Emulation of the Crazyflie 2.1 Hardware for Embedded Control System Testing

Josefine Möllerström

Max Nyberg Carlsson

LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6135
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2021

Abstract

Embedded systems are hard to debug and the field of control software testing is relatively unexplored. Progress in these areas could provide better testing heuristics and safer systems. More specifically, hardware emulation is a potentially powerful tool that can help improve the speed and quality of the development cycle. Therefore, this study developed a hardware emulator for supporting embedded testing, debugging and development. As target hardware, the Bitcraze Crazyflie 2.1 quadcopter was used. The emulation was done in the open source framework Renode. The development of the emulator is accompanied by a discussion on the uses both in the industry and research environments. In order to set up the emulation, we extended Renode by implementing different peripherals such as sensors, the EEPROM and a basic timer. These, together with the created platforms were pushed to the forked Bitcraze Renode repositories. The repositories are open source and available for future research projects to use. The emulator allows interactive use to debug and explore the virtual system without extra hardware. It can also be setup to automatically test proposed firmware changes. The usage is showcased in several test cases, where different bugs have been injected into the firmware and then found using the emulation. The main goal for the thesis was to run the same firmware as used in real Crazyflies and pass a built-in, start-up, self test. Accuracy of hardware emulators is also an open research problem. As such, the thesis provides a thorough discussion on the accuracy of the proposed tool. The discussion also includes possible future work to improve it.

Acknowledgements

We would like to express our sincere thanks to our supervisor PhD student Claudio Mandrioli at the department of Automatic Control at LTH for his valuable support and knowledge during the thesis. Also, an extra thank you to him for the use of his illustration of the different testing levels.

In addition, we would also like to thank the members of Bitcraze AB for their interest and support. Special thanks to Marcus Eliasson for his insight and guidance, to Tobias Antonsson for his hardware expertise and to Arnaud Taffanel for his help with automatic testing.

Contents

1

Introduction

The introduction provides an explanation of the relevance and difficulty of firmware testing, with focus on the different levels of abstraction that can be used. It also discusses the purpose and the aim of this thesis as well as a description of the report structure.

1.1 Firmware Testing in Embedded Systems

Control systems today are often implemented digitally, with the control laws implemented as firmware and run on a microcontroller. Control calculations, in practice compared to mathematical models in a theoretical system, have limited precision and are not instantaneous. Therefore numerical properties and real-time capabilities of the microcontroller and written software are crucial for the correctness of the control system. The physical nature of embedded systems means that flawed implementations may have severe consequences including death [Leveson and Turner, 1993].

As firmware is developed, changes may unintentionally cause the implementation to diverge from the original design [Balasubramaniam et al., 2020]. Examples of changes include altering the type of a variable or allocating different amounts of memory which may have unexpected side effects. In order to assure the correctness of the control software the changes needs to be tested. One way to test for the effect of these changes is uploading the firmware to the microcontroller and running tests on the physical process. While accurate, assuming the hardware is working correctly, this method has some flaws.

Compiling and flashing new versions of the firmware and running tests on the physical process can be a slow and time-consuming task, which needs to be redone after each firmware change to ensure the feasibility of the changes. Since controllers evolve and are constantly updated throughout their lifetime [Balasubramaniam et al., 2020], speeding up tests may provide faster feedback and more time can be used for development instead of testing.

Non-technical reasons, such as economical or safety factors, may also limit the possibility to perform test scenarios on real hardware. Examples include rockets, usually single-use devices, and nuclear power plants, where experiments can be detrimental. If hardware is emulated, functionality could be removed or altered to identify how hardware faults affect the firmware. Emulated hardware also allows execution at a chosen timescale.

Despite simulations being faster than real world testing it is not used to the same extent in practice when for example developing service robots. One of the main reasons to why is the limitations of how well simulations capture real world behaviour [García et al., 2020]. Further research to find ways to improve correctness of simulations could prove fruitful to enable more usage of simulations.

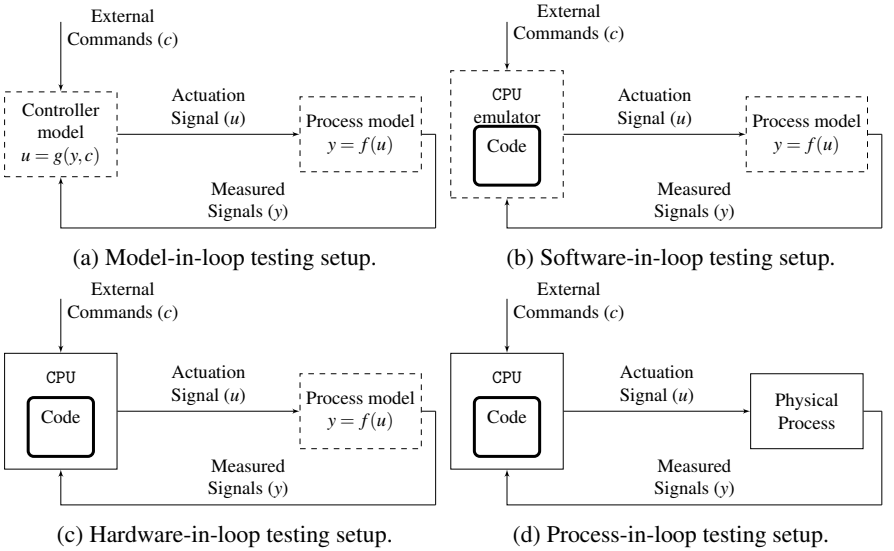
1.2 Levels of Control Firmware Testing

Failures are consequences of errors i.e. the system being in a faulty state. Errors occur due to anomalies in the software, commonly known as bugs, caused by human mistakes. There are several different levels of abstraction that can be used when designing and testing control systems to detect failures and finding these bugs. The level of abstraction depends on which parts are realised and which are virtual. Some of the most common concepts can be summarised in the four levels model-in-loop (MIL), software-in-loop (SIL), hardware-in-loop (HIL) and process-in-loop (PIL) with a decreasing usage of emulated components. While providing more accurate test results and covering more possible problems, using additional hardware can cause tests to be more expensive and time consuming. It may therefore be a waste to use additional hardware if problems are detectable with more emulated components of the system. What needs further research is which level of testing is optimal in a given situation, which could provide heuristics to streamline control software testing.

As seen in the setup presented in Figure 1.1a, the MIL level is based on the use of both a controller model and a process model. The MIL method is most commonly used when starting to design a controller and directly implements a mathematical model, such as a transfer function or state-space model. During MIL testing the simulated model of the physics of the plant is used to develop the controller. This is often done in simulation tools such as Simulink.[1] At this level one would expect to find problems with the control law, such as instability even during ideal circumstances and whether the performance is satisfactory.

From the controller algorithm designed during the MIL level the control software is written. During SIL testing this software is executed instead of using a mathematical model as done during MIL testing. The setup model for SIL testing is presented in Figure 1.1b. How this simulation is designed varies. In some cases

[1] `https://se.mathworks.com/products/simulink.html`



(a) Model-in-loop testing setup.

(b) Software-in-loop testing setup.

(c) Hardware-in-loop testing setup.

(d) Process-in-loop testing setup.

Figure 1.1: Comparison of the setup for the different testing levels. It shall be noted that the process-in-loop setup shown in Figure 1.1d is equivalent to the setup of the final implementation.

SIL means that the software is tested with the same plant model as in MIL but with the controller model replaced by executing firmware. In other cases it is tested in simulation software designed to be as close to the real intended hardware as possible [Sarhadi and Yousefpour, 2014]. During SIL testing the main focus would be on discovering software bugs and run-time errors.

