Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

David Barahona Pereira

Virtual Prototypes for Embedded Software Testing: A SystemC Based Proof of Concept Implementation for Nordic Semiconductor

Master's thesis in Embedded Computing Systems
Supervisor: Milica Orlandic
June 2020

NTNU
Norwegian University of Science and Technology

David Barahona Pereira

Virtual Prototypes for Embedded Software Testing: A SystemC Based Proof of Concept Implementation for Nordic Semiconductor

Master's thesis in Embedded Computing Systems
Supervisor: Milica Orlandic
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Acknowledgment

I would like to extend my sincere thanks to my supervisors in NTNU and Nordic Semiconductor. To my NTNU supervisor, Milica Orlandic, thank you for always making time between your many obligations to provide valuable feedback and guidance throughout this thesis. To Isael Diaz, Joakim Urdahl, and Shankar Gunasekaren in Nordic Semiconductor, thank you for giving me the opportunity to execute my thesis in such an amazing company, and especially, for all the dedication, effort, and support that went into making this project possible. I would also like to extend my gratitude to Motaz Thiab for being a great companion when working together with me on this project. I would like to recognize the effort of the European Commission to sustain programs like the Erasmus Mundus Joint Master Degrees to promote and facilitate the access to first-class education in the most prestigious European universities. Completing this thesis would not have been possible without the support of my friends and family. I would like to thank all my friends in Costa Rica for always being in touch and supporting me despite the distance. I would also like to thank all the friends I made during this Masters in Germany and Norway. Thank you for all the fantastic memories and for making my time abroad such a joyful experience. Lastly, I want to extend my deepest gratitude to my family who despite the distance, has always been there to support me and be the motivation behind everything I do.

David Barahona Pereira

Abstract

The use of virtual prototypes in the semiconductor industry has proven to be a successful approach to deliver products earlier and with superior quality. This thesis focuses on implementing a system that can serve as a proof of concept to evaluate the use of virtual prototypes for embedded software testing in Nordic Semiconductor. To achieve this goal, one software test was selected, and all the hardware interactions present in this test were extracted and modelled in the form of a virtual prototype using SystemC. Once the system model was able to run that test, it was scaled to execute two more tests. Results show that the virtual prototype was able to replicate passing and failing tests, provide configurable levels of hardware observability that were not available before, and execute tests close to 150 times faster. These results could translate into shorter times for product delivery, new resources for architectural improvements, and enhancements in the test design productivity.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Background

Nordic Semiconductor is a Norwegian fabless semiconductor company with headquarters in the city of Trondheim. The company specializes in ultra-low-power wireless system on a chip (SoC) commonly found in wireless PC peripherals, fitness devices, game controllers, and many others. As the semiconductor market demands grow, embedded SoCs become more powerful and complex. These factors have started to influence more and more companies to incorporate non-traditional approaches during the design and development of their products to remain competitive.

Virtual prototypes represent a successful approach for improving productivity and reducing the time to market (TTM) of new products [1]. A virtual prototype is an executable software model of a hardware system that can run software just as a physical system would. These prototypes have been exploited by big semiconductor companies around the world. For example, Texas Instruments adopted virtual prototypes during the development of their OMAP SoC, allowing them to bring up Linux and Android on the eventual hardware in just a few days when this milestone used to take weeks or months instead. Other companies like Siemens, Altera, Fujitsu, Bosch, Hitachi, General Motors, and ARM share similar experiences with this technology.

Given the potential benefit these prototypes can provide, Nordic Semiconductor is considering to adopt them in several stages of their SoC design flow. One of the applications in which Nordic Semiconductor is planning to exploit their capabilities is the development of embedded software. Currently, the software team in charge of developing the low-level software that goes into the wireless communication software stack of Nordic Semiconductor's SoCs, test their code using a two-step approach divided into what they call unit tests and target tests.

The main goal of the unit tests is to check the software logic. Unit tests run on a host PC, so all the code sections that interact with the hardware are replaced by mock functions. These are just empty functions that let the code execute due to the absence of a hardware device that should support these interactions. On the other hand, target tests are more complex. They exercise both the software logic and the remaining pieces of code that unit tests cannot cover. Target tests cannot run on a host PC, so they require a specific hardware device to execute. For this reason, they run using development boards equipped with Nordic Semiconductor's SoCs.

During target tests, hardware interactions involve accesses to different peripherals and CPU registers. Register accesses intend to trigger a specific behaviour or to return some value needed for the execution. The idea at Nordic Semiconductor is to implement a new testing stage in which these hardware interactions could carry out in a virtual prototype. This prototype should model the behaviour of the SoC at a higher level of abstraction. By doing so, the virtual prototype can simulate whatever occurs in the hardware when a test executes, and the same target tests that used to require a physical SoC to run can now also run in any host PC.

The use of virtual prototypes for embedded software testing has a lot of potential benefits for different teams. In the case of the system architecture group, they could measure the performance of their device in the presence of the software it supports, allowing them to make significant architectural improvements. For the software team, not depending on the hardware device could mean the possibility to enable more complex testing scenarios. It could also mean that they should not wait until the hardware is ready to start testing the software, resulting in a shorter TTM for new products. Lastly, the fact that virtual prototypes work on higher abstraction levels allows them to execute faster than the real hardware, improving the test design productivity.

1.2 Thesis Statement

This thesis aims to deliver a proof of concept of how a virtual prototype modelling an SoC from Nordic Semiconductor can be used to test their embedded software. This system was developed by taking an existing generic virtual prototype as a base, expanding the functionalities of its abstract CPU model, and by creating models of different system peripherals. The first design of the virtual prototype can execute one specific software test, and different extensions allowed it to execute two more tests. Therefore, the final virtual prototype can execute a total of three tests. Benefits, limitations, and recommendations for this software testing approach are derived from the implementation so that Nordic Semiconductor can consider whether to integrate it or not into their design flow. Given the magnitude of this work, two different projects were working in parallel to achieve this goal. Both projects worked on different tasks of the hardware components modelling and verification, the simple test interface and execution, and the system refinement effort to run more tests. This thesis emphasizes the work done by this particular project but also references components borrowed from the other.

1.3 Outline

Chapter 1 includes background information to help the reader understand what motivated Nordic Semiconductor to consider virtual prototypes for embedded software testing. It states the goal behind this project and also indicates the approach taken to achieve this goal.

Chapter 2 introduces the software testing methodology at Nordic Semiconductor and explains what unit and target tests are. It describes the Multi-Protocol Service Layer (MPSL) library, more specifically the MPSL target tests and the hardware interactions carried out during the execution of one simple test. This chapter also includes a literature study about what is virtual prototyping and how it could improve the software development cycle, basics of the C++ SystemC library for hardware modelling as well as the concept of Transaction-Level Modelling (TLM). Lastly, it describes the Bauhaus generic virtual prototype from Nordic Semiconductor that was used as a base to develop the virtual prototype proof of concept.

Chapter 3 delves into the methodology followed throughout this thesis. It explains the approach taken to understand the current testing methodology and also the criteria used to select a specific set of tests to focus as a starting point. Since the project scope did not include modelling a whole SoC, this chapter explains the reasoning that led to the selection of which components to model. It also describes in a detailed manner how each of these models works. This chapter also includes a verification plan for each hardware model. It covers how it was possible to interface the virtual prototype with a simple test and describes the implementation of two extra tests. Finally, it presents the procedure used to evaluate the potential benefits of the virtual prototype implementation.

Chapter 4 is devoted to the implementation results. It presents simulations for each of the hardware models that show how they meet their functional specifications. The chapter also includes a simulation of the entire system in which different peripherals communicate with each other by using interrupts. After the virtual prototype simulation results, this section compares the virtual prototype implementation to the current testing approach. The execution of passing and failing tests was employed to verify the virtual prototype's capability to execute MPSL target tests. Lastly, the chapter compares the hardware observability of the new implementation to the current testing approach as well as test execution times.

Chapters 5 and 6 deal with results discussion and conclusions. The discussion chapter provides insights into what the results mean, why they are relevant, as well as their limitations and recommendations for future work. The conclusions wrap-up this project by presenting a summary of the most important ideas discussed during this thesis.

Chapter 2

Theory

2.1 Embedded Software Testing at Nordic Semiconductor

The software stack that makes it possible to implement wireless communication in Nordic Semiconductor's SoCs is quite complex. Wireless protocols require thousands of lines of code that interact with peripherals like clock controllers, timers, counters, radio, among others. It is because of this magnitude and complexity that each component of the software stack goes through an extensive testing process before being deployed as a part of real applications. The software team in charge of this matter currently tests their software stack using the approach depicted in Figure 2.1. This approach makes a distinction between unit tests and target tests based on their goal and on the platform in which they execute.

2.1.1 Unit Tests

Unit tests are simple and have a scope usually limited to a single source file. An important characteristic of these tests is that they execute on a host PC, which means they do not require a Nordic Semiconductor's SoC to run. Unit tests exist mostly to evaluate the logic of the software. When these source files need to perform activities that demand some action from the hardware, mock functions, which are empty, come into play so the code can continue its execution. These mock functions have the same name but different implementations from the original ones that interact with the hardware when the source code runs in an SoC.

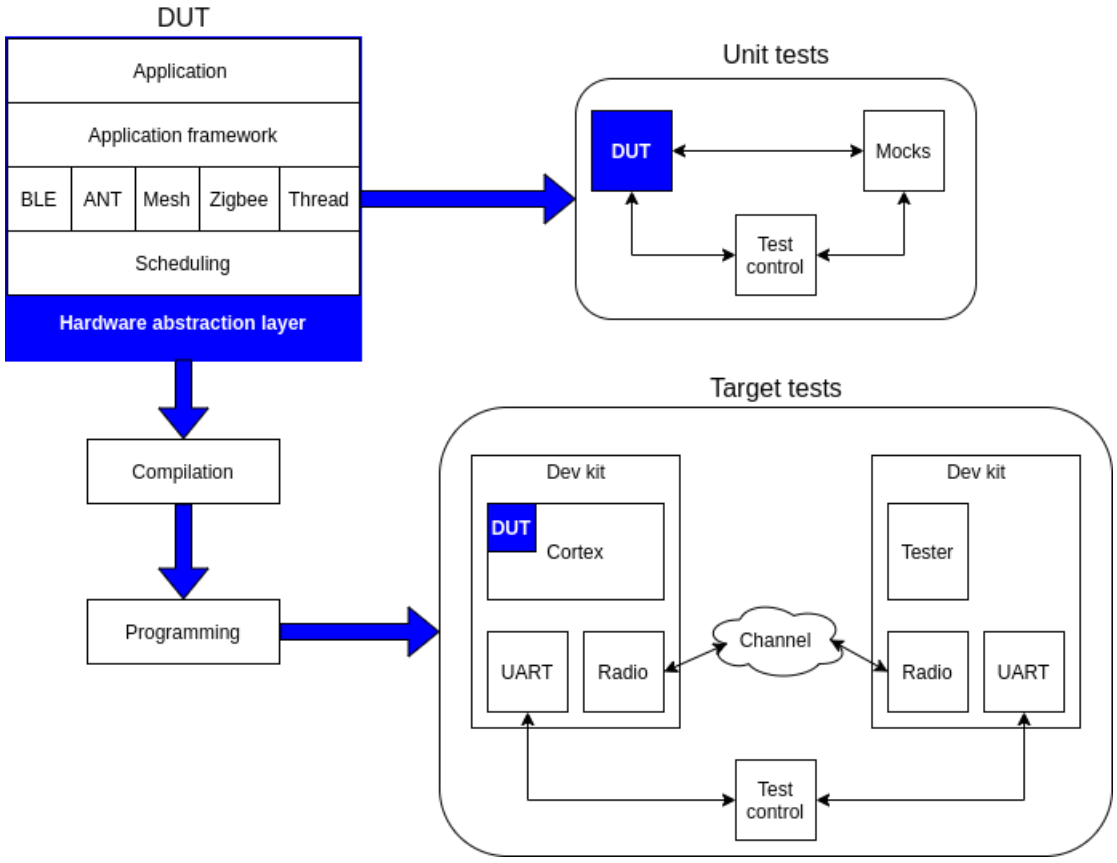


Figure 2.1: Testing approach of the embedded software team

2.1.2 Target Tests

Target tests have a broader scope than unit tests, and they exercise specific behaviours that involve one or more source files. Target tests, unlike unit tests, do not run on a host PC and require a physical connection to a Nordic Semiconductor's SoC. Because of this, they execute using development boards that hold extra circuitry for peripheral access and debugging. Since target tests run on real boards, they need to compile and flash the program into the device memory to run as a regular application would. As Figure 2.1 shows, two boards are required to execute target tests. One of the boards becomes the device under test (DUT), and the other becomes a tester. They communicate through a radio channel and coordinate the test control using the UART peripheral. However, not all the tests demand two boards to execute, tests that do not exercise any wireless communication require only a single board. These tests verify behaviours related to the clock control, timing, interrupt handling, and other functionalities in which the wireless protocols rely on to operate correctly.

Not every single file of the software stack performs some class of hardware interaction. There is a lowermost software layer that is in charge of performing most of the register accesses. Register names are the same for all Nordic Semiconductor's SoCs, but their addresses may differ between devices. The tests remain generic by referring to the same register names but associating them to different addresses. It is possible to employ header files that replace the register names with the correct address value on the SoC. These header files are also known as the hardware abstraction layer shown in Figure 2.1.

Figure 2.2 shows an example of how this abstraction works. In this particular case, a test from the Multi-Protocol Service Layer (MPSL) library reads the temperature sensor value by calling a generic initialization routine defined in the mspl.c file. This routine calls a function that initializes only the clock by modifying several registers of the clock peripheral. Inside this function, one of the modified registers is used to disable interrupts. During compilation, a library header file assigns the correct address of this specific hardware register.

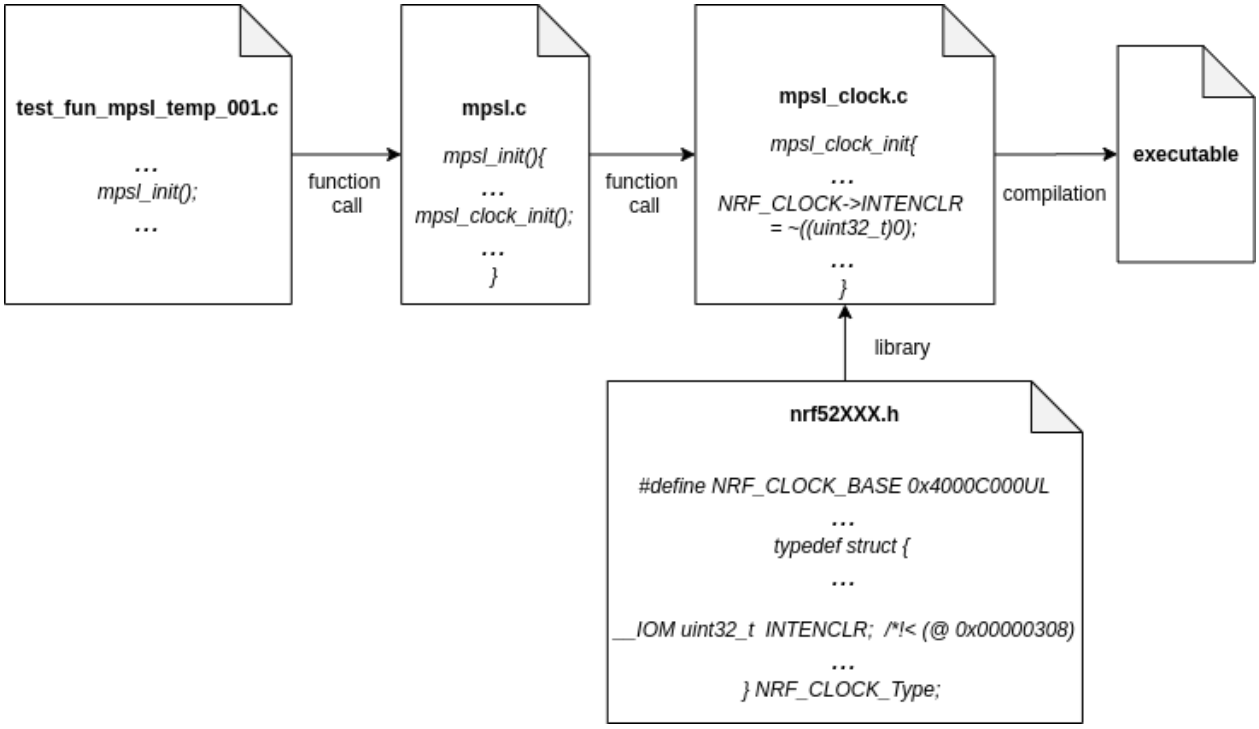


Figure 2.2: Simplified call sequence in a target test

2.2 MPSL Library Target Tests

Given the fact that target tests are present in different software stacks, this thesis focuses only on the ones from the MPSL library that require one board. MPSL includes services for single and multi-protocol wireless implementations. There are some peripherals which are exclusive to this library and should not be accessed by application software such as one real-time clock (RTC0), one timer/counter (TIMER0), the 2.4 GHz radio peripheral (RADIO), the clock control peripheral (CLOCK), and temperature sensor peripheral (TEMP) [2].

All MPSL target tests follow a similar structure as the one shown in Listing 2.1. To test different parts of the library, it is necessary to execute a shared initialization and uninitialization routine. These routines perform tasks like peripheral resets, temperature checks, clock parameter checks, clock calibration, clock triggering, among others. Usually, these two functions are considerably bigger than the actual test which is designed to target a specific behaviour. This behaviour could be returning a value from a temperature sensor, making sure a timer callback executes, or performing a particular clock calibration sequence.

```
1 void sample_mpsl_test(void){
2   mpsl_init();
3
4   /* Test body */
5
6   mpsl_uninit();
7 }
```

Listing 2.1: Structure of a MPSL target test

2.2.1 Hardware Interaction in the MPSL Init Test

The MPSL library is composed of several tests that range from very simple tests to very elaborate ones. The development of this thesis revolves around one of the most basic MPSL tests, the MPSL Init test. The MPSL Init test purpose is to check that the initialization routines enable or disable interrupts for the different peripherals.

To execute MPSL target tests in a virtual prototype, it is necessary to identify which hardware is accessed by the different tests. These hardware accesses need to be defined since they dictate the minimum hardware components that the virtual prototype needs to model. All the hardware interactions that occur during the MPSL Init test are listed and described below. The virtual prototype that models these accesses works as the base that enables the execution of more complex tests. Hardware interactions in the MPSL Init test include:

- Peripherals
  - Clock control
  - Real-time counter
  - Timer/counter
  - General purpose input/output (GPIO)
  - 2.4 GHz radio
  - Temperature sensor
- CPU components
  - Nested vector interrupt controller (NVIC)
  - System control block (SCB)
  - Assembler function calls
- Memory accesses

Clock Control Peripheral

The CLOCK peripheral is the one in charge of sourcing system clocks from a range of internal or external oscillators. It is also the one in charge of distributing them to the modules according to their specific needs. Figure 2.3 shows the components of the CLOCK.

