. Finally, Chapter 8 offers a brief summary of the thesis after which some future work is presented.

2. Failsafe Programmable Logic Controllers

2.1. SIMATIC PLC System

A brief description on how a PLC works is explained through an example of the SIEMENS SIMATIC PLC system depicted in the Figure 2.1. The CPU (1) and the peripheral I/O modules (2) are mounted on a mounting rail (6) and supplied with a supporting power supply. Optionally an interface module (5) is used in a distributed system to connect the distributed I/O group with the CPU. Automation application runs on either a regular CPU or a failsafe CPU (F-CPU). The system supports both Ethernet, PROFINET and PROFIBUS communication. The data between CPU and peripheral modules is exchanged cyclically and acyclically. Cyclic data exchange is used for process data (e.g. input data from sensors) and acyclic for other communication (e.g. parametrization of the module). For exchanging the safety relevant process data, the PROFIsafe protocol is applied. Peripheral I/O modules form an interface between the controller and the plant. They handle physical input or output signals obtained from sensors and provided to actuators. Depending on the signal type, I/O modules can be generally divided into following categories:

- Digital input (DI)
- Digital output (DQ)
- Analog input (AI)
- Analog output (AQ)

For safety-critical plants, the yellow-labeled modules (3) in Figure 2.1, are used. These are a special SIMATIC I/O module product family for safety integration.

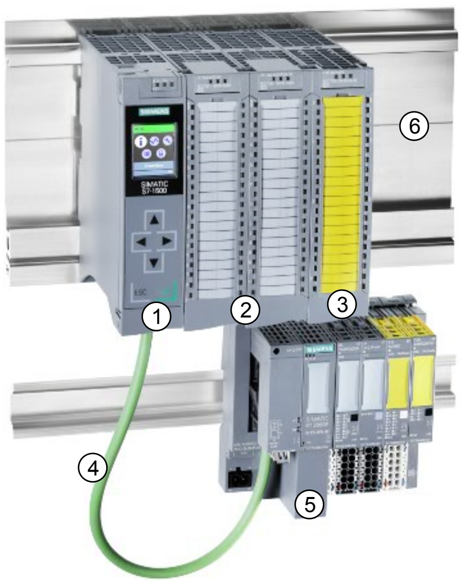


Figure 2.1.: SIMATIC PLC system: S7-1500 CPU with ET 200SP (Image courtesy of SIEMENS)

PLC Program Cycle

The main application is implemented as an infinite loop with instructions executed in a specified order and interrupted with time-critical tasks in form of interrupts. This is illustrated in Figure 2.2. Each cycle starts by reading the input values from sensors and ends with setting the output data for actuators. In-between, instructions are executed to provide functionality of the loaded application. Total cycle time depends on the complexity of the application and performance of the used CPU. However, the cycle time must be short enough to ensure that the system operates almost in real time. Typical time values range from $1ms$ to $20ms$ [1].

2.2. Failsafe PLC Peripheral Module

Fail-safe modules cover a number of functional safety measures from various international standards (e.g. IEC 61508). Furthermore, the modules have to get certified by an accredited functional safety certification organization (e.g. TÜV). They are designed for safety-related use up to SIL (Safety Integrity

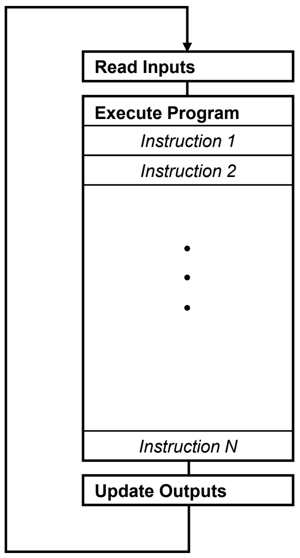


Figure 2.2.: PLC program execution

Level) 3 according to IEC 62061 and PL (Performance Level) "e" according to ISO 13849 [6].

The general internal block structure of a failsafe module is shown in Figure 2.3. The module is connected to the F-CPU via bus interface. On the other side, sensors/actuators in the plant are connected via different I/O channels. The data is redundantly processed on two identical micro-controllers, which run the same firmware images and are constantly being synchronized. Each module has a group of module-specific hardware. The peripheral analog modules, for example, feature a third-party ADC hardware component.

Functional Safety Measures

The failsafe peripheral module essentially realizes functionality as regular modules do: exchange process data between the plant and controller. However, to ensure failsafe operation, several functional safety measures according to the IEC 61508 are integrated into the failsafe modules. Most important is the use of one-out-of-two-diagnostic (1oo2D) architecture. This means that the process data is evaluated using two identical microprocessors running the same firmware images. Supplementary, a number of diagnostic monitoring measures are implemented.

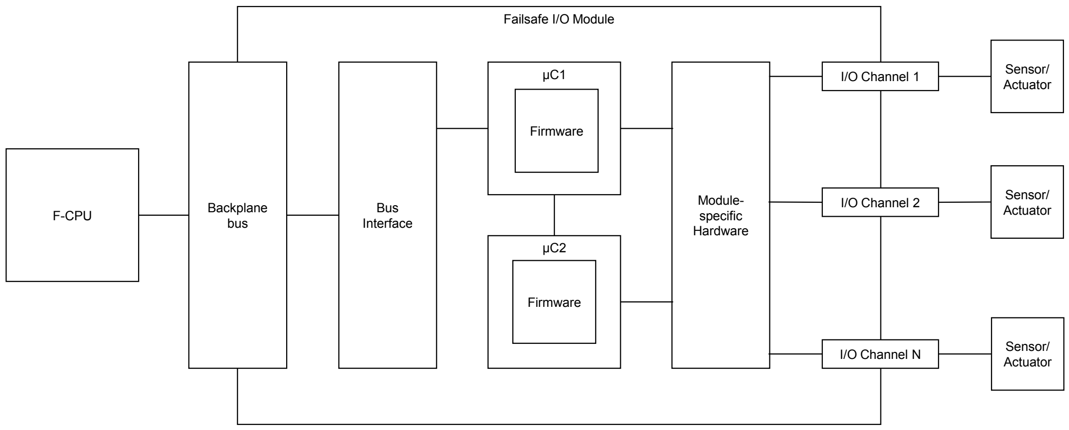


Figure 2.3.: Generic failsafe I/O peripheral module block diagram

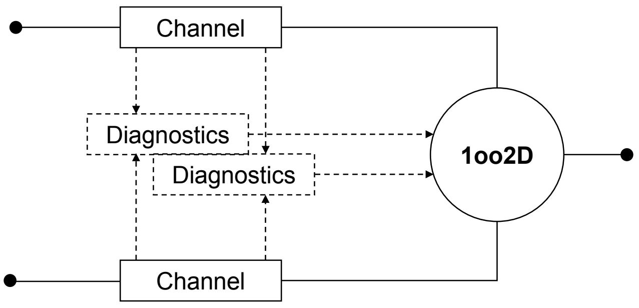


Figure 2.4.: 1oo2D architecture [7]

1oo2D Architecture

The sensor values are processed and evaluated using a dual channel structure, depicted in Figure 2.4. The channels are connected in parallel. If there is a diagnostic fault in one of channels or a discrepancy between the two of them, then the module goes to the safe state [7].

Diagnostic Monitoring

As recommended by IEC 61508, a number of monitoring functions are implemented to ensure the correct operation of the peripheral modules. Examples of such monitoring functions are voltage monitoring, temperature monitoring, RAM Comparison, watchdog monitoring etc. If one of the monitored values does not match the reference or expected values, the module goes into the failsafe state.

2.3. PROFIsafe

PROFIsafe [8] is a safety communication technology defined as an international standard in IEC 61784-3-3. PROFIsafe protocol can be used for safety applications up to SIL3 according to IEC 61508. It allows a failsafe communication on existing standard networks such as PROFIBUS and PROFINET without any impacts on them. Both standard and failsafe messages are transmitted on the same cables. However, from the perspective of a PROFIsafe profile, the underlying bus system and other network components (switches and routers i.e.) are seen as "black channel". This concept is shown in Figure 2.5. A safety communication profile must deliver updated and correct data (data integrity) to the intended destination (authenticity) just-in-time (timeliness). To meet these tasks, [8] includes following safety measures:

1. Consecutive numbering of PROFIsafe messages: a 24-bit consecutive number is used to assure that the receiver obtained the complete message within the correct sequence.
2. Time expectation with acknowledgment: in the safety critical systems, it not only matters if the received message is complete as mentioned above, but it is just as important that the message arrives within a fault tolerant time. This is solved by utilizing a watchdog timer.
3. Codename between sender and receiver: to avoid misdirecting messages, both sender and receiver have a unique network-wide identification in a form of failsafe address.
4. Data integrity checks (CRC): A cyclic redundancy check is used for detecting corrupted bits. PROFIsafe uses 24-bit and 32-bit CRC generator polynomial to calculate the CRC signatures.

PROFIsafe services

As previously shown in Figure 2.5, the PROFIsafe layers are located above the "black channel" and are implemented in software as drivers with a central state machine controlling message processing, CRC error handling and exceptions such as startup and power on/off. Figure 2.6 shows the service interaction between the host (e.g. F-CPU) and the device (peripheral module).

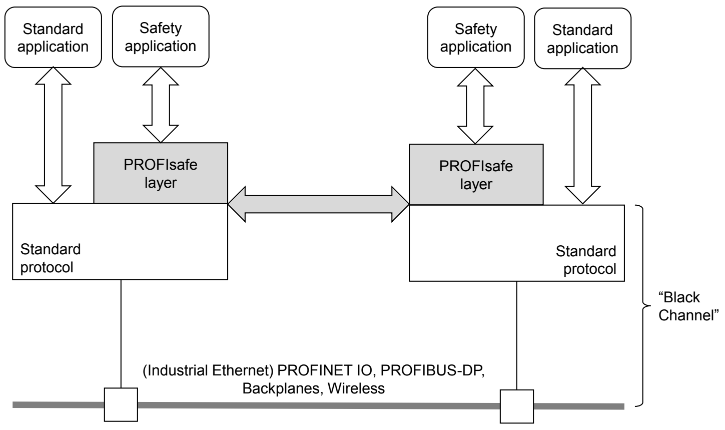


Figure 2.5.: PROFIsafe "black channel" concept [8]

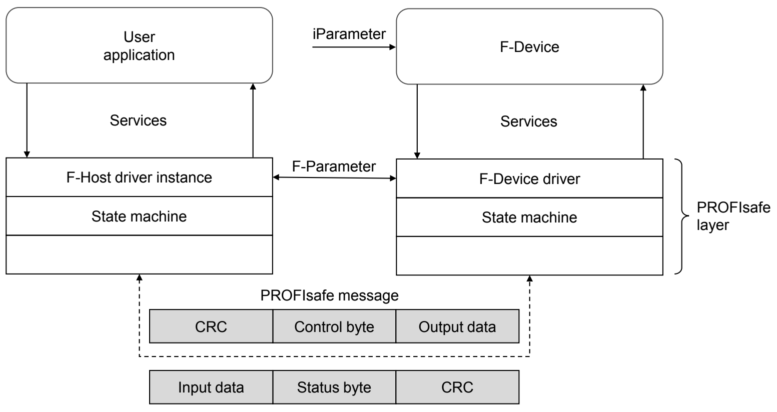


Figure 2.6.: PROFIsafe layer structure [8]

F-Input/Output data	Status/Control byte	CRC signature
Maximum of 12 or 123 bytes	1 byte	3 or 4 bytes

Figure 2.7.: PROFIsafe message format [8]

Host Services

The main services implement the exchange of input/output process data. In case of errors, the process values are replaced by failsafe values which are by default zero in order to force the receiver in safe state. There are devices in which the de-energize is not the only possible state. Instead, an alternative state during which the device is put to low speed is also possible. This is also covered by the host services. After the devices switches to the safe state, it is usually not allowed to return to normal operation without human interaction. PROFIsafe provides additional service for operator acknowledgment.

Device Services

The PROFIsafe services for the device technology cover the same aspects mentioned above, but from the perspective of the device itself. Beside services for reporting faults, a special service is used for passing over the diagnostic information. Last but not least, the device uses PROFIsafe services for parametrization. There are two groups of parameters: I-parameters which are technology-specific device parameters and F-parameters containing information for the PROFIsafe layer (i.e. failsafe address).

PROFIsafe Message Format

Figure 2.7 shows the default PROFIsafe message format. The main part is the input/output data. Factory automation and process automation have different requirements: one deals with short signals, the other involve longer process values. Consequently, there are two different lengths of data structure: one limited to maximum of 12 bytes with 3-byte CRC signature and the other limited to 123 bytes requiring a 4-bye CRC signature. The CRC signature ends the PROFIsafe data message. The above mentioned consecutive number is not transmitted within the message. Instead, the counter values of sender/receiver and synchronized via the control/status byte.

3. Testing Embedded Systems

Previous chapter introduced the failsafe PLC system. The failsafe peripheral module is basically an embedded system. Testing embedded systems covers testing not just the software the hardware as well. For the modules to go to production, they need to be developed in accordance to the international functional safety standard IEC-61508 and pass a national certification process. However, this thesis solely focuses on th software part. As mentioned before, these modules are usually used in safety critical industrial systems. A failure or malfunction in the PLC module could potentially have catastrophic consequences for the environment and/or the people. In the best-case scenario, a failure would result in the temporary shut down of the production, which would nevertheless cause massive financial losses and bad reputation for the manufacturer of the automation system. The best solution therefore is to reduce the number of potential failures to a bare minimum. This is archived by thoroughly testing the system including the embedded software. Even though it is practically impossible to release a fault-free software, by applying a systematic testing approach it is possible to discover and eliminate as many faults as possible. This approach requires a lot of resources. The relation between the profit and the invested verification effort is shown in Figure 3.1. If insufficient resources are allocated for the testing and verification, the system is much more likely to fail in operation and, as a result, cause immense profit loss. On the other hand, if the system is tested to extensive measures, it may result in a better, more fault-free product but the overall profitability is questionable. Additionally, Figure 3.2 shows that a failure is likely to cost more if discovered in later development stages. Testing and verification of the software is as important as developing. It is a complex, repetitive and resource-consuming process. This chapter is used as a general introduction to embedded software testing based on the V-model according to IEC 61508.

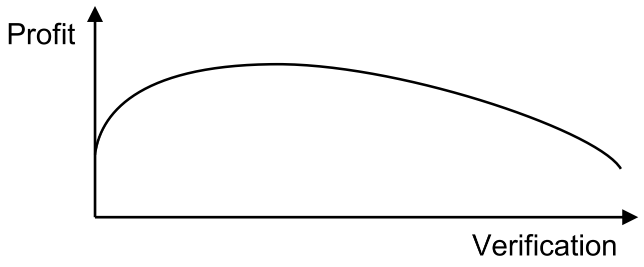


Figure 3.1.: Profit vs. verification effort in software testing [9]

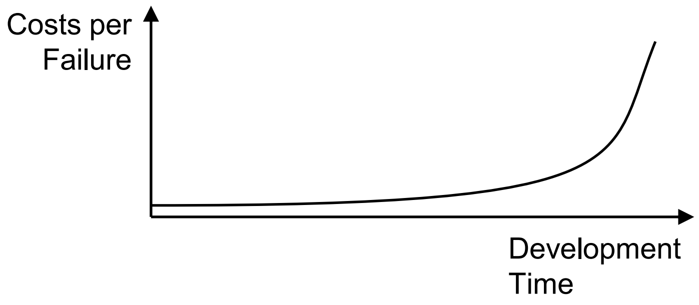


Figure 3.2.: Cost vs. development time in software testing [10]

3.1. Safety V-Model Life-cycle

Software testing and verification are parts of the general software development process. Software life-cycle models are used for a more structured and methodical approach on the software development. The main idea is to break down the whole software development process into multiple and distinct phases. There are numerous software life-cycle models nowadays. The failsafe peripheral embedded software is developed based on the V-model recommended by the IEC 61508-3 [4] and depicted in Figure 3.3. Compared to a standard V-model life-cycle used in standard software development, the V-model recommended by IEC 61508 additionally handles safety-critical requirements derived from the general system safety requirements specification. The phases of this model form a shape in the form of “V”, which explains its name. Development phases are placed at the left side of the “V”. Each development phase has an associated testing phase on the opposite side. The goal of each testing phase is to verify the opposite development phase. The abstraction level is represented by the horizontal axis while the vertical axis represents the total time of the development. The

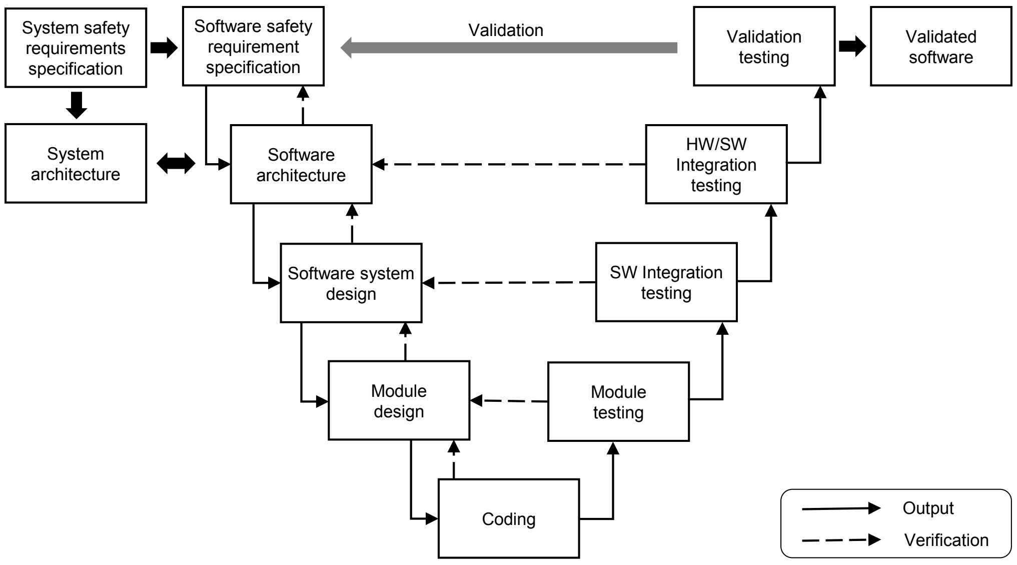


Figure 3.3.: Safety V-model [4]

V-model includes following phases:

- **Software safety requirement specification:** The objective here is to specify the requirements for safety-related software. These requirements are derived from the general system safety requirements specification,
- **Software architecture:** Software architecture, derived from the general system architecture is created to fulfill the above specified safety-related requirements.
- **Software system design:** Technical implementation of the software architecture. A set of tools, languages, compilers, user interfaces etc. is selected to be used in further development process. Major elements and subsystems of the software architecture are also defined.
- **Module design:** The software system submodules are refined in individual software modules with more implementation detail.
- **Coding:** Actual coding of the previous developed modules.
- **Module testing:** The goal is to verify if every module is delivering the specified functionality.
- **SW integration testing:** Used to verify the interaction of multiple modules as defined by the software system design.
- **HW/SW integration testing:** Verifies that the interface between the

software and hardware is working properly.
- **Validation testing:** The system is tested to certify that it meets the set requirements and functionality.

