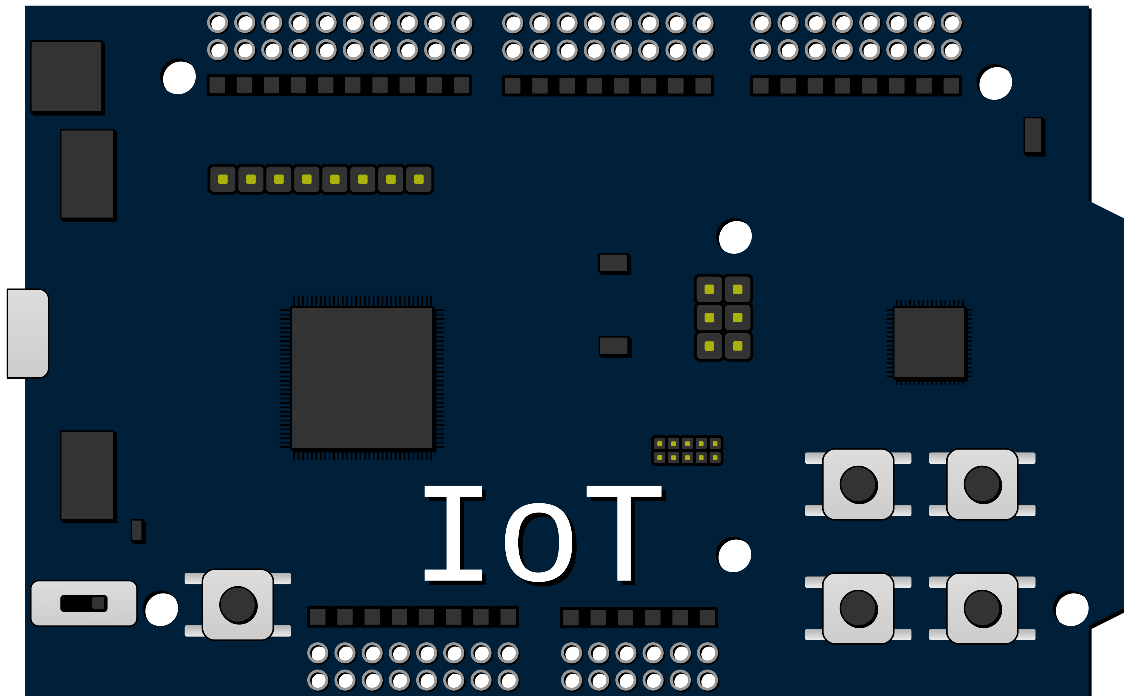
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG




# Emulating the Internet of Things with QEMU

Master's thesis in the Computer Science Programme

Gyokan O. Osman

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

# Emulating the Internet of Things with QEMU

GYOKAN O. OSMAN

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Emulating the Internet of Things with QEMU
GYOKAN O. OSMAN

© GYOKAN O. OSMAN, 2019.

Supervisor: Olaf Landsiedel, Department of Computer Science and Engineering
Examiner: Magnus Almgren, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Sketch of Nordic Semiconductor nRF51-DK. © Gyokan O. Osman, 2019

Typeset in LaTeX
Gothenburg, Sweden 2019

Emulating the Internet of Things with QEMU
GYOKAN O. OSMAN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

This thesis project implements and evaluates the virtual nRF51 platform in QEMU. The purpose of the project is to be able to run nRF51 binaries in QEMU and find out whether it is feasible to perform a full system emulation for IoT devices. The nRF51 platform is a wireless system on chip design with ARM® Cortex™ M0 from Nordic Semiconductor. QEMU already supports the ARM architecture.

Embedded application development comes with its own limitations. Debugging is much harder and usually requires additional hardware. In most cases, it is necessary to have a network of devices when working with IoT. Platform emulation is one of the most convenient ways to overcome the limitations mentioned above.

There are not many open–source IoT emulation projects with complete hardware support. Usually, hardware emulation is provided in the software layer, or the embedded program is compiled and run on the desktop platform. This thesis project provides true peripheral emulation direct binary execution. Therefore one of the most challenging aspects was to understand, implement, and evaluate the hardware behavior under variable conditions.

It was only required to provide hardware emulation for nRF51 peripherals. The communication between peripherals is provided using UNIX sockets and UDP for a simpler implementation. Therefore all the evaluation is aimed at testing the performance and the functionality by comparing results against the physical hardware.

The evaluation was done under two main categories, namely the performance and the functionality. Open–source projects such as the mbed library, Zephyr and the nRF51 SDK are used during evaluation. Emulated nRF51 programs run much faster than the hardware on an average desktop computer. As for the functionality, most applications show the expected behavior when they don't depend on the actual hardware timings. In some cases, faster execution of the instructions or insufficient timer accuracy can change the program behavior. Evaluation results show that QEMU can run nRF51 programs as stable as the hardware except for the execution timings, and system timers. The results also show that it is possible to run real–time operating systems in an emulated environment.

Most desktop platforms have timers with higher resolutions but context switches and delays caused by the other host tasks can introduce time drifts in the guest. There is no correlation between the guest CPU cycle timings and the host system timers. From that perspective, it requires more work to truly emulate CPU features with correct timings such as caching, fetching and reading operations. QEMU might need changes or a different emulation mode for this purpose.

It is possible to replace UDP communication with a more reliable, high–performance interface. Power consumption is of crucial importance in IoT environments. Power statistics can be implemented based on the peripheral state and the number of CPU cycles with some effort.

Keywords: nRF51, Nordic Semiconductor, QEMU, emulation, simulation, Bluetooth, GPIO, virtualization.

# Acknowledgements

Many thanks to the Department of Computer Science at the University of Gothenburg and the academic staff. It has been a privilege to study here and this helped me to gain a new perspective on the life besides the scientific knowledge.

I am also grateful to my supervisor, Olaf Landsiedel, not only for his support in my thesis but also keeping the morale high at all times.

Lastly I would like to thank to my examiner Magnus Almgren for his constructive feedback.

Gyokan O. Osman, Gothenburg, September 2019

# Contents

**List of Figures** **xiii**

**1 Introduction** **1**
1.1 Problem Statement . . . 2
1.2 Purpose . . . 2
1.3 Thesis Outline . . . 3

**2 Background** **5**
2.1 Internet of Things . . . 5
2.2 QEMU . . . 6
2.3 The nRF51 Platform . . . 7
2.4 Terminology and QEMU Internals . . . 7
2.4.1 Terminology . . . 7
2.4.2 QEMU Internals . . . 8
2.5 Discussion . . . 8
2.5.1 Development Process . . . 8
2.5.2 Strengths and Weaknesses of a Virtual Environment . . . 9

**3 Related Work** **11**
3.1 QEMU . . . 11
3.1.1 Existing Implementations . . . 11
3.1.2 Discussion . . . 12
3.2 TriCore Simulator . . . 12
3.2.1 Details . . . 12
3.2.2 Discussion . . . 12
3.3 Avrora . . . 13
3.3.1 Details . . . 13
3.3.2 Discussion . . . 13
3.4 Cooja . . . 13
3.4.1 Details . . . 13
3.4.2 Discussion . . . 14
3.5 TINA . . . 14
3.5.1 Details . . . 14
3.5.2 Discussion . . . 14
3.6 ACRN . . . 14
3.6.1 Details . . . 14

3.6.2 Discussion . . . . . 15
3.7 OMNeT++ . . . . . 15
3.7.1 Details . . . . . 15
3.7.2 Discussion . . . . . 15

**4 Design 17**
4.1 Goal . . . . . 17
4.2 Approach . . . . . 17
4.3 Design Details . . . . . 18
4.3.1 General Purpose Input/Output (GPIO) . . . . . 18
4.3.2 GPIO Tasks and Events (GPIOTE) . . . . . 18
4.3.3 Universal Asynchronous RX/TX (UART) . . . . . 19
4.3.4 AES Electronic Codebook Mode (ECB) . . . . . 19
4.3.5 2.4 GHz Radio (RADIO) . . . . . 20
4.3.6 Analog to Digital Converter (ADC) . . . . . 21
4.3.7 Clock Management (CLOCK) . . . . . 21
4.3.8 Random Number Generator (RNG) . . . . . 21
4.3.9 Real Time Counter (RTC) . . . . . 21
4.4 Guest-to-Guest Communication . . . . . 22
4.4.1 Router Script Configuration . . . . . 23
4.5 Discussion . . . . . 23

**5 Implementation 27**
5.1 Overview . . . . . 27
5.2 QEMU Peripheral Interface . . . . . 28
5.3 Discussion . . . . . 29

**6 Evaluation 31**
6.1 Evaluation Method . . . . . 31
6.1.1 Metrics . . . . . 31
6.1.2 Performance Tests . . . . . 32
6.1.3 Functional Tests . . . . . 32
6.1.4 Hardware Setup . . . . . 33
6.1.5 Experiments . . . . . 33
6.2 Results . . . . . 36
6.2.1 Exp1: GPIO and RTC Test with LEDs . . . . . 36
6.2.2 Exp2: UART Test . . . . . 37
6.2.3 Exp3: AES Performance . . . . . 38
6.2.4 Exp4: Raw Execution Time . . . . . 39
6.2.5 Exp5: Simple GPIO using mbed Library . . . . . 40
6.2.6 Exp6: Blinky LED Example from Zephyr . . . . . 40
6.2.7 Exp7: CPP Synchronization Example from Zephyr . . . . . 41
6.2.8 Exp8: Entropy Example from Zephyr . . . . . 41
6.2.9 Exp9: Fibonacci Benchmark Test & Analysis . . . . . 42
6.2.10 Exp10: RADIO Test . . . . . 43
6.2.11 Exp11: RADIO Test with Zephyr Bluetooth Stack . . . . . 44
6.3 QEMU Resource Usage . . . . . 46

6.4 Hardware and Virtual Environment Comparison . . . . . . . . . . . . 46
6.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

**7 Conclusion and Future Work 49**
7.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
7.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
7.3 Ethics and Sustainability . . . . . . . . . . . . . . . . . . . . . . . . 50

**Bibliography 51**

**A Test Code I**

# List of Figures


4.1 RADIO States . . . . . 20
4.2 Generic UDP packet . . . . . 23
4.3 Send Device Identifier packet . . . . . 23
4.4 RADIO packet with type 0x1 . . . . . 24
4.5 GPIO packet with type 0x2 . . . . . 24

5.1 nRF51 Code, SRAM and Peripheral Area . . . . . 28
5.2 R/W Callbacks in QEMU . . . . . 29

6.1 AES Throughput . . . . . 38
6.2 Number of AES Rounds in 5 seconds . . . . . 38
6.3 Number of Empty Loops in 1 Second . . . . . 39

# 1 Introduction

The importance of virtualization has been on the rise in the last decade. There are many specific reasons why people choose it over real hardware. Mainly this technique is used to gain physical space, cut costs, or reduce the time when deploying in the industry [1].

Many companies and people around the world use virtualization for different purposes. These purposes can be specific to the field. Cloud computing, testing, environment isolation, network simulation, and penetration testing are some solid examples. Virtualization makes it possible to share a single physical system by many other users in an isolated environment. Dozens of users can share mainframe computers at the same time without knowing the actual infrastructure. Most popular companies like Google and Amazon depend on this technology with some of their services. Some other companies provide a type of service known as VPS or "virtual private server" using virtualization. VPS services allow people to publish their websites, set up private Git servers, and create a testing environment with minimal costs. The testing environment can be on the cloud or a personal computer. With the help of some tools, a new virtual environment can be deployed in seconds to evaluate a specific part of a software package.

There are more extreme cases where virtualization brings a great advantage when it is costly to have access to the target environment. For example, it is hard to perform a full test for a program that runs on an embedded system on personal computers without the help of this technology.

In the automotive industry, most systems run on PowerPC and the ARM platform. Developers that work with entertainment and information systems for cars make use of virtual machines. They usually run the Android OS on a virtual ARM microprocessor using their computers and perform all the testing and development activities there. There are also tools that can run PowerPC binaries that allow developers to run unit tests without needing a car or a specific unit from the vehicle.

Well known virtual machine host applications include Microsoft Hyper–V, Oracle VM VirtualBox, VMware and QEMU (Quick Emulator). In this project, we focus on QEMU for its ability to emulate applications written for different architectures [2]. It is an open–source project and is backed up by an active community.

The evaluation was done for performance and functionality using open–source real–time OS and libraries also an open–source Bluetooth stack. It involves running a variety of test applications with specific purposes under different conditions.

## 1.1 Problem Statement

Embedded software development differs from desktop software development in many ways. The key differences are:

- Debugging tools
- Runtime environment
- Hardware
- Toolchains

Each of the differences listed above introduces a limitation of its own. Notably, testing and debugging steps can be quite challenging. Setting up the environment for development can be time-consuming as well. Some applications may require multiple hardware units or special equipment. Emulating a development platform can remove all these limitations. In addition to that, modern emulator and virtual machine applications can save and restore machine states. This functionality can help to debug bugs that are hard to reproduce.

The nRF51 platform is quite popular among the students and the embedded developers. It comes with a Bluetooth module, and it usually requires multiple development boards or a device network when testing embedded applications that make use of Bluetooth technology. In most cases, companies can afford it but running a network of devices can be costly for students and hobbyists. Live software debugging in an active network is out of the question most of the time.

## 1.2 Purpose

The purpose of this project is evaluating an extension for QEMU to simulate systems and peripherals that exist in an embedded environment. For example, development boards are usually used in the prototyping phase. They often come with onboard wireless interfaces such as Bluetooth, Wi–Fi, or other low–frequency transceivers. More advanced boards can also have built–in CAN or Ethernet interfaces. This project aims to remove the need of having physical devices and provide a fully working virtualized development and debugging environment for students and embedded developers. Debugging often requires additional hardware and it may not be accessible by everyone. Wireless applications are usually tested within a device network, in that case it may be difficult to obtain the necessary hardware whereas emulated devices are only limited by the host resources.

QEMU is the base for this project and it already provides virtualization for well–known architectures including the ARM platform [3] but it does not provide emulation specifically for IoT purposes.

For example, Raspbian, a Debian port for RaspberryPi, can be run under QEMU but it will not provide simulation for peripherals. There are already extensions that can provide basic functionality for GPIO, real–time clock or ADC. The most significant difference between existing implementations and this project is that the main focus here is to fully virtualize a specific development board, including advanced peripherals such as wireless communication interfaces and sensors.

Since QEMU is an open–source project, this extension will contribute to the open–

source community, universities, and IT companies that focus on embedded systems.

This project can also help the hardware design. It is not in the scope of this project, but it will be possible to integrate specific peripherals into another architecture or CPU model.

Expected benefits of this project are as follows:

- Reduce costs: One may want to run a network of devices in order to test and debug. Students can learn and develop without buying real hardware.
- Provide determinism: Since sensors, wireless networks, and similar peripherals do not show a fully deterministic behavior, reproducing a bug every time might be challenging and time–consuming.
- Make debugging easier: QEMU already provides access to registers and memory. That functionality can be used for IoT devices as well.