This peripheral holds both a high-frequency and a low-frequency controller. The high-frequency controller can use a 64 MHz internal oscillator or an external crystal oscillator to provide a 64 MHz clock for the CPU and also clocks at 1 MHz, 16 MHz, and 32 MHz for the peripherals. The internal oscillator is active when there is a clock request, and the crystal oscillator is not running. Starting a high-frequency clock generates an event. The low-frequency controller can use a 32 kHz RC oscillator, a 32 kHz crystal oscillator, or a 32 kHz clock synthesized from the high-frequency clock as sources to provide a single 32 kHz clock. Just like in the high-frequency controller, starting a low-frequency clock generates an event.

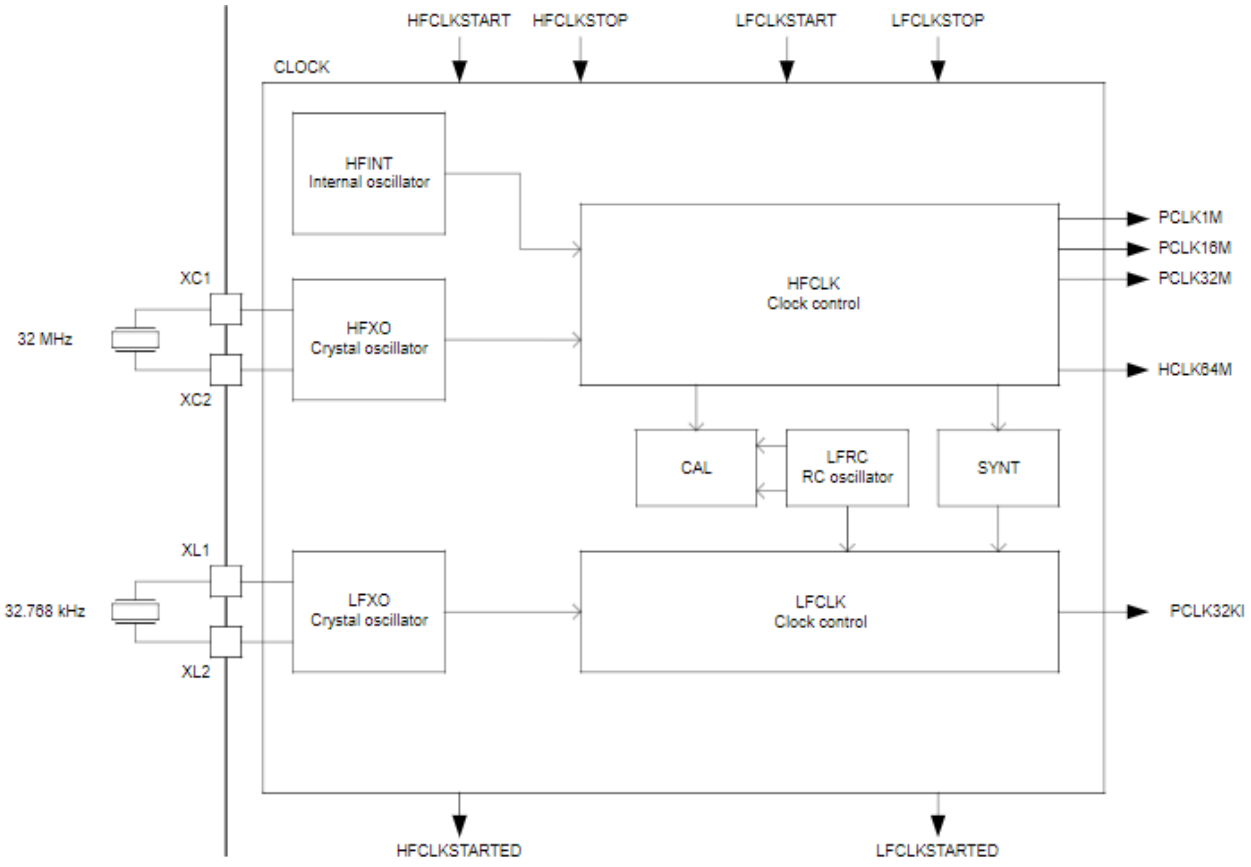


Figure 2.3: Block diagram of the CLOCK peripheral [3]

The CLOCK is the most used peripheral during the mspl_init() and mpsl_uninit() routines. The mpsl_init() function employs this peripheral for the clock initialization. During this process, the initialization routine disables interrupts and clears pending requests for the CLOCK in the interrupt controller. It also resets the events that show when the high or low-frequency clocks start, as well as events for the calibration termination and timeout. Part of the routine checks that the high-frequency clock was not triggered before the initialization and also that it was not running. The last part of the initialization is in charge of starting the low-frequency clock. On the other hand, the mpsl_uninit() routine stops the high-frequency clock, clears events, stops the calibration timer if it was running, and disables the interrupts for both the interrupt controller and the CLOCK.

Real-time Clock Peripheral

The RTC works as a simple timer that runs from the low-frequency clock source. The RTC can also use a prescaler. Figure 2.4 shows a simplified block diagram of this hardware component.

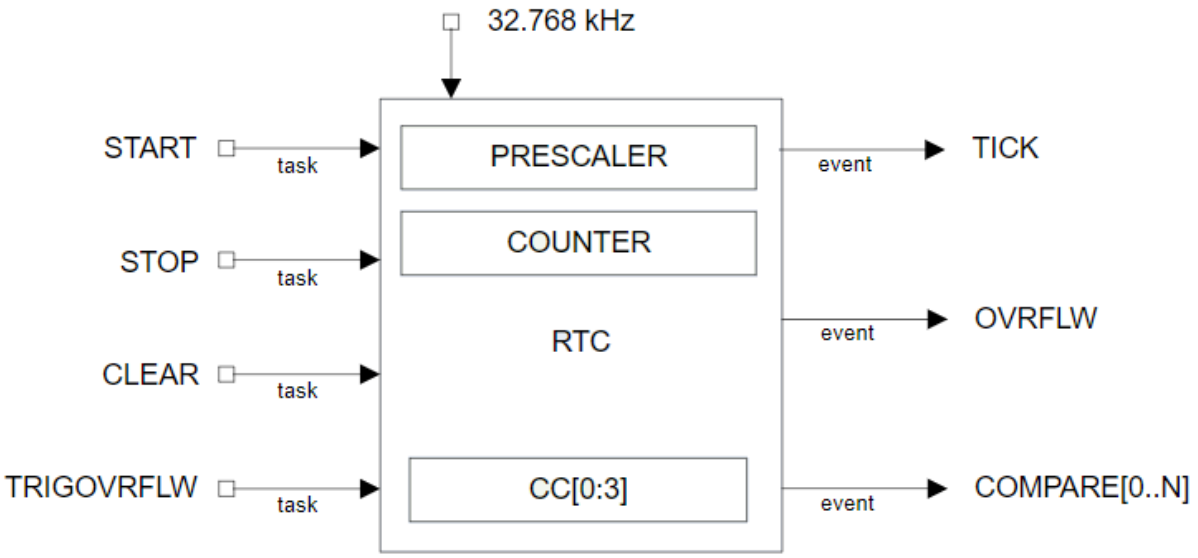


Figure 2.4: Block diagram of the RTC peripheral [3]

The RTC can be started, stopped, or cleared at any time. The trigger overflow task sets the counter to the value of 0xFFFFFF to let the software test the overflow condition. When the internal count matches the value set in one of the compare registers, a compare event triggers an interrupt if they were previously enabled. This peripheral includes an overflow event to indicate that the count reached the maximum value, and a tick event that generates a signal for each counter increment.

There are several RTC peripherals. However, the MPSL initialization functions only make use of the RTC0. During both the mpsl_init() and mpsl_uninit() functions, a routine in charge of resetting and clearing the peripheral registers and interrupts employs the RTC0. The counter is started once during the execution of the mpsl_init() function.

Timer/Counter Peripheral

There are several timers in each SoC, but just like the case of the RTC, TIMER0 belongs to the MPSL library. Figure 2.5 illustrates how the TIMER works.

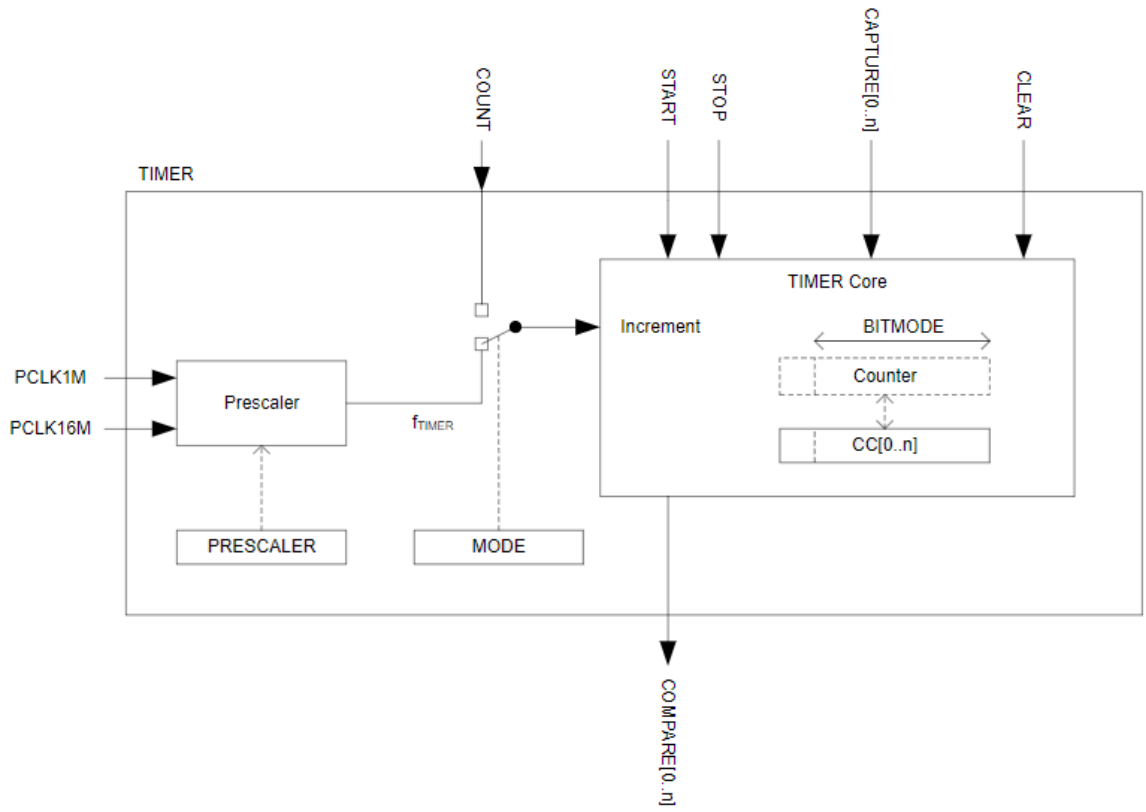


Figure 2.5: Block diagram of the TIMER peripheral [3]

The TIMER can operate either as a timer or as a counter. In both modes, the peripheral can receive the order to start or stop at any time. However, when it stops and then starts again, the count does not start from scratch, but it resumes in the value before the stop request. In the timer mode, the internal counter value increments after every tick of the timer frequency signal, which can use the prescaler from either a 16 MHz or a 1 MHz reference clock. In counter mode, the internal counter value increments every time the counting task triggers. If interrupts are enabled, a compare event triggers when the internal counter value matches the value of one of the compare registers. On the other hand, when a capture task triggers, the value of the counter is copied in one of the compare registers.

TIMER0 finds use in both mpsl_init() and mpsl_uninit() functions by routines in charge of resetting and cleaning up the timer. These routines stop the timer from running, disable interrupts, reset the events, set to zero the compare registers, and shut down the timer.

General Purpose Input/Output Peripheral

A GPIO is a digital signal pin whose behaviour is controlled by the user at run time. They have no predefined use, so the user can specify the pin's functionality depending on the application requirements. These pins are unused by default. The pins of the GPIO peripheral appear in groups defined as ports. Each port holds up to 32 GPIOs. This peripheral appears only once during the mpsl_init() routine when the RTC starts. One pin of the GPIO works as a debugging tool to inform that the RTC startup is taking place.

2.4 GHz Radio Peripheral

The RADIO is mainly composed of a 2.4 GHz radio receiver and a 2.4 GHz radio transmitter. It is a complex peripheral which also integrates modules to simplify direct memory acecss (DMA), automate packet assembly and disassembly, and also to automate cyclic redundancy check (CRC) generation and checks.

This peripheral barely finds use during the MPSL library's initialization routines. Only one function in charge of resetting the RADIO employs this peripheral. This routine is present in both the mpsl_init() and mpsl_uninit() functions, and the only thing it does is to turn the peripheral off and then on, to cause a full reset.

Temperature Sensor Peripheral

The TEMP measures the die temperature and has a resolution of 0.25 degrees. After the software triggers a temperature measurement, an event notifies that the measurement is complete, and that means the result is ready to be read from the temperature register.

It is only used during the mspl_init() function to return a temperature value during the clock initialization. In this specific routine, interrupts are disabled, and the temperature data is obtained by polling a status register.

Nested Vector Interrupt Controller

The NVIC is not a peripheral but a part of the ARM CPU. Some of its functionalities need to be included in the CPU model since the abstract CPU in the virtual prototype only performs function calls instead of modelling a real ARM processor. The NVIC provides configurable interrupt handling abilities to the processor. It acts as an intermediary between the peripherals and the CPU. During the execution of the initialization routines and the test body of the MPSL Init test, the NVIC enables, disables, and checks for interrupts. It also clears pending interrupts in the CPU.

System Control Block

The SCB is also part of the CPU, and it provides system implementation information and system control. During the execution of the mpsl_init() and mpsl_uninit() functions, the SCB accesses only one register. A bit can be set to either cause all interrupts and events to wake the processor, or only to restrict this action to the enabled interrupt and events.

Assembler Function Calls

Some parts of the MPSL library include direct references to assembler functions for the CPU core. Since they cannot run as they are on the host computer, they also need to be considered inside the virtual prototype. During the MPSL initialization, assembler calls handle system interrupts, perform wait-for-event instructions, and no-operation instructions.

Memory Accesses

Certain parts of the code present in the mpsl_init() and mpsl_uninit() routines attempt to access values by dereferencing a pointer from a fixed address. These pieces of code are not able to execute in the host PC as they are. The virtual prototype needs to somehow take care of these accesses, given the absence of a device from Nordic Semiconductor. Luckily, normal accesses to specific peripherals can replace them, or in some cases, they can be ignored by the implementation since they are not relevant to run the two initialization routines of interest.

2.3 Virtual Prototyping

There are many definitions for virtual prototype both in literature and in industry. They usually differ in aspects like the difference between a virtual prototype and a digital mock-up, the functions of a virtual prototype, the fact that a virtual prototype should include human-product interaction or not, among others. Considering all these factors, [4] provides the following definition: *"Virtual prototype, or digital mock-up, is a computer simulation of a physical product that can be presented, analyzed, and tested from concerned product life-cycle aspects such as design/engineering, manufacturing, service, and recycling as if on a real physical model. The construction and testing of a virtual prototype is called virtual prototyping (VP)"*.

In terms of hardware design, a virtual prototype is a high-speed functional software model of physical hardware. This system model often describes the system at a higher abstraction level while keeping the same functionality as the hardware. The most significant benefit of virtual prototypes is that they simplify the hardware design space exploration and allow concurrent hardware and software development. All this ends up saving time to market and resources, finding and fixing bugs earlier, allowing good test coverage, and even improving teamwork between hardware engineers, software developers, and testers [1]. There are different opinions about the use of hardware, virtual or hybrid prototypes in the SoC design flow. However, most of them agree that given the complexity of modern designs and market demands, the integration of virtual prototypes into the design flow is a growing need [5].

2.3.1 Virtual Prototypes for Embedded Software Development

Virtual prototypes have a broad range of applications [6]. They can be used to develop high-level models that can be synthesized directly into hardware when written using specific guidelines. These models could also find use in verification activities. A virtual prototype can be used as a golden reference to verify an equivalent piece of RTL. A test can apply identical stimuli to both the prototype and the RTL, to later compare their response. System architects could also use prototypes to perform design space exploration. Take the example of an engineer who is trying to evaluate different RAM technologies for a particular design. The engineer could have a virtual prototype of a complete system in which he or she can add and remove different RAM models, measure their performance, and make a decision based on the results. All this without the need of having any physical hardware component. In general, the use of virtual prototypes can reduce development time, get products to the market faster, and deliver better products. This can result in great economic benefits as shown in an example case study depicted in Figure 2.6 [1].

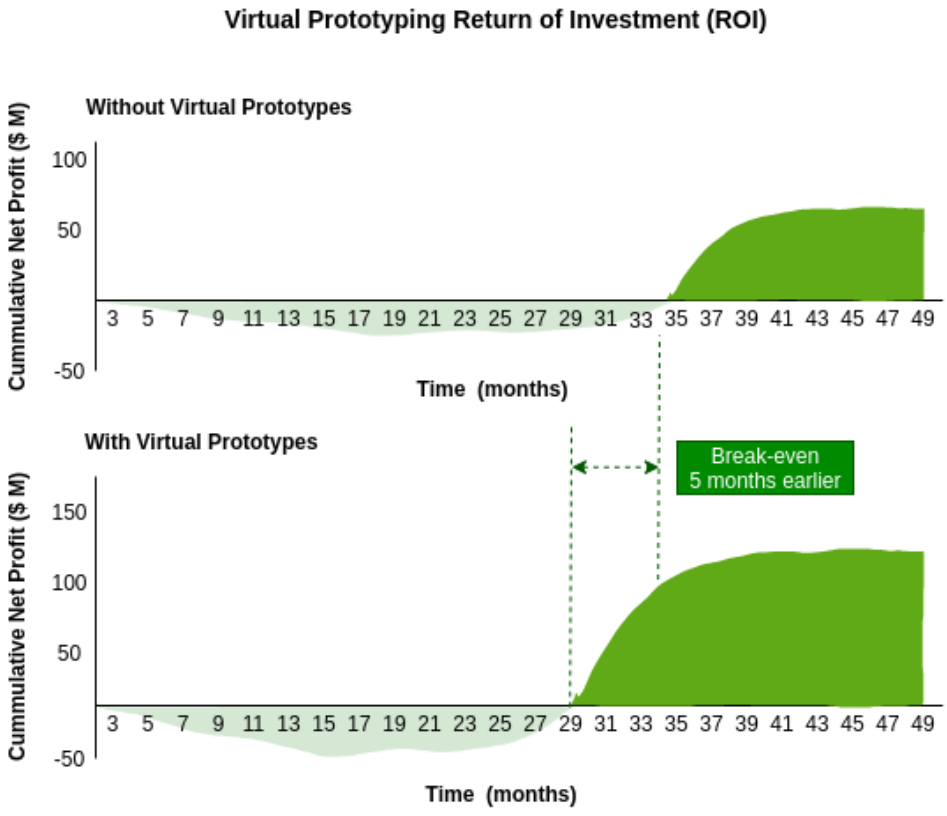


Figure 2.6: Net profit from a real case study of the use of virtual prototypes

One of the most important applications for these models is early software development. The amount of software that engineers can develop and test before a device is out of the factory is usually limited. Engineers may develop some software components in advance, but the fact that specific information on the physical chip is not yet available can become a limitation. Testing can also start only after the device has left the factory. Software development and testing is not an easy task. Therefore once the device is ready, it takes several weeks or months to deliver a functional version of a software stack. If software developers could have a model of the SoC that is accurate enough to allow them to run the software just as it would on the physical device, the development and testing could be done much earlier in the design flow.