Test Driven Development

Another approach to software testing is the test driven development. It was introduced as part of agile development process. In this approach, the developer writes new functionalities and corresponding test cases at the same time. In some cases tests are written even before writing the code. The test cases are derived from the system requirements. However, there needs to be a clear relationship between the written code and the requirements specification. This is not always possible [11].

3.2. Testing Techniques

Different testing techniques are used for different test levels. A white-box approach is usually applied for lower level tests such as unit tests. Black-box approach is used for testing at a higher level (e.g., HW/SW integration tests). Both are described in more detail in the following sections.

3.2.1. Black-box Testing

A black-box testing technique verifies the components functionality without knowledge about the component's internal structure. In the example of an object-oriented class, a black-box test is conducted without the original source code. The only important resource is the components design specification as we use it as a test reference to verify the proper functionality of the component. A component is represented as a black-box with inputs and outputs 3.4. To completely test the functionality of the component, a combination of all possible inputs and outputs shall be tested. This is of course, practically impossible, as it would be heavily time and resource consuming. To achieve a reasonable amount of test possibilities, methods like equivalence class partitioning and boundary value analysis techniques are used. A major disadvantage of black-box testing is that it does not provide insight about the test coverage.

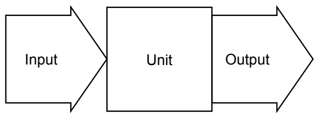


Figure 3.4.: Black-box testing

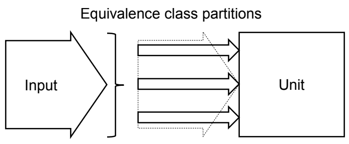


Figure 3.5.: Equivalence class partition

Equivalence Class Partitioning

One way of reducing the number of inputs to a reasonable amount is partitioning the input range into finite number of partitions. This is illustrated on the example shown in Figure 3.5. The input is partitioned in three input equivalence classes. A partition or equivalence class has a representative member of that class. The test cases can be generated under the assumption that all members of an equivalence class are processed in an equivalent way. Identifying the equivalence classes is the main challenge here. To effectively identify the groups following guidelines by [12] are used:

1. If an input condition specifies a range of values, one valid equivalence class and two invalid equivalence classes must be identified.
2. If an input condition specifies several values, again, one valid equivalence class and two invalid equivalence classes must be identified
3. If an input condition specifies a set of input values and the software handles each differently, a valid and an invalid equivalence class must be identified for each.

If there is a “must-be” situation specified, one valid and one invalid equivalence class must be identified. The valid equivalence classes represent all valid inputs to the program while the invalid classes represent all other possible states, such as erroneous inputs. Although equivalence partitioning covers a decent amount of test cases, it still overlooks some test cases. One example are the boundary values of the appropriate equivalence classes.

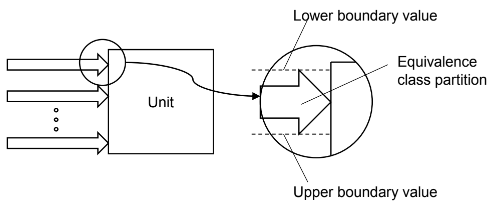


Figure 3.6.: Boundary value analysis

Boundary Value Analysis

Boundary value analysis is a supplementation method for the equivalence class partitioning. When testing equivalence classes, one usually selects a typical test input value, while overlooking atypical values. These values are often found on lower and upper boundary values. Consequently, test cases using those values shall also be written [11]. Figure 3.6 shows an example of applying the boundary value analysis one equivalence class defined above where the upper and lower boundary values are selected as test case inputs.

Other Black-box Techniques

The above-mentioned techniques cover most of the unit test cases. Additional test cases can be designed by using one of the following black-box techniques:

- **Cause-Effect graphing:** Formal language used to yield test cases which explore combinations of different inputs.
- **Error guessing:** The tester/developer uses his intuition, testing experience and existing knowledge to write additional uncategorized test cases.

3.2.2. White-box Testing

A white-box testing approach uses the knowledge of the internal structure of a component to derive test-cases and ensure that all elements of the unit are functioning properly. The source code of the components is usually available at the unit test level. The code contains structural elements such as statements and branches. White-box tests are focused on the degree to which these elements are executed. Test coverage defines the proportion of structural elements which are exercised in the test. Table 3.1 by [13] shows an overview of selected coverage types with an ”•” indicating which coverage criteria is applied. IEC 61508-3 [4] recommends different code coverage metrics for different safety integrity levels. For SIL 3, a 100% coverage for entry points, statements and branches is highly recommended. One should be careful: archiving a high code coverage does not mean that the code is fault free! It can, however, give an overview of which code has been overlooked during testing.

Statement Coverage

The simplest method to evaluate the test coverage is the statement coverage. The goal is to check if every statement in the program is executed at least once. The minimum requirement for most programs is the decision coverage which in most cases satisfy statement coverage. Consequently, statement coverage is rather an unpopular criterion [14].

Decision Coverage (Branch Coverage)

To achieve complete decision coverage, each decision element (if-else, switch-case and do-loop) must execute all possible outcomes at least once. The advantage of decision coverage is that it includes the statement coverage as well [15].

Condition Coverage

The decision coverage may be stronger than the statement coverage, but it is still considered to be weak because it does not cover all possible condition outcomes in a decision. This is covered by the condition coverage. Condition coverage, however, does not require that all decisions take all possible outcomes. This is covered by the next test coverage.

Decision/Condition Coverage

If, in addition to 100 % decision coverage, every condition outcome is also required, then it is called decision/condition coverage. The problem with this coverage is that certain conditions mask other conditions.

Table 3.1.: Coverage criteria overview: 1) Statement coverage 2) Decision coverage 3) Condition Coverage 4) Decision/Condition Coverage 5) Modified Condition/Decision Coverage [13]

Coverage Criteria	1	2	3	4	5
Every point of entry and exit in the program has been invoked at least once		●	●	●	●
Every statement in the program has been invoked at least once	●				
Every decision in the program has taken all possible outcomes at least once		●		●	●
Every condition in a decision has taken all possible outcomes at least once			●	●	●
Every condition in a decision has been shown to independently affect that decisions outcome					●

Modified Condition/Decision Coverage

MC/DC criteria requires that each condition is shown to independently affect the outcome of the decision. This ensures that each condition is tested in relation to the other condition. MC/DC, however, requires considerably more test cases [13].

Other Coverage Criteria

The above coverage criteria are widely used and are highly recommended criteria. There are also some other optional techniques mentioned in IEC 61508-7 [16] like LCSAJ (Linear Code Sequence and Jump) coverage, data-flow coverage and path coverage. However, these techniques are not used in testing failsafe peripheral modules and, thus not covered in the scope of this thesis.

Table 3.2.: Brief overview of test levels and their purpose in the life-cycle

Test level	**Purpose**
SW unit	Testing SW components in isolation
SW integration	Testing interaction between SW components
HW/SW integration	Testing interaction between HW and SW components
Validation	Testing that the system works as specified

3.3. Test Levels

The V-model software life-cycle covers different test levels. Table 3.2 shows a brief overview and the main objective of each level.

3.3.1. Unit Test

Unit test is the first testing activity after coding. The software components which were defined and programmed in the previous development phase are now being tested. Unit testing ensures that the component is functioning properly in isolation. It is decisive to ensure that the individual components are tested properly before moving onto the integration test. How to define a component under test? A component should be a smallest meaningful unit which can be tested individually and has a specified function. If a fault is detected, it is then assigned to the tested component. A good example is a class as unit-under-test in object-oriented programming.

3.3.2. SW Integration Test

Precondition for the software integration test is that all the modules are fully tested. This should ensure the correct functionality of the individual modules. Integration tests are used in order to show that all software modules interact correctly. The integration test strategy is explained on the example from [15] as shown in in Figure 3.7. A group of modules is represented as rectangles named from `M1` to `M13`. A line from an upper module to a lower means that the upper calls the lower. To test all the submodules, there are basically two

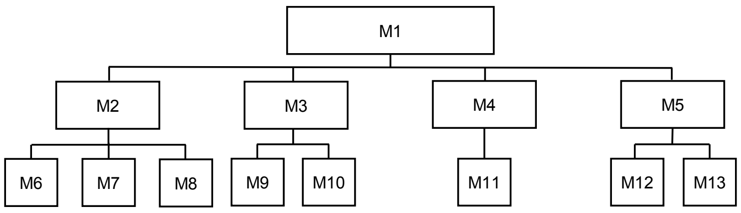


Figure 3.7.: Integration test strategy example [15]

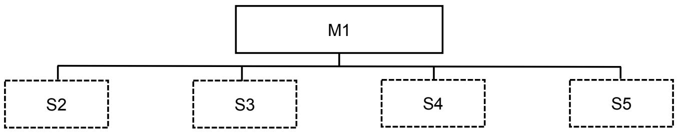


Figure 3.8.: Top down integration strategy [15]

main approaches: from bottom to top and from top to bottom.

Top-down Integration Test

Top-down integration starts with the top module. In the example shown in Figure 3.8, `M1` is the highest-level module. To write test cases, stubs for modules `M2`, `M3` and `M4` are used. The next modules which are integrated are the subordinate modules of the top module. One stub at a time is replaced by the actual module until all modules at this level are tested (`M2`, `M3` and `M4`). After that, the lowest-level modules are integrated by the same principle as for the higher-level modules. In some cases, testing can be done in parallel. For example, while one tester is testing `M2` and its subordinates, the other can test the `M3` subsystem. The top-level module is tested first in this type of integration. This can be quite useful for more complex modules which need more time for testing. Furthermore, it can also be advantageous if major flaws are found in the top-level module.

Bottom-up Integration Test

Bottom-up integration is explained on the same example as above. It begins with testing the lowest-level modules. These are the modules which do not include any other modules. In the example above, these are the modules from `M6` to `M13`. After testing the lowest module, the next step is to integrate the modules on the upper level. The upper level module, which was used as a driver

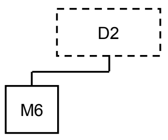


Figure 3.9.: Bottom up integration strategy [15]

for the lower level modules, is now replaced by the actual module. At this stage, this module needs a driver. The same procedure is followed for all other modules until the highest-level module is reached. Lowest level modules are tested well at the beginning of the process. This can be useful if flaws occur toward the bottom of the program. However, due to the lack of time at the later process of integration, the top module may not be well tested. This can be quite risky if the top module is safety-critical. Another major disadvantage of the bottom-up integration is that the program does not exist until the last module is integrated.

Neither of the methods have a clear-cut advantage. Thus, in many cases a combination of the two approaches is used. Also, risk factors and complexity are taken into consideration. Safety-critical modules should be tested adequately and earlier in the integration process. Another aspect to consider is the availability of the modules: not all modules will be available at any time. The software integration is completed after all software modules are integrated. The next step is to test the interaction between software and hardware.

3.3.3. HW/SW Integration Test

After all of the software modules have been integrated, the next step is to integrate the software subsystems into hardware. The goal here is to verify the interaction and interfaces between the software and hardware.

3.3.4. Validation Testing

While previous test phases focus on discovering faults in the system, validation testing validates if the system fulfills all the requirements. Primary goal of the validation is to show that the system delivers specified functionality and is good enough in terms of performance and reliability. Validation tests usually include requirements-based testing, scenario testing and performance tests. Typical test cases are derived from system specifications and use a black-box approach [11].

3.4. Simulation Techniques in Testing

Table 3.3 is an extended version of Table 3.2. Added columns specify which components of a system are used as experimental, prototyped or real on different testing levels. It is evident that different parts of the embedded system are simulated through different test levels. To cover that, the general term X-in-the-loop has been established in the literature [10]. The term X-in-the-loop covers a group of simulation techniques: model-in-the-loop, software-in-the-loop, hardware-in-the-loop and processor-in-the-loop. Following is a brief description of the x-in-the-loop techniques:

- **Model-in-the-Loop (MiL):** Except the host PC where the simulation is executed, the MiL approach does not include any hardware components. This technique is used at an early stage of safety life-cycle and is not covered in this thesis.
- **Software-in-the-Loop (SiL):** In some literature the term SiL is often referred to as MiL because, just like MiL, no real hardware is used. The only difference is that the model of the embedded software is replaced by the real prototyped software. The software is compiled on an ISS (Instruction Set Simulator) and a simulation of hardware components is used. This is the simulation technique covered in this thesis.
- **Processor-in-the-Loop (PiL):** PiL is like SiL with one key difference: the software is compiled and run on the target processor instead of using the ISS. The PiL is important because it can reveal faults caused specifically by the target architecture.
- **Hardware-in-the-Loop (HiL):** HiL approach is used at later stages of the software development for validation purposes. The target embedded software is compiled on the target processor and hardware. The environment of the embedded system is simulated.

The example of development of automotive ECUs, [17] offered a good overview how different x-in-the-loop techniques are implemented in different phases of the V-model. This is illustrated in Figure 3.10. Early in the development phase, the functional specification phase is supported by a MiL simulation. In the

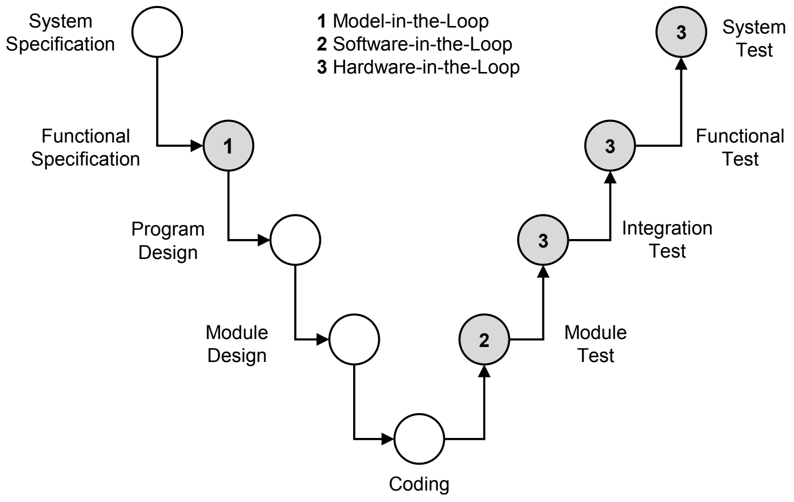


Figure 3.10.: Simulation techniques in the development of modern ECUs [17]

testing phase, SiL simulation is used in module testing. The later test stages such as integration, functional and system testing are supported by a HiL simulation.

3.5. Conclusion

Software development of peripheral failsafe modules directly follows the V-model life-cycle, especially in the testing phase. Traditional testing only carried out tests on levels mentioned in Table 3.2. However, the need to discover faults and test new functionalities as early in development phase as possible, combined with the need to have more transparent overview of the HW/SW integration test, led to an idea to use simulated hardware components. Consequently, a set of test levels was introduced which can be found emphasized in Table 3.3. It is important to note that the proposed test levels are not meant to substitute existing test levels but rather to supplement them. From the x-in-the-loop approach, these tests are closest to a SiL simulation, because the hardware is not being used. Basic SiL simulation block diagram is shown in Figure 3.11. The device under test (DUT) is the virtual peripheral module which includes the software code compiled on an ISS (Instruction Set Simulator) and simulated hardware. Test environment controls the simulation, provides input to the DUT

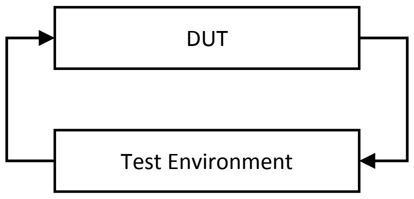


Figure 3.11.: SiL simulation diagram

and gets output data from it. Finally, to successfully implement and fully utilize the proposed testing levels, software models of the target hardware is needed. Therefore, the subsequent chapter elaborates how to model the hardware of an embedded system.

Table 3.3.: Test levels and simulation

Test level	Software	Hardware	Processor
SW unit	experimental	prototype	real
SW unit (S)	**experimental**	**simulated**	**simulated**
SW integration	experimental	prototype	real
SW integration (S)	**experimental**	**simulated**	**simulated**
HW/SW integration	real	prototype	real
HW/SW integration (S)	**real**	**simulated**	**simulated**
Validation	real	real	real

4. Modeling Embedded Systems

In the previous chapter, simulation-based testing was introduced which requires simulated hardware. It was established that, software models of the hardware are needed in particular. Modeling in general and specifically modeling embedded systems is by no means a straightforward and streamlined process. By using one of the many modeling methodologies and frameworks, the challenge is not the modeling process by itself, but rather the preparation. Before modeling, ofollowing questions arise: what is the main task that the model should be used for and, in respect to that task, how much detail is needed. Furthermore, a model can be realized on different abstraction levels. Optimal modeling effort is achieved by choosing the right abstraction level.