## 1.3 Thesis Outline

The thesis consists of 7 chapters, including the introduction. Chapter 2 gives brief information about the emulated target platform and the QEMU project, including some technical details. Chapter 3 presents and compares the related work. Chapter 4 describes the design goals and the approach for this project. In addition to that, the design details are given for each peripheral emulated. Chapter 5 presents the implementation details and the concept behind it. Chapter 6 presents the test setup and evaluation results. Chapter 7 summarizes the project and discusses future work.

# 2
# Background

This chapter is divided into 5 sections each of them explaining the concepts related to the project.

## 2.1 Internet of Things

The Internet of Things (IoT) is the concept of connecting electronic devices using a communication mechanism to allow interaction and data transfer in between. It is also referred to as the network of smart devices. Almost every electronic device and gadget we see around today is part of the Internet of Things. There are countless examples of such devices. Some of them are smart televisions, Bluetooth speakers, radio–controlled light bulbs, shirts that report workout data, refrigerators, washing machines, and cellphones [4]. Such devices are usually built around a microcontroller, and communication is provided through a wired or low powered wireless interface.

The IoT devices have a variety of uses. With the advantage of cost and small scale architecture, IoT devices gained popularity increasingly. A decade ago, small devices that accomplish a single task were usually designed and sold by companies for commercial purposes. In today's world, most home users have access to open source hardware and software. They can build their own IoT devices. One such an example is the IoT based carbon dioxide monitoring system [5] which also uses cloud technologies for real–time monitoring and data analysis. The research makes use of a well–known IoT product ESP8266 Wi–Fi module [6].

There are other well–known open source hardware designs such as RaspberryPi, Arduino, and Beagleboard. These projects contributed to the IoT in many ways, but mostly they provided low–cost hardware to the people, and that helped people to build their own smart devices. As a result of this, the number of IoT devices is increasing every day, faster than ever before. These development boards mentioned above are widely used in IoT projects. They are usually classified into two groups:

- Small single–board computers: In addition to the CPU, these devices usually have a GPU module, monitor output and a large RAM ranging from a few hundred megabytes to several gigabytes. Such examples are Beagleboard and RaspberryPi.
- Single–board microcontrollers: These devices are usually suitable for applications requiring low power consumption. They are built around a smaller and slower microcontroller compared to the first group. Some configurations might include Bluetooth or Wi-Fi interface.

The nRF51 platform falls into the single board microcontroller category. It is energy efficient and has a Bluetooth interface, which makes it very suitable for wireless applications. The nRF51 platform is introduced verbosely in a separate section.

## 2.2 QEMU

Most hypervisor applications are limited by the platform they run on because of the performance considerations. In general, a virtual machine runs on the host CPU, but it is completely isolated from the host operating system. QEMU [2] can simulate the CPU architectures that are different from the host CPU. Simulation is accomplished by translating the guest CPU instructions into the host CPU instructions, and the translation is provided by the "Tiny Code Generator" or TCG.

Only emulating the guest instructions alone does not provide a full emulation for the nRF51 platform. During runtime, guest applications configure the required peripherals using peripheral registers on the embedded device, and each register is mapped to a specific memory location. QEMU can intercept memory read and write requests and pass them to the callbacks. These callbacks are designed and implemented as part of this project in order to simulate the system peripherals. Based on the request from the application, an IRQ can be generated, or some device registers can be modified to provide feedback to the application.

There are many official and unofficial virtual machine implementations for QEMU, but the lack of documentation is still an obstacle for new developers. The official QEMU getting started guide for developers states the following: "QEMU does not have a high–level design description document - only the source code tells the full story" [7]. The best way to understand how a virtual machine works, is by browsing through the source code and studying patches submitted by other developers. Some information regarding virtual machine development can be found around the internet as a starting point. There are mailing lists and blogs written by the QEMU developers.

In the IoT world, QEMU is widely used for security research and testing. Usually, that requires some additional implementation and extensions. The Linux based operating systems such as OpenWRT have good support in QEMU, and it can be used for malware analysis for IoT devices [8]. Some other tools are also based on QEMU. For example, NEMU [9] is a test framework used for anomaly detection on emulated devices. There other use cases for QEMU as well. In addition to device and peripheral emulation, QEMU can be used for fault injection [10] in order to analyze embedded software.

As already mentioned, QEMU is the base for this project and it already provides virtualization for well–known architectures including the ARM platform [3] but it does not provide emulation specifically for IoT purposes.

For example, Raspbian, a Debian port for RaspberryPi, can be run under QEMU but it will not provide simulation for peripherals. There are already extensions that can provide basic functionality for GPIO, real–time clock or ADC. The most significant difference between existing implementations and this project is that the main focus here is to fully virtualize a specific development board, including advanced

peripherals such as wireless communication interfaces and sensors. In addition to that, it must be indistinguishable from the real environment when executing the application. By accomplishing that, this project will bring existing virtualization techniques one level higher for IoT devices. The target SoC[1] platform for this project is the nRF51822 development board by Nordic Semiconductor [11]. It comes with many peripherals that are vital for most embedded applications. These peripherals include 2.4 GHz Bluetooth–capable radio, real–time counter, AES crypto accelerator, SPI, $I^2C$, and UART.

This project can also help the hardware design. It is not in the scope of this project, but it will be possible to integrate specific peripherals into another architecture or CPU model. In that way, one can configure a custom virtual board for a specific purpose, duplicate peripherals, remove unwanted ones then do a specific performance and functionality test without having a physical prototype. The machine extension was designed with portability in mind. For example, it is possible to increase the number of most peripherals and configure the RAM size. QEMU has already been previously used for virtual prototyping of embedded systems [12].

## 2.3 The nRF51 Platform

The nRF51 board is an embedded platform; for that reason, it does not require all the advanced features that a desktop platform may demand. For example, it does not have video and sound output, and it does not have a USB peripheral or PCI peripheral. The software controls the platform through device registers. Based on the preliminary study, it requires the following vital features:

- Memory–mapped I/O: Provides access to the device registers.
- Interrupt Requests: Most peripherals generate interrupts for various purposes.
- Cortex-M0 compatible CPU: The nRF51 platform runs on Cortex-M0. It is not directly supported, but Cortex-M3 can be used for its compatibility.
- Accurate Periodic Timers: This will be used for RTC and TIMER modules.

It is expected for QEMU to be able to run the programs developed for the nRF51 platform and show the exact behavior of the physical board without any modifications to the source code.

## 2.4 Terminology and QEMU Internals

### 2.4.1 Terminology

The following terminology is used:

- Peripheral: A unit that is designed to perform a specific task; it can be on board or embedded in the CPU. The word ‘device’ is used interchangeably.
- Guest: The emulated virtual machine. Depending on the context, it can mean the sequence of instructions being executed.
- Register: A device register. CPU registers are referred to as ‘CPU register’. Each device register is 4 bytes wide.

[1] System-On-Chip

Peripheral names are written with all capital letters in the nRF51 reference manual. This convention is followed in this report as well.

### 2.4.2 QEMU Internals

QEMU currently supports many types of machines working on well–known architectures such as ARM, x86, MIPS, and PPC. The nRF51 platform runs on the ARM Cortex-M0 processor. QEMU does not have support for Cortex-M0, but it has support for Cortex-M3, which is backwards compatible, which means that the binary code that is generated for Cortex-M0 can also be run on Cortex-M3. This project defines Cortex-M3 as its platform.

In order to support a new machine type, it needs to be introduced in QEMU, by merely adding a new source file and defining the machine type in that particular source file. Then QEMU initializes the machine and calls the machine initialization function. A similar approach is followed when implementing a new peripheral or device. The difference is that the machine initializer must call the initialization functions. The nRF51 board controls its peripherals through memory–mapped I/O. These particular memory regions that peripherals use are assigned during the machine initialization phase and are created on a virtual bus called 'sysbus.' Each virtual peripheral reports its virtual I/O space size during the device initialization phase. There are also I/O read and write functions that are given as a parameter to QEMU and are encapsulated in a structure called 'MemoryRegionOps.'

After the device initialization phase finishes, the guest operating system starts executing. The virtual machine can run a simple loop without task scheduling or a simple operating system, but first, it needs to perform setup for specific peripherals to generate events. The only way to perform setup is writing configuration to the device registers that reside in memory–mapped I/O space. When the guest tries to modify or read RAM regions, this is handled by QEMU itself but memory–mapped I/O is handled by 'read' and 'write' callback functions that are passed as a pointer during initialization. nRF51 configures devices by writing to registers, and likewise, when the guest wants to fetch requested information, it reads the registers. There are few special cases where registers are not used for information retrieval. The RADIO module and ECB modules can be given as examples. Since they operate on large arrays that are bigger than 4 bytes, they make use of DMA. The nRF51 platform has a mechanism called EasyDMA, but this is on the hardware level, and it is out of scope.

## 2.5 Discussion

### 2.5.1 Development Process

QEMU has been around for more than a decade. It has hundreds of active contributors, and most of them work for prominent companies such as Red Hat, IBM, and Intel. Despite having a substantial amount of support, the project does not have documentation for developers, and it requires a great effort to understand the internals before doing any implementation work. On the other hand, the nRF51

platform has a data sheet that explains how peripherals work and interact with the CPU. This case is very typical for embedded systems as developers need to know low–level details and understand the way the system works.

A considerable number of IT companies do not focus on development documents which force new developers to learn the product by reading the source code. One advantage of QEMU is that it is possible to study how other platforms are implemented and follow the same steps in order to add support for a new platform. While it is more challenging to jump into a big project, the QEMU community makes it easier to start. There many useful sources on the web written by QEMU developers that explain certain concepts of the project.

### 2.5.2 Strengths and Weaknesses of a Virtual Environment

There are differences between physical and virtual environments; the most important ones are as follows:

- Running an ARM application on a different platform introduces some performance penalty. The same instructions cannot be executed directly on the host CPU. The host will provide driver logic for the peripherals; almost no additional performance loss is expected.
- Emulated devices tend to work perfectly. For example, a peripheral may require some time to initialize internally, and there might be some delays in specific cases when interacting with the device, the same device may require some delay when writing/reading pins. Those things are very hard to catch and simulate because the host platform will always try to deliver the best performance and therefore will try to finish execution for each operation as soon as possible.
- If there is a bug in the peripheral firmware, that will not be possible to catch with QEMU. Assume that a peripheral is using I2C communication, but there is an implementation bug in the peripheral firmware. That can lead to false assumptions when debugging and can increase the amount of time spent on troubleshooting.
- If a programmer does something that violates the procedures to follow when using the device, that may show a different behaviour on the physical hardware and in the simulation environment. For example, the nRF51 reference manual, for the RADIO device, states that illegal transitions between device states will show undefined behaviour.
- QEMU provides a remote debugging facility. It is possible to debug the guest operating system using GDB without the need for any hardware.
- Machine state can be saved and restored.
- Certain properties of the guest machine can be configured, such as RAM and the flash memory size. The nRF51 platform has several options with different RAM and flash sizes.
- External devices can be simulated by using external scripts or programs without modifying the QEMU source code.

There is always a tradeoff between using a virtual and a physical device. Device emulation can reduce testing and development time but can also hide possible

problems. Final testing should always be done on the hardware, and it should be assumed that every software and hardware design comes with bugs. For example, virtual devices can accomplish tasks according to the hardware specifications, but the actual hardware might require additional precautions in the embedded software. Such issues are usually described in an erratum for the offending hardware.

# 3

# Related Work

This chapter compares and discusses existing open–source and commercial tools targeting the IoT.

Many available implementations do not focus on IoT but can simulate basic peripherals with minimal support. One of the goals of this thesis project is providing a complete I/O support for all peripherals on the virtual nRF51 board. For example, this project makes it possible to send and receive radio or Bluetooth packets between devices, synchronize GPIO states and transfer data over UART and I²C interfaces. Since this operation is performed over network sockets, one can run external programs or scripts to interact with the virtual device. Network sockets make it possible to emulate devices on different computers with inter–device communication.

Existing closed and open–source projects and the related work are summarized in Table 3.1, and further described below.

**Table 3.1:** Related Work, categorized in its level of maturity and way of distribution (research paper, open source projects [O-S], closed source projects [C-S])

| Name | Paper | O-S | C-S |
|---|---|---|---|
| Raspberry Pi (QEMU) | | * | |
| STM32 (QEMU) | | * | |
| TriCore Simulator | | | * |
| Avrora | * | * | |
| Cooja | * | * | |
| TINA | | | * |
| ACRN | * | * | |
| OMNeT++ | * | * | |

## 3.1 QEMU

### 3.1.1 Existing Implementations

QEMU has support for more than 30 machine types on the ARM platform [13]. Currently, nRF51 is not included. In general, these virtual machines are not designed with IoT in mind. For example, Raspberry Pi can be run as a virtual machine, it has display output, it supports network devices, but there is no GPIO interaction with other virtual devices. It must be noted that there are patches available on the web for providing GPIO interaction with the outer world. There is also an unofficial

STM32 microcontroller implementation for QEMU [14], and it allows interaction through QEMU console.

### 3.1.2 Discussion

QEMU, in general, is used for virtualizing traditional desktop computers but it has powerful features for small scale platforms as well. Its Tiny Code Generator can speed up the simulation process by avoiding unnecessary steps while translating instructions for different processors. For the same reason, QEMU is a suitable tool for microprocessor and peripheral simulation but contributors have not implemented currently supported virtual machines for IoT specific purposes. For example, the nRF51 extension supports external Bluetooth, GPIO, and ADC data over a UDP interface.

## 3.2 TriCore Simulator

### 3.2.1 Details

TriCore Simulator or TSIM is a simulation tool for the TriCore platform. TriCore microcontrollers are manufactured by Infineon Technologies, and TSIM is also developed by the same company. TSIM is used for code profiling as well as unit tests and simple debugging. It usually comes as an extension or as a separate tool with some compiler suites and hardware debuggers such as Tasking TriCore Toolset [15], HighTec TriCore Tool Chain [16], iSYSTEM winIDEA [17] debugger and Lauterbach Trace32 [18] debugger. TSIM can simulate CPU cycles and load/store times with very accurate timing close to the hardware.

### 3.2.2 Discussion

TSIM is a commercial, closed source tool. It provides an API to act as a virtual debugger and an API to implement peripherals. From that perspective, it is quite similar to QEMU. The only difference is that peripheral emulation code is part of QEMU, but for TSIM, peripheral emulation is implemented as a dynamic library. TSIM only provides the functionality that is required to run a basic application on a virtual TriCore CPU. Every company designs its own hardware models; therefore, these microcontrollers can be found in many different PCB configurations with completely different peripherals. The nRF51 extension targets a single development board with a unique configuration. Another difference is that TSIM tries to emulate instruction times, whereas QEMU tries to run as fast as possible. It still may be possible to calculate instruction times, but it involves many parameters for every architecture and QEMU targets several guest architectures.

## 3.3 Avrora

### 3.3.1 Details

Avrora [19] is an open–source framework that provides a set of tools for AVR microprocessor simulation. One of the greatest features of this framework is that it can run AVR programs with cycle–accurate execution times. It has profiling tools to analyze the program behavior during simulation. It can graphically represent the program flow, and it can provide power statistics for the program being run in the simulation. Avrora allows debugging using a builtin GDB server.