HIL testing is where the target hardware is incorporated into the testing as illustrated in Figure 1.1c. Hardware in the loop is a real-time simulation where the plant and environment can still be simulated, while the software is run on real hardware such as embedded computers with actuators and sensors [Sarhadi and Yousefpour, 2014]. Incorporating the target hardware enables timing and communication problems to be discovered.

Testing the real setup, PIL in Figure 1.1d, should have the ability to catch every possible problem. In PIL testing everything is setup as the final version i.e the physical process is included in comparison to the HIL testing where it was simulated. The main problem is that it in most cases becomes harder to diagnose the problem and identify whether it is caused by the hardware or the firmware.

1.3 Purpose and Aim

In order to enable further research in control software testing, this thesis provides a case study of SIL modelling for the Crazyflie 2.1. Developed by Bitcraze,[2] the Crazyflie is a small indoor quadcopter used for both research and education.

The aim of the master thesis is to emulate the Crazyflie hardware in the open sourced software framework Renode.[3] The goal is that it should be possible to execute the unmodified firmware on the emulated hardware and pass the built-in self test.

The purpose of the emulated system is to be used both for improving development cycle speed for Bitcraze and for research to study control software testing methods. The feasibility of emulating the hardware in Renode is discussed as well as the feasibility of the hardware emulation itself. This thesis will thus provide both industrial and academic value.

The scope of the thesis is limited to SIL testing of the Crazyflie firmware with only the required peripherals emulated. The emulation was also limited to the main application microcontroller, handling the flight loop.

The requirements are therefore summarised as:

- Emulate the Crazyflie hardware using Renode
- Flash the unmodified firmware
- Pass the Crazyflie start-up self test
- Integrate with the Bitcraze development cycle
- Discuss feasibility of the hardware emulation

All of these requirements are met with the exception that the firmware had to be slightly modified, explained in Section 3.3.

1.4 Report structure

The following section summarises the different chapters of this thesis.

Introduction

The introduction contains a general background to firmware testing in embedded systems and different levels of testing. The chapter also explains the purpose and aim of the thesis as well as an outline of the report structure. A list of abbreviations and keywords is also presented.

[2] `https://www.bitcraze.io/`
[3] `https://renode.io/`

Background

In this chapter a background to the Crazyflie and the microcontroller used by the Crazyflie is presented. The software framework Renode is also presented with an explanation of how Renode platform files are built. An example of such a file is also given.

Emulator Implementation

In this chapter a description, of the method used when implementing the Crazyflie hardware in Renode, is given. The pre-existing emulation elements used are described while the specific peripherals that were modified or implemented are explained and described in more detail.

Example Usage

This chapter gives a brief explanation of how to use the emulator and run simulations in Renode with the use of Renode script files. An overview of different usage areas are presented and example test cases are given to showcase the usage.

Discussion

The advantages and drawbacks with the approach as well as the accuracy of the emulator are discussed in this chapter. Also discussed are the benefits and expectations of using the emulation and future work that can be done to improve the emulator.

Conclusion

This chapter concludes the report by summarising how the project went with reflections of the goals and if they were achieved.

Appendix A

Includes a description of how to install and use the emulator.

Appendix B

Includes the Renode script file used to load the Crazyflie platform.

Appendix C

Includes the Renode script file used for automatic testing.

Appendix D

Includes the shell script file used to automatically test if the firmware self test is passed.

1.5 Division of Labour

Both students involved in this project have, in all parts of the report and the emulation, provided an equal amount of time and effort. No specific division can therefore be presented. Both students have written and revised all documents, contributed to the code and the assembled the emulation to the same extent. Both students have also performed the simulations and tests.

1.6 List of abbreviations and keywords

Table 1.1: An overview of words and acronyms used in this report.

Keyword	Description
Crazyflie 2.1	A small expandable quadcopter with open source firmware, see Section 2.1
MCU	Microcontroller Unit
Flashing firmware	Uploading firmware from a host machine to the target system, saving it to flash memory or directly execute it from RAM
CI pipeline	Continuous integration, automatic tests run to check changes proposed to software
GDB	Gnu Debugger
OpenOCD	Open On-Chip Debugger, debug software for embedded system running on a host machine
I/O	Input/Output
MIL	Model-in-Loop, see Section 1.2
SIL	Software-in-Loop, see Section 1.2
HIL	Hardware-in-Loop, see Section 1.2

Continued on next page

Table 1.1 – Continued from previous page

Keyword	Description
PIL	Process-in-Loop, see Section 1.2
RESC	Renode Script, see Section 4.1
REPL	Renode Platform, see Section 2.3
STM32F405	MCU where the main firmware of the Crazyflie 2.1 is executed
nRF51822	MCU handling power, radio and expansion decks of the Crazyflie 2.1
KiB	2^{10} bytes
MiB	2^{20} bytes
SRAM	Static Random Access Memory
CCM	Core Coupled Memory, a type of work memory not connected to a DMA, meant for executable code and data
OTP	One-Time Programmable, memory location written to during production
DMA	Direct Memory Access, see Section 2.2
I^2C	Protocol to transfer data between two devices, see Section 3.3
1-wire	A type of communication protocol using only one wire
BMI088	Inertial sensor for motion in six degrees of freedom, see Section 3.3
BMP388	Pressure and temperature sensor, see Section 3.3

2

Background

This chapter gives a short background to the Crazyflie 2.1 hardware along with information about the firmware and a motivation to why the Crazyflie was chosen as a case study. Information regarding the microcontroller used by the Crazyflie and how it works is also presented. The chapter also gives general information about Renode and how the Renode platform files are built.

2.1 The Crazyflie 2.1 Quadcopter

The Crazyflie is an open source flying development platform developed by Bitcraze. The quadcopter, shown in Figure 2.1, weighs approximately 27 g. The size of the Crazyflie motor-to-motor, including the motor mount feet, is approximately 92x92x29 mm^3.

Figure 2.1: The Crazyflie 2.1.

The Crazyflie has been used in several different research projects. Among other things, it is often used in swarm control research [Preiss et al., 2017]. The CrazyS [Silano and Iannelli, 2020] is an extension of the Robot operating system RotorS, aimed to modelling, developing and integrating the Crazyflie in the physics based simulation environment Gazebo and it is used in several different research projects [Silano et al., 2019][Silano et al., 2018]. The Crazyflie is also used by hobbyist in smaller projects.

The firmware used, being open source, allows other users to modify it and contribute with changes. For task scheduling and synchronization, the real time operating system FreeRTOS[1] is used.

From the fact that the Crazyflie is designed for indoor flight and the firmware is open source it follows that the Crazyflie is a good case study. The firmware is also considered manageable and not too complex in comparison to the more general drone controllers like ArduPilot[2] and PX4[3] which both aim to support a variety of different boards. The Crazyflie may be simpler, but it is still more than an academic proof of concept. It still has many of the functionalities expected from any larger drone, like autonomous flight, radio communication, possibility of carrying a camera and streaming images.

There are two microcontroller units (MCU) on the Crazyflie, an STM32F405 controller connected to an nRF51822 controller. The STM32F405 is the main application MCU, handling e.g. the flight loop, whereas the nRF51822 is the radio and power management MCU. It is possible to add extension decks to the main deck, adding more features than provided by the base configuration.

Not all peripherals available on the microcontrollers are used for the Crazyflie. Which are used can be found by examining the Crazyflie firmware. The Crazyflie firmware is accessible at Bitcraze Github repository.[4] The ones that were added to the emulation are presented in Table 2.1. For information about the unused peripherals, see the reference manual [STMicroelectronics, 2019]. Aside from internal peripherals in the microcontroller, external peripherals such as an EEPROM, described in section 3.3, and sensors are soldered onto the Crazyflie board.