4.1. Abstraction Levels

One approach to categorize abstraction levels in the embedded systems is the Gajski-Khun Y-Chart, invented in 1983 [18]. It was originally introduced to deal with the classification and structuring of design process by using a set of well-defined abstraction levels. Four different abstraction levels are represented: system, processor, logic and circuit level. They are graphically represented as concentric circles shown in Figure 4.1. The levels are defined by the components derived on the specified level. On every abstraction level, three different aspects are considered: behavioral, structural and physical. Behavioral aspect covers the functionality of the design. The design is considered as set of interconnected components in the structural aspect. Finally, physical aspect adds dimension specifications to each component. Gajski [19] also lists the typical components found on each abstraction level, which helps in identifying the right level. These components, sorted by levels, are:

- **Circuit level:** cells consisted of P/N-type transistors
- **Logic level:** registers, register files, ALUs and multipliers

- **Processor level:** processors, memory controllers, bridges and different interface components
- **System level:** embedded systems made of processors, memories, buses and other components.

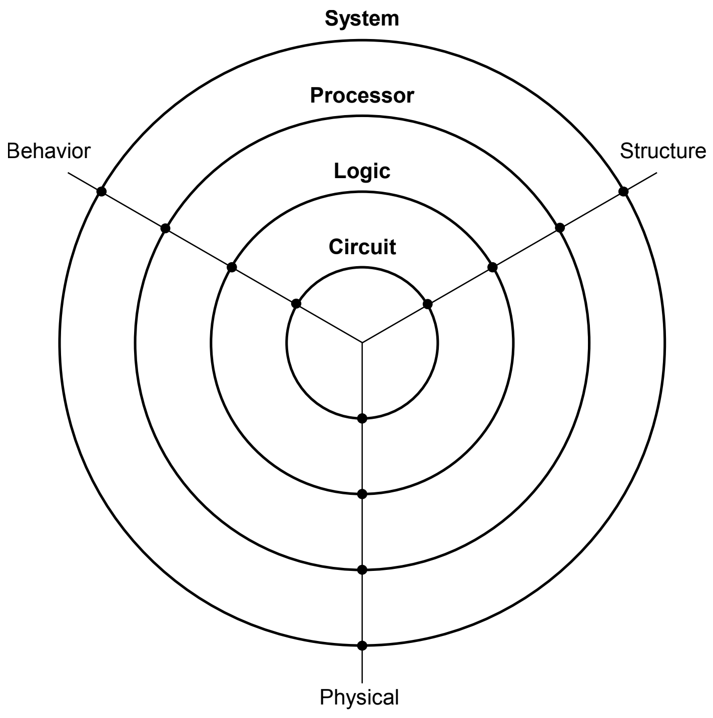


Figure 4.1.: Y-chart diagram [19]

4.2. Abstraction Pyramid

To choose the right abstraction level always requires the trade-offs to be made. This involves mainly three different issues: the modeling effort, evaluation effort and the model accuracy. To put those aspects into the perspective of the system level design, the abstraction pyramid was introduced by [20]. Abstraction levels presented in the previous section are placed in the context of the abstraction pyramid shown in Figure 4.2.

Level of Detail. Concentric circles from the previously introduced Y-Chart diagram are directly mapped as horizontal lines on the abstraction pyramid.

Cost of Modeling. Moving down in the pyramid means moving down to lower abstraction levels. This results in a more detailed architecture where more details need to be considered and which consequently results in an increasing amount of effort. This is indicated on the axis on the right side of the pyramid.

Opportunity to Change. Different design choices lead to lower abstraction levels. At that point, to consider another architecture becomes costlier. Consequently, the exploration opportunity at that level is low.

Accuracy. The accuracy of the models is represented by the axis on the left side of the pyramid. The model accuracy increases with lower abstraction levels.

Cost of Evaluation. Placed on the same axis as the cost of the modeling is the cost of evaluation. It was already mentioned that the modeling effort increases proportionally with the amount of modeling detail involved. This also means an increasing evaluation effort.

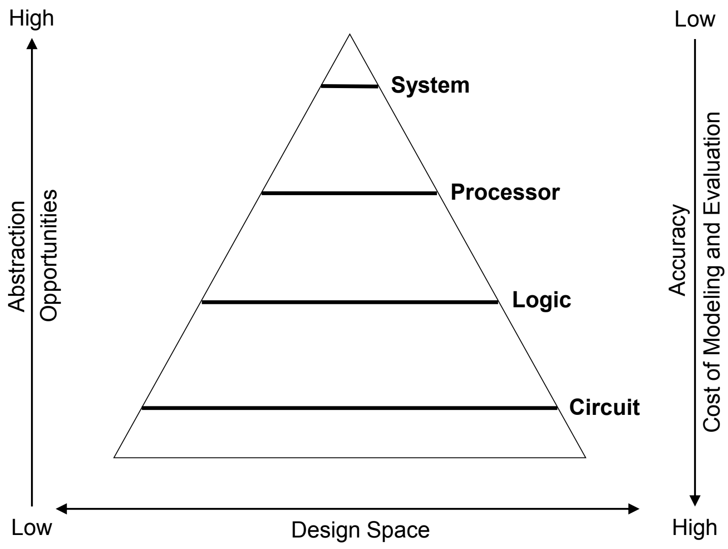


Figure 4.2.: Abstraction pyramid [20]

With a specific modeling task in mind and in respect to the abstraction pyramid above, it is time to consider the abstraction level. Since the main task is simulation-based testing and not hardware synthesis, level of detail and accuracy is traded for simulation speed and effort of modeling. This puts the target model on a system abstraction level which by definition handles embedded system components such as processors, memories, peripherals etc.

4.3. System Level Modeling

After choosing the right abstraction level, the next decision to make is how to approach the modeling process. According to [21], there are basically two different modeling approaches used to model a system: homogeneous/single-language modeling and heterogeneous/multi-language modeling. By using the homogeneous approach, a single-language is used to model the complete system including the hardware and software. This approach is often not applicable to complex systems. In such cases the heterogeneous approach is applied, where multiple languages are used for different domains of the system.

4.3.1. Single-Language Approach

By using a single-language approach, the intent is to find possibly one general language to describe the complete system. Homogeneous modeling approach results in a single, executable system. Only one simulator is needed to execute and verify the model and simulation handling is therefore much easier. However, it is hard to find a system which is simple enough to be described by just one language. Most of the systems require different languages for different domains. To tackle such systems, a multi-language approach is introduced. This is illustrated on Figure 4.3

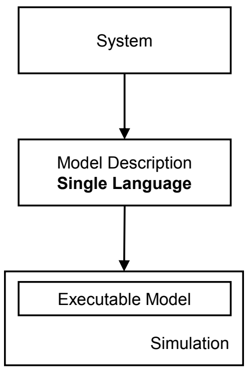


Figure 4.3.: Single-language approach [21]

4.3.2. Multi-Language Approach

Multi-language approach uses different modeling languages for different domains simultaneously. This results in multiple execution models with different simulation environments. A major problem here is the coupling of these simulation environments. One of the possible solutions is to use a common intermediate language. This way, different models are converted into a common language which is later executed in a single simulation environment. This method is an extension of the single-language approach but provides more flexibility and makes modeling of the complex systems possible. Another option is to skip the intermediate common language altogether and combine the respective execution models and corresponding simulation environments into a multi-domain simulator. Every model is simulated separately and at the final stage synchronized with a common simulator. Although the formal verification of the complete system is hardly achievable, the approach provides a platform for developing complex simulations which can be executed efficiently. Multi-language approach is shown in the Figure 4.4.

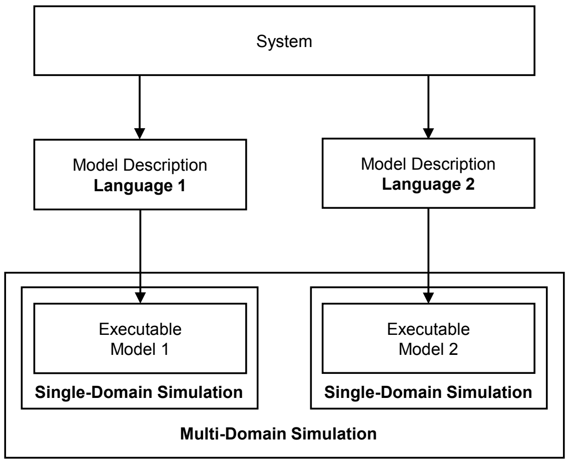


Figure 4.4.: Multi-language approach [21]

4.4. Existing System-Level Modeling Languages

After choosing the right modeling approach, the designer faces one more challenge: to select the modeling language or platform. In early stages of hardware modeling, dealing with gate level and register-transfer-level design, hardware-description-languages (HDLs) such as VHDL and Verilog were introduced. Although, these languages were suitable for modeling hardware on lower abstraction levels, the transition to the system level modeling resulted in new requirements such as modeling parts of the system implemented in software. Additionally, a system-level language must be executable, modular and complete [22]. This lead to an introduction of system-level design languages (SLDL) based on C/C++ [19].

4.4.1. SpecC

One of the system-level design languages is SpecC [23], which is based on C programming language. Thus, it covers the complete set of C constructs. Additionally, it supports constructs covering the system level design requirements. One of those requirements is modularity, which is required to separate the behavior of the system from its structure. With behavioral hierarchy the system behavior is decomposed in multiple sub-behaviors, whereas structural hierarchy allows decomposing a system into multiple interconnected components. A SpecC program typically consists of a set of behaviors, channels and interfaces [22]. A behavior describes a functionality and consists of ports, component instantiations, variables and functions and a `main` function. Channels encapsulate the communication while the interfaces represent a link between behaviors and channels. An example of a SpecC system is shown in Figure 4.5. The system realizing a behavior `B` is hierarchically decomposed into two sub-behaviors `b1` and `b2` which communicate via channel `c1`. The sub-behaviors can be executed either sequentially or concurrently [23]. A special SpecC compiler is needed to compile the program. The compilation results in an intermediate C++ model, which can be then compiled using a standard C++ compiler [22].

4.4.2. SystemC

Another system-level modeling language is SystemC [24], which is basically a set of C++ classes. It allows fast simulation at different abstraction levels, from system to register-transfer-level. Another great feature of the SystemC is the model interoperability, which was introduced with the Transaction Level Modeling (TLM) [25]. SystemC meets the imposed requirements set by ESL design. These include the abstraction span on several levels, a standardized

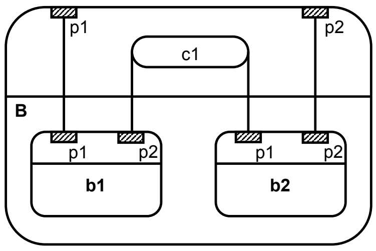


Figure 4.5.: Basic structure of a SpecC model [23]

language, proper simulation speeds and performance and support of TLM concepts. For that reason, SystemC is rapidly adopted in the industry [26].

Figure 4.6 describes SystemC as a set of blocks and layers. The base layer on the bottom shows that SystemC is built and based on the standard C++ language. The bold emphasized group of blocks are parts of the SystemC standard which includes the simulation kernel and core language elements. Alongside the core language is the data-type group. The elementary channels layer above the core language and data-types include models such as signals, timers and FIFO buffers. Unlike the emphasized layers, the topmost layers are not part of the SystemC standard but represent other models, libraries and extensions to support additional features [27]. A study by [28] used SystemC to model a safety-critical embedded system. A co-design and a simulation of fault injection in train on-board safety-critical odometry system has been proposed. The goal was to develop a simulation environment to implement fault-injection techniques in multiple steps of the design process, as recommended by IEC 61508. This approach resulted in preventing late discovered faults. The study demonstrated advantages of SystemC such as describing both hardware and software in common language and execution of concurrent processes.

SystemC and VHDL Comparison

A case study [29] directly compared traditional hardware-definition-language VHDL with SystemC by modeling a simple load-store processor on a register-transfer-level. The comparison relied upon the simulation time and modeling effort. In terms of simulation time, SystemC is much faster compared to VHDL, reaching a difference in an order of magnitude. Although hardware-software co-simulation is possible in VHDL, it requires external tools and libraries. SystemC uses the same language for hardware description and for co-simulation. Hence,

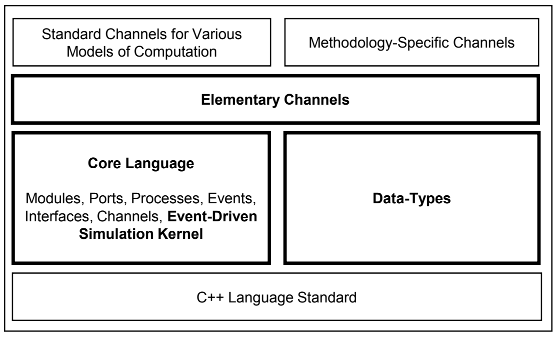

Standard Channels for Various Models of Computation
Methodology-Specific Channels
Elementary Channels
Core Language
Modules, Ports, Processes, Events, Interfaces, Channels, Event-Driven Simulation Kernel
Data-Types
C++ Language Standard

Figure 4.6.: SystemC language architecture [26]

the modeling effort with SystemC is much lower compared to VHDL. In conclusion, SystemC is much better suited for higher abstraction levels.

4.4.3. C/C++ Modeling

A methodology for hardware/software co-verification in C/C++ presented by [30] provides several advantages of C/C++ based methodologies over HDL-based methodologies. In traditional hardware-software co-simulations the hardware is usually described in an HDL whereas the software is mainly C/C++ based. Data is transferred between software and hardware written in different languages. By using just one language, the data transfer can be made much efficiently. Productivity can be further improved by eliminating the transition to a HDL and reusing test benches written in C/C++. Avoiding the translation to a HDL not only improves the efficiency of the simulation but also removes bugs produced by this translation. The programmers, which are already proficient in using C/C++ save time and effort by not learning a new language. Instead, they can focus on writing better functional models. [31] discussed the use of C++ in modeling digital systems and came to the conclusion that C++ is a well-suited language for system modeling. Without introducing new syntax or compiler, it allows the writing of modeling primitives based on C++ mechanisms such as classes, templates and operator overloading. The main drawbacks of C/C++ modeling are lack of concurrency, missing the ability to entail structural information and constraints and lack of support for timing constraints. However, these drawbacks can be neglected when using functional models on a system level and are not relevant for hardware synthesis [32].

4.5. System-Level Modeling Platforms

4.5.1. Commercial Tools

There are several commercial virtual prototyping tools available. Wind River Simics [33] is a full-system simulation tool which can simulate the processor and dedicated peripherals in such detail that it can run the target software. It features very fast bit-accurate instruction set simulators which can emulate targets such as PowerPC, MIPS, MIPS64, ARM, x86 or SPARC without any special host hardware and software. Alongside the processors, Simics offers a number of models for flash memories, I2C buses, timers and other components. If a specific target model needs to be developed and it is not available by Simics, a specific language called DML is created for writing new models [5]. Synopsis [34] also offers a virtual prototyping solution covered by Platform Architect, Virtualizer Studio and according to their website, the largest portfolio of TLM models. Virtual System Platform by Cadence [35] is another commercial tool for virtual prototyping. Just like Wind River and Synopsis, Cadence also offers a library of TLM IP models. Additionally, they also offer support for Imperas OVP fast processor models (which is covered later). Another great feature is an automatic TLM 2.0 code generation, which reads a custom text-based language called IP-XACT and produces a TLM 2.0 model without requiring TLM 2.0 knowledge. There are many other commercial virtual prototyping commercial tools. This section only mentioned the most popular ones. These tools have high licence fees. However, there are a number of open-source alternatives covered below.

4.5.2. OVPSim

Imperas Open Virtual Platforms [36] offer open source software for developing virtual platforms: an API, a simulator and a library of free open source processor and peripheral modules. All OVP models are fully SystemC and TLM 2.0 compatible by using APIs and TLM 2.0 wrappers. However, every model needs to be redesigned individually. [37] offered a much easier way to integrate OVP models in SystemC using a SystemC bridge. In [38] OVP was used in a SiL simulation of an embedded control application.

4.5.3. QEMU

QEMU [39] is a generic and open source machine emulator and virtualizer. Two emulation modes are available: full-system and user-mode. In the user-mode, QEMU can launch processes compiled for one CPU on another CPU. Full-system mode is used to emulate the processor and its peripherals. QEMU offers and provide support for systems based on following architectures: x86, ARM, PowerPC and MIPS. Unlike OVP, there is no native support for integration of SystemC models. However, [37] developed a SystemC bridge for integrating QEMU in SystemC. Moreover, a number of co-simulations have worked on that. [40] added a set of plug-ins that enabled to integrate SystemC modules in QEMU as peripherals to the emulated platform. The connection between was implemented using TLM channels. [41] offered a different approach by integrating QEMU in a SystemC. The QEMU virtualizer is treated as a standard SystemC module. In modeling cyber-physical systems, [42] used SystemC and QEMU for modeling hardware of the micro-controller. [43] developed a simulator for networked embedded systems. The simulation framework is based on QEMU and SystemC, where QEMU was used for the execution of target software while SystemC was used for accurate modeling of network protocols and topologies. The simulator was then applied for robustness testing of the communication layers in a embedded fire alarm system. The test results only confirmed the benefits of using a virtual prototype such as test automation, exploration of unlikely test scenarios and increased observability. [44] presented another approach in connecting QEMU and SystemC using a TLM-2.0 interface. [45] used QEMU, SystemC with other commercial ESL simulation tools.