### 3.3.2 Discussion

In many ways, Avrora is similar to QEMU. It provides a debugger interface as well. Besides that, it provides cycle–accurate execution timings and power statistics. These two features are missing in this project because the nRF51 extension is based on QEMU. The advantage of using QEMU is that it has been widely accepted by the community and it provides an easy interface to emulate multiple architectures as a foreign architecture on the host system. QEMU covers small and large scale devices and it focuses on the performance while Avrora focuses on cycle–accurate microcontroller emulation and the AVR platform.

## 3.4 Cooja

### 3.4.1 Details

Cooja is a network simulation tool for Contiki OS [20]. It can emulate `TI MSP430` and `Atmel AVR` microcontrollers. Cooja is not only limited to wireless networks but can also simulate full hardware for the supported architectures. Each emulated device is called a node in Cooja. It uses different techniques to simulate nodes. Any compiled binary file can be executed in Cooja if the hardware of a node is emulated. This method has the ability to run non-Contiki nodes [21], but it can only be used for the supported microcontrollers. Another method is compiling Contiki OS for the host operating system. In that way, any application can be emulated regardless of the target architecture, since the code will be running on the host. In addition to these methods mentioned, Cooja also has Java bindings. Contiki OS is compiled for the host platform as a shared library, and it is loaded through Java Native Interface. If the application logic is implemented as a Java class, Cooja can also emulate the node on the host regardless of the target architecture.

Device emulation is one of the main features, but in addition to that Cooja is also used for power profiling [22]. It makes it easier to emulate, and at the same time accurately analyze the power consumption in wireless networks.

### 3.4.2 Discussion

Cooja is a robust emulation tool. It has some advantages and disadvantages when compared to QEMU and the nRF51 extension. There is no tool to visualize the nodes and the data traveling between the emulated nodes for the nRF51 extension. Cooja provides a visual representation of the nodes in the simulation network, it can analyze and show IP packets, and it has a visual representation of the event timeline. On the other hand, QEMU specializes in different areas. For example, QEMU has support for GDB. It is possible to use all the standard features of GDB. GDB allows developers to see variables, registers, and memory contents during runtime with the help of breakpoints. GDB makes it possible to perform low-level debugging while running the native binary in a simulation. QEMU can also run the native binary without any interpretation or translation when the host and guest systems have the same architecture while still providing a full hardware emulation.

## 3.5 TINA

### 3.5.1 Details

TINA is a commercial, microcontroller circuit simulation tool [23]. It supports a wide range of microcontroller architecture types such as `PIC`, `AVR`, `XMC`, and `ARM`. TINA was initially released in 1990, since then it is under active development. There is a free version called TINA-TI which is only limited to integrated circuits produced by Texas Instruments. TINA is not only a microcontroller emulation tool. It has a PCB designer where one can draw circuits and place emulated buzzers, switches, displays, semiconductors, and other similar integrated circuit elements. It can also emulate and analyze RF networks.

### 3.5.2 Discussion

Compared to the other tools mentioned in this chapter, TINA has very advanced features. The difference between other tools and TINA is that TINA provides a generic emulation for each type of microcontroller and the user designs the integrated circuit while the other tools focus on emulating a specific type of development board or operating system and some others only provide emulation without external input/output.

## 3.6 ACRN

### 3.6.1 Details

ACRN is an open–source bare–metal embedded hypervisor for IoT development [24]. It aims to provide real–time features for IoT products. ACRN does not have support for many platforms. Currently, the only supported platform is x86. It directly runs on the host machine's hardware without needing a traditional operating system. ACRN also allows hardware sharing between the guest machines.

### 3.6.2 Discussion

This project has advanced virtualization features, and a different approach compared to the most open–source projects. One of the most significant features is that ACRN can fulfill real–time requirements for IoT applications. It can run a well–known IoT operating system, namely Zephyr OS. ACRN can prioritize and isolate safety–critical tasks and has a very small footprint. With all these features combined, ACRN seems to be a promising tool for embedded developers, but that results in a limited set of supported hardware.

## 3.7 OMNeT++

### 3.7.1 Details

OMNeT++ [25] [26] [27] is an open–source C++ library that provides network simulation features mainly. It is also possible to simulate different subsystems using additional extensions such as a file system or the lower Bluetooth communication layers. This tool has been used in many research projects, possibly more than a couple of hundred. Other than network and node simulation, it has advanced features, such as 3D visualization of the nodes using Google Earth. It can be used for testing and algorithm development. The library is distributed under the Academic Public License.

### 3.7.2 Discussion

In many aspects, OMNeT++ is quite similar to Cooja but it is not focused on running native binaries. It provides developers and researchers a rich set of functionalities. As a positive result of that users can directly start working on their design and save time. The tool has no use if the intention is to work on a specific type of hardware or run the binary files directly. While it can be used to achieve similar tasks, the methodology is completely different from this project.

# 4
# Design

## 4.1 Goal

Emulating a system requires having almost identical features in the guest system. The purpose of emulating the nRF51 board is with providing developers and students a system that can be used for learning, testing, development, and debugging.

This project aims to provide an ability to run the binaries compiled for the nRF51 platform under QEMU. One of the main focuses is to be able to run a network of virtual devices so that they can communicate through pins, radio, UART, and similar I/O interfaces.

## 4.2 Approach

QEMU and this project are implemented in the C language. There are no additional dependencies and libraries. The only exception is that guest-to-guest communication is provided over UDP communication, and it requires an external application to route packets between guests. Currently, the server application is provided and is written in Python.

The physical hardware itself provides a typical configuration of peripherals that can be found in an embedded system. In addition to that, it has a radio module with Bluetooth capabilities. The platform focuses on ultra-low power wireless solutions.

The nRF51 platform has a concept of tasks and events. A task can be considered as an action that is being performed by a peripheral. For example, ECB (Electronic Codebook Module) is used for AES encryption. The CPU module can trigger the encryption task by writing to the device registers. The CPU performs no action in that case except the encryption request. An event, in general, indicates that an operation is finished. The event flag can be accessed through the device registers to see if it is set. If the event flag is set, it means that the event has occurred, if not, the companion task has not been triggered, or it is still in progress. Each peripheral has registers to enable or disable interrupt requests. If the interrupt flag is set, a particular event will be generated for specific events. Currently, each device has a single interrupt allocated. In other words, multiple events map to a single interrupt. The CPU must check for each flag in the corresponding register to see which events have occurred after an interrupt. It is possible to disable or enable interrupts for specific events, but that will not avoid the event flag from being set in the device registers. Therefore, the CPU can check the event by polling without having the interrupt overhead. This method might be useful in some instances, for example, if

too many interrupts are generated for a specific event.

Each interrupt is allocated during the device initialization phase in QEMU, and each device has its own I/O space for registers. Each register read and write request is routed to given callback functions by QEMU. In that case, read callback must return a value to pass to the guest CPU and write callback must save it or take action depending on the type of register.

One of the critical things that was considered during the development of the virtual machine was that every module should be duplicable. This is to say, no module should depend on constant factors, and no module should have common dependencies with others. This goal has been accomplished for most peripherals, but there are exceptions such as GPIO and GPIOTE module. These two modules depend on each other as if they operate like a single device on the hardware. Other peripherals such as real–time counter, random number generator, UART module can have multiple instances. That provides a possibility to support future hardware configurations. The nRF52 platform can also be supported with small changes, but this project focuses on the nRF51 platform.

## 4.3 Design Details

This section describes the design and evaluation steps for each peripheral. The evaluation for each peripheral was performed using the provided SDK to avoid any ambiguity, and the resulting behavior is compared against a physical device.

### 4.3.1 General Purpose Input/Output (GPIO)

The GPIO module has simple configuration options. Its direction (in or out) can be set through registers. The output value is set in the same way, and the input value can be read through a register. GPIO is very easy to use for simple tasks, but its control can be taken over by another peripheral. There are some hardware–specific configuration options. For that reason, some features are not emulated. For example, it is not possible to configure pull up and pull down for pins. GPIO module does not have interrupts, but the GPIOTE module provides this functionality.

It is designed to be able to communicate with other guest machines. This mechanism is provided over UDP communication. Each state change is reported to the router script using a predefined packet structure. The router modifies the pin number in the packet and sends it to another guest. Router script has a configuration file that defines the GPIO connection between guests using a simple syntax. Configuration parameters require device identifiers for two machines and a pin number for each device, providing a virtual connection between guests. A state change is routed to the target guest, and it reads the contents of the package, extracts GPIO pin number and state then sets corresponding registers.

### 4.3.2 GPIO Tasks and Events (GPIOTE)

GPIOTE stands for GPIO tasks and events. As the name suggests, this module is primarily used for interrupts. When GPIOTE is configured to use a GPIO pin,

it takes ownership of that particular pin, and the same pin cannot be used by the GPIO module anymore. It has been designed in the same way in the project. There is an owner parameter for each pin in the GPIO module. Write requests that come from the GPIO module are ignored, and the input state that was received from another guest is routed to the GPIOTE module.

The GPIOTE module is able to generate interrupt requests. If a GPIO state change packet is received, this directly generates interrupt and event in the guest machine. It differs from the GPIO module when writing the output value. CPU must trigger the 'out' task in the GPIOTE module in order to set pin output value. It is also possible to connect some peripherals with the GPIOTE module; for example, RTC can be connected with GPIOTE to toggle LED output without involving the CPU.

### 4.3.3 Universal Asynchronous RX/TX (UART)

The UART module is designed to use UNIX sockets as the communication method. UNIX sockets can be changed to use the router script for simplicity. Currently all UART I/O is routed to a UNIX socket. QEMU creates a socket file at '/tmp/nrf51_DEVID.sock' where '`DEVID`' is replaced with a hexadecimal device identifier.

The nRF51 SDK provides built-in printf and scanf functions that use the UART interface. It has not been individually tested, but UART has been used as the machine console in the provided test programs. So it makes it possible to see the output from the virtual machine. In order to do that, the following command can be used: '`socat - UNIX:/tmp/nrf51_DEVID.sock`'. The same program, socat, can also be used to connect two guest UART interfaces by giving two UNIX socket parameters and replacing '-' which stands for stdin/stdout.

UART is not a high-speed communication interface. In order to simulate low–speed communication rates, the current design has a periodic timer that handles byte transfer for this module. This method introduces a delay between each transmitted byte.

### 4.3.4 AES Electronic Codebook Mode (ECB)

The nRF51 platform also supports AES encryption, and the ECB module provides encryption in electronic codebook mode. This module has a simple mechanism. The CPU writes a pointer in one of the device registers (ECBDATAPTR) and triggers the STARTECB task. QEMU then performs the encryption and generates an interrupt. It must be noted that on the nRF51 platform, this module only supports encryption and does not support decryption.

There is also a task called STOPECB that is used to terminate an ongoing encryption process. This is not reasonable to simulate in QEMU because of its nature. When the STARTECB task is triggered, the host immediately performs the encryption. This operation takes place in I/O write callback function, and it is quite fast to encrypt just a single AES block. From the guest CPU perspective, this operation is completed even before the next instruction is executed. So there is no chance to

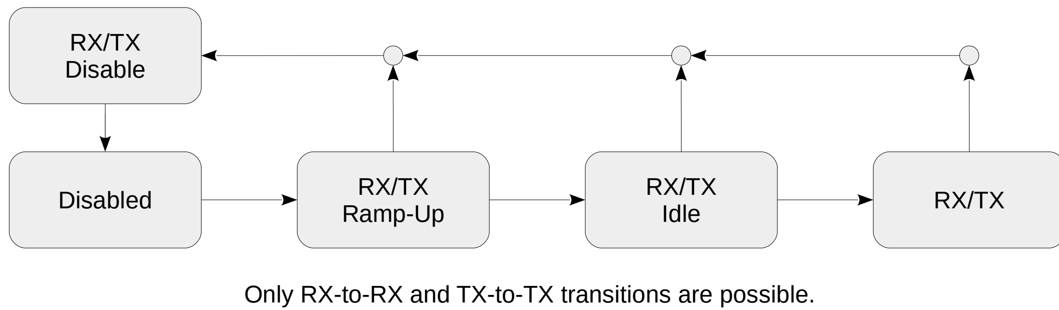



**Figure 4.1:** RADIO States. Inspired by the state diagram presented in the nRF51 Manual [11], p. 85.

cancel the operation.

The evaluation has been performed using AES test vectors, and these vectors are also embedded in the test program. The guest software compares the encrypted output with the hardcoded ciphertext to verify that the AES module works correctly.

The ECB module uses EasyDMA. In QEMU, this is performed by reading the plaintext from the guest memory by using 'address_space_read' API, and the ciphertext is written into the target memory region by using 'address_space_write' API. Plaintext, ciphertext, and key are contained in a single array, and ECBDATAPTR must point to the first byte of this array.

### 4.3.5 2.4 GHz Radio (RADIO)

RADIO module is the most complex peripheral on this platform. It has different operating modes. It can also interact with other peripherals. For example, it can use AES CCM mode encryption with the CCM module.

This module is capable of transmitting a maximum of 254 bytes in a single packet at three different speeds: 250 Kbit, 1Mbit, and 2Mbit. The reference manual also mentions technical details about the on-air packet layout and how they are transferred, but this is out of scope. The data is transferred over UDP and delivered to all guest machines that have the matching mode configured in their device registers.

Currently, the UDP protocol can only do filtering based on the radio mode. Address filtering is not defined in the protocol. CRC calculation always indicates success. It is not required at this stage, but it might be required for specific reasons other than only providing information.

RADIO has device states in which CPU can start transmission and other states in which CPU can use this peripheral to receive data. Figure 4.1 shows the valid RADIO state transitions. nRF51 reference manual [11] also presents the tasks and events between state transitions under the RADIO section.

The RADIO module in QEMU is designed to run a state machine according to the device state diagram. The state machine is run by a timer when any device task is active. This module also uses EasyDMA. Packet pointer is given to the PACKETPTR register by the guest CPU. Data is read from this location when transmitting, and new data is written to the pointed location.

### 4.3.6 Analog to Digital Converter (ADC)

The ADC module is a very simple analog to digital converter peripheral. Currently, it has been designed to add a small randomized noise on a hardcoded number, and it has not been connected to the other guests. Some configuration values can be used for voltage prescaling and pin selection. These are currently ignored. The reasons are explained in the communication section.

The test program uses SDK to read the analog input, and the value is printed on the UART console.

### 4.3.7 Clock Management (CLOCK)

The CLOCK module has been designed to provide register access. It only allows reading from and writing to the device registers. Also, it can generate an interrupt on a calibration request. This module is used for power management, GPIO input sensing speed, and similar purposes. It is mainly related to the hardware and has not much use for emulation.

### 4.3.8 Random Number Generator (RNG)

The random number generator module uses internal noise to generate random numbers. Therefore it provides true non-deterministic output. In QEMU, this is not designed in that way as it is not going to be used for the real environment. The algorithm used for the random generator is xorshift. The initial seed value is a prime number that is XORed with some bytes that are read from '/dev/urandom'. This method has been used to avoid outputting the same number in QEMU when the machine is started for the first time.

