NTNU – Trondheim
Norwegian University of
Science and Technology

TFE4930 ELECTRONIC SYSTEMS DESIGN, MASTER'S THESIS
THESIS REPORT

Virtual prototype of low-power digital architectures in SystemC

Written by :

Motaz Thiab

Supervisors :

Milica Orlandic

Isael Diaz

June 18, 2020

Abstract

Nordic Semiconductor is considering to include the use of SystemC virtual prototypes into their System-on-Chip (SoC) design flow. These prototypes would be useful in the development of embedded software. Nowadays, the software team at Nordic semiconductors tests their software in two stages, the first one is the Unit test, It's a set of functional tests of the software, which they run completely on the machine. And the second one is the Target test, They run more complex tests, it tests the communication between peripherals. The tests use one or two development boards, in the case of two devices, One device runs the software under test and the other runs a test program.

Our goal was to investigate the feasibility of using a virtual prototype in the design flow of Nordic Semiconductor. We selected some of the tests used by the software team at Nordic Semiconductor to decide what are the peripherals to be modeled and how detailed these models need to be. We have then modeled and verified the individual components and the overall system. Before running the tests, we had to modify them to be compatible with the SystemC prototype before running them on it. We evaluated the performance of the prototype for the tests and compare it with the performance of the development boards. The results showed that the virtual prototype provided a huge speedup in the testing which we believe would increase the testing productivity and test coverage. The prototype provides more visibility over the system components, allowing for more debugging capabilities. It is also proven to be easily expandable, either expanding the system by adding more models or expanding the individual models by adding extra functionalities.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

In System-on-Chip (SoC) development, both software and hardware need to be developed, where the hardware will interface with the external environment like peripherals and sensors. The software provides the interfaces between high-level software and the hardware components of the chip, which makes the software development an essential part of the hardware design. Software content on System-on-Chip (SoC) comprises low-level firmware or telecom/communication stacks, and these need to be tested and validated.
Here we are interested in the software testing of Nordic Semiconductor, where their software team relies on two types of software testing, unit tests, and target tests.

The issue that we are interested in is the productivity of testing, and we going to focus on the second stage of testing used by Nordic Semiconductor, target test. One of the target test drawbacks is its dependency on development boards. And in the early stages of SoC design flow, they will not be available which will force the delay of the SoC testing until the board design is finished and the board itself has been manufactured. A workaround to reduce the waiting time for development boards is the use of FPGAs. The FPGA can be configured to behave similarly as the under-development board. Even though the use of FPGAs will save the waiting issue it would add extra cost to the product development which is proportional to the number and type of used FPGAs.
Another issue target tests are suffering from is the time need to execute the tests. The number of tests that need to be run to cover a specific system increases with system complexity. And

having a complex system with different peripherals requires extensive software testing. With a massive number of cases and states to be tested, the testing time will increase drastically. This could force the developers to decrease the number of tests and focus on the corner cases to stay within a reasonable testing time.

An approach to reduce this effect is the use of virtual prototypes. A virtual prototype is a hardware-mimicking software that can run its software. The main principle of virtual prototypes is to have a trade-off between accuracy and simulation speed. It would provide a functional simulation while abstracting some of the unnecessary details to increase the simulation speed. Transaction level modeling (TLM) is one of the methodologies used to abstract the communication details between different system components. Its principle is to have an object of the payload that we would like to send from the initiator to a target, and instead of passing the whole object, the initiator would pass a pointer to that object. SystemC is one of the virtual prototyping languages that also use TLM methodology to transmit transactions between different system's components.

1.2 Thesis scope

This thesis is working with another one [1] to address the previously mentioned issue where we gonna be working on different parts of the same system, and the results for both theses will be based on the complete system which will be the combination of our work. We will try to answer on the following questing:

Does the use of SystemC virtual prototypes in the design flow of Nordic Semiconductor provide a good testing platform for their software? and whether it can replace the physical boards in software development?

We will be trying to develop a virtual prototype to run and evaluate the performance of these tests. The prototype will use some of the components that are already available and used by Nordic Semiconductor like a simple CPU that initiates simple read and write transactions, interconnect between the CPU and the rest of the models, and a system builder to connect different models into a single complete system. Our focus will be on the modeling of the peripherals and some CPU components like Nested Vector Interrupt Controller (NVIC) and System Control Block (SCB). We will also evaluate the performance of the prototype and

compare it with the tests' performance on the development boards.

1.3 Outline

The thesis consists of 6 chapters including the introduction, acronyms appendix, and appendix for some of the code mentioned through the thesis. Chapter 2 is the theory chapter and it discusses software testing in general then it narrows down to the testing methodology used at Nordic Semiconductor. It will also talk a bit about the software that we are going to test. Then it moves to talk about the hardware components that will be modeled either in this thesis or in [1]. We will also discuss SystemC as the used modeling language and explain some of its concepts and components, in addition to the in-house components that we have used from Nordic Semiconductor.

Chapter 3 will list the methodology and the steps taken through the thesis. It will start by discussing the selection of the initial subset of tests and the identifying of the hardware accesses performed by that subset of tests. It will also move to peripherals that are going to be modeled followed by their modeling and verification plans. Finally, it will discuss the modification of tests to be compatible with the prototype.

Chapter 4 will have the results of the verification plans and the results of executing the tests. It will also mention some of the features the prototype has like visibility and expandability. It will finish by listing a comparison between the virtual prototype and the development boards for different tests.

Chapter 5 will discuss the results and findings of the thesis and comment on them, and try to answer the thesis question. While chapter 6 will summarize the work, findings, and conclusion of the research.

Chapter 2

Theory

2.1 Software testing

The correctness of embedded software functionality and performance plays a big role in software quality. Software testing is an important asset of software quality assurance[7]. The embedded software testing stretches across the software and the hardware. It deals with the software and hardware issues, as well as their coordination [11].
There are several embedded development models that deal with different stages of development including the writing and testing of software. They are different level of testing that can be applied to embedded systems. We will discuss two of them here which are relevant to this thesis. The two types of embedded software testing are:

- Software unit testing:
  It tests a function or a class from the software, a single unit of the software. The test cases are developed based on the specification, and it tests the logic of the software.

- Software/hardware integration testing:
  It could also be referred to as target testing. It focuses on testing the interaction between the software within hardware components. It tests the integrity of communication between different parts of the software. It usually requires the use of physical boards, either development boards of the target hardware or configured FPGAs.

The main goal of all types of testing is to discover design bugs and errors at earlier stages of development. The later the bugs are being detected the higher the cost and the delay of

fixing them. The delay caused by discovering a bug depends on if we have to go back to previous stages of the design flow to fix it or how many stages we would have to revisit. The further we have to roll back in the design flow the more time and cost are needed.
The testing gets more complicated when moving to target testing as the target hardware of the software might not be available at this point in the design flow. As the software is tightly coupled with the hardware in an embedded system, this could be a bottleneck of the design flow. There are different ways to deal with this issue, either by delaying the software development until the target hardware is available, but the product development time would increase. Another solution is the use of configured FPGAs, that behaves similarly to the target hardware. This would overcome the bottleneck and save time in the development, but it would add extra costs to the development due to the high cost of FPGAs. A more recent approach is the use of virtual prototypes, which promises to allow for early testing at a cost cheaper than the use of FPGAs. This would be the focus of this thesis and will be discussed more in details later on in this chapter.

So to recap, all models aim to develop a design that fits the requirements and specifications as soon as possible while trying to minimize the development cost. Having the right testing solution for a certain product will increase the testing productivity. If the testing productivity increased this will lead to better test coverage or shorter software development time, which both can be translated to saved development costs. That might impact the final product in two major ways, first is being able to publish the product earlier to the market, in other words, it shortens the time-to-market (TTM). Second, is with lower development costs for the product there is the potential to have the product available for customers at a competitive price or increase the profit margin.

2.2 Current testing methodology

This thesis is building on the current testing approach at Nordic Semiconductor to try and provide a proof-of-concept for a potentially improved approach with increased testing productivity. Nordic Semiconductor design its own System-on-Chip (SoC), the Hardware access layer (HAL) to access it and its software.

And this part will present the software testing approach used at Nordic Semiconductor in their design flow. Their testing methodology consists of two stages, unit tests, and target tests.

2.2.1 Unit tests

Unit tests testing is an early stage of testing, that tests the software logic. It tests that individual pieces of code software are functioning correctly, separate from other software components. The unit tests are run on a machine, with a simple mock hardware file to mimic the existence of hardware. Mock hardware is a *C* code that has a dummy definition for some of the functions called in the tested software. The testing framework of unit tests is illustrated in 2.1. The unit testing framework consists of three main elements:

1. DUT (device under test) :

   Even though it is called the device-under-test it refers to the software to be tested. It represents the newly developed software that needs to be tested before further development. The DUT is being tested by checking the correctness of its functions' implementations. Its access to correct peripherals' registers in the correct order is also being covered in unit testing.

2. Mock hardware:

   This is a simplified virtual platform usually a single `C` file. it is a functional model to perform simple tests written in `C`. Each test has its own mock hardware to cover the function calls needed by each test.

3. Test control:

   It is a *C* file contains the definition of multiple tests that will be performed on a piece of the software. It initializes both mock hardware and the software. It returns the results of all tests indicating which tests have passed and which has not. It uses a unit test testing framework for *C* to define the tests and run them. The test control will return the results of the tests if they pass or fail.

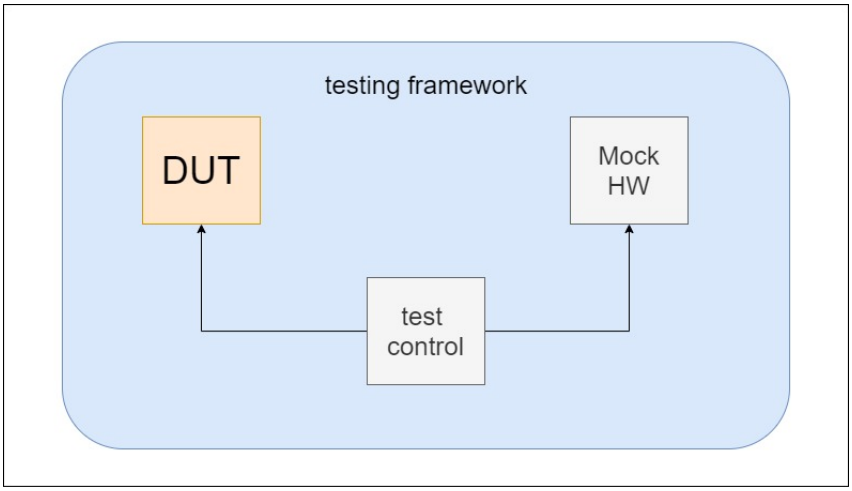

Figure 2.1: Unit testing

2.2.2 Target tests

This is a later stage of testing, which tests the communication between different peripherals and different parts of the system. It uses development boards to flash and run the tests. The test setup consists of two boards, one is loaded with the compiled software to be tested and the other contains the tester software as shown in figure 2.2. Both boards communicate via radio channel if the test requires. The test control block for figure 2.2 represents a machine, where both boards are connected to a machine during the testing. The machine runs the tests which include flashing both boards with tests code and logging the test steps and results.

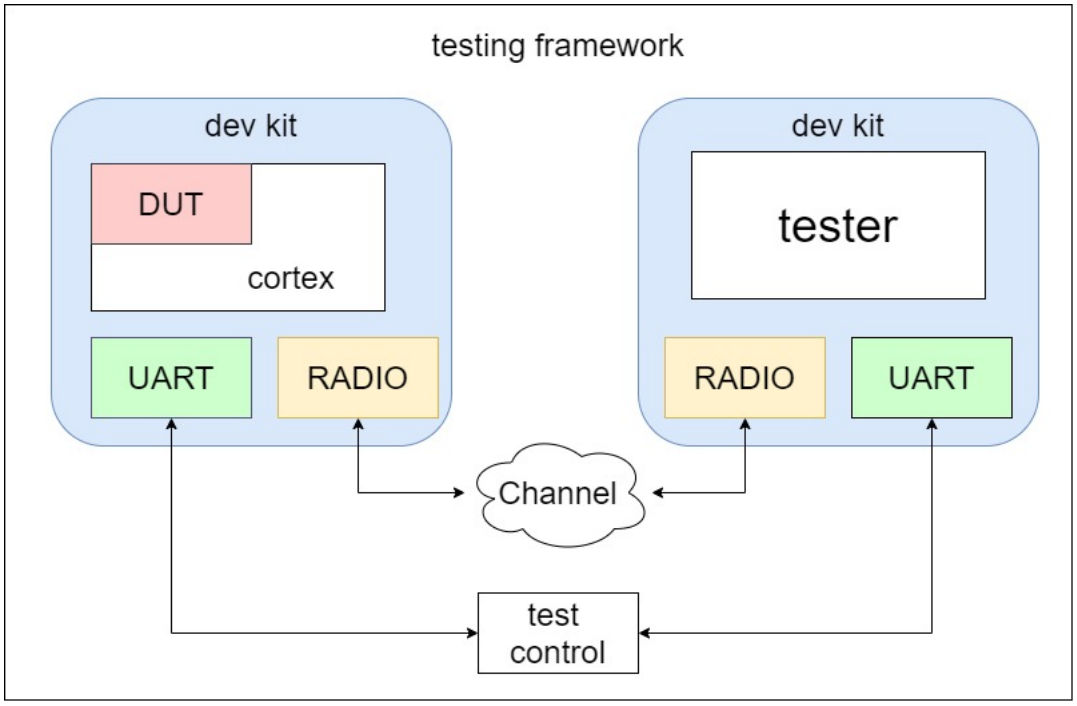

Figure 2.2: Target test

2.3 Virtual Prototyping, SystemC and TLM

This section will discuss the approach and the literature behind the methodology that will be used in this thesis. The topics that will be presented are wide and has extensive material describing all different aspects of them, but here we will introduce them and focus on the components and parts which are relevant to this work.

2.3.1 Virtual prototyping

In a general definition virtual prototype is an executable software that models a hardware system that runs on a host computer. A virtual prototype is binary compatible, meaning it can run unmodified binary images of entire software stacks. It abstracts away levels of hardware details essential to hardware developer but not relevant to a software developer, where it can be used similar to development boards by a software developer to load, execute and debug.

This abstraction allows the virtual prototype to run the simulation at higher performance compared to other hardware-oriented emulation. Virtual prototypes can be modeled to simulate part of the software stack, and continuously extend the virtual prototype along with

software development.
It shows that virtual prototyping allows software development to start earlier concurrently with hardware. Software/Hardware integration becomes easier using a virtual prototype instead of using FPGAs or development boards from a software developer perspective. The ultimate goal from the previously mentioned features that virtual prototypes provided is to reduce development time and effort, and subsequently, it could lower the development cost[4].

2.3.2 Transaction level modeling (TLM) methodology

Transaction level modeling (TLM) is a transaction-based modeling approach founded on high-level programming languages such as SystemC. It highlights the concept of separating communication from computation within a system. It is also used as a set of standards like TLM2.0, which ensures interoperability between different TLM compliant models. TLM compliant models referrers to the models that use the abstraction provided by TLM approach.

In TLM approach, the components of a system are modeled as modules, with concurrent processes to represent their behavior and functionalities. The modules exchange communication through interfaces, where processes can access these interfaces through module sockets. The communication exchanged is in the form of transactions, which is passed through channels. Channels are used by TLM to encapsulate communication protocol by implementing interfaces within channels. [3].
Various communication protocols can be defined on top of the core TLM interface layer. These protocols rely fully on the core TLM interface to transfer a transaction between two different points (modules) in a system.
The transfers of TLM are refined by these protocols in terms of transaction payload and blocking/non-blocking transfer[5]. The following Figure 2.3 shows how payload objects transferred as a transaction through the sockets of the modules from initiator socket to target socket. It also shows a typical TLM communication setup which consists of:

- **An initiator component**: it is a module that initiates (constructs and send) new transactions.

- **A target component**: it is the target module of transactions that acts as the endpoint of the transaction. It is responsible for executing requests from the initiator and send responses.
- **An interconnect component**: It forwards and routes transaction objects between initiator and target.

References to the object are passed along the forward path from the initiator socket to the target socket, and the responses are passed through return path form target sockets to initiator sockets.

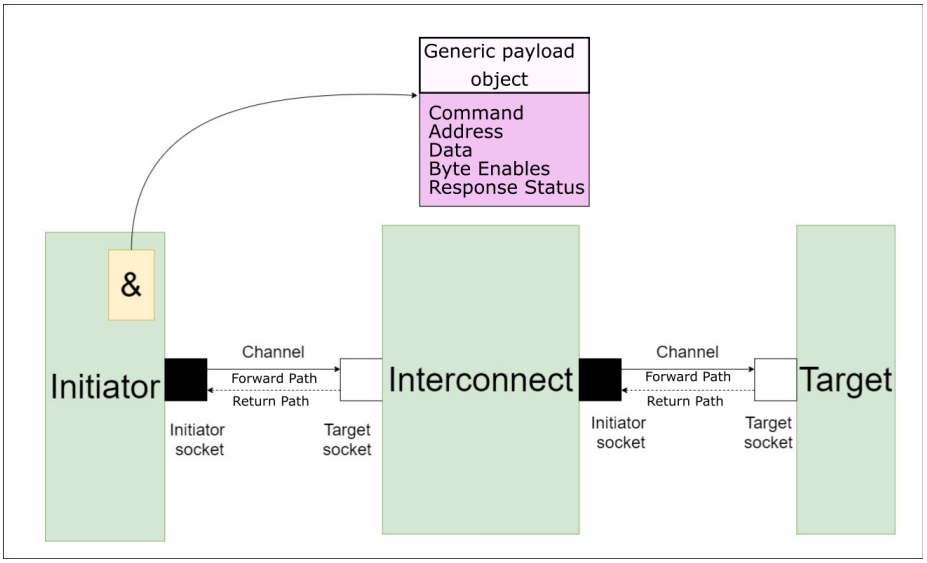

Figure 2.3: Model setup to communicate using TLM transaction

TLM models have different coding styles depending on the timing-to-data dependency that they must obey:

- Loosely-timed models:

  These models have a loose dependency between timing and data. They can provide timing information and the requested data at the point when a transaction is being initiated.

  These models prioritize the speed of the simulation, and so they are particularly useful for doing software development on a Virtual Platform. The processes in this style of TLM can run ahead of simulation time (temporal decoupling). Transactions completes in one blocking function call.

- Approximately-timed models:

  These models have a much stronger dependency between timing and data. They are not able to provide timing information and/or the requested data when a transaction is

being initiated. they are forced to trigger multiple context switches in the simulation, resulting in performance penalties. It is sufficient for architectural hardware design space exploration, since it introduced more details than the previous style.

Processes run in locksetp with simulation time and cannot run ahead of it, unlike the previous style. Transactions in this style have usually four timing points and non-blocking behaviour.

The TLM coding style that will be used is loosely-timed since the focus is to enhance software testing. And therefore the transactions will all be blocking transactions.

2.3.3 SystemC

There are different virtual prototyping languages, and here the focus is on *SystemC*. SystemC addresses the modeling of both hardware and software using C++ as it is not a separate language, but rather a set of classes and macros in a C++ library that supports hardware modeling. SystemC provides several hardware-oriented constructs that are not normally available in a software language like modeling concurrency, synchronization, inter-process communication and simulation kernel (scheduler) and figure 2.4 shows some of the elements used to do so. These constructs are essential for simulating hardware behaviour[2].

And here we will discuss the some of SystemC components from figure 2.4, and provide an overview of them.

Modules and Hierarchy

Large designs are mostly broken down hierarchically to manage the design complexity and ease the understanding of the design. SystemC provide constructs to implement hardware hierarchy. Some of these constructs is module entity. Modules are classes that inherit from the *sc_module* base class. They encapsulate design component and they may contains other modules, processes and channels and ports for connectivity. Instantiating these module classes in other modules creates the system hierarchy.

SystemC simulation kernel

SystemC simulation kernel consists of 6 phases, that govern the behaviour of the models and

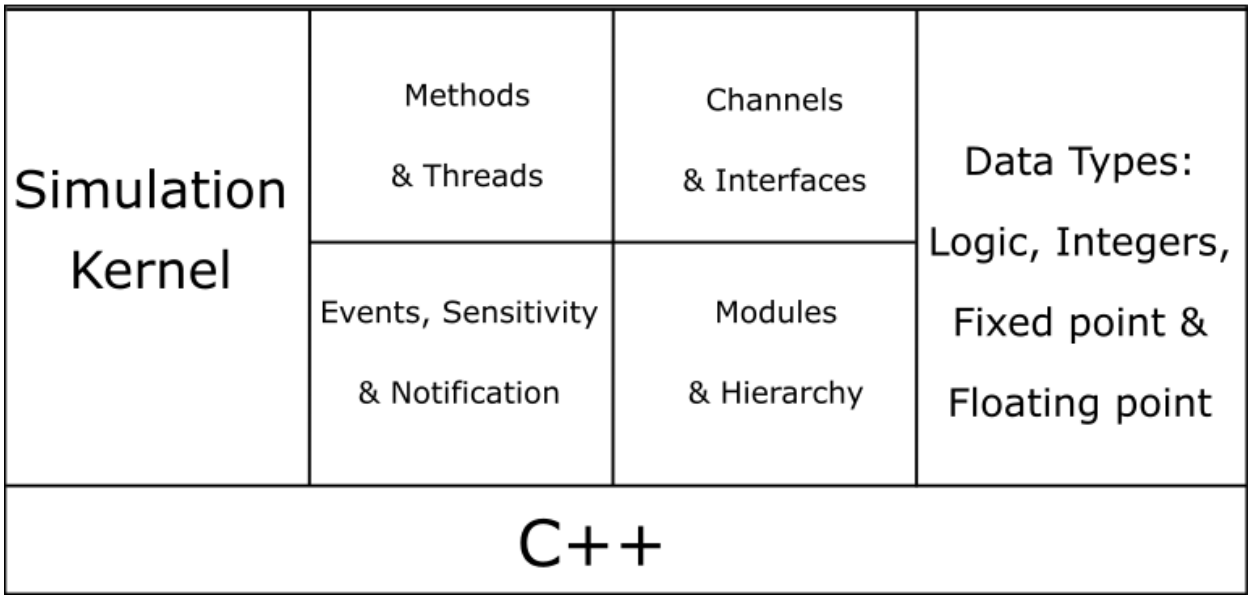

Figure 2.4: SystemC language architecture

simulate their execution concurrency. Figure 2.5 shows the phases of systemC simulation kernel.

1. Elaboration phase:

   This phase referrers to the execution of all states prior to the *sc_start()* call. All *SC_MODULES* are called during this phase and the connection between modules is checked, and if there a port which is not bound the simulation will complain about it at the beginning.

2. Initialization phase:

   During this phase, all processes are invoked in unspecific deterministic. Methods are executed once while threads will be executed if its synchronization point is reached(i.e. *wait()*).

3. Evaluation phase:

   SystemC simulator implements a cooperative multitasking environment where processes keep executing until they yield control At this phase, all processes marked as executable are executed successively an in undefined order, and their marking is removed. Methods are executed until they return, while threads get suspended by *wait()*. During their execution processes cannot be interrupted. Some of the processes might generate

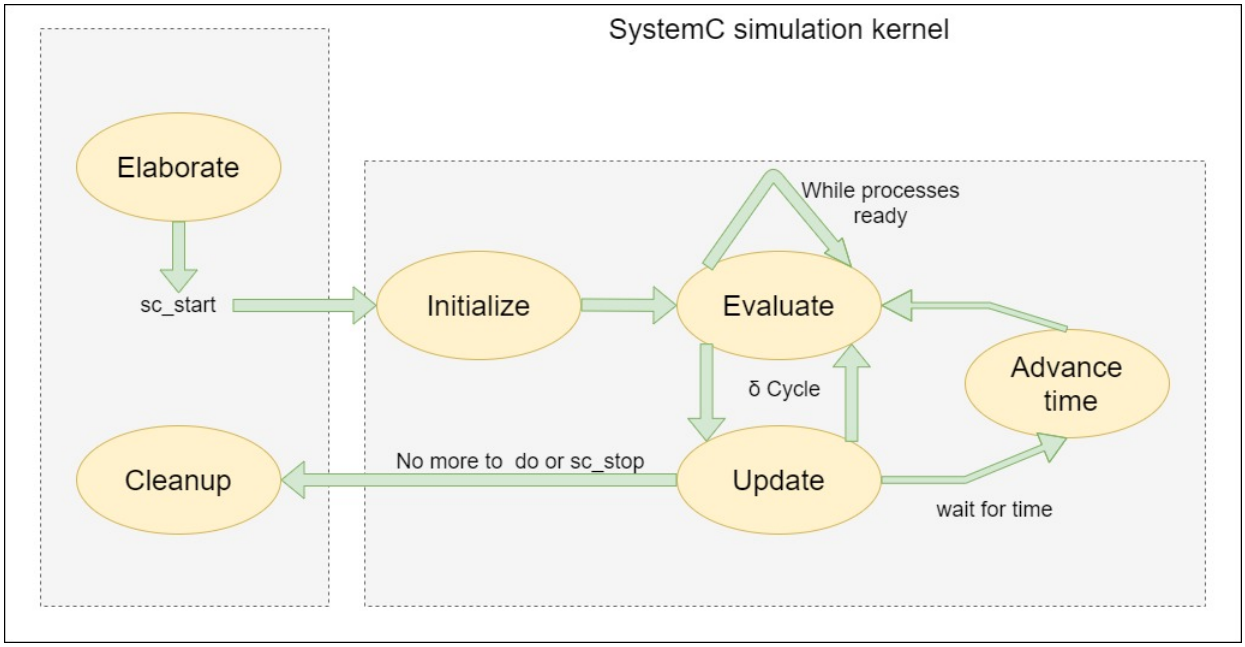

Figure 2.5: SystemC simulation kernel

"update request" by writing to *sc_signal* or *sc_fifio* etc. These requests are created for assignments to be made in the update phase. Furthermore, the execution of a *wait()* may result in a "timeout". This means that this process should be continued at a later time and they are stored in the event queue.

4. Update phase:

   In this phase the previously requested updates are performed. And the simulator estimates if any of the process are sensitive to the updates of these signals and if so marks them as executable. The scheduler goes back to evaluation phase to execute the processes marked as executable which might generate new update requests. The loop between the evaluation and the update phases is called delta cycle. This loop keeps going until there are no update requests left and no processes to execute.

5. Advance time phase:

   In this stage the simulation time is advanced to to time of events in the event queue with the smallest time. The processes sensitive to these events are marked as executable and the scheduler proceeds to the evaluation phase.