A virtual prototype could precisely serve this purpose. A model of the SoC does not need to be cycle-accurate to be able to run the software. Many details of the internals of the design can be abstracted to produce a functional model that fulfils the purpose of executing the software. There are many ways to develop these models. However, SystemC is the IEEE standard and the most popular approach. The SystemC standard also includes information regarding guidelines on how to develop models for this specific purpose.

2.4 SystemC Library Core

2.4.1 Origins of SystemC

In the past, there have been efforts to find an electronic system-level (ESL) design language that can be suited for both hardware and software high-level descriptions [7]. Attempts to develop an entirely new language like Rosetta [8] or SuperLog [9] have been made. Other attempts have tried to extend VHDL or Verilog without big success. Considering that the biggest component of modern systems is software, C/C++ based approaches have proven to be the most promising ones. It is easier to adapt C to make it work for hardware designs, and in most cases, hardware designers already know how to use the language.

SystemC was born as an effort to find a standard ESL design language. It was developed in 1999 by the Open SystemC Initiative (OSCI), and it became an IEEE standard [10] in 2006. It is a system design and modelling language based on C++. In the strictest sense, SystemC is

not a language itself but a C++ class library. SystemC finds use in industry for the development of hardware blocks at various levels of abstraction, architectural exploration, development of virtual prototypes for hardware/software co-design, verification, and even high-level synthesis.

2.4.2 Virtual Prototyping with SystemC

As mentioned, SystemC is no more than a C++ class library. SystemC focuses on modelling both hardware and software using C++. Since C++ already covers most of the software issues, SystemC focus is more on the side of the non-software issues. As depicted in Figure 2.7, the SystemC library sits on top of C++. It consists of a set of structural elements, predefined channels, data types, utilities, and an event-driven simulation kernel. One important component of the SystemC library is TLM, which stands for Transaction-Level Modeling. TLM and its components are the ones that make ESL design practical by focusing on communication-centric high-level modelling. On top of the SystemC core, there are special libraries like the Analog/Mixed-Signal (AMS), SystemC Verification (SCV), and the Configuration, Control and Inspection (CCI) library. For detailed information about SystemC refer to [11].

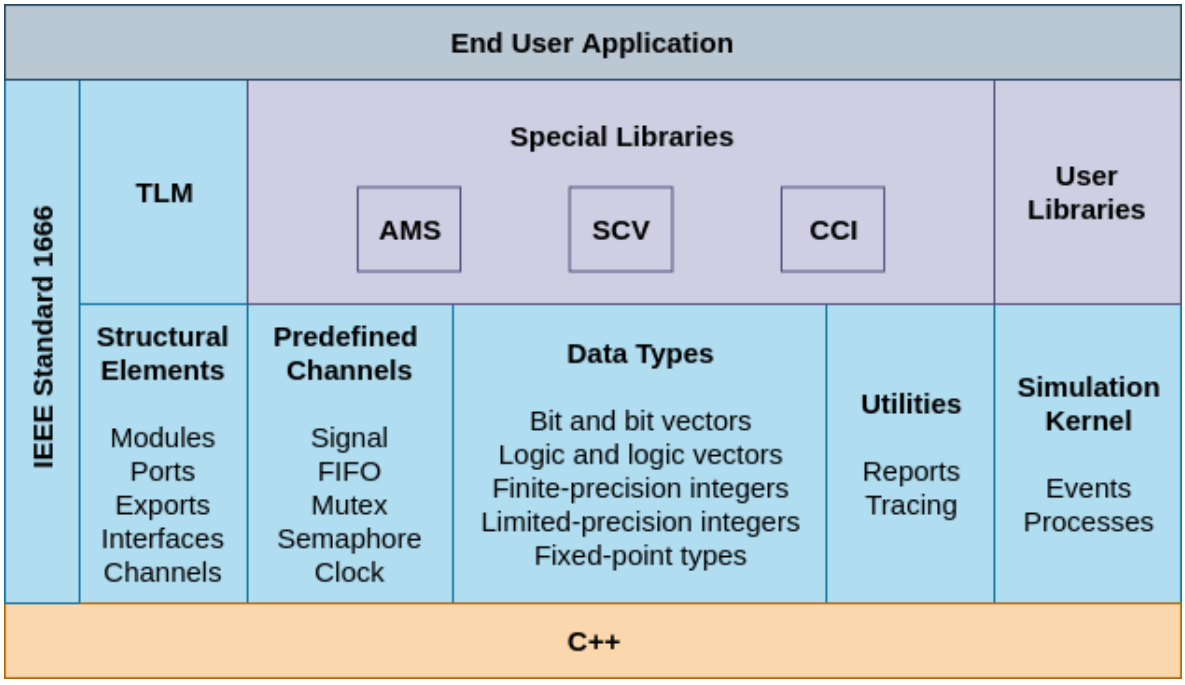


Figure 2.7: Architecture of the SystemC library

2.4.3 Overview of the SystemC Components

Structural Elements

In SystemC, hardware or software design components are encapsulated as modules. A module is no more than a C++ class definition. Therefore, it is possible to use a regular class or the sc_module macro to declare one. Every module must, at least, include a constructor that is in charge of performing SystemC specific setup tasks. Listing 2.2 shows a module that describes an OR gate. First, the model includes the systemc.h library and uses the sc_module macro for the module declaration. Sc_in and sc_out declare input and output ports of type sc_bit, which is a SystemC specific data type. A process describes the model behaviour, which in this case, is a simple OR operation between both inputs. Finally, the sc_ctor macro is in charge of the constructor creation. This constructor registers the process in the SystemC scheduler using the sc_method macro and, in this case, defines a sensitivity list for the SystemC process.

```
#include "systemc.h"

SC_MODULE(or_gate) {
    sc_in<sc_bit> a;
    sc_in<sc_bit> b;
    sc_out<sc_bit> c;

    void prc_or_gate() {
        c= a | b;
    }

    SC_CTOR(or_gate) {
        SC_METHOD(prc_or_gate);
        sensitive << a << b;
    }
};
```

Listing 2.2: SystemC description of a 2-input OR gate

Ports, exports, interfaces, and channels are ways to achieve communication in SystemC. Ports are bound to communication channels, and they can interface modules. In VHDL and Verilog, signals are the ones in charge of communication. Nevertheless, in SystemC, a signal is just a particular case of a channel. Channels are, in general, containers of communication protocols and synchronization events. They sit on top of interfaces that define a set of pure virtual methods that channels must implement. This feature enables to implement modules that are independent of the implementation details of the communication channel. For example, two different channels can describe a fast and a slow memory protocol using the same interface. As long as the system's modules communicate using the same interface, the implementation details of these protocols are not relevant, and they can easily switch. Exports are specific types of ports that work as a pointer to a channel inside another module.

Predefined Channels

SystemC also includes several predefined channels known as primitive channels. Among others, these channels include sc_signal, sc_fifo, sc_mutex, and sc_semaphore. The predefined channel sc_signal provides simple write and read methods that generate events only when the signal changes its value. A channel that behaves like a FIFO can use sc_fifo which provides methods for blocking and non-blocking read and write operations. It also has methods to return the number of busy and free elements. Access to shared resources is possible using primitive channels like sc_mutex or sc_semaphore. The fist provides methods to lock or unlock a resource while the second provides methods to wait, signal, or get the semaphore value.

Data Types

C++ does not provide a wide variety of data types that digital hardware requires. SystemC can provide data types that support arbitrary bit widths, digital representations such as tri-state or unknown, and special types for integers, floating-point numbers, literals, and strings. SystemC also provides methods to operate over these data types such as bit selection, range selection, conversions, testing, and bit reduction.

Utilities

Reports and error handling are essential components of simulations. SystemC provides an error report system that simplifies this task. Messages have different classifications, which include sc_info, sc_warning, sc_error, and sc_fatal. Also, every type of message can trigger actions like writing information into a log file, displaying the message on the screen, and stopping the simulation. The SystemC library even allows performing signal tracing for debugging purposes. There is no built-in graphic viewer in SystemC. However, it is possible to copy values into a VCD file that is compatible with most waveform viewers.

Event-driven Simulation Kernel

Concurrency is a fundamental part of SystemC simulations. SystemC uses an event-driven simulation kernel to model this concurrency. This kernel uses non-preemptive simulation processes, which can be defined either as methods, as shown in line 13 of Listing 2.2 or as threads using the sc_thread macro. Figure 2.8 shows a simplified model of the simulation kernel.

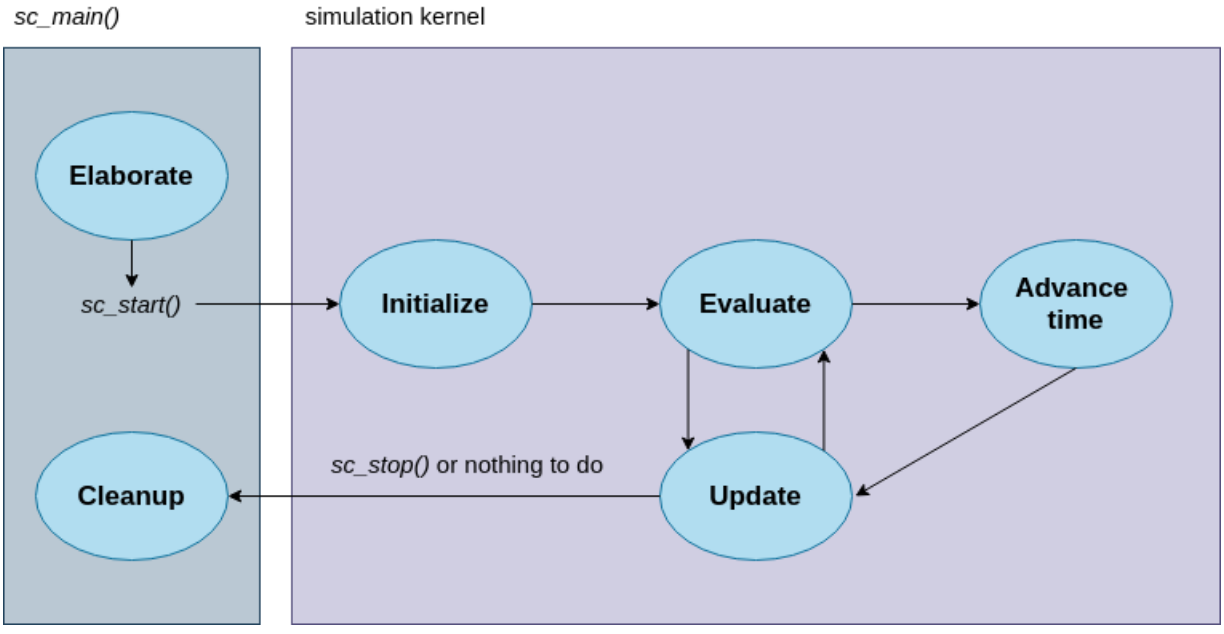


Figure 2.8: Simplified SystemC simulation kernel

A SystemC simulation can be divided into six stages:

1. **Elaborate:** components are instantiated and connected to create a model ready for simulation. Process registration happens here. A call to sc_start invokes the simulation kernel and moves to the next stage.

2. **Initialize:** processes are identified and placed in a runnable or waiting state.

3. **Evaluate:** all the processes marked as runnable execute successively and in an undefined order until they hit a wait statement or return. During execution, processes can perform update requests, which are noted by the scheduler. A wait statement tells the scheduler that a process should run later.

4. **Update:** here the update requests are performed. The scheduler marks a process as runnable if it is sensitive to the performed updates and goes again into the evaluation phase. When there are no runnable processes, the simulation advances in time.

5. **Advance time:** simulation time moves forward and processes marked as runnable execute in the evaluate stage. If there are no more processes to run, the scheduler moves into the cleanup stage. The cleanup stage is also scheduled next if a process has stopped simulation calling sc_stop or a timeout occurs.

6. **Cleanup:** simulation is over, and necessary cleanup routines are performed.

2.5 System C and TLM

2.5.1 TLM Concept

As mentioned earlier in this chapter, TLM and its components make ESL design practical. TLM is a transaction-based modeling approach in which communication details are independent of the implementation details. It is more important what data moves to and from which locations than the protocol used for the data transfer. Cycle accurate simulations, like RTL simulations, consider all the events on all pins. Therefore, they are usually very slow. TLM makes use of transactions, which represent data transfers, to simulate only the relevant events. This concept

makes TLM simulations considerably faster than cycle-accurate simulations. TLM use cases cover software development, software performance analysis, architectural analysis, and hardware verification.

2.5.2 Overview of TLM 2.0

TLM 2.0 [12] is the transaction-level modeling standard for SystemC. Its main focus is the modeling of systems based on memory-mapped busses. It provides a standard API for both blocking and non-blocking transports as well as a standard payload object. TLM models are written using initiators, interconnects, and targets. These modules act over a generic payload object, which includes typical attributes of memory-mapped buses. This payload object also enables the use of extensions to implement more complex communication schemes. The initiator is in charge of creating new transactions, the target executes requests and sends back responses, and the interconnect routes transaction objects between initiators and targets. Communication employs function calls that pass a reference to a single transaction object along the backward and forward paths, as shown in Figure 2.9. Targets must implement the methods defined by the forward path interface, while initiators must implement the backward path interface methods.

For example, to model a CPU read request from memory, a transaction can involve a CPU acting as a initiator, a bus as the interconnect and a memory device as a target. The CPU will set the payload object by specifying parameters like the action (read), and the memory location that wants to be accessed. It can use the forward path to give control of the transaction to the interconnect which modifies the address of the payload object to route the read request to the correct memory block. The target will read the data and place it in the data field of the payload object which can be now accessed by the CPU.

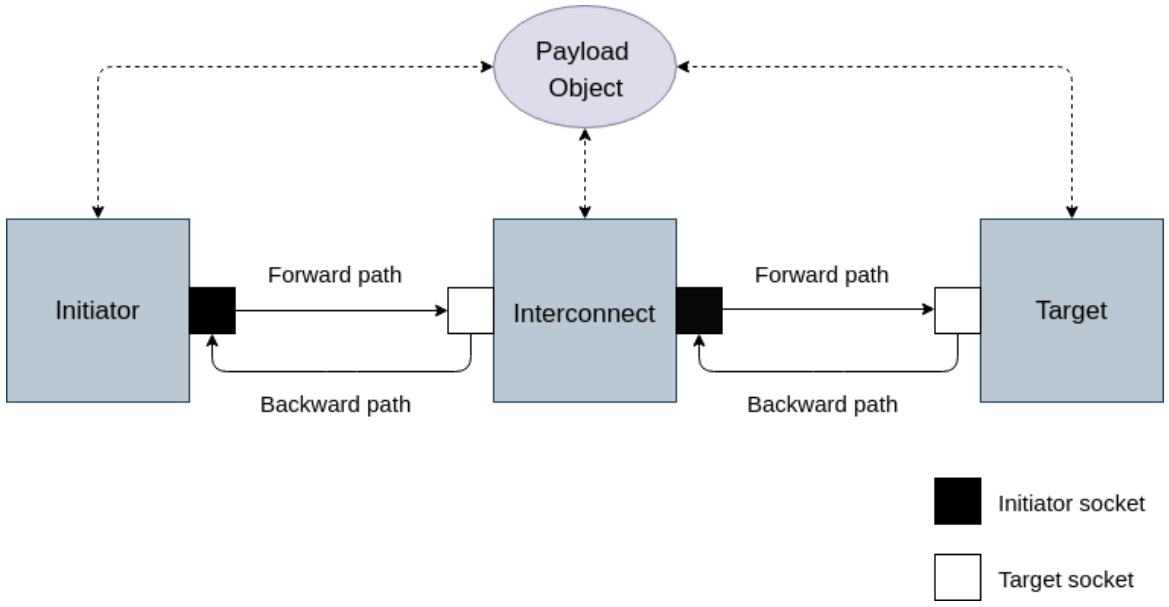


Figure 2.9: TLM components

2.5.3 TLM 2.0 Coding Styles

Loosely-Timed TLM

Loosely-Timed (LT) TLM uses a single-phase blocking API and finds applications in software development and software performance analysis. This blocking API associates only two points in time with a transaction; these are the blocking transport function call and return. This mechanism only uses the forward path from the initiator to the target. The b_transport method takes a reference to a generic payload object and a timing annotation as arguments. The idea of the timing annotation is to model the behavior of a receiver getting the information with a specific delay. A diagram explaining a blocking transport message sequence is shown in Figure 2.10. TLM LT also supports temporal decoupling. Individual processes that use temporal decoupling can run ahead with no progress in the simulation time until they execute for a specific time slice known as quantum or until they reach a synchronization point. This technique reduces the context switches between processes, accelerating the simulation speed significantly. The quantum size represents a trade-off between accuracy and simulation time.