The test program requests an array of random numbers using SDK API and prints some numbers on the UART console that were provided by the RNG module.

On real hardware, the time required to generate a single random number is non-deterministic as well. In QEMU, this behaviour is different. Random numbers are generated immediately.

### 4.3.9 Real Time Counter (RTC)

The nRF51 platform provides three individual real–time counter instances. As mentioned earlier, most devices are designed in a way that they can be instantiated multiple times. The current implementation creates these virtual devices with different memory I/O regions according to base addresses given in the reference manual. I/O read and write callbacks are the same, and the implementation is unique.

RTC provides compare and tick events. Compare event is generated when the real–time counter value matches one of the values that are provided in 4 different compare registers.

Some test programs use the RTC module for delay functionality. This is achieved by using an interrupt handler in the test program that counts the ticks and provides a global system time, and when the delay function is called, it blocks the execution until the limit is reached. Since the interrupt handler is not affected by that, it still

keeps counting. There is also a second RTC module used in the test program with a configured prescaler value to have a lower frequency of interrupts. That also toggles one of the pins which can be seen in the router script. Neither of these were affected by the delay function blocking the main loop of the program.

## 4.4 Guest-to-Guest Communication

The communication with other guests is provided using a global UDP connection in QEMU. It is possible to use TCP. For now, UDP has been used for its simplicity and connectionless approach. UDP has some well-known disadvantages. If one side sends too many packets, some of them will be dropped. Since guest communication is on the same host machine, this does not cause any issues, but if two remote guests try to communicate, some data can be lost easily. In order to avoid any side effects, delay function has been called between some actions in the test program, specifically for GPIO operations.

QEMU listens on a UDP port when it starts and sends the device identifier to the router script then it waits for incoming packets so that the script is aware of QEMU as a client. After a packet is received, it checks the target peripheral state to see whether the packet is acceptable. For example, when a RADIO packet arrives, QEMU checks if the RADIO is in the RX state. If not, the packet will be dropped. After validating that the RADIO is in the RX state, the packet can be accepted. The packet pointer that was given in the RADIO configuration register is already saved before switching to the RX state. QEMU then writes the payload content into the memory location specified in the packet pointer, sets the nRF51 events, and finally generates an interrupt.

Currently, only RADIO and GPIO modules use this method for communication. Each GPIO pin requires a definition in the configuration file for a virtual connection between guests. The RADIO module does not require any configuration. One packet that was sent from a guest will be broadcast to all other guests like it was coming from the air. However, it is QEMU's job to drop it if the RADIO configuration does not match the received packet. If the configuration allows, then the guest can get all the packets that were broadcast.

UDP communication has a simple protocol. A typical UDP packet is presented in Figure 4.2.

Figure 4.3 shows the 'Send ID' packet with type 0x0. This message type is used to send a device identifier to the router script. So the router script is aware which device resides on which address. For now, it is only useful for GPIO, but that will be used for other peripherals as well. RADIO packets do not need to match any particular device for the reasons mentioned earlier.

Figure 4.4 shows the RADIO packet with type 0x1, and Figure 4.5 shows GPIO packet with type 0x2.

The router script reads the pin number and finds the corresponding pin number on the target machine. Then it replaces the source pin number with the target pin number and sends it to the target guest. When the packet is received on the target machine, the corresponding pin state is changed.

The UART implementation uses a different communication method. All data

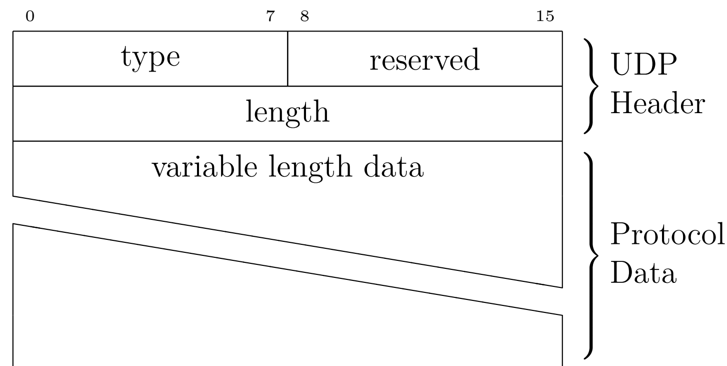



**Figure 4.2: Generic UDP packet.** Where type specifies the type of content that comes after the header and length is the total length of the content excluding the UDP header.

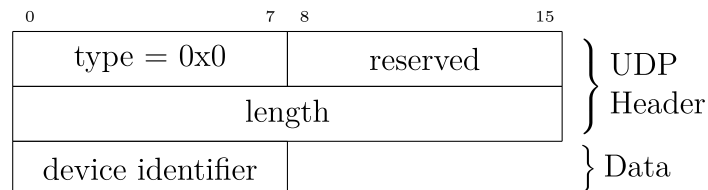



**Figure 4.3: Send Device Identifier packet.** This message type is used to identify each QEMU instance.

transferred is performed over UNIX sockets. This method was chosen because the UART interface is usually used as the system console. Therefore, it was not included in the UDP protocol. There are applications that can interact with UNIX sockets pretty easily. For example `socat` is a well–known software that can use a UNIX socket for user input, and display the incoming data.

### 4.4.1 Router Script Configuration

The router script has a simple configuration format to create pin pairs between the guest machines. The format is presented in Listing 4.1 where guest_id is the hexadecimal device identifier and pin is the pin number on that machine. Each pin pair must be defined in the configuration file (nrf51.cfg) in a single line.

**Listing 4.1:** Configuration File Entry Example

```
(guest_id1, pin1) (guest_id2, pin1)
```

## 4.5 Discussion

UDP was chosen for its simplicity. There is no need to keep track of all TCP streams, and UDP allows sending packets without having a connection which means that the

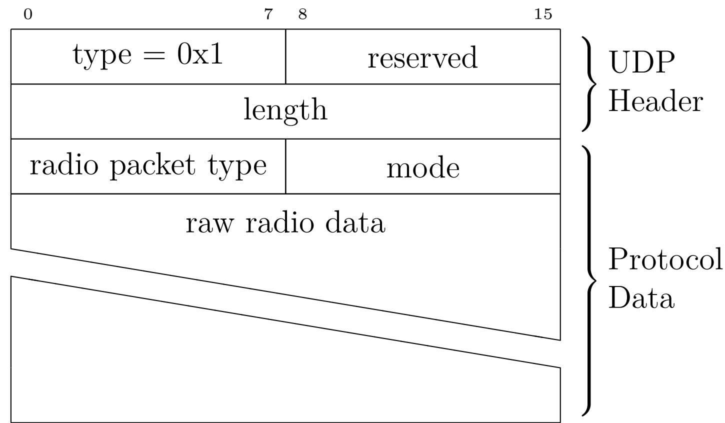



**Figure 4.4: RADIO packet with type 0x1.** Where type specifies the radio packet type. This was previously used for a different purpose, and for now, it is only defined as a data type with a value of 0x1. The mode field is the sender's Bluetooth mode. This value is sent as it was read from the mode configuration register.

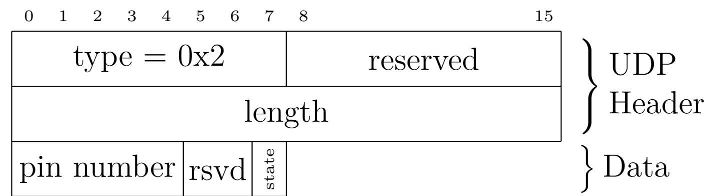



**Figure 4.5: GPIO packet with type 0x2.** Where the state field is the pin state indicating whether it is LOW or HIGH. The pin field is the pin number between 0 and 31. Reserved field (bits 5,6) is always zero.

router script does not have to check connection state and it does not have to wait for a connection to be established.

There are some drawbacks in the current design. It relies on UDP communication, and some packets might be dropped during communication. It is unlikely to miss packets when running all the devices on a local machine but chances are higher if the local machine is overloaded or the virtual devices are on different computers.

Another disadvantage is that timing will not be so accurate in the case of GPIO communication. A physical device will be able to react in a much shorter time on a GPIO event. In general, GPIO is not used for high–frequency output. Instead, clock sources or timers are used for this kind of operations, but it must be noted that the nRF51 platform allows GPIO ports to be indirectly connected to the TIMER module. A pin can be configured as GPIO, but the TIMER module can toggle its output without involving the CPU. There is a mechanism that allows one peripheral to write into another peripheral device register. In that case, it is not guaranteed that the UDP packet will reach its destination at the same interval.

In case of a missing UDP packet, GPIO state will be desynchronized. That might seem like a defect in the program running on the virtual device. On the next UDP

packet, GPIO state will be in sync again.

# 5 Implementation

## 5.1 Overview

This chapter explains how the implementation was done in QEMU. Also, it describes the communication methods used between the emulated devices.

It is a generic technique to use device registers to control peripherals and transfer data through direct memory access. The nRF51 platform contains a CPU and some integrated peripherals. They interact with each other in the same conventional way. QEMU allows developers to emulate these actions by providing the necessary function callbacks.

The memory on the nRF51 platform is divided into few areas. They can be named as 'Code', 'SRAM' and the 'Peripheral' area. The code section is directly loaded from a binary file that is produced by the compilers. It contains the program contents that will be executed during device operation. Also, any data that is saved into non-volatile memory is saved in the same file because that binary file is the exact mirror of the static data stored in the code area. SRAM is the program memory being used during operation which is only a byte array in QEMU. The peripheral area is treated in a special manner and it is the most interesting part of this project. It is safe to assume that the peripheral area is some sort of a message passing interface. For that reason, each 'read' and 'write' operation must be processed by the emulator. These input and output operations are mapped into the device memory.

The code area starts at `0x00000000`, SRAM starts at `0x20000000` and the peripheral area starts at `0x40000000` as seen in Figure 5.1. It does not necessarily mean that every R/W access to the peripheral area is valid. Attempting to access an invalid register will generate an exception on the hardware. QEMU allows developers to register R/W callbacks for each peripheral and specify the valid address range. If the device is not implemented, it will automatically generate an exception on the virtual CPU as it was invalid. The read and write callbacks receive the necessary parameters, such as the register offset and the write value. The value returned by the read function is immediately available in the virtual CPU. The value written into the register directly comes from the virtual CPU.

Depending on the peripheral, a write operation might start a job or change the active configuration. For example, interrupts are disabled and enabled through a set of registers for each peripheral. The value read from the interrupt registers gives us information about the interrupt configuration.

In addition to that, the guest system may start and abort jobs through write–only device registers. This is a way of sending a signal to the relevant peripheral.

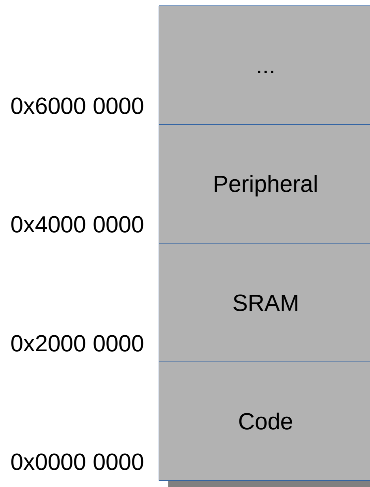



**Figure 5.1:** nRF51 Code, SRAM and Peripheral Area

## 5.2 QEMU Peripheral Interface

In QEMU, each peripheral device is implemented in the same way for this project. On the nRF51 platform, peripherals are controlled through device registers, and they are accessed through memory–mapped I/O. `RNG` module can be given as a simple example. Its address space starts at `0x4000D000`, and there are nine documented device registers. The `START` register resides at offset `0x0`, and it triggers the random number generation operation when the user code modifies it. Then the CPU continuously reads the `VALRDY` register at memory address `0x4000D100` when using poll mode. If the register indicates that the value is ready, CPU can fetch a newly generated number from the `VALUE` register at memory address `0x4000D508`. A simple flowchart describing the relation between the program and QEMU is presented in Figure 5.2.

QEMU requires the following function callbacks in the source code for all peripheral devices:

- `read`: Called for each memory mapped I/O read operation.
- `write`: Called for each memory mapped I/O write operation.
- `instance_init`: Called once for each device instance initialization.
- `class_init`: Called only once when QEMU has started.

These callback functions are the only way to interact with the virtual nRF51 device. Whenever a guest requests to read or write a register, the corresponding callback is fired by QEMU, and after that, the implemented code is responsible for keeping and modifying the machine state. The '`read`' and '`write`' callbacks receive relevant parameters such as device context, register offset, and the value to write. The '`read`' function returns the requested value. Some device registers are read–only, some are partially read–only, and some are write–only. The device implementation handles all the logic according to the nRF51 datasheet. Actual register values are kept in a separate structure, and this structure is part of the device context, named

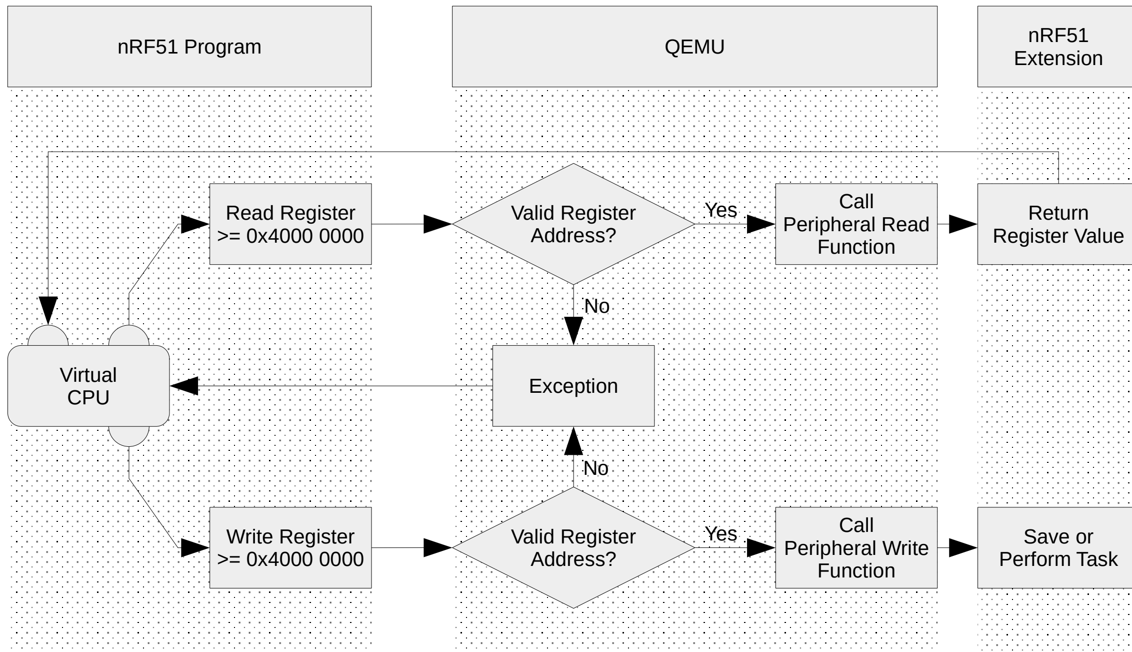



**Figure 5.2:** R/W Callbacks in QEMU

`REG`. This allows the same callbacks to be used for every instance of the same device class.