6. cleanup phase:

   If there are no events left in the event queue or if *sc_stop()* is called, the scheduler

proceeds to the cleanup phase. In the cleanup phase all destructors are called and the simulation is over.

SystemC Threads and Methods

The systemC simulation kernel schedules the executions of all simulation processes. Simulation processes are member functions of *sc_module* class that is registered with the simulation kernel during the elaboration phase. There are two ways of representing processes in systemC modules, Methods, and Threads. They register with the simulation kernel using *SC_METHOD* or *SC_THREAD* SystemC macro. Methods are similar to VHDL's process, and they can be invoked as often as needed. Methods are being executed atomically with no preemption, therefore, infinite loops must be avoided. Methods are usually sensitive to signals and events in the sensitivity list, and they do not take arguments and return no value.

While threads are started once at begin of the simulation and die when the end of its scope is reached, and for that, they can be suspended using *wait()* statements to wait for an event to be notified. Threads are recommended to have infinite loops to prevent the thread from reaching the end of its scope.
Both methods and threads are the basic blocks to simulate concurrency of execution. They are invoked by the simulation kernel, where the user has indirect control over which and when a process is invoked through events, notifications, and sensitivity.

Events, Sensitivity, and Notification

Events, sensitivity, and notification are essential for the simulation of concurrency in SystemC models. Events are implemented with the SystemC *sc_event* and *sc_event_queue* like shown.

```
1 //creating event that can be notified once during evaluation phase.
2         sc_event myEvent ;
3 // create an event that could be notified multiple times during the
     evaluation phase.
4         sc_event_queue myEvent_queue;
```

Events are caused or notified through the event member function *notify*. The notification can occur within a simulation process or as a result of activity in a channel. Events are notified during evaluation phase of SystemC kernel but the effect of the notification could occur at the immediately at the current evaluation phase or it could be time stamped to be notified at a point

in the future as shown below.

```
    void triggerProcess(){
//notify the event at next update phase
        myEvent.notify();
//notify the event after 10 ns from current time
        myEvent.notify(10,SC_NS);
        }
```

The difference between *sc_event* and *sc_event_queue* rises when the same event is notified multiple times at evaluation phase. In the case of *sc_event* only the first notification will be registered and all the following ones will be ignored, so in the example shown before the second notification will not be executed. While for *sc_event_queue* all the notifications will be stored and executed at their perspective times. There are two types of sensitivities, static and dynamic. Static sensitivities are specified in the constructor of the model at the elaboration phase for both methods and threads, and cannot be changed. While dynamic sensitivities allow the simulation process to change their sensitivity by calling different functions with the process. function *wait(myEvent)* is used for threads' dynamic sensitivity, and *next_triger(myEvent)* for methods' dynamic sensitivity.

SystemC Data Types

Digital hardware requires data types which are not provided inside the natural boundaries of C++ native data types. SystemC provides hardware-compatible data types that support explicit bit width for both integral and fixed-point. Non-binary hardware types are supported with four-state logic (0,1,X(unknown),Z(high impedance)) data types (e.g., *sc_logic*). Familiar data types like *sc_logic* and *sc_lv<T>* are provided for RTL hardware designers who need a data type to represent basic logic values or vectors of logic values. SystemC also allows the user to define new hardware types for new hardware technology.

Ports, Interfaces, and Channels

processes need to communicate with other processes both locally and in other modules. Processes communicate locally using events or channels. processes can also communicate across the boundaries of modules which may interconnect with other modules using channels.

Channels are used to separate communication from functionality and they act as a container of communication protocols and synchronization events. SystemC provides several built-in channels common to hardware and software design. They include locking mechanisms and hardware concepts like FIFOs, signals, and others. The ability to have interchangeable channels is implemented through a component called interface. An interface is defined as a set of pure virtual functions which is used as a base class. While channels provide the implementation of one or more interface(s).

2.4 Multiprotocol Service Layer (MPSL)

As this work is trying to improve the testing methodology used at Nordic Semiconductor, it uses Multiprotocol Service Layer (MPSL). It is a library that provides access to the peripherals of Real-time counter (RTC)0, TIMER0, CLOCK, and Temperature sensor. It allows higher-level software to communicate with the peripherals using MPSL functions.
The target tests of the MPSL runs test for different peripherals and different functionalities of the library. All different target tests have a similar structure. Each test calls the initialization routine *mpsl_init* and the beginning of the test. And an un-initialization *mpsl_uninit* routine is called when hardware accesses for the test are done. The function *mpsl_init* initializes the peripherals, configures some of their registers, and resets the interrupt lines (clear pending requests and disable them). While the function *mpsl_uninit* would stop some of the peripherals like CLOCK and resets others like the temperature sensor and the radio.

The following shows how MPSL tests are structured.

```
1  void mpsl_test(void){ //The test main function
2
3      //initializing the mpsl library
4      mpsl_init(&mpsl_clock_config,MPSL_TEST_IRQn, mpsl_assert_handler);
5
6      /*
7      *The body of the test
8      *all the function calls to mpsl library function happen here
9      */
10
11     //uninitializing the mpsl library
```

```
12      mpsl_uninit();
13
14      /*
15      *you can assert some values here,
16      *but not use any of the mpsl library functions
17      */
18  }
```

2.5 System components and peripherals

This section is discussing and explaining the behavior and functionalities of the components that we will base the models on. It includes the peripherals used by the tests and some CPU components. We will discuss the peripherals and components within the scope of this thesis since their description can get a bit complex and lengthy. So not all registers or functionalities of each peripheral will be explained, it will be limited to the functionalities and registers used by the selected tests.

2.5.1 Nested Vector Interrupt Controller (NVIC)

It is part of the CPU, and it controls the incoming interrupt requests from peripherals to the CPU. Interrupts are used as a way to divert the CPU from its current task to another task with a higher priority. The CPU can be triggered internally from the processor and it is called exceptions, or externally by external peripherals, and it is called interrupts. NVIC uses interrupt vectors to determine which service routine should be executed for a specific interrupt request. Each vector holds the address of interrupt handler function to a specific interrupt request. Interrupt vectors are arranged in interrupt vector tables. Interrupts have different priorities, which decide which interrupt service routine should be performed in case of multiple interrupt requests at the same time.

In ARM Cortex-M CPUs which are used in Nordic Semiconductor's System-on-Chip (SoC), the CPU receive the interrupt requests via Nested Vector Interrupt Controller (NVIC) as shown in figure 2.6 . The Nested Vector Interrupt Controller (NVIC) manages and prioritizes the external interrupts, where it can be used to enable or disable interrupt request lines. Some interrupt lines cannot be masked out(disable), they are referred to as Non-maskable interrupts

(NMI). NVIC interrupts the CPU with the highest priority interrupt request and the CPU stops the currently executing process and uses the interrupt request number to access the vector table and get the interrupt handler to start address for this specific request, which the CPU starts executing.

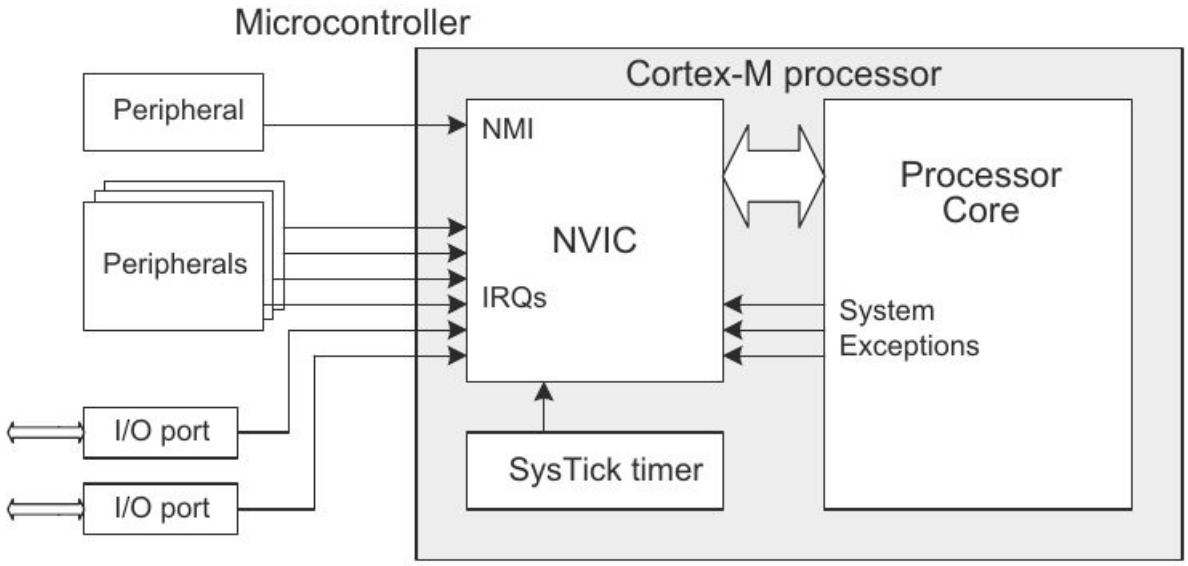

Figure 2.6: Nested Vector Interrupt Controller (NVIC) Block diagram

The Nested Vector Interrupt Controller (NVIC) consists of the multiple registers, but for the scope of the thesis we will limit the discussion to the ones used for the modeling, which are the following registers:

- Interrupt Set Enable Register (ISER):
  The bits in ISER correspond to different Interrupt request line (IRQ). The value of each bit decides if the corresponding IRQ is enabled or not in the NVIC. Setting any of the bits in ISER enables the interrupt of the corresponding IRQ, while clearing it does not affect the status of IRQ.

- Interrupt Clear Enable Register (ICER):
  ICER is similar to ISER that each bit corresponds to an IRQ. However, it differ in the functionality, where setting a bit in ICER would disable the corresponding IRQ, and its interrupt requests will not interrupt the execution of the CPU. Clearing bits in ICER has no effect of the IRQ.

Function	Description
NVIC_DisableIRQ	Enable interrupts
NVIC_EnableIRQ	Disable interrupts
NVIC_ClearPendingIRQ	Clears the status of IRQ pending
NVIC_GetEnableIRQ	Returns if IRQ is enabele or not
NVIC_GetPendingIRQ	Returns the status of IRQ pending
NVIC_SetPriority	Sets the priority of IRQ

Table 2.1: List of used Nested Vector Interrupt Controller (NVIC) registers

- Interrupt Set Pending Register (ISPR):
  ISPR is modified by the NVIC, where the the NVIC would set a bit if its corresponding IRQ has a pending interrupt request waiting for a process with higher priority to be over.

- Interrupt Clear Pending Register (ICPR):
  This register is a complement to ISPR, where ICPR is used to flag the IRQs which its pending requests have been cleared. If a bit is set, it means that its corresponding IRQ is no longer pending an interrupt request.

- Interrupt Priority Register (IPR):
  This register used to configure the priority of different IRQs. This register, unlike the ones before, takes an integer value to represent the priority of interrupt requests. The IRQ with lower integer value is the one with the highest priority and vice versa.

The table 3.3 lists the NVIC functions that are needed by the tested software. These functions operate on the NVIC registers mentioned earlier. All functions take IRQ as a sole argument expect *NVIC_SetPriority* which takes additionally the priority value of associated IRQ.

2.5.2 System Control Block (SCB)

SCB is also another component of the CPU similar to NVIC. It provides system implementation information and system control. This includes configuration, control, and reporting of the system exceptions. And it has different registers to store this information or to be configured, but here we will limit the discussion to the ones which are used by the tests. For the tests that we ran, there is a need for two register from System Control Block's registers, System Control Register (SCR) and System Handler Priprity Register (SHP) as shown in table

Register	Function
SCR	System Control Register
SHP	System Handler Priprity Register

Table 2.2: List of used System Control Block (SCB) registers

2.2. *SCR* controls features of entry and exit from low power state. While *SHP* is used to configure the priority of system interrupts (exceptions).

2.5.3 CLOCK

The clock control system can provide the system clocks from a range of high and low frequencies from internal or external oscillators. It also distributes different clocks to different modules based on the individual module's needs. Clock distribution is automated and grouped independently by modules to limit current consumption in unused branches of the clock tree [9].

Figure 2.7 shows the internal and external oscillators of the clock control of both High Frequency Clock (HFCLK) and Low Frequency Clock (LFCLK) controllers. The blue marked blocks are the sources of HFCLK. while the orange ones for the LFCLK.

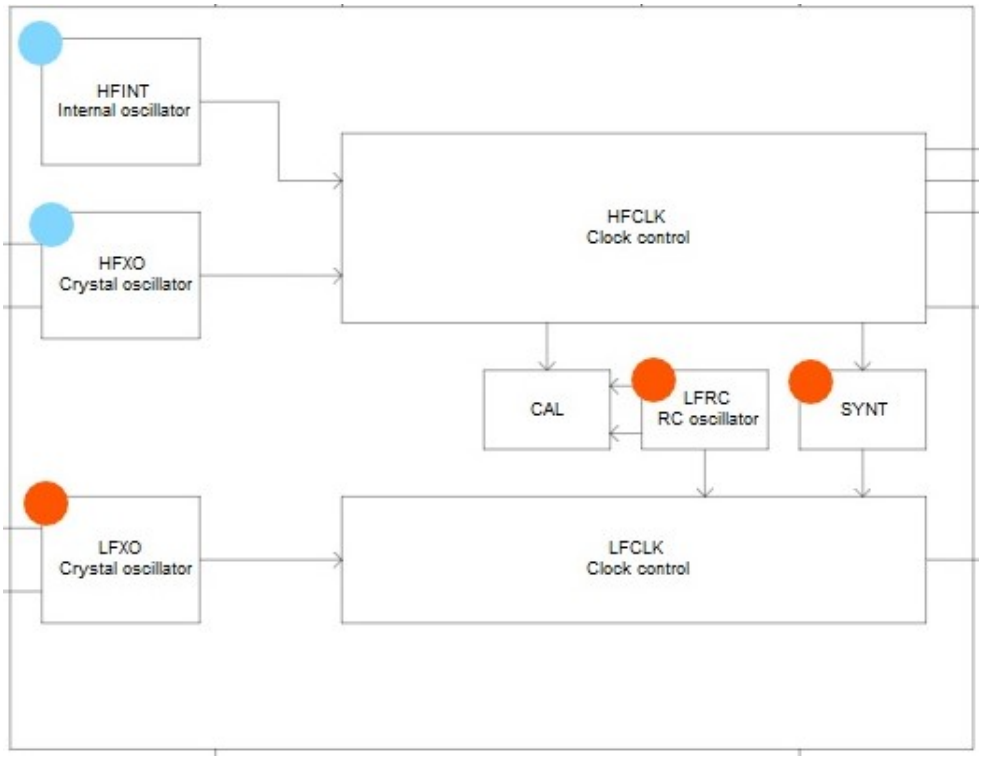

Figure 2.7: clock control block diagram,©Nordic Semiconductors

The High Frequency Clock (HFCLK) controller provides the high-frequency clocks when requested otherwise it enters power-saving mode. It can provide the following frequencies:

- 64 MHz CPU clock.
- 1 MHz peripheral clock.
- 16 MHz peripheral clock.
- 32 MHz peripheral clock.

On the other hand, Low Frequency Clock (LFCLK) controller can provide only a CLOCK of 32.768 KHz. It can generate this frequency from the supported following sources(marked in orange in figure 2.7):

- 32.768 KHz RC oscillator .
- 32.768 kHz crystal oscillator.
- 32.768 kHz synthesized from High Frequency Clock (HFCLK).

The RC oscillator is the default source of LFCLK. The RC oscillator can be calibrated since its frequency will be affected by variation in temperature. The High Frequency Clock (HFCLK) oscillator is used as a reference to calibrate the RC oscillator. A calibration timer is used to time the calibration interval of the 32.768 kHz RC oscillator, where the block *CAL* in figure 2.7 is responsible for the calibration process of the RC oscillator.

2.5.4 RADIO

The RADIO contains a 2.4 GHz radio receiver and a 2.4 GHz radio transmitter that is compatible with Nordic's proprietary 1 Mbps and 2 Mbps radio modes in addition to 1 Mbps and 2 Mbps Bluetooth® low energy mode. For the tests we are planning to run, they will not use the RADIO peripheral, but it is still part of the initialization routine. The register that is used from the CLOCK peripheral in the initialization routine is the power register to power up and shut off the peripheral.

Register	Function
NRF_RADIO->POWER	Peripheral power control

Table 2.3: List of used registers in RADIO peripheral

2.5.5 Real-time counter (RTC)

The RTC peripheral features a 24-bit COUNTER, a 12-bit (1/X) Prescaler, capture/compare registers, and a tick event generator for low power, tickless RTOS implementation. the RTC runs off 32.768 kHz of Low Frequency Clock (LFCLK) [10]. The value of the counter's *PRESCALER* register decides the overflow time value and the counter frequency (resolution) based on the following equation:

$$f_{RTC}[KHz] = \frac{32.768}{(PRESCALER + 1)}$$

For example, if the desired frequency is 100Hz (100ms counter period) then the *PRESCALER* register value will be as follow:

$$PRESCALER = (\frac{32.768kHz}{100Hz}) - 1 = 327$$

The RTC contains four compare registers *CC[0] - CC[3]*, these registers are configured with the counter values at which the RTC would trigger match event. When the *COUNTER* register value transitions from M-1 to M and one of the compare registers has the value M, an event for the corresponding compare register will be triggered. When a match event is triggered the corresponding *EVENT_COMPARE[i]* register value will be set to 1.
The figure 2.8 shows the block diagram of Real-time counter (RTC). And table 2.4 lists names and descriptions of the registers used by selected tests.

2.5.6 TIMER

The TIMER runs on the High Frequency Clock and it contains a four-bit (1/2X) Prescaler to divide the timer input clock from the HFCLK controller.
It operates in two modes Timer mode and Counter mode as shown in figure 2.9. In both modes, the timer starts by triggering the *START* task and stops by triggering the *STOP* task. Counting/timing can be resumed if it has been stopped by triggering *START* task again, and it

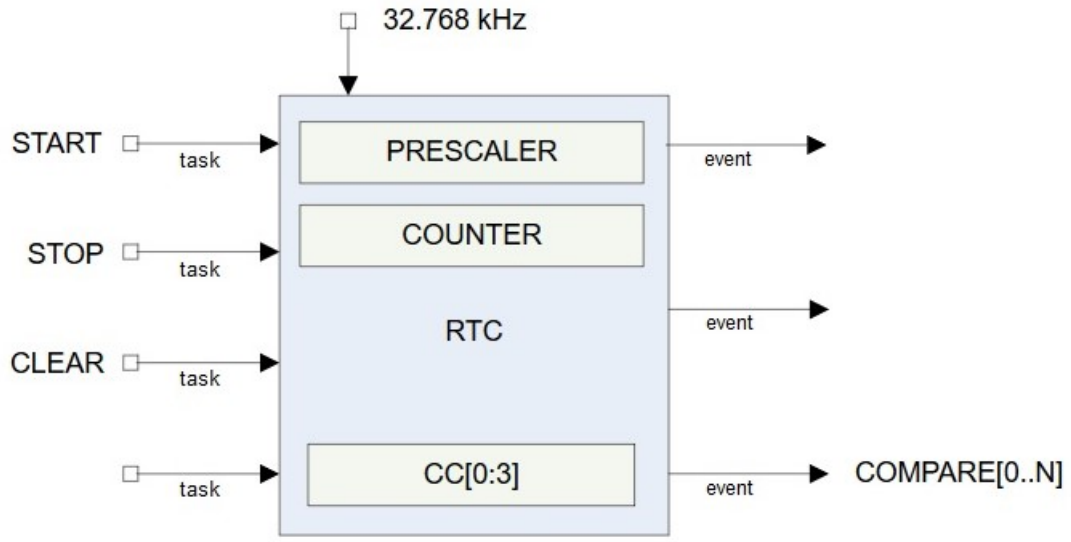

Figure 2.8: Real-time counter (RTC) block diagram,©Nordic Semiconductors

Register	Function
NRF_RTC0->TASKS_START	Start RTC COUNTER
NRF_RTC0->TASKS_STOP	Stop RTC COUNTER
NRF_RTC0->TASKS_CLEAR	Clear RTC COUNTER
NRF_RTC0->CC[i]	Capture/Compare register i
NRF_RTC0->EVENTS_COMPARE[i]	Compare event on CC[i] match
NRF_RTC0->INTENCLR	Disable interrupt
NRF_RTC0->EVENTCLR	Disable event routing
NRF_RTC0->EVENTSET	Enable event routing
NRF_RTC0->COUNTER	Current COUNTER value

Table 2.4: List of used registers in RTC peripheral

continues from the prior value it had before stopping.

During timer's mode, the TIMER internal register is incremented by 1 for each tick of the timer frequency. The timer frequency is derived from 16 MHz peripheral clock (PCLK16M) using the value specified in *PRESCALER* register according to the following equation:

$$f_{TIMER} = \frac{16MHz}{(2^{PRESCALER})}$$

. If f_{TIMER} is less than 1 MHz the timer will use 1 MHz peripheral clock (PCLK1M) instead of 16 MHz peripheral clock (PCLK16M). During the timer's mode, the *COUNTER* register is not used. On the other hand, in counter mode, the TIMER's internal register increments its value each time the COUNTER task is triggered. The timer's frequency and the timer prescaler are

not used in counter mode.

The timer can generate *COMPARE* events. One *COMPARE* event is generated for each capture/compare register. The *COMPARE* event is triggered when the *COUNTER* register's value reaches a value similar to one of the *capture/compare (CC[i])* registers.

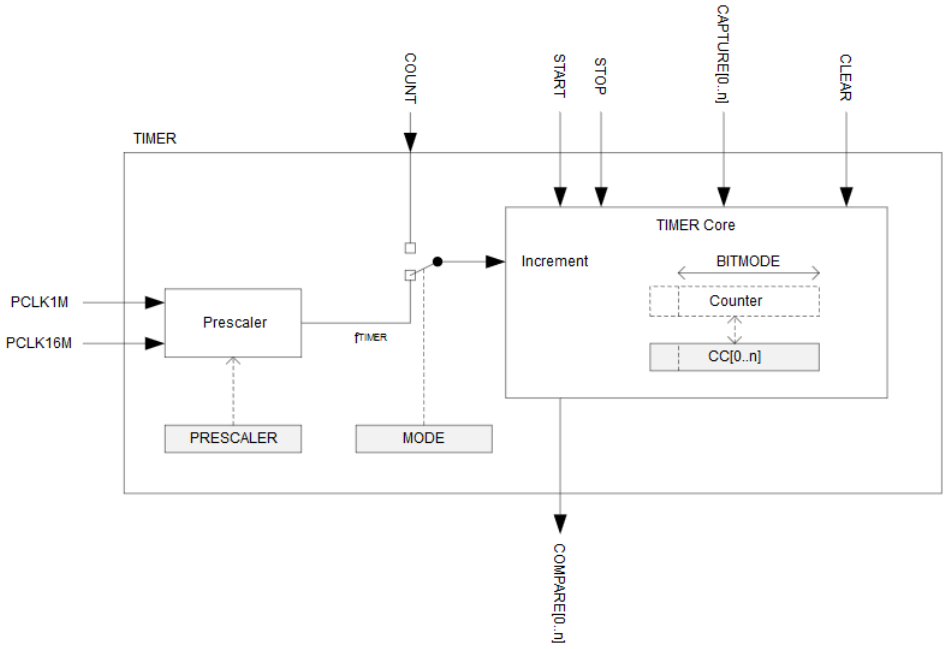

Figure 2.9: The timer block diagram,©Nordic Semiconductors

Register	Function
NRF_TIMER0->TASKS_START	Start Timer
NRF_TIMER0->TASKS_STOP	Stop Timer
NRF_TIMER0->TASKS_SHUTDOWN	Shut down timer
NRF_TIMER0->INTENSET	Enable interrupt
NRF_TIMER0->INTENCLR	Disable interrupt
NRF_TIMER0->CC[i]	Capture/Compare register i
NRF_TIMER0->EVENTS_COMPARE[i]	Compare event on CC[i] match

Table 2.5: List of used registers in timer peripheral

2.5.7 Temperature sensor

The temperature sensor measures the die temperature over the temperature range of the device. It has a resolution of 0.25 degrees. The temperature measurements triggered by the *START* task, and a *DATARDY* event is generated when the measurement is done and the result can be read from *TEMP* register. Temperature measurement supports the only one-shot operation and it powers down after measurement is completed to save power.
The temperature measurement does not start automatically and has to be explicitly started using *START* task. The table 2.6 lists the registers of temperature sensor which are used by selected tests.

Register	Function
NRF_TEMP->TASKS_START	Start temperature measurement
NRF_TEMP->TASKS_STOP	Stop temperature measurement
NRF_TEMP->EVENTS_DATARDY	Temperature measurement complete, data ready
NRF_TEMP->INTENSET	Enable interrupt
NRF_TEMP->INTENCLR	Disable interrupt
NRF_TEMP->TEMP	Temperature in ^{o}C (0.25^{o} steps)

Table 2.6: List of used registers in temperature sensor peripheral

2.5.8 General purpose input/output (GPIO)

The General purpose input/output has multiple ports with each port having 32 pins. The use of this module in our test is very limited. It is mainly used to set or clear some of the GPIO pins. The functionalities of the GPIO can be performed at a bit level, setting and clearing individual bits, where each bit in the GPIO registers corresponds to a pin of the port. Table **??** lists the registers used by the selected tests.
The register *OUTSET* of the GPIO sets the values of the pins based on their corresponding bits, if the bit is set then its corresponding bit in register *OUT* is also set. While if the bits are set in register *OUTCLR* then the corresponding bit in register *OUT* is cleared.
And the values of *OUT* register corresponds to the output of the port's pins, where the pin's value is HIGH if its corresponding pin in *OUT* register is set and it is LOW if it is cleared.

2.6 System core Bauhaus

The work of this thesis will focus mainly on modeling the peripherals for the system. Our system will be built on existing models at Nordic Semiconductor called Bauhaus. Bauhaus is a proof-of-concept and experimentation testbench for virtual modeling with SystemC used by Nordic Semiconductor [8].

It provides a simple CPU that initiate blocking read and write transactions to system components. It also provides system builder functionality to easily add new modules to the system with their base addresses. The top-level of Bauhaus institute the system CPU and allow developers to add their modules to the system. The binding of modules ports and signals is done at the top level as well.