4.6. Modeling Timing Behavior

Timing is an important aspect to cover in system modeling. When considering integrating timing behavior in a virtual prototype, the same universal modeling principles apply: making design decisions in terms of abstraction level and simulation performance. [46] illustrated that in Figure 4.7. Below the approximately timed (AT) abstraction level are clocked levels. This means that the simulation is clock-driven. Above the AT level, simulation is advanced by transactions (data transfer e.g.). The diagram shows that the simulation performance differs by one or two orders of magnitude between two levels. Additionally, the time required to develop the model is also increasing with lower abstraction level. Levels below the AT are typically used in the domain of hardware developers. In scope of system modeling, higher abstraction levels are interesting.

Code Instrumentation

[47] introduces a basic approach for integrating timing behavior into a system: code instrumentation. The base idea is to instrument the code by timing statements with a dedicated time values. The main advantage of this approach are very fast simulation times. However, the determination of execution time requires detailed code analysis and is very time consuming. There are basically two approaches in determining execution times: analytical and simulative. Analytical approaches use detailed code analysis and WCET/BCET (Worst Case Execution Time/Best Case Execution Time). The result is an approximate mean value of execution times which gives a rough impression of the behavior of the system. Simulative approaches use a simulation on a target ISS and simulated hardware peripherals to obtain timing behavior. [48] presented a hybrid approach by combining the two. First, they used a static analysis of code. The extracted timing information is then back-annotated in the simulation code. This considers the aspects not covered by static approaches. Moreover, the approach showed significant simulation speeds while maintaining good accuracy.

As a conclusion, to accurately integrate timing behavior in the simulation, one needs a cycle-accurate processor model. The simulation however, is at a system level and the timing information is only used for logging and diagnosis purposes. Thus, for the implementation of the target system a software-timed model is preferred. The timing will be implemented using the code instrumentation method.

4.7. Conclusion

The chapter concludes with a decision about what kind of a system modeling platform/language is used for modeling the target system. Virtual prototyping commercial tools would be a great choice because they offer a number of toolboxes, large database of IP models and even support importing of additional SystemC-TLM models. However, commercial tools are not available without

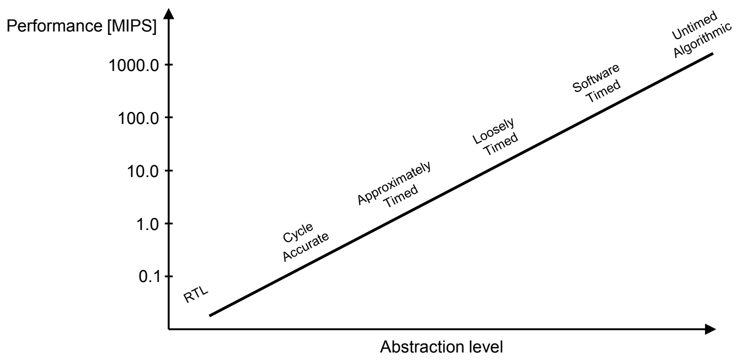


Figure 4.7.: Level of abstraction vs. performance in modeling timing behavior [46]

paying high license fees, thus they are not taken into consideration. Alternatively, there are open-source platforms such as OVP and QEMU. Even though, they both offer support for a great number of processors, including the target ARM processor, they lack a database of our target peripheral IP models. This means, that one would have to manually create, not only the SystemC models, but also wrappers and bridges to connect them to the simulation. This would lead to an excessive modeling effort. When it comes to system-level modeling languages, literature research above showed a massive use of SystemC. SystemC in a combination with a instruction set simulator would indeed be a good choice if the sole purpose of this thesis was to create a virtual prototype of the target hardware. However, this thesis deals with modeling failsafe programmable systems in accordance with the international standards. IEC 61508 also covers the aspect of software tools used during the software development life-cycle and according to [49], all software tools used during the software development life-cycle are divided in three classes:

- T1: tools that generate no outputs that can directly or indirectly contribute to the executable code (e.g., text editor)
- T2: tools that support the test or verification of the executable code, where errors in the tool can fail to reveal defects but cannot directly create errors in the executable (e.g., test coverage tools)
- T3: tools which generate outputs that directly or indirectly contribute to the executable code (e.g., compiler tools)

Either way, to integrate tools from classes T2 and T3, verification and qualification activities are applied to the tool property. This means that the tools must undergo a system FMEA analysis or, if applicable, a comment if the tool property is already qualified by the tool vendor. Since the above tools either directly (modeling platforms) or indirectly (compilers) classify as T2 or T3, the same verification and qualification activities would be applied. Additionally, the proposed system shall be easily integrated in the existing test environment. This includes the reuse of the tools for code coverage analysis test reporting. Furthermore, to introduce new tools and languages would mean increased learning and integration effort. For the given reasons, a modeling approach with C/C++ integrated in existing tools, which already passed the tool qualification activity, is chosen. The tool-chain also provides an integrated instruction set simulator of the target ARM processor. Other devices are modeled using C/C++. This approach should be sufficient in terms of performance and modeling accuracy for creating a simulation platform for testing. Details about the modeling approach are in the chapter below.

5. System Design

The complete simulation-based system design and all the components are shown in Figure 5.1. Each subsection covers one block from the layered structure in the figure. The only hardware used is a host machine (PC). The simulation project is compiled and runs on this machine. The simulation utilizes the hardware resources such as memory and processing power from this machine. Thus, the machine shall offer enough computing power and memory resources.

5.1. Simulated Hardware

The failsafe peripheral module was previously introduced in Chapter 2.2. It includes a redundant microcontroller system which consists of two STMicroelectronics STM32F2xx microcontrollers and each featuring an ARM 32-bit Cortex M3 CPU. For more details about the microcontroller see the reference manual [50]. The block diagram of the board is found in the Appendix A.

5.1.1. Generic I/O Pin Model

Different hardware components are all connected to the board via digital input/output pins. This fact is used to create a generic digital I/O pin model which can be utilized for modeling other peripheral devices on the board. The pin model is basically a C++ class with its class diagram shown in Figure 5.2. Each `Pin` instance has its pin value stored as a boolean to `true/false`. Additionally the previous values are stored. The `Pins` class has following methods:

- `isInputHigh` allows an external class to check the boolean status of a pin. If the pin value is set to high, the method returns `true`.
- `isOutputHigh` is the identical method like the one above. It is used to check the status of an output pin. Returns `true` if the output pin is set to high.

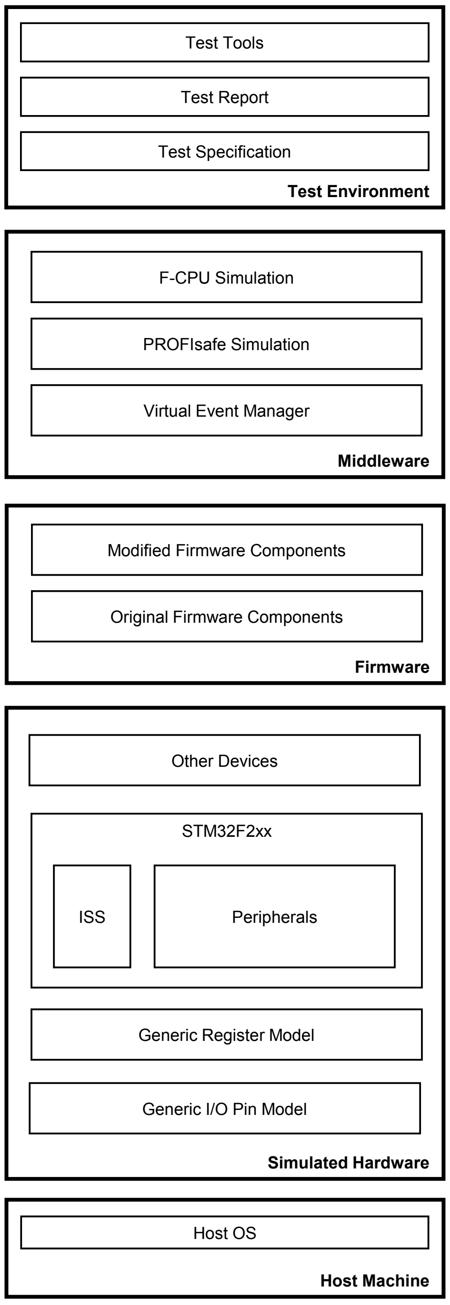


Figure 5.1.: Complete system overview

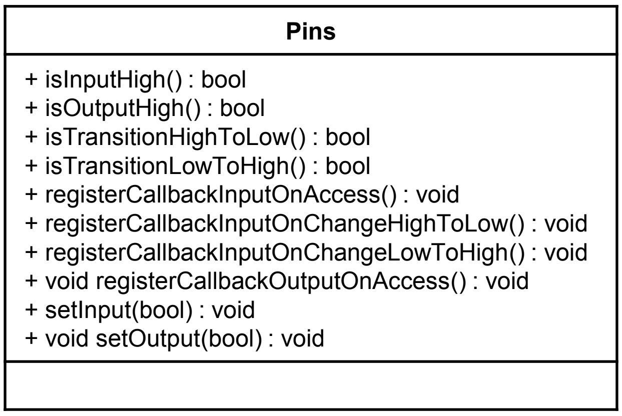

Pins
+ isInputHigh() : bool
+ isOutputHigh() : bool
+ isTransitionHighToLow() : bool
+ isTransitionLowToHigh() : bool
+ registerCallbackInputOnAccess() : void
+ registerCallbackInputOnChangeHighToLow() : void
+ registerCallbackInputOnChangeLowToHigh() : void
+ void registerCallbackOutputOnAccess() : void
+ setInput(bool) : void
+ void setOutput(bool) : void

Figure 5.2.: Pins class diagram

- `isTransitionHighToLow` method returns `true` if the pin status changed from high to low.
- `isTransitionLowToHigh` is identical to the method above. Returns `true` if the pin value changed from low to high.
- `registerCallbackInputOnAccess` allows to register a callback function for the case when the input pin is accessed.
- `registerCallbackOutputOnAccess` is the same method as above but applied for the output.
- `registerCallbackInputOnChangeHighToLow` is another method for registering a callback function, thus in this case when ab input change from high to low has occurred.
- `registerCallbackInputOnChangeLowToHigh` is the same method as above just for the output pin direction.
- `setInput` is used to set the input pin value to `true/false`. The method then sets the current state of the pin to the desired value. After updating the current state, it compares the value to the old state and checks if there was a transition from low to high or the other way around and calls the corresponding callback function mentioned above. Finally, it executes the callback function registered on the input pin access.
- `setOutput` is used to change the output pin value the same way as described above for the input pin direction.

5.1.2. Generic Register Model

This relatively simple class is used to model the `STM32Fxx` hardware registers. The class diagram is shown in the Figure 5.3. The register model value is basically a 32-bit unsigned integer value. With the public methods `set` and `get` one can set and get the register value, respectively. Similar to the generic pin model, the class offers the method `registerCallbackOnAccess` to register a callback function in case a register is accessed.

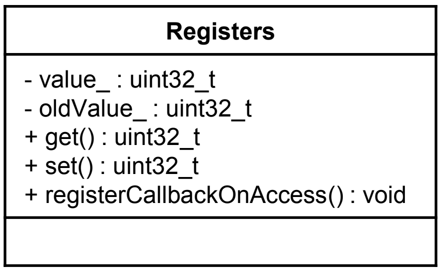


Figure 5.3.: Generic register class diagram

5.1.3. ARM Cortex M3

The target ARM processor is simulated by using the Green Hills Software instruction set simulator `armsim`. This ISS interpretively executes ARM programs on the host PC without the need for target hardware by simulating the execution of the target processor at the instruction level. It is a part of the Green Hills Software MULTI toolchains which also provides full debug features, host I/O, command window, extended profiling and hardware breakpoints. The executable software is compiled and debugged on the `simarm` [51].

5.1.4. STM32F2xx Peripherals

The complete list of `STM32F2xx` peripherals is found in the Appendix A. They are modeled as classes containing dummy registers (using the class `Register`). In the device specific file `stm32f2xx.cpp`, the peripheral instances are allocated. At the same time, the original header file `stm32f2xx.h` is modified. Particularly, the peripheral memory map has been changed. The physical addresses in the memory map are exchanged through addresses of the virtual peripherals in the hosts RAM. This is illustrated in the Figure 5.4. Depending on the application, the functionality of individual peripherals can be added. In the scope of the thesis, only the timer functionality is modeled and included in the simulation.

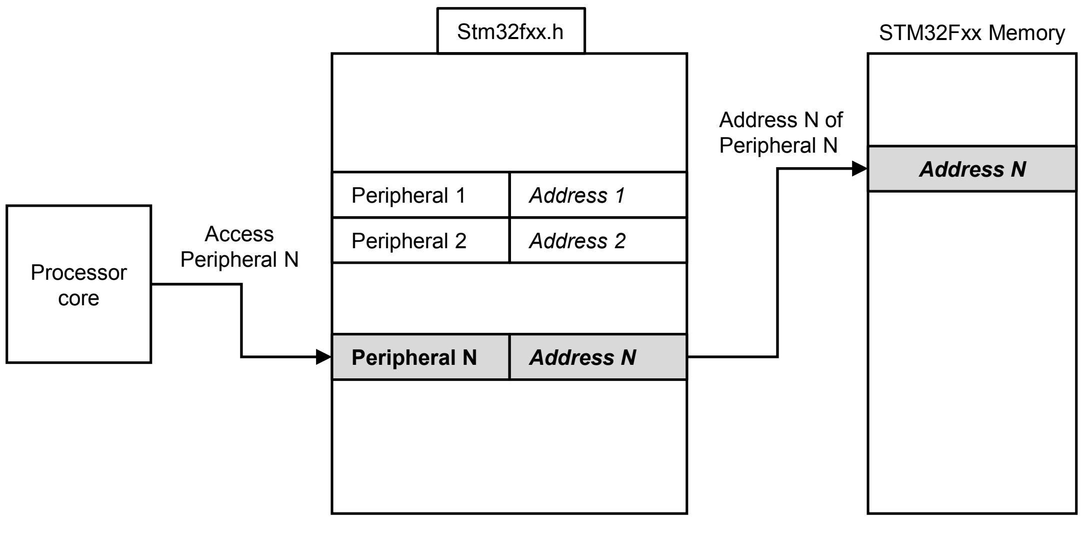
5.1.5. Timer

STM32F2xx board has a total of 14 timers: 2 basic (`TIM6` and `TIM7`), 2 advanced-control (`TIM1` and `TIM8`) and 10 general-purpose timers (`TIM2` to `TIM5` and `TIM9` to `TIM14`). They consist of a 16-bit/32-bit auto-reload counter driven by a programmable prescaler. Most of the timers support up, down, up/down auto-reload counter-mode. The 16-bit programmable prescaler is used to divide the counter clock frequency by any factor between 1 and 65535. For this thesis, the timer is modeled based on the basic model. The way the basic timer works is explained on the timer block diagram in Figure 5.5. The main blocks of the programmable timer are the counter register (`TIMx_CNT`), prescaler register (`TIMx_PSC`) and auto-reload register (`TIMx_ARR`). The counter is clocked by the prescaler output `CK_CNT`, which is enabled only when the counter enable bit (`CEN`) in the `TIM_CR1` register is set to `true`. The auto-reload register is preloaded with a specific value. After the counter reaches the overflow value an update event is sent.

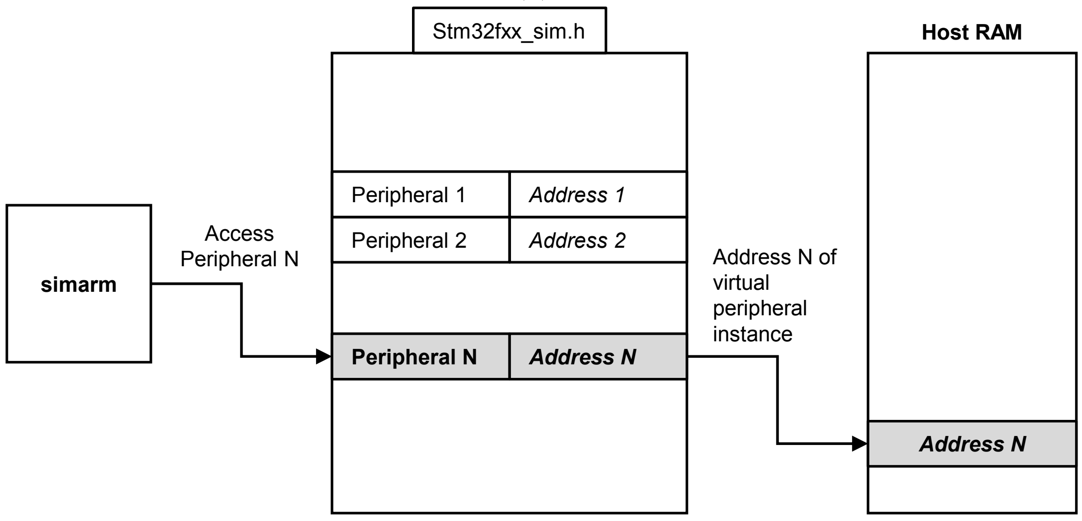
The functionality of the timer is modeled as a C++ class. Figure 5.6 shows the class diagram. It contains all the relevant registers modeled as the class `Register`. The timer is called with the `doExecute` method. This checks the `TIM_CR1` content to see if the `CEN` flag is enabled. If the counter is enabled, it calls the `doCountUp` method which basically realizes the up-counting mode based on the configured values of the auto-reload and prescaler register. When the counter reaches the overflow value (`TIM_ARR`) it sends an update event. Additionally, the option to register an interrupt handler is also implemented but not used in the scope of this thesis. The timer is already utilized to realize a virtual time system in the event manager in the section below.