In addition to these callbacks, there is a global UDP socket that receives GPIO states and RADIO packet data from a router script that serves as a gateway between virtual devices. Based on the received data, the machine state can be modified by changing peripheral device registers. When the pin input is changed through the UDP interface, QEMU will generate the corresponding interrupt if interrupts are already enabled by the guest. If interrupts are not enabled, the UDP handler function will only change the device state by updating relevant device registers in the `REG` structure. The guest will only be able to determine any GPIO state change by reading the device registers. In that case, the '`read`' callback will return the requested value from the `REG` structure. Likewise, any change to the GPIO state generates an outgoing UDP packet to the router script.

The RADIO module uses DMA to avoid CPU interaction for relatively large data transfers. Upon the reception of a RADIO packet, an interrupt might be generated based on the peripheral configuration, but the received data is not kept in the device registers. Instead, it is directly written to the configured memory space. UDP interface extracts the raw packet data, puts it in the configured memory location, and updates the device registers. The nRF51 extension generates an outgoing UDP packet as well when the guest triggers a transmission operation.

This implementation is publicly available on GitHub [28].

## 5.3 Discussion

QEMU has all the required facilities to emulate the nRF51 platform. However, in some instances, there are behavioral differences between a physical device and an

emulated device. The ECB module performs AES encryption using a cryptographic accelerator. On the physical system, it is possible to interrupt this operation because it takes multiple CPU cycles to finalize encryption. In this implementation, as soon as the `write` callback is fired with relevant parameters, encryption is performed in the same call. Therefore, QEMU does not get the chance to return to the event loop and execute the next guest CPU cycle. From the guest's perspective, this operation takes only a single CPU cycle in a virtual environment, so there is no possibility to interrupt ongoing ECB encryption. This has been chosen to be implemented in that way as AES encryption is performed on the host CPU. Modern desktop platforms have a particular AES instruction set and assuming that this implementation will mostly be used on desktop computers, it is the preferred way to block the `read` callback for encryption operation rather than immediately returning to the event loop in this project.

As already mentioned, UDP communication is not the best method for state synchronization. It can be changed to TCP with little more effort but requires a more advanced router script or application. There are other reliable techniques, as well. For example, the OpenMPI project [29] provides a message passing interface that can be utilized over the network or by shared memory.

There has been a great challenge when implementing TIMER and RTC peripherals. On the hardware, an interrupt will immediately stop the program flow and be handled by the corresponding service routine. In QEMU, it works in the same way but with slight differences. The translation blocks that are generated by TCG may include multiple instructions for better performance. Therefore preemption is not possible between some cycles. In addition to that, any interrupt generated by the TIMER or the RTC module will not be as accurate as the hardware. The periodic events occur at the correct time interval but when the CPU handles the interrupt, the deviation in the TIMER or RTC counter, depending on the resolution, might be zero or less than few ticks. This causes problems within the programs that require high accuracy. For example, some programs check the RTC counter before and after setting a future RTC event to avoid multitasking problems. Any interruption in the middle of the setup process can delay program execution but the RTC will be running, and there is a possibility of setting the event for the time in the past. For the reasons mentioned above, some deviation will be seen. In addition to that, some overhead in the emulator itself will introduce more differences in the timing functions. Since this problem does not always occur, it was not easy to understand the root cause. It has been one of the greatest challenges in this project.

# 6
# Evaluation

This chapter presents the evaluation results and explains why and how the experiments were performed.

## 6.1 Evaluation Method

In order to understand if the implementation is sufficient enough to replace hardware for testing and development, it needs to be evaluated under two different categories, namely functionality, and performance. Functional tests are performed to observe the internal and external behavior of the virtual system and compare them with the hardware. Performance tests consist of running different types of programs.

### 6.1.1 Metrics

The implementation is evaluated for performance and functionality. Performance is measured in terms of resource consumption, such as CPU and memory usage. In addition to that, executions are measured using time units both on the hardware and in the virtual system. It is very likely to get different results on different host systems. In the virtual system, executions are measured using the host system timer. QEMU provides a simple API to get the host system time which is not affected by the emulation. On the hardware, the executions are measured using the TIMER peripheral which runs independently from the CPU.

Functionality is not something that can be measured using only the numbers. The best way to make a judgement on the functionality and yield a set of results with high confidence is to perform experiments using different operating systems, libraries and a variety of programs with different purposes. The judgement is made based on the output from the physical hardware. Emulated device output should always match the physical device. There is an exception to this rule. For example, timing based behavior can differ and it is accepted.

Most of the time, hosts will have different hardware configuration, e.g., CPU and the chipset, because of the availability of a wide range of choices on the desktop and headless environments. The host system used for emulation to perform experiments in this project runs Intel® Core™ i5-5350U CPU at 1.80GHz. It can be considered as an average CPU for personal computers. Other components are not specified in this report, such as the RAM size or the chipset model.

The implementation itself does not allocate any dynamic memory; therefore, it is not possible to perform any experiment that can use a higher or lower amount of

memory. A different chipset or CPU will affect the performance results. In most cases, results will differ. Some systems may have faster or slower bus transfer rates between RAM and the CPU. The CPU cache is also a significant factor.

The experiments performed in this report are enough to make a judgement if the implementation is sufficient enough to use it as an emulation tool on an average host system because the implementation does not have any other purpose than emulating nRF51 in QEMU for application development and testing.

The compiler used in these experiments is `GCC version 7.2.1 20170904` from the GNU ARM toolchain. It is likely to get slightly different results from different compiler versions and toolchains in terms of code size and performance.

In a virtual environment, there is no requirement for a special testbed setup. QEMU only requires the kernel file which is the compiler output for the relevant program. The test programs are run using the nRF51-DK development board on the physical hardware. There is no requirement for the physical environment either other than connecting the board to the computer using a USB cable.

### 6.1.2 Performance Tests

The performance tests are required to compare the CPU execution time and the AES encryption time. The instructions executed are not native; they are emulated through TCG[1]; therefore, an evaluation is required to see how fast the host can run the guest applications. The expected behavior for the host is to run applications a lot faster than the physical device. There are mainly two reasons, TCG and the nature of microcontrollers.

TCG is a way of running non-native instructions. It is very similar to a compiler as the name suggests. In simple words, TCG compiles a set of guest instructions into host instructions, and it performs optimization during compilation. In QEMU, guest functions are translated into QEMU Translated Block (TB) and branches are translated into basic blocks.

Low power microcontrollers do not have much processing capacity when compared to desktop processors. They are designed to accomplish specific tasks in a limited environment.

These reasons mentioned above suggest that QEMU is likely to give good results for performance tests. It is worth mentioning; this does not mean that the host system will consume the same amount of power while running the emulation. It is evident that the average host system will consume more energy. For this project, energy consumption is not a matter of interest.

As already mentioned, measurements are done using the TIMER peripheral on the hardware and the host system timer in QEMU.

### 6.1.3 Functional Tests

The functional tests are executed to see the behavior of the emulated system. Tests are focused on the peripherals rather than the CPU. The reason is that the implementation only includes peripherals for the nRF51 development board. QEMU has

[1]Tiny Code Generator

had support for the Cortex-M3 core for a while, and it has been tested and used by many. Specific modules are tested in order to see if they differ from the physical device.

The most important thing with functional tests is to see that one can use it as a virtual replacement for the hardware. It is very crucial to have a matching behavior for the peripherals; it is the only way to run nRF51 applications without any modifications.

### 6.1.4 Hardware Setup

The tests do not require a specific hardware setup. The only required components are the nRF51 development board and a serial or USB cable. The UART interface is used to get some human–readable information from the hardware regarding the tests. The development board model used for experiments is `nRF51 pca10028`. It has USB to TTL converter onboard. It is possible to get UART output through USB with the help of the converter. The same output that QEMU gives us is also received from the physical device through the USB cable without a serial interface. Usually, GPIO tests for similar hardware would require jumper wires. In this project, some functional tests are performed using onboard LEDs which are attached to the GPIO outputs. Some performance tests are executed using an Arduino Uno development board which was attached to nRF51 through GPIO pins.

### 6.1.5 Experiments

Experiments are performed both on the physical hardware and in QEMU. It is crucial to run the same resulting binary file on both platforms after compiling the test code. This is required to compare the behavior and performance. Each experiment has its own test application that is appropriately designed for the task, and some of them are run under an operating system called Zephyr [30].

Zephyr is an open–source operating system for resource–limited environments. It can run on multiple platforms. While it is designed for microcontrollers, Zephyr also supports compute-intensive architectures such as x86 which is usually classified as a desktop platform. It features a small footprint kernel and comes with various protocol stacks.

For this project, Zephyr OS is one of the most suitable tools for evaluation. It has support for almost all the peripherals on the nRF51 platform.

The importance of running certain tests under an operating system is that it can show the correctness of the implementation using multiple peripherals simultaneously, because a typical operating system uses interrupts, system timers, and other various peripherals after startup in order to provide some services for the user code such as task scheduling, driver management, and other kernel–level jobs.

Zephyr makes it easier to develop an application and run it on multiple platforms. For example, an application targeting the Arduino board may very well run on the nRF51 platform with minimal modifications, and most of the time, changing the target during the configuration phase is enough.

Zephyr OS comes with a Bluetooth stack implementation that supports many

Bluetooth services and device types. It allows us to create a program that acts as a heart rate sensor with less than 100 lines of code.

The eleven experiments are summarized in Table 6.1, and further described below.

**Table 6.1:** List of Experiments Performed

| Name | Test | Type | Based On | Comment |
|---|---|---|---|---|
| Exp1 | GPIO and RTC | Func | nRF51 SDK | Change GPIO with RTC |
| Exp2 | UART | Func | nRF51 SDK | Echo program for UART |
| Exp3 | AES | Perf | nRF51 SDK | Encrypt a block in loop |
| Exp4 | Raw Execution | Perf | nRF51 SDK | UDP protocol impact |
| Exp5 | Simple GPIO | Func | mbed Library | mbed Library demo |
| Exp6 | Blinky Led | Func | Zephyr Project | GPIO test in Zephyr OS |
| Exp7 | CPP Sync | Func | Zephyr Project | Multitasking test |
| Exp8 | Entropy Test | Perf | Zephyr Project | RNG test in Zephyr OS |
| Exp9 | Fibonacci | Perf | Zephyr Project | Benchmark and analysis |
| Exp10 | RADIO Test | Func | nRF51 SDK | Test RADIO using SDK |
| Exp11 | Bluetooth Test | Func | Zephyr Project | Test RADIO in Zephyr OS |

**Exp1:** The GPIO is the most used unit on a typical embedded system. It is easy to observe the behavior visually on the hardware using the nRF51 onboard LEDs. The first functional experiment is to see whether the virtual system behaves as expected when using RTC and GPIO together. This test manipulates the pin output regularly with the help of the RTC module. On the physical hardware, it turns LEDs on and off. In QEMU, UDP packets are sent to the router script which listens on a specific port, and if there are any available clients, it sends incoming packets to the other QEMU instances. In this experiment, the router script prints out the information for the relevant GPIO pin state when a new UDP packet is received. It should show a matching pin state information for each GPIO pin. For example, if LED 1 is connected to pin 21 on the physical device and LED 1 is turned on at `t` where `t` is a relative point of time, the script output must show that GPIO pin 21 at `t` is HIGH. This experiment is performed in order to understand if two different units can operate without affecting each other. At the same time, this experiment tests the functionality of GPIO and RTC peripherals.

**Exp2:** The next experiment in the list tests whether the UART peripheral works. This experiment might be considered as one of the most straightforward experiments, but still, it is an effective and easy way to see how the system interacts with the user through the UART interface. The implementation provides communication through UNIX domain sockets. On startup, the implementation creates a UNIX socket at '`/tmp/nrf51_x.sock`' where `x` is the virtual device instance ID encoded in hexadecimal numbers. Then, this socket is used for outputting and inputting data to and from the user. The data itself can be text or binary. For this experiment, the test application reads any type of data from the UART and prints out the data in hexadecimal format. The application works as an echo server with a slight difference that it echoes encoded data. Since the test application outputs data in hexadecimal format, it can accept binary input then we can verify what the system receives. For

example, one can see line breaks in the output as well. This method provides a way to run the test without any particular input tool.

So far, only functional tests have been discussed, but the performance is another point that needs to be experimented with.

**Exp3:** The nRF51 platform has an AES accelerator that performs the encryption on a separate unit rather than using the CPU. The implementation acts as a crypto accelerator and uses the host CPU for encryption. The experiment involves encrypting a block of data many times. Since encryption is a fast operation, it is easier to measure the throughput for the AES unit rather than comparing single block encryption time. It will be accomplished by counting the number of rounds for a specific amount of time.

**Exp4:** The next performance experiment is the raw execution test. Under this experiment, only the CPU execution time is measured with little interaction to peripherals. The purpose of this test is comparing the processing power for both the physical and the virtual system. The test involves running a loop that is filled with assembly instruction `NOP`. For example, running the loop for 20 million rounds would give us a rough idea for raw performance. There are so many factors that affect the execution time. Instead of running an empty loop, it might be replaced with a function that has so many branches or no branches at all. Depending on the optimizations done by the TCG, we could get different results for different types of programs. We can make a judgement if QEMU can perform worse or better by performing this experiment. It is a known fact that QEMU gives exceptional results for microcontrollers, and this is explained with numbers in the following sections. The implementation provides a way of communicating through the UDP connection with other virtual devices. Therefore it is expected to have a delay on the UDP channel. GPIO channel can be accepted to have no delay if we exclude the speed-of-light delay on the wire. The reasons mentioned above require us to perform an experiment that triggers UDP communication. It must be noted that the GPIO peripheral itself can have a tiny amount of delay that can only be measured with an oscilloscope. The purpose of this test is to measure the impacts of UDP communication on the execution time. The test involves changing GPIO state for a specific pin very rapidly and it is being measured by an external development board, namely Arduino Uno.

**Exp5:** Demonstrates the use of the mbed Library in a virtual machine by manipulating the GPIO states. The mbed Library supports other peripherals as well but they are not tested in this project.

Experiments from **Exp6** to **Exp8** are performed using Zephyr OS to observe the behavior of a generic real–time OS. These tests involve using GPIO and RNG peripherals and the multitasking feature from Zephyr OS.

**Exp9:** This experiment is performed for extensive performance analysis in QEMU by describing the program assembly output.

**Exp10:** The nRF51 SDK provides a basic driver for radio communication and this experiment uses a simple approach to transfer small data.

**Exp11:** This experiment uses the Bluetooth stack from Zephyr OS. It is the most advanced experiment in this report because it depends on many peripherals such as TIMER, RTC, PPI and RADIO. The main purpose of the experiment is to test the

RADIO peripheral using the Bluetooth protocol between two virtual devices.

For this thesis project, Zephyr is mostly used for functional verification. All the experiments are performed under `Zephyr v1.14.0` using `nrf51_pca10028` as the build target.