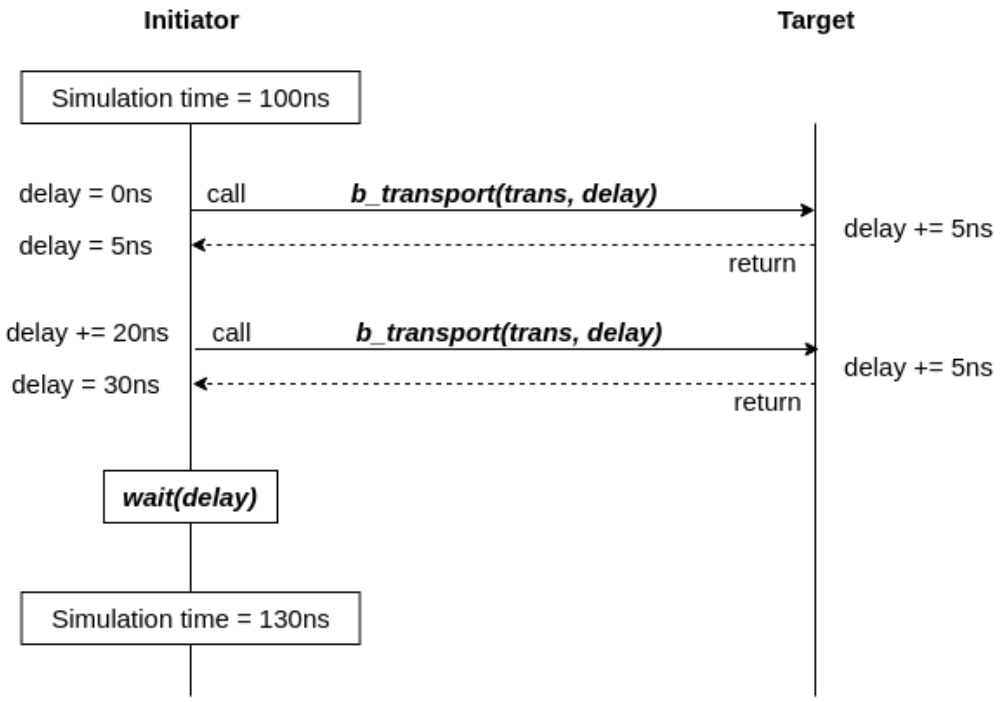


Figure 2.10: Example of a message sequence in TLM LT

Approximately Timed TLM

Approximately-Timed (AT) TLM uses a multi-phase non-blocking API and is mostly used for architectural analysis and hardware verification. For the base protocol case, a transaction involves exactly four timing points. These points represent the beginning and end of a request or a response. Since TLM AT needs accurate timing, it runs in lockstep with the simulation time, and it cannot make use of temporal decoupling. In this coding style, every interaction has a delay, representing data transfer times and the latency of the target. The characteristics of the non-blocking API facilitate the modeling of pipelined transactions. The non-blocking transport uses both the forward and the backward communication paths; each of these paths has its own interface. In addition to the payload object reference and timing annotation, a phase indicating the state of the transaction must be passed as a parameter. Calling a non-blocking function will return a status value representing the state of the transaction. Figure 2.11 shows a diagram that illustrates the non-blocking base protocol. In addition to this four-phase handshake protocol, the base protocol can terminate transactions sooner and also offers the possibility of implementing specific communication protocols by adding further timing points.

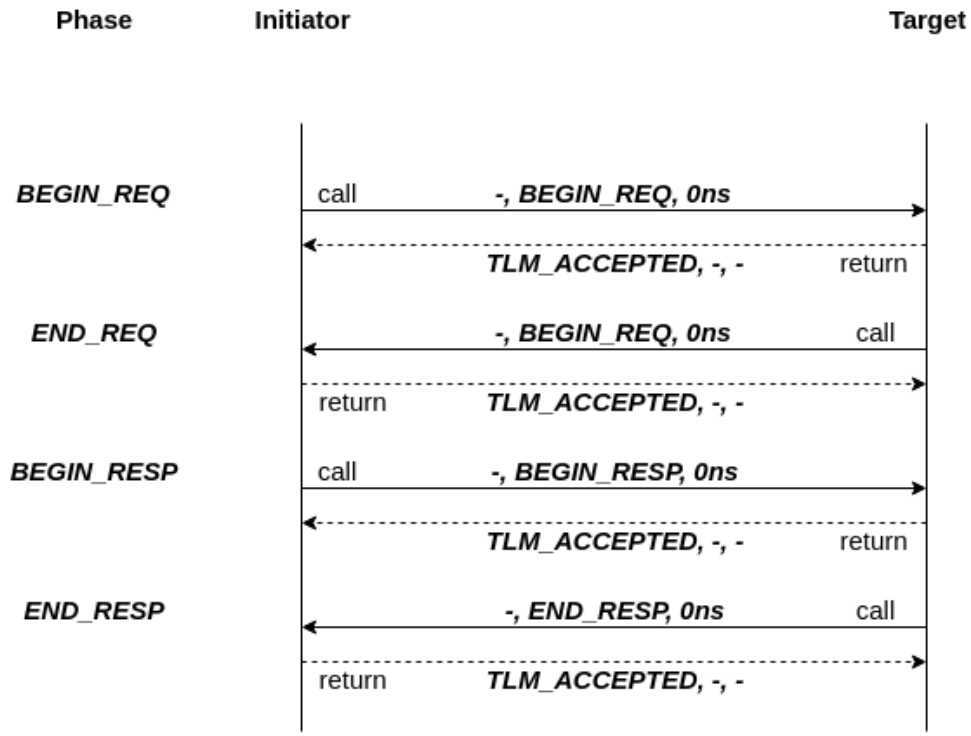


Figure 2.11: Example of a message sequence in TLM AT

2.5.4 Direct Memory Interface and Debug Transport Interface

Besides the blocking and non-blocking transport APIs, the forward and the backward interfaces provide methods to grant direct access and debug access to a memory block. Direct Memory Interface (DMI) is widely used in TLM AT, and it provides a considerable speedup in simulation time. When DMI is active, an initiator can bypass the interconnect and access the memory directly. It uses both a forward and a backward path interface. The debug transport interface is, as the name implies, for debug proposes. It provides the possibility of accessing memory like a regular transaction but free of delays and the overhead that regular transactions posses. This interface only uses the forward interface.

2.6 Bauhaus: Nordic Semiconductor's Generic Virtual Prototype

Bauhaus is a proof of concept and experimentation testbench for modelling Nordic Semiconductor's devices with SystemC. It serves as a framework under which more complex systems can be developed. These are some of its most important features:

- **CPU:** many virtual prototypes use an instruction set simulator (ISS) and a specific processor model to execute each instruction just like a physical processor does. However, the CPU model in Bauhaus does not intend to work this way to keep the system simple and generic. Instead, the CPU model implements only functions to write and read from registers in specific memory addresses. The CPU module points to a software class that holds the software to be executed in the CPU. This software will then execute as any other software in the host PC but will need to use the read or write functions when interacting with the virtual prototype's elements.

- **Interconnect:** the Bauhaus model also comes with an interconnect element. This module is in charge of receiving the transactions created in the CPU and routing them to the correct address so that the CPU can interact with other peripherals.

- **Dummy IP:** the virtual prototype lacks any peripheral implementation. However, it has a dummy module that can serve as a base for other peripherals. This module has no particular functionality, and it only logs an event every time the CPU writes to it.

- **System Builder:** one of the most important features of Bauhaus is the system build automation. It possesses functions to add a CPU element, an interconnect, and peripherals, respectively. It also has a system build function that binds the initiator ports of the CPU to the interconnect's target ports, as well as the initiator ports of the interconnect to each peripheral's target ports. This functionality allows having a generic interconnect which can hold an arbitrary number of peripherals and also automates and simplifies the binding process in the top-level module of the system.

- **Other features:** the Bauhaus virtual prototype works on top of Nordic Semiconductor's library for SystemC development. The library offers a power control module that can switch the system to different power modes to which each module can react. It also offers a clock control module that can offer different clock frequencies to the system and can provide reset events. Bauhaus also offers a specific header file to set technology-specific parameters, including timing and power. Some files can be modified to set different timing and power parameters like clock default period, startup time, power-off time, power modes, power transition times, among others.

A diagram of Bauhaus with its main components is depicted in Figure 2.12.

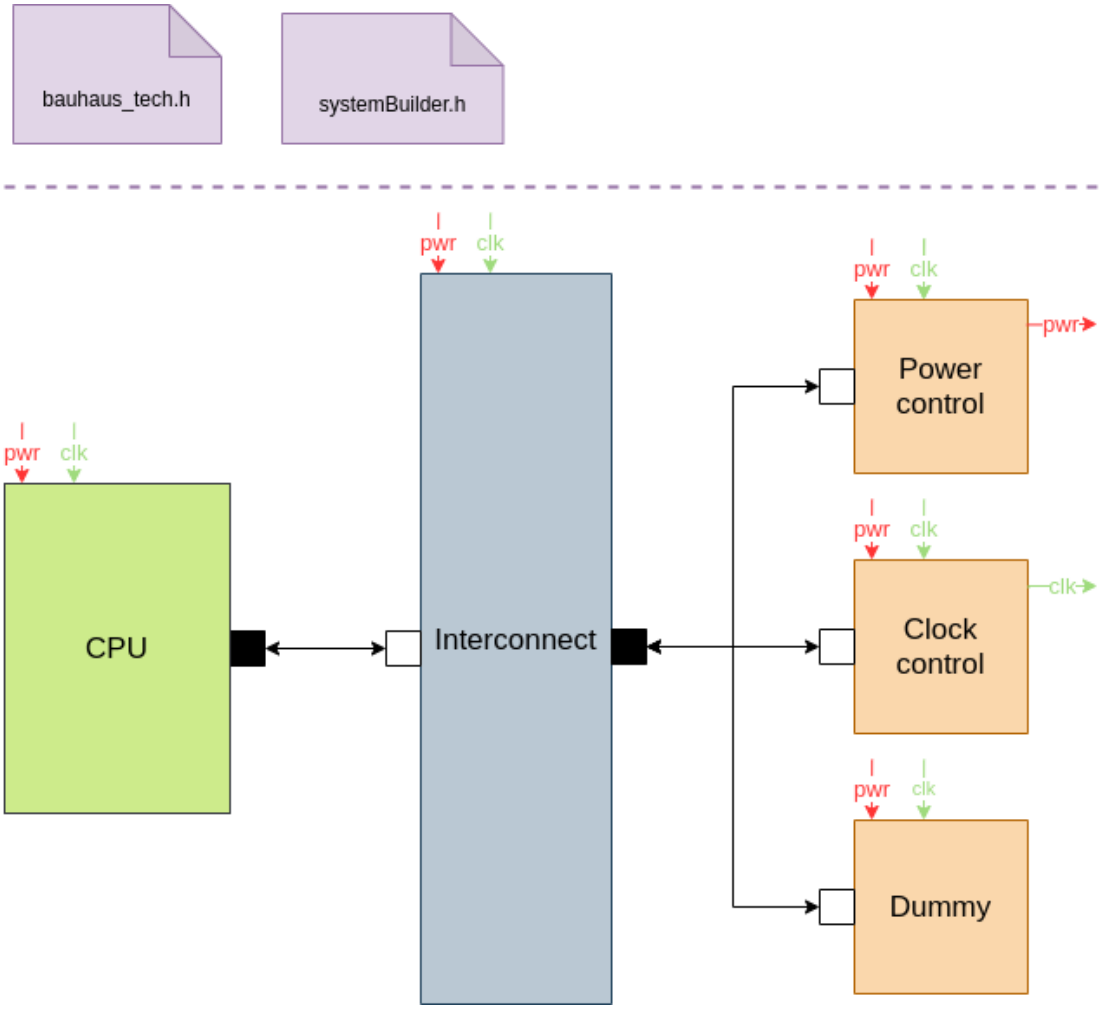


Figure 2.12: Components of the Bauhaus generic virtual prototype

Chapter 3

Methods

The goal of this thesis is to develop a virtual prototype that can run a subset of the MPSL target tests and explore the potential benefits of this approach. In order to achieve that goal, the process depicted in Figure 3.1 was followed.

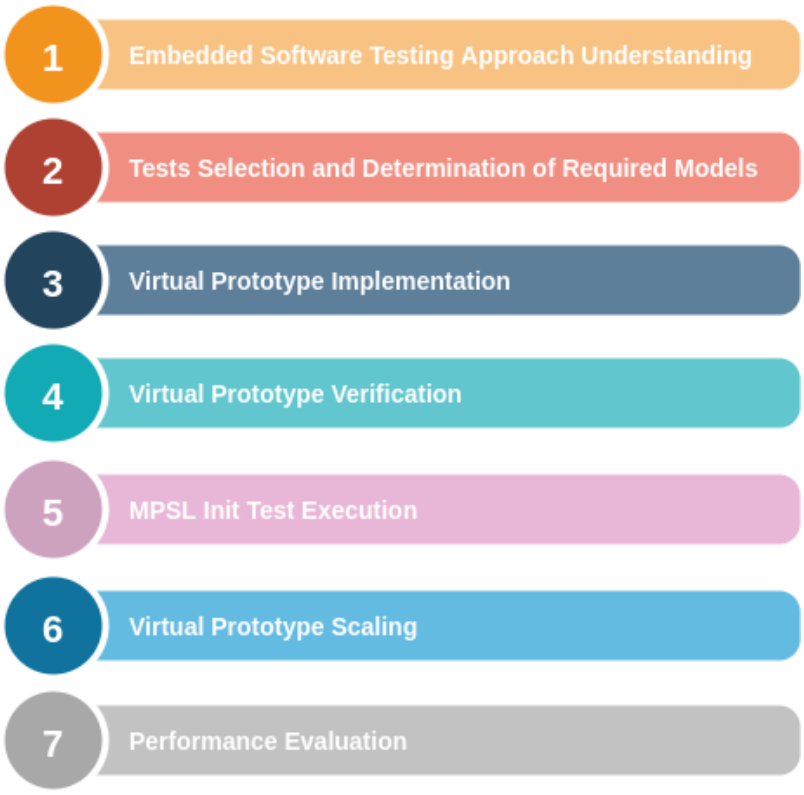


Figure 3.1: Thesis development approach

The first step of this project involved getting familiar with the embedded software testing approach. Understanding MPSL tests was necessary to provide a baseline to evaluate the final implementation. The following step involved selecting a simple test to focus on since targeting to cover all tests is out of the scope of this thesis. Selecting one simple test also helped determine precisely which hardware components needed to be modelled by inspecting the code and extracting the hardware that is accessed by this specific test. After this was decided, the models were developed in order to build a functional virtual prototype based on the test requirements. Once the virtual prototype was complete, each of its components, as well as the system as a whole, had to be verified in order to make sure its behaviour matched the specifications. When the system was fully verified, it was possible to interface the virtual prototype with the MPSL software to execute the MPSL Init test. Once this test could execute successfully on top of the virtual prototype, the system was scaled to execute two more tests and demonstrate how it is possible to expand the virtual prototype to cover more tests. As the final step, the performance of the virtual prototyping approach was measured and compared against the baseline of the current approach using development boards.

3.1 Embedded Software Testing Approach Understanding

In order to get a clear idea of how the virtual prototype would fit the current testing framework, it was necessary to study the relationship between the software stack source files, the tests, and the parts of the code dedicated to interact with the hardware. After examining the code of several tests, it was noticed that the software under test usually does not access the hardware directly. However, instead, it uses dedicated files that are in charge of most of the hardware interactions. This set of files is known as the hardware access layer. This layer holds some logic but is mostly surrounded by register accesses to peripherals, which are the primary way software can interact with the hardware elements in the SoC. Other hardware interactions include the manipulation of CPU registers, interrupt controllers, assembly language instructions, and even memory accesses to fixed addresses. Identifying these hardware interactions was essential to define what needed to be modelled.

It was important not only to understand the purpose and dynamics of the target tests but also to execute them just like the software team does. This execution would allow defining a baseline for future comparison between the implementation using a virtual prototype and the current approach. Running the tests on the development boards is not a plug and play process. Since there are hundreds of tests, the process to build, compile, and flash the software stack is quite complex. For this reason, there is a complex framework in charge of automating all this process. In order to run target tests on the development boards, it was necessary to set up a virtual environment running Linux. Inside this environment, it was required to run a docker container holding the packages and dependencies for the tests. It was also necessary to share the USB control between the host, virtual machine, and docker container. Once this was all set, it was possible to run target tests on physical boards and use these executions as a reference for the upcoming implementation.

3.2 Tests Selection and Determination of Required Models

Given the fact that there is a significant number of tests and also because different tests access different hardware components, it was unrealistic for the scope of the thesis to implement a virtual prototype capable of running all the tests. Therefore, it was necessary to select a test or a subset of tests that are feasible to run on a virtual prototype as a proof of concept. One of the goals of this proof of concept is to provide a prototype that can easily be scaled in case there are promising results. It was noticed by inspecting the tests, that all of them share a similar structure in which they use the same initialization routines. Between these routines, the tests usually exercise some specific behaviour like returning a temperature value, triggering a couple of timers, among others. The MPSL Init test was selected as the test to be run on the virtual prototype. It employs the initialization functions, and the main test consists of checking flags in the interrupt controller. The execution of a single test may not seem significant, considering there is a large number of tests in the MSPL library. However, if this test runs successfully, it is possible to extend this idea to all the other tests since they share most of the code in these initialization routines. Other tests would require small tweaks and system additions to run.

Once the MPSL Init test was selected as a candidate to run on top of the virtual prototype, it was necessary to define which hardware components are accessed by this test. A single test usually deals only with some parts of the SoC. Clearly defining these interactions enables focusing the development for only these components. This avoids spending time on hardware models that are not relevant for the moment. When determining the hardware to model, it was necessary to go through each function called during the test. Each line of code was inspected in order to find any register access to a peripheral, CPU, references to the interrupt controller, assembly instructions, or memory accesses. Once this was performed, it was possible to list all the hardware references along with the specific functionalities exercised in each component. There are hardware components that only require access to a couple of registers. Therefore their model should be relatively simple since all the behaviour related to the rest of the registers can be ignored. On the other hand, others access many different registers, requiring a more complex model.

3.3 Virtual Prototype Implementation

Having decided which hardware components needed to be modelled and how complex each model should be, it was possible to start with the models' development. Before doing so, it was necessary to get familiar with SystemC and how hardware models can be written using this C++ library. After covering the basics of SystemC and experimenting with simple models, the experimental virtual prototype from Nordic Semiconductor, Bauhaus, was chosen as a base for developing the models and final prototype. Since two simultaneous projects were working on writing the system models, a peripheral template was developed to establish a standard modelling style for all the peripherals. Once the template was ready, each of the peripheral models was developed on top of it, CPU register accesses and interrupt handling were covered, as well as assembly calls and memory accesses. When all the components were separately modelled, they were put together into a complete system that constitutes the virtual prototype over which the target tests can execute. This thesis focused on modelling the RADIO, TEMP, RTC, and TIMER peripherals as well as the SCB in the CPU, the assembly function calls, and the direct memory accesses.

3.3.1 Bauhaus as a Base for the Virtual Prototype

Before developing the hardware components models, it was discussed whether to start the virtual prototype from scratch or use an existing virtual prototype as a base. Since Nordic Semiconductor already made efforts into developing a generic virtual prototype for SystemC experimentation, this was an excellent place to start looking. After studying the prototype and running simple tests, it was determined that Bauhaus provided a solid base to build the desired system. It already contemplates a simple CPU model that can be scaled up to the needs of the project, an automated interconnect element and a lack of peripherals. This last characteristic is not a problem since one of the main activities of this thesis is peripherals model development. These features make Bauhaus a perfect base for the virtual prototype. The use of Bauhaus represented considerable time savings by avoiding activities related to the virtual prototype framework, which, in the end, are necessary for the system to work correctly but not relevant for this thesis.