When Bauhaus simulation start it starts be executing the CPU initialization thread. The CPU initialization thread calls the software routine after running initialization checks. The software routine is a member function of one of the CPU elements. The software routine's implementation contains the user-defined software or in our case the tests that we want to run on Bauhaus. The software routine uses a pointer to the CPU object to initiate read and write transactions to different modules of the system. Once the function is executed the CPU stops and the simulation is over.

Bauhaus generates log files of the simulation which capture all the transaction that have been carried out by the simulation with their time stamps. So as a user of Bauhaus you would have to take care of correctly binding the ports of your modules to the system. and write the read or write transaction that you want the system to simulate in the software routine using a pointer to a CPU object.

Chapter 3

Methodology

Figure 3.1 illustrates the proposed steps to carry out throughout this work. Steps have been explained in more detail in separate sections of this chapter. Methodology steps start with understanding and getting familiar with Nordic Semiconductor's technology and testing process. The work moved from simple to more complex tests to have an easier and less complicated debugging process.

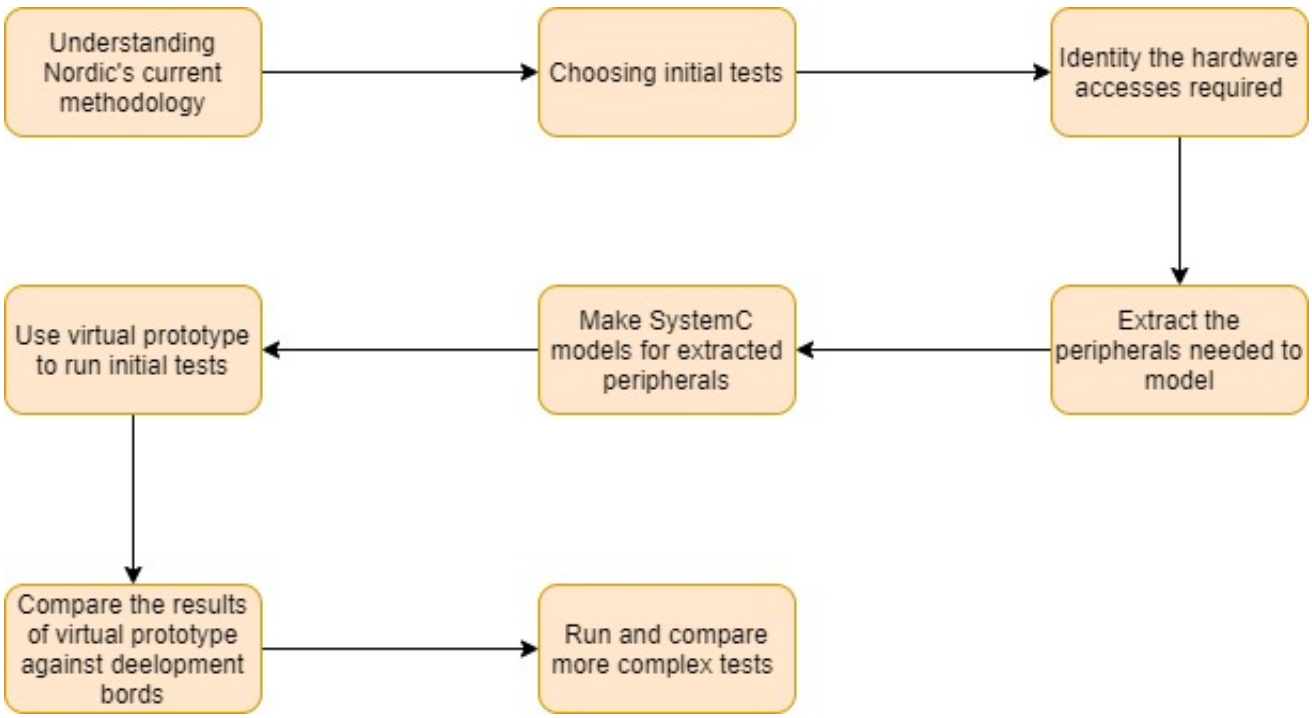

Figure 3.1: Proposed steps of methodology

3.1 The initial subset of tests

The first task of the thesis was to define an initial test to focus on, from the different tests performed by the software team at Nordic Semiconductors. Any chosen test to run on our virtual platform will affect its architecture in the following points:

- Modeled hardware

  Each test is performed to verify specific functionalities of the system carried by one or more peripherals. And in turn, only some of the peripherals' models will be needed for each test. And that will govern which of the system's peripherals are going to be modeled in our virtual platform.

- Peripherals' models' Details

  After deciding which peripherals are needed to be modeled to run a certain test on the platform, we had to decide how many details are present in the models. The details of each model include the registers and the different functionalities of each peripheral. The introduced details to each model should mainly cover the registers and functionalities of tests to be run.

The previous points show how significant the effect of tests on the complexity of the platform. That makes selecting the initial test to start model the system based on very critical. The initial test needs to provide meaningful testing to compare its results and performance with the development boards later on. It also should use a range of common peripherals that are reusable by future tests.

As mentioned in section 2.4, running any of Multiprotocol Service Layer (MPSL) tests follows a standard structure. All of MPSL tests assert the function *mpsl_init* at the beginning of the test and call the function *mpsl_uninit* at the end of the test as shown in script below 1.

```
1
2         void dummy_test (void)
3         {
4             ASSERT_EQ(mpsl_init(arguments),0);
5             /*
6             *
```

```
7               *The rest of the test body
8               *
9               */
10              mpsl_uninit();
11          }
12
```

Listing 3.1: MPSL test structure

The function *mpsl_init* initializes the system for other operations. It initializes some registers' values, configures the Nested Vector Interrupt Controller, and triggers different peripherals needed for the system to operate. While the function *mpsl_unint* performs similar to *mpsl_init* except it turns off some of the peripherals as they are not needed anymore.

For the initial test, the initialization test has been selected which we will refer to as the base test since it fits the criteria of the initial test and will be the base of all further tests to be implemented. The initialization test uses the functions *mpsl_init* and *mpsl_unint* and asserts the values of Nested Vector Interrupt Controller (NVIC)'s registers. The base test will access the peripherals implemented in *mpsl_init* and *mpsl_unint*. The next section will elaborate on the peripherals and functionalities that need to be implemented for the base test.

3.2 Identifying hardware accesses

To identify the accessed register for the base test we had to account for the peripherals' accesses by NVIC which implemented in the body of the test, and the hardware accesses due to the functions *mpsl_init* and *mpsl_uninit*. And to identify the accessed peripherals from the functions *mpsl_init* and *mpsl_unint* we went over the definition of each of the functions. We also identified which functionalities of the accessed peripherals are used. The figure 3.2 illustrates the used functions in *mpsl_init* which perform accesses to one or more peripherals. For example, in figure 3.2, the main routine *mpsl_init* calls for four immediate functions. They are listed with their accessed peripherals in table 3.1.

And figure 3.3 does the same for the function *mpsl_uninit*. The blocks with the same color indicate that the functions are in the same C file.

Functions	**accessed peripheral**
m_reset_and_clear_temp()	temperature sensor
m_reset_and_clear_radio	RADIO
mpsl_clock_init	CLOCK
rem_start()	TIMER and RTC

Table 3.1: immediate functions after *mpsl_init*

In both graphs, they software start from calling the main function either *mpsl_init* or *mpsl_uninit*. The arrows represent a function call from the function at the arrow's base to the function at its base. Each block has the name of the function and its containing file, with blocks with the same color existing in the same file. The functions illustrated are only the ones that perform one peripheral access or more. While other functions that do not access peripherals are used by the tests, they are not visible in the graph to keep its size from exploding and for irrelevance to the modeling process.

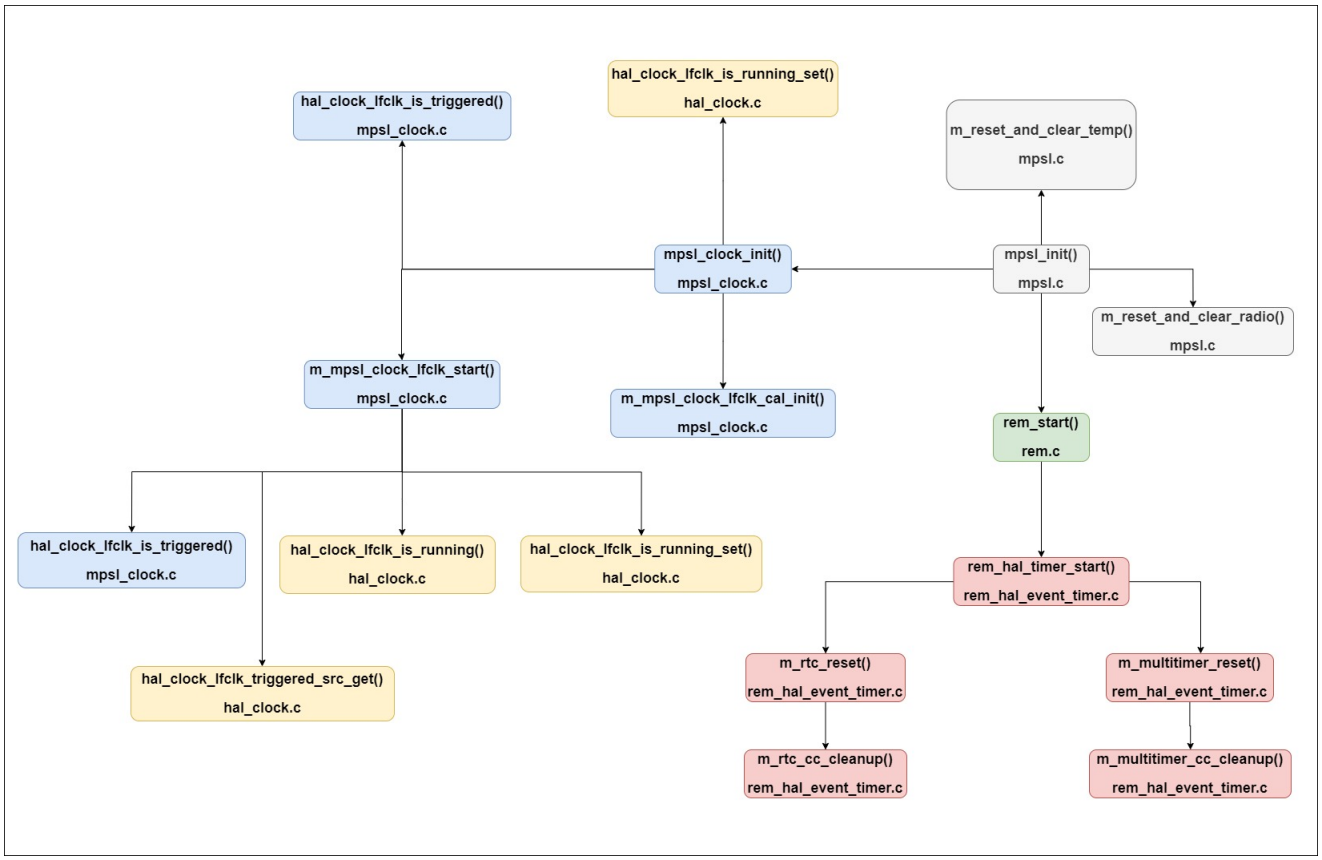

Figure 3.2: mpsl_init functions that access the peripherals

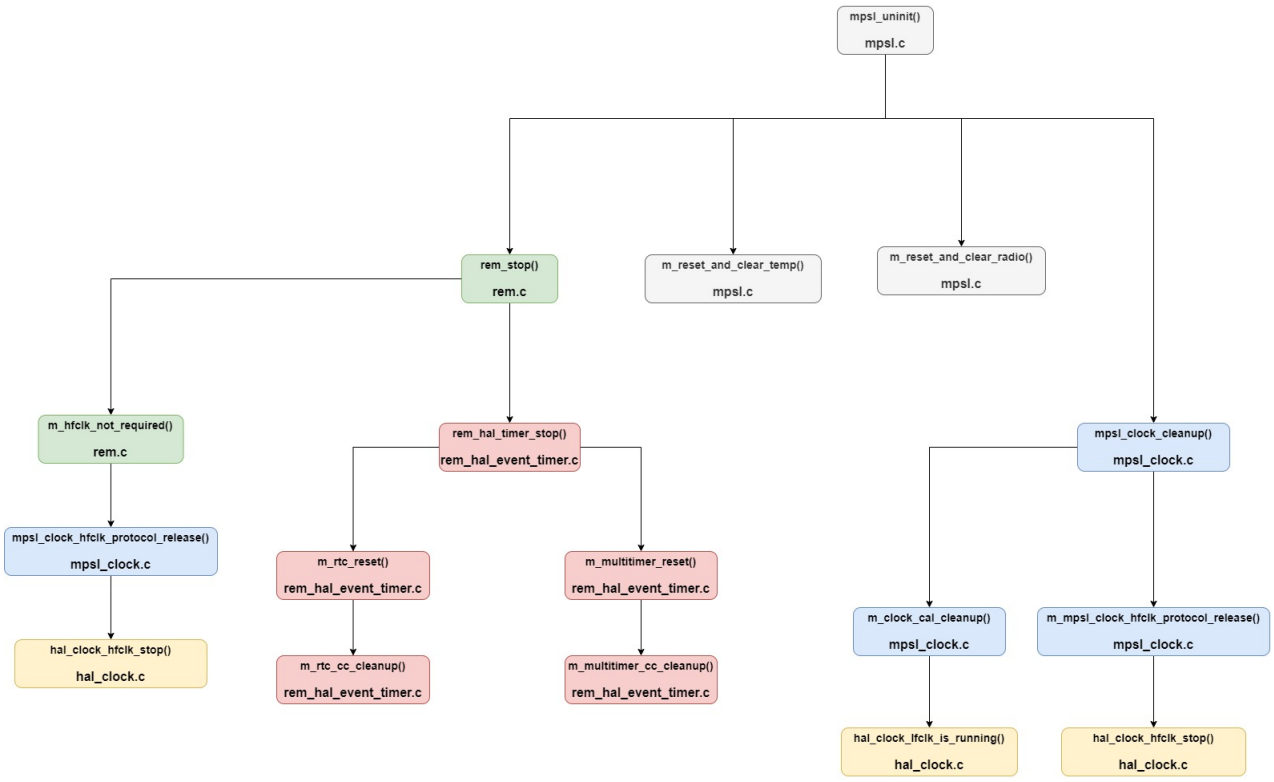

Figure 3.3: mpsl_uninit functions that access the peripherals

3.3 Extracted peripherals

After going through the peripherals' accesses needed to perform the base test, we have listed the peripherals that need to be modeled to be able to run the base test. The list is made by finding registers accesses performed in by the software and gather the accessed registers of each peripheral to form to realize how would the peripheral model will consist of. After gathering registers accesses to different peripherals we got a list of the following peripherals to model:

- CLOCK peripheral
- RADIO peripheral
- Real-time counter (RTC) peripheral
- Temperature sensor peripheral
- TIMER peripheral
- General purpose input/output (GPIO) peripheral

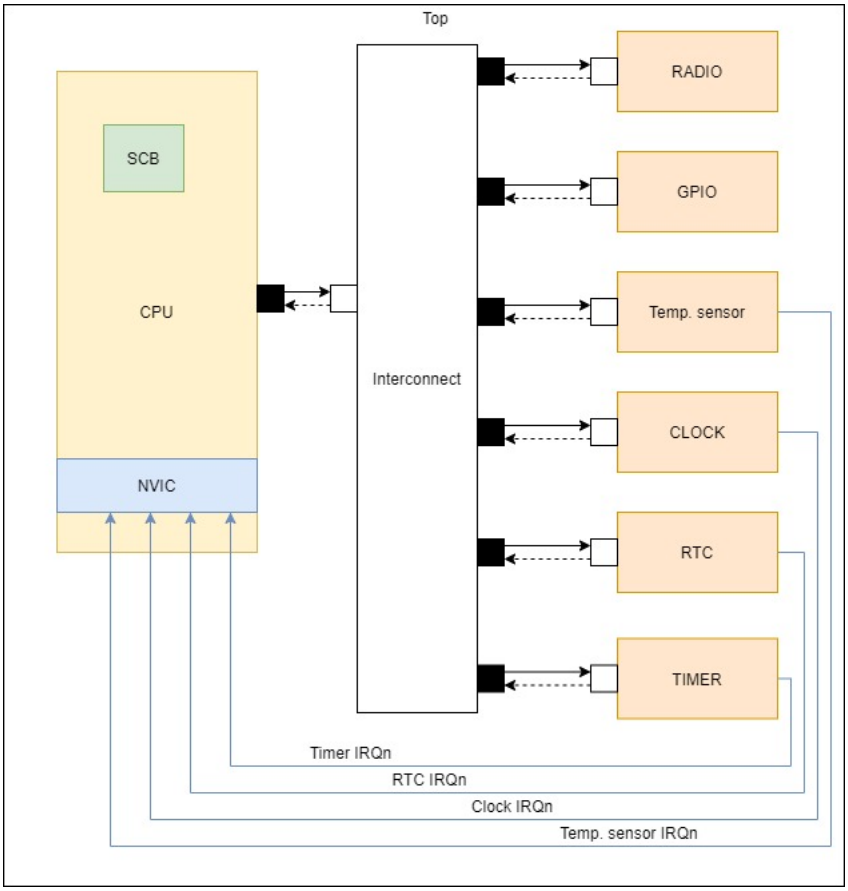

Figure 3.4: The virtual prototype structure

Figure 3.4 illustrates the architecture of the virtual platform in the SystemC environment. A detailed description of each peripheral models is in the next section.

3.4 Modeling of peripherals

We are both working on the same virtual prototype to cover its peripherals with each thesis developing a different set of peripherals models. To simplify the collaboration between two theses we have used a peripheral's model template. It aims to standardize the modeling process and reduce the complexity of integrating the models together to build the platform.

The template provides a target socket for the models to connect to the interconnect with a blocking transport implementation to communicate over the interconnect. It also has a generic base structure to define registers of each peripheral and generic read and write functions to and from these registers. The interaction (read and write) with a peripheral registers in the template

NVIC registers
Interrupt Set Enable Register (ISER)
Interrupt Clear Enable Register (ICER)
Interrupt Clear Pending Register (ICPR)
Interrupt Set Pending Register (ISPR)
Interrupt Priority Register (IPR)

Table 3.2: NVIC's registers

Function	Description
NVIC_DisableIRQ	Enable interrupts
NVIC_EnableIRQ	Disable interrupts
NVIC_ClearPendingIRQ	Clears the status of IRQ pending
NVIC_GetEnableIRQ	Returns if IRQ is enabele or not
NVIC_GetPendingIRQ	Returns the status of IRQ pending
NVIC_SetPriority	Sets the priority of IRQ

Table 3.3: List of used Nested Vector Interrupt Controller (NVIC) registers

is implemented using *BusRead* function and *BusWrite* functions. These functions allow the registers of the peripherals' models to be accessed. Each transaction that occurs to a peripheral will be either read or write and based on the transaction type the functions *BusRead* or *BusWrite* will be called. Both functions use switch expression based on the register's address to perform read, write, and other required changes. The template of the models can be found in appendix B.1.

3.4.1 Nested Vector Interrupt Controller (NVIC) model

The implementation of NVIC model has been added to the CPU model by defining its registers and functions as part of the CPU model. Only the registers used by the running tests are implemented.

The registers in table 3.2 were represented by a struct inside the CPU constructor. The functionalities of the Nested Vector Interrupt Controller present as CPU methods and table **??** list the NVIC functions that have been implemented.

The interrupt lines are modeled using *sc_signal* to connecting the interrupt output from the peripherals models to the CPU model. the interrupt output at the peripherals is of type *sc_out* while at CPU is defined of type *sc_in*. All different interrupt signals are binded in the *TOP* model to create the Interrupt request line (IRQ) for different peripherals.

We have used *std::function* type as pointer to interrupt handler routines(interrupt vector). This type can be considered as a safer version of a function pointer and can reference any type of callable target. A vector type *cb_arg_t* was made to store the addresses of all interrupt vectors. Type *cb_arg_t* is defined as follows:

```
    struct cb_arg_t {
    //the callback - takes a uint32_t input.
    std::function<void(void)> cb;
    //value to return with the callback.
    uint32_t arg;
};
    std::vector<cb_arg_t> callbacks_;
```

New interrupt vectors can be added to interrupt vector table bu passing a pointer to the interrupt handler and an integer representing the IRQ number. The code written to implement the NVIC can be found in appendix B.2.

For now, the interrupt handlers are predefined as simple print commands. While the current NVIC model can detect interrupt requests from other peripherals models. It is still can not execute complex interrupt handlers. This could be further developed but for our test which uses NVIC to enable or disable some interrupt lines without actually using interrupts to trigger other actions.

3.4.2 CLOCK peripheral's model

The CLOCK model has been implemented as a separate peripheral unlike the NVIC which was part of the CPU. The model follows the peripherals' template structure where it has the following registers:

The description of the registers in table 3.4 can be found in section 2.5.3.

When starting the platform, the initialization thread of the CLOCK model runs a wait command and triggers the internal High Frequency Clock (HFCLK) to simulates the startup delay in the peripheral til the High Frequency Clock signal is stable and ready to be used by the system. After the initialization of the CLOCK peripheral, there are three main functionalities the CLOCK provides:

CLOCK model registers	Description
nrfclockHFCLKSTART	Start HFCLK crystal oscillator
nrfclockHFCLKSTOP	Stop HFCLK crystal oscillator
nrfclockLFCLKSTART	Start LFCLK source
nrfclockLFCLKSTOP	Stop LFCLK source
nrfclockCAL	Start calibration of LFRC oscillator
nrfclockCTSTART	Start calibration timer
nrfclockCTSTOP	Stop calibration timer
nrfclockHFCLKSTARTED	HFCLK oscillator started
nrfclockLFCLKSTARTED	LFCLK oscillator started
nrfclockDONE	Calibration of LFCLK RC oscillator complete event
nrfclockCTTO	Calibration timer timeout
nrfclockINTENSET	Enable interrupt
nrfclockINTENCLR	Disable interrupt
nrfclockHFCLKSTAT	HFCLK status
nrfclockLFCLKSTAT	LFCLK status
nrfclockLFCLKSRC	Clock source for the LFCLK
nrfclockHFXODEBOUNCE	Sets the debounce time to start the external HFCLK oscillator
nrfclockCTIV	Calibration timer interval

Table 3.4: Registers of the CLOCK model

- High Frequency Clock (HFCLK) control.
- Low Frequency Clock (LFCLK) control.
- Low Frequency Clock (LFCLK) calibration.

High Frequency Clock (HFCLK) control

As mentioned when the CLOCK powers up, it starts with the internal High Frequency Clock (HFCLK). After that the external High Frequency Clock could be used changing different sources for the High Frequency Clock can be performed.

The figure 3.5 shows how to start the external high frequency oscillator. It starts by triggering the *TASKS_HFCLKSTART* which can be done by writing 1 to register *nrfclockHFCLKSTART*. After that, the model checks the status of the High Frequency Clock (HFCLK) if it is already running, If so, it returns a message reporting that the external oscillator is already in use. If it is not running, the status register will be updated to change the state of the external oscillator to "Running" and the source of High Frequency Clock (HFCLK) to the external oscillator. followed by a simulation of the start-up time needed for the external oscillator to generate a stable clock. The waiting period depends on the sum of the oscillator

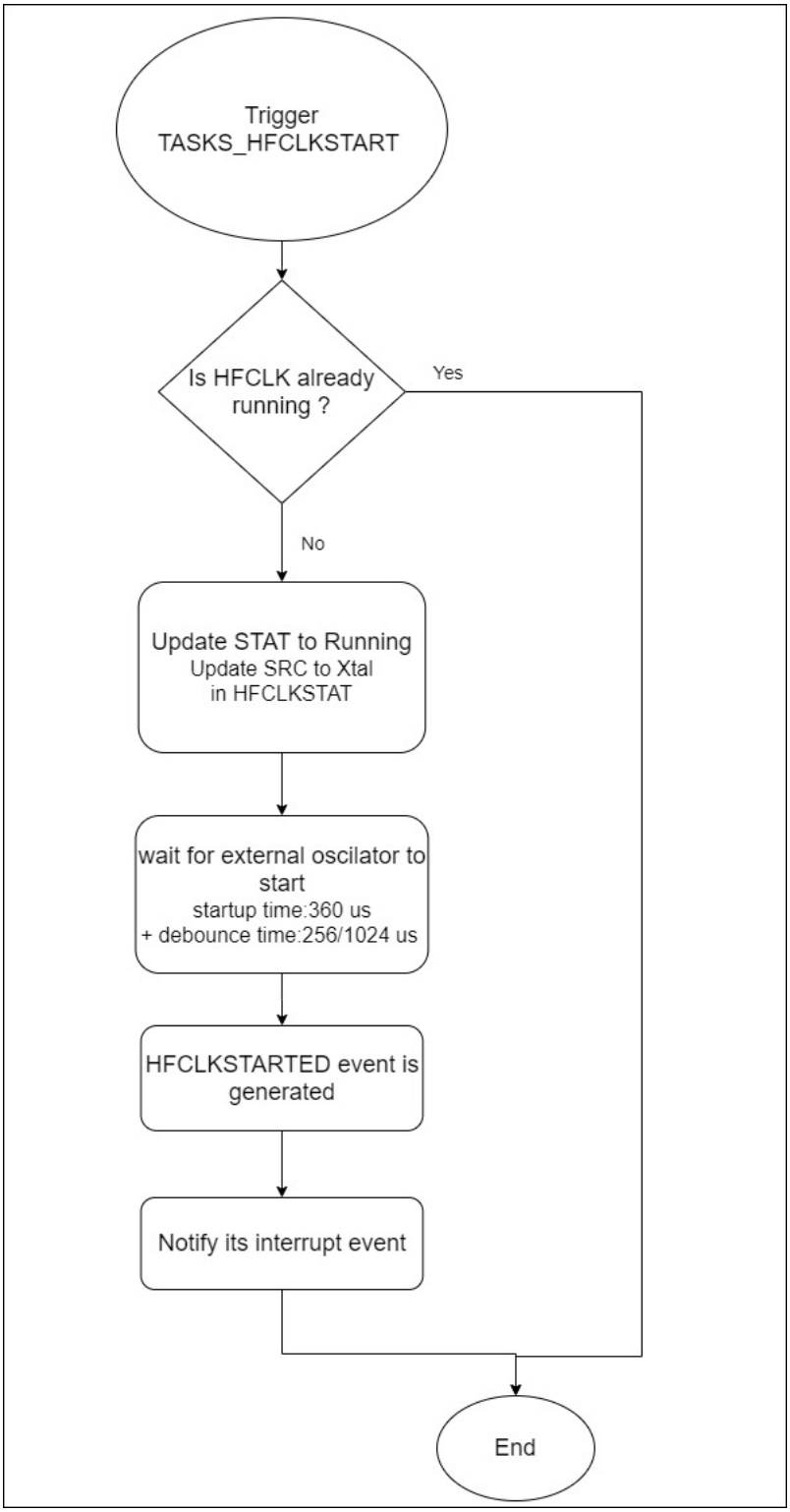

Figure 3.5: The steps to start the external high frequency oscillator

startup time (which is 360us) and the debouncing time. The debouncing time is either 256us or 1024us depending on the value of *nrfclockHFXODEBOUNCE* register. After the waiting period has passed the model will set the value of *nrfclockHFCLKSTARTED* and notify the

interrupt event of starting external High Frequency Clock which could trigger an interrupt of its conditions are met.