[1] https://www.freertos.org/index.html
[2] https://ardupilot.org/
[3] https://px4.io/
[4] https://github.com/bitcraze/crazyflie-firmware

Table 2.1: The peripherals added to the emulation and the ones not added. The used peripherals are presented in Chapter 3.

Peripherals added	Peripherals not added
Flash	DMA2D
CPU	ADC
SRAM	DAC
CCM	DCMI
OTP	IWDG
BitBanding	WWDG
NVIC	RNG
RCC	SPI
RTC	SDIO
USART 3, 6	USART 1, 2, 4, 5
EXTI	bxCAN
I^2C 2, 3	I^2C 1
DMA 1, 2	USB
GPIO	FSMC
Timer 2, 4, 7	Timer 1, 3, 5, 6, 8–14

2.2 The STM32F405 Microcontroller

As previously mentioned, the main MCU is the STM32F405 microcontroller. The MCU is manufactured by STMicroelectronics and uses a 168 MHz ARM Cortex-M4 core and several peripherals. Information regarding the peripherals were found in the reference manual [STMicroelectronics, 2019].

Peripherals

The microcontroller includes numerous peripherals for more functionality. The peripherals are connected to the core via buses in a bus matrix. Different peripherals are connected to different buses which limits how they can communicate.

Peripheral units include pins for inputs and outputs and are interacted with via registers. Those registers are memory mapped to one unique address each, which means that they are accessible via firmware instructions. One notable peripheral for this work is the inter-integrated circuit (I^2C) interface, used to communicate with external peripherals such as sensors and non-volatile memory.

The I^2C interface can connect several devices into an I^2C-network, although only two devices, a master and a slave, communicate simultaneously. Each device in the network has their own address from which they are identified. The master device initiates communication with the slave device by sending a start signal, which the slave acknowledges, and after each transferred byte either an acknowledged or not

acknowledged signal is sent from the receiver. Finally the transaction is finished with a stop signal from the master device.

Addressing

Using 32-bit addresses, divided into eight 512 MiB blocks as illustrated in Figure 2.2, the core can access peripheral registers and memory. Each block may also be divided into multiple sub-regions, e.g. the flash memory region with executable instructions starts at address 0x08000000 and is where the program counter should point to normally. The third memory block, starting at address 0x40000000, is mapped to the peripherals. This memory mapping enables configuring and control of the peripherals by simple read and write instructions. The three flexible static memory controller (FSMC) blocks, i.e. blocks 3, 4 and 5, are not used by the Crazyflie. Block 6 is never used by the STM32F405.

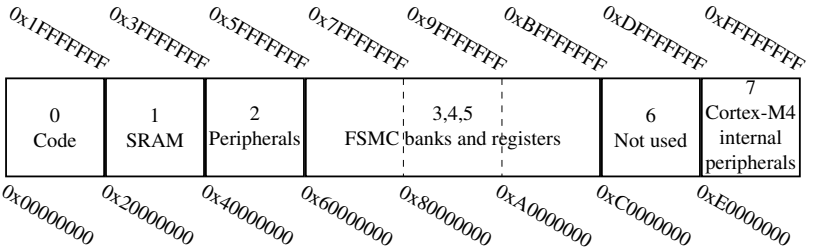


Figure 2.2: Memory mapping of the STM32F405.

Interrupts

Hardware interrupts allow hardware to signal the core that a certain event has happened or a condition is fulfilled. These signals are sent from peripherals but may be caused by external events. When an interrupt occurs it is masked i.e. decided whether the interrupt should be handled. After being masked the interrupt is appropriately prioritised relative other interrupts and may cause the core to execute a subroutine to handle the interrupt. Once handled the firmware can continue. Interrupts enable the firmware to respond immediately and may be configurable for each peripheral. An example is setting a peripheral to interrupt once data transfer has been completed.

Direct Memory Access (DMA)

As a way to transfer data without using instructions to the central processing unit (CPU), direct memory access can be used. During normal data transfer, the CPU is occupied for the entire duration of the read or write operations. With the DMA, the CPU initiates the transfer and then it does other operations while the transfer is performed by the DMA. This allows data to be sent between peripherals and memory or

between two memories connected by the same bus as the DMA peripheral without occupying the CPU. The STM32F405 has two DMA peripherals, each connected to different sets of peripherals depending on the buses.

2.3 Renode

Renode is an open source development framework that lets the user build and assemble virtual systems-on-chips (SoC). This is done using a variety of available cores, communication buses and interfaces. A SoC is an integrated circuit known as a chip that almost always include a CPU, memory and input/output ports along with other components. Everything is on a single microchip.

Firmware in Renode can be emulated as though flashed from ELF (executable and linkable format) binary files. This type of binary is built by default when making the Crazyflie firmware along a smaller binary file which normally is the file flashed to the real Crazyflie. The ELF file is larger and contains extra information useful for debugging.

Emulators in Renode are not cycle-accurate, the time required to execute instructions on real hardware is not reflected in the emulations. The core emulation method instead reflects the focus on fast simulations to achieve efficient function validations of large and complex systems [Herdt et al., 2020]. The time model is also made for multi-node systems. The highest time resolution in Renode, a "quant", is one millionth of a second. Each machine in an emulation has a local clock and resolution, allowing different machines to exist and synchronise with a given period.

Two alternative simulation platforms not chosen were QEMU [Bellard, 2005] and gem5 [Binkert et al., 2011], both focus on emulating core architectures instead of entire platforms. However, for testing control systems, the entire platform with actuators and sensors are needed. Considering the scope of this thesis Renode was chosen as the simulation framework. Note that the core emulation library, tlib,[5] used in Renode is based on QEMU.

Renode Peripherals

The peripheral modules are primarily written in C# and use features of object oriented languages, such as interfaces and inheritance, to build the communication connections between the peripherals. If a peripheral has registers, they can be implemented with an existing register class. The register class supports callback functions, which can be used to emulate hardware responses to registers being read or written. Certain bits or whole registers may be reserved, named or defined as read or write only, violating these annotations causes Renode to print a message informing the user of the violation. Public functions are used for interacting with other parts of the emulation or the user, typically functions to read or write registers or reading the state of a peripheral for example an LED.

[5] `https://github.com/antmicro/tlib/tree/master`

Some functions are also written in C and there is support for Python as well. Mainly, to make it possible to create and integrate simple Python scripts acting as peripherals. It is also possible to add hooks listening to certain events which execute Python scripts when triggered, such as when access to a peripheral is made.

There is an already existing library of peripherals in Renode. Renode therefore has built in support for several different microcontrollers and boards. There are also several board descriptions provided, one of them for a microcontroller from the STM32F4 family.

Renode Platforms

Platforms are described in Renode platform (REPL) description files and are used to attach peripherals to a machine. The syntax used when attaching peripherals is exemplified in Listing 2.1. Note that the indentations matters. Single line comments start with two slashes. Peripherals create instances of C# classes and attach them to a parent peripheral at the specified address. The parent peripheral can either be another peripheral such as the I^2C or the sysbus. The sysbus is used in the emulation as a base to which the peripherals can be attached. The given name is used to reference the peripheral via Renode and thus several instances of the same class can be used. Additional arguments for the object constructor can be provided when creating the peripheral, for example the memory size. Finally an arrow can be used to connect signals between peripherals, commonly used for interrupts. In the example, when the object named Interrupt in ClassName is activated input 3 in the peripheral interrupthandler is triggered. Further details are available in the Renode documentation.[6]

Listing 2.1: Example REPL file syntax.

```
// I am a comment

peripheralname: Namespace.ClassName @ parent 0x08000000
    numericConstructorField: 0x100000
    stringConstructorField: "template"
    Interrupt -> interrupthandler@3
```

REPL File Example

The example of a REPL file is presented in Listing 2.2. The listing, a subset of the finished Crazyflie platform, showcases how a complete platform with multiple peripherals added may look. No explanations of these peripherals are given here, as previously mentioned the details are presented in Chapter 3.