## 6.2 Results

### 6.2.1 Exp1: GPIO and RTC Test with LEDs

This experiment uses RTC interrupts to count the number of ticks that are triggered by the RTC module. Then the number of ticks is used as the global time during the execution. For this experiment, the only requirement is to provide some delay mechanism. During the startup phase, the system configures the RTC module, and the LEDs then enters an infinite loop which toggles $LED_1$ on the hardware. The LEDs are numbered from 1 to 4 on the physical board, but in the source code, they start from $LED_0$. See Listing A.2 for the application contents. The RTC is configured with the prescaler value of 327, which approximately corresponds to 100 Hz. In this experiment, the toggle delay is 100 ticks. Since the RTC works at 100 Hz, the toggle delay corresponds to 1 second. This experiment is run both on the physical hardware and in QEMU. The LED states for the hardware can be seen in Table 6.2 after the application has started. $LED_2$ stays ON during the lifetime of the application and $LED_1$ toggles every second.

**Table 6.2:** LED states on the hardware for $t_{sec} = 0, 1, \dots 5$.

| $t_{sec}$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $LED_1$ | ON | OFF | ON | OFF | ON | OFF |
| $LED_2$ | ON | ON | ON | ON | ON | ON |

QEMU gives us the same results with more details, but it does not specify any LED number; instead, we can only observe the pin numbers and their states because QEMU sends out UDP packets after a state change. Listing 6.1 shows the output from the UDP router script, and Table 6.3 shows the state for each pin according to Listing 6.1.

**Listing 6.1:** Script Output

```
(0xface) pin: 21 state: 1 /* t = 0, startup phase */
(0xface) pin: 22 state: 0 /* t = 0, startup phase */
(0xface) pin: 21 state: 0 /* t = 0, startup phase */
(0xface) pin: 21 state: 1 /* t = 1 */
(0xface) pin: 21 state: 0 /* t = 2 */
(0xface) pin: 21 state: 1 /* t = 3 */
(0xface) pin: 21 state: 0 /* t = 4 */
(0xface) pin: 21 state: 1 /* t = 5 */
```

**Table 6.3:** Pin states in QEMU for $t_{sec} = 0, 1, \ldots 5$.

| $t_{sec}$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $Pin_{21}$ | LOW | HIGH | LOW | HIGH | LOW | HIGH |
| $Pin_{22}$ | LOW | LOW | LOW | LOW | LOW | LOW |

On the development board used in this experiment, $pin_{21}$ is attached to $LED_1$ and $pin_{22}$ is attached to $LED_2$. The results from this experiment show us that we observe the same behavior in QEMU as the physical hardware.

### 6.2.2 Exp2: UART Test

In this experiment, the nRF51 board receives a simple string that can be typed on the serial console; then it prints out the hexadecimal representation of the input data. In that way, we verify that the UART interface works and it can both receive and send any data. See Listing A.3 for the example code used to verify the functionality of the serial console. We expect to see the same results from the hardware and the QEMU instance. For this experiment, the UNIX tool `socat` is used to interact with the virtual console that is connected to a UNIX socket. QEMU creates a UNIX socket when the UART interface is initialized from the application; then the same socket is used for data exchange. Listing 6.2 shows the command that was used to start `socat` and its output. The input string is '`nrf51 test input`' which is 16 bytes in length, and its hexadecimal representation is '`6e 72 66 35 31 20 74 65 73 74 20 69 6e 70 75 74`'.

**Listing 6.2:** QEMU socat input/output

```
$ socat - UNIX:/tmp/nrf51_face.sock
system start
nrf51 test input /* typed by the user */
6e 72 66 35
31 20 74 65
73 74 20 69
6e 70 75 74
```

The same application binary that was executed in QEMU is downloaded on the hardware; then `minicom` is used to connect to the hardware through the serial interface. `minicom` is an interactive terminal emulation tool for UNIX. See Listing 6.3 for its output.

**Listing 6.3:** nRF51 minicom output

```
Welcome to minicom 2.7.1

OPTIONS:
Compiled on Aug 20 2018, 10:22:42.
Port /dev/cu.usbmodem0006815648711, 22:51:36

```

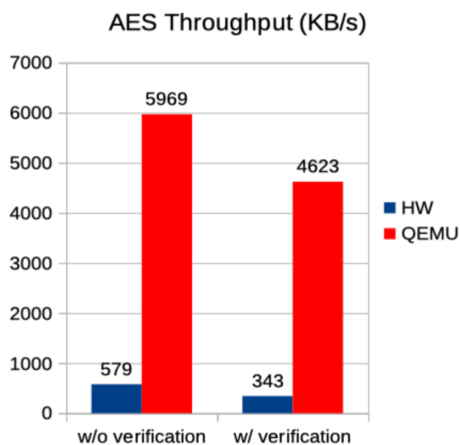



**Figure 6.1:** AES Throughput

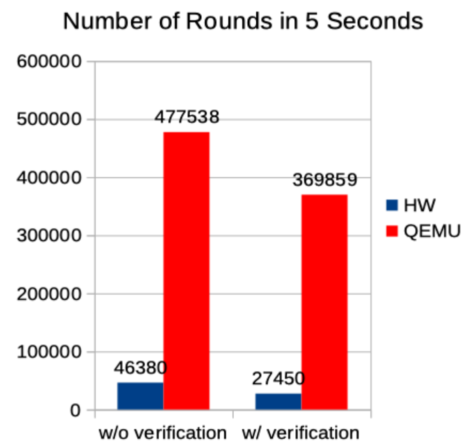



**Figure 6.2:** Number of AES Rounds in 5 seconds

```
Press Meta-Z for help on special keys

system start
6e 72 66 35
31 20 74 65
73 74 20 69
6e 70 75 74
```

The output shows the same hexadecimal data as we received from the QEMU instance. In this example, user input is not shown. Unlike `socat`, `minicom` does not print the user input by default. This experiment shows us that the UART implementation matches the physical hardware behavior.

### 6.2.3 Exp3: AES Performance

The AES performance experiment involves encrypting a fixed data block continuously for 5 seconds. The experiment is performed in two different ways. The encrypted data is verified against the expected ciphertext that resides on the program memory. The same experiment is also performed without any verification. Comparing the resulting encrypted data with the expected data introduces a performance penalty. Experiment results can be seen in Figure 6.1 and Figure 6.2.

Results show us that the QEMU instance surpasses the physical hardware performance multiple times. In Figure 6.2, we can see that the nRF51 board can do 46k rounds in 5 seconds while QEMU can do 478k rounds. Figure 6.1 shows almost 6000 KB/s AES throughput, while this is a positive trait for QEMU, it might be misleading for users that the actual hardware can exhibit the same performance.

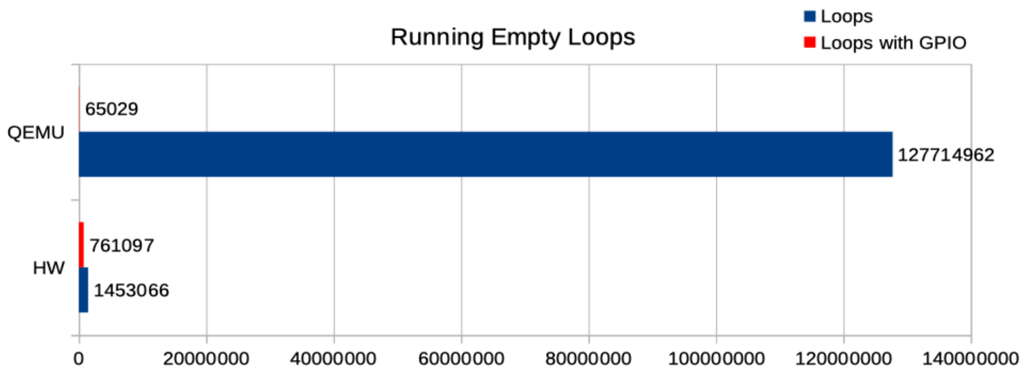



**Figure 6.3:** Number of Empty Loops in 1 Second

**Table 6.4:** Performance Gain in QEMU

| *Test* | w/ verification | w/o verification |
|---|---|---|
| *Throughput* | 13.5× | 10.3× |
| *Rounds* | 13.5× | 10.3× |

Table 6.4 shows how many times QEMU can run faster than the hardware for this performance test. It is no surprise that both throughput and AES round tests show the same performance gain. The tests without verification give us better results. The reason is that the nRF51 board uses a crypto accelerator for AES operations, which is faster than the microprocessor. Under QEMU this operation is performed by the host CPU. Both platforms use the CPU for the verification job. In this case, the verification process gives us reduced performance results for the nRF51 platform. If more CPU instructions are executed during AES tests, it is likely to get better results under QEMU.

### 6.2.4 Exp4: Raw Execution Time

This experiment is performed in two categories in order to understand the impacts of running pure application code with and without any peripheral interaction. See the source code in Listing A.4. So far, in the previous experiments, we have seen that QEMU outperformed physical hardware in terms of execution time. In this experiment, an empty loop is executed with pure ARM instructions. Later the same application code in the Listing A.4 is run, but this time, only LED toggle function is added to understand the performance impact better.

In Figure 6.3, we can see that QEMU can run 128M loops in one second while the physical hardware is only able to run 1.5M loops in a second. The most interesting point is that the nRF51 board has a reduced performance with 761K loops, almost 50% speed difference while the QEMU instance only run 65K loops with the same application code. There are two main reasons for this performance loss. The peripheral implementation sends out UDP packets, and it prints out GPIO state information on the QEMU console. These two actions require interaction with the host operating system, and they have to wait until completion. Performance can be improved by using another inter–process communication method such as shared

memory. Removing the print outs from the QEMU console will also reduce the time spent for operating system calls on the host. From this experiment, we can conclude that QEMU can execute native guest code a lot faster than the hardware, but the synchronization method used in the implementation introduced a great performance impact in the emulation. Also, printing any debug information has many adverse effects for fast operations.

**Table 6.5:** Performance Gain & Loss in QEMU

| Test | QEMU | nRF51 | Gain |
|---|---|---|---|
| Loops | 127714962 | 1453066 | 87.9× |
| Loops w/ GPIO | 65029 | 761097 | −11.7× |

Based on the previous test results, Table 6.5 shows the performance gain and loss in terms of magnitude. This experiment is the raw execution test of pure CPU instructions with and without GPIO peripheral interaction. While pure instructions can run approximately 88 times faster than the real hardware, having some GPIO communication causes the emulation to run approximately 12 times slower than the real hardware because of the reasons mentioned above.

### 6.2.5 Exp5: Simple GPIO using mbed Library

In this experiment, the mbed platform is used for testing to show that the implementation can work with this library. Upon startup, the mbed library initializes the required peripherals. This process is invisible to the user. It is only needed to include the relevant header files. Listing A.8 shows the source code used in this experiment. It is a simple loop for manipulating GPIO output using wait function at a regular interval. The library initializes the RTC, so that wait function is available to the user.

**Listing 6.4:** Script Output

```
(0xbeef) pin: 21 state: 1
(0xbeef) pin: 21 state: 0
(0xbeef) pin: 21 state: 1
(0xbeef) pin: 21 state: 0
...
```

In Listing 6.4, it shows that the output state for $pin_{21}$ is changed in every loop. In this case, $pin_{21}$ corresponds to $LED_1$ on the hardware. If this program is run on the nRF51 board, $LED_1$ blinks.

### 6.2.6 Exp6: Blinky LED Example from Zephyr

This sample project can be found under `samples/basic/blinky` in Zephyr tree. It simply toggles `LED1` on the development board at an interval of 1 second using the `k_sleep` function with RTC peripheral. It does not output anything on the console, but it is possible to see the state change for the LED pin in the router script output as seen in Listing 6.5.

**Listing 6.5:** Script Output

```
(0xbeef) pin: 21 state: 1
(0xbeef) pin: 21 state: 0
(0xbeef) pin: 21 state: 1
(0xbeef) pin: 21 state: 0
...
```

In this case pin 21 corresponds to `LED1` on the physical board.

### 6.2.7 Exp7: CPP Synchronization Example from Zephyr

This synchronization example is written in `C++`, and it demonstrates the synchronization between two threads. According to the description, this sample also demonstrates the basic sanity of the kernel.

The example can be found under `samples/cpp_synchronization` folder in Zephyr tree.

The application has a main and a secondary thread. Each of them outputs a text containing '`Hello World!`' in an infinite loop. It uses semaphores to avoid mixed output, and `printk` function is called in a critical section. Listing 6.6 shows the output from the application running in QEMU.

**Listing 6.6:** QEMU UART output from socat

```
$ socat - UNIX:/tmp/nrf51_beef.sock
***** Booting Zephyr OS 1.14.0 *****
Create semaphore 0x20000084
Create semaphore 0x20000070
main: Hello World!
coop_thread_entry: Hello World!
main: Hello World!
coop_thread_entry: Hello World!
main: Hello World!
coop_thread_entry: Hello World!
main: Hello World!
...
```

### 6.2.8 Exp8: Entropy Example from Zephyr

This example demonstrates the use of entropy functions. The use of these functions is the same on other supported boards. When it is compiled for the nRF51 platform, the driver implementation uses the RNG peripheral. The program outputs nine random numbers on the screen every second. This example can be found under `samples/drivers/entropy` folder in Zephyr Tree. Listing 6.7 shows the output from the virtual board.

**Listing 6.7:** QEMU UART output from socat

```
$ socat - UNIX:/tmp/nrf51_beef.sock
***** Booting Zephyr OS 1.14.0 *****
Entropy Example! arm
entropy device is 0x20000edc, name is ENTROPY_0
0xf5  0xa8  0x2e  0x11  0x78  0x85  0xb3  0xc2  0x66
0xf9  0x26  0x4a  0x06  0x95  0xfd  0xc4  0x25  0x11
...
```

### 6.2.9 Exp9: Fibonacci Benchmark Test & Analysis

This test runs the Fibonacci calculation function in order to calculate the Fibonacci number at the 47$^{th}$ position in the sequence for 50000 times. This test is performed under the Zephyr OS for its simplicity so that the operating system does the required peripheral configuration. Listing A.5 shows the source code used in the experiment. Listing A.6 and Listing A.7 show the assembly output for the '`main`' and '`fib`' function.

The purpose of this experiment is to calculate the amount of average time spent per instruction. Note that each instruction has a different execution time. There are many reasons for that difference such as CPU cache, SRAM access, size of the registers used in the instruction. Listing 6.8 and Listing 6.9 show the output from QEMU and the nRF51 hardware.

**Listing 6.8:** QEMU UART output from socat

```
$ socat - UNIX:/tmp/nrf51_beef.sock
***** Booting Zephyr OS 1.14.0 *****
start
fib(47) = 2971215073
ns spent:1007080
```

**Listing 6.9:** nRF51 UART output from minicom

```
***** Booting Zephyr OS 1.14.0 *****
start
fib(47) = 2971215073
ns spent:2653381347
```

In Listing A.6, there are 21 instructions. Since that function contains a loop, the total executed number of instructions is 596 in a single call. The Fibonacci function is called 50000 times from the 'main' function, and there are six instructions overhead for the loop and the call to `fib` function. So for this experiment, approximately $(596 + 6) * 50000 = 30100000$ instructions are executed between the start and stop time.