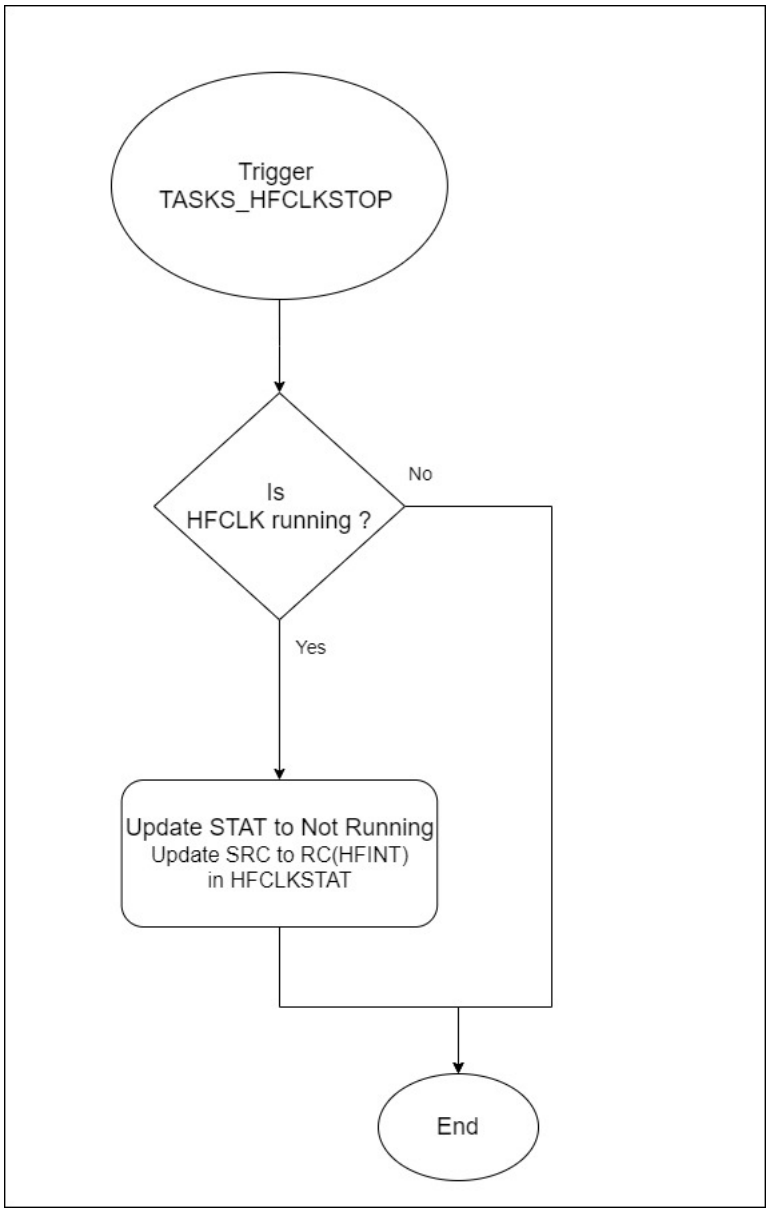

Figure 3.6: Steps to stop the external High frequency oscillator

Alternatively, to stop the external High Frequency Clock (HFCLK) the tasks *TASKS_HFCLKSTOP* has to be triggered by writing 1 to register *nrfclockHFCLKSTOP*. It model first checks the status of the external oscillator, if it isn't running, it displays a message stating that. If the external oscillator is running it will update the status register *nrfclockHFCLKSTAT* to indicate changing the state to "Not Running" and the source to internal RC oscillator. And the figure 3.6 illustrates these steps.

Value	Source
00	32.768 kHz RC oscillator (LFRC)
01	32.768 kHz crystal oscillator (LFXO)
10	32.768 kHz synthesized from HFCLK (LFSYNT)

Table 3.5: List of Low Frequency Clock (LFCLK) sources

Low Frequency Clock (LFCLK) control

Unlike the High Frequency Clock (HFCLK) which starts when the CLOCK's model powers up, the Low Frequency Clock (LFCLK) needs to be explicitly started regardless of its source. To use the Low Frequency Clock (LFCLK) the source needs to be decided by writing the source value into register *nrfclockLFCLKSRC* like shown in table 3.5. If the source has not been chosen the default option is 32.768 kHz RC oscillator.

Figure 3.7 shows the steps to start the Low Frequency Clock (LFCLK), it is started when the task *TASKS_LFCLKSTART* is triggered by writing 1 to register *nrfclockLFCLKSTART*. The model checks the state register of Low Frequency Clock (LFCLK), if it is already running it prints a message stating that. While if it was not running, it then checks which oscillator is the source for the Low Frequency Clock, in the case of the external oscillator as the source, the model will delay the simulation to imitate the start-up time needed by the external oscillator. Then it updates the state of LFCLK to "Running", generate *LFCLKSTARTED* event and he notifying interrupt event for starting starting LFCLK. This event might trigger an interrupt to the CPU if its conditions are met.

The stopping process of the LFCLK is illustrated in figure 3.8 the task *TASKS_LFCLKSTOP* need to be triggered by writing 1 to the register *nrfclockLFCLKSTOP*. The model then checks the state of LFCLK, if it is "Not Running" it prints a message indicating that. While if the state is "Running" the state is changed to "Not Running".

Low Frequency Clock (LFCLK) calibration

As mentioned before the frequency of the RC internal oscillator might be affected by

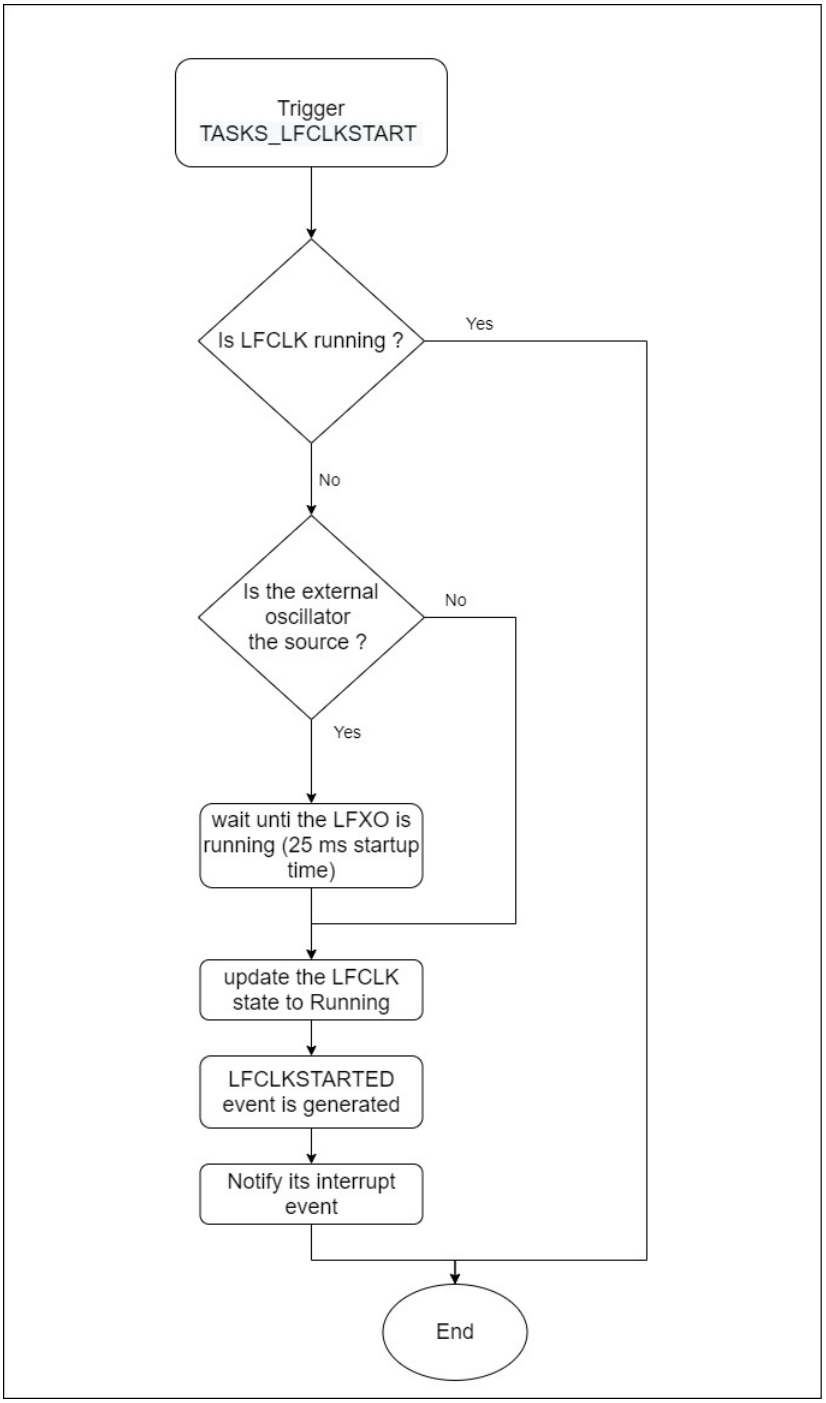

Figure 3.7: Steps to start the LFCLK

variation in temperature. And it can be calibrated to compensate for these variations. The calibration uses the external high-frequency oscillator to calibrate the RC oscillator, so before performing the calibration the Low Frequency Clock (LFCLK) needs to be running with RC oscillator as its source as well as the high-frequency oscillator. Before running the calibration process the calibration timer needs to run first. To run the calibration timer the register *CTIV*

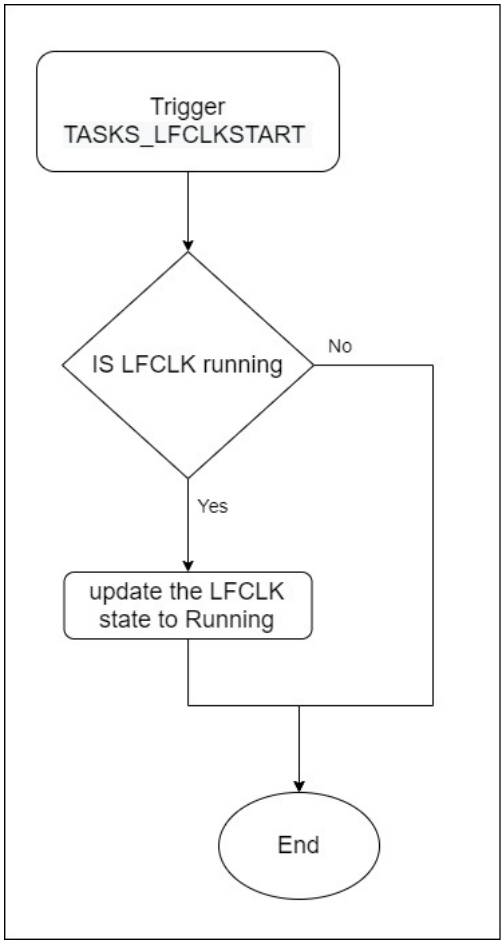

Figure 3.8: Steps to stop the LFCLK

should configure first with the calibration timer interval. The start of the calibration timer is triggered by writing to the *TASKS_CTSTART* register.
The calibration timer keeps decrementing the value of *CTIV* until it reaches zero or until *TASKS_CTSTOP* is triggered. When any of the previous occurs the calibration timer stops and the event *EVENTS_CTTO* is triggered.

After that the calibration process can be triggered via *TASKS_CAL* by writing 1 to register *nrfclockCAL*. At the end of the calibration, the event *EVENTS_DONE* is generated indicating the completion of the calibration process. The points and steps mentioned before are illustrated in figure 3.9.

Interrupt configuration

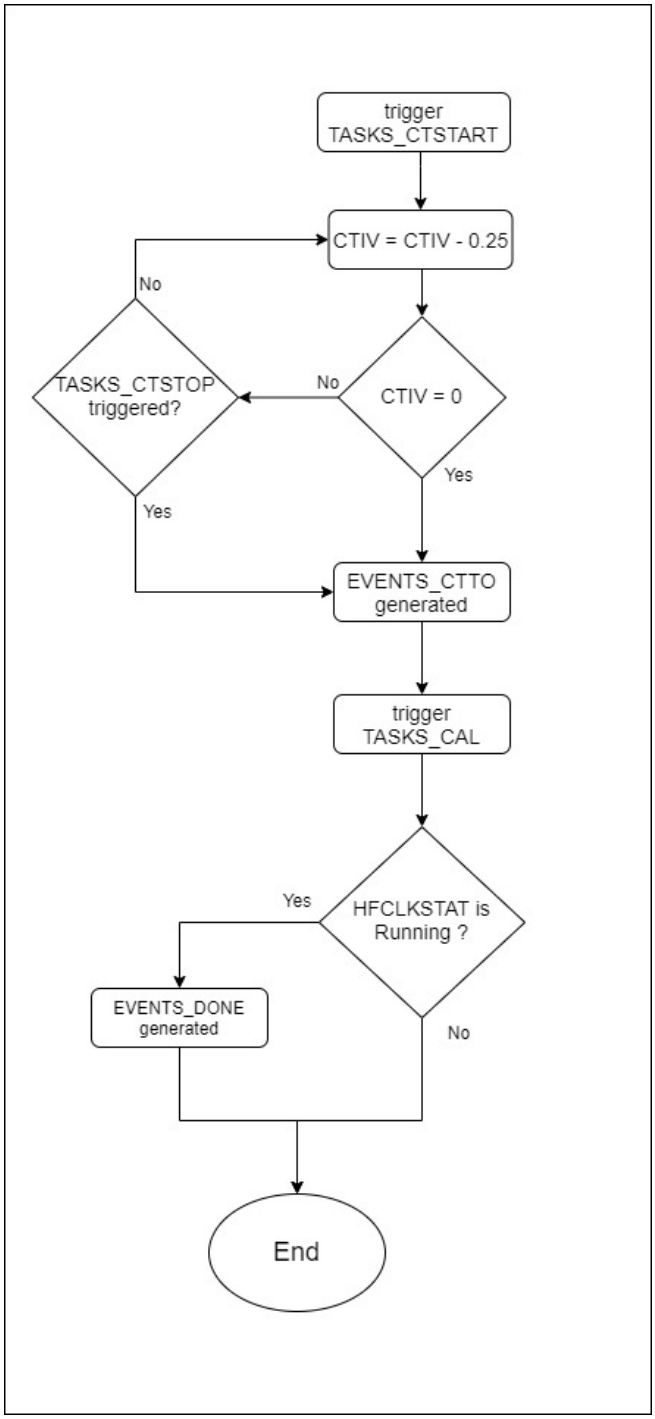

Figure 3.9: Low Frequency Clock (LFCLK) calibration flowchart

The interrupts firing mechanism is modeled by a thread in the model waiting for interrupt events to be notified. When an interrupt SystemC event is notified the thread check if the interrupt generated this event is enabled. The Interrupt request lines of a modeled are enabled by configuring *nrfclockINTENSET* register. The used code to implement the clock mode can be found in section B.3

GPIO registers	Description
nrfgpioOUT	GPIO pins value
nrfgpioOUTSET	set individual bits in GPIO port
nrfgpioOUTCLR	clear individual bits in GPIO port

Table 3.6: List of General purpose input/output registers

3.4.3 General purpose input/output (GPIO) module model

This model follows the template structure. The table 3.6 contains the registers used in GPIO model, They are explained in section 2.5.8.

The module executes two functions, setting an individual bit of a specific port and clearing an individual bit of a specific port. To set a bit in the GPIO port, write '1' to the corresponding bit in *OUTSET* register. Alternatively, to clear a bit in the GPIO port, write '1' to the corresponding bit in *OUTCLR* register. Writing '0' to any of the previous registers will not affect the GPIO port. The drivers of the pins can be checked by reading *OUT* register. The code used to implement the General purpose input/output model can be found in section B.4.

3.5 Models verification

Before moving on to running tests on the virtual platform the models' behavior needs to be verified that is as expected. Running a comprehensive verification of the models should reduce the debugging process later on when running tests on the platform. To verify the system, first, we have individually verified the models' behavior before moving on verifying the system behavior as a whole. In this part only the peripherals modeled by this thesis will be verified which are General purpose input/output (GPIO), NVIC and CLOCK models.

The verification methodology used in this work is simulation-based verification. There are some papers which discusses formal verification methodology for *SystemC* models like [6] and [12]. But the formal approach will not be used in this work since the used models are not extremely complex and formal verification of SystemC models is outside the scope of this work.

3.5.1 General purpose input/output (GPIO) model behaviour verification

As the main functionality of General purpose input/output model is to set and clear bits of GPIO ports. Table 3.7 lists the verification conditions and their corresponding tests.

Verification condition	**Verification test**
Correctly set and clear individual bits	individual bit set/clear()
Correctly set and clear all bits	accumulative set/clear()

Table 3.7: General purpose input/output verification conditions and test

1. Individual bit set/clear

   The correct bit in *OUT* register is being affected by changes in *OUTSET* or *OUTCLR* registers. This is done by setting the each bit in *OUTSET* individually and read the value of *OUT* register. All 32 bits in *OUTSET* need to set the corresponding bits in *OUT* register. The same is done for clear. All 32 bits in *OUTCLR* need to clear the corresponding bits in *OUT*.

2. Accumulative set/clear

   Setting a bit in *OUTSET* or *OUTCLR* registers will only affect the corresponding bit in *OUT* register. While the rest of the bits keep their values. This is done by setting the bits in *OUTSET* register one by one and then check the new value of *OUT* register. The new change should not affect the previous state of other bits. The same is done for *OUTCLR* register. Setting the bits in it one by one and check the value of *OUT* register. Where the previous state of other bits should not change.

The code of the verification can be found in appendix B.5.2.

3.5.2 CLOCK model behaviour verification

The CLOCK model has three main behaviours that has been verified.

- High Frequency Clock (HFCLK) operation:

  It would verify the correctness of High Frequency Clock (HFCLK) controller behavior. It includes initialization and shutting down sequence illustrated in figures 3.5 and 3.6.

- Low Frequency Clock (LFCLK) operation:

  It would also verify the correctness of the Low Frequency Clock (LFCLK) controller. It covers the sequence illustrated in figures 3.7 and 3.8.

- Low Frequency Clock (LFCLK) calibration:

  It verifies the correctness of the LFCLK calibration process. The verification plan runs the sequence illustrated in figure 3.9.

The following table illustrates the verification conditions covered by each verification test.

Verification condition	Verification test
Correctly start HFCLK	HFCLK_verification()
Correctly update HFCLK status	
Correctly generate HFCLKSTARTED event	
Correctly stop HFCLK	
Correctly start LFCLK	LFCLK_verification()
Correctly update LFCLK source	
Correctly generate LFCLKSTARTED event	
Correctly stop LFCLK	
Correctly start calibration timer	LFCLK_calibration_verification()
Correctly decrements calibration timer interval	
Correctly generate DONE event	
Correctly stop calibration timer	

Table 3.8: CLOCK verification conditions and test

The verification of the previously mentioned behaviors is done by running the software sequence for each one of these functionalities and compare the results of the affected registers with the expected value. The code for these verification plans for different functionalities can be found in appendix B.5.3.

3.5.3 NVIC Verification

The Nested Vector Interrupt Controller is implemented as part of the CPU. It has five functions that had been verified.

These functions are:

- *NVIC_EnableIRQ*:

  This function takes the Interrupt request line number and set the corresponding bit in Interrupt Set Enable Register (ISER). It has been verified by setting all 30 available IRQ

of register ISER. Then compare the value of NVIC interrupt enable register (ISER) with the expected value.

- *NVIC_DisableIRQ*:

  This function takes the Interrupt request line number and sets the corresponding bit in Interrupt Clear Enable Register (ICER). It has been verified by setting all 30 available IRQ of ICER register . Then compare the value of NVIC Interrupt Clear Enable Register (ICER) with the expected value.

- *NVIC_ClearPendingIRQ*:

  This function takes the Interrupt request line number and sets the corresponding bit in Interrupt Clear Pending Register. It has been verified by setting all 30 available IRQ of ICPR register. Then compare the value of NVIC interrupt disable register (ICPR) with the expected value.

- *NVIC_GetEnableIRQ*:

  This method returns the bit value to the corresponding IRQ in ISER register It has been verified by setting all 30 available IRQ of ISER register one by one. Then compare the returned value from *NVIC_GetEnableIRQ* method.

- *NVIC_GetPendingIRQ*:

  This method returns the bit value to the corresponding IRQ in Interrupt Set Pending Registerregister It has been verified by setting all 30 available IRQ of ISPR register one by one. Then compare the returned value from *NVIC_GetPendingIRQ* method.

The following table illustrates the verification conditions covered by each verification test.

Verification condition	**Verification test**
Correctly use NVIC_EnableIRQ method	NVIC_Verification_EnableIRQ()
Correctly use NVIC_DisableleIRQ method	NVIC_Verification_DisableIRQ()
Correctly use NVIC_ClearPendingIRQ method	NVIC_Verification_ClearPendingIRQ()
Correctly use NVIC_GetEnable method	NVIC_Verification_GetEnable()
Correctly use NVIC_GetPending method	NVIC_Verification_GetPending()

Table 3.9: NVIC verification conditions and test

The code used for the verification can be found in section B.5.1.

3.5.4 System Verification

After verifying the correctness of the models' behaviors separately we need to verify their operation together. Ensuring that the system components will not unexpectedly affect each other.

The test that has been implemented included the use of CLOCK, GPIO, and NVIC models and figure 3.10 shows the steps of the verification plan. The system would be verified by starting the Low Frequency Clock in the CLOCK model and enable its interrupt. This will fire an interrupt signal that would be detected by the NVIC. The NVIC will in turn execute an interrupt routine that would set some bits in the GPIO model and then compare the contents of GPIO OUT register with the expected results. The code used to verify the system is in appendix B.5.4.

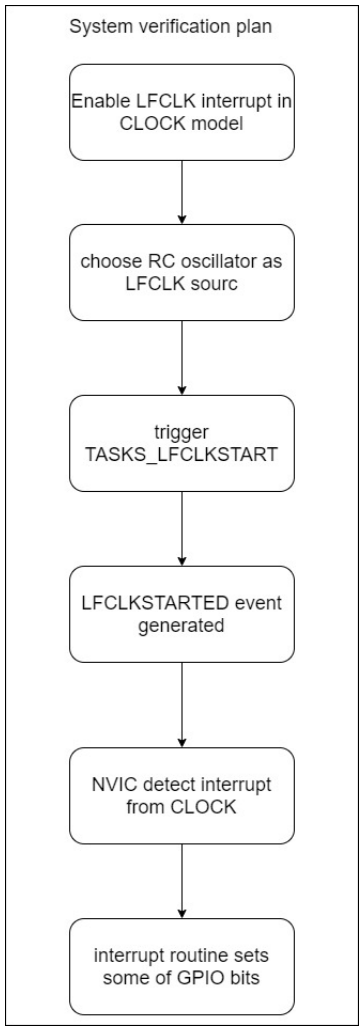

Figure 3.10: system verification plan

3.6 Running tests on the virtual platform

After developing the peripherals models and verifying them, the next step is testing the platform by running the software tests. We started by running the base test discussed earlier (section 3.2). To run the tests on the virtual platform we have modified the tests to be compatible with SystemC models. The test consist mainly of two functions, initialization function *mpsl_init* and un-initialization function *mpsl_uninit* and this thesis worked on modifying and running the function *mpsl_init*. Below are some of the modifications that we have performed on the test files to make them compatible with the platform:

- Use some of the test files unchanged:
  Some of the files which define variables used by test functions or declaration/implement functions that do not perform hardware accesses. These files kept unchanged and used by the virtual platform.

- Merging the functions:
  We have gathered all functions definitions needed by *mpsl_init* in a single `C++` file. These are the functions the perform hardware accesses.

- Passing CPU pointer:
  A pointer to a CPU object is passed to all functions that perform hardware accesses. The CPU pointer allows the function to send transactions to different models in the platform.

The wall-time of the software execution is being measured during the test runs. It starts recording the time when the top module of the virtual prototype is constructed, while the timer stopping point is when the simulation is over. This time, the difference between the start and endpoints, is used in the performance comparison later on.

3.6.1 Running the base test

After successfully running the initialization and un-initialization functions *mpsl_init* and *mpsl_uninut*. We moved on to the base test which checks the correctness of the initialization and un-initialization process. It calls *mpsl_init* function and asserts its return value is the same as expected. Then it checks the values of the different register of NVIC by asserting their

values of different IRQ to the expected values. It calls *mpsl_uninit* function and checks the values of the NVIC registers similarly to before. The test code can be found in appendix B.6.

3.6.2 CLOCK peripheral tests

After running the base test on the platform, the next step was to run more tests that provide more data to compare and more test coverage. We have run three tests that test different scenarios of CLOCK callback. The function *mpsl_clock_hfclk_request()* grants access to HFCLK by the tests, where only one device at a time can access the HFCLK. While the function *mpsl_clock_hfclk_release()* release the HFCLK making it available for other processes to access it. Both functions provide synchronization mechanism to control the software access to the HFCLK These tests are the following:

1. **One-shot CLOCK callback**:

   Tests that callback given to *mpsl_clock_hfclk_request()* is executed when HFCLK is enabled.

   Test procedure:

   - Check that *POWER_CLOCK* IRQ is not pending
   - Check that HFCLK is not running
   - Call *mpsl_clock_hfclk_request()*
   - Wait till HFCLK is running
   - Check that callback given to *mpsl_clock_hfclk_request()* was executed once.