In the example a core is added to the sysbus. As mentioned the sysbus is the base used in the emulation to attach peripherals to. The added core is of the type

[6] https://renode.readthedocs.io/en/latest/

Cortex-M4 and the NVIC used by the core is defined as the peripheral named nvic. A flash memory of the size 0x100000 is defined at the address 0x08000000 followed by the definition of the NVIC peripheral named nvic. The EXTI is added and the different external interrupts are set to different NVIC interrupts.

A GPIO port, added at 0x40020C00 with size 0x400, is connected to EXTI channels while the second interrupt channel is connected to the LED. The LED is added by connecting it to the GPIO port and is an example of how some peripherals can be attached to other peripherals and not directly to the sysbus.

Listing 2.2: Example REPL file with multiple peripherals.

```
cpu: CPU.CortexM @ sysbus
    cpuType: "cortex-m4"
    nvic: nvic

flash: Memory.MappedMemory @ sysbus 0x08000000
    size: 0x100000

nvic: IRQControllers.NVIC @ sysbus 0xE000E000
    priorityMask: 0xF0
    systickFrequency: 72000000
    IRQ -> cpu@0

exti: IRQControllers.EXTI @ sysbus 0x40013C00
    [0-6] -> nvic@[6-10, 23, 40]

gpioPortD: GPIOPort.STM32F4GPIOPort @ sysbus <0
    x40020C00, +0x400>
    [0,1,3-15] -> exti@[0,1,3-15]
    2 -> ledB@0

ledB: Miscellaneous.LED @ gpioPortD
```

3

Emulator Implementation

Section 3.1 presents the outline of the method used when creating the emulation and the Renode platform. The pre-existing emulation elements used are explained in Section 3.2 followed by the specific peripherals modified or implemented for the thesis in Section 3.3. A summary of peripherals not included in the emulator concludes the chapter in Section 3.4.

3.1 Outline of Method

Since hardware emulation itself is not a well-established field, the implementation process used in this thesis could prove useful for future emulations. The method was based on the task to run the firmware and pass the start-up self test, not implementing the entire system and thus some peripherals were excluded.

The Crazyflie firmware was examined to get a better understanding of the Crazyflie 2.1 platform and how the firmware is structured. After understanding the general structure of the firmware, Renode was studied. The existing tutorials were read and done to get familiarised with the emulator framework before starting to implement the emulated platform.

As a reference for writing the platform description file for the STM32F405, the layout of the pre-existing STM32F4 platform description was used. While being a part of the same microcontroller family with several reusable peripherals, there were important differences between the pre-existing STM32F4 and the STM32F405 that had to be included in the implementation.

The implementation in Renode was done by going through the Crazyflie firmware and adding peripherals step by step to iteratively execute more of the firmware. Since not all available peripherals on the STM32F405 are used by the Crazyflie only the relevant ones were added. The unused peripherals could not be properly tested without major changes to the firmware and it was therefore decided that focus should be on the ones needed.

While the emulation should mimic the real system, some simplifications were made. As mentioned in Section 2.2 the real peripherals should be connected to dif-

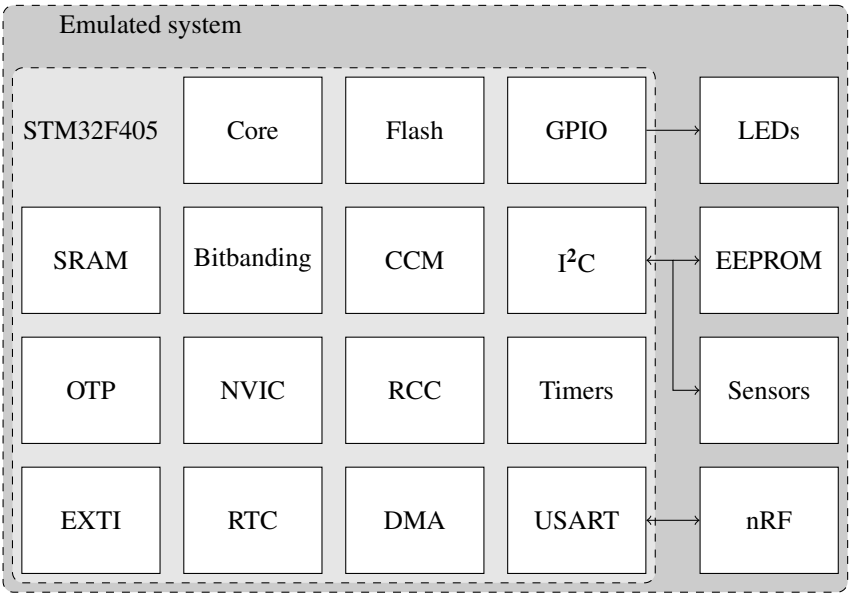


Figure 3.1: The emulated system illustrating how peripherals relate to the STM32F405. Those connected to the STM32F405 are attached to the sysbus. Sensors and the EEPROM are attached to I^2C peripherals and LEDs are outputs from GPIOs. Due to how it was implemented, the nRF is also attached to the sysbus.

ferent buses, but instead peripherals that are a part of the board were connected to the sysbus, explained in Section 2.3. This mainly affects how peripherals may interact and is further discussed in Section 5.3. The single sysbus was used because that is how Renode is structured and different platforms are built on the sysbus.

During the implementation Renode's `showAnalyzer` function was used to display debug print messages added to the firmware on the USART, described in Section 3.2. The log provided information when registers were read or written, with verbosity configurable for each peripheral. To Renode it was also possible to connect a GNU Debugger (GDB) session and e.g. stop at breakpoints at specific lines in the firmware source code to be able to read specific registers at certain times during a simulation.

Once the emulation worked, the peripherals were sorted into two REPL files, one containing the peripherals on the STM32F405 and one for the external peripherals added to the Crazyflie board. This is illustrated in Figure 3.1.

3.2 Existing Emulation Elements

Since a library of different components and peripherals already existed in Renode, there were some components that could be reused for the Crazyflie platform without any changes. Even though the peripherals already existed they had to be checked to make sure they worked as intended before they were used. Different properties and settings, such as memory sizes and interrupt locations, also had to be found and configured when putting the platform together.

Core

The core used in the STM32F405 is an Arm Cortex-M4 core. The Cortex-M4 core is intended for deeply embedded applications that require fast interrupt response features. Two of the processor features are the processor core and the nested vector interrupt controller (NVIC), which is explained later in this section. Emulation of the Arm Cortex-M4 core was already supported, handled by the C library tlib based on QEMU. As seen in the example given in Listing 2.2 the core is defined to be "cortex-m4" since the Cortex-M implementation supports other cores, such as "cortex-m3", in the Cortex-M series. How the NVIC used by the core is defined is also shown in the example.

Flash, SRAM, CCM and OTP

Memory peripherals implemented were the flash and one-time programmable (OTP) storage memory as well as static random access memory (SRAM) and core coupled memory (CCM) work memory. The main 1 MiB flash memory is where executable code is stored. As an example, Listing 3.1 shows how the flash memory was included in the STM32F405 REPL file. The 528 byte OTP is written to during production and contains e.g. an identification number of what platform the chip is mounted on. Adjacent to the OTP is a 30 KiB system memory used for a specific boot mode. The work memories consist of 128 KiB SRAM and 64 KiB CCM. The main difference between them is that CCM is only connected to the core and thus inaccessible from a DMA unit but also provides lower access latency.