3.3.2 Peripheral Template

A peripheral template was created to standardize the development of the peripheral models. A simplified version of the peripheral template header file is shown in Listing 3.1. The main objective of this template is to provide all the common infrastructure around the model of a peripheral so that it is necessary only to include peripheral specific information like registers and functions that implement the peripheral behaviour. Another motivation for this was to standardize the development of the models for the two projects involved in this work. The use of this peripheral template reduced the development time of the peripheral models significantly. It also made it easier to integrate and understand models written by others.

```
#include "systemc.h"
#include "tlm.h"
#include "n_ip_apb.h"
// Define here the register names
#define DUMMY_ADDR_MASK 0xFFF
#define DR      0x000
...
struct nrf_dummy : Nip_Apb{
  public:
    nrf_dummy(sc_module_name name_, int ahb_base_addr);
    sc_signal<bool> pwr_sig;

  // Define here a struct for the registers
    struct {
    uint32_t dummy_DR;
    ...} regs;

    void  b_transport(tlm::tlm_generic_payload &payload, sc_time &delay);
    unsigned int  transport_dbg(tlm::tlm_generic_payload& gp);

    // If needed implement functions to handle power and reset
    void power_domain_handler();    void reset_handler();

    void set(...);  void clr(...);  bool isSet(...);  bool isClr(...);

  private:
  // Placeholder for peripheral functionalities
    void  dummy_thread();     void  dummy_function();

  // Implement actions when writing or reading to a register if needed
    virtual void    busWrite(uint32_t uaddr, uint32_t wdata);
    uint32_t        busRead(uint32_t uaddr);
};
```

Listing 3.1: Simplified peripheral template header file

The main features of the peripheral template are (code lines refer to Listing 3.1):

- Incorporates all the dependencies to attach any peripheral to Bauhaus (line 3).
- Provides a reset and a power domain handler to manage reset events and reactions to different power states in the system if needed (lines 11 and 22).
- Provides a section for register address definition and a predefined data structure to hold each register. The user only needs to provide register names and addresses for the desired peripheral (lines 5-6 and 14-16).
- Abstracts TLM communication details, so the only thing to worry about is the model itself. The blocking transport and debug transport, along with all the generic payload handling is already covered. To write the model, it is only necessary to associate behaviours to register accesses in the template. (lines 18-19 and 31-32).
- Provides functions to set, clear, check if set, and check if clear for specific register bits to simplify register handling (line 24).

3.3.3 2.4 GHz Radio Peripheral Model

The radio peripheral itself is substantially complex since it involves packet assembly and disassembly, data whitening and dewhitening, cyclic redundancy checks, and some other complex processes. However, during the execution of the mpsl_init() and mpsl_uinit() functions, it is barely used. Only one register out of more than 60 registers of the peripheral is accessed. Therefore the model for this peripheral is greatly simplified. The register that is accessed is the POWER register, which is in charge of turning on and off the peripheral. The peripheral model is written to log simulation information when the POWER register is accessed. When the least significant bit (LSB) of the register is set to 1, it informs that the radio peripheral is on, and when it is 0, it informs that it is off. Since no other functionality is required from the radio peripheral at this point, this implementation is enough to run the MPSL Init test.

3.3.4 Temperature Sensor Peripheral Model

Table 3.1 specifies the registers that were modelled for the TEMP peripheral along with the corresponding register description.

Table 3.1: Register accesses for the TEMP peripheral

Register	Description
TASKS_START	Start temperature measurement
TASKS_STOP	Stop temperature measurement
EVENTS_DATARDY	Measurement is complete
INTENSET	Enable interrupt
INTENCLR	Disable interrupt
TEMP	Temperature value

The peripheral model is based on two threads that run in parallel thanks to the SystemC simulation kernel. The behaviour of the temperature sensor peripheral model is shown in Figure 3.2. The first thread is in charge of performing the temperature measurement. A start event is triggered when a logic 1 is written to the LSB of the TASK_START register. The thread waits for the start measurement event to be triggered and then performs the temperature measurement, which in this case, is simulated by just waiting for some predefined amount of time. In case the measurement was stopped while it was being carried out, the thread waits for the next start event. If not, it writes the temperature value to the TEMP register, sets the EVENTS_DATARDY register, and triggers the data ready event. In the case of the model, the temperature value is just a constant since this physical quantity cannot be obtained from the host CPU.

The second thread waits for the data ready event, which is triggered by the first thread when a temperature measurement is complete. When this event is triggered, the model checks if the interrupts have been enabled or not, in case they are, it forwards an interrupt to the NVIC in the CPU. Otherwise, it just waits for the next data ready event.

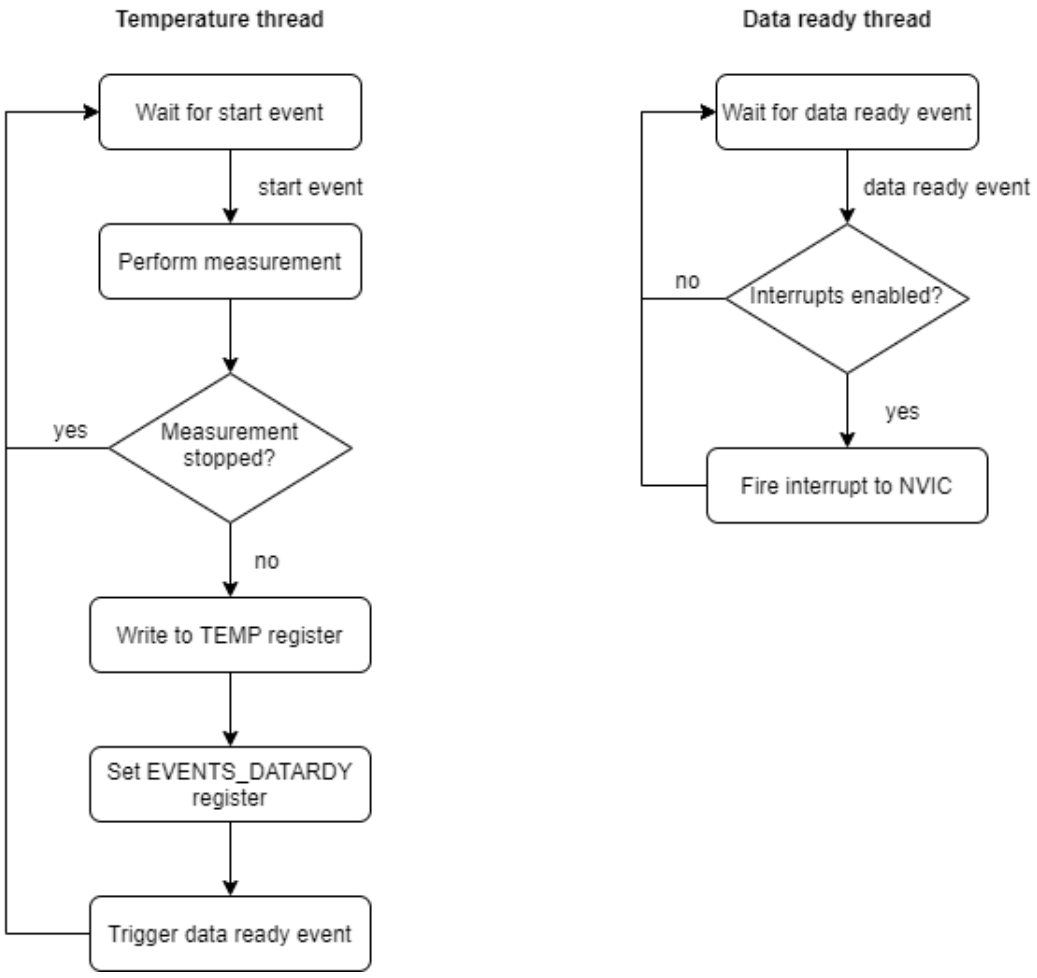


Figure 3.2: Behaviour of the TEMP peripheral model

3.3.5 Real-time Clock Peripheral Model

Table 3.1 specifies the registers that were modelled for the RTC peripheral along with the corresponding register description.

Table 3.2: Register accesses for the RTC peripheral

Register	Description
TASK_START	Start the counter
TASK_STOP	Stop the counter
TASKS_CLEAR	Clear the counter
EVENTS_COMPARE[0-2]	Compare event on CC[0-2] match
EVTENCLR	Disable event routing
INTENCLR	Disable interrupt
COUNTER	Current counter value
CC[0-2]	Capture/Compare register 0-2

The RTC peripheral model behaviour is also based on two parallel threads, as shown in Figure 3.3. The counter thread waits for a start event, triggered when the LSB of the TASK_START register has a logic 1. It waits for the clock period, which in this case, since the preescaler has a value of 0, is 30.157 us. If the counter was stopped in the meantime, it does not increase the COUNTER register. If it was not stopped, it increments the COUNTER register by one. Every time the counter is increased, the counter value is compared to each of the CC registers. In case the count equals the value in one or more of these registers, a compare event is triggered.

The other thread, the compare thread, waits for a compare event. When it is triggered, the COUNTER value is compared to each of the CC registers. When a match is found, the EVENT_COMPARE register is set. In case interrupts have been enabled for that specific CC register, an interrupt is fired to the NVIC. If the interrupts are disabled, only the EVENT_COMPARE is set, and the thread waits for the next compare event.

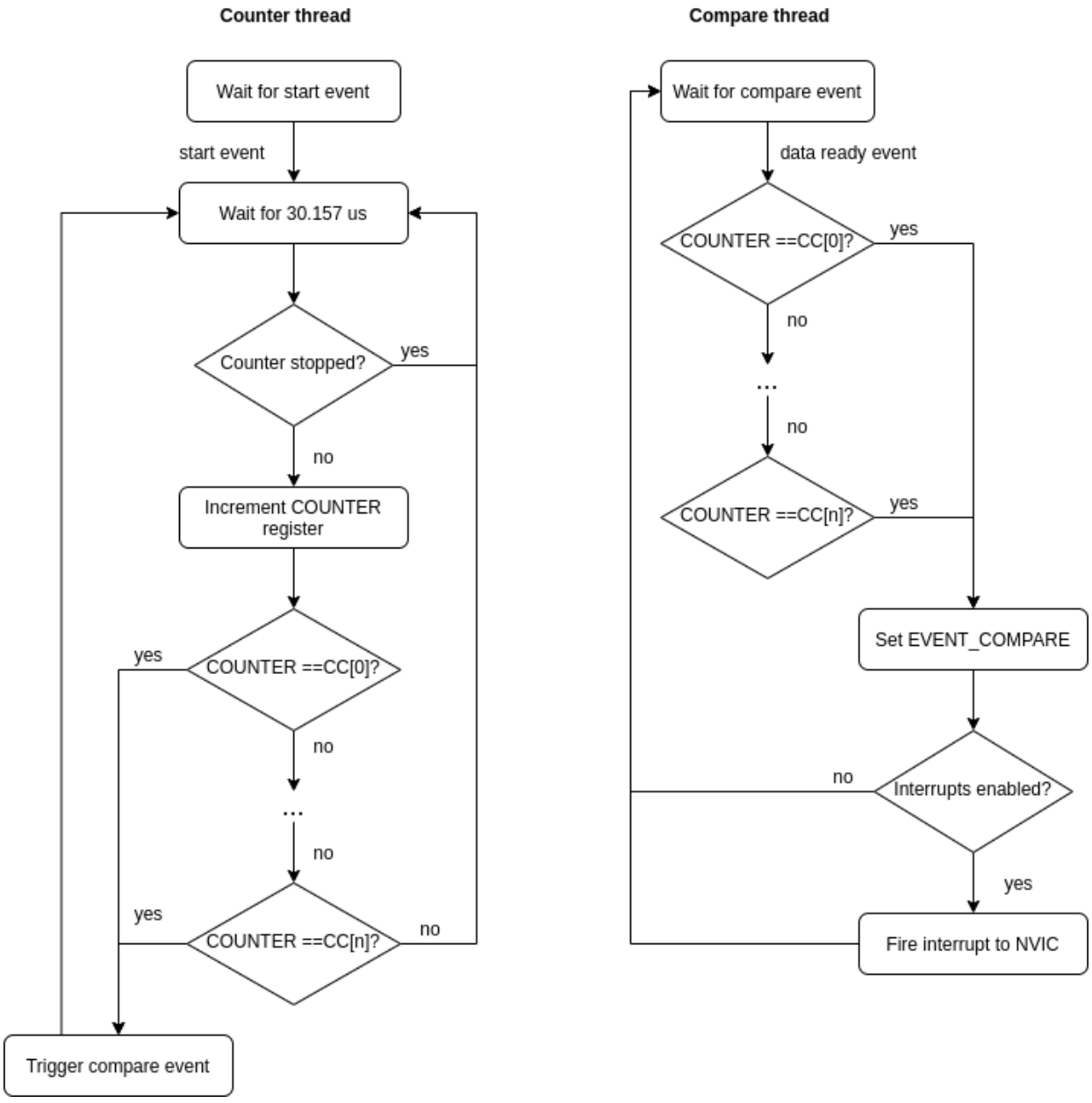


Figure 3.3: Behaviour of the RTC peripheral model

3.3.6 Timer Peripheral Model

Table 3.1 specifies the registers that were modelled for the TIMER peripheral along with the corresponding register description.

Table 3.3: Register accesses for the TIMER peripheral

Register	Description
TASK_START	Start the timer
TASK_STOP	Stop the timer
TASKS_CLEAR	Clear the timer
TASKS_SHUTDOWN	Shut down the timer
EVENTS_COMPARE[0-3]	Compare event on CC[0-3] match
INTENCLR	Disable interrupt
CC[0-3]	Capture/Compare register 0-3

The TIMER peripheral model shares most functionalities with the RTC model. For the purposes of this thesis, and in order to run the MPSL Init test, the TIMER and the RTC present only subtle differences in their models. The behaviour shown in Figure 3.3 applies for both. However, the TIMER peripheral does not have an accessible register with the counter value, therefore, for the TIMER model it is just presented as an internal variable. Also, the TIMER runs on the high-frequency clock in contrast to the RTC, which runs on the low-frequency one. This means the wait time between counts is much shorter, and in this case, since the preescaler value is also zero, there is a count every 62.5 ns instead of 30.157 us.

3.3.7 System Control Block

The SCB was implemented as a part of the CPU model. A structure that holds the SCB registers was created from which, during the MPSL Init test, only one register is accessed. This register is the System Control Register (SCR) which can configure all interrupts and events to wake up the processor or to only restrict this action to the ones enabled. Two generic functions to read and write to CPU registers were developed so that these can be used not only for this register but also for the other CPU registers in case the functionality of the model needs to be scaled. When the software needs to either set the SCR register in the SCB or to read it, it can make use of these generic functions.

3.3.8 Assembly Function Calls

The assembly function calls were also implemented inside the CPU model. The idea is that the software running in the virtual prototype substitutes the assembly calls for the ones implemented in the CPU. The CPU functions that implement the assembly calls during the tests are summarized in Table 3.4. The __NOP() assembly instruction is the most straightforward since as its name implies it does nothing. In the case of the __WFE() call, since it is only used to enter the sleep mode while the low-frequency clock starts, there is no real need to model it, and it can also be used as a __NOP(). This because the starting time of the low-frequency clock for the virtual clock model can be neglected. In the case of the __enable_irq() and __disable_irq() calls, they only need to take care of setting and clearing the PRIMASK register inside the CPU. The PRIMASK register prevents the activation of all exceptions with configurable priority in the CPU. Finally, the __get_PRIMASK() call is mapped to a function call that returns the value of the PRIMASK register.

Table 3.4: CPU functions that implement assembly function calls

Assembly call	Function in CPU	Function in CPU description
__WFE()	wfe_cpu()	Do nothing. Report that __WFE() was executed.
__NOP	nop_cpu()	Do nothing. Report that __NOP() was executed.
__enable_irq()	enable_irq_cpu()	Clear PRIMASK bit.
__disable_irq()	disable_irq_cpu()	Set PRIMASK bit.
__get_PRIMASK()	get_PRIMASK_cpu()	Return PRIMASK bit.

3.3.9 Memory Accesses

Even though there are not that many memory accesses to fixed addresses in the MPSL Init test, the few accesses that do exist needed to be taken care of since as they are, they would not allow the software to execute on the virtual prototype. It was necessary to inspect each of these accesses, determine their purpose and map them to a behaviour that could be modelled in the virtual device. The following are the accesses that needed to be handled:

- Two memory accesses configured a couple of registers in the clock peripheral which are not visible to customers but only internally for purposes of Nordic Semiconductor. In order to address this issue, the mentioned accesses were be substituted by usual peripheral accesses to the CLOCK peripheral.

- There are a couple of variables that take their value by reading two registers of the CLOCK peripheral by using a fixed address. These calls also needed to be replaced with regular peripheral accesses.

- There is a variable which reads a register from the Non-volatile Memory Controller by using a fixed address. However, this value is not used during execution, so this function call can just be ignored.

- Another memory access intends to get a part number of the device which is set during manufacturing. Since this is not relevant to execute the desired functions, the return value of this access can be hardcoded.

- The last access tries to retrieve some timing parameters, which again are not relevant and can be ignored.

3.3.10 Complete Virtual Prototype

Once all the models that are required to execute the MPSL Init test were finished, it was time to build a complete system in which the MPSL target tests could execute. Using Bauhaus as a base, it was necessary only to instantiate the modified CPU model and each of the peripherals models before attaching them to an interconnect element. It was also necessary to connect each interrupt request signal from the peripherals to the CPU so that the NVIC can take care of interrupts coming from them. Figure 3.4 shows a block diagram of the complete virtual prototype.