2. **No CLOCK callback is called when HFCLK started by protocol**:

   Tests that clock is started when *mpsl_clock_hfclk_protocol_request()* is called, and that the user supplied callback is not called. Test procedure:

   - Start clock using *mpsl_clock_hfclk_request()*, wait unti it is running and check that the callback is called once.
   - Stop the clock using *mpsl_clock_hfclk_release()*, wait until it is stopped
   - Call *mpsl_clock_hfclk_protocol_request()*
   - Wait until clock is running

- Call *mpsl_clock_hfclk_protocol_release()*
- Check that call count for the user supplied callback is still 1

3. **CLOCK callback is called when HFCLK is already triggered**:

   Tests that callback given to *mpsl_clock_hfclk_request()* is executed when HFCLK is enabled if HFCLK clock start is triggered before the call. Test procedure:

   - Check that HFCLK is not running
   - Trigger HFCLK using *hal_clock_hfclk_start()*
   - Call *mpsl_clock_hfclk_request()*
   - Wait untill HFCLK is running
   - Check that callback given to *mpsl_clock_hfclk_request()* was executed once

Chapter 4

Results

4.1 Results of Models verification

As the verification plans of each model have been presented in the previous chapter, here we would show their outcomes for the different models. The simulation log files are presented as the results of verification plans. The logs capture and display all the transactions carried out by the simulation.

4.1.1 CLOCK model verification

The verification plans of the CLOCK peripheral verified the operation of its three main functionalities:

- **High Frequency Clock controller verification plan**:

  The figure 4.1 shows the results of verification of HFCLK start when triggering *TASKS_HFCLKSTART*. The figure displays information messages describing each transaction and its implications on the system behavior. While figure 4.2 shows the results of the verification when *TASKS_HFCLKSTOP* is triggered.

  Table 4.1 shows the passing verification conditions covered by HFCLK controller verification test.

```
2 us: Info: CPU: CPU is starting
37 us: Info: NVIC: Interrupt vectors table is ready
47 us: Info: CPU: Initiated read transaction to 4000040c
82 us: Info: nrf_clock: Received expected read request from 40c
82 us: Info: nrf_clock: HFCLK source is 64 MHz internal oscillator (HFINT)
82 us: Info: nrf_clock: HFXO state is NotRunning
97 us: Info: CPU: Initiated read transaction to 40000100
132 us: Info: nrf_clock: Received expected read request from 100
132 us: Info: nrf_clock:  EVENTS_HFCLKSTARTED has not been generated yet
147 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000000
182 us: Info: nrf_clock: Received expected write request to 0
182 us: Info: nrf_clock: HFXO will start
182 us: Info: nrf_clock: switching to HFXO
182 us: Info: nrf_clock: waiting for 616        till HFXO is running
197 us: Info: CPU: Initiated read transaction to 4000040c
227 us: Info: nrf_clock: Received expected read request from 40c
227 us: Info: nrf_clock: HFCLK source is 64 MHz crystal oscillator (HFXO)
227 us: Info: nrf_clock: HFXO state is Running
242 us: Info: CPU: Initiated read transaction to 40000100
272 us: Info: nrf_clock: Received expected read request from 100
272 us: Info: nrf_clock:  EVENTS_HFCLKSTARTED has been generated
```

Figure 4.1: HFCLK controller start Verification

```
287 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000004
317 us: Info: nrf_clock: Received expected write request to 4
317 us: Info: nrf_clock: HFXO will stop
332 us: Info: CPU: Initiated read transaction to 4000040c
362 us: Info: nrf_clock: Received expected read request from 40c
362 us: Info: nrf_clock: HFCLK source is 64 MHz internal oscillator (HFINT)
362 us: Info: nrf_clock: HFXO state is NotRunning
377 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000004
407 us: Info: nrf_clock: Received expected write request to 4
407 us: Info: nrf_clock: HFXO is already stopped
422 us: Info: CPU: Initiated read transaction to 4000040c
452 us: Info: nrf_clock: Received expected read request from 40c
452 us: Info: nrf_clock: HFCLK source is 64 MHz internal oscillator (HFINT)
452 us: Info: nrf_clock: HFXO state is NotRunning
457 us: Info: CPU: main finished
```

Figure 4.2: HFCLK controller stop Verification

verification test	verification condition	Pass/Fail
HFCLK_verification	Correctly start HFCLK	Pass
	Correctly update HFCLK status	Pass
	Correctly generate HFCLKSTARTED event	Pass
	Corrctly stop HFCLK	Pass

Table 4.1: Verification results HFCLK controller

- **Low Frequency Clock controller verification**:

Figure 4.3 highlights the behaviour of the peripheral when *TASKS_LFCLKSTART* is triggered. While the figure 4.4 highlights part showing the behaviour when *TASKS_LFCLKSTOP* is triggered. And the figure 4.5 illustrates the verification results when changing the source of the LFCLK to the external oscillator.

```
2 us: Info: CPU: CPU is starting
37 us: Info: NVIC: Interrupt vectors table is ready
47 us: Info: CPU: Initiated read transaction to 40000418
82 us: Info: nrf_clock: Received expected read request from 418
82 us: Info: nrf_clock: The LFCLK source is 32.768 kHz RC oscillator
82 us: Info: nrf_clock: LFCLK state is NotRunning
97 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000008
132 us: Info: nrf_clock: Received expected write request to 8
132 us: Info: nrf_clock: LFCLK will start
132 us: Info: nrf_clock: LFCLKSTARTED interrupt has been fired
147 us: Info: CPU: Initiated read transaction to 40000418
182 us: Info: nrf_clock: Received expected read request from 418
182 us: Info: nrf_clock: The LFCLK source is 32.768 kHz RC oscillator
182 us: Info: nrf_clock: LFCLK state is Running
```

Figure 4.3: LFCLK controller start Verification

```
197 us: Info: CPU: Initiated write transaction of 0x1 to 0x4000000c
232 us: Info: nrf_clock: Received expected write request to c
232 us: Info: nrf_clock: LFCLK will stop
247 us: Info: CPU: Initiated read transaction to 40000418
282 us: Info: nrf_clock: Received expected read request from 418
282 us: Info: nrf_clock: The LFCLK source is 32.768 kHz RC oscillator
282 us: Info: nrf_clock: LFCLK state is NotRunning
```

Figure 4.4: LFCLK controller stop Verification

```
297 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000518
332 us: Info: nrf_clock: Received expected write request to 518
332 us: Info: nrf_clock: LFCLKSRC has been updated
332 us: Info: nrf_clock: Normal XTAL operation
347 us: Info: CPU: Initiated read transaction to 40000418
382 us: Info: nrf_clock: Received expected read request from 418
382 us: Info: nrf_clock: The LFCLK source is 32.768 kHz crystal oscillator
382 us: Info: nrf_clock: LFCLK state is NotRunning
397 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000008
432 us: Info: nrf_clock: Received expected write request to 8
432 us: Info: nrf_clock: LFCLK will start
432 us: Info: nrf_clock: LFRCO is LFCLK source
3002 us: Info: nrf_clock: initialization routine is over
25432 us: Info: nrf_clock: wait for 25ms before switching to LFXO
25432 us: Info: nrf_clock: switching to LFXO
25432 us: Info: nrf_clock: LFCLKSTARTED interrupt has been fired
25447 us: Info: CPU: Initiated read transaction to 40000418
25482 us: Info: nrf_clock: Received expected read request from 418
25482 us: Info: nrf_clock: The LFCLK source is 32.768 kHz crystal oscillator
25482 us: Info: nrf_clock: LFCLK state is Running
25497 us: Info: CPU: Initiated read transaction to 40000104
25532 us: Info: nrf_clock: Received expected read request from 104
25532 us: Info: nrf_clock:  EVENTS_LFCLKSTARTED has been generated
25537 us: Info: CPU: main finished
```

Figure 4.5: external oscillator start Verification

Table 4.2 shows the results of the verification test of the LFCLK controller.

verification test	verification condition	Pass/Fail
LFCLK_verification	Correctly start LFCLK	Pass
	Correctly update LFCLK source	Pass
	Correctly generate LFCLKSTARTED event	Pass
	Correctly stop LFCLK	Pass

Table 4.2: Verification results LFCLK controller

- **Low Frequency Clock calibration verification**:

  Figure 4.6 highlights the calibration timer operation in the verification plan. The calibration timer starts operation when *TASKS_CTSTART* is triggered. While figure 4.7 highlights the calibration process when *TASKS_CAL* is triggered.

```
2 us: Info: CPU: CPU is starting
37 us: Info: NVIC: Interrupt vectors table is ready
47 us: Info: CPU: Initiated write transaction of 0x0 to 0x40000100
82 us: Info: nrf_clock: Received expected write request to 100
82 us: Info: nrf_clock: HFCLKSTARTED event has been updated to   0
97 us: Info: CPU: Initiated write transaction of 0x10 to 0x40000538
132 us: Info: nrf_clock: Received expected write request to 538
132 us: Info: nrf_clock: CTIV register has been updated to   4
147 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000014
182 us: Info: nrf_clock: Received expected write request to 14
182 us: Info: nrf_clock: Calibration timer is starting
182 us: Info: nrf_clock: calibration timer is starting
197 us: Info: CPU: Initiated read transaction to 40000538
207 us: Info: nrf_clock: /* CTIV */ 3
232 us: Info: nrf_clock: /* CTIV */ 2
232 us: Info: nrf_clock: Received expected read request from 538
232 us: Info: nrf_clock: Calibration timer interval is  2 seconds
247 us: Info: CPU: Initiated read transaction to 40000538
257 us: Info: nrf_clock: /* CTIV */ 1
282 us: Info: nrf_clock: /* CTIV */ 0
282 us: Info: nrf_clock: calibration timer has finished
282 us: Info: nrf_clock: /* CTTO is */ 1
282 us: Info: nrf_clock: Received expected read request from 538
282 us: Info: nrf_clock: Calibration timer interval is  0 seconds
297 us: Info: CPU: Initiated read transaction to 40000110
332 us: Info: nrf_clock: Received expected read request from 110
332 us: Info: nrf_clock: CTTO event is   1
```

Figure 4.6: verification of calibration timer operation

```
347 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000000
382 us: Info: nrf_clock: Received expected write request to 0
382 us: Info: nrf_clock: HFXO will start
382 us: Info: nrf_clock: switching to HFXO
382 us: Info: nrf_clock: waiting for 616        till HFXO is running
397 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000010
427 us: Info: nrf_clock: Received expected write request to 10
427 us: Info: nrf_clock: Calibration process is starting
442 us: Info: CPU: Initiated read transaction to 4000010c
472 us: Info: nrf_clock: Received expected read request from 10c
472 us: Info: nrf_clock: EVENT DONE register is:  1
477 us: Info: CPU: main finished
1002 us: Info: nrf_clock: HFXO is running
1002 us: Info: nrf_clock: HFCLKSTARTED interrupt has been fired
```

Figure 4.7: verification of calibration process operation

Table 4.3 shows the verification results of LFCLK calibration verification test.

4.1.2 GPIO model verification

The verification of General purpose input/output (GPIO) model covers the control of the bits of the peripheral's port.

verification test	verification condition	Pass/Fail
LFCLK_calibration_verification	Correctly start calibration timer	Pass
	Correctly decrements calibration timer interval	Pass
	Correctly generate DONE event	Pass
	Correctly stop calibration timer	Pass

Table 4.3: Verification results of LFCLK's oscillator calibration

Figure 4.8 shows the results of the individual verification of the bits. The individual verification performs set and clear operation on the individual bits of the ports. While the figure 4.9 verifies that operating on a bit will not affect the rest. It performs set operation on the ports' bits one by one and when all bits are set it performs the clear operation similarly. The figures show only the final results since the same operation is performed for all bits in the port. Table 4.4 shows the results of GPIO verification test.

```
5932 us: Info: nrf_gpio: Received expected write request to 4
5947 us: Info: CPU: Initiated read transaction to 50000000
5997 us: Info: CPU: Initiated write transaction of 0x20000000 to 0x50000008
6032 us: Info: nrf_gpio: Received expected write request to 8
6047 us: Info: CPU: Initiated read transaction to 50000000
6097 us: Info: CPU: Initiated write transaction of 0x40000000 to 0x50000004
6132 us: Info: nrf_gpio: Received expected write request to 4
6147 us: Info: CPU: Initiated read transaction to 50000000
6197 us: Info: CPU: Initiated write transaction of 0x40000000 to 0x50000008
6232 us: Info: nrf_gpio: Received expected write request to 8
6247 us: Info: CPU: Initiated read transaction to 50000000
6297 us: Info: CPU: Initiated write transaction of 0x80000000 to 0x50000004
6332 us: Info: nrf_gpio: Received expected write request to 4
6347 us: Info: CPU: Initiated read transaction to 50000000
6397 us: Info: CPU: Initiated write transaction of 0x80000000 to 0x50000008
6432 us: Info: nrf_gpio: Received expected write request to 8
6447 us: Info: CPU: Initiated read transaction to 50000000
6487 us: Info: GPIO verification: Indvidual bits SETOUT verification PASSED
6487 us: Info: GPIO verification: Indvidual bits CLROUT verification PASSED
6487 us: Info: CPU: main finished
```

Figure 4.8: verification of GPIO bits individually

```
6132 us: Info: nrf_gpio: Received expected write request to 8
6147 us: Info: CPU: Initiated read transaction to 50000000
6197 us: Info: CPU: Initiated write transaction of 0x3fffffff to 0x50000008
6232 us: Info: nrf_gpio: Received expected write request to 8
6247 us: Info: CPU: Initiated read transaction to 50000000
6297 us: Info: CPU: Initiated write transaction of 0x7fffffff to 0x50000008
6332 us: Info: nrf_gpio: Received expected write request to 8
6347 us: Info: CPU: Initiated read transaction to 50000000
6397 us: Info: CPU: Initiated write transaction of 0xffffffff to 0x50000008
6432 us: Info: nrf_gpio: Received expected write request to 8
6447 us: Info: CPU: Initiated read transaction to 50000000
6487 us: Info: GPIO verification: accemulative SETOUT verification PASSED
6487 us: Info: GPIO verification: accemulative CLROUT verification PASSED
6487 us: Info: CPU: main finished
```

Figure 4.9: verification of GPIO bits cumulatively

Verification condition	Verification test	Pass/Fail
Correctly set and clear individual bits	individual bit set/clear()	Pass
Correctly set and clear all bits	accumulative set/clear()	Pass

Table 4.4: General purpose input/output verification results

4.1.3 NVIC model verification

The description of the NVIC can be found in section 3.5.3. NVIC verification covers the operation of its functions. The verification of NVIC tested all IRQ lines, but the results will show only the final since we repeat the same operation for the different IRQs. The results of NVIC verification are as follow:

- ***NVIC_EnableIRQ* function verification**:

```
37 us: Info: NVIC: IRQ 26 has been enabled
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 7ffffff
37 us: Info: NVIC: IRQ 27 has been enabled
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read fffffff
37 us: Info: NVIC: IRQ 28 has been enabled
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 1fffffff
37 us: Info: NVIC: IRQ 29 has been enabled
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 3fffffff
37 us: Info: NVIC Verification: EnableIRQ: verification PASSED
37 us: Info: CPU: main finished
```

Figure 4.10: *EnableIRQ* function verification

- ***NVIC_DisableIRQ* function verification**:

```
37 us: Info: NVIC: IRQ 26 has been disabled
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 7ffffff
37 us: Info: NVIC: IRQ 27 has been disabled
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read fffffff
37 us: Info: NVIC: IRQ 28 has been disabled
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 1fffffff
37 us: Info: NVIC: IRQ 29 has been disabled
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 3fffffff
37 us: Info: NVIC Verification: DiableIRQ: Verification PASSED
37 us: Info: CPU: main finished
```

Figure 4.11: *DisableIRQ* function verification

- ***NVIC_ClearPendingIRQ* function verification**:

```
37 us: Info: NVIC: IRQ 26 pending has been cleared
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 7ffffff
37 us: Info: NVIC: IRQ 27 pending has been cleared
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read fffffff
37 us: Info: NVIC: IRQ 28 pending has been cleared
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 1fffffff
37 us: Info: NVIC: IRQ 29 pending has been cleared
37 us: Info: CPU: Received expected read request
37 us: Info: CPU: Read 3fffffff
37 us: Info: NVIC Verification: ClearPendingIRQ: Verification PASSED
37 us: Info: CPU: main finished
```

Figure 4.12: *ClearPendingIRQ* function verification

- ***NVIC_GetEnableIRQ* function verification**:

```
37 us: Info: NVIC: IRQ 23 has been enabled
37 us: Info: NVIC: GetEnable IRQ ( 23 ) =  1
37 us: Info: NVIC: GetEnable IRQ ( 24 ) =  0
37 us: Info: NVIC: IRQ 25 has been enabled
37 us: Info: NVIC: GetEnable IRQ ( 25 ) =  1
37 us: Info: NVIC: GetEnable IRQ ( 26 ) =  0
37 us: Info: NVIC: IRQ 27 has been enabled
37 us: Info: NVIC: GetEnable IRQ ( 27 ) =  1
37 us: Info: NVIC: GetEnable IRQ ( 28 ) =  0
37 us: Info: NVIC: IRQ 29 has been enabled
37 us: Info: NVIC: GetEnable IRQ ( 29 ) =  1
37 us: Info: NVIC: GetEnable Verification PASSED
37 us: Info: CPU: main finished
```

Figure 4.13: *GetEnableIRQ* function verification

- ***NVIC_GetPendingIRQ* function verification**:

```
37 us: Info: NVIC: GetPending IRQ ( 23 ) =  1
37 us: Info: NVIC: GetPending IRQ ( 24 ) =  0
37 us: Info: NVIC: GetPending IRQ ( 25 ) =  1
37 us: Info: NVIC: GetPending IRQ ( 26 ) =  0
37 us: Info: NVIC: GetPending IRQ ( 27 ) =  1
37 us: Info: NVIC: GetPending IRQ ( 28 ) =  0
37 us: Info: NVIC: GetPending IRQ ( 29 ) =  1
37 us: Info: NVIC: GetPending Verification PASSED
37 us: Info: CPU: main finished
```

Figure 4.14: *GetPendingIRQ* function verification

Table 4.5 shows the results of NVIC's verification results.

Verification condition	Verification test	Pss/Fail
Correctly use NVIC_EnableIRQ method	NVIC_Verification_EnableIRQ()	Pass
Correctly use NVIC_DisableleIRQ method	NVIC_Verification_DisableIRQ()	Pass
Correctly use NVIC_ClearPendingIRQ method	NVIC_Verification_ClearPendingIRQ()	Pass
Correctly use NVIC_GetEnable method	NVIC_Verification_GetEnable()	Pass
Correctly use NVIC_GetPending method	NVIC_Verification_GetPending()	Pass

Table 4.5: NVIC verification results

4.1.4 System verification

After verifying the components individually, we have verified the system as a whole. The description of the system verification is in section 3.5.4. We ran software that would use the models developed in this thesis (GPIO, CLOCK, and NVIC). Figure 4.15 captures the transactions of the system verification showing what is the system doing at each point in time.

```
0 s: Info: SystemBuilder:: System build successful
0 s: Info: Top:: +++ MPSLVP: Virtual Prototype for MPSL Target Tests +++
0 s: Info: (I702) default timescale unit used for tracing: 1 ps (bauhaus_wave.vcd)
2 us: Info: nrf_clock: start initialization routine
2 us: Info: CPU: CPU is starting
37 us: Info: NVIC: Interrupt vectors table is ready
37 us: Info: NVIC: IRQ 0 has been enabled
52 us: Info: CPU: Initiated write transaction of 0x2 to 0x40000304
87 us: Info: nrf_clock: Received expected write request to 304
87 us: Info: nrf_clock: INTENSET register has been updated to   00000000000000000000000000000010
102 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000518
137 us: Info: nrf_clock: Received expected write request to 518
137 us: Info: nrf_clock: LFCLKSRC has been updated
137 us: Info: nrf_clock: Normal XTAL operation
152 us: Info: CPU: Initiated write transaction of 0x1 to 0x40000008
187 us: Info: nrf_clock: Received expected write request to 8
187 us: Info: nrf_clock: LFCLK will start
187 us: Info: nrf_clock: switch to LFRCO
3002 us: Info: nrf_clock: initialization routine is over
25187 us: Info: nrf_clock: wait for 25ms before switching to LFXO
25187 us: Info: nrf_clock: switching back to LFXO
25187 us: Info: nrf_clock: LFCLKSTARTED interrupt has been fired
25187 us: Info: nrf_clock: Interrupt has been fired to NVIC
25187 us: Info: NVIC: Interrupt has been received
25187 us: Info: NVIC: CLOCK interrupt handler
25197 us: Info: CPU: Initiated write transaction of 0x10 to 0x50000004
25202 us: Info: CPU: Initiated read transaction to 50000000
25232 us: Info: nrf_gpio: Received expected write request to 4
25232 us: Info: nrf_gpio: GPIO OUT register 16
25232 us: Info: system verification: system verification PASSED
25232 us: Info: CPU: main finished
```

Figure 4.15: system verification results

4.2 Executed tests on the platform

The first piece of software that ran on the platform was the initialization routine of the base test *mpsl_init()*. This routine would initiate the peripherals and configure some of their registers. We managed to run the initialization routine on the virtual platform, and figure 4.16 shows some of the transactions carried out by the platform.

```
27367157 ns: Info: nrf_rtc: RTC incremented
27387 us: Info: nrf_gpio: Received expected write request to 8
27387 us: Info: NVIC: IRQ 25 pending has been cleared
27387 us: Info: NVIC: IRQ 25 has been enabled
27397314 ns: Info: nrf_rtc: RTC incremented
27402 us: Info: CPU: Initiated write transaction of 0x1 to 0x42000004
27427471 ns: Info: nrf_rtc: RTC incremented
27437 us: Info: nrf_rtc: Received expected write request to 4
27437 us: Info: nrf_rtc: RTC stopped
27437 us: Info: CPU: main finished
27462 us: Info: MPSLVP: Wall-clock Time = 7.65395 ms
```

Figure 4.16: The results of the initialization routine

4.2.1 Results of the base test

The first complete test to be run on the platform was the base test, which calls the initialization routine (*mpsl_init()*), checks the contents of NVIC registers, performs the un-initialization routine (*mpsl_uninit()*) and checks the contents of some of NVIC registers again. Figure 4.17 shows part of the base test's transactions, as well as the execution wall-time for the test on the platform.

```
28572 us: Info: CPU: Initiated write transaction of 0xffffffff to 0x40000308
28607 us: Info: nrf_clock: Received expected write request to 308
28607 us: Info: nrf_clock: INTENCLR register has been updated to   11111111111111111111111111111111
28617 us: Info: CPU: Initiated write transaction of 0x0 to 0x41000ffc
28652 us: Info: nrf_radio: Received expected write request to ffc
28652 us: Info: nrf_radio: Peripheral is powered off
28662 us: Info: CPU: Initiated write transaction of 0x1 to 0x41000ffc
28697 us: Info: nrf_radio: Received expected write request to ffc
28697 us: Info: nrf_radio: Peripheral is powered on
28697 us: Info: NVIC: IRQ 1 has been disabled
28697 us: Info: NVIC: IRQ 1 pending has been cleared
28707 us: Info: CPU: Initiated write transaction of 0x1 to 0x28000004
28742 us: Info: nrf_temp: Received expected write request to 4
28742 us: Info: nrf_temp: Temperature measurement stopped
28752 us: Info: CPU: Initiated write transaction of 0xffffffff to 0x28000308
28787 us: Info: nrf_temp: Received expected write request to 308
28787 us: Info: nrf_temp: DATARDY interrupt disabled
28787 us: Info: NVIC: IRQ 12 has been disabled
28787 us: Info: NVIC: IRQ 12 pending has been cleared
28787 us: Info: NVIC: IRQ 25 has been disabled
28787 us: Info: CPU: main finished
28812 us: Info: MPSLVP: Wall-clock Time = 9.70006 ms
```

Figure 4.17: The results of the base test

4.2.2 Results of CLOCK's tests

We have ran three tests regarding the CLOCK callback in a different scenarios, and their results were as follow:

1. **One-shot CLOCK callback**:

```
55462 us: Info: CPU: Initiated write transaction of 0x1 to 0x28000004
55463304 ns: Info: nrf_rtc: RTC incremented
55493461 ns: Info: nrf_rtc: RTC incremented
55497 us: Info: nrf_temp: Received expected write request to 4
55497 us: Info: nrf_temp: Temperature measurement stopped
55507 us: Info: CPU: Initiated write transaction of 0xffffffff to 0x28000308
55523618 ns: Info: nrf_rtc: RTC incremented
55542 us: Info: nrf_temp: Received expected write request to 308
55542 us: Info: nrf_temp: DATARDY interrupt disabled
55542 us: Info: NVIC: IRQ 12 has been disabled
55542 us: Info: NVIC: IRQ 12 pending has been cleared
55542 us: Info: NVIC: IRQ 25 has been disabled
55552 us: Info: CPU: Initiated write transaction of 0x1 to 0x42000004
55553775 ns: Info: nrf_rtc: RTC incremented
55583932 ns: Info: nrf_rtc: RTC incremented
55587 us: Info: nrf_rtc: Received expected write request to 4
55587 us: Info: nrf_rtc: RTC stopped
55587 us: Info: CPU: main finished
55614089 ns: Info: MPSLVP: Wall-clock Time = 12.156 ms
```

Figure 4.18: The results of One-shot CLOCK callback test

2. **No CLOCK callback is called when HFCLK started by protocol**:

```
56222 us: Info: CPU: Initiated write transaction of 0x1 to 0x41000ffc
56247386 ns: Info: nrf_rtc: RTC incremented
56257 us: Info: nrf_radio: Received expected write request to ffc
56257 us: Info: nrf_radio: Peripheral is powered on
56257 us: Info: NVIC: IRQ 1 has been disabled
56257 us: Info: NVIC: IRQ 1 pending has been cleared
56267 us: Info: CPU: Initiated write transaction of 0x1 to 0x28000004
56277543 ns: Info: nrf_rtc: RTC incremented
56302 us: Info: nrf_temp: Received expected write request to 4
56302 us: Info: nrf_temp: Temperature measurement stopped
56307700 ns: Info: nrf_rtc: RTC incremented
56312 us: Info: CPU: Initiated write transaction of 0xffffffff to 0x28000308
56337857 ns: Info: nrf_rtc: RTC incremented
56347 us: Info: nrf_temp: Received expected write request to 308
56347 us: Info: nrf_temp: DATARDY interrupt disabled
56347 us: Info: NVIC: IRQ 12 has been disabled
56347 us: Info: NVIC: IRQ 12 pending has been cleared
56347 us: Info: NVIC: IRQ 25 has been disabled
56357 us: Info: CPU: Initiated write transaction of 0x1 to 0x42000004
56368014 ns: Info: nrf_rtc: RTC incremented
56392 us: Info: nrf_rtc: Received expected write request to 4
56392 us: Info: nrf_rtc: RTC stopped
56392 us: Info: CPU: main finished
56417 us: Info: MPSLVP: Wall-clock Time = 12.4402 ms
```

Figure 4.19: The results of No CLOCK callback is called when HFCLK started by protocol test

3. **CLOCK callback is called when HFCLK is already triggered**:

```
56122 us: Info: nrf_radio: Peripheral is powered on
56122 us: Info: NVIC: IRQ 1 has been disabled
56122 us: Info: NVIC: IRQ 1 pending has been cleared
56126758 ns: Info: nrf_rtc: RTC incremented
56132 us: Info: CPU: Initiated write transaction of 0x1 to 0x28000004
56156915 ns: Info: nrf_rtc: RTC incremented
56167 us: Info: nrf_temp: Received expected write request to 4
56167 us: Info: nrf_temp: Temperature measurement stopped
56177 us: Info: CPU: Initiated write transaction of 0xffffffff to 0x28000308
56187072 ns: Info: nrf_rtc: RTC incremented
56212 us: Info: nrf_temp: Received expected write request to 308
56212 us: Info: nrf_temp: DATARDY interrupt disabled
56212 us: Info: NVIC: IRQ 12 has been disabled
56212 us: Info: NVIC: IRQ 12 pending has been cleared
56212 us: Info: NVIC: IRQ 25 has been disabled
56217229 ns: Info: nrf_rtc: RTC incremented
56222 us: Info: CPU: Initiated write transaction of 0x1 to 0x42000004
56247386 ns: Info: nrf_rtc: RTC incremented
56257 us: Info: nrf_rtc: Received expected write request to 4
56257 us: Info: nrf_rtc: RTC stopped
56257 us: Info: CPU: main finished
56282 us: Info: MPSLVP: Wall-clock Time = 12.0761 ms
```

Figure 4.20: The results of CLOCK callback is called when HFCLK is already triggered test

4.3 Verbosity and visibility

The virtual platform provides great visibility to the developer where they can see how a certain functionality is being implemented in any of the system models. Having such a high degree of visibility makes the debugging process easier either for hardware or software developers. The platform captures the transaction carried out and generates a log file listing these transactions. Having such a log file could shorten the debugging process since these log messages are readable and easy to understand. The user of the platform will have the ability to control what is being captured and what is being logged. This should be useful to narrow down the amount of information to the suspected model or peripheral. The code below shows how to configure the verbosity of the platform where it will only log and display the transactions related to CLOCK models while ignoring the GPIO model.

```
1 // For debugging only one or more system components
2 // Uncomment the following line and the lines  corresponding to specific
     system components to debug
3  sc_report_handler::set_actions(SC_INFO, SC_DO_NOTHING);
4  sc_report_handler::set_actions("nrf_clock", SC_INFO, SC_DISPLAY | SC_LOG);
5 //sc_report_handler::set_actions("nrf_gpio", SC_INFO, SC_DISPLAY | SC_LOG);
```

In addition to that, the ability to add custom messages and breaking points might decrease the development time. And with shorter development time, the product would have a shorter time-to-market(TTM), and it could also reduce the development cost.