5.1.6. Other Hardware

The modular system architecture allows simple adding or removing of additional simulated hardware components from the project. Simulated hardware components are modeled using C++ classes. To include a hardware component in a project, a `.cpp` is simply included in the simulation project. An example is



(a)



(b)

Figure 5.4.: Modified device header file: a) Physical processor accessing hardware perihperals b) Accessing modeled peripherals using the modified header file

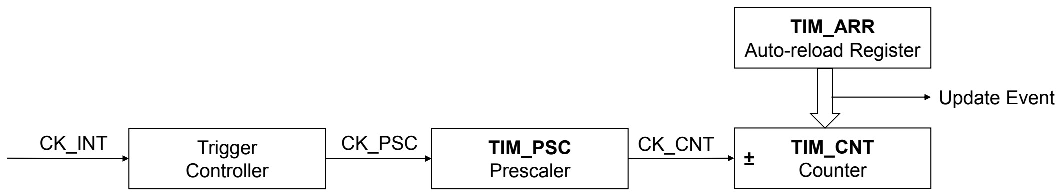


Figure 5.5.: Timer block diagram [50]

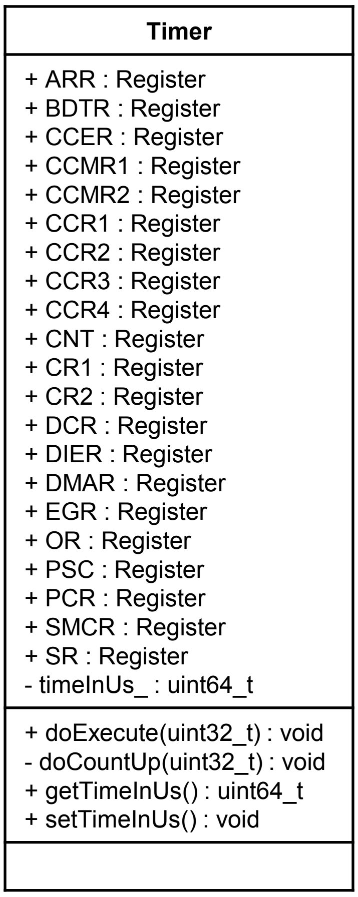


Figure 5.6.: Timer class diagram

the third-party ADC component used in the analog-input modules. Although, not relevant for the scope of this thesis, it can be simply added for simulation-based testing of FAI modules. Other examples include the I2C bus and I2C devices such as the EEPROM memory or the temperature sensor, which can be connected to the modeled bus. The same is applied to virtually any hardware IP component in the system.

5.2. Firmware

The firmware, which runs on both processors, covers the complete software functionality of the peripheral module. The base component of the firmware is the framework. The framework implements common functionality for every failsafe module. Additionally, some module specific components are included. A simplified firmware architecture is shown in the Figure 5.7 and described below:

- `ModuleSpecific` component represents functionalities implemented explicitly for one group of modules. An example would be the functionality of the ADC for the analog input modules.
- `DeviceSpecific` component contains the device specific headers and sources, which are provided by the toolchain or microcontroller vendors and are needed for realizing hardware access on the supported target. platforms.
- `Configuration` contains various files which make possible that the framework can be easily tailored to the module-specific application.
- `BusConnection` contains the externally developed bus communication components needed for realizing the bus communication.
- `BusInterface` component is a wrapper between the internal bus communication interface provided for all framework components, and the interfaces of all possible bus communication components, which realize peripheral bus communication. With this solution the framework components have a unified way of transferring data over the bus.
- `HardwareAbstractionLayer` is a bundle of various components providing hardware management functionality. These components and their functionality are accessible for all firmware components. The hardware abstraction layer isolates the rest of the firmware from direct hardware management.

- `ModuleController` component provides the base infrastructure for the firmware covered in Section 2.1. It implements the main state machine, manages the module cycles and provides entry points for the interrupt calls of the framework. The module specific part does not need to take care of these activities.`ModuleController` also controls the maintenance functionality of the module such as: calibration, reading/writing hardware version and reading fatal error information.
- `Utilities` provide miscellaneous functionality for modules. For example: an endianess converter, a CRC calculation unit, a random number generator etc. Utilities also include a timer component which provides functionalities for time measurement.

The firmware is compiled and run on the simulated processor. Ideally, the original binary image would be used. However, some of the firmware components had to be modified due to implementation choices. These are the above mentioned STM32F2xx header files from `DeviceSpecific` component. The `ModuleCycleController` has also been modified to include instrumentation for the virtual time system. Details about that are described in the section below. Another firmware component that needed modification in order to be compatible with the virtual time system is the `Timer` class from `Utilities` category. The class originally provides functionalities as time measurement and timeout and is utilized for measuring simulation cycle times and event times in the simulation event manager.

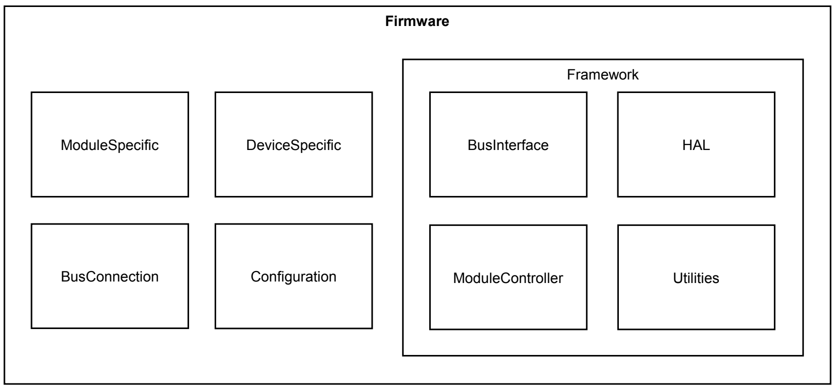


Figure 5.7.: Firmware architecture

5.3. Middleware

The middleware layer can be individually tailored to support different simulation test cases. Components can be added and removed in a modular way. For the sake of the implementation use-case demonstrated later, two additional components are added to the system: an F-CPU simulation and the PROFIsafe simulation.

5.3.1. F-CPU and PROFIsafe Simulation

Bus Interface

The data between the F-CPU and the peripheral module is exchanged cyclically and acyclically. Cyclic communication is used for process data exchange. The later one is used for exchanging data records, parametrization and sending diagnostic data. This communication is simulated with the `BusInterfaceFCPU` class. As illustrated by the Figure 5.8, a memory set in the hosts RAM is allocated for data records, process and parameter data individually. Furthermore, a set of methods is included to simulate the F-CPU and peripheral module sending/receiving data over the bus. This is basically realized by reading/writing from/to the respective memory blocks. Additionally, the class contains methods for parameterizing the virtual module.

PROFIsafe Simulation

Section 2.3 described the PROFIsafe technology realizing failsafe communication aspects between the F-CPU and the failsafe peripheral module. The class `ProfiSafeSimulation` simulates this. It provides the functionalities described in the mentioned chapter such as parametrization and CRC calculation for example.

5.3.2. Simulation Event Manager

A simulation event manager including a virtual time system has been implemented. Based on the Figure 2.2, the Figure 5.9 illustrates how the `MainCycleController` has been instrumented to implement a simulation event manager. After each component in the main cycle is called and executed, immediately after a `TimeTrigger` call follows. The time trigger has an argument of the class `Event`.

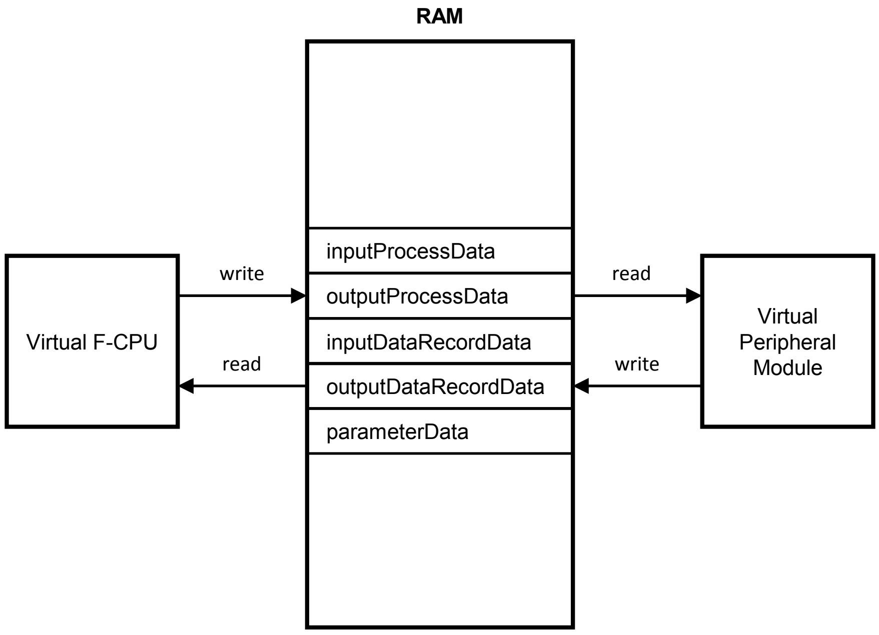


Figure 5.8.: Simulated bus interface

In the `Events.xml`, time durations in microseconds are assigned to every type of event. A complete list of the time events including their duration in microseconds is found in the Table 5.1[1]. The listed time values are just rough approximations based on developers experience. For example, the Event ID 2 corresponds to the Event `ResetWatchdogTimers` which takes approximately 100μs. On the other hand, the Event ID 5 is the `ProcessDataExchange` and takes much longer, approximately 1500μs. These time values are sufficient for logging and diagnosis purposes. However, approaches introduced in the Section 4.6 could be used for a more accurate approach. Additionally, one could also use hardware tools to measure the event types. Based on the time sequence diagram shown in the Figure 5.10, the virtual time system is realized in following steps:

- **1.0** The `ModuleCycleController` dispatches a new time event `Event` to the `ModuleEventTrigger`
- **1.1** The `Event` is further handled by the `ModuleEventHandler`. It identifies the event assigns the event its dedicated time duration from the Table 5.1.

[1]Event names are confidential company IP

- **1.2** It calls the method `doExecute(uint32_t EventTimeInUs)` which basically forwards the event time to the timers.
- **1.3** After receiving the time duration, the `Timer` checks if the timer is enabled and if it is, calls the `doCountUp` method.
- **1.4** The timer counter register is up-counted according to the description in the Subsection 5.1.5.
- **1.5** After the counting has finished, the `ModuleEventHandler` requests the time value stored from the `Timer`.
- **1.6** The time value in microseconds is returned to the `ModuleEventHandler`
- **1.7** Using the time value from above step as a current time stamp and the measured minimum and maximum cycle times, the `Event` is logged on the console in the format:
  `"[Cycle Time]: min: _us max: _us current: _us [Event]: EventID"`

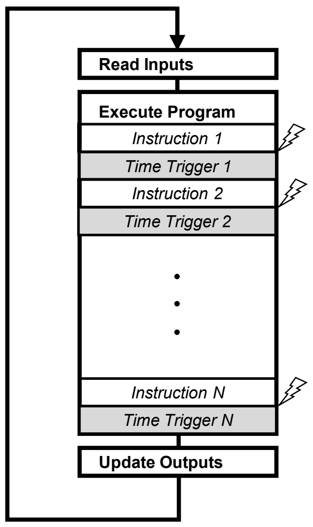


Figure 5.9.: Code instrumentation in module controller

5.4. Test Environment

The final part of the simulation system is the test environment. This is the part where the user interacts with the system. The user configures the simulation, defines and executes test cases. It is shown in Figure 5.11. The test specification

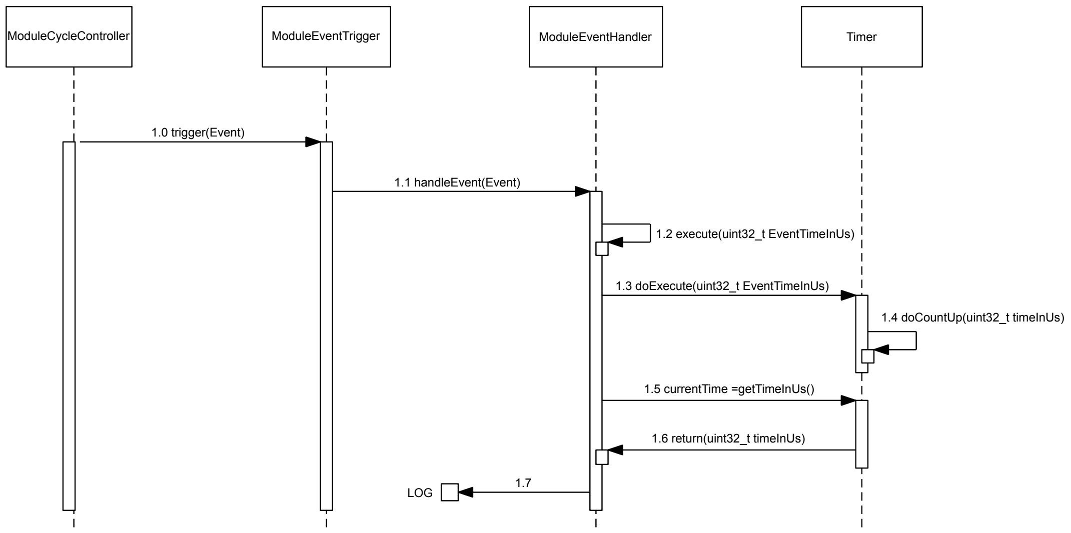

ModuleCycleController
ModuleEventTrigger
ModuleEventHandler
Timer
1.0 trigger(Event)
1.1 handleEvent(Event)
1.2 execute(uint32_t EventTimeInUs)
1.3 doExecute(uint32_t EventTimeInUs)
1.4 doCountUp(uint32_t timeInUs)
1.5 currentTime =getTimeInUs()
1.6 return(uint32_t timeInUs)
1.7
LOG

Figure 5.10.: Simulation event manager time sequence diagram

in form of a `.xml` file is included. It defines the name of the test and includes a list of drivers, stubs and units under tests. A generic test configuration is shown in the listing below. The test execution file is a `.cpp` file. It is the entry point of the simulation platform. The test sequence is written in this file in form of test steps. A set of language macros for effortlessly writing test cases is available to the user. The simulation image is then compiled and run on two different platforms: the Green Hills MULTI ARM platform and the GCC toolchain platform. Both platforms then generate a test report. Additionally, the GCC toolchain includes tools such as `LCOV` and `gcov` [52]. `LCOV` is a graphical front-end for GCC's coverage testing tool `gcov`. It collects `gcov` data for multiple source files and creates `HTML` pages containing the source code annotated with coverage information. `LCOV` supports statement, function and branch coverage measurement.

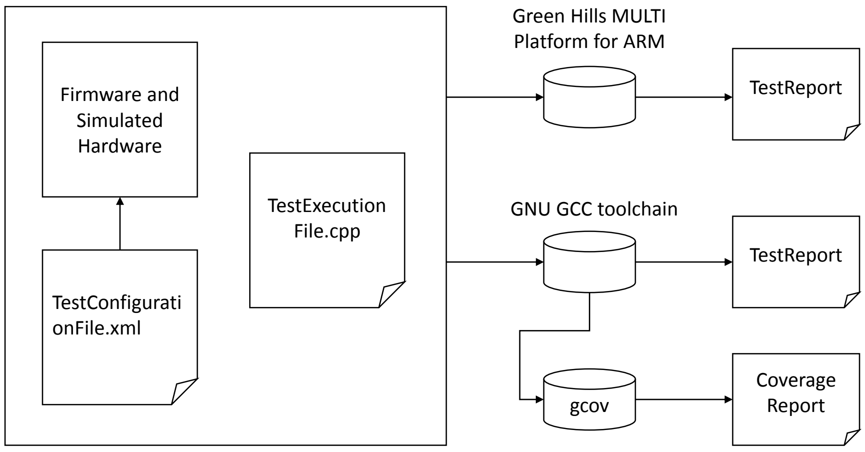

Firmware and Simulated Hardware
TestConfigurationFile.xml
TestExecution File.cpp
Green Hills MULTI Platform for ARM
TestReport
GNU GCC toolchain
TestReport
gcov
Coverage Report

Figure 5.11.: Test environment configuration

```
<?xml version="1.0" encoding="iso-8859-1"?>
<UnitTests xmlns:xi="http://www.w3.org/2001/XInclude">
<UnitTest name="IntegrationTest">
        <Drivers>
         ...
        </Drivers>

        <UnitsUnderTest>
         ...
        </UnitsUnderTest>

        <UsedStubs>
         ...
        </UsedStubs>
</UnitTest>
</UnitTests>
```

Listing 5.1: Test specification .xml file

Table 5.1.: Time events and duration

Time Event ID	Duration [μs]
1	100
2	100
3	100
4	1000
5	1500
6	500
7	500
8	100
9	100
10	100
11	100
12	100
13	1000
14	100
15	100
16	100
17	100
18	100
19	100
20	100

6. Implementation

In this chapter the proposed simulation platform is implemented in an use-case. The implementation use-case covers failsafe address assignment of a peripheral module. In section 2.2, a number of safety measures have been introduced. Another safety specific feature is the failsafe address assignment. To ensure that the data is exchanged between the F-CPU and the correct peripheral module, every module gets an unique address assigned. At start-up, the F-CPU checks the address of every module against the expected address. If the address does not match, the system goes in a failsafe state.