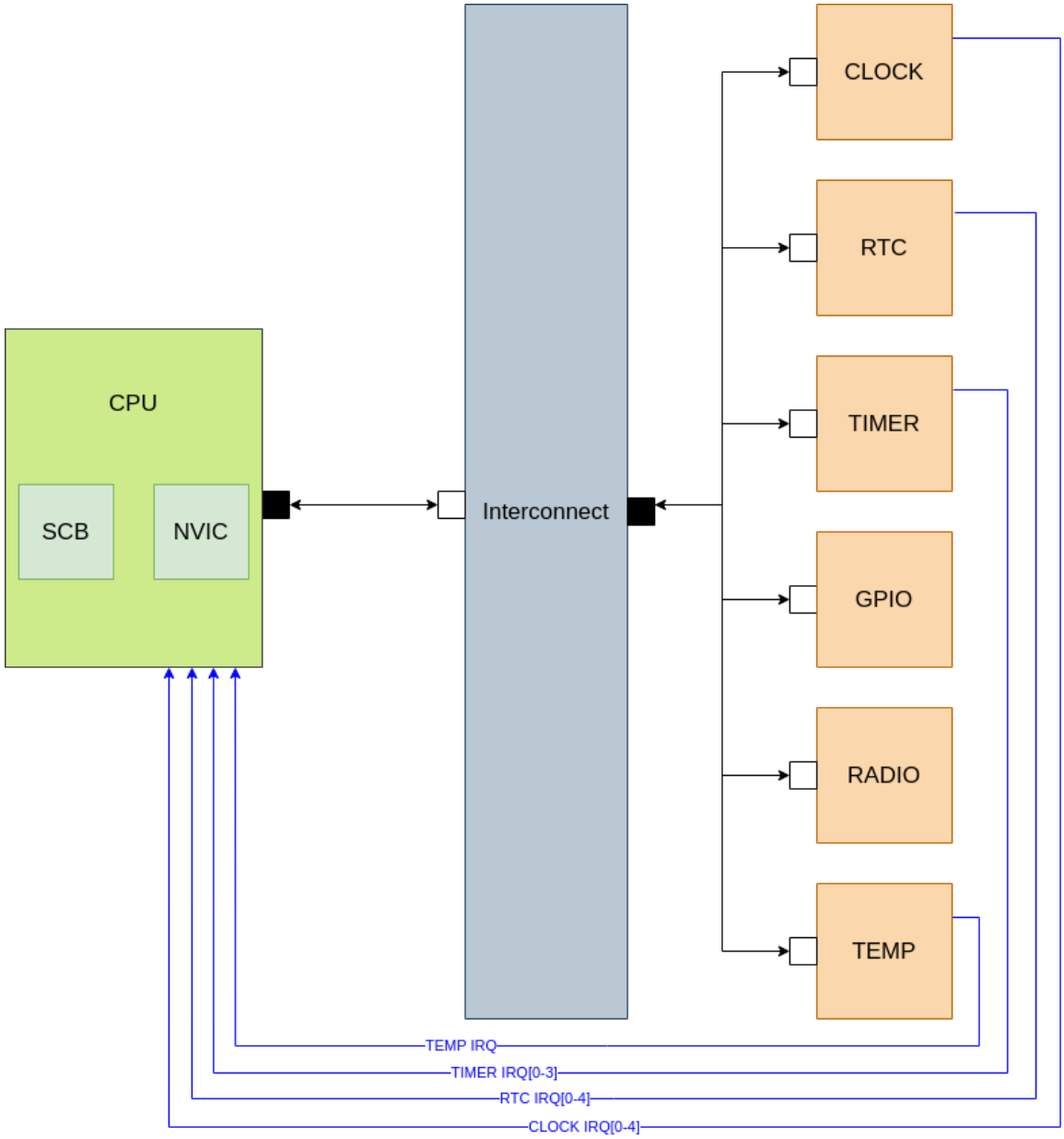


Figure 3.4: Representation of the complete virtual prototype

3.4 Virtual Prototype Verification

Before running any target test on the virtual prototype, it was necessary to make sure that every single component model and also the system as a whole work as intended. Therefore, a verification plan was designed for each hardware component and also for the complete system. Making sure that all the functional aspects of the prototype are tested, ensures a robust design which is ready to support the software tests that want to be executed. Verification plans are showed only for the models developed in this thesis.

3.4.1 2.4 GHz Radio Peripheral Verification

For the virtual prototype, the RADIO peripheral needs to model the behaviour of one register. Hence, the verification plan for this mode involves checking that writing a logic 1 to the LSB of the POWER register must turn the peripheral on and that writing a logic 0 must turn it off.

3.4.2 Temperature Sensor Peripheral Verification

The verification plan for the TEMP peripheral contemplates testing the following cases:

1. **Interrupted measurement:** when a temperature measurement is started and then stopped before it is complete, no temperature value shall be registered, and the peripheral must wait for the next start event.

2. **Normal measurement without interrupts:** when a temperature measurement is not stopped before completion, the TEMP register must hold a new temperature value after 50 us (hard-coded value representing the time the measurement takes). It should set the register that informs that the data is ready and trigger the corresponding event. Since interrupts are disabled by default, firing a data ready event with interrupts disabled should not fire an interrupt to the NVIC.

3. **Normal measurement with interrupts:** same case as the previous one but enabling interrupts. Firing a data ready event should trigger an interrupt signal to the NVIC.

3.4.3 Real-Time Clock Peripheral Verification

The verification plan for the RTC peripheral contemplates testing the following cases:

1. **Normal count:** in normal operation, the count shall increment every 30.157 us. When the counter stops, the count should remain. If the counter starts again, the count should resume where it stopped.

2. **Compare event without interrupts:** when a compare register is set to a value different than zero, the counter should trigger a compare event when reaching that count. If the interrupts are disabled, no interrupt should be fired to the NVIC.

3. **Compare event with interrupts:** same as the last test case but with enabled interrupts. If the interrupts are enabled when a trigger compare event is generated, an interrupt should be fired to the NVIC.

3.4.4 Timer Peripheral Verification

Since the TIMER and the RTC peripheral models for the virtual prototype share most of their functionality, the same verification plan for the RTC peripheral applies for the TIMER. The only difference being that the TIMER must wait less between counts and that the COUNTER register is not available anymore.

3.4.5 System Control Block and Assembly Calls Verification

Since the System Control Block and the assembly calls are both embedded in the CPU, it makes sense to verify them together. The verification plan for these components involve:

1. **System Control Block:** for the system control block, it is only necessary to check that it is possible to write and read the SCR register.

2. **Assembly calls:** for the NOP and WFE function calls it is only necessary to check that they are correctly called since they implement no functionality. For the functions that involve the PRIMASK register, the get_PRIMASK_cpu() function can be used to check that

calling the enable_irq_cpu() function clears the LSB of this register and that calling the disable_irq_cpu() sets this bit.

3.4.6 Complete Virtual Prototype Verification

Once every single component had been separately verified, the only thing left to check was that the communication between peripherals also worked as expected. In order to verify this behaviour, a simple test case was designed. The RTC peripheral was be configured to fire an interrupt when the count reaches the value of 10. When this happens, the CPU shall handle the interrupt and trigger a temperature measurement. When the measurement is done, another interrupt shall be generated and handled by the CPU to read the temperature value.

3.5 MPSL Init Test Execution

3.5.1 Running the MPSL Init and Uninit Functions

Before attempting to run the complete MPSL Init test, it was ensured that the system could execute the mpsl_init() and mpsl_uninit() functions which are common to all other tests. The first approach that was taken to run these functions involved only replacing the hardware accesses with the functions modelled in the CPU to interact with the virtual prototype. However, several obstacles were faced when following this idea:

- Inside the virtual prototype, the functions to write and read from registers are members of the CPU class that implements the CPU model. When a CPU object is created, the main software function that executes is a member of a Software class which must point to that specific CPU object. This means that all the functions called from main that require to use either the read or write functions need a pointer to the CPU class as an extra argument. Therefore, not only the hardware accesses needed to be modified but also function definitions and function calls.

- Since the virtual prototype at this point is only able to handle the hardware interactions inside the MPSL Init test, the source files that contain other accesses which are not yet

implemented will not compile. There is no way of translating these accesses because either the required register or peripheral does not yet exist.

- The MPSL source code is all written in C while the virtual prototype, based on SystemC, is written in C++. Even though it is possible to call C functions from C++ and vice versa, there were plenty of compilation issues faced when trying to interface the code and the virtual prototype.

Considering these factors, taking into account the complexity of the software stack and the dependency on multiple sources, another approach was taken to execute the mpsl_init() and mpsl_uninit() functions. Each of these functions was rewritten into a single C++ file. Each C++ file contained only the variables, macros, function declarations and definitions that the function needs to execute. All the function were modified to include the CPU pointer, and the hardware accesses were modified to use the virtual prototype instead. By doing so, the problems that were faced during the first approach could be addressed, and the functions could execute properly. This process is depicted in Figure 3.5.

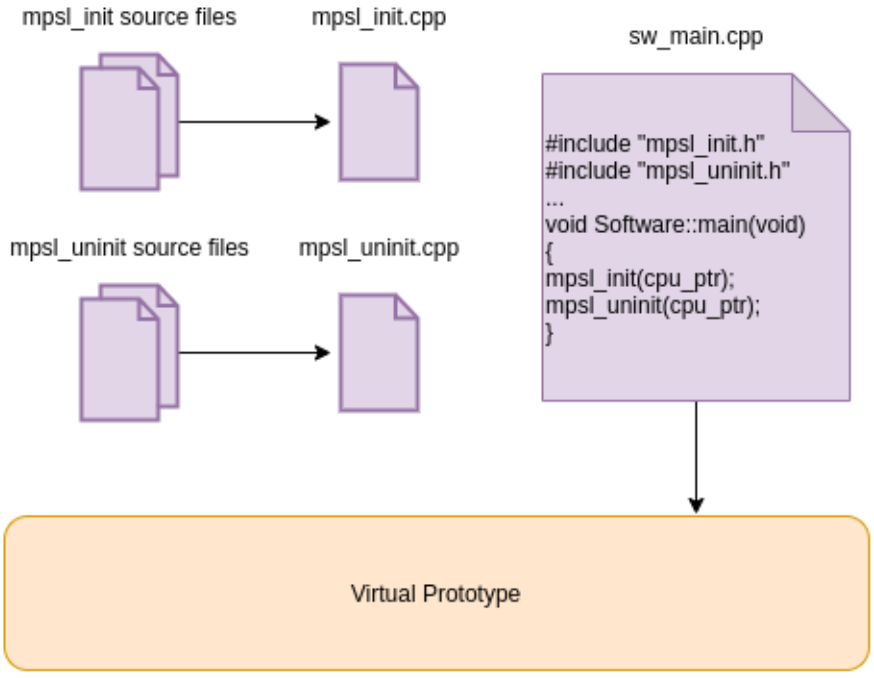


Figure 3.5: Execution approach of the MPSL initialization routines

3.5.2 Running the Complete Test

Once both the mpsl_init() and mpsl_uninit() functions ran successfully in the virtual prototype, the system was in a state in which it could be scaled to run the MSPL tests. The MPSL Init test checks that the mpsl_init() and mpsl_uninit() functions enable or disable the interrupts for the right peripherals. Listing 3.2 shows the body of the code for this test.

```
1  void fun_mpsl_init_001(void)
2  {
3    ASSERT_EQ(mpsl_init(&mpsl_clock_config, MPSL_TEST_IRQn,
       mpsl_assert_handler), 0);
4    /* Check that POWER_CLOCK and RTC0 IRQ is enabled by mpsl_init */
5    ASSERT(NVIC_GetEnableIRQ(POWER_CLOCK_IRQn));
6    ASSERT(NVIC_GetEnableIRQ(RTC0_IRQn));
7    ASSERT(NVIC_GetEnableIRQ(MPSL_TEST_IRQn));
8    /* RADIO, TIMER0, and TEMP IRQs shouldn't be enabled by mpsl_init */
9    ASSERT(NVIC_GetEnableIRQ(TIMER0_IRQn) == 0);
10   ASSERT(NVIC_GetEnableIRQ(RADIO_IRQn) == 0);
11   ASSERT(NVIC_GetEnableIRQ(TEMP_IRQn) == 0);
12
13    mpsl_uninit();
14   /* Check that IRQs for mpsl perihperals are disabled when MPSL is
       uninitialized */
15   ASSERT(NVIC_GetEnableIRQ(POWER_CLOCK_IRQn) == 0);
16   ASSERT(NVIC_GetEnableIRQ(RTC0_IRQn) == 0);
17   ASSERT(NVIC_GetEnableIRQ(TIMER0_IRQn) == 0);
18   ASSERT(NVIC_GetEnableIRQ(RADIO_IRQn) == 0);
19   ASSERT(NVIC_GetEnableIRQ(TEMP_IRQn) == 0);
20   ASSERT(NVIC_GetEnableIRQ(MPSL_TEST_IRQn) == 0);
21 }
```

Listing 3.2: MPSL Init test

As it can be appreciated in the test code, the test makes sure that the mpsl_init() function executes properly by returning zero, checks the state of some interrupts, executes the mpsl_uninit() function, and checks the state of the interrupts once again. These checks are performed through the ASSERT and ASSERT_EQ function, which stop the test and return a fail status when the condition they evaluate is not fulfilled.

3.6 Virtual Prototype Scaling

While the implementation of the MPSL Init test demonstrates that it is feasible to run MPSL target tests on top of a virtual prototype, running more complex tests would help to evaluate the scalability of the system. It would also allow gathering data for a more accurate performance assessment. Two extra tests were adapted to run on the prototype in addition to the MPSL Init test. These tests were chosen because they represent different degrees of effort regarding how much should the tests be adapted and the virtual prototype scaled to run a new test.

3.6.1 Temperature Measurement Test

The purpose of this test is to perform a temperature reading and then verify that it is within the expected range. Figure 3.6 shows how the test works. The functionality to perform a temperature reading is already contemplated in the mpsl_init() and mpsl_uninit() functions, but it does not get to execute in the simple test given the arguments passed to them. However, during the Temperature Measurement test, this function is called directly from the test body itself. Given that the TEMP peripheral model can support this reading, it was only required to modify the test and its dependencies to interface the virtual prototype instead of the hardware on the boards.

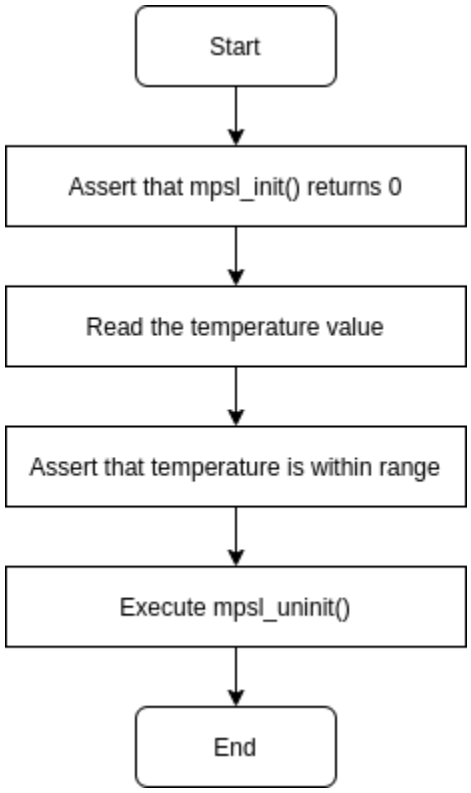


Figure 3.6: Execution flow of the Temperature Measurement test

3.6.2 One-shot Timer Callback Test

This test checks for the termination of two internal one-shot timer events. It initializes both events, starts the timers, waits until they are done, and verifies that the callbacks were executed for each of the timers. Figure 3.7 shows how the test works. Unlike the temperature measurement test, this one required not only modifications in the test code and its sources but also required to expand the virtual prototype. The test uses one NVIC function which was not implemented in the system. The NVIC model inside the CPU had to be modified to implement the NVIC_SetPendingIRQ function which takes the interrupt number as an input argument and sets the corresponding bit in the CPU register that handles the pending requests.

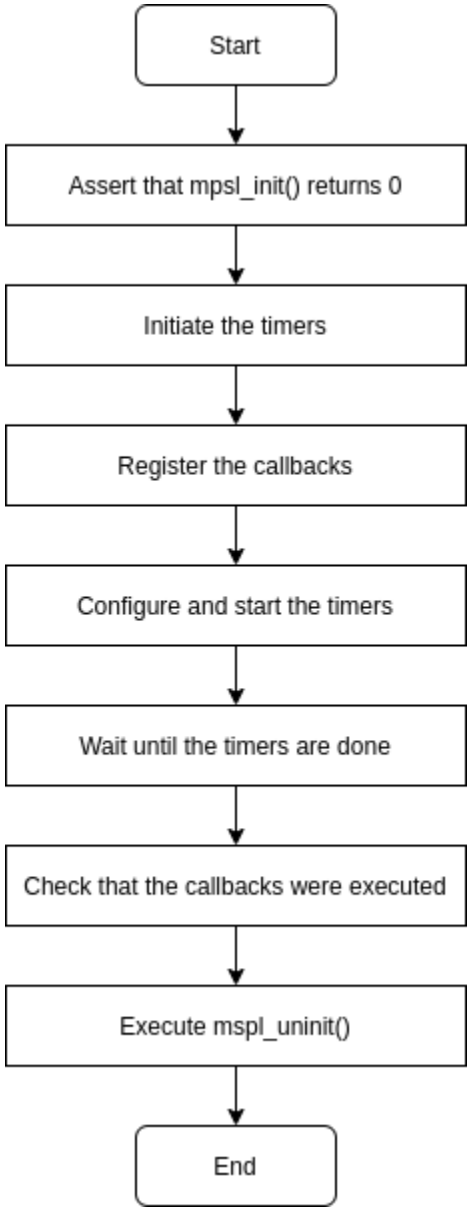


Figure 3.7: Execution flow of the One-shot Timer Callback test

3.7 Performance Evaluation

Prior to the development of this work, a set of potential benefits were expected from running MPSL target tests on top of the virtual prototype. These benefits included eliminating the need for physical hardware to test the software library, an improvement in the hardware observability throughout the execution of the test, and a speedup in the test execution time.

When evaluating the first two benefits, a qualitative approach has to be taken. Being able to run a successful test on top of the virtual prototype as well as being able to replicate errors can demonstrate that no physical hardware is needed to execute a test. Furthermore, all the information that is logged by the virtual prototype regarding the system transactions throughout the test execution can be compared to the simulation output when running the tests on the boards to evaluate the hardware observability. The speedup of the current implementation against the virtual prototype is the only quantifiable benefit of the three. The speedup can be computed by comparing the execution time of the tests running on the boards and the virtual prototype. The current implementation is already able to measure the test execution time; therefore a way to measure the test execution time on the virtual prototype had to be implemented in order to make a comparison.