**Table 6.6:** Average time spent per instruction.

| Platform | QEMU | nRF51 |
|---|---|---|
| Total Time (ns) | 1007080 | 2653381347 |
| Avg./instr. (ns) | 0.334 | 88.152 |

The measurements will be the same with insignificant differences in time on the hardware. In QEMU, it is possible to see variable results, depending on the other processes working under the host operating system. According to these results, QEMU runs the code 2634 times faster than the hardware.

### 6.2.10 Exp10: RADIO Test

This experiment sends a packet from one node to another one in the network. The test can be found in the nRF51 SDK under `examples/peripheral/radio` folder. There are two applications, a transmitter, and a receiver, that should be run for this test. The transmitter program waits for a button to be pressed. This action is simulated by sending artificial GPIO events. After the transmitter receives a GPIO event, it sends the packet using the RADIO peripheral. The receiver application waits for the packet arrival event. When the two programs are run at the same time with an emulated GPIO input, the data that is being sent can be observed in the router script output. Listing 6.10 shows the output from the transmitter, and Listing 6.11 shows the output from the receiver.

**Listing 6.10:** QEMU transmitter UART output from minicom

```
Press Any Button
The packet was sent
The contents of the package was 1
```

**Listing 6.11:** QEMU receiver UART output from minicom

```
Wait for first packet
Packet was received
The contents of the package is 1
```

In Listing 6.12 we can see the entire contents of the packet that was sent over UDP socket. It contains the UDP packet header, RADIO packet header, and the actual content. The last byte represents the relevant data sent over the RADIO interface in this experiment. The RADIO and Bluetooth packets may contain some protocol–specific headers. It is possible to configure the length of header fields when the Bluetooth protocol is not used. Therefore, applications using the RADIO interface can have less data overhead.

**Listing 6.12:** Router Script Output

```
new client seen: ('127.0.0.1', 55743)
Client ID: 0xface
new client seen: ('127.0.0.1', 58463)
Client ID: 0xbeef
mode: 0
sending to: ('127.0.0.1', 55743)

('127.0.0.1', 58463):
01 00 00 03 01 00 01    | . . . . . . .
```

### 6.2.11 Exp11: RADIO Test with Zephyr Bluetooth Stack

Zephyr OS provides an open–source Bluetooth stack and allows developers to create Bluetooth enabled applications. Few sample applications come with Zephyr OS showing how to use the Bluetooth stack.

For this experiment, the most suitable application is the heart rate sensor example program. The samples can be found under `samples/bluetooth/central_hr` and `samples/bluetooth/peripheral_hr` in the Zephyr project folder.

The `peripheral_hr` application acts as a heart rate monitor device and sends random measurements over Bluetooth protocol. The other application, `central_hr` acts like a central device which can be a computer or a mobile phone in this case. It connects to the heart rate monitor device and reports measurings. In this experiment, virtual devices are able to detect each other and try to establish a connection. However, they fail to establish a connection and report/read measurements. Further analysis showed that the TIMER and the RTC implementations are not accurate enough to schedule interrupt events. The 'ticker' module in Zephyr OS checks the RTC counter and the last compare value that was set for a future compare event. If the difference is more than 3 ticks it tries to reconfigure the RTC device to avoid setting an event for a past time. In the case of an incorrect RTC configuration, the expected compare event will occur when the RTC timer overflows and reaches the expected counter value. It means that the event will occur in several hours while it is expected in 50 milliseconds. There are a few reasons for this issue. The implementation uses QEMU timers. They are not able to interrupt the execution of the guest system like the hardware modules. They are highly accurate but because of the nature of QEMU and translation blocks, it is not always possible to interrupt the virtual CPU execution at the exact moment. Translation blocks may contain more than one guest instructions and execution is only interrupted when the block execution is complete. In addition to that, there is an emulation overhead. QEMU runs an event loop, does some checks, runs other peripherals and other QEMU timers if requested. This deviation is not always observed. The reasons mentioned above can be observed in a non-deterministic manner. For example, other applications running in the host system may also affect this behavior.

While running two programs, we can observe their output and the data transmitted between them. Listing 6.13 shows the output from the heart rate monitor

device. It tries to establish a connection but fails with reason 62. According to Bluetooth Specification v4.0 [31] error code 0x3E means 'CONNECTION FAILED TO BE ESTABLISHED'.

**Listing 6.13:** peripheral_hr sample output

```
***** Booting Zephyr OS 1.14.0 *****
Bluetooth initialized
Advertising successfully started
[00:00:00.097,961] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
[00:00:00.098,022] <inf> bt_hci_core: HW Variant: nRF51x (0x0001)
[00:00:00.098,175] <inf> bt_hci_core: Firmware: Standard Bluetooth controller (0x00) Version 1.14 Build 0
[00:00:00.099,273] <inf> bt_hci_core: Identity: eb:96:87:a4:a8:ac (random)
[00:00:00.099,334] <inf> bt_hci_core: HCI: version 5.0 (0x09) revision 0x0000, manufacturer 0x05f1
[00:00:00.099,365] <inf> bt_hci_core: LMP: version 5.0 (0x09) subver 0xffff
Connected
Disconnected (reason 62)
```

Listing 6.14 shows the output from the central_hr device. It shows information about the discovered device but again, it fails with the same reason.

**Listing 6.14:** central_hr sample output

```
***** Booting Zephyr OS 1.14.0 *****
Bluetooth initialized
Scanning successfully started
[00:00:00.071,716] <inf> bt_hci_core: HW Platform: Nordic Semiconductor (0x0002)
[00:00:00.071,807] <inf> bt_hci_core: HW Variant: nRF51x (0x0001)
[00:00:00.071,929] <inf> bt_hci_core: Firmware: Standard Bluetooth controller (0x00) Version 1.14 Build 0
[00:00:00.073,852] <inf> bt_hci_core: Identity: eb:96:87:a4:a8:16 (random)
[00:00:00.073,913] <inf> bt_hci_core: HCI: version 5.0 (0x09) revision 0x0000, manufacturer 0x05f1
[00:00:00.073,944] <inf> bt_hci_core: LMP: version 5.0 (0x09) subver 0xffff
[DEVICE]: eb:96:87:a4:a8:ac (random), AD evt type 0, AD data len 11, RSSI -67
[AD]: 1 data_len 1
[AD]: 3 data_len 6
Connected: eb:96:87:a4:a8:ac (random)
Discover complete
Disconnected: eb:96:87:a4:a8:ac (random) (reason 62)
[DEVICE]: eb:96:87:a4:a8:ac (random), AD evt type 0, AD data len 11, RSSI -67
[AD]: 1 data_len 1
[AD]: 3 data_len 6
Connected: eb:96:87:a4:a8:ac (random)
Discover complete
Disconnected: eb:96:87:a4:a8:ac (random) (reason 62)
```

The router script output also shows the entire packet conversation between the devices. Figure 6.15 shows a single Bluetooth packet that contains the heart rate monitor device name, including the UDP packet header. When this sample program is run on the hardware, other Bluetooth enabled devices can list this device as 'Zephyr Heartrate Sensor'.

**Listing 6.15:** Router script output for the heart rate monitor sample program

```
...
('127.0.0.1', 59802):
01 00 00 27 01 03 44 1f | . . . ' . . D .
ac a8 a4 87 96 eb 18 09 | . . . . . . . .
5a 65 70 68 79 72 20 48 | Z e p h y r   H
65 61 72 74 72 61 74 65 | e a r t r a t e
20 53 65 6e 73 6f 72 00 |   S e n s o r .
00 00 00                | . . .
mode: 3
sending to: ('127.0.0.1', 60138)
...
```

## 6.3 QEMU Resource Usage

As expected, QEMU performs better on an average desktop computer since the platform being emulated is a low–power embedded device.

The host uses 100% of a single CPU core on the host platform if the embedded program does not have any methods to pause execution on the microprocessor when not needed. This approach causes unnecessary power consumption on any embedded platform. In general, each embedded program is designed to run a main loop. Even if the main loop is empty, the instructions are still executed. In an emulated environment, the behavior will not change. An empty loop will still be executed, but faster this time. The reason is that QEMU will always try to utilize the CPU at the highest rate, even if it exceeds the actual target hardware speed. If the guest CPU is suspended, QEMU will not execute any guest instructions.

The test programs in this chapter use the `WFI` instruction to wait for an interrupt in the delay function. `WFI` stands for wait for interrupt, and it suspends the execution by stopping the clock until an interrupt is raised. It is a tradition to use similar techniques to save power on an embedded device. If an embedded program keeps running when not needed, it will exhaust the host CPU as well, and this is not an implementation–specific behavior. QEMU simply keeps running the emulation. Observing 100% CPU usage on the host is an indication of inefficient program design in terms of power consumption unless it is intentional.

When the `WFI` instruction is used in guest applications, the CPU usage is usually less than 1% on the host. Periodic timer implementation in QEMU can cause higher CPU consumption up to 10% when the RTC or the TIMER module is used at high frequency in some cases.

Memory consumed by QEMU is reported as 8 MB by the operating system (macOS 10.13). The virtual machine implementation does not allocate any dynamic memory.

More test programs are publicly available on GitHub [32].

## 6.4 Hardware and Virtual Environment Comparison

Real hardware and virtual hardware can behave differently. Many factors are affecting this. Some examples:

- The host can perform faster operations and can act faster upon a request. For example, ECB encryption is relatively faster and does not allow the operation to be stopped in QEMU because an ECB operation is completed in a single cycle from the guest machine's perspective.
- The host is not affected by the environment and does not truly emulate the physical world. For example, electrical issues cannot be simulated; a cold or hot environment cannot be simulated.
- Some differences can be observed when faults occur on the real hardware as this implementation does not simulate any hardware errors.

## 6.5 Discussion

The fundamental purpose of this project has been to emulate the nRF51 platform in QEMU, and understand the issues with IoT emulation. In most cases, performance can be sacrificed to find and solve the defects in the application code. The experimental results suggest that the implementation provides a fast and convenient way to emulate nRF51 applications. Some cases can negatively affect the emulation, such as GPIO synchronization where the application suffers from a substantial performance penalty, but it is possible to reduce these issues with slightly different techniques.

The work conducted so far shows that it is feasible to use a virtual environment for development and debugging. According to the experimental results, implemented functionalities fulfill the expected outcome. The guest machine can use all the implemented functionalities through the SDK API.

The RTC and the TIMER modules require higher accuracy. QEMU itself has a timer implementation which depends on the host operating system. For that reason, nRF51 emulation fails to meet the real–time requirements, and some programs may break. QEMU is faster than the nRF51 platform but any context switch on the host operating system can introduce delays. Since the host timer does not stop during this kind of event, it affects the guest's execution as well. QEMU has different threads for various tasks such as the main event loop, timer handlers, file descriptor handlers, and guest read/write operations. In some places, it is required to use mutexes to avoid corrupting data structures and saving invalid values. Therefore one thread can block another one. These events may cause very small delays for the timer implementation. Several different solutions were tried, such as moving the TIMER and the RADIO implementation into dedicated threads, reducing background tasks on the host, using different host timer sources. There are other ideas that require testing:

- Running QEMU under a real–time operating system.
- Pinning the QEMU process into a single CPU core and disabling the execution of other programs on the same core.
- Implementing a custom timer in QEMU that runs in sync with the guest processor.

Although results support the expected outcome, some bugs have been discovered during testing. That means, more time needs to be invested in testing. As this project contributes to the open–source community, it will be available to everyone, and there is a high chance that people will contribute by using it for learning, development, or any other purpose. The increased amount of users would also help with finding bugs.

QEMU heavily depends on the community. It does not have any documentation, but it has mailing lists and many developers. All patches are submitted to mailing lists, so this is also a possibility to submit this project to the main QEMU repository.

# 7
# Conclusion and Future Work

## 7.1 Conclusion

All the work done so far shows that QEMU can support IoT projects in many aspects and improve productivity for embedded system developers. It is a widespread practice to evaluate computer software using virtual systems. Embedded systems have more limitations compared to desktop software; therefore, it is more challenging to work with IoT. The emulation of an embedded device extends our possibilities for debugging and testing. Developers and engineers become free from physical challenges. In turn, they can invest more time in development. Virtual systems give us an ability to automate all the steps required in a software development cycle.

This project focused on the nRF51 platform to show that it is possible and feasible to use emulation for IoT by providing implementation and running binaries in QEMU targeting nRF51 boards. For this project, it is safe to say that there are no concerns about the performance penalty by looking at the execution time comparison of physical and virtual systems.

QEMU provides a remote GDB interface that allows users to observe and manipulate the execution directly on the virtual CPU. It gives access to the memory and CPU registers as well and removes the need for having a microprocessor debugger.

QEMU is a project with a long history and has substantial support in the field. It does not focus on IoT entirely, but it supports dozens of embedded devices. People still do not use it much for IoT projects. That might be a result of the lack of complete support for target devices, but this project intends to change this view.

## 7.2 Future Work

In this project, most efforts were spent on the implementation and necessary evaluation of functionalities. There is a considerable need for evaluation with advanced IoT projects. Such projects can help us pinpoint the possible behavioural differences between a real and a virtual system. Since an embedded system is mostly used for controlling and communicating with other systems, real–world projects that include a network of devices would help to understand what parts to improve.

At this point, there are a few things that require a better solution:

- Virtual Inter–Device Communication: This part relies on UDP communication. It is likely to lose some synchronization info during execution. TCP would avoid this problem, but it is still susceptible to performance degradation in case of frequent state changes. Another problem is that this method

cannot provide an accurate simulation of I/O in terms of timing. It is possible to use inter–process communication and run all instances of QEMU in sync so that all virtual devices execute one CPU cycle at a time. This method would limit the possibility of using separate hosts for multiple virtual devices, but that is not necessary for the nRF51 platform because an average desktop computer already provides enough processing power to run multiple virtual machines in parallel.

- Power Statistics: Power consumption is quite critical for small–scale devices. Therefore, it would be possible to provide statistics by simulating processor and peripheral sleep states as well.
- Physical Interaction: QEMU and other similar host applications can take control of peripherals on the host and assign them to the guest system. For example, it is possible to interact with an external UART interface on the host, but it would be a significant improvement to do the same for an external Bluetooth interface and transmit physical radio packets from the virtual system.
- Full Peripheral Emulation: Some peripherals or functionalities were excluded from the project. For example, it is possible to read ADC value, but on the hardware, it depends on the physical voltage input and the voltage configuration. It is possible to simulate that as well.

## 7.3 Ethics and Sustainability

The number of IoT devices has been exponentially increasing in the last decade. That said, each day lots of devices are manufactured and sold around the world. This project will allow people to test their programs and try the development board out without buying it, thus reducing the number of chips produced. The nRF51 chip alone is relatively small but a development board contains many other side products.

During the process of a single microchip production, a considerable amount of different chemicals and fuel is used in many steps such as photoresist, washing, and etching. Therefore, emulators and similar applications may help us reduce the impacts we have on the environment.