For instructions to be executable in the emulation, the mapped memory peripheral operates at C level like the core emulation instead of C#. A memory peripheral on the C# level exists but is slower and code cannot be executed from it, for those reasons only the C peripheral was used. The C memory peripheral has size constraints and the peripheral used for the OTP therefore had to be larger than 528 bytes. This limitation caused the system memory and some reserved memory to be included in the emulated OTP peripheral, with a total size of 32 KiB.

Listing 3.1: Description of the flash memory.

```
flash: Memory.MappedMemory @ sysbus 0x08000000
    size: 0x100000
```

The firmware periodically writes an increasing value to a reserved memory location. As a way to suppress the warning messages and use the values written, a dummy memory peripheral was added at address 0xE0000000. Due to the same Renode limitations as for the OTP the peripheral had to be larger than one byte.

Bit-Banding

The Cortex-M4 supports bit-banding for the SRAM and peripheral registers. Bit-bands allow atomic operations to access register bits since addressing normally has a byte-wise resolution. The two bit-band regions start at 0x20000000 and 0x40000000 respectively as seen in Listing 3.2. To access the bit-band region bits, an aliased region, where each 32-bit word is an alias to one bit in bit-band region, is used. The two aliased regions in the STM32F405 starts at 0x22000000 and 0x42000000 respectively and both are of size 0x02000000.

Listing 3.2: Description of the bitbanding regions added in the REPL file.

```
bitbandPeripherals: Miscellaneous.BitBanding @ sysbus
    <0x42000000, +0x02000000>
     peripheralBase: 0x40000000

bitbandSram: Miscellaneous.BitBanding @ sysbus <0
    x22000000, +0x02000000>
     peripheralBase: 0x20000000
```

General-Purpose I/Os (GPIO)

The 9 GPIO ports, with 16 pins each, are used to connect certain external units, for example the LEDs. Only 4 GPIO ports were added to the emulator since the majority are unused in the Crazyflie. Internally when creating peripherals, GPIO objects can be used to emulate interrupts from peripherals. Thus when describing platforms, GPIO connections are made with the same syntax as interrupts.

Nested Vectored Interrupt Controller (NVIC)

The NVIC manages all interrupts including the core exceptions. The NVIC also handles the priority of the interrupts. To enable low latency interrupt processing and efficient processing of late arriving interrupts, the NVIC and the processor core are closely coupled. The NVIC provides 256 priority levels. The interrupts are masked and prioritised, then handled by the core via a function call from the NVIC or ignored.

For the emulation, interrupting peripherals must be connected to the NVIC in the platform description file. In the file they are also given their position in accordance to the reference manual i.e. the interrupt is connected to the NVIC, an example can be seen in Listing 2.2 where the EXTI is connected to the NVIC.

External Interrupt/Event Controller (EXTI)

The EXTI peripheral is used to signal an interrupt when triggered by a peripheral connected via GPIO. The real EXTI can also wake up the system when a wake up event occurs. This functionality is not used and therefore not implemented.

The EXTI was connected to the NVIC according to the STM32F405 reference manual. Up to 16 GPIO pins can be configured as external interrupts. In Listing 2.2 it can be seen how 15 GPIO pins are configured as inputs that may trigger interrupts.

Reset and Clock Control (RCC) & Real-time Clock (RTC)

In real systems the RCC controls whether peripherals should receive clock signals and be enabled. Although with no apparent use during the emulation, it was implemented with the help of the RTC based on another STM32F4 board described in Renode. The RTC, providing absolute time information such as current day of the week or year, is not used by the Crazyflie but the RCC dependency meant that it was added.

Universal Synchronous Asynchronous Receiver Transmitter (USART)

The USART peripheral is used to interconnect the STM32F405 with other systems. In the Crazyflie the USART communication interface is used to connect the two microcontrollers, the STM32F405 and the nRF51822. Debug messages may also be read via a USART connection using a built-in Renode monitor.

For the implementation of the USART in the emulation, the existing Renode peripheral STM32_UART was used. This was possible since the Crazyflie does not use the synchronised communication functionality specific to the USARTs.

Different USART peripherals e.g. USART3 and USART6, are described and added in the platform description file whereas connections between them are made via Renode commands. Different USARTs can be collected in hubs. When connected to a hub all USARTs will receive a message being sent by one of the USARTs. This property is used when mimicking the syslink, which is described in Section 3.3. External communication outside the emulation is possible by linking a USART peripheral to a socket server, allowing an external program to affect the emulation.

Light Emitting Diodes (LEDs)

The LEDs are external peripherals that are connected to the STM32F405 via the GPIO. An LED can print to the log when changing state (on or off). The signal for changing state is an example of a signal between peripherals that is not an interrupt. In Listing 3.3, output 2 from GPIO port D is connected to an undefined peripheral ledB which is then created, demonstrating how peripherals can be created in any order when describing platforms.

Listing 3.3: The blue LED connected to the STM32F405.

```
gpioPortD:
    2 -> ledB@0

ledB: Miscellaneous.LED @ gpioPortD
```

Direct Memory Access (DMA)

The DMA is a communications peripheral for transferring data, presented in Section 2.2. Because of the complex structure and the fact that the DMA appeared to work with the registers that were implemented, no changes were made to the pre-existing STM32_DMA peripheral. When running the emulation there are a few warnings from unhandled read/writes to unimplemented registers. These registers would control a buffer which is not implemented in the emulated peripheral and is unused by the Crazyflie. The settings the Crazyflie tries to write to the registers would disable the use of this buffer. Note how on a physical system, the bus connections would limit what peripherals a DMA could transfer to and from. In the emulation there are no such hardware limitations and this is discussed further in Section 5.3.

3.3 Modified and New Peripherals

Whereas the previous section described pre-existing peripherals that could be used without any modifications, the following where either implemented or modified.

Inter-Integrated Circuit (I^2C)

The I^2C connects the STM32F405 with external peripherals. One master device, here the I^2C peripheral on the STM32F405, controls the communication to a slave device with a clock signal and data is transferred via a data signal. The flow of messages is handled via physical start and stop conditions i.e. when the clock signal is high and data signal switches. The master requests either reading or writing when initiating the transfer by sending an address with a read-or-write bit to all connected peripherals. Note that only 7-bit addressing is used in the Crazyflie although 10-bit addressing is possible. As an example, when writing to the EEPROM the byte 0x51 would be sent from the master whereas 0x50 would be used for reading. The corresponding peripheral then acknowledges and the rest of the communication, between the master and the slave, is peripheral dependant.

The emulated version is based on writing data to a buffer and then burst transferring the data as an argument to a function in the receiving peripheral. There is a C# interface which must be implemented by all classes that connect via I^2C. The interface mainly specifies the signatures of Read and Write functions.

Using the original I^2C implementation caused issues during transfer of data to peripherals. The status register for when byte transfer was completed got set before

all data was sent, causing the firmware to believe the transaction was finished without sending all data. In an attempt to fix this issue, a task is scheduled with a delay to wait for more data to be transferred. If no extra data has been received, the I^2C byte transfer finished register is set. The I^2C peripheral now sends one extra byte which can be handled by ignoring the final byte in the receiving peripheral.

When reading over the I^2C interface, direct memory access is used by the Crazyflie firmware which for unknown reasons does not work in the emulated system. By adding a function in the firmware to enable DMA before every read, everything works as intended. This was the only functional change made to the firmware and has been incorporated into the real firmware, tagged for future removal. The impact of the I^2C problems is discussed in Section 5.2.

With the current implementation of the I^2C, the connected peripheral does not know how many bytes should be read. For this to be possible the read function would need to be rewritten. Further work to be done on the I^2C is discussed in Section 5.4.

Timers

The STM32F405 has three types of timers: the advanced, general purpose and basic timers. Of the three types only the latter two are used by the Crazyflie firmware.