6.1. PROFIsafe Address Assignment

Every failsafe module has its unique PROFIsafe address, which consists of a destination and a source address. The source address refers to the F-CPU. Destination address is different for every module. When using a PLC system with one F-CPU, then the source address is the same for every module in the configuration. The uniqueness of the F-address is ensured by the destination addresses. The F-Address is stored as `F-Reference` in a non-volatile storage element (e.g., EEPROM). The PROFIsafe address must be assigned to every module before commissioning. There are also other cases which involve address reassignment such as:

- later placement of a failsafe module during initial commissioning,
- intentional modification of the source address parameter for the associated F-CPU,
- replacement of the non-volatile element and
- replacement of the failsafe module after commissioning.

To assign the address, there are basically two approaches: to manually set the address using a hardware switch and to use an engineering station (e.g., laptop) equipped with engineering software (e.g., SIEMENS TIA Portal).

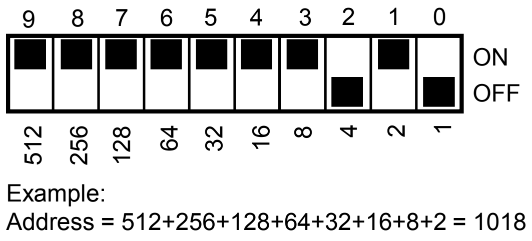


Figure 6.1.: Setting the failsafe address using a hardware switch

6.1.1. Manual Setting Using a Hardware Switch

The classic solution is based on manual setting of the F-Address using a hardware address switch (10-pin DIP switch). The valid range of the addresses is 1 to 1022. Not all product families have the DIP switch (e.g., the `ET200SP` product family) and not every module is always accessible. Environmental conditions (e.g., poor light conditions) make the hardware accessibility difficult. Alternatively, the address is assigned using the engineering platform (e.g., laptop) equipped with the engineering software (TIA Portal). However, this method requires the availability of an engineering station at the commissioning site. Furthermore, all modules must be online during the entire process, which is not possible in every plant.

6.1.2. Address Assignment via Engineering Station

The address assignment process via engineering station is performed using the SIEMENS TIA Portal software package installed on a computer. The software then communicates with the F-CPU. A central source address is assigned to the F-CPU and destination addresses are assigned to modules. The destination address, however, shall be a 16-bit number between 1024 and 64534 to avoid any possible address conflicts with numbers assigned using the hardware switch. Following section briefly illustrates the steps used for address assignment on an example project shown in the Figure 6.2. The hardware configuration consists of a CPU and a decentralized `et200sp` peripheral station including one failsafe module, to which the address is assigned to. The connection between the CPU and the peripheral group is realized with a PROFINET connection. The assignment process is performed using the following steps:

1. In TIA Portal, a right-click on the station opens the context menu where

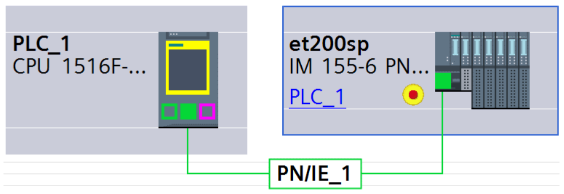


Figure 6.2.: Implementation configuration in TIA Portal

to user clicks on "Assign PROFIsafe address". (Figure 6.3)

2. A pop-up window shows up. A click on "Identification" button flashes the module. The module LEDs start to flash green. Simultaneously, a timer starts. The user has 60 seconds before a timeout occurs. (Figure 6.4)
3. After the user clicks "Assign PROFIsafe address" button, a final confirmation dialog pops up. Again, a new timer is started and the user has 60 seconds to confirm the assignment. If the user clicks "Yes", the assignment process is confirmed. Otherwise, if the user quits with clicking the "No" button or the timeout is over, the module stops flashing and the assignment process is aborted. (Figure 6.5)

The same process is shown in the time sequence diagram in Figure 6.6. The user interacts with TIA Portal interface. Each user action results in sending information over the F-CPU. The F-CPU then communicates with the peripheral module using data records.

6.1.3. HW-SW Integration Test Cases

The `ET200SP F-Module` test specification file[1] defines the test environment and all the test cases for the HW/SW integration test of the address assignment process mentioned above. The numbered test cases, including a short description is shown in the Table 6.1. The table additionally features the time needed to complete each test cases. The test duration is later discussed in the Section 7.2. There are in total 13 test cases featuring different scenarios. Each test case has exactly defined test steps and expected diagnosis data in the specification file. Based on that information, the simulation-based test cases were written. Ultimately, the goal is to execute all the test cases on the simulation platform.

[1]Confidential company IP

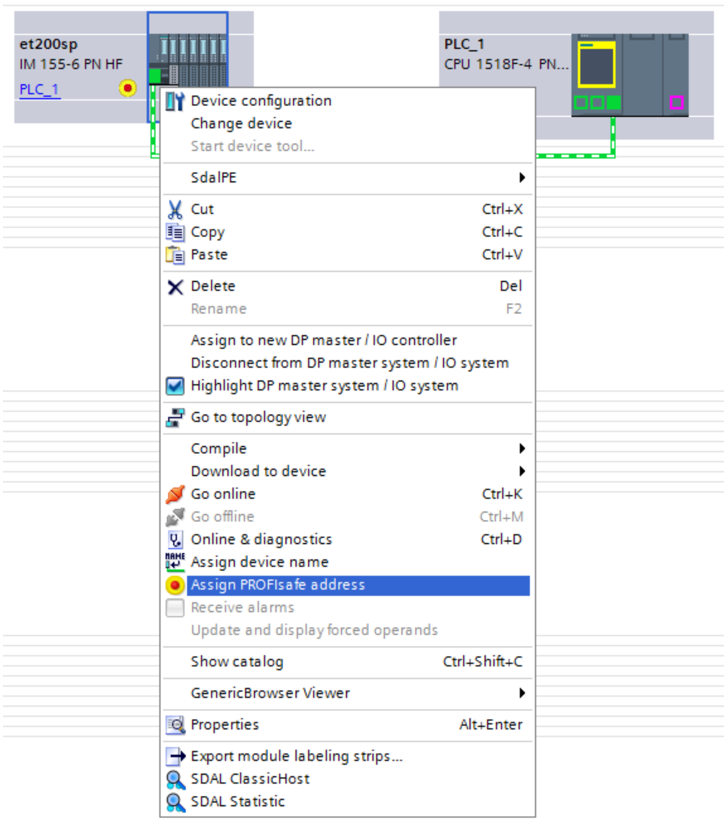
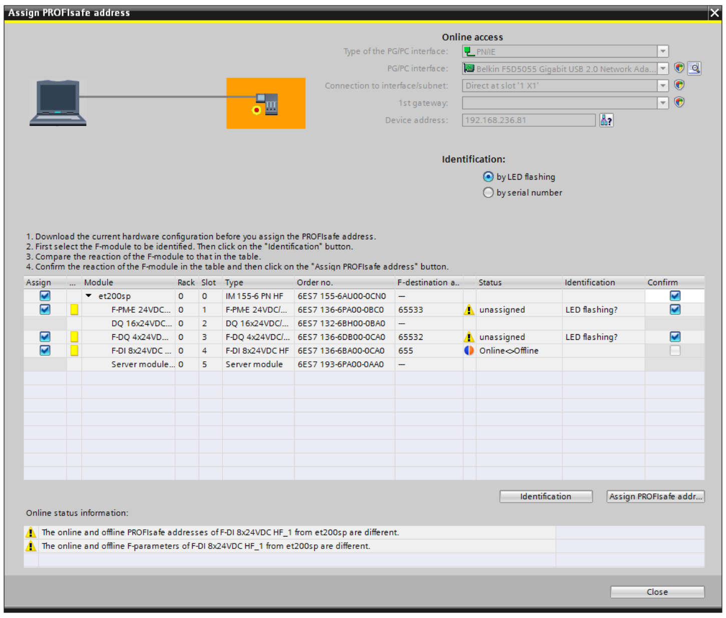


Figure 6.3.: Step 1 of the address assignment in TIA Portal



Online access

Type of the PG/PC interface: PN/IE

PG/PC interface: Belkin F5D5055 Gigabit USB 2.0 Network Ada...

Connection to interface/subnet: Direct at slot '1 X1'

1st gateway:

Device address: 192.168.236.81

Identification:

[✓ by LED flashing] by serial number

1. Download the current hardware configuration before you assign the PROFIsafe address.
2. First select the F-module to be identified. Then click on the "Identification" button.
3. Compare the reaction of the F-module to that in the table.
4. Confirm the reaction of the F-module in the table and then click on the "Assign PROFIsafe address" button.

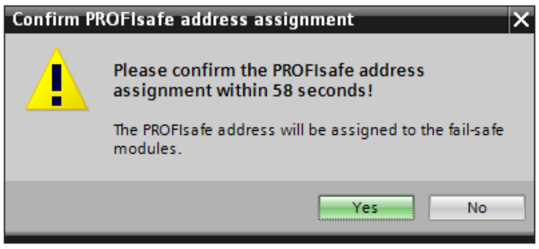
Assign	...	Module	Rack	Slot	Type	Order no.	F-destination a..		Status	Identification	Confirm
✓		et200sp	0	0	IM 155-6 PN HF	6ES7 155-6AU00-0CN0	—				✓
✓	🟨	F-PM-E 24VDC...	0	1	F-PM-E 24VDC/...	6ES7 136-6PA00-0BC0	65533	⚠	unassigned	LED flashing?	✓
		DQ 16x24VDC...	0	2	DQ 16x24VDC/...	6ES7 132-6BH00-0BA0	—				
✓	🟨	F-DQ 4x24VD...	0	3	F-DQ 4x24VDC/...	6ES7 136-6DB00-0CA0	65532	⚠	unassigned	LED flashing?	✓
✓	🟨	F-DI 8x24VDC ...	0	4	F-DI 8x24VDC HF	6ES7 136-6BA00-0CA0	655	◐	Online<>Offline		☐
		Server module...	0	5	Server module	6ES7 193-6PA00-0AA0	—				

Online status information:

⚠ The online and offline PROFIsafe addresses of F-DI 8x24VDC HF_1 from et200sp are different.

⚠ The online and offline F-parameters of F-DI 8x24VDC HF_1 from et200sp are different.

Figure 6.4.: Step 2 of the address assignment in TIA Portal



Please confirm the PROFIsafe address assignment within 58 seconds!

The PROFIsafe address will be assigned to the fail-safe modules.

Figure 6.5.: Step 3 of the address assignment in TIA Portal

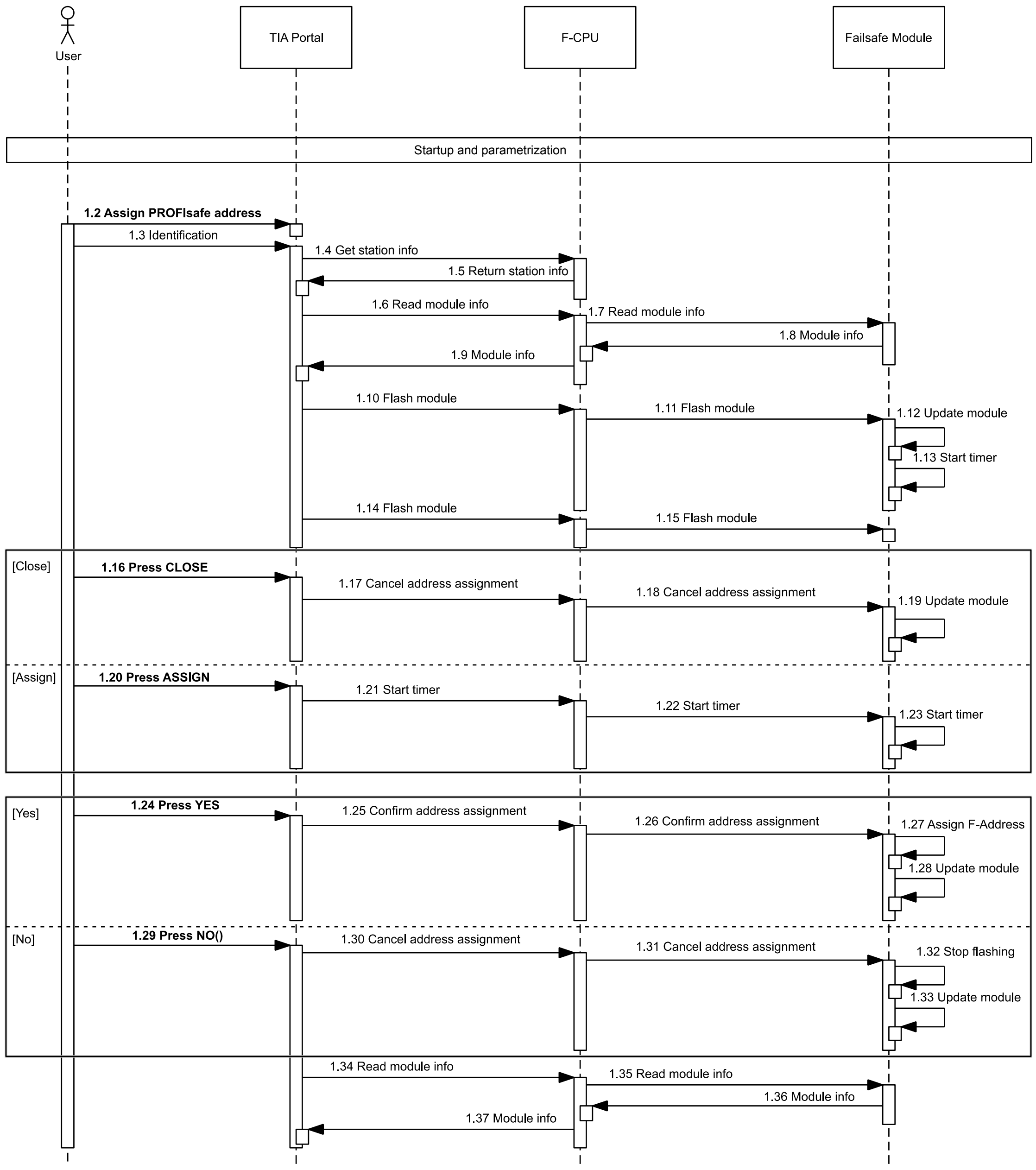


Figure 6.6.: TIA Portal address assignment time sequence diagram

Table 6.1.: HW-SW integration test cases

Test ID	Description	Duration [min]
1	Assign a new (valid) failsafe address to an unaddressed F-module.	5
2	Assign a new (valid) failsafe address to an invalid addressed F-module (destination address mismatch).	3
3	Assign a new (valid) failsafe address to an invalid addressed F-module (source address mismatch).	3
4	Assign a new (valid) failsafe address to an invalid addressed F-module (source and destination address mismatch).	2
5	Startup with non-volatile storage element containing invalid address verification information (1/2).	3
6	Startup with non-volatile storage element containing invalid address verification information (2/2).	2
7	Power up an F-module without the non-volatile storage element.	1
8	Power up F-module without a non-volatile storage element and in an environment where no parameterization is attempted.	20
9	Cancel address assignment.	2
10	Let connection timer expire.	3
11	Let address assignment timer expire.	3
12	Verify that the session ID changes during an ongoing address assignment.	5
13	Check system behavior in case that users try to concurrently assign addresses to the same F-module from two individual engineering Stations.	10
	TOTAL	**62**

6.2. Virtual Address Assignment

Following the same modeling principles in Chapter 5 and based on the description of address assignment using the engineering station covered above, the simulation-based counterpart has been modeled using C++. The class also includes the simulation of the non-volatile storage element. One can set the content of the storage element in the test environment or even make the element not accessible (used in test ID 7 and 8). The complete address assignment functionality is implemented in the state machine with nine states in total:

- `SendConfigurationData`
- `GetModuleInfo`
- `FlashModule1`
- `FlashModule2`
- `StartTimer`
- `Cancel`
- `Confirm`
- `Assigned`
- `Canceled`

What states are executed depends on the test case. The activity diagram is shown in the Figure 6.7. For the default test case for assigning a new failsafe address (test ID: 1 to 4 in Table 6.1), the program goes into all states except for `Cancel` and `Canceled`. More execution details about each state is shown in a time sequenced diagram in the Figure 6.8. Compared to the actual assignment process shown in Figure 6.6, in the simulation-based one the user and TIA Portal actions are represented in the test environment. The communication between the virtual components is done using the simulated asynchronous data exchange using data records. In a given state, the data record structure is prepared. After the data record is initialized, the data record is assigned to the simulated bus interface between the `F-CPU` and `Module`. This is realized using `FCPUSimulation.cpp` and `PROFISafeSimulation.cpp` classes from Section 5.3.1). After that, the module cycle is executed. The data record is interpreted in the firmware and module is updated. Depending on the test case, the state machine ends in either `Assigned` or `Canceled` state. In the `End` state, the address assignment result is returned to the test environment.