Regarding the ethical perspective, equality of opportunity is required in every area of our lives. Whether it is for education or hobby, not every individual has access to the required tools such as development kits and debuggers in order to learn IoT systems. An open–source project like this will be available to anyone for learning and performing experiments without any physical equipment.

This project, may or may not have negative effects on the business of Nordic Semiconductor. Since the company also manufactures the nRF51 development kit, this emulator may reduce the number of boards sold in the long run. Assuming that the development kits are only a small fraction of the company profit, some trade-off should be made for the reasons mentioned above.

On the other hand, this project can reach more people and let them learn about Nordic Semiconductor chips, resulting in more product designs that use the nRF51 platform.

# Bibliography


[1] I. Pogarcic, D. Krnjak, and D. Ozanic, “Business Benefits from the Virtualization of an ICT Infrastructure,” *International Journal of Engineering Business Management*, vol. 4, p. 42, 2012.

[2] Fabrice Bellard, “QEMU, a Fast and Portable Dynamic Translator.” https://www.usenix.net/legacy/events/usenix05/tech/freenix/full_papers/bellard/bellard.pdf, 2005.

[3] “QEMU ARM Platform Documentation.” https://wiki.qemu.org/Documentation/Platforms/ARM, 2019.

[4] “Towards a Definition of the Internet of Things (IoT).” https://iot.ieee.org/definition.html, 2015.

[5] F. X. Ming, R. A. A. Habeeb, F. H. B. Md Nasaruddin, and A. B. Gani, “Real-Time Carbon Dioxide Monitoring Based on IoT & Cloud Technologies,” in *Proceedings of the 2019 8th International Conference on Software and Computer Applications*, ICSCA ’19, (New York, NY, USA), pp. 517–521, ACM, 2019. URL http://doi.acm.org/10.1145/3316615.3316622.

[6] “ESP8266 Overview.” https://www.espressif.com/en/products/hardware/esp8266ex/overview, 2019.

[7] “Getting Started Developers.” https://wiki.qemu.org/index.php/Documentation/GettingStartedDevelopers#Getting_to_know_the_code, 2019.

[8] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, “IoTPOT: Analysing the Rise of IoT Compromises,” in *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT’15, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2015. URL http://dl.acm.org/citation.cfm?id=2831211.2831220.

[9] S. Brady, A. Hava, P. Perry, J. Murphy, D. Magoni, and A. O. Portillo-Dominguez, “Towards an emulated IoT test environment for anomaly detection using NEMU,” in *2017 Global Internet of Things Summit (GIoTS)*, pp. 1–6, June 2017. URL https://ieeexplore.ieee.org/document/8016222.

[10] D. Ferraretto and G. Pravadelli, “Simulation-based Fault Injection with QEMU for Speeding-up Dependability Analysis of Embedded Software,” *J. Electron. Test.*, vol. 32, pp. 43–57, Feb. 2016. URL http://dx.doi.org/10.1007/s10836-015-5555-z.

[11] “nRF51 Series Reference Manual.” https://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.pdf, 2019.

[12] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, “Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration,” in

*Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '19, (New York, NY, USA), pp. 3:1–3:8, ACM, 2019. URL http://doi.acm.org/10.1145/3300189.3300192.

[13] "QEMU ARM Platforms." https://wiki.qemu.org/Documentation/Platforms/ARM, 2019.

[14] "QEMU with an STM32 Microcontroller Implementation." https://github.com/beckus/qemu_stm32, 2019.

[15] "Tricore VX Software Development Tools." https://www.tasking.com/products/tricore-vx-software-development-tools, 2019.

[16] "Hightec." https://hightec-rt.com/, 2019.

[17] "iSYSTEM winIDEA." https://www.isystem.com/products/software/winidea.html, 2019.

[18] "Trace32." https://www.lauterbach.com/frames.html?download_trace32.html, 2019.

[19] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable Sensor Network Simulation with Precise Timing," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, (Piscataway, NJ, USA), IEEE Press, 2005.

[20] "Contiki OS." http://www.contiki-os.org/, 2019.

[21] K. Roussel, Y.-Q. Song, and O. Zendra, "Using Cooja for WSN Simulations: Some New Uses and Limits," in *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, EWSN '16, (USA), pp. 319–324, Junction Publishing, 2016. URL http://dl.acm.org/citation.cfm?id=2893711.2893790.

[22] J. Eriksson, F. Österlind, N. Finne, A. Dunkels, N. Tsiftes, and T. Voigt, "Accurate Network-Scale Power Profiling for Sensor Network Simulators," in *Proceedings of the 6th European Conference on Wireless Sensor Networks*, EWSN '09, (Berlin, Heidelberg), pp. 312–326, Springer-Verlag, 2009. URL http://dx.doi.org/10.1007/978-3-642-00224-3_20.

[23] "TINA." https://www.tina.com/, 2019.

[24] H. Li, X. Xu, J. Ren, and Y. Dong, "ACRN: A Big Little Hypervisor for IoT Development," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, (New York, NY, USA), pp. 31–44, ACM, 2019. URL http://doi.acm.org/10.1145/3313808.3313816.

[25] A. Varga, "The OMNET++ discrete event simulation system," *Proc. ESM'2001*, vol. 9, 01 2001.

[26] A. Varga and R. Hornig, "An Overview of the OMNeT++ Simulation Environment," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, (ICST, Brussels, Belgium, Belgium), pp. 60:1–60:10, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[27] "OMNeT++." https://omnetpp.org/.

[28] "nRF51 Extension for QEMU." https://github.com/gvarol/qemu, 2019.

[29] "OpenMPI Project." https://www.open-mpi.org, 2019.

[30] "Zephyr Project." https://zephyrproject.org/, 2019.
[31] "Bluetooth Core Specification V4.0, Volume 2, p. 355." https://www.bluetooth.com/specifications/archived-specifications.
[32] "nRF51 Tests." https://github.com/gvarol/nrf51test, 2019.

# A

# Test Code

**Listing A.1:** LED Blinker

```
/* Modified version of 'Blinky' example from NRF51 SDK */
void delay(uint32_t cnt)
{
    for (volatile uint32_t i = 0; i < cnt; i++)
    {
        __asm("nop");
        __asm("nop");
        __asm("nop");
        __asm("nop");
        __asm("nop");
        __asm("nop");
    }
}

int main(void)
{
    /* Configure board. */
    bsp_board_leds_init();

    /* Toggle LEDs. */
    while (true)
    {
        for (int i = 0; i < LEDS_NUMBER; i++)
        {
            bsp_board_led_invert(i);
            delay(20000000);
        }
    }
}
```

**Listing A.2:** GPIO Test with RTC Module

```
int main(void)
{
    uint32_t err_code;
    const app_uart_comm_params_t comm_params =
        {
            RX_PIN_NUMBER,
            TX_PIN_NUMBER,
            RTS_PIN_NUMBER,
            CTS_PIN_NUMBER,
            APP_UART_FLOW_CONTROL_ENABLED,
            false,
            UART_BAUDRATE_BAUDRATE_Baud115200
        };
    APP_UART_FIFO_INIT(&comm_params,
                        UART_RX_BUF_SIZE,
                        UART_TX_BUF_SIZE,
                        uart_error_handle,
                        APP_IRQ_PRIORITY_LOWEST,
                        err_code);
    APP_ERROR_CHECK(err_code);

    printf("system start\r\n");
    leds_config();
    lfclk_config();
    rtc1_config(); //RTC1 is used for the delay function.
    gpio_init();

    //LED_1 will be ON (polarity setting)
    nrf_gpio_pin_clear(BSP_LED_1);

    for (;;)
    {
        printf("pin toggle\r\n");
        nrf_gpio_pin_toggle(BSP_LED_0);
        delay_tick(blink_delay); //blink_delay is 100 ticks
    }
}
```

**Listing A.3:** UART Hex Echo

```c
int main(void)
{
    uint32_t err_code;
    const char hex[] = "0123456789abcdef";

    const app_uart_comm_params_t comm_params =
        {
            RX_PIN_NUMBER,
            TX_PIN_NUMBER,
            RTS_PIN_NUMBER,
            CTS_PIN_NUMBER,
            APP_UART_FLOW_CONTROL_ENABLED,
            false,
            UART_BAUDRATE_BAUDRATE_Baud115200
        };

    APP_UART_FIFO_INIT(&comm_params,
                        UART_RX_BUF_SIZE,
                        UART_TX_BUF_SIZE,
                        uart_error_handle,
                        APP_IRQ_PRIORITY_LOWEST,
                        err_code);

    APP_ERROR_CHECK(err_code);

    printf("system start\r\n");
    bsp_board_leds_init();
    lfclk_config();
    gpio_init();

    for (;;)
    {
        static uint8_t pos;
        const int c = fgetc(stdin);
        const char l = hex[c >> 4];
        const char r = hex[c & 0xF];
        printf("%c%c ", l, r);
        pos++;
        if ( (pos & 0x03) == 0x00 )
        {
            printf("\r\n");
        }
        nrf_gpio_pin_toggle(BSP_LED_0);
    }
}
```

**Listing A.4:** Raw Execution

```
int main(void)
{
    uint32_t err_code;

    const app_uart_comm_params_t comm_params =
        {
            RX_PIN_NUMBER,
            TX_PIN_NUMBER,
            RTS_PIN_NUMBER,
            CTS_PIN_NUMBER,
            APP_UART_FLOW_CONTROL_ENABLED,
            false,
            UART_BAUDRATE_BAUDRATE_Baud115200
        };

    APP_UART_FIFO_INIT(&comm_params,
                        UART_RX_BUF_SIZE,
                        UART_TX_BUF_SIZE,
                        uart_error_handle,
                        APP_IRQ_PRIORITY_LOWEST,
                        err_code);

    APP_ERROR_CHECK(err_code);

    printf("system␣start\r\n");
    leds_config();
    lfclk_config();
    rtc1_config(); //this rtc is used for delay function.

    for(;;)
    {
        static volatile uint32_t rounds;
        rounds++;
        /* performance impact in QEMU */
        nrf_gpio_pin_toggle(BSP_LED_0);
        if (uptime > 100)
        {
            printf("Number␣of␣rounds␣in␣"
                   "empty␣loop:␣%lu\r\n", rounds);
            break;
        }
    }

    for(;;) {__WFI();}
}
```

**Listing A.5:** Fibonacci Experiment

```
#include <stdio.h>
#include <zephyr.h>
#include <arch/cpu.h>
#include <misc/printk.h>

u64_t __attribute__ ((noinline)) fib(volatile int n)
{
    u64_t prev = 1;
    u64_t cur = 1;
    u64_t next = 1;

    for (int i = 3; i <= n; ++i)
    {
        next = cur + prev;
        prev = cur;
        cur = next;
    }
    return next;
}
void main(void)
{
  u32_t start_time;
  u32_t stop_time;
  u32_t cycles_spent;
  u32_t nanoseconds_spent;
  volatile u64_t ret;

  printk("start\r\n");
  printk("fib(47)␣=␣%u\r\n", (u32_t)fib(47));
  /* capture initial time stamp */
  start_time = k_cycle_get_32();

  for (int i = 0; i < 50000; i++)
  {
    /* do calculation 1000 times */
    ret = fib(47);
  }

  /* capture final time stamp */
  stop_time = k_cycle_get_32();
  cycles_spent = stop_time - start_time;
  nanoseconds_spent = SYS_CLOCK_HW_CYCLES_TO_NS(cycles_spent);

  printk("ns␣spent:%u\r\n", nanoseconds_spent);

  for(;;);
}
```

**Listing A.6:** Fibonacci Function Assembly Output

```
00001f28 <fib>:
1f28:    b573       push    {r0, r1, r4, r5, r6, lr}
1f2a:    2100       movs    r1, #0
1f2c:    9001       str     r0, [sp, #4]
1f2e:    2001       movs    r0, #1
1f30:    2603       movs    r6, #3
1f32:    0002       movs    r2, r0
1f34:    000b       movs    r3, r1
1f36:    9c01       ldr     r4, [sp, #4]
1f38:    42b4       cmp     r4, r6
1f3a:    da00       bge.n   1f3e <fib+0x16>
1f3c:    bd7c       pop     {r2, r3, r4, r5, r6, pc}
1f3e:    1812       adds    r2, r2, r0
1f40:    414b       adcs    r3, r1
1f42:    0014       movs    r4, r2
1f44:    001d       movs    r5, r3
1f46:    0002       movs    r2, r0
1f48:    000b       movs    r3, r1
1f4a:    3601       adds    r6, #1
1f4c:    0020       movs    r0, r4
1f4e:    0029       movs    r1, r5
1f50:    e7f1       b.n     1f36 <fib+0xe>
```

**Listing A.7:** Fibonacci Experiment `main` Function Assembly Output

```
000004c4 <main>:
4c4:    b537        push    {r0, r1, r2, r4, r5, lr}
4c6:    4812        ldr     r0, [pc, #72]; (510 <main+0x4c>)
4c8:    f001 ff11   bl      22ee <printk>
4cc:    202f        movs    r0, #47 ; 0x2f
4ce:    f001 fd2b   bl      1f28 <fib>
4d2:    0001        movs    r1, r0
4d4:    480f        ldr     r0, [pc, #60]; (514 <main+0x50>)
4d6:    f001 ff0a   bl      22ee <printk>
4da:    f000 fb75   bl      bc8 <z_timer_cycle_get_32>
4de:    0005        movs    r5, r0
4e0:    4c0d        ldr     r4, [pc, #52]; (518 <main+0x54>)
4e2:    202f        movs    r0, #47 ; 0x2f
4e4:    f001 fd20   bl      1f28 <fib>
4e8:    3c01        subs    r4, #1
4ea:    9000        str     r0, [sp, #0]
4ec:    9101        str     r1, [sp, #4]
4ee:    2c00        cmp     r4, #0
4f0:    d1f7        bne.n   4e2 <main+0x1e>
4f2:    f000 fb69   bl      bc8 <z_timer_cycle_get_32>
4f6:    2300        movs    r3, #0
4f8:    1b40        subs    r0, r0, r5
4fa:    4a08        ldr     r2, [pc, #32]; (51c <main+0x58>)
4fc:    0021        movs    r1, r4
4fe:    f7ff fe99   bl      234 <__aeabi_lmul>
502:    044b        lsls    r3, r1, #17
504:    0bc1        lsrs    r1, r0, #15
506:    4319        orrs    r1, r3
508:    4805        ldr     r0, [pc, #20]; (520 <main+0x5c>)
50a:    f001 fef0   bl      22ee <printk>
50e:    e7fe        b.n     50e <main+0x4a>
510:    00002860    .word   0x00002860
514:    00002868    .word   0x00002868
518:    0000c350    .word   0x0000c350
51c:    3b9aca00    .word   0x3b9aca00
520:    00002877    .word   0x00002877
```

**Listing A.8:** GPIO Test with RTC Module

```
#include "mbed.h"

DigitalOut led(LED1);

int main ()
{
    bool state = true;
    while (1) {
        led = state;
        state = !state;
        wait (0.2);
    }
}
```