The general purpose timers have all the features of a typical timer-counter module. Some of the basic uses are pulse width modulation (PWM) generation, input capture, time-base generation and output compare. Basic timers are in comparison strictly used for time-base generation purposes since they have neither I/O channels for input capture nor PWM generation. The advanced timers are however very similar to the general purpose timers, but they have the additional ability to generate complementary PWM signals and to generate brake and dead-time for such signals.

The existing STM32 general purpose timer in Renode did not allow reading or writing 16-bit words. This was changed so that 16-bits words could be read and written as well.

While the general purpose timers are used by the motors, the basic timer is used when timing is needed with microsecond resolution in the firmware. Several registers and functionalities regarding the capture and compare ability, were removed from the general purpose timer to create the basic timer peripheral used in the emulation.

Electrically Erasable Programmable Read-Only Memory (EEPROM)

The EEPROM is storage memory used to store configuration data such as the radio address. It is re-writable with a physical pin used to toggle write protection, which is grounded in the Crazyflie i.e. always disabled. Communication is done via I^2C.

No EEPROM peripheral existed, thus the 24AA64 EEPROM had to be implemented using the I^2C interface. Aside from read and write functions, the default

stored data is the default Crazyflie radio configuration which can be changed easily. Although the EEPROM is a storage peripheral, memory access is not trivial since communication is done via the I²C interface. Whenever data should be read or written two addresses are needed, one for the EEPROM peripheral and one internal pointing to the memory location of interest. This can be compared to memory mapped storage such as the flash which only requires one address.

The 8 KiB available memory is divided into 256 pages, each 32 bytes and writing data is done either as a single byte or multiple in a single page. If more bytes would be written than the available space in the current page, a wrap-around happens to the beginning of the page instead of continuing to the next. This could lead to accidentally overwriting data and further complicates the implementation.

The three read modes available are current address, random and sequential read. Current address mode reads the next byte following the internal address pointer, random address mode first receives the address to be read from and in sequential mode consecutive bytes are read until a stop signal is received [Microchip, 2012]. Given the way the I²C is implemented in Renode, the EEPROM cannot know how many bytes are read (even though the Read function in the I²C-peripheral interface takes an integer argument) and thus the entire memory buffer is returned upon reading as illustrated in Listing 3.4. As such sequential read, which should increase the address pointer for each byte read, followed by current address read does not work in the emulation since only one increment is made.

Listing 3.4: The EEPROM read function without comments and logging commands.

```
public byte[] Read(int count = 1)
{
    byte[] result = new byte[storage.Length];

    var ap = ((ushort)highAddress<<8)+lowAddress;
    Array.Copy(storage, ap, result, 0, storage.Length - ap);
    Array.Copy(storage, 0, result, storage.Length - ap, ap);

    lowAddress++;
    if(lowAddress == 0)
    {
        highAddress++;
        highAddress &= 0x1F;
    }
    return result;
}
```

nRF/Syslink

In the real platform, the syslink connects the STM32F405 to the nRF via a USART connection. The syslink is the name used in the Crazyflie firmware for this communication connection. In the emulation the nRF was implemented as a simple peripheral echoing back the received messages. This was achieved by letting the mock nRF be a simplified UART peripheral named CF_Syslink. As previously mentioned, in the "USART" subsection in Section 3.2, the communication was achieved using a hub. The function for sending back a message is presented in Listing 3.5.

Aside from power and radio management, the nRF also handles attached expansion decks via 1-wire, which is the communication bus used for the expansion decks. When a message is sent to the nRF, the STM32F405 expects a response. This is usually the sent message echoed back but exceptions exist. As seen in Listing 3.5 the case when the message is `SYSLINK_OW_SCAN` a different message, telling the emulated STM32F405 that no expansion decks are attached, is returned. How this implementation could be improved is discussed in Section 5.4.

Listing 3.5: The function for the emulated nRF to echo back messages after receiving and storing data in `receiveFifo`.

```
private void SendBack()
{
  byte[] data = receiveFifo.ToArray();
  switch(data[2])
  {
    case 0x20: // SYSLINK_OW_SCAN
      byte[] OwScanData = CreateMessage(0x20,
        0x01, new byte[]{0x00});
      for(int i = 0; i < OwScanData.Length; ++i)
      {
        CharReceived?.Invoke((byte)OwScanData[i]);
      }
      receiveFifo.Clear();
      break;
    default:
      while(receiveFifo.Count > 0)
      {
        CharReceived?.Invoke((byte)receiveFifo.Dequeue());
      }
      break;
  }

  this.Log(LogLevel.Noisy, "Complete data sent back!");
}
```

Sensors

There are two sensors used by the Crazyflie. A BMI088, with both a gyroscope and accelerometer, and a BMP388 barometer. Since neither of these sensors were pre-existing in the Renode library they had to be implemented based on existing Renode C# interfaces similarly to other sensors. The BMI088 is effectively two sensors, a gyroscope and accelerometer with separate pins and registers, and those were therefore implemented separately. Communication with the sensors is done via the I^2C and the interrupts are set by EXTI and GPIO pins. The inclusion of the sensors in the REPL file is shown in Listing 3.6. Due to the way the interrupts are wired according to the Crazyflie schematics,[1] only the interrupts from the gyroscope are used by the firmware. Because of this only the interrupts in the gyroscope were implemented.

Listing 3.6: The sensors used by the Crazyflie platform.

```
bmi_gyro: Sensors.BMI088_Gyroscope @ i2c3 0x69
    Int3 -> exti@14

bmi_accel: Sensors.BMI088_Accelerometer @ i2c3 0x18

bmp_baro: Sensors.BMP388 @ i2c3 0x77
```

The sensors can parse mock data from a text file, which can be used to test the emulation's ability to handle data values read from the sensors via the I^2C. Data may also be fed directly via the Renode interface when running the emulated system. Due to the way the firmware works, data from the sensors is read when interrupts from the gyroscope occur. It is possible to trigger the gyroscope interrupt using an external signal sent over sockets.

The BMI088 unit is configurable to select the measured data range and resolution by writing to configuration registers. This allowed the conversion from given input data to corresponding bytes to be performed in the sensor unit. When read by the microcontroller, these bytes are interpreted as the measured data. The BMP388 is calibrated during production and calibration data is stored in the unit. Since these values are specific for each sensor, data from a real unit was extracted and hard coded into the emulated one.

The sensors include built-in hardware self tests performed within the sensors. How these checks are performed by the sensors is not provided in the sensor manuals [Bosch, 2020a][Bosch, 2020b]. The results of these self tests are read from from specific registers. Since the tests could not be performed in the emulated sensors and would only indicate faulty hardware, the default values of said registers were changed so the tests succeeded i.e. they always return the value for success when

[1] https://www.bitcraze.io/documentation/hardware/crazyflie_2_1/crazyflie_2.1_schematics_rev.b.pdf

they are read by the firmware. The changed default value for the gyroscope is shown in Listing 3.7.