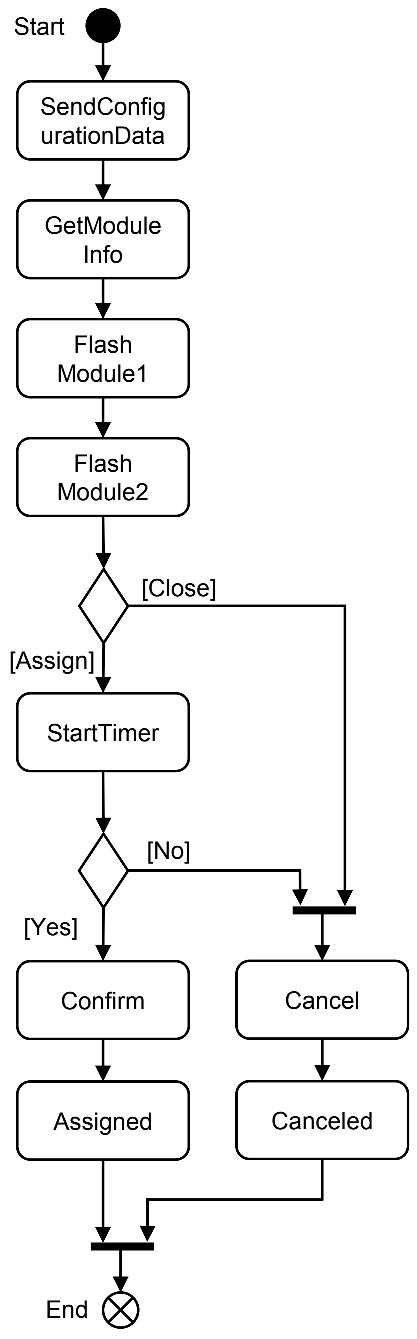


Figure 6.7.: Activity diagram for the simulation-based address assignment

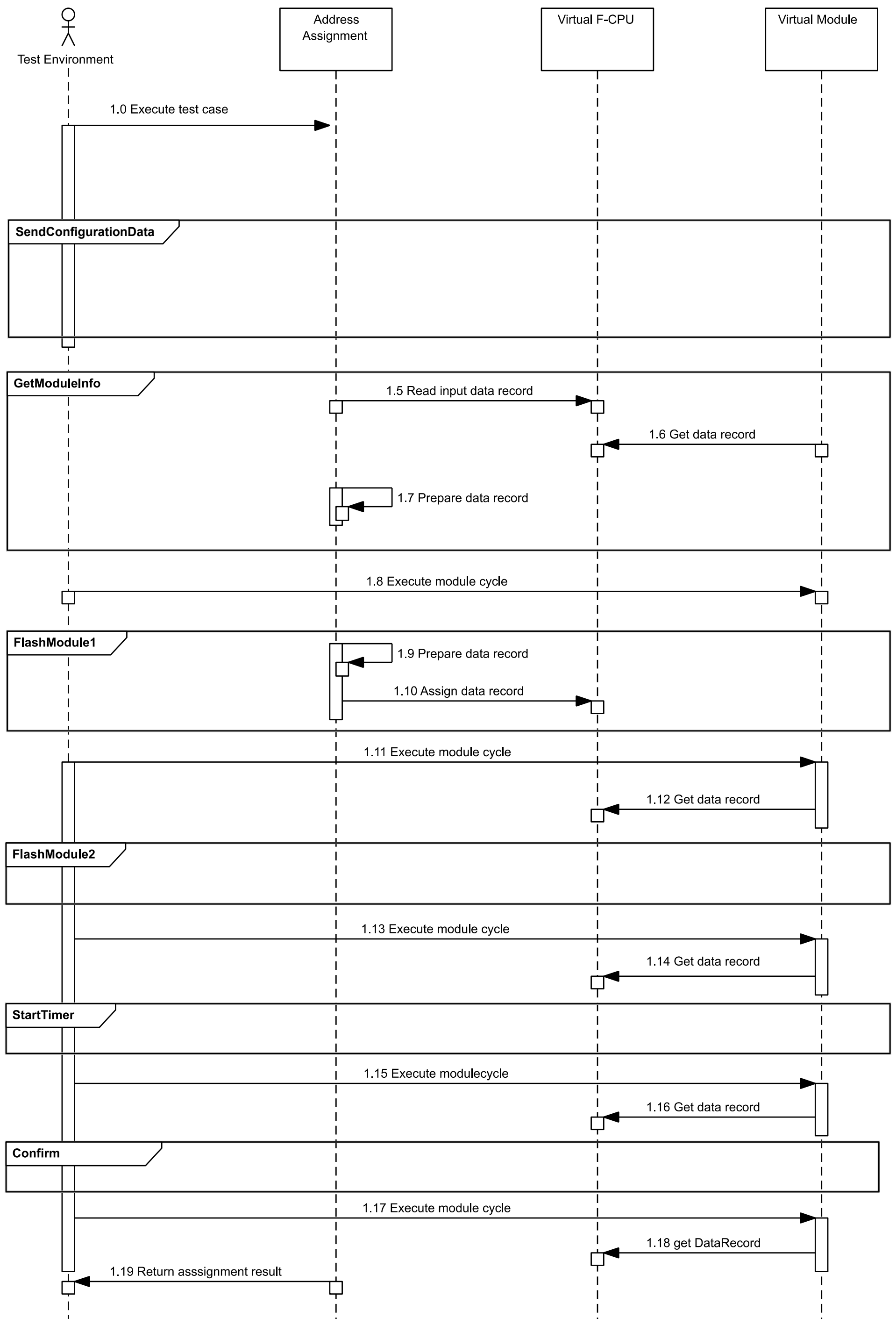


Figure 6.8.: Simulation-based address assignment time sequence diagram

6.3. Improving Test Coverage

Having the address assignment implemented in the simulation platform, there is a possibility not only to write identical test cases from Subsection 6.1.3, but also to include additional simulation-based test cases. Additional test cases can be written by using the testing techniques introduced in the Chapter 3.

Class Partitioning and Boundary Analysis

Given the fact that the complete address assignment process is based on the data record communication, black-box techniques such as equivalence class partitioning, boundary value analysis and fault injection can be applied to the communication package throughout the assignment process. This provides several test cases per each state described in the Figure 6.7. The black-box testing approach applied to the data record communication is illustrated in the Figure 6.9. The figure represents a generic data record with N data blocks. The data record is partitioned so that each data block represents an equivalence class, which means at least N test cases. Applying the boundary value analysis and selecting a minimum, maximum and middle value for each equivalence class, a minimum of $3N$ test cases is achieved.

Fault-injection

Fault-injection is another method to write additional test cases. This technique is also applied on the data-record communication. In each data record sent throughout the assignment process a fault has been injected, resulting with additional test cases. To demonstrate how the test coverage can be increased, seven additional test cases are written using class partitioning and fault-injection methods on specific data records. One could generate even more test cases but these were sufficient to show the coverage increase. The results are shown and discussed in the chapter below.

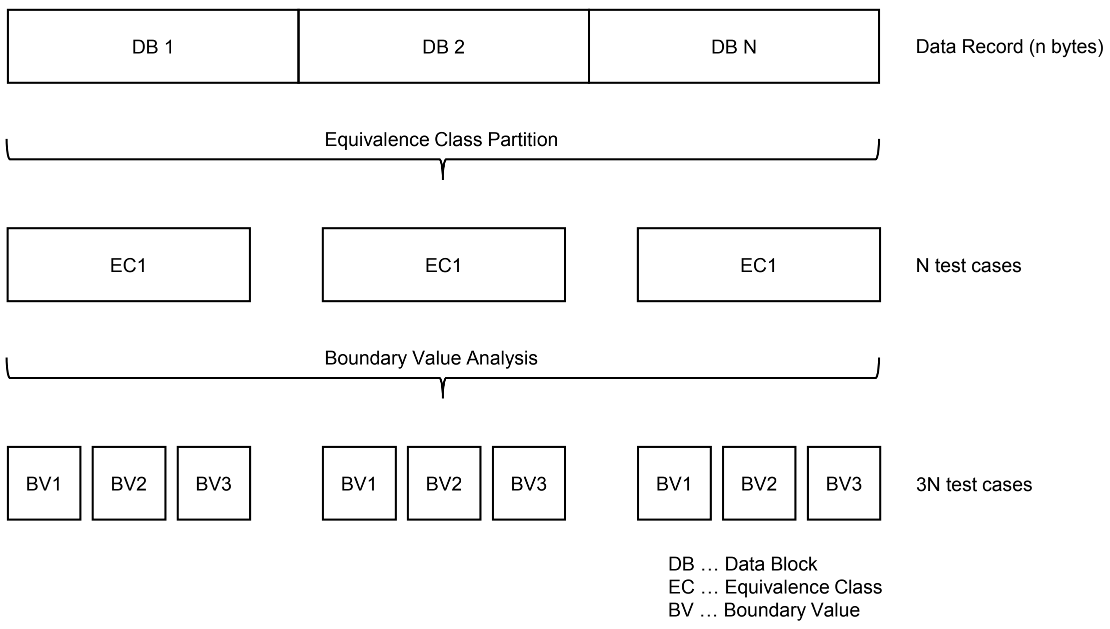


Figure 6.9.: Additional test cases derived from data records

7. Results and Evaluation

7.1. Test Coverage

Table 7.1 shows the test coverage for each test case included in Table 6.1. The coverage data is extracted individually from the `lcov` coverage report for each test ID, for the test cases in total and the additional written test cases. The `gcov` coverage tool measures the line, function and decision coverage for the unit under test classes, for which the framework classes[1] associated with the address assignment process are selected. Lowest test coverage was archived in the test IDs 7, 8 and 9. This was expected, as the assignment process is started without the coding element or, as for test ID 9, the assignment process is canceled at the beginning. Highest test coverage was archived for test IDs 1 to 4. This was also expected as the assignment process is completed in these cases. A total test coverage of 78,1% for line, 89,7% for function and 59,3% for decision coverage is archived. The additional test cases showed an increase of approximately 2%. Although a modest value, it showed how to increase the coverage writing additional simulation-based cases. The above mentioned data records have two protocol version. It should be noted that the implementation covered only one protocol version. By implementing the test cases with the second protocol versions, a slightly higher coverage would be archived. The coverage data reported in this thesis have been observed using pure black box tests, therefore only slight coverage improvements have been achieved. With carefully selected white box tests, these coverage values can easily be brought to the maximum, at least for function and line coverage (branch coverage would require more effort). It is infeasible to archive this test coverage using traditional HW-SW integration tests.

[1]Confidential company IP

Table 7.1.: Test coverage for virtual HW-SW integration test cases

Test ID	Coverage [%]		
	Line	Function	Decision
1	72,5	87,2	46,2
2	72,5	87,2	46,2
3	72,5	87,2	46,2
4	72,5	87,2	46,2
5	31,8	74,4	11,1
6	31,8	74,4	11,1
7	23,8	69,2	7,0
8	16,2	53,8	5,0
9	16,2	53,8	5,0
10	57,2	84,2	35,2
11	62,6	84,6	42,2
12	58,6	84,6	39,7
13	61,2	84,6	41,2
Total	**76,2**	**87,2**	**57,3**
Additional	78,1	89,7	59,3
Increase	1,9	2,5	2,0

7.2. Test Resource Savings

Table 6.1 includes for each HW-SW integration test case the time duration needed to complete the test. The time values are approximate and were measured using a stopwatch. The preparation for the testing is not taken into account. It shall also be noted that the values highly depend on the testers experience. For the sake of this thesis, they were conducted by a tester with several years of experience in integration testing. For most test cases, it took about 2 to 3 minutes to complete. Test case with ID 8 took most time to be completed, a total of 20 minutes but this was because the tester has to wait for the maximal timeout value to be reached.

Compared to the hardware test cases, the simulation-based counterparts are executed almost instantaneously. The execution time depends on the computing power, but a standard PC configuration executes them in a few seconds. However, the compilation takes some more time: approximately 10 minutes. The code is however, compiled just once. This means that, after compiling the project, the test reports can be finalized within few minutes. The comparison is made to show that the tester can have the test report in a matter of minutes instead of setting up the hardware test environment and waiting for hours. This is especially useful when making small changes in code or when testing new functionalities.

7.3. Discussion

The simulation-based testing approach certainly offers some advantages compared to the traditional testing. One of these is the ability to write highly unlikely test scenarios A good example is the Session ID test case. In the address assignment process the Session ID is generated by a hardware random number generator. Although the probability to generate two identical session numbers is highly unlikely, it is still conceivable but it is almost impossible to test this scenario using the traditional HW-SW integration test methods. It is doubtful if this scenario would occur even with years of testing. With the use of simulation-based testing however, even highly unlikely scenarios like this one can be written and simulated with just few lines of code.

Another feature is the simulation of hardware fault injections. A good example is the defective non-volatile storage element. For testing this scenario, one would require special hardware tools. However, by using a software model of the non-volatile storage element, one only needs to modify the code. The same applies for basically every hardware model in the simulation platform.

Finally, the simulation platform offers a possibility to add and test functionalities for new peripheral modules before the delivery of the first hardware prototype. This results in early verification of the system and can reduce the number of faults in later stages.

However, there are also some shortcomings of the simulation platform. Just like with every other simulation, one should keep in mind that it is after all, only a simulation and can-not be compared to the real system in terms of accuracy and level of detail, especially when the platform was modeled on a system level of abstraction, where accuracy is traded for modeling effort. An example is the modeling of the timing system. In real hardware, the system is driven by a hardware clock whereas the timing system in the simulation was realized by approximate code instrumentation.

Another problem is that the simulation platform is device specific. It was written for one peripheral module with specific microcontroller. For a new peripheral module, with new hardware architecture, a new simulation platform must be written from scratch. Thus, it is not reusable. Apart from that, even for existing modules, the simulation code leads to a increased code complexity and results in decreased maintainability of the code.

At this place, with all the advantages and disadvantages in mind, it shall be noted that the simulation-based testing is by no means meant to substitute the traditional HW-SW integration test. It is rather a supplement to the hardware testing. After all, the product would never get certified without testing it on all levels in accordance with IEC 61508. However, simulation-based testing provides valuable information about integration testing such as coverage overview. Additionally, highly unlikely test scenarios can be tested. Lastly, the simulation platform allows to carry out integration test cases long before the first hardware prototype is available. When implementing new functionalities, one could easily integrate it and verify its compatibility with the system.

8. Conclusion

8.1. Summary

This thesis covers the development of failsafe peripheral modules which are used in safety-critical industrial systems. Thus, it features a number of hardware and software measures to ensure functional safety. The modules are developed according to the IEC 61508 standard. For the software part, IEC 61508 recommends the V-model. It is explained in the Chapter 3 along with software testing methods used for verification of the failsafe software. The chapter was finalized with the introduction of simulation-based techniques in testing, the term which is well-established in automotive industry. The idea of introducing a software-in-the-loop simulation as a supplementation method for integration testing is presented. This means simulation-based testing with no use of real hardware, except a host machine. Chapter 4 dealt with modeling embedded systems. A well established field in embedded systems development. Literature research showed a number of different modeling platforms and languages for system-level modeling. Some of them are commercial and some open-source tools. However, to integrate any of these methods into current development process, the associated tools would have to pass a special certification process. For this reason, a hardware modeling approach using C/C++ language and integrated in existing tools was chosen. One of the tools is the Green Hills Software MULTI environment which features an instruction set simulator for the target ARM processor. The simulation platform is handled in the Chapter 5. It features the above-mentioned ISS and hardware peripherals modeled as C++ classes. Additionally, the platform features a test environment which, among others, incorporates a simulation event manager. The simulation event manager realizes a virtual time system used for diagnostic and logging during the simulation-based testing. Another powerful feature of the simulation platform is the modular software architecture. Different hardware models can be easily added/removed to support various use-cases. In scope of this thesis, the simulation platform was implemented to realize a specific simulation-based hardware-software test case. For this purpose, the address assignment test case is chosen. It was not only possible to write identical simulation-based test cases, but for the first time, one got a transparent overview of the testing code coverage. Not only that, but the possibility to write additional test cases, even ones that with a highly unlikely occurrence is presented. Ultimately, the simulation-based platform allows software development to some extent without a real hardware prototype.

8.2. Future Work

Future work certainly involves adapting additional HW-SW integration tests in the simulation platform. This would automatically mean the expanding and improving of the simulation platform with new hardware models. The simulation platform could further be improved by introducing an online GUI-based testing tool. It would provide real-time feedback of the simulation-based tests. This would include event and diagnosis logging output. Additionally, the status of the diagnostic LEDs would be shown and recorded at the same time. Finally, the tool could also be extended to feature a new GUI for coverage which would provide a transparent coverage overview over every firmware component. Coverage data would be exported from the lcov report. The program would then plot the firmware components as squares. Each square would have two attributes: size and color. The size of the square would be related to the size of the software component (e.g., lines of code), while the color of the square would be determined by the coverage. This would provide an unique and even better feedback from the hardware-software integration test to the user.

Bibliography

[1] Frank D Petruzella. *Programmable logic controllers.* Tata McGraw-Hill Education, 2005 (cit. on pp. 1, 6).

[2] Ron Bell. “Introduction to IEC 61508.” In: *Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55.* Australian Computer Society, Inc. 2006, pp. 3–12 (cit. on p. 1).

[3] David J Smith and Kenneth GL Simpson. *The Safety Critical Systems Handbook: A Straightforward Guide to Functional Safety: IEC 61508 (2010 Edition), IEC 61511 (2015 Edition) and Related Guidance.* Butterworth-Heinemann, 2016 (cit. on p. 2).

[4] International Electrotechnical Commission. *IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements.* International Standard. 2010 (cit. on pp. 1, 13, 14, 18).

[5] Jakob Engblom, Guillaume Girard, and Bengt Werner. “Testing Embedded Software using Simulated Hardware.” In: *ERTS 2006* (2006), pp. 1–9 (cit. on pp. 3, 34).

[6] SIEMENS. *Failsafe I/O Module SM 526.* URL: `https://w3.siemens.com/mcms/distributed-io/en/ip20-systems/et-200mp/io-modules/pages/failsafe-io.aspx` (cit. on p. 7).

[7] International Electrotechnical Commission. *IEC 61508-6: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3.* International Standard. 2010 (cit. on p. 8).

[8] PROFIsafe. *PROFIsafe System Description: Technology and Application.* URL: `https://www.automation.com/pdf_articles/profinet/PROFIsafe_system_description_v_2010_English.pdf` (cit. on pp. 9–11).

[9] Stephan Grünfelder. *Software-Test für Embedded Systems: Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter*. dpunkt. verlag, 2017 (cit. on p. 13).

[10] T Bäro, E Sax, and S Schmerler. "Erhöhung der Testtiefe durch HiL-Testing." In: *Proceedings der Jahrestagung der ASIM/GI-Fachgruppe 4.5.5 Simulation technischer Systeme* December (2005), pp. 4–13 (cit. on pp. 13, 23).