3.7.1 Test Timing and Verbosity Control

In order to measure the execution time of tests running on top of the virtual prototype, the time.h library was employed, and timing points were added in the constructor and destructor of the top system class as depicted in Listing 3.3. By doing so, the timing feature is embedded in the virtual prototype itself, and there is no need to add timing code for every single test.

```
...
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)) return 0;
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}
...
 SC_CTOR(Top) : clk("clk", tech.clock_tech), pwr_dmn("pwr_dmn", tech.
    power_tech){
    start = get_wall_time();
    ...
}
...
 ~Top(){
    stop = get_wall_time();
    cout<<endl<<"Wall-clock Time = "<< 1000*(stop - start)<<" ms"<<endl;
  }
...
```

Listing 3.3: Execution time measurement approach

Also, since one of the objectives of the virtual prototype is to increase the hardware observability, the system is designed to print and log plenty of information related to the hardware activity. Both printings to the terminal and logging information represent a significant part of the execution time. Therefore a function to control the system verbosity was implemented. As shown in Listing 3.4, this function is called in the system constructor and makes uses of the SystemC report handler. This approach is quite useful since it allows to configure the verbosity of the whole system with just a few lines of code, avoiding the need to comment or uncomment every single debug line or the use of debugging macros. The function allows to disable both printing and logging, to disable only one of them, and even allows to only handle information regarding a specific element in the system for debugging purposes. By default, the system is designed to print and log everything, uncommenting sections of the function allows changing this behaviour.

```
...
void configure_verbosity(){
  // Uncomment to not log any information and not print anything
  //sc_report_handler::set_actions(SC_INFO,SC_DO_NOTHING);

  // Uncomment to not log any information and only print
  //sc_report_handler::set_actions(SC_INFO,SC_DISPLAY);

  // Uncomment to only log information and not print
  //sc_report_handler::set_actions(SC_INFO,SC_LOG);

  // For debugging only one or more system components
  //sc_report_handler::set_actions(SC_INFO,SC_DO_NOTHING);
  //sc_report_handler::set_actions("nrf_clock",SC_INFO,SC_DISPLAY|SC_LOG);
  //sc_report_handler::set_actions("nrf_radio",SC_INFO,SC_DISPLAY|SC_LOG);
  //sc_report_handler::set_actions("nrf_rtc",SC_INFO,SC_DISPLAY|SC_LOG);
  //sc_report_handler::set_actions("nrf_temp",SC_INFO,SC_DISPLAY|SC_LOG);
  //sc_report_handler::set_actions("nrf_timer",SC_INFO,SC_DISPLAY|SC_LOG);
  //sc_report_handler::set_actions("nrf_gpio",SC_INFO,SC_DISPLAY|SC_LOG);
  //sc_report_handler::set_actions("CPU",SC_INFO,SC_DISPLAY|SC_LOG);
  //sc_report_handler::set_actions("NVIC",SC_INFO,SC_DISPLAY|SC_LOG);
}
...
 SC_CTOR(Top) : clk("clk", tech.clock_tech), pwr_dmn("pwr_dmn", tech.
    power_tech){
  start = get_wall_time();
  configure_verbosity();
    ...
}
...
```

Listing 3.4: Verbosity control function

Chapter 4

Results

4.1 Virtual Prototype Simulation

The following results consist of a set of simulations that demonstrate how each hardware component model in the system, as well as the complete virtual prototype work as intended. These simulations were derived from the test cases proposed in the verification plan from the previous chapter. Results are presented only for the components that were modelled during this thesis.

4.1.1 2.4 GHz Radio Peripheral Simulation

The results of the simulation for the RADIO peripheral are shown in Figure 4.1. In this simple simulation, a logic 1 is written to the LSB of the POWER register, causing the peripheral to turn on. Immediately after that, a logic 0 is written to turn it off.

```
2 us: Info: CPU: CPU is starting
82 us: Info: nrf_radio: Received expected write request to ffc
82 us: Info: nrf_radio: Peripheral is powered on
127 us: Info: nrf_radio: Received expected write request to ffc
127 us: Info: nrf_radio: Peripheral is powered off
127 us: Info: CPU: main finished
```

Figure 4.1: Simulation accessing the POWER register of the RADIO peripheral

4.1.2 Temperature Sensor Peripheral Simulation

Interrupted Measurement

Figure 4.2 shows the simulation results for the first test case of the TEMP peripheral. The temperature measurement is started and then stopped after 45 us, precisely 5 us before the thread is expected to update the temperature value. Since the measurement was stopped, the temperature value read is 0. This means that the temperature measurement was not taken and the value was not updated.

Normal Measurement without Interrupts

A normal measurement is shown in Figure 4.3. The measurement is started, and precisely 50 us later, the measurement is ready. When the value is read, it reads the value of 42 (42 is the just a hard-coded temperature value since it was not relevant for the model to generate different temperatures). Also, since interrupts are disabled by default, it can be noticed that no interrupt is fired to the NVIC.

Normal Measurement with Interrupts

Figure 4.4 shows a simulation in which interrupts are enabled, and as expected, after the data measurement is ready, one interrupt is fired to the NVIC.

```
2 us: Info: CPU: CPU is starting
67 us: Info: nrf_temp: Received expected write request to 0
67 us: Info: nrf_temp: Temperature measurement started
112 us: Info: nrf_temp: Received expected write request to 4
112 us: Info: nrf_temp: Temperature measurement stopped
257 us: Info: nrf_temp: Received expected read request from 508
257 us: Info: nrf_temp: Temperature value 0
257 us: Info: CPU: main finished
```

Figure 4.2: Simulation of a stopped temperature measurement

```
2 us: Info: CPU: CPU is starting
67 us: Info: nrf_temp: Received expected write request to 0
67 us: Info: nrf temp: Temperature measurement started
117 us: Info: nrf temp: Temperature data ready
207 us: Info: nrf_temp: Received expected read request from 508
207 us: Info: nrf temp: Temperature value 42
207 us: Info: CPU: main finished
```

Figure 4.3: Simulation of a normal temperature measurement without interrupts

```
2 us: Info: CPU: CPU is starting
67 us: Info: nrf_temp: Received expected write request to 304
67 us: Info: nrf temp: DATARDY interrupt enabled
112 us: Info: nrf_temp: Received expected write request to 0
112 us: Info: nrf temp: Temperature measurement started
162 us: Info: nrf_temp: Temperature data ready
162 us: Info: nrf temp: Interrupt has been fired to NVIC
252 us: Info: nrf_temp: Received expected read request from 508
252 us: Info: nrf temp: Temperature value 42
252 us: Info: CPU: main finished
```

Figure 4.4: Simulation of a normal temperature measurement with interrupts

4.1.3 Real-Time Clock Peripheral Simulation

Normal Count

For the case of normal operation Figure 4.5 shows that after the counter is started, three counts were made after approximately 90 us. After the counter is stopped and the count restarted, one more count is made, which rises the counter value to 4 as expected.

Compare Event without Interrupts

In the case a CC register is set to a value different than zero, the timer should trigger a compare event. For this simulation, the CC0 register is set to a value of two, producing a compare event in the second count. Interrupts are disabled by default, so in this case, no interrupt is fired to the NVIC. Figure 4.6 shows this behavior.

Compare Event with Interrupts

In the case interrupts are enabled for the specific event compare that is triggered, an interrupt is fired to the NVIC. In the simulation results from Figure 4.7, it can be noted that interrupts are

enabled for the CC0 register. This produces a compare event and also fires an interrupt when the count reaches the value of two.

```
2 us: Info: CPU: CPU is starting
82 us: Info: nrf_rtc: Received expected write request to 0
82 us: Info: nrf_rtc: RTC started
112157 ns: Info: nrf_rtc: RTC incremented
142314 ns: Info: nrf_rtc: RTC incremented
172471 ns: Info: nrf_rtc: RTC incremented
192 us: Info: nrf_rtc: Received expected write request to 4
192 us: Info: nrf_rtc: RTC stopped
282 us: Info: nrf_rtc: Received expected read request from 504
282 us: Info: nrf_rtc: Counter value 3
327 us: Info: nrf_rtc: Received expected write request to 0
327 us: Info: nrf_rtc: RTC started
357157 ns: Info: nrf_rtc: RTC incremented
382 us: Info: nrf_rtc: Received expected write request to 4
382 us: Info: nrf_rtc: RTC stopped
427 us: Info: nrf_rtc: Received expected read request from 504
427 us: Info: nrf_rtc: Counter value 4
427 us: Info: CPU: main finished
```

Figure 4.5: Simulation testing the normal counter operation

```
2 us: Info: CPU: CPU is starting
82 us: Info: nrf_rtc: Received expected write request to 540
82 us: Info: nrf_rtc: Wrote 2 to CC0
127 us: Info: nrf_rtc: Received expected write request to 0
127 us: Info: nrf_rtc: RTC started
157157 ns: Info: nrf_rtc: RTC incremented
187314 ns: Info: nrf_rtc: RTC incremented
187314 ns: Info: nrf_rtc: Event compare 0 triggered
217471 ns: Info: nrf_rtc: RTC incremented
242 us: Info: nrf_rtc: Received expected write request to 4
242 us: Info: nrf_rtc: RTC stopped
287 us: Info: nrf_rtc: Received expected read request from 504
287 us: Info: nrf_rtc: Counter value 3
287 us: Info: CPU: main finished
```

Figure 4.6: Simulation testing a count compare event without interrupts

```
2 us: Info: CPU: CPU is starting
82 us: Info: nrf_rtc: Received expected write request to 540
82 us: Info: nrf_rtc: Wrote 2 to CC0
127 us: Info: nrf_rtc: Received expected write request to 304
127 us: Info: nrf_rtc: COMPARE0 interrupt enabled
172 us: Info: nrf_rtc: Received expected write request to 0
172 us: Info: nrf_rtc: RTC started
202157 ns: Info: nrf_rtc: RTC incremented
232314 ns: Info: nrf_rtc: RTC incremented
232314 ns: Info: nrf_rtc: Event compare 0 triggered
232314 ns: Info: nrf_rtc: Event compare 0 interrupt has been fired to NVIC
262471 ns: Info: nrf_rtc: RTC incremented
287 us: Info: nrf_rtc: Received expected write request to 4
287 us: Info: nrf_rtc: RTC stopped
332 us: Info: nrf_rtc: Received expected read request from 504
332 us: Info: nrf_rtc: Counter value 3
332 us: Info: CPU: main finished
```

Figure 4.7: Simulation testing a count compare event with interrupts

4.1.4 Timer Peripheral Simulation

The results of the TIMER simulation are presented similarly to the RTC peripheral since the same tests apply for both with small differences. Even though the TIMER peripheral does not possess a counter register, this was added to the model for verification purposes. Also, the timer increment log information is omitted due to the large number of counts made in each test.

Normal Count

For the case of normal operation, Figure 4.8 shows that after the timer starts, 719 (0x2CF in hexadecimal) counts were made after approximately 45 us. This is the expected value given the 62.5 ns between counts. After the timer is stopped and the count restarted, another 45 us go by, and the timer increases 719 units more which raises the count to 1438 (0x59E in hexadecimal).

Compare Event without Interrupts

In the case a CC register is set to a value different than 0, the timer should trigger a compare event. For this simulation, the CC0 register is set to 1000 (0x3E8 in hexadecimal), so a compare event is reached in that count. The count takes this value 62.5 us after the start event. Interrupts are disabled, so in this case, no interrupt is fired to the NVIC. Figure 4.9 illustrates this behavior.

Compare Event with Interrupts

In the case interrupts are enabled when a specific event compare is triggered, an interrupt is fired to the NVIC. In the simulation results from Figure 4.10, it can be appreciated that when interrupts are enabled for the CC0 register, an interrupt is fired when the count reaches the value of 1000.

```
2 us: Info: CPU: CPU is starting
82 us: Info: nrf_timer: Received expected write request to 0
82 us: Info: nrf timer: TIMER started
127 us: Info: nrf_timer: Received expected write request to 4
127 us: Info: nrf timer: TIMER stopped
217 us: Info: nrf_timer: Received expected read request from 504
217 us: Info: nrf timer: Counter value 2cf
262 us: Info: nrf_timer: Received expected write request to 0
262 us: Info: nrf timer: TIMER started
307 us: Info: nrf_timer: Received expected write request to 4
307 us: Info: nrf timer: TIMER stopped
352 us: Info: nrf_timer: Received expected read request from 504
352 us: Info: nrf timer: Counter value 59e
352 us: Info: CPU: main finished
```

Figure 4.8: Simulation testing the normal timer operation

```
2 us: Info: CPU: CPU is starting
82 us: Info: nrf_timer: Received expected write request to 540
82 us: Info: nrf timer: Wrote 3e8 to CC0
127 us: Info: nrf_timer: Received expected write request to 0
127 us: Info: nrf timer: TIMER started
189500 ns: Info: nrf timer: Event compare 0 triggered
217 us: Info: nrf_timer: Received expected write request to 4
217 us: Info: nrf timer: TIMER stopped
262 us: Info: nrf_timer: Received expected read request from 504
262 us: Info: nrf timer: Counter value 59f
262 us: Info: CPU: main finished
```

Figure 4.9: Simulation testing a count compare event without interrupts

```
2 us: Info: CPU: CPU is starting
82 us: Info: nrf_timer: Received expected write request to 540
82 us: Info: nrf_timer: Wrote 3e8 to CC0
127 us: Info: nrf_timer: Received expected write request to 304
127 us: Info: nrf_timer: COMPARE0 interrupt enabled
172 us: Info: nrf_timer: Received expected write request to 0
172 us: Info: nrf_timer: TIMER started
234500 ns: Info: nrf_timer: Event compare 0 triggered
234500 ns: Info: nrf_timer: Event compare 0 interrupt has been fired to NVIC
287 us: Info: nrf_timer: Received expected write request to 4
287 us: Info: nrf_timer: TIMER stopped
332 us: Info: nrf_timer: Received expected read request from 504
332 us: Info: nrf_timer: Counter value 72f
332 us: Info: CPU: main finished
```

Figure 4.10: Simulation testing a count compare event with interrupts

4.1.5 System Control Block and Assembly Calls Simulation

System Control Block

In order to test the SCB, it is only necessary to verify that writing and reading to the SCR register works. The simulation results of a test that verify this behaviour are shown in Figure 4.11. The test reads the register value which should be zero at the beginning, then writes 0xF to the register, and eventually reads 0xF as well.

Assembly Calls

Figure 4.12 shows the simulation for the assembly calls. First, it is verified that both the NOP and WFE function are properly called. For the rest of the calls, the get_PRIMASK() function is called to return the initial PRIMASK value, then the disable_irq() function is called to set the PRIMASK, and the value of 1 is returned. Lastly, the PRIMASK value is cleared by the enable_irq() function and finally returned one last time reading a 0.

```
2 us: Info: CPU: CPU is starting
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 0
37 us: Info: CPU: Received expected write request
37 us: Info: CPU: Wrote f
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read f
37 us: Info: CPU: main finished
```

Figure 4.11: Simulation of accesses to the SCR register in the SCB

```
2 us: Info: CPU: CPU is starting
37 us: Info: CPU: __WFE() was executed
37 us: Info: CPU: __NOP() was executed
37 us: Info: CPU: PRIMASK value 0
37 us: Info: CPU: __disable_irq() was executed. PRIMASK was set
37 us: Info: CPU: PRIMASK value 1
37 us: Info: CPU: __enable_irq() was executed. PRIMASK was cleared
37 us: Info: CPU: PRIMASK value 0
37 us: Info: CPU: main finished
```

Figure 4.12: Simulation of the assembly calls

4.1.6 Complete Virtual Prototype Simulation

Figure 4.13 simulates a test that checks the behaviour of the complete system and more specifically, the communication between peripherals through the use of interrupts. First, the value of 10 (0xA in hexadecimal) is written to the CC0 register in the RTC. Interrupts are enabled for this compare event and also for the temperature measurement. After this is done, the RTC is started. It can be noticed that after 10 counts, a compare event is triggered as well as an interrupt. The NVIC detects this interrupt and triggers the RTC interrupt service routine which stops the RTC counter, reads the initial temperature which is 0 (given that no measurements have been performed), and starts the temperature measurement. The measurement runs in the background, and when the data is ready, it triggers an interrupt which is handled by the NVIC. This time the NVIC executes the TEMP interrupt service routine and reads the new temperature value.

```
2 us: Info: CPU: CPU is starting
37 us: Info: NVIC: Interrupt vectors table is ready
82 us: Info: nrf_rtc: Received expected write request to 540
82 us: Info: nrf_rtc: Wrote a to CC0
127 us: Info: nrf_rtc: Received expected write request to 304
127 us: Info: nrf_rtc: COMPARE0 interrupt enabled
172 us: Info: nrf_temp: Received expected write request to 304
172 us: Info: nrf_temp: DATARDY interrupt enabled
217 us: Info: nrf_rtc: Received expected write request to 0
217 us: Info: nrf_rtc: RTC started
252 us: Info: nrf_rtc: RTC incremented
287 us: Info: nrf_rtc: RTC incremented
322 us: Info: nrf_rtc: RTC incremented
357 us: Info: nrf_rtc: RTC incremented
392 us: Info: nrf_rtc: RTC incremented
427 us: Info: nrf_rtc: RTC incremented
462 us: Info: nrf_rtc: RTC incremented
497 us: Info: nrf_rtc: RTC incremented
532 us: Info: nrf_rtc: RTC incremented
567 us: Info: nrf_rtc: RTC incremented
567 us: Info: nrf_rtc: Event compare 0 triggered
567 us: Info: nrf_rtc: Event compare 0 interrupt has been fired to NVIC
567 us: Info: NVIC: Interrupt has been received
567 us: Info: NVIC: RTC interrupt handler
612 us: Info: nrf_rtc: Received expected write request to 4
612 us: Info: nrf_rtc: RTC stopped
662 us: Info: nrf_temp: Received expected read request from 508
662 us: Info: nrf_temp: Temperature value 0
712 us: Info: nrf_temp: Received expected write request to 0
712 us: Info: nrf_temp: Temperature measurement started
762 us: Info: nrf_temp: Temperature data ready
762 us: Info: nrf_temp: Interrupt has been fired to NVIC
762 us: Info: NVIC: Interrupt has been received
762 us: Info: NVIC: TEMP interrupt handler
812 us: Info: nrf_temp: Received expected read request from 508
812 us: Info: nrf_temp: Temperature value 42
```

Figure 4.13: Simulation of two components communicating in the virtual prototype

4.2 Hardware vs. Virtual Prototype Comparison

4.2.1 Test Execution

Before making other comparisons between the current testing approach and the virtual prototype proof of concept, it was necessary to compare both implementations to assess their equivalency. In order to do that, passing and failing simulations were executed for the MPSL Init test on development boards and the virtual prototype.