4.4 Reusability and expandability

One of the features that the virtual prototype could provide is the reusability of the system components. It will be useful in case of having different systems that share similar peripheral or components or upgrading one aspect of the system while leaving the rest of the system unchanged. Another feature provided by the system is expandability, which includes expanding the individual models or the whole system. For the individual models, they could be expanded by adding extra functionalities, registers, ports, or signals. Or modifying existing ones to accommodate for changes in the system.
While for the overall prototype, it allows adding new models easily, which increases the testing capabilities of the prototype.

4.5 Virtual prototype vs. development boards

In this section, we evaluate and compare the performance of running the tests on the virtual prototype against the development boards. The verbosity configuration of the tests of virtual prototype disabled the logging and displaying of transactions to get the shortest execution time. The table 4.6 shows the comparison between the execution time of virtual prototype and development boards. The table has the tests implemented by this thesis and by [1].

Test name	Execution Time (ms)		speedup
	Development boards	Virtual prototype	
MPSL init	1390	8.92	155
Temperature Measurement	1384	9.55	144
Oneshot Timer Callback	1535	10.28	149
Oneshot CLOCK callback	1425	9.79	145
CLOCK callback is called when HFCLK is already triggered	1392	9.8	142
No CLOCK callback is called when HFCLK started by protocol	1368	10.25	134

Table 4.6: Comparison table of virtual prototype and development board

Chapter 5

Discussion

This thesis started with the goal of creating a proof-of-concept to virtual prototype in a software testing setup, and provide arguments if it is a good approach to be adopted by Nordic Semiconductor into their development process.
As a start, the virtual prototype managed to execute the same tests as the development boards and obtained the same results. It proved that the virtual prototype was accurate enough for the tests we ran to give the same results. Although they both have the same results for similar tests, having a virtual prototype allows for the testing process to start even before manufacturing the development boards. That would allow for concurrent development of both hardware and software.

While the development boards provide a good platform to debug and test software, we can argue that the virtual prototype provides more debugging capabilities compare to the use of development boards. Due to its better visibility over the system components, and its readable and more comprehensive logs, both of which will help the developers and it could decrease the time of the development process.

The main and perhaps the most interesting finding of this thesis is the speedup of the testing which is obtained from running the software tests on the virtual prototype. The table 4.6 lists the comparisons where we got a speedup between 134 - 155 times across the different tests that we ran. This is a big advantage of the virtual prototype over the use of the development boards, where being able to run the test faster increases the testing productivity and provides a wider test coverage. This could decrease the time spent in software testing and

in turn decrease the time of the development process. And having a shorter development time could decrease the cost of development, and decrease the time needed for the product to be available for customers, time-to-market(TTM).
An additional point for the use of virtual prototype it that it is expandable, modifying or adding functionalities to models or adding models to the system can be done relatively easier and cheaper compared to using different development boars with different capabilities.
From the results of this thesis and based on the tests we ran it seems that using a virtual prototype in the development process would allow for early testing and concurrent software and hardware development. The speedup provided by a virtual prototype would increase the testing productivity and the test coverage. It could also decrease the time-to-market (TTM) by having a shorter development time. It might also reduce the production cost since using virtual prototype would be cheaper than having physical boards or using FPGAs.

The modeled prototype based on the actual hardware used by Nordic Semiconductor, and the tests also based on the actual tests run by the software team. The results of the thesis based on comparing the performance of the actual tests run by the software team against the performance of the virtual prototype. In turn, this gives the results more depth and credibility. However, despite the promising results and their implications, but it should be kept in mind that these results are based on a few tests, so to get more concrete results for the rest of the tests or more complex tests. Our prototype had models of a handful of peripherals that are needed by the tests we chose, but these models are a few percentages of the full system. So adding more models and running more complicated tests that require more peripherals might provide a more accurate representation of the prototype behavior. Another thing which the prototype would benefit greatly from is having a better interface between the prototype and the tested software, as right now we compile and execute both of the software and prototype together, which forced us to modify the tested software to become compatible with the systemC prototype and run correctly. And despite all the advantages that are provided by the use of virtual prototypes, we do not think that it will be able to completely replace the use of physical boards, at least in its current state. It could be introduced as an intermediate stage between the unit testing and target testing for now, and it might be able to replace them in the future with more refined modeling and verification.

Chapter 6

Conclusion

We started this thesis with the goal of providing a proof-of-concept to the feasibility of using a virtual prototype in Nordics Semiconductor's development process. We started by studying their current approach of testing at Nordic Semiconductor followed by identifying some of their tests to proceed with. From the selected tests we extracted the peripherals needed to be modeled and how detailed are they, and we started modeling the peripherals in a similar structure to ease their integration later on to the overall system and ran verification plans for each model individually and the complete system. After having the complete system, we started by modifying the previously selected tests to be compatible with the virtual prototype. We ran the tests on the prototype and evaluated their performance, then compared the performance of each test on the prototype against the development boards. The virtual prototype provided more visibility over the system components which increases the debugging capabilities of the prototype. It is also proven to be expandable, allowing for an easier upgrade of the system functionalities. From the results we got from the tests we ran, we managed to get a speedup of almost 150 for most of the tests, and hence the virtual prototype would increase the testing productivity and potentially reduce the development time, and with shorter development time, it would decrease the time-to-market(TTM) of the products. It could also reduce the cost of concurrent hardware/software development, since having multiple virtual prototypes could be cheaper than the use of multiple FPGAs. But it still unable to replace the physical boards in its current state, it could be used as an intermediate stage between the unit and the target testing.

Appendix A

Acronyms

SoC System-on-Chip

BLE Bluetooth

API Application programming interface

ABI Application Binary interface

HCI Host Controller Interface

PPI Programmable peripheral interconnect

EEP event end point

TEP task end point

NVIC Nested Vector Interrupt Controller

ISER Interrupt Set Enable Register

ICER Interrupt Clear Enable Register

ISPR Interrupt Set Pending Register

ICPR Interrupt Clear Pending Register

IPR Interrupt Priority Register

HFCLK High Frequency Clock

LFCLK Low Frequency Clock

RTC Real-time counter

PCLK1M 1 MHz peripheral clock

PCLK16M 16 MHz peripheral clock

HAL Hardware access layer

IRQ Interrupt request line

MPSL Multiprotocol Service Layer

GPIO General purpose input/output

TLM Transaction level modeling

SCB System Control Block

SCR System Control Register

SHP System Handler Priprity Register

Appendix B

Code

B.1 Template code

```
1
2  #ifndef N_IP_APB_H
3  #define N_IP_APB_H
4
5  #include <systemc>
6
7  #include "tlm.h"
8
9  #include "n_ip.h"
10
11 using namespace sc_core;
12
13 struct Nip_Apb: Nip, tlm::tlm_fw_transport_if<>{
14
15   const int AHB_BASE_ADDR;
16
17   tlm::tlm_target_socket<> apb_target_socket;
18
19   Nip_Apb(sc_module_name name, int ahb_base_addr);
20
21   // TLM target functions
22   virtual void b_transport(tlm::tlm_generic_payload& trans, sc_core::
       sc_time& delay);
```

```
23   virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::
       tlm_dmi& dmi_data);
24   virtual unsigned int transport_dbg(tlm::tlm_generic_payload& trans);
25   virtual tlm::tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload&
       trans, tlm::tlm_phase& phase, sc_time& delay);
26
27 };
28
29
30 #endif
```

B.2 Nested Vector Interrupt Controller (NVIC) model as part of the CPU

```
1
2
3  truct Cpu: NORDIC::n_module, tlm::tlm_bw_transport_if<>{
4
5    NORDIC::CLOCK::n_clock_slave_port clk;
6
7    tlm::tlm_initiator_socket<> apb_init_socket;
8
9    Software* sw;
10
11   sc_event ev_start_cpu;
12
13   // Constructor
14   Cpu (sc_module_name name);
15
16   // SCB register definition
17     struct {
18     uint32_t scbCPUID;                    /*!< Offset: 0x000 (R/ )  CPUID
     Base Register                                      */
19     uint32_t scbICSR;                     /*!< Offset: 0x004 (R/W)
     Interrupt Control and State Register               */
20     uint32_t scbVTOR;                     /*!< Offset: 0x008 (R/W)  Vector
     Table Offset Register                              */
21     uint32_t scbAIRCR;                    /*!< Offset: 0x00C (R/W)
     Application Interrupt and Reset Control Register   */
22     uint32_t scbSCR;                      /*!< Offset: 0x010 (R/W)  System
     Control Register                                   */
23     uint32_t scbCCR;                      /*!< Offset: 0x014 (R/W)
     Configuration Control Register                     */
24     uint8_t  scbSHP[12];                  /*!< Offset: 0x018 (R/W)  System
     Handlers Priority Registers (4-7, 8-11, 12-15) */
25     uint32_t scbSHCSR;                    /*!< Offset: 0x024 (R/W)  System
     Handler Control and State Register                 */
```

```
26    uint32_t scbCFSR;                    /*!< Offset: 0x028 (R/W)
      Configurable Fault Status Register                    */
27    uint32_t scbHFSR;                    /*!< Offset: 0x02C (R/W)
      HardFault Status Register                             */
28    uint32_t scbDFSR;                    /*!< Offset: 0x030 (R/W)  Debug
      Fault Status Register                          */
29    uint32_t scbMMFAR;                   /*!< Offset: 0x034 (R/W)
      MemManage Fault Address Register                      */
30    uint32_t scbBFAR;                    /*!< Offset: 0x038 (R/W)  BusFault
       Address Register                           */
31    uint32_t scbAFSR;                    /*!< Offset: 0x03C (R/W)
      Auxiliary Fault Status Register                       */
32    uint32_t scbPFR[2];                  /*!< Offset: 0x040 (R/ )
      Processor Feature Register                            */
33    uint32_t scbDFR;                     /*!< Offset: 0x048 (R/ )  Debug
      Feature Register                               */
34    uint32_t scbADR;                     /*!< Offset: 0x04C (R/ )
      Auxiliary Feature Register                            */
35    uint32_t scbMMFR[4];                 /*!< Offset: 0x050 (R/ )  Memory
      Model Feature Register                         */
36    uint32_t scbISAR[5];                 /*!< Offset: 0x060 (R/ )
      Instruction Set Attributes Register                   */
37    uint32_t scbRESERVED0[5];
38    uint32_t scbCPACR;                   /*!< Offset: 0x088 (R/W)
      Coprocessor Access Control Register                   */
39    } SCBregs;
40
41   // SCB register definition
42    struct {
43    uint32_t cpuPRIMASK;                 /*Priority Mask Register
                                          */
44    } CPUregs;
45
46 /******** NVIC ********/
47
48    //NVIC registers definition
49    struct {
50     uint32_t nvicISER[8];      // 0x000;
```

```
51      uint32_t nvicICER[8];     // 0x080;
52      uint32_t nvicICPR[8];     // 0x180;
53  } NVICregs;
54
55  /// List of callback functions.
56  std::vector<cb_arg_t> callbacks_;
57
58  /******** NVIC ********/
59
60
61  void main_sw_thread();
62
63  // -- Empty dummy functions for TML compatibility
64  virtual tlm::tlm_sync_enum nb_transport_bw(tlm::tlm_generic_payload&
     trans, tlm::tlm_phase& phase, sc_time& delay);
65  virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::
     uint64 end_range);
66
67  // -- Convenience apb access functions
68  void apb_simple_write(int addr, int& data);
69  void apb_simple_read(int addr, int& data);
70  void apb_simple_transport(int addr, int& data, bool write);
71
72  // -- Functions to write into CPU/SCB/NVIC registers
73  void cpu_reg_write(uint32_t &reg, uint32_t data);
74  uint32_t cpu_reg_read(uint32_t &reg);
75
76  // -- Assembler related functions
77  void wfe_cpu(void);
78  void nop_cpu(void);
79  void enable_irq_cpu(void);
80  void disable_irq_cpu(void);
81  uint32_t get_PRIMASK_cpu(void);
82
83  // -- NVIC functions
84  void NVIC_DisableIRQ(int IRQn );
85  void NVIC_EnableIRQ(int IRQn );
86  void NVIC_ClearPendingIRQ(int IRQn );
```

```
87
88    // -- interrupt handling Functions
89    void InterruptHandlers_registration();
90    void Register_InturreptHandler( const cb_t &cb,  uint32_t val );
91    void callback_handller( uint32_t index);
92    int  callback_check( uint32_t index);
93
94
95
96    // Flag handling utilities
97    void set(uint32_t &reg, uint32_t flag);
98    void clr(uint32_t &reg, uint32_t flag);
99    bool isSet(uint32_t reg, uint32_t flag);
100   bool isClr(uint32_t reg, uint32_t flag);
101
102   void delay_us(double us);
103
104 };
```