[11] Ian Sommerville. *Software Engineering: Global Edition*. 2016, p. 811. ISBN: 978-1-292-09613-1 (cit. on pp. 15, 17, 23).

[12] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011 (cit. on p. 16).

[13] Kelly J Hayhurst et al. *Decision Tutorial Coverage on Modified Condition /*. May. 2001. ISBN: 2001210876 (cit. on pp. 17, 19).

[14] Peter Liggesmeyer. *Peter Liggesmeyer, Software-Qualität*. Vol. 2. 2009, pp. 1689–1699. ISBN: 978-3-8274-2056-5. DOI: `10.1017/CBO9781107415324.004`. arXiv: `arXiv:1011.1669v3` (cit. on p. 18).

[15] Ilene Burnstein. *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006 (cit. on pp. 18, 20–22).

[16] International Electrotechnical Commission. *IEC 61508-7: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 7: Overview of techniques and measures*. International Standard. 2010 (cit. on p. 19).

[17] C Gühmann. "Model-Based Testing of Automotive Electronic Control Units." In: *3rd International Conference on Materials Testing: Test 2005* 49.30 (2005) (cit. on pp. 23, 24).

[18] Daniel D Gajski and Robert H Kuhn. "New VLSI tools." In: *Computer* 12 (1983), pp. 11–14 (cit. on p. 26).

[19] Daniel D. Gajski et al. *Embedded System Design*. Vol. 35. 1. 2009, p. 366. ISBN: 978-1-4419-0503-1. DOI: `10.1007/978-1-4419-0504-8` (cit. on pp. 26, 27, 31).

[20] Bart Kienhuis et al. "A methodology to design programmable embedded systems." In: *International Workshop on Embedded Computer Systems*. Springer. 2001, pp. 18–37 (cit. on pp. 27, 28).

[21] Stefan Eilers. "Zeitgenaue Simulation gemischt virtuell-realer Prototypen." In: (2006), pp. 27–28 (cit. on pp. 29, 30).

[22] Daniel D Gajski, Rainer Dömer, and Jianwen Zhu. "Ip-centric methodology and design with the specc language." In: *System-Level Synthesis.* Springer, 1999, pp. 321–358 (cit. on p. 31).

[23] Daniel D. Gajski et al. *SPECC: Specification Language and Methodology.* 2000, p. 313. ISBN: 0792378229. DOI: `10.1007/978-1-4615-4515-6` (cit. on pp. 31, 32).

[24] Accellera. *SystemC.* URL: `https://accellera.org/downloads/standards/systemc` (cit. on p. 31).

[25] Daniel Große and Rolf Drechsler. *Quality-Driven SystemC Design.* Springer, 2010 (cit. on p. 31).

[26] David C. Black et al. *Systemc: From the ground up.* 2005, pp. 1–279. ISBN: 9781402080876. DOI: `10.1007/978-0-387-69958-5`. arXiv: `arXiv:1011.1669v3` (cit. on pp. 32, 33).

[27] Thorsten Grotker. "System Design with SystemC." In: *Journal of Experimental Psychology: General* 136.1 (2007), pp. 23–42 (cit. on p. 32).

[28] Jon Perez, Mikel Azkarate-Askasua, and Antonio Perez. "Codesign and simulated fault injection of safety-critical embedded systems using systemC." In: *EDCC-8 - Proceedings of the 8th European Dependable Computing Conference* MiL (2010), pp. 221–229. DOI: `10.1109/EDCC.2010.34` (cit. on p. 32).

[29] N. Calazans et al. "From VHDL register transfer level to SystemC transaction level modeling: A comparative case study." In: *Proceedings - 16th Symposium on Integrated Circuits and Systems Design, SBCCI 2003* (2003), pp. 355–360. DOI: `10.1109/SBCCI.2003.1232853` (cit. on p. 32).

[30] Luc Séméria. "Methodology for Hardware / Software Co-verification in C / C ++." In: () (cit. on p. 33).

[31] Diederik Verkest, Joachim Kunkel, and Frank Schirrmeister. "System level design using C++." In: *Proceedings of the conference on Design, automation and test in Europe - DATE '00* February 2000 (2000), pp. 74–83. DOI: `10.1145/343647.343709`. URL: `http://dl.acm.org/citation.cfm?id=343709%7B%5C%%7D5Cnhttp://portal.acm.org/citation.`

cfm?doid=343647.343709%7B%5C%%7D5Cnhttp://portal.acm.org/citation.cfm?doid=343647.343709 (cit. on p. 33).

[32] Abhijit Ghosh et al. "Hardware Synthesis from C / C ++." In: *Design* () (cit. on p. 33).

[33] Wind River. *Simics Website.* URL: https://www.windriver.com/products/simics/ (cit. on p. 34).

[34] Synopsis. *Virtual Prototyping Website.* URL: https://www.synopsys.com/verification/virtual-prototyping.html (cit. on p. 34).

[35] Cadence. *Cadence Virtual System Platform.* URL: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/Archive/virtual_system_platform_ds.pdf (cit. on p. 34).

[36] Open Virtual Platforms. *Open Virtual Platforms Website.* URL: https://www.windriver.com/products/simics/ (cit. on p. 34).

[37] Filippo Cucchetto, Alessandro Lonardi, and Graziano Pravadelli. "A common architecture for co-simulation of SystemC models in QEMU and OVP virtual platforms." In: *IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC* 2015-Janua.January (2015). ISSN: 23248440. DOI: 10.1109/VLSI-SoC.2014.7004154 (cit. on pp. 34, 35).

[38] Stephan Werner et al. "Software-in-The-Loop simulation of embedded control applications based on Virtual Platforms." In: *25th International Conference on Field Programmable Logic and Applications, FPL 2015* (2015). ISSN: 1946-1488. DOI: 10.1109/FPL.2015.7294020 (cit. on p. 34).

[39] QEMU. *QEMU Website.* URL: https://www.qemu.org/ (cit. on p. 35).

[40] Màrius Montón et al. "Mixed SW/systemC SoC emulation framework." In: *IEEE International Symposium on Industrial Electronics* (2007), pp. 2338–2341. ISSN: 2163-5137. DOI: 10.1109/ISIE.2007.4374971 (cit. on p. 35).

[41] Guillaume Delbergue et al. "QBox : an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2 . 0." In: (2016), pp. 1–10 (cit. on p. 35).

[42] Wolfgang Mueller et al. "Virtual prototyping of cyber-physical systems." In: *17th Asia and South Pacific Design Automation Conference.* IEEE. 2012, pp. 219–226 (cit. on p. 35).

[43] Massimiliano D'Angelo et al. "A Simulator based on QEMU and SystemC for Robustness Testing of a Networked Linux-based Fire Detection and Alarm System." In: *6th European Congress on Embedded Real-Time Software and Systems* (2012) (cit. on p. 35).

[44] Tse Chen Yeh, Zin Yuan Lin, and Ming Chao Chiang. "Enabling TLM-2.0 interface on QEMU and SystemC-based virtual platform." In: *2011 IEEE International Conference on Integrated Circuit Design and Technology, ICICDT 2011* May 2011 (2011). DOI: `10.1109/ICICDT.2011.5783207` (cit. on p. 35).

[45] Shye Tzeng Shen, Shin Ying Lee, and Chung Ho Chen. "Full system simulation with QEMU: An approach to multi-view 3D GPU design." In: *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems* (2010), pp. 3877–3880. DOI: `10.1109/ISCAS.2010.5537690` (cit. on p. 35).

[46] Daniel Aarno and Jakob Engblom. *Software and System Development using Virtual Platforms: Full-System Simulation with Wind River® Simics®*. 2014, pp. 1–349. ISBN: 9780128008133. DOI: `10.1016/C2013-0-14366-8` (cit. on pp. 35, 37).

[47] Wolfgang Ecker. *Hardware-dependent Software Principles and Practice.* Vol. 39. 5. 2008, pp. 561–563. ISBN: 9781402094354 (cit. on p. 36).

[48] Jürgen Schnerr et al. "High-performance timing simulation of embedded software." In: *Proceedings - Design Automation Conference* July (2008), pp. 290–295. ISSN: 0738100X. DOI: `10.1109/DAC.2008.4555825` (cit. on p. 36).

[49] International Electrotechnical Commission. *IEC 61508-4: Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 4: Definitions and abbreviations.* International Standard. 2010 (cit. on p. 37).

[50] STMicroelectronics. *Reference Manual: STM32F205xx, STM32F207xx, STM32F215xx and STM32F217xx advanced Arm-based 32-bit MCUs.* URL: `https://www.st.com/content/ccc/resource/technical/document/reference_manual/51/f7/f3/06/cd/b6/46/ec/CD00225773.pdf/files/CD00225773.pdf/jcr:content/translations/en.CD00225773.pdf` (cit. on pp. 39, 45).

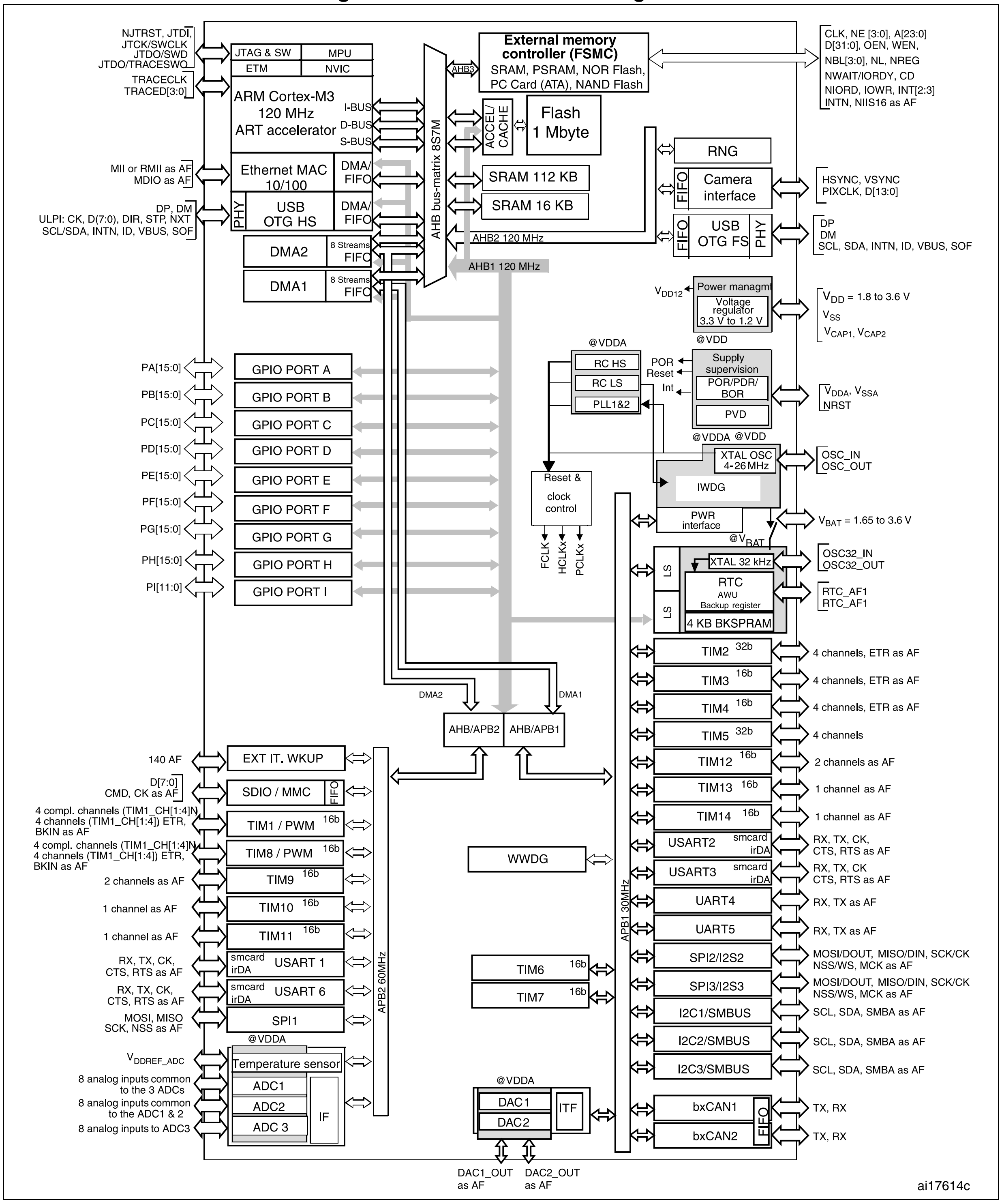
[51] Gren Hills Software. *MULTI development environment.* URL: `https://www.ghs.com/products/xilinx_zynq.html` (cit. on p. 42).

[52] GNU GCC. *Introduction to gcov.* URL: `https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro` (cit. on p. 51).

Appendix

Appendix A.
STM32F2xx Block Diagram

Figure 4. STM32F20x block diagram



NJTRST, JTDI, JTCK/SWCLK JTDO/SWD JTDO/TRACESWO
TRACECLK TRACED[3:0]
JTAG & SW
MPU
ETM
NVIC
ARM Cortex-M3 120 MHz ART accelerator
I-BUS
D-BUS
S-BUS
MII or RMII as AF MDIO as AF
Ethernet MAC 10/100
DMA/ FIFO
DP, DM ULPI: CK, D(7:0), DIR, STP, NXT SCL/SDA, INTN, ID, VBUS, SOF
PHY
USB OTG HS
DMA2
DMA1
8 Streams FIFO
AHB bus-matrix 8S7M
AHB3
External memory controller (FSMC)
SRAM, PSRAM, NOR Flash, PC Card (ATA), NAND Flash
CLK, NE [3:0], A[23:0] D[31:0], OEN, WEN, NBL[3:0], NL, NREG NWAIT/IORDY, CD NIORD, IOWR, INT[2:3] INTN, NIIS16 as AF
ACCEL/ CACHE
Flash 1 Mbyte
SRAM 112 KB
SRAM 16 KB
AHB2 120 MHz
AHB1 120 MHz
RNG
FIFO
Camera interface
HSYNC, VSYNC PIXCLK, D[13:0]
USB OTG FS
DP DM SCL, SDA, INTN, ID, VBUS, SOF
VDD12
Power managmt
Voltage regulator 3.3 V to 1.2 V
VDD = 1.8 to 3.6 V
VSS
VCAP1, VCAP2
@VDDA
@VDD
RC HS
RC LS
PLL1&2
POR Reset Int
Supply supervision
POR/PDR/ BOR
PVD
VDDA, VSSA NRST
@VDDA @VDD
XTAL OSC 4-26MHz
OSC_IN OSC_OUT
IWDG
Reset & clock control
FCLK HCLKx PCLKx
PWR interface
VBAT = 1.65 to 3.6 V
@VBAT
LS
XTAL 32 kHz
RTC AWU Backup register
4 KB BKSPRAM
OSC32_IN OSC32_OUT
RTC_AF1 RTC_AF1
PA[15:0] GPIO PORT A
PB[15:0] GPIO PORT B
PC[15:0] GPIO PORT C
PD[15:0] GPIO PORT D
PE[15:0] GPIO PORT E
PF[15:0] GPIO PORT F
PG[15:0] GPIO PORT G
PH[15:0] GPIO PORT H
PI[11:0] GPIO PORT I
DMA2
DMA1
AHB/APB2
AHB/APB1
TIM2 32b 4 channels, ETR as AF
TIM3 16b 4 channels, ETR as AF
TIM4 16b 4 channels, ETR as AF
TIM5 32b 4 channels
TIM12 16b 2 channels as AF
TIM13 16b 1 channel as AF
TIM14 16b 1 channel as AF
USART2 smcard irDA RX, TX, CK, CTS, RTS as AF
USART3 smcard irDA RX, TX, CK CTS, RTS as AF
UART4 RX, TX as AF
UART5 RX, TX as AF
SPI2/I2S2 MOSI/DOUT, MISO/DIN, SCK/CK NSS/WS, MCK as AF
SPI3/I2S3 MOSI/DOUT, MISO/DIN, SCK/CK NSS/WS, MCK as AF
I2C1/SMBUS SCL, SDA, SMBA as AF
I2C2/SMBUS SCL, SDA, SMBA as AF
I2C3/SMBUS SCL, SDA, SMBA as AF
bxCAN1 TX, RX
bxCAN2 TX, RX
FIFO
APB1 30MHz
140 AF EXT IT. WKUP
D[7:0] CMD, CK as AF SDIO / MMC FIFO
4 compl. channels (TIM1_CH[1:4]N, 4 channels (TIM1_CH[1:4]) ETR, BKIN as AF TIM1 / PWM 16b
4 compl. channels (TIM1_CH[1:4]N, 4 channels (TIM1_CH[1:4]) ETR, BKIN as AF TIM8 / PWM 16b
2 channels as AF TIM9 16b
1 channel as AF TIM10 16b
1 channel as AF TIM11 16b
RX, TX, CK, CTS, RTS as AF smcard irDA USART 1
RX, TX, CK, CTS, RTS as AF smcard irDA USART 6
MOSI, MISO SCK, NSS as AF SPI1
@VDDA
VDDREF_ADC Temperature sensor
8 analog inputs common to the 3 ADCs ADC1
8 analog inputs common to the ADC1 & 2 ADC2
8 analog inputs to ADC3 ADC 3
IF
APB2 60MHz
WWDG
TIM6 16b
TIM7 16b
@VDDA
DAC1
DAC2
ITF
DAC1_OUT as AF
DAC2_OUT as AF
ai17614c

1. The timers connected to APB2 are clocked from TIMxCLK up to 120 MHz, while the timers connected to APB1 are clocked from TIMxCLK up to 60 MHz.
2. The camera interface and Ethernet are available only in STM32F207xx devices.