First, it was checked that a passing MPSL Init test on the development boards passed as well on the virtual prototype. This check should hold since the only difference between both test implementations lies in the hardware interactions which are now performed in the previously validated virtual prototype. Figure 4.14 shows an extract of a passing execution for the MPSL Init test on the boards while Figure 4.15 does the same for the virtual prototype. The board simulation explicitly indicates that the test was successful. A test simulation in the virtual prototype is considered to be successful if the CPU main routine executes completely since any failure will abort the execution and prevent this routine from completing.

```
2020-05-04 10:22:38,421:INFO: CTF: ok (1.375s)
2020-05-04 10:22:38,424:INFO: CTF:
2020-05-04 10:22:38,425:INFO: CTF: ----------------------------------------------------
2020-05-04 10:22:38,425:INFO: CTF: Ran 1 test in 4.564s
2020-05-04 10:22:38,426:INFO: CTF:
2020-05-04 10:22:38,427:INFO: CTF: OK
2020-05-04 10:22:38,427:INFO: CTF:
2020-05-04 10:22:38,428:INFO: CTF: Generating XML reports...
2020-05-04 10:22:38,431:INFO: CTF: Generated XML report: verification/outcomes/results
```

Figure 4.14: Passing execution of the MPSL Init test on the board

```
Info: CPU: Initiated write transaction of 0x1 to 0x42000004

Info: nrf_rtc: Received expected write request to 4

Info: nrf_rtc: RTC stopped

Info: NVIC: IRQ 25 has been disabled

Info: CPU: main finished
```

Figure 4.15: Passing execution of the MPSL Init test on the virtual prototype

In order to check that tests running on the virtual prototype are not just always passing, it was necessary also to exercise a failing test scenario. A bug was introduced in the source code that executes during the MPSL Init test. The purpose of this bug was to produce a failure on the boards, and check if it was also replicated on the virtual implementation. In this case, the line of code inside the mspl_init() function that enables the MPSL test associated interrupt was commented (the MPSL Init test checks that this interrupt is enabled after running the mpsl_init() function). Figure 4.16 shows how this failure looks on the development boards. It is shown which line of code was the one that made the test fail and also a FAIL status. Figure 4.17 shows how the same failure was replicated on the virtual prototype. In this case, the simulation shows, as well, what caused the simulation to fail. Passing and failing conditions are checked through the tests by means of assertions. A failing assertion aborts the test execution and does not let the main routine finish as it is the case of the test execution on the virtual prototype.

```
2020-04-27 13:10:32,461:ERROR: CTF: !ASSERT: ../verification/tests/misc/src
/tp_fun_mpsl_init_001.c:32 (NVIC_GetEnableIRQ(MPSL_TEST_IRQn))
2020-04-27 13:10:32,470:INFO: CTF: FAIL (1.442s)
2020-04-27 13:10:32,472:INFO: CTF:
2020-04-27 13:10:32,474:INFO: CTF: ========================================
==============================
2020-04-27 13:10:32,475:INFO: CTF: FAIL [1.442s]: test_fun_mpsl_init_001 (t
ests.misc.misc_tests.mpsl_MiscTest_52fp1_tgt_armcc_nrf52)
```

Figure 4.16: Failing execution of the MPSL Init test on the board

```
Info: CPU: Initiated write transaction of 0x800 to 0x50000008

Info: nrf_gpio: Received expected write request to 8

Info: NVIC: IRQ 25 pending has been cleared

Assertion failed: cpu_ptr->NVIC_GetEnableIRQ(MPSL_TEST_IRQn)
in file /pri/dape/thesis/bauhaus_dev/bauhaus/prj/src/mpslvp/tests/misc/tp_fun_mpsl_init_001.cpp
at line 8

Abort (core dumped)
```

Figure 4.17: Failing execution of the MPSL Init test on the virtual prototype

4.2.2 Hardware Observability

Having compared the test execution equivalency on the development boards and the virtual prototype, it was possible to move on to other comparisons like the hardware observability. The current and virtual prototype implementations can generate log files that hold valuable information regarding the test execution.

In the case of the current implementation that involves development boards, the log file is an XML file like the one shown in Listing 4.1. Information like the number of executed tests, the number of failures, the test name, the execution time, and timestamp can be collected. However, it can be seen that there is no information about what the hardware is doing at all.

The virtual prototype, on the other hand, was designed to have hardware observability in mind. It is possible to keep track of every action that is being carried out by the hardware in a high level of abstraction. Figure 4.18 shows an extract of the log file generated by a test executing on the virtual prototype. As can be seen, there is plenty of information that is provided after the execution. Hardware components can inform about events like, for example, the creation of the interrupt vectors table. The CPU logs every write or read transaction along with the address of the register that wants to be accessed. The peripherals log information about each write or read request that was received. Peripherals not only register these requests, but they also inform their effects whether it is turning a peripheral on or off, starting a counter, disabling an interrupt, among others. All the events are logged with a timestamp, so a timing analysis of the execution can be performed if desired.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <testsuite errors="0" failures="0" file="tests/misc/misc_tests.py" name="
    tests.misc.misc_tests.mpsl_MiscTest_52fp1_tgt_armcc_nrf52" skipped="0"
    tests="1" time="1.390" timestamp="2020-04-22T10:46:51">
3   <testcase classname="tests.misc.misc_tests.
    mpsl_MiscTest_52fp1_tgt_armcc_nrf52" name="test_fun_mpsl_init_001" time
    ="1.390" timestamp="2020-04-22T10:46:51"/>
4 </testsuite>
```

Listing 4.1: Log file after executing the MPSL Init test on the development boards

```
0 s: Info: SystemBuilder:: System build successful
0 s: Info: Top:: +++ MPSLVP: Virtual Prototype for MPSL Target Tests +++
2 us: Info: nrf_clock: start initialization routine
2 us: Info: CPU: CPU is starting
37 us: Info: NVIC: Interrupt vectors table is ready
52 us: Info: CPU: Initiated write transaction of 0x0 to 0x41000ffc
87 us: Info: nrf_radio: Received expected write request to ffc
87 us: Info: nrf_radio: Peripheral is powered off
102 us: Info: CPU: Initiated write transaction of 0x1 to 0x41000ffc
137 us: Info: nrf_radio: Received expected write request to ffc
137 us: Info: nrf_radio: Peripheral is powered on
137 us: Info: NVIC: IRQ 1 has been disabled
```

Figure 4.18: Log file after executing the MPSL Init test on the virtual prototype

4.2.3 Execution Time

The last comparison between implementations involves the execution time to determine if there is a speedup when using the virtual prototype. This section presents how the execution time was measured, followed by the execution time comparison.

Execution Time Measurement and Verbosity Impact

By implementing timing points in the top-level system constructor and destructor, it was possible to measure the execution time of any test running on top of the virtual prototype. Figure 4.19 shows how this implementation can measure the execution time of the MPSL Init test.

```
Info: NVIC: IRQ 12 pending has been cleared

Info: CPU: Initiated write transaction of 0x1 to 0x42000004

Info: nrf_rtc: Received expected write request to 4

Info: nrf_rtc: RTC stopped

Info: NVIC: IRQ 25 has been disabled

Info: CPU: main finished

Wall-clock Time = 16.1099 ms
3445 dape@cad06%
```

Figure 4.19: Execution time measurement for the MPSL Init test

Figure 4.20 shows how the system verbosity can be configured to avoid displaying debug information on the terminal and also to avoid writing to the log file. It is evident how this configuration impacts the total execution time. On the other hand, Figure 4.21 shows how the system can be configured to only handle information about a single element in the system. In this case, the TEMP peripheral.

```
3446 dape@cad06% ./mpslvp.x

        SystemC 2.3.1-Accellera --- Dec 11 2018 15:01:04
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED

Wall-clock Time = 8.77285 ms
3447 dape@cad06%
```

Figure 4.20: Simulation output without verbosity

```
3448 dape@cad06% ./mpslvp.x

        SystemC 2.3.1-Accellera --- Dec 11 2018 15:01:04
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED

Info: nrf_temp: Completed constructor

Info: nrf_temp: Received expected write request to 4

Info: nrf_temp: Temperature measurement stopped

Info: nrf_temp: Received expected write request to 308

Info: nrf_temp: DATARDY interrupt disabled

Info: nrf_temp: Received expected write request to 4

Info: nrf_temp: Temperature measurement stopped

Info: nrf_temp: Received expected write request to 308

Info: nrf_temp: DATARDY interrupt disabled

Wall-clock Time = 10.9751 ms
3449 dape@cad06%
```

Figure 4.21: Simulation output displaying information only for the TEMP peripheral

Execution Time Comparison

After implementing a way to measure the test execution time on the virtual prototype, it was possible to compare it against the current board's implementation. Execution time was measured for the MPSL Init test, the Temperature Measurement test, and the One-shot Timer Callback test. Times from the board implementation were taken from the execution output that shows two different times: the total execution time and the test execution time. The test execution time refers only to the time it takes for the test to execute, while the total execution time also includes the time it takes to erase the memory and flash the code. Since the virtual prototype only measures the execution time of the test itself, the test execution time of the board implementation is the one used as a reference. Table 4.1 summarizes the results of the execution time comparison.

Table 4.1: Execution time comparison

Test name	**Execution time (ms)**		**Speedup**
	Development boards	**Virtual prototype**	
MPSL init	1390	8.92	155.82
Temperature Measurement	1384	9.55	144.92
One-shot Timer Callback	1535	10.28	149.31

Chapter 5

Discussion

A proof of concept of a virtual prototype able to run and test specific embedded software from the company Nordic Semiconductor was developed during this thesis. Evaluating possible benefits such implementation could deliver would allow considering future improvements that can lead to the incorporation of virtual prototypes into the software testing cycle of the company. Results show that it is possible to model a complex SoC with a level of abstraction that is high enough to run and test embedded software. They also demonstrate that, unlike the development boards, the virtual prototype provides observability of what occurs in the hardware when the software executes. Last but not least, the results show how software executes considerably faster in the virtual prototype than in the physical SoC on the development boards.

Simulation results show that each component of the virtual prototype, as well as the interaction between them, match the functional specifications that are required to run, at least, the MPSL target tests under study. A total of three tests, together with the source code they exercise, were modified to run on the virtual prototype with positive results. It was possible to replicate both passing and failing test executions just like they occur on the development boards. These executions translate to the fact that it is possible to modify target tests to run on a host computer without any physical hardware from Nordic Semiconductor. Under these circumstances, the software team does not need to wait until the hardware design is ready and manufactured to develop and execute their tests as they do now. The software team could do all this work earlier after the hardware specifications are ready, giving the chance to develop

and debug most of the code before the physical hardware is ready. This situation could significantly reduce the TTM of a new product, providing the company with a significant advantage against its competitors. It is important to point out that the proof of concept models the SoC partially and that including more peripheral models and refining the existing system is necessary to cover all the tests. Even though this is a time-consuming activity, once a robust SoC model has been developed, it can be used as a base for future models. Another factor to highlight is that modifying the tests and related code is a highly time consuming and manual task given the amount of code and the fact that it was not written with a virtual prototype in mind. Future work could address this problem by replacing the simple CPU model with a commercial CPU model like the ones provided by ARM [13]. The use of these kinds of models will eliminate the need to modify the source code since the commercial CPU model can handle hardware accesses like the ones that target the physical hardware. It will provide a more complete and accurate CPU that will allow focusing only on the peripheral development and their interface to the new CPU.

Another important outcome of the implementation is that the virtual prototype can provide observability of what is going on in the hardware during the software execution. Having this visibility was not possible using physical hardware. This feature may not be relevant to the software team, but other teams like the hardware architecture team may find it useful. The software team assumes all the hardware to be working, and they do not focus on testing the hardware but the software that will run on top of it. However, the hardware architecture team can benefit from this observability in early design stages to find bugs or make architectural improvements. By using execution results, architecture teams can identify, for example, which registers accesses are performed more often and modify the system to speed up these specific accesses. This kind of information can be used to make many important decisions before the SoC design is sent to the manufacturing facility, producing better quality devices. The system observability is, of course, limited to the level of abstraction of the models that make the virtual prototype. The abstraction level that the virtual prototype needs to run these tests may offer limited but still valuable information about the hardware.

Finally, results also show a significant speedup when running MPSL target tests on the virtual prototype compared to the physical SoC on the development boards. This speedup is possible due to the high level of abstraction of the models that constitute the virtual prototype. Since these models are TLM-based models, they simulate only data transactions, and all the complex communication protocols and delays that are present in the physical hardware are abstracted away. The execution time of the three implemented tests was close to 150 times faster. This speedup could represent a significant boost in the test design productivity. One limitation of the virtual prototype framework is that, unlike the setup, it is not possible to configure the system to run a set of tests or even all the tests at once. Future work could also adapt the python-based test control framework to work on the virtual prototype tests as it is currently done on the development boards.

Chapter 6

Conclusions

This thesis demonstrates, through a proof of concept design, that it is possible to develop a virtual prototype able to substitute a physical SoC to run and test proprietary embedded software in Nordic Semiconductor. The elimination of this hardware dependency enables developing and testing code earlier, reducing the TTM of new product designs. The virtual prototype also proved to deliver hardware observability during software execution, which can allow teams in charge of the hardware architecture to use this data to improve their designs. Lastly, it is shown how a virtual prototype can significantly speed the software execution and design, improving productivity. Even though the virtual prototype developed during this thesis, as a proof of concept, does not model the entire SoC and is not able to run all the existing tests, it can provide a good idea of the benefits of full-scale implementation. Future work suggests the use of a commercially licensed CPU model, the development of new peripherals and the improvement of the existing ones, the consideration of new tests and libraries beyond the MPSL, and the inclusion of test automation features.

Appendix A

Abbreviations and Acronyms

AMS Analog/Mixed-Signal library

AT Approximately-Timed

CCI Configuration, Control and Inspection library

CLOCK Clock Control

CRC Cyclic Redundancy Check

DMA Direct Memory Access

DMI Direct Memory Interface

DUT Device Under Test

ESL Electronic System-Level

GPIO General Purpose Input/Output

ISS Instruction Set Simulator

LSB Least Significant Bit

LT Loosely-Timed

MPSL Multi-Protocol Service Layer

NVIC Nested Vector Interrupt Controller

OSCI Open SystemC Initiative

RADIO 2.4 GHz Radio

RTC Real-time Clock

SoC System on Chip

SCV SystemC Verification library

SCB System Control Block

SCR System Control Register

TIMER Timer/Counter

TEMP Temperature Sensor

TLM Transaction-Level Modeling

TTM Time-to-Market

VP Virtual Prototyping

Bibliography

[1] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, 2014.

[2] "MPSL Library Internals." [Online]. Available: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrfxlib/mpsl/doc/mpsl.html

[3] "Nordic Semiconductor Infocenter." [Online]. Available: https://infocenter.nordicsemi.com

[4] G. G. Wang, "Definition and Review of Virtual Prototyping ," *Journal of Computing and Information Science in Engineering*, vol. 2, no. 3, pp. 232–236, 01 2003.

[5] T. Borgstrom, E. Haritan, R. Wilson, D. Abada, R. Chandra, C. Cruse, A. Dauman, O. Mielo, and A. Nohl, "System prototypes: Virtual, hardware or hybrid?" in *2009 46th ACM/IEEE Design Automation Conference*, July 2009, pp. 1–3.

[6] J. Sanguinetti, "A Different View: Hardware Synthesis from SystemC is a Maturing Technology," *IEEE Design Test of Computers*, vol. 23, no. 5, pp. 387–387, May 2006.

[7] G. Arnout, "SystemC standard," in *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, Jan 2000, pp. 573–577.

[8] P. Alexander, "Rosetta: Standardization at the System Level," *Computer*, vol. 42, no. 1, pp. 108–110, Jan 2009.

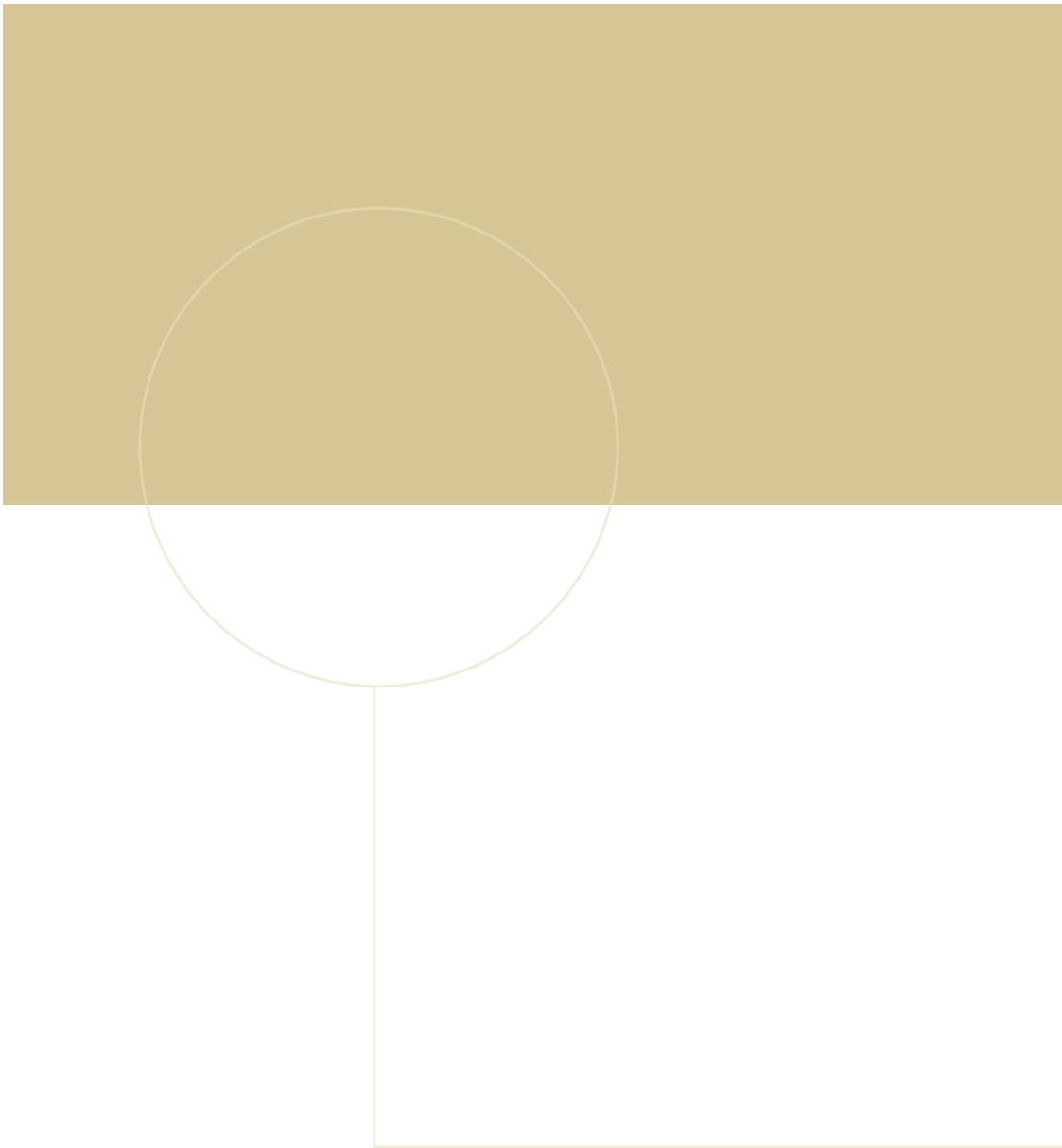
[9] P. L. Flake and S. J. Davidmann, "Superlog, a unified design language for system-on-chip," in *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, Jan 2000, pp. 583–586.

[10] "IEEE Standard for Standard SystemC Language Reference Manual," pp. 1–638, Jan 2012.

[11] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*, 2nd ed. Springer Publishing Company, 2009.

[12] "OSCI TLM-2.0 Language Reference Manual," 2009.

[13] "Fast models." [Online]. Available: https://developer.arm.com/tools-and-software/simulation-models/fast-models



NTNU

Norwegian University of
Science and Technology