B.3 CLOCK code

```
1  //nrf_clock.h
2  #ifndef NRF_CLOCKC_H
3 #define NRF_CLOCKC_H
4
5 #include <iostream>
6 #ifdef _WIN32
7   typedef unsigned __int32 uint32_t;
8 #else
9   #include <stdint.h>
10 #endif
11
12 #include "systemc.h"
13 #include "tlm.h"
14 #include "tlm_utils/simple_target_socket.h"
15
16 #include "n_ip_apb.h"
17
18
19
20 using namespace sc_core;
21
22
23 // Address of the peripheral registers (example)
24 #define CLOCK_ADDR_MASK 0xFFF
25
26 #define TASKS_HFCLKSTART        0x000 //
27 #define TASKS_HFCLKSTOP         0x004 //
28 #define TASKS_LFCLKSTART        0x008 //
29 #define TASKS_LFCLKSTOP         0x00C //
30 #define TASKS_CAL               0x010
31 #define TASKS_CTSTART           0x014
32 #define TASKS_CTSTOP            0x018
33 #define EVENTS_HFCLKSTARTED     0x100 //
34 #define EVENTS_LFCLKSTARTED     0x104 //
35 #define EVENTS_DONE             0x10C
36 #define EVENTS_CTTO             0x110
```

```
#define INTEN                    0x300 // Added register
#define INTENSET                 0x304 //
#define INTENCLR                 0x308 //
#define HFCLKRUN                 0x408 // added (will be used by the HAL)
#define HFCLKSTAT                0x40C //
#define LFCLKSTAT                0x418 //
#define LFCLKSRCCOPY             0x41C // added (will be used by the HAL)
#define LFCLKSRC                 0x518 //
#define HFXODEBOUNCE             0x528 //added register
#define CTIV                     0x538



struct nrf_clock : Nip_Apb{
  public:
  // Constructor
    nrf_clock(sc_module_name name_, int ahb_base_addr);
  // Peripheral register definition (example)
    struct {
       uint32_t clockHFCLKSTART;        //0x000
       uint32_t clockHFCLKSTOP;        //0x004
       uint32_t clockLFCLKSTART;        //0x008
       uint32_t clockLFCLKSTOP;        //0x00C
       uint32_t clockCAL;        //0x010
       uint32_t clockCTSTART;        //0x014
       uint32_t clockCTSTOP;        //0x018
       uint32_t clockHFCLKSTARTED;        //0x100
       uint32_t clockLFCLKSTARTED;        //0x104
       uint32_t clockDONE;        //0x10C
       uint32_t clockCTTO;        //0x110
       uint32_t clockINTEN;        //0x300
       uint32_t clockINTENSET;        //0x304
       uint32_t clockINTENCLR;        //0x308
       uint32_t clockHFCLKRUN;        //0x408
       uint32_t clockHFCLKSTAT;        //0x40C
       uint32_t clockLFCLKSTAT;        //0x418
       uint32_t clockLFCLKSRCCOPY;        //0x41C
       uint32_t clockLFCLKSRC;        //0x518
```

```
75        uint32_t clockHFXODEBOUNCE;        //0x528
76        float clockCTIV;        //0x538
77     } regs;
78
79
80     // Events
81     sc_event start_event;
82     sc_event interrupt_event0;
83     sc_event interrupt_event1;
84     sc_event calibration_timer_event;
85
86     // Peripheral signals
87     sc_signal<bool> pwr_sig; // Signal to handle power changes
88     //sc_out<bool> HFCLKSTARTED_interrupt;
89     sc_out<bool> interrupts[5];
90
91  // Blocking transport function
92     virtual void        b_transport(tlm::tlm_generic_payload &payload,
       sc_time &delay);
93
94     // Debug function
95     unsigned int        transport_dbg(tlm::tlm_generic_payload& gp);
96
97     // Functions to handle power and reset
98     void        power_domain_handler();
99     void        reset_handler();
100
101    // Flag handling utilities
102    void        set(uint32_t &reg, uint32_t flag);
103    void        clr(uint32_t &reg, uint32_t flag);
104    bool        isSet(uint32_t reg, uint32_t flag);
105    bool        isClr(uint32_t reg, uint32_t flag);
106
107
108 private:
109
110     /*Threads*/
111    void        initialization_thread();
```

```
112     void        HFCLK_interrupts_thread();
113     void        LFCLK_interrupts_thread();
114     void        Calibration_Timer_thread();
115 
116 
117   // Function to write to a register
118     virtual void        busWrite(uint32_t  uaddr, uint32_t wdata);
119   // Function to read from a register
120     uint32_t        busRead(uint32_t uaddr);
121 
122     /*delay functions*/
123     void LFXO_start(void);
124     void HFXO_start(void);
125 
126     /*logging functions*/
127     void LFCLKSRClog(void);
128     void LFCLKSTATlog(void);
129     void HFCLKSTATlog(void);
130     void INTENlog(void);
131 
132 };
133 
134 #endif
```

```cpp
//nrf_clock.cpp
#include "nrf_clock.h"
#include <bitset>



nrf_clock::nrf_clock(sc_module_name name_, int ahb_base_addr) : Nip_Apb(
    name_, ahb_base_addr){
     // Constructor
  SC_HAS_PROCESS(nrf_clock);

  //Threads
  SC_THREAD(initialization_thread);
  SC_THREAD(HFCLK_interrupts_thread);
  SC_THREAD(LFCLK_interrupts_thread);
  SC_THREAD(Calibration_Timer_thread);

  // Make power signal asynchronous
  async_reset_signal_is(pwr_sig, false);

  // Register functions to handle power and reset
    SC_METHOD(power_domain_handler);
    sensitive << power_port;
    SC_METHOD(reset_handler);
    sensitive << clk.reset;

    // Clear regs
    memset(&regs, 0, sizeof(regs));
    regs.nrfclockHFXODEBOUNCE = 0x10; //set the debounce time to 256 us
    regs.nrfclockINTEN = 0x0;

    SC_REPORT_INFO("nrf_clock", "Completed constructor");
}


// init thread
```

```
37 void nrf_clock::initialization_thread(){
38     while (1) {
39     wait(start_event);
40     SC_REPORT_INFO("nrf_clock", "start initialization routine");
41     //reset all the registers
42     regs.nrfclockHFXODEBOUNCE = 0x10; //set the debounce time to 256 us
43     regs.nrfclockINTEN = 0x0;
44     wait(3, SC_MS);//wait for HFINT to start up
45     SC_REPORT_INFO("nrf_clock", "initialization routine is over");
46   }
47 }
48
49
50 // Reset behaviour
51 void nrf_clock::reset_handler(){
52        start_event.notify();
53 }
54
55 // Power domain handler
56 void nrf_clock::power_domain_handler() {
57     start_event.notify();
58     switch (power_port->get_power_status()){
59     case NORDIC::POWER::n_power_status::OFF:
60       pwr_sig.write(false);
61       break;
62     case NORDIC::POWER::n_power_status::ON:
63       pwr_sig.write(true);
64       break;
65     default:
66       break;
67     }
68 }
69
70 //Calibration Timer
71 void nrf_clock::Calibration_Timer_thread(){
72     std::ostringstream ostr;
73     while(1){
74       wait(calibration_timer_event);
```

```
75        SC_REPORT_INFO("nrf_clock","calibration timer is starting");
76        while(isClr(regs.nrfclockCTSTOP,0x1)&& (regs.nrfclockCTIV != 0)){
77        wait(25,SC_US);
78        regs.nrfclockCTIV = regs.nrfclockCTIV - 0.25;
79        ostr << "/* CTIV */ " <<regs.nrfclockCTIV ;
80        SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
81        ostr.str("");
82        // std::cout << "/* CTIV */" << regs.nrfclockCTIV << '\n';
83
84        }
85        SC_REPORT_INFO("nrf_clock","calibration timer has finished");
86        regs.nrfclockCTTO = 0x1;
87        ostr << "/* CTTO is */ " <<regs.nrfclockCTTO ;
88        SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
89        ostr.str("");
90        // std::cout << "/* CTTO is */" <<  regs.nrfclockCTTO << '\n';
91
92      }
93
94  }
95
96  //interrupts
97  void nrf_clock::HFCLK_interrupts_thread(){
98    while(1){
99      wait(interrupt_event0);
100     SC_REPORT_INFO("nrf_clock", "HFCLKSTARTED interrupt has been fired");
101     if ( isSet(regs.nrfclockINTEN, 0x1)  ){   // If interrupts are enabled
102       SC_REPORT_INFO("nrf_clock", "Interrupt has been fired to NVIC");
103       interrupts[0].write(true);
104       wait(3, SC_US);//reset the signal to false so we can get future
      interrupts
105       interrupts[0].write(false);
106
107     }
108
109   }
110 }
111
```

```
112 void nrf_clock::LFCLK_interrupts_thread(){
113   while(1){
114     wait(interrupt_event1);
115     SC_REPORT_INFO("nrf_clock", "LFCLKSTARTED interrupt has been fired");
116     if ( isSet(regs.nrfclockINTEN, 0x2)  ){   // If interrupts are enabled
117       SC_REPORT_INFO("nrf_clock", "Interrupt has been fired to NVIC");
118       interrupts[1].write(true);
119       wait(3, SC_US); //reset the signal to false so we can get future
    interrupts
120       interrupts[1].write(false);
121     }
122   }
123 }
124
125 //logging functions
126 void nrf_clock::LFCLKSRClog(){
127   switch(regs.nrfclockLFCLKSRC & 0x3){
128     case 0:
129       if(isClr(regs.nrfclockLFCLKSRC,0x10000) && isClr(regs.
    nrfclockLFCLKSRC,0x20000)){
130         SC_REPORT_INFO("nrf_clock", "Normal operation, RC is source");
131       }else{SC_REPORT_INFO("nrf_clock", "DO NOT USE, change source
    configuration");}
132       break;
133     case 1:
134       if(isClr(regs.nrfclockLFCLKSRC,0x20000)){
135         if(isClr(regs.nrfclockLFCLKSRC,0x10000)){SC_REPORT_INFO("nrf_clock"
    , "Normal XTAL operation");}else{SC_REPORT_INFO("nrf_clock", "DO NOT USE
    , change source configuration");}
136       }else{if(isClr(regs.nrfclockLFCLKSRC,0x10000)){SC_REPORT_INFO("
    nrf_clock", "Apply external low swing signal to XL1, ground XL2");}else{
    SC_REPORT_INFO("nrf_clock", "Apply external full swing signal to XL1,
    leave XL2 unconnected");}
137     }
138     break;
139     case 2:
140       if(isClr(regs.nrfclockLFCLKSRC,0x20000)){
141         SC_REPORT_INFO("nrf_clock", "Normal operation, synth is source");
```

```
142       }else{
143         SC_REPORT_INFO("nrf_clock", "DO NOT USE, change source
    configuration");
144       }
145       break;
146       default:
147       break;
148   }
149
150   // if(isSet(regs.nrfclockLFCLKSRC,0x1))SC_REPORT_INFO("nrf_clock", "The
     LFCLK source is 32.768 kHz crystal oscillator");
151   // if(isSet(regs.nrfclockLFCLKSRC,0x2)) SC_REPORT_INFO("nrf_clock", "The
     LFCLK source is 32.768 kHz synthesized from HFCLK");
152   // if(isClr(regs.nrfclockLFCLKSRC,0x1) && isClr(regs.nrfclockLFCLKSRC,0x2
     ))SC_REPORT_INFO("nrf_clock", "The LFCLK source is 32.768 kHz RC
     oscillator");
153   // if(isSet(regs.nrfclockLFCLKSRC,0x10000)){SC_REPORT_INFO("nrf_clock", "
     BYPASS is Enable");}else{SC_REPORT_INFO("nrf_clock", "BYPASS is Disable
     ");}
154   // if(isSet(regs.nrfclockLFCLKSRC,0x20000)){SC_REPORT_INFO("nrf_clock", "
     Enable use of external source");}else{SC_REPORT_INFO("nrf_clock", "
     Disable use of external source");}
155 }
156
157 void nrf_clock::LFCLKSTATlog(){
158   if(isSet(regs.nrfclockLFCLKSTAT,0x1))SC_REPORT_INFO("nrf_clock", "The
     LFCLK source is 32.768 kHz crystal oscillator");
159   if(isSet(regs.nrfclockLFCLKSTAT,0x2)) SC_REPORT_INFO("nrf_clock", "The
     LFCLK source is 32.768 kHz synthesized from HFCLK");
160   if(isClr(regs.nrfclockLFCLKSTAT,0x1) && isClr(regs.nrfclockLFCLKSRC,0x2))
     SC_REPORT_INFO("nrf_clock", "The LFCLK source is 32.768 kHz RC
     oscillator");
161   if(isSet(regs.nrfclockLFCLKSTAT,0x10000)){SC_REPORT_INFO("nrf_clock", "
     LFCLK state is Running");}else{SC_REPORT_INFO("nrf_clock", "LFCLK state
     is NotRunning");}
162 }
163
164 void nrf_clock::HFCLKSTATlog(){
```

```
165   if(isSet(regs.nrfclockHFCLKSTAT,0x1)){SC_REPORT_INFO("nrf_clock", "HFCLK
        source is 64 MHz crystal oscillator (HFXO)");}else{SC_REPORT_INFO("
        nrf_clock", "HFCLK source is 64 MHz internal oscillator (HFINT)");}
166   if(isSet(regs.nrfclockHFCLKSTAT,0x10000)){SC_REPORT_INFO("nrf_clock", "
        HFXO state is Running");}else{SC_REPORT_INFO("nrf_clock", "HFXO state is
         NotRunning");}
167 }
168
169 void nrf_clock::INTENlog(){
170   if(isSet(regs.nrfclockINTEN,0x1)){SC_REPORT_INFO("nrf_clock", "
        HFCLKSTARTED is Enabled");}else{SC_REPORT_INFO("nrf_clock", "
        HFCLKSTARTED is Disabled");}
171   if(isSet(regs.nrfclockINTEN,0x2)){SC_REPORT_INFO("nrf_clock", "
        LFCLKSTARTED is Enabled");}else{SC_REPORT_INFO("nrf_clock", "
        LFCLKSTARTED is Disabled");}
172   if(isSet(regs.nrfclockINTEN,0x8)){SC_REPORT_INFO("nrf_clock", "DONE is
        Enabled");}else{SC_REPORT_INFO("nrf_clock", "DONE is Disabled");}
173   if(isSet(regs.nrfclockINTEN,0x10)){SC_REPORT_INFO("nrf_clock", "CTTO is
        Enabled");}else{SC_REPORT_INFO("nrf_clock", "CTTO is Disabled");}
174   if(isSet(regs.nrfclockINTEN,0x400)){SC_REPORT_INFO("nrf_clock", "
        CTSTARTED is Enabled");}else{SC_REPORT_INFO("nrf_clock", "CTSTARTED is
        Disabled");}
175   if(isSet(regs.nrfclockINTEN,0x800)){SC_REPORT_INFO("nrf_clock", "
        CTSTOPPED is Enabled");}else{SC_REPORT_INFO("nrf_clock", "CTSTOPPED is
        Disabled");}
176
177 }
178
179 //delay functions
180
181 //model the delay till HFXO start
182 void nrf_clock::HFXO_start(){
183   int HFXO_delay;
184   SC_REPORT_INFO("nrf_clock", "switching to HFXO");
185   if(isSet(regs.nrfclockHFXODEBOUNCE,0x10)){HFXO_delay = 360 + 256;}else{
        HFXO_delay = 360 + 1024;}
186   std::ostringstream ostr;
187   ostr << "waiting for "  << HFXO_delay<<'\t'<<"till HFXO is running";
```

```
188   SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
189   ostr.str("");
190   wait(clk.get_delay(HFXO_delay, SC_US));// Simulate time to start external
        HFXO 256 Dbounce 360 startup time
191   SC_REPORT_INFO("nrf_clock", "HFXO is running");
192   regs.nrfclockHFCLKSTARTED = 0x1;
193   interrupt_event0.notify();
194 }
195
196 //model the delay till LFXO start
197 void nrf_clock::LFXO_start(){
198 if(isSet(regs.nrfclockLFCLKSRC,0x1)){
199   SC_REPORT_INFO("nrf_clock", "switch to LFRCO");
200   clr(regs.nrfclockLFCLKSTAT , 0x1); //switch to the internal RC LFCLKSRC
201   wait(clk.get_delay(25, SC_MS));// Simulate time to start external
       LFCLKSRC
202   SC_REPORT_INFO("nrf_clock", "wait for 25ms before switching to LFXO");
203   SC_REPORT_INFO("nrf_clock", "switching back to LFXO");
204   set(regs.nrfclockLFCLKSTAT , 0x1); //switch back to the LFXO
205 }
206   regs.nrfclockLFCLKSTARTED = 0x1;
207   interrupt_event1.notify();
208 }
209
210
211 // Read from registers. Call different functions here as required
212 uint32_t nrf_clock::busRead(uint32_t uaddr){
213   float data = 0;
214   sc_time delay;
215
216   std::ostringstream ostr;
217   ostr << "Received expected read request from " << std::hex << uaddr;
218   SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
219   ostr.str("");
220
221   switch(uaddr){
222   case NRF_CLOCK_TASKS_HFCLKSTART:
223     data = regs.nrfclockHFCLKSTART;
```

```
224     ostr <<  "TASKS_HFCLKSTART"<<std::bitset<32>(data);
225     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
226     ostr.str("");
227     break;
228   case NRF_CLOCK_TASKS_HFCLKSTOP:
229     data = regs.nrfclockHFCLKSTOP;
230     ostr <<  "TASKS_HFCLKSTOP"<<std::bitset<32>(data);
231     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
232     ostr.str("");
233     break;
234   case NRF_CLOCK_TASKS_LFCLKSTART:
235     data = regs.nrfclockLFCLKSTART;
236     ostr <<  "TASKS_LFCLKSTART"<<std::bitset<32>(data);
237     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
238     ostr.str("");
239     break;
240   case NRF_CLOCK_TASKS_LFCLKSTOP:
241     data = regs.nrfclockLFCLKSTOP;
242     ostr <<  "TASKS_LFCLKSTOP"<<std::bitset<32>(data);
243     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
244     ostr.str("");
245     break;
246   case NRF_CLOCK_TASKS_CAL:
247     SC_REPORT_INFO("nrf_clock", " TASKS_CAL is a write-only register");
248     break;
249   case NRF_CLOCK_TASKS_CTSTART:
250     SC_REPORT_INFO("nrf_clock", " TASKS_CTSTART is a write-only register");
251     break;
252   case NRF_CLOCK_TASKS_CTSTOP:
253     SC_REPORT_INFO("nrf_clock", " TASKS_CTSTOP is a write-only register");
254     break;
255   case NRF_CLOCK_EVENTS_HFCLKSTARTED:
256     data = regs.nrfclockHFCLKSTARTED;
257     if (isSet(regs.nrfclockHFCLKSTARTED, 0x1)){
258     SC_REPORT_INFO("nrf_clock", " EVENTS_HFCLKSTARTED has been generated");
259   }else{
260     SC_REPORT_INFO("nrf_clock", " EVENTS_HFCLKSTARTED has not been
      generated yet");
```

```
261   }
262     break;
263   case NRF_CLOCK_EVENTS_LFCLKSTARTED:
264     data = regs.nrfclockLFCLKSTARTED;
265     if (isSet(regs.nrfclockLFCLKSTARTED, 0x1)){
266     SC_REPORT_INFO("nrf_clock", " EVENTS_LFCLKSTARTED has been generated");
267   }else{
268     SC_REPORT_INFO("nrf_clock", " EVENTS_LFCLKSTARTED has not been
      generated yet");
269   }
270     break;
271   case NRF_CLOCK_EVENTS_DONE:
272     data = regs.nrfclockDONE;
273     ostr <<  "EVENT DONE register is:  "<<data;
274     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
275     ostr.str("");
276
277     break;
278   case NRF_CLOCK_EVENTS_CTTO:
279     data = regs.nrfclockCTTO;
280     ostr <<  "CTTO event is   "<<data;
281     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
282     ostr.str("");
283     break;
284   case NRF_CLOCK_INTENSET:
285     data = regs.nrfclockINTENSET;
286     ostr <<  "INTENSET register is   "<<std::bitset<32>(data);
287     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
288     ostr.str("");
289
290     break;
291   case NRF_CLOCK_INTENCLR:
292     data = regs.nrfclockINTENCLR;
293     ostr <<  "INTENCLR register is   "<<std::bitset<32>(data);
294     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
295     ostr.str("");
296
297     break;
```

```
298   case NRF_CLOCK_HFCLKSTAT:
299     data=regs.nrfclockHFCLKSTAT;
300     HFCLKSTATlog();
301     break;
302   case NRF_CLOCK_LFCLKSTAT:
303     data=regs.nrfclockLFCLKSTAT;
304     LFCLKSTATlog();
305     break;
306   case NRF_CLOCK_LFCLKSRC:
307     data = regs.nrfclockLFCLKSRC;
308     LFCLKSRClog();
309     break;
310   case NRF_CLOCK_CTIV:
311     data = regs.nrfclockCTIV;
312     ostr <<  "Calibration timer interval is  "<<data<<" seconds";
313     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
314     ostr.str("");
315     break;
316   case NRF_CLOCK_INTEN:
317     data = regs.nrfclockINTEN;
318     INTENlog();
319     break;
320   case NRF_CLOCK_HFCLKRUN:
321     data = regs.nrfclockHFCLKRUN;
322     ostr <<  "HFCLKRUN register is   "<<std::bitset<32>(data);
323     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
324     ostr.str("");
325
326     break;
327   case NRF_CLOCK_LFCLKSRCCOPY:
328     data = regs.nrfclockLFCLKSRCCOPY;
329     ostr <<  "LFCLKSRCCOPY register is   "<<std::bitset<32>(data);
330     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
331     ostr.str("");
332
333     break;
334   case NRF_CLOCK_HFXODEBOUNCE:
335     data = regs.nrfclockHFXODEBOUNCE;
```

```
336     ostr <<  "HFXODEBOUNCE register is   "<<std::bitset<32>(data);
337     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
338     ostr.str("");
339 
340     break;
341   }
342   //SC_REPORT_INFO("nrf_clock", "Received expected read request");
343   return (data);
344 }
345 
346 // Write to registers. Call different functions here as required
347 void nrf_clock::busWrite(uint32_t uaddr, uint32_t wdata) {
348   sc_time delay;
349 
350   std::ostringstream ostr;
351   ostr << "Received expected write request to "<<std::hex<<  uaddr;
352   SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
353   ostr.str("");
354 
355   switch(uaddr){
356   case NRF_CLOCK_TASKS_HFCLKSTART:
357     if(isSet(regs.nrfclockHFCLKSTAT,0x10000)){
358       SC_REPORT_INFO("nrf_clock","HFXO is already running");
359     }else{
360       if(wdata == 0x1){
361     regs.nrfclockHFCLKSTART = wdata;
362     SC_REPORT_INFO("nrf_clock","HFXO will start \n you can use the RADIO");
363     set(regs.nrfclockHFCLKSTAT,0x10001);//SRC external oscillator and STAT
    is running
364     regs.nrfclockHFCLKRUN = 0x1;
365     HFXO_start();
366       }}
367     break;
368   case NRF_CLOCK_TASKS_HFCLKSTOP:
369     if(isClr(regs.nrfclockHFCLKSTAT,0x10000)){
370       SC_REPORT_INFO("nrf_clock","HFXO is already stopped");
371     }else{
372       if(wdata & 0x1){
```

```
373     regs.nrfclockHFCLKSTOP = wdata;
374     SC_REPORT_INFO("nrf_clock","HFXO will stop");
375     clr(regs.nrfclockHFCLKSTAT,0x10001);//STAT is NotRunning
376     //regs.nrfclockHFCLKSTARTED = 0x0000;//resets the event //this step
    done by the user before calibration
377     regs.nrfclockHFCLKRUN = 0x0;}
378       }
379     break;
380   case NRF_CLOCK_TASKS_LFCLKSTART:
381   if(isSet(regs.nrfclockLFCLKSTAT,0x10000)){
382     SC_REPORT_INFO("nrf_clock","LFCLK is already running");
383   }else{
384     regs.nrfclockLFCLKSTART = wdata;
385     SC_REPORT_INFO("nrf_clock","LFCLK will start");
386     LFXO_start();
387     set(regs.nrfclockLFCLKSTAT,0x10000);//STAT is Running
388     regs.nrfclockLFCLKSRCCOPY = regs.nrfclockLFCLKSTAT;
389       }
390     break;
391   case NRF_CLOCK_TASKS_LFCLKSTOP:
392   if(isClr(regs.nrfclockLFCLKSTAT,0x10000)){
393       SC_REPORT_INFO("nrf_clock","LFCLK is already stopped");
394     }else{
395     regs.nrfclockLFCLKSTOP = wdata;
396     SC_REPORT_INFO("nrf_clock","LFCLK will stop");
397     clr(regs.nrfclockLFCLKSTAT,0x10000);//STAT is Not Running
398     //regs.nrfclockLFCLKSTARTED = 0x0;//resets the event //this step done
    by the user before calibration
399     }
400     break;
401   case NRF_CLOCK_TASKS_CAL:
402     if(isClr(regs.nrfclockHFCLKSTARTED,0x1)){
403       SC_REPORT_INFO("nrf_clock","Start HFXO first");
404     }else{
405       regs.nrfclockCAL = wdata ;
406       regs.nrfclockDONE = wdata;
407       SC_REPORT_INFO("nrf_clock","Calibration process is starting");
408
```

```
409     }
410     break;
411   case NRF_CLOCK_TASKS_CTSTART:
412   if(isSet(wdata,0x1) || isClr(wdata,0x1)){
413     regs.nrfclockCTSTART = wdata;
414     SC_REPORT_INFO("nrf_clock","Calibration timer is starting");
415
416     if(wdata == 1) calibration_timer_event.notify();
417
418   }else{
419     SC_REPORT_INFO("nrf_clock","incorrect value for calibration timer start
      ");
420   }
421     break;
422   case NRF_CLOCK_TASKS_CTSTOP:
423   if(isSet(wdata,0x1) || isClr(wdata,0x1)){
424     regs.nrfclockCTSTOP = wdata ;
425     SC_REPORT_INFO("nrf_clock","Calibration timer is stoping");
426 }
427     else {
428       SC_REPORT_INFO("nrf_clock","incorrect value for calibration timer
     stop");
429     }
430     break;
431   case NRF_CLOCK_EVENTS_HFCLKSTARTED:
432     regs.nrfclockHFCLKSTARTED = wdata;
433     ostr << "HFCLKSTARTED event has been updated to   "<< wdata;
434     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
435     ostr.str("");
436     break;
437   case NRF_CLOCK_EVENTS_LFCLKSTARTED:
438     regs.nrfclockLFCLKSTARTED = wdata;
439     ostr << "LFCLKSTARTED event has been updated to   "<< wdata;
440     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
441     ostr.str("");
442     break;
443   case NRF_CLOCK_EVENTS_DONE:
444     regs.nrfclockDONE = wdata;
```

```
445     ostr << "DONE event has been updated to   "<< wdata;
446     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
447     ostr.str("");
448     break;
449   case NRF_CLOCK_EVENTS_CTTO:
450     regs.nrfclockCTTO = wdata;
451     ostr << "CTTO event has been updated to   "<< wdata;
452     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
453     ostr.str("");
454     break;
455   case NRF_CLOCK_INTENSET:
456     regs.nrfclockINTENSET = wdata;
457     set(regs.nrfclockINTEN,regs.nrfclockINTENSET);
458     ostr << "INTENSET register has been updated to   "<<std::bitset<32>(
    wdata);
459     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
460     ostr.str("");
461     break;
462   case NRF_CLOCK_INTENCLR:
463     regs.nrfclockINTENCLR = wdata;
464     clr(regs.nrfclockINTEN,regs.nrfclockINTENCLR);
465     ostr << "INTENCLR register has been updated to   "<<std::bitset<32>(
    wdata);
466     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
467     ostr.str("");
468     break;
469   case NRF_CLOCK_HFCLKSTAT:
470     SC_REPORT_INFO("nrf_clock","LFCLKSTAT is read-only register");
471     break;
472   case NRF_CLOCK_LFCLKSTAT:
473     SC_REPORT_INFO("nrf_clock","LFCLKSTAT is read-only register");
474     break;
475   case NRF_CLOCK_LFCLKSRC:
476     //check that the LFCLK is Not Running
477     if(isClr(regs.nrfclockLFCLKSTAT, 0x10000)){
478       if (isSet(wdata, 0x3)){
479         SC_REPORT_INFO("nrf_clock", "source unknown, try again");
480       }else{
```

```
481         regs.nrfclockLFCLKSRC = wdata;
482         set(regs.nrfclockLFCLKSTAT,(wdata & 0x3));
483         SC_REPORT_INFO("nrf_clock", "LFCLKSRC has been updated");
484         LFCLKSRClog();
485 
486       }
487     }else{
488       SC_REPORT_INFO("nrf_clock", "LFCLK is running, can't change LFCLK
    source");
489     }
490     break;
491   case NRF_CLOCK_CTIV:
492     regs.nrfclockCTIV = wdata * 0.25;
493     ostr << "CTIV register has been updated to   "<<  regs.nrfclockCTIV ;
494     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
495     ostr.str("");
496     break;
497   case NRF_CLOCK_INTEN:
498     regs.nrfclockINTEN = wdata;
499     ostr << "INTEN register has been updated to   "<< std::bitset<32>(wdata
    );
500     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
501     ostr.str("");
502 
503     break;
504   case NRF_CLOCK_HFCLKRUN:
505     SC_REPORT_INFO("nrf_clock","HFCLKRUN is read-only register");
506     break;
507   case NRF_CLOCK_LFCLKSRCCOPY:
508     SC_REPORT_INFO("nrf_clock","LFCLKSRCCOPY is read-only register");
509     break;
510   case NRF_CLOCK_HFXODEBOUNCE:
511     regs.nrfclockHFXODEBOUNCE = wdata;
512     ostr << "HFXODEBOUNCE register has been updated to   "<<wdata;
513     SC_REPORT_INFO("nrf_clock", ostr.str().c_str());
514     ostr.str("");
515 
516     break;
```

```
517   }
518   //SC_REPORT_INFO("nrf_clock", "Received expected write request");
519 
520 }
521 
522 // TLM b_transport function
523 void nrf_clock::b_transport(tlm::tlm_generic_payload &gp, sc_time &delay){
524   wait(clk.get_delay(3));
525 
526   // Get address from the generic payload
527     sc_dt::uint64 addr = gp.get_address();
528 
529     uint32_t          *dataPtr;   // Pointer to data in the generic payload
530     int                offset;    // Data byte offset in word
531     uint32_t           uaddr;   // Peripheral address after the mask
532 
533     // Mask off the address to its range
534     uaddr = (uint32_t)(addr) & CLOCK_ADDR_MASK;
535     offset = 0;
536 
537   // Get data pointer
538     dataPtr = reinterpret_cast<uint32_t*>(gp.get_data_ptr());
539 
540     // Read of write to a register depending on the command
541     switch(gp.get_command()){
542       case tlm::TLM_READ_COMMAND:
543         dataPtr[offset] = busRead(uaddr);
544         break;
545       case tlm::TLM_WRITE_COMMAND:
546         busWrite(uaddr, dataPtr[offset]);
547         break;
548       case tlm::TLM_IGNORE_COMMAND:
549         gp.set_response_status(tlm::TLM_GENERIC_ERROR_RESPONSE);
550         break;
551         return;
552     }
553 
554     gp.set_response_status(tlm::TLM_OK_RESPONSE);
```

```
555     delay += sc_time(10, SC_NS);
556 }
557
558 // TLM transport_dbg function
559 unsigned int nrf_clock::transport_dbg(tlm::tlm_generic_payload& gp){
560
561     sc_dt::uint64 addr    = gp.get_address();
562     uint32_t        *dataPtr;
563     unsigned int    len = gp.get_data_length();
564
565     int             offset;
566     uint32_t        uaddr;
567
568     uaddr = (uint32_t) addr;
569     offset = 0;
570
571     dataPtr = reinterpret_cast<uint32_t*>(gp.get_data_ptr());
572
573     switch (gp.get_command()){
574       case tlm::TLM_READ_COMMAND:
575         dataPtr[offset] = busRead(uaddr);
576         break;
577       case tlm::TLM_WRITE_COMMAND:
578         busWrite(uaddr, dataPtr[offset]);
579         break;
580       case tlm::TLM_IGNORE_COMMAND:
581         gp.set_response_status(tlm::TLM_GENERIC_ERROR_RESPONSE);
582         return (0);
583     }
584
585     gp.set_response_status(tlm::TLM_OK_RESPONSE);
586     return (len);
587 }
588
589 // Flag handling utilities
590 void nrf_clock::set(uint32_t &reg, uint32_t flags){
591     reg |= flags;
592 }
```

```
593
594 void nrf_clock::clr(uint32_t &reg, uint32_t flags){
595     reg &= ~flags;
596 }
597
598 bool nrf_clock::isSet(uint32_t reg, uint32_t flags){
599     return  flags == (reg & flags);
600 }
601
602 bool nrf_clock::isClr(uint32_t reg, uint32_t flags){
603     return  flags != (reg & flags);
604 }
```