Listing 3.7: Three of the defined register in the gyroscope included the GyroSelfTest with the modified default value. In real hardware the default value is 0x00.

```
Registers.Int3Int4IOConf.Define(this, 0x0F)
    .WithFlag(0, name: "int3_lvl")
    .WithFlag(1, name: "int3_od")
    .WithFlag(2, name: "int4_lvl")
    .WithFlag(3, name: "int4_od")
    .WithReservedBits(4, 4);
Registers.Int3Int4IOMap.Define(this, 0x00)
    .WithFlag(0, out int3Data, name: "int3_data")
    .WithReservedBits(1, 1)
    .WithFlag(2, out int3Fifo, name: "int3_fifo")
    .WithReservedBits(3, 2)
    .WithFlag(5, out int4Fifo, name: "int4_fifo")
    .WithReservedBits(6, 1)
    .WithFlag(7, out int4Data, name: "int4_data");
Registers.GyroSelfTest.Define(this, 0x12);
```

3.4 Excluded Peripherals

Depending on relevance and if they already existed in Renode, not all peripherals used by the Crazyflie were implemented. The USB port, while important for the real Crazyflie if radio communication is not possible, was excluded since it is unused if no USB connection is detected. Since power management is handled by the nRF, the power controller was not implemented. The analog-to-digital converter (ADC) did not already exist as a Renode peripheral and is only used for battery level measurement, therefore it was excluded. The independent watchdog (IWDG) was excluded since there was no built-in peripheral and the RTC occupied the memory space used for the watchdog. The emulation runs and signals are sent to the watchdog, but there is no check if the software hangs.

4

Example usage

This chapter explains some basic functionality of the Renode simulation using the Crazyflie platform and Renode script files. Furthermore, a brief overview of different areas where the emulation can be useful is presented along with test cases to showcase the usage.

4.1 How to use

The custom Renode version is available for download from Bitcraze repositories using the same build instructions[1] as the standard version. The Crazyflie platform was added to the Bitcraze Renode repository[2] whereas the modified or implemented peripherals explained in Section 3.3 were pushed to the Bitcraze Renode infrastructure repository.[3]

The Renode user interface is a custom command-line. Sequences of commands can be collected in Renode scripts (RESC) files and automatically executed, see Appendix A for further details how to manually use Renode.

Two custom RESC files are provided for the Crazyflie platform, available at the Bitcraze repository. Calling `crazyflie.resc` sets up a machine using the Crazyflie REPL file. These command lines from the RESC file are presented in Listing 4.1.

Listing 4.1: Example of the command lines from the `crazyflie.resc` used to create a machine and loading the `cf2.repl` file.

```
mach create "CF2.1"
machine LoadPlatformDescription @platforms/boards/cf2.
        repl
```

After the REPL file has been loaded, data is written to the one-time programmable memory area in order to identify the emulated Crazyflie as "CF21".

[1] `https://renode.readthedocs.io/en/latest/advanced/building_from_sources.html`
[2] `https://github.com/bitcraze/renode/tree/crazyflie`
[3] `https://github.com/bitcraze/renode-infrastructure/tree/crazyflie`

This emulates the data written during production. The firmware binary is loaded, peripherals are connected and finally some mock data is loaded for the gyroscope to pass its firmware initialisation test. The command line for loading some data to the gyroscope is given in Listing 4.2. The entire `crazyflie.resc` file is given in Appendix B.

Listing 4.2: Example of the command line from the `crazyflie.resc` used to load some mock data to the gyroscope.

```
# Two initial data samples required to pass
        initialisation and test
sysbus.i2c3.bmi_gyro FeedGyroSample 10 20 30 2
```

In `crazyflie_test.resc`, the `crazyflie.resc` script is called followed by some hooks getting set. These hooks enable logging of debug messages from the USART to a log file and automatic termination once a certain line is printed, which can be used for firmware testing. An example of a hook from `crazyflie_test.resc` is given in Listing 4.3 while the entire `crazyflie_test.resc` file is given in Appendix C.

Listing 4.3: Example of the command line from the `crazyflie_test.resc` used to set a hook to exit Renode when "Free heap" has been printed in the USART.

```
sysbus.usart3 AddLineHook "Free heap" "Antmicro.Renode.
        Emulator.Exit()"
```

4.2 Usage areas

There are several different areas where Renode and SIL testing could be used. These include being used as an interactive development tool, as a part of a CI pipeline for automatic testing and for further studies of control software testing.

Development Tool

When no hardware is available, the emulated hardware can be used as an interactive development aid. Compared to only emulating the core without peripherals, emulating the entire platform includes the hardware functionality and side-effects from peripherals and how they affect the registers. As the emulation depends only on software, the test results are easier to repeat than when running them on real hardware. With the possibility to save and load system states, tests can be shared and reproduced. It is also easier to create faulty states that are difficult to reproduce in physical hardware and test them in an emulation. One example could be corrupt EEPROM data which can be created by changing the initial data.

Emulation enables checking register values whenever desired using external debugging software, something not always possible in real systems e.g. pausing a

system mid-flight. With the program counter available and debug information in the executed binary, it is also possible to easily trace execution at a specific point in time to the corresponding source code line.

As an example it is possible to check that all necessary peripherals are correctly enabled when adding functions or enabling the use of peripherals in the firmware. If a peripheral or necessary bit is forgotten, the emulation would either generate a warning or it would not be possible to perform the requested task. Additionally, it is easier to get information of what might be wrong from the emulation than connecting the hardware, in this case the Crazyflie, to an external debugger and with the use of breakpoints try to find the cause of the error.

Continuous Integration

Continuous integration (CI) involves automatically testing firmware changes. The simulation is started and writes the USART output to a log file. After a certain amount of time or once a success message is written, the simulation stops. Depending on the result the new firmware either passes or fails the tests, with a log file available for debugging.

By including Renode in a CI pipeline, the simulation is automatically run whenever a firmware change is proposed. The automatic tests performed may also be expanded to check more than the start-up self test and additional tests can be added. This enable users to discover errors, such as peripherals not being enabled correctly, before flashing to actual hardware. In open source projects like the Crazyflie firmware there are numerous different contributors. Although the contributions must be manually approved, an extra check in the pipeline acts as one more barrier to help prevent bugs in the firmware. It is also possible to get more information from the simulation than from a test that checks if building works. This will aid Bitcraze in their work to check and make sure the code is correct, since the open source nature results in contributions from outside the Bitcraze team.

Research of Control Software Testing

The study of control software testing is a relatively unexplored research area. Systems with tests at different levels can be used as case studies. The emulator developed in this master thesis is available as open source for future researchers to further investigate the field of testing control software.

A research direction enabled by this work is the comparison of different levels of abstraction in testing of control firmware. The result of the simulation, being software-in-loop, after introducing bugs can be compared to the same bugged firmware run on hardware-in-loop and the real process. This would make it possible to compare the different testing methods and analyse their strengths and weaknesses for different classes of bugs.

It would also be possible to compare SIL with the other levels in regards to how easy and useful they are to work with during development. Additionally, the

complexity of setting up the level and getting started can be compared.

4.3 Test Cases

In order to demonstrate how the emulator can be used as a tool for testing, different bugs were injected in the firmware and the emulation was run to automatically test whether the firmware self test is passed. During the automatic testing, bugs expected to be discovered are those that either cause the firmware to crash or a part of the self test to fail. If a bug affects something not checked in the self test or uses unimplemented parts of a peripheral, it cannot be expected to be found. With criteria based on the self test, the utility of the automatic test depends on how well the self test detects failures.

The modified firmware, where the bug has been introduced, was compiled and copied into the Renode folder to use the correct path in the `crazyflie.resc` file. The test is then performed by a shell script created as a general test model, see Appendix D for the full shell script. The script starts the simulation by loading the `crazyflie_test.resc` file and initiating a timeout clock to catch the cases where an infinite loop occurs. The test script is successful if the line `Self test passed!` is found in the log file written from the USART. As mentioned in Section 4.1, the second hook, shown in Listing 4.3, terminates the simulation after the self test has been passed. For the sake of the automatic test, the emulation is no longer interesting once the self test has given a result. The check to determine if the test was successful is then made and if the line is found the script exits with a zero exit code. The test fails if the line cannot be found or if the script timeouts, both resulting in a non-zero exit code.

By changing the firmware and introducing bugs different errors occurs, resulting in failed tests when using the automatic test shell script included in Appendix D. Five test cases used to illustrate different failures are presented in the following subsections.