B.4 GPIO code

```
1  //nrf_gpio.h
2
3  #ifndef NRF_GPIO_H
4  #define NRF_GPIO_H
5
6  #include <iostream>
7  #ifdef _WIN32
8    typedef unsigned __int32 uint32_t;
9  #else
10   #include <stdint.h>
11 #endif
12
13 #include "systemc.h"
14 #include "tlm.h"
15 #include "tlm_utils/simple_target_socket.h"
16
17 #include "n_ip_apb.h"
18
19
20
21 using namespace sc_core;
22
23
24 // Address of the peripheral registers (example)
25 #define GPIO_ADDR_MASK 0xFFF
26
27 #define NRF_GPIO_OUT           0x000 //added
28 #define NRF_GPIO_OUTSET        0x004 //
29 #define NRF_GPIO_OUTCLR        0x008 //
30
31
32
33
34 struct nrf_gpio : Nip_Apb{
35   public:
36   // Constructor
```

```
37    nrf_gpio(sc_module_name name_, int ahb_base_addr);
38  // Peripheral register definition (example)
39    struct {
40       uint32_t nrfgpioOUT;            //0x000
41       uint32_t nrfgpioOUTSET;        //0x004
42       uint32_t nrfgpioOUTCLR;        //0x008
43    } regs;
44
45
46    // Events
47    sc_event start_event;
48
49
50    // Peripheral signals
51    sc_signal<bool> pwr_sig; // Signal to handle power changes
52
53
54  // Blocking transport function
55    virtual void        b_transport(tlm::tlm_generic_payload &payload,
     sc_time &delay);
56
57    // Debug function
58    unsigned int        transport_dbg(tlm::tlm_generic_payload& gp);
59
60    // Functions to handle power and reset
61    void        power_domain_handler();
62    void        reset_handler();
63
64    // Flag handling utilities
65    void        set(uint32_t &reg, uint32_t flag);
66    void        clr(uint32_t &reg, uint32_t flag);
67    bool        isSet(uint32_t reg, uint32_t flag);
68    bool        isClr(uint32_t reg, uint32_t flag);
69
70
71  private:
72
73     /*Threads*/
```

```
74      void        initialization_thread();
75
76
77
78    // Function to write to a register
79      virtual void        busWrite(uint32_t  uaddr, uint32_t wdata);
80    // Function to read from a register
81      uint32_t        busRead(uint32_t uaddr);
82
83
84  };
85
86  #endif
```

```
\\ nrf_gpio.cpp
#include "nrf_gpio.h"
#include <bitset>



nrf_gpio::nrf_gpio(sc_module_name name_, int ahb_base_addr) : Nip_Apb(name_
    , ahb_base_addr){
     // Constructor
  SC_HAS_PROCESS(nrf_gpio);

  //Threads
  SC_THREAD(initialization_thread);

  // Make power signal asynchronous
  async_reset_signal_is(pwr_sig, false);

  // Register functions to handle power and reset
    SC_METHOD(power_domain_handler);
    sensitive << power_port;
    SC_METHOD(reset_handler);
    sensitive << clk.reset;

    // Clear regs
    memset(&regs, 0, sizeof(regs));


    SC_REPORT_INFO("nrf_gpio", "Completed constructor");
}


// init thread
void nrf_gpio::initialization_thread(){
    while (1) {
    wait(start_event);
    SC_REPORT_INFO("nrf_gpio", "start initialization routine");
```

```
37      //reset all the registers
38
39      SC_REPORT_INFO("nrf_gpio", "initialization routine is over");
40    }
41  }
42
43
44  // Reset behaviour
45  void nrf_gpio::reset_handler(){
46         start_event.notify();
47  }
48
49  // Power domain handler
50  void nrf_gpio::power_domain_handler() {
51      start_event.notify();
52      switch (power_port->get_power_status()){
53      case NORDIC::POWER::n_power_status::OFF:
54        pwr_sig.write(false);
55        break;
56      case NORDIC::POWER::n_power_status::ON:
57        pwr_sig.write(true);
58        break;
59      default:
60        break;
61      }
62  }
63
64
65
66
67
68
69  // Read from registers. Call different functions here as required
70  uint32_t nrf_gpio::busRead(uint32_t uaddr){
71    uint32_t data = 0;
72    sc_time delay;
73    //int wdata = 0; //temporary for the simulation
74
```

```
75   switch(uaddr){
76   case NRF_GPIO_OUT:
77     data = regs.nrfgpioOUT;
78     std::cout <<"regs OUT"<<'\t'<< std::bitset<32> (data) << '\n';
79
80     break;
81   case NRF_GPIO_OUTSET:
82     data = regs.nrfgpioOUTSET;
83     break;
84   case NRF_GPIO_OUTCLR:
85     data = regs.nrfgpioOUTCLR;
86     break;
87   default:
88     break;
89   }
90   //SC_REPORT_INFO("nrf_gpio", "Received expected read request");
91   return (data);
92 }
93
94 // Write to registers. Call different functions here as required
95 void nrf_gpio::busWrite(uint32_t uaddr, uint32_t wdata) {
96   sc_time delay;
97
98   std::ostringstream ostr;
99   ostr << "Received expected write request to "<<std::hex<<  uaddr;
100  SC_REPORT_INFO("nrf_gpio", ostr.str().c_str());
101  ostr.str("");
102
103  switch(uaddr){
104  case NRF_GPIO_OUT:
105    regs.nrfgpioOUT = wdata;
106    break;
107  case NRF_GPIO_OUTSET:
108    regs.nrfgpioOUTSET = wdata;
109    // std::cout << "regs OUTSET"<<'\t'<< std::bitset<32>(regs.
     nrfgpioOUTSET) << '\n';
110    set(regs.nrfgpioOUT, regs.nrfgpioOUTSET);
111    // std::cout <<"regs OUT"<<'\t'<< std::bitset<32>(regs.nrfgpioOUT) <<
```

```
    '\n';
112
113     break;
114   case NRF_GPIO_OUTCLR:
115     regs.nrfgpioOUTCLR = wdata;
116     clr(regs.nrfgpioOUT, regs.nrfgpioOUTCLR);
117     break;
118   default:
119     break;
120   }
121   //SC_REPORT_INFO("nrf_gpio", "Received expected write request");
122
123 }
124
125 // TLM b_transport function
126 void nrf_gpio::b_transport(tlm::tlm_generic_payload &gp, sc_time &delay){
127   wait(clk.get_delay(3));
128
129   // Get address from the generic payload
130     sc_dt::uint64 addr = gp.get_address();
131
132     uint32_t          *dataPtr;   // Pointer to data in the generic payload
133     int                offset;    // Data byte offset in word
134     uint32_t           uaddr;   // Peripheral address after the mask
135
136
137
138
139     // Mask off the address to its range
140     uaddr = (uint32_t)(addr) & GPIO_ADDR_MASK;
141     offset = 0;
142
143   // Get data pointer
144     dataPtr = reinterpret_cast<uint32_t*>(gp.get_data_ptr());
145
146     // Read of write to a register depending on the command
147     switch(gp.get_command()){
148       case tlm::TLM_READ_COMMAND:
```

```
149         /* The if statment is used to fix an issue when reading
150         *data that is more than 3-byte size
151         */
152         // unsigned int count;
153         // count = 0;
154         // uint32_t read;
155         // read = busRead(uaddr);
156         // if(read > 0xFFFFFF){
157        //
158         // }
159         // std::cout << std::bitset<32>(read) << '\n';
160        // while (read) {
161         //   count += read & 1;
162         //   read >>= 1;
163        // }
164        // std::cout << count << '\n';
165         // if ( busRead(uaddr) > 0xFFFFFF){
166         // dataPtr[1] = busRead(uaddr);
167         // dataPtr[offset] = busRead(uaddr) >> 24;}
168         // else{
169           dataPtr[offset] = busRead(uaddr) ;//}
170         break;
171       case tlm::TLM_WRITE_COMMAND:
172         busWrite(uaddr, dataPtr[offset]);
173         break;
174       case tlm::TLM_IGNORE_COMMAND:
175         gp.set_response_status(tlm::TLM_GENERIC_ERROR_RESPONSE);
176         break;
177         return;
178     }
179
180     gp.set_response_status(tlm::TLM_OK_RESPONSE);
181     delay += sc_time(10, SC_NS);
182 }
183
184 // TLM transport_dbg function
185 unsigned int nrf_gpio::transport_dbg(tlm::tlm_generic_payload& gp){
186
```

```
187     sc_dt::uint64 addr    = gp.get_address();
188     uint32_t        *dataPtr;
189     unsigned int    len = gp.get_data_length();
190 
191     int             offset;
192     uint32_t        uaddr;
193 
194     uaddr = (uint32_t) addr;
195     offset = 0;
196 
197     dataPtr = reinterpret_cast<uint32_t*>(gp.get_data_ptr());
198 
199     switch (gp.get_command()){
200       case tlm::TLM_READ_COMMAND:
201         dataPtr[offset] = busRead(uaddr);
202         break;
203       case tlm::TLM_WRITE_COMMAND:
204         busWrite(uaddr, dataPtr[offset]);
205         break;
206       case tlm::TLM_IGNORE_COMMAND:
207         gp.set_response_status(tlm::TLM_GENERIC_ERROR_RESPONSE);
208         return (0);
209     }
210 
211     gp.set_response_status(tlm::TLM_OK_RESPONSE);
212     return (len);
213 }
214 
215 // Flag handling utilities
216 void nrf_gpio::set(uint32_t &reg, uint32_t flags){
217     reg |= flags;
218 }
219 
220 void nrf_gpio::clr(uint32_t &reg, uint32_t flags){
221     reg &= ~flags;
222 }
223 
224 bool nrf_gpio::isSet(uint32_t reg, uint32_t flags){
```

```
225     return  flags == (reg & flags);
226 }
227
228 bool nrf_gpio::isClr(uint32_t reg, uint32_t flags){
229     return  flags != (reg & flags);
230 }
```

B.5 Verification tests

B.5.1 NVIC verification tests

```
1  #include "sw_main.h"
2  #include "cpu.h"
3  #include <bitset>
4
5  void Software::sw_main(void){
6
7      NVIC_verification_EnableIRQ();
8    NVIC_verification_DisableIRQ();
9    NVIC_verification_ClearPendingIRQ();
10   NVIC_verification_GetEnable();
11   NVIC_verification_GetPending();
12
13 }
14
15 void Software::NVIC_verification_EnableIRQ(void){
16 int irq = 0;
17 uint32_t reg;
18 int pass = 0;
19
20 for (irq; irq<30;irq++){
21 cpu_ptr->NVIC_EnableIRQ(irq);
22 reg = cpu_ptr->cpu_reg_read(cpu_ptr->NVICregs.nvicISER[0]);
23 if(!(reg & 1 << irq)){
24   SC_REPORT_INFO("NVIC Verification","EnableIRQ: Verification FAILED");
25 }else{pass++;}
26 }
```

```
if(pass == 30){
  SC_REPORT_INFO("NVIC Verification","EnableIRQ: verification PASSED");

}
}

void Software::NVIC_verification_DisableIRQ(void){
int irq = 0;
uint32_t reg;
int pass = 0;

for (irq; irq<30;irq++){
cpu_ptr->NVIC_DisableIRQ(irq);
reg = cpu_ptr->cpu_reg_read(cpu_ptr->NVICregs.nvicICER[0]);
if(!(reg & 1 << irq)){
  SC_REPORT_INFO("NVIC Verification","DiableIRQ: Verification FAILED");

}else{pass++;}
}
if(pass == 30){
  SC_REPORT_INFO("NVIC Verification","DiableIRQ: Verification PASSED");

}
}

void Software::NVIC_verification_ClearPendingIRQ(void){
int irq = 0;
uint32_t reg;
int pass = 0;

for (irq; irq<30;irq++){
cpu_ptr->NVIC_ClearPendingIRQ(irq);
reg = cpu_ptr->cpu_reg_read(cpu_ptr->NVICregs.nvicICPR[0]);
// std::cout <<std::bitset<32>(reg & 1 << irq) << '\n';
if(!(reg & 1 << irq)){
  SC_REPORT_INFO("NVIC Verification","ClearPendingIRQ: Verification FAILED"
    );
}else{pass++;}
```

```
64 }
65 // std::cout << pass << '\n';
66 if(pass == 30){
67   SC_REPORT_INFO("NVIC Verification","ClearPendingIRQ: Verification PASSED"
       );
68
69 }
70 }
71
72 void Software::NVIC_verification_GetEnable(void){
73   uint32_t read;
74   int pass;
75     uint32_t i=0;
76   for(int irq = 0;irq<30;irq++){
77 if(i) cpu_ptr->NVIC_EnableIRQ(irq);
78   read = cpu_ptr->NVIC_GetEnableIRQ(irq);
79   std::cout << "/* message */"<< i << '\n';
80   if(read == i) pass++;
81   i = !i;
82 }
83 if(pass == 30){
84   SC_REPORT_INFO("NVIC","GetEnable Verification PASSED");
85 }else{
86   std::cout << pass <<" tests passed out of 30" << '\n';
87   SC_REPORT_ERROR("NVIC","GetEnable Verification FAILED");
88 }
89 }
90
91 void Software::NVIC_verification_GetPending(void){
92   uint32_t read;
93   int pass;
94   int i = 0;
95   for(int irq = 0;irq<30;irq++){
96
97   cpu_ptr->NVICregs.nvicISPR[((uint32_t)(irq) >> 5)] |= i << ((uint32_t)(
       irq) & 0x1F);
98
99   read = cpu_ptr->NVIC_GetPendingIRQ(irq);
```

```
100   if(read == i) pass++;
101   i = !i;
102 }
103 if(pass == 30){
104   SC_REPORT_INFO("NVIC","GetPending Verification PASSED");
105 }else{
106   std::cout << pass <<" tests passed out of 30" << '\n';
107   SC_REPORT_ERROR("NVIC","GetPending Verification FAILED");
108 }
109 }
```

B.5.2 GPIO verification tests

```
1  #include "sw_main.h"
2  #include "cpu.h"
3  #include <bitset>
4  #include "addresses.h"
5  // #include "scv.h"
6
7  void Software::sw_main(void){
8
9      int data_in = 0 ;
10
11     //reset OUT register before the test
12      cpu_ptr->apb_simple_write(NRF_GPIO_OUTSET_address, data_in);
13      cpu_ptr->delay_us(5);
14
15     individual_bit_SET_CLR_TEST();
16
17
18     bits_SET_then_CLR_TEST();
19
20 }
21
22 /* This test goal is to verify that each individual bit that has been set
      in OUTSET
```

```
* register is actuall sit in the OUT register, and each bit that has been
    set
* in OUTCLR register is cleared in the OUT register.
*/
void Software::individual_bit_SET_CLR_TEST(void){
  int data_in = 0;
  int data_out;
  int i;
  int set_pass = 0;
  int clr_pass = 0;

  for (i = 0;i<32;i++){
    data_in = 0;
    data_in = data_in |  (1 << i) ;

    cpu_ptr->apb_simple_write(NRF_GPIO_OUTSET_address, data_in);
    cpu_ptr->delay_us(5);

    cpu_ptr->apb_simple_read(NRF_GPIO_OUT_address, data_out);
    cpu_ptr->delay_us(5);


    if (data_out & (1 << i) ){
      set_pass++;
    }

    cpu_ptr->apb_simple_write(NRF_GPIO_OUTCLR_address, data_in);
    cpu_ptr->delay_us(5);

    cpu_ptr->apb_simple_read(NRF_GPIO_OUT_address, data_out);
    cpu_ptr->delay_us(5);

    // std::cout <<std::bitset<32> (data_out )<< '\n';

    if (!(data_out & (1 << i)) ){
      clr_pass++;
    }
}
```

```
60
61
62 if(set_pass == i){
63   SC_REPORT_INFO("GPIO verification","Indvidual bits SETOUT verification
       PASSED");
64 }else{
65   SC_REPORT_INFO("GPIO verification","Indvidual bits SETOUT verification
       FAILED");
66
67 }
68
69 if(clr_pass == i){
70   SC_REPORT_INFO("GPIO verification","Indvidual bits CLROUT verification
       PASSED");
71 }else{
72   SC_REPORT_INFO("GPIO verification","Indvidual bits CLROUT verification
       FAILED");
73
74 }
75 }
76
77
78 /* The goal of this tes is to verify that setting a bit in OUTSET
79 *register would only affect the corrosponding bit on OUT registers
80 * leaving the rest of the bits untouched, and the same for CLROUT
81 */
82 void Software::bits_SET_then_CLR_TEST(void){
83   int data_in = 0;
84   int data_out;
85   int i;
86   int set_pass = 0;
87   int clr_pass = 0;
88   int t;
89   uint32_t total_set_pass;
90   uint32_t total_clr_pass;
91
92       data_in = 0;
93
```

```
  for (i = 0;i<=31;i++){
    data_in = 0;
    data_in = data_in |  (1 << i) ;

    cpu_ptr->apb_simple_write(NRF_GPIO_OUTSET_address, data_in);
    cpu_ptr->delay_us(5);

    cpu_ptr->apb_simple_read(NRF_GPIO_OUT_address, data_out);
    cpu_ptr->delay_us(5);
    std::cout << std::bitset<32>(data_in) << '\n';
    std::cout << std::bitset<32>(data_out) << '\n';

    if (data_out & (1 << i) == data_in ){
      set_pass++;
    }

  }
std::cout << set_pass << '\n';
  if(set_pass==32){
    total_set_pass = 1;
 }else{
total_set_pass = 0;
}


      data_in = 0;

  for (i = 0;i<=31;i++){

    data_in = data_in |  (1 << i) ;

    cpu_ptr->apb_simple_write(NRF_GPIO_OUTCLR_address, data_in);
    cpu_ptr->delay_us(5);

    cpu_ptr->apb_simple_read(NRF_GPIO_OUT_address, data_out);
    cpu_ptr->delay_us(5);

    if (!(data_out & (1 << i)) ){
```

```
132       clr_pass++;
133     }
134  }
135
136
137       if(clr_pass==32){
138         total_clr_pass = 1;
139
140  }else{ total_clr_pass = 0;
141 }
142
143 if(total_set_pass){
144
145   SC_REPORT_INFO("GPIO verification","accemulative SETOUT verification
      PASSED");
146 }else{       SC_REPORT_INFO("GPIO verification","accemulative SETOUT
      verification FAILED");
147 }
148
149 if(total_clr_pass){
150   SC_REPORT_INFO("GPIO verification","accemulative CLROUT verification
      PASSED");
151 }else{
152 SC_REPORT_INFO("GPIO verification","accemulative CLROUT verification FAILED
      ");
153 }
154
155 }
```

B.5.3 CLOCK verification tests

```
1 #include "sw_main.h"
2 #include "cpu.h"
3 #include <bitset>
4 #include "addresses.h"
5
6
```

```
void Software::sw_main(void){

 HFCLK_verification();
 LFCLK_verification();
LFCLK_calibration_verification();


}


  void Software::HFCLK_verification(void){

    int data_in;
    int data_out;

    //check that the clock does not start when
    // the HFCLK task hasn't triggered
    //data_in = 0x0 ;
    //cpu_ptr->apb_simple_write(TASKS_HFCLKSTART_address, data_in);
    //cpu_ptr->delay_us(5);

    cpu_ptr->apb_simple_read(NRF_CLOCK_HFCLKSTAT_address, data_out);
    cpu_ptr->delay_us(5);

    if(data_out & 0x10000){
    std::cout << "Error: the HFCLK has not started yet";
    }

    // check that HFCLKSTARTED event has not been triggered
     cpu_ptr->apb_simple_read(NRF_CLOCK_EVENTS_HFCLKSTARTED_address,
   data_out);
     cpu_ptr->delay_us(5);

     if(data_out & 0x1){
    std::cout << "Error: the HFCLK has not started yet";
    }

    //check that the state and the source change
```

```
44    // when the HFCLK task is triggered
45    data_in = 0x1 ;
46    cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_HFCLKSTART_address, data_in);
47    cpu_ptr->delay_us(5);
48
49    cpu_ptr->apb_simple_read(NRF_CLOCK_HFCLKSTAT_address, data_out);
50    cpu_ptr->delay_us(5);
51
52    if(!(data_out & 0x10000) | !(data_out & 0x1)){
53    std::cout << "Error: the HFCLK has started already";
54    }
55
56    //check that HFCLKSTARTED event has been generated
57     cpu_ptr->apb_simple_read(NRF_CLOCK_EVENTS_HFCLKSTARTED_address,
    data_out);
58     cpu_ptr->delay_us(5);
59
60     if(!(data_out & 0x1)){
61    std::cout << "Error: the HFCLK has started already";
62    }
63
64    //check that when the HFCLKSTOP task triggered
65    // the state of the HFCLK will stop
66     data_in = 0x0;
67     cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_HFCLKSTOP_address, data_in);
68     cpu_ptr->delay_us(5);
69
70     cpu_ptr->apb_simple_read(NRF_CLOCK_HFCLKSTAT_address, data_out);
71     cpu_ptr->delay_us(5);
72
73     if (!(data_out & 0x1)){
74        std::cout << "Error: the HFCLK has not been stopped yet";
75     }
76
77
78    //check that the state of the FCLK will go back to zero
79    //after stopping it it
80
```

```
    data_in = 0x1;
    cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_HFCLKSTOP_address, data_in);
    cpu_ptr->delay_us(5);


    cpu_ptr->apb_simple_read(NRF_CLOCK_HFCLKSTAT_address, data_out);
    cpu_ptr->delay_us(5);

    if (data_out & 0x1){
       std::cout << "Error: the HFCLK has been stopped ";
    }
}

void Software::LFCLK_verification(void){

  int data_in;
  int data_out;

  //check the registers before LFCLKSTART is triggered
  cpu_ptr->apb_simple_read(NRF_CLOCK_LFCLKSTAT_address, data_out);
  cpu_ptr->delay_us(5);

  if(data_out & 0x10000){
  std::cout <<"Error: LFCLKSTART has not been triggered";
  }
    if(data_out & 0x3){
  std::cout <<"Error: LFCLKSRC is not RC";
  }

  //start the LFCLK with RC as source
  data_in = 1;
  cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_LFCLKSTART_address,data_in);
    cpu_ptr->delay_us(5);

  cpu_ptr->apb_simple_read(NRF_CLOCK_LFCLKSTAT_address, data_out);
  cpu_ptr->delay_us(5);

    if(!(data_out & 0x10000)){
```

```
119     std::cout <<"Error: LFCLKSTART has been triggered";
120     }
121       if((data_out & 0x3)!= 0x0){
122     std::cout <<"Error: LFCLKSRC is not RC";
123     }
124
125     //stop LFCLK
126     data_in = 1;
127     cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_LFCLKSTOP_address,data_in);
128       cpu_ptr->delay_us(5);
129
130     cpu_ptr->apb_simple_read(NRF_CLOCK_LFCLKSTAT_address, data_out);
131     cpu_ptr->delay_us(5);
132
133       if(data_out & 0x10000){
134     std::cout <<"Error: LFCLKSTART has been stopped";
135     }
136
137     //set the SRC as external oscillator
138     data_in = 0x1;
139     cpu_ptr->apb_simple_write(NRF_CLOCK_LFCLKSRC_address,data_in);
140       cpu_ptr->delay_us(5);
141
142     cpu_ptr->apb_simple_read(NRF_CLOCK_LFCLKSTAT_address, data_out);
143     cpu_ptr->delay_us(5);
144
145       if((data_out & 0x3)!= 0x1){
146     std::cout <<"Error: LFCLKSTART is not external oscillator";
147     }
148
149     //start the LFCLK with external oscillator as source
150     data_in = 1;
151     cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_LFCLKSTART_address,data_in);
152       cpu_ptr->delay_us(5);
153
154     cpu_ptr->apb_simple_read(NRF_CLOCK_LFCLKSTAT_address, data_out);
155     cpu_ptr->delay_us(5);
156
```

```
157       if(!(data_out & 0x10000)){
158     std::cout <<"Error: LFCLKSTART has been triggered";
159     }
160
161     //check that LFCLKSTARTED event has been generated
162       cpu_ptr->apb_simple_read(NRF_CLOCK_EVENTS_LFCLKSTARTED_address,
    data_out);
163     cpu_ptr->delay_us(5);
164
165           if(!(data_out & 0x1)){
166     std::cout <<"Error: LFCLKSTART has been triggered";
167     }
168
169
170   }
171
172   void Software::LFCLK_calibration_verification(void){
173
174     int data_in;
175     int data_out;
176
177     //clear the HFCLKSTARTED register
178     data_in = 0x0;
179     cpu_ptr->apb_simple_write(NRF_CLOCK_EVENTS_HFCLKSTARTED_address,
    data_in);
180     cpu_ptr->delay_us(5);
181
182
183     //configure the calibration timer interval
184     data_in = 16;
185     cpu_ptr->apb_simple_write(NRF_CLOCK_CTIV_address, data_in);
186     cpu_ptr->delay_us(5);
187
188     //start CT
189     data_in = 0x1;
190     cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_CTSTART_address,data_in);
191       cpu_ptr->delay_us(5);
192
```

```
193     cpu_ptr->apb_simple_read(NRF_CLOCK_CTIV_address,data_out);
194     cpu_ptr->delay_us(5);
195 
196     while(data_out != 0){
197       cpu_ptr->apb_simple_read(NRF_CLOCK_CTIV_address,data_out);
198       cpu_ptr->delay_us(5);
199     }
200 
201     //read the CTTO
202     cpu_ptr->apb_simple_read(NRF_CLOCK_EVENTS_CTTO_address, data_out);
203     cpu_ptr->delay_us(5);
204 
205     if(data_out!= 1){
206       SC_REPORT_ERROR("nrf_clock","The timer has not reached zero");
207     }
208 
209     //start HFCLK
210     data_in = 0x1;
211     cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_HFCLKSTART_address,data_in);
212       cpu_ptr->delay_us(5);
213 
214     //starting the calibration process
215     data_in = 0x1;
216     cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_CAL_address,data_in);
217       cpu_ptr->delay_us(5);
218 
219     //check that calibration DONE event has been generated
220     cpu_ptr->apb_simple_read(NRF_CLOCK_EVENTS_DONE_address, data_out);
221     cpu_ptr->delay_us(5);
222 
223     if(!(data_out & 0x1)){
224       SC_REPORT_ERROR("nrf_clock","DONE event has not been generated");
225     }
226 
227   }
```

B.5.4 System verification tests

```
#include "sw_main.h"
#include "cpu.h"
#include <bitset>
#include "addresses.h"


void Software::sw_main(void){

  system_verification();
}


/*This verification will work as follow: it will turn on LFCLK with external oscillator
* as a source which will generate an interrupt the interrupt will be deteted by the NVIC
* and it would set some bits in the GPIO module.
*/
  void Software::system_verification(void){
  int data_in;
  int data_out;

  //enable interrupt in the NVIC_ClearPendingIRQ
  cpu_ptr->NVIC_EnableIRQ(0x0);

  //enable the interrupt for EVENTS_LFCLKSTARTED
  cpu_ptr->delay_us(5);
  data_in = 0x2;
  cpu_ptr->apb_simple_write(NRF_CLOCK_INTENSET_address, data_in);

  // choose the LFCLK source
  cpu_ptr->delay_us(5);
  data_in = 0x1;
  cpu_ptr->apb_simple_write(NRF_CLOCK_LFCLKSRC_address, data_in);

  // start the LFCLK
```

```
  cpu_ptr->delay_us(5);
  data_in = 0x1;
  cpu_ptr->apb_simple_write(NRF_CLOCK_TASKS_LFCLKSTART_address, data_in);

  //read OUT register
  cpu_ptr->delay_us(5);
  cpu_ptr->apb_simple_read(NRF_GPIO_OUT_address, data_out);

if (data_out == 0x10){
  SC_REPORT_INFO("system verification","system verification PASSED");
}else{
  SC_REPORT_INFO("system verification","system verification FAILED");
} }
```

B.6 The base test code

```
        void fun_mpsl_init_001(Cpu* cpu_ptr){
  ASSERT_EQ(mpsl_init(cpu_ptr),0);

  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(POWER_CLOCK_IRQn));
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(RTC0_IRQn));
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(MPSL_TEST_IRQn));

  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(TIMER0_IRQn) == 0);
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(RADIO_IRQn) == 0);
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(TEMP_IRQn) == 0);

  mpsl_uninit(cpu_ptr);

  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(POWER_CLOCK_IRQn) == 0);
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(RTC0_IRQn) == 0);
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(TIMER0_IRQn) == 0);
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(RADIO_IRQn) == 0);
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(TEMP_IRQn) == 0);
  ASSERT(cpu_ptr->NVIC_GetEnableIRQ(MPSL_TEST_IRQn) == 0);
}

```

Bibliography

[1] David Barahona. Virtualization of low-power digital architectures in systemc, 2020.

[2] David C Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the ground up*, volume 71. Springer Science & Business Media, 2009.

[3] L. Cai and D. Gajski. Transaction level modeling: an overview. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, pages 19–24, Oct 2003.

[4] Tom De Schutter. *Better Software. Faster! Best Practices in Virtual Prototyping*. Synopsys Press, Mountain View, CA, USA, 2014.

[5] Frank Ghenassia et al. *Transaction-level modeling with SystemC*, volume 2. Springer, 2005.

[6] Paula Herber and Bettina Hünnemeyer. Formal verification of systemc designs using the blast software model checker. volume 1250, 09 2014.

[7] H. Qian and C. Zheng. A embedded software testing process model. In *2009 International Conference on Computational Intelligence and Software Engineering*, pages 1–5, Dec 2009.

[8] Nordic Semiconductor. Bauhaus, internal document, June 2020. Available: `https://projecttools.nordicsemi.no/bitbucket/projects/ESL/repos/bauhaus/browse`.

[9] Nordic Semiconductor. Clock — clock control, February 2020. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52832.ps.v1.1%2Fclock.html&cp=4_2_0_18&anchor=frontpage_clock.

[10] Nordic Semiconductor. Rtc — real-time counter, February 2020. Available: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52832.ps.v1.1%2Frtc.html&cp=4_2_0_24&anchor=concept_rvn_vkj_sr.

[11] L. Shuping and P. Ling. The research of v model in testing embedded software. In *2008 International Conference on Computer Science and Information Technology*, pages 463–466, Aug 2008.

[12] Moshe Vardi. Formal techniques for systemc verification. pages 188–192, 01 2007.