Case 1 — Assertion fail

The EEPROM is used to store the configuration block, data with e.g. the Crazyflie radio address, and the connection, via I^2C, must be initialised. This is done when the configuration block is initialised. Not initialising this connection, as done by deleting line 131 from `src/modules/src/system.c` and shown in Listing 4.4, leads to a null pointer assertion, from FreeRTOS, to fail. Both the program counter and stack pointer are set to zero and the core emulation terminates. If running the automatic test script, this results in a timeout since the termination occurs before reaching the hook exiting and closing the simulation.

Listing 4.4: src/modules/src/system.c

```
131  configblockInit();
132  storageInit();
133  workerInit();
```

Case 2 — Infinite loop

The firmware uses DMA to transfer data over the syslink to the nRF. By not enabling the DMA, achieved by removing line 266 in Listing 4.5, sending data is impossible. Since data cannot be sent the semaphore that is taken on line 267 will never be given. This blocking causes the system to never continue the start-up initialisation, which results in a timeout when running the automatic test since the emulation never exits.

Listing 4.5: src/drivers/src/usart_syslink.c

```
264  USART_ClearFlag(UARTSLK_TYPE, USART_FLAG_TC);
265  /* Enable DMA USART TX Stream */
266  DMA_Cmd(UARTSLK_DMA_STREAM, ENABLE);
267  xSemaphoreTake(waitUntilSendDone, portMAX_DELAY);
268  xSemaphoreGive(uartBusy);
```

Case 3 — Illegal address

By altering the number of iterations when initiating the motors, as done in Listing 4.6 where NMB_OF_MOTORS is increased with one, an index can exceed the bounds of an array. The data where the extra element would be is interpreted as an extra element. When the firmware assumes a structure has an address to a function and is called, the code at that address is executed. Depending on other values in memory that address might be illegal, 0xA in this case as shown in Figure 4.1 after attempts to read from and write to non-implemented peripherals, which causes the core emulation to halt since nothing can be executed from that address and the automatic test timeouts.

Listing 4.6: src/drivers/src/system.c

```
109    for (i = 0; i < NBR_OF_MOTORS+1; i++)
110    {
111      //Clock the gpio and the timers
112      MOTORS_RCC_GPIO_CMD(motorMap[i]->gpioPerif, ENABLE);
```

```
15:36:37.9961 [WARNING] sysbus: [cpu: 0x8007C96] ReadDoubleWord from non exis
ting peripheral at 0x3C.
15:36:37.9961 [WARNING] sysbus: [cpu: 0x8007C96] ReadDoubleWord from non exis
ting peripheral at 0x40.
15:36:37.9961 [WARNING] sysbus: [cpu: 0x8007C96] WriteByte to non existing pe
ripheral at 0x42, value 0x0.
15:36:37.9961 [WARNING] sysbus: [cpu: 0x8007C96] WriteByte to non existing pe
ripheral at 0x41, value 0x0.
15:36:37.9961 [WARNING] sysbus: [cpu: 0x8007C96] WriteByte to non existing pe
ripheral at 0x40, value 0x0.
15:36:37.9962 [WARNING] sysbus: [cpu: 0x8007C96] WriteByte to non existing pe
ripheral at 0x3F, value 0x2.
15:36:37.9962 [WARNING] gpioPortB: Unhandled write to offset 0x2C, value 0x4.
15:36:37.9962 [WARNING] gpioPortB: Unhandled read from offset 0x28.
15:36:37.9962 [WARNING] gpioPortB: Unhandled write to offset 0x28, value 0x0.
15:36:37.9963 [WARNING] sysbus: [cpu: 0xA] ReadByte from non existing periphe
ral at 0xA.
15:36:37.9963 [ERROR] cpu: CPU abort [PC=0xA]: Trying to execute code outside
 RAM or ROM at 0x0000000A.
```

Figure 4.1: The final log output from test case 3 before crashing. Accessing memory with reads and writes precedes an attempt to execute code at an illegal location.

Case 4 — Failed self test

By not initialising the worker queue, as done in Listing 4.7 where line 133 has been removed from `src/modules/src/system.c`, the start-up self test (which explicitly includes a worker queue test) fails. The failure is printed in the output log, but the firmware still continues since the failure is non-fatal. When running the automatic test the simulation exits automatically but the test script exits with a non-zero exit code, indicating an error, since the expected self test passed line cannot be found. Note that the worker queue initialisation is completely software based, no specific peripheral is initialised, and is used to buffer tasks to be done. This indicates that the emulation can find bugs in the firmware not directly linked to a certain peripheral.

Listing 4.7: `src/modules/src/system.c`

```
131  configblockInit();
132  storageInit();
133  workerInit();
134  adcInit();
135  ledseqInit();
```

Case 5 — False success

A case not caught by the emulation is where the command to enable DMA requests on line 135 in `src/drivers/src/i2c_drv.c` has been removed, as seen in Listing 4.8. Running this bug on the emulated system causes no problems while on the real hardware the I^2C communication fails.

Listing 4.8: src/drivers/src/i2c_drv.c

```
131 // Disable buffer I2C interrupts
132 I2C_ITConfig(i2c->def->i2cPort, I2C_IT_EVT | I2C_IT_BUF,
        DISABLE);
133 // Enable the Transfer Complete interrupt
134 DMA_ITConfig(i2c->def->dmaRxStream, DMA_IT_TC | DMA_IT_TE
        , ENABLE);
135 I2C_DMACmd(i2c->def->i2cPort, ENABLE); // Enable before
        ADDR clear
```

5

Discussion

In this chapter the advantages and drawbacks with the approach used when developing the emulator is discussed. The chapter also presents the benefits and expectations of using the emulation and uses the test cases presented in previous chapter as examples. A discussion on the accuracy of the emulator and further work conclude the chapter.

5.1 Approach

The main approach used to construct the emulator was to add the peripherals one at a time, based on the order they are initialised in the firmware. Due to the way the Crazyflie firmware is structured, with the initialisation functions being called in `system.c` before starting the main loop, this could easily be done. By doing so it is possible to follow along using debug prints and log messages to make sure each peripheral works as intended. Advantages with this approach are that it is easier to find problems as they are introduced and get early results before the whole system is described. A flaw of this approach is that for it to work, a correct and working firmware must already exist. It also assumes that the firmware is written in a readable and manageable way, where it is possible to identify and initialise one part at a time.

The approach used built-in debug messages from the Crazyflie firmware to get feedback from the firmware during the emulation. There were several benefits with using the debug prints. No external debug program is needed to get information since the information can be printed over the USART alongside pre-existing messages providing additional context. It is also possible to get an acknowledgement on how far the firmware executes. Debug prints are also customisable and the results easily repeatable, during execution the same messages are printed without interaction from the user.

The drawbacks consist of the fact that source code needs to be changed and then reverted. The firmware has to be compiled as well after making each change. This leads to several compilations since the process is iterative and the messages are

refined in several instances. If the firmware was larger and not built as efficiently this could become a time consuming inconvenience. Another problem discovered was that if an interrupt occur while a message was being written on the USART then the interrupt might lead to other messages being written before the first is completed. It is thus not a thread safe operation.

The work required to implement an emulator of this kind should decrease as the library of implemented peripherals in Renode increases. This because the peripherals can be reused in different platforms and fewer have to be created. The complexity of creating new peripherals is reflected by how advanced the real peripheral is. A more advanced peripheral will lead to a more complex emulated peripheral implementation. If a similar peripheral already exists it can be used as a template and simplify the implementation of the new peripheral.

The sensors all use the same structure as existing I^2C sensor peripherals. Once the gyroscope had been implemented, as explained in Subsection "Sensors" in Section 3.3, it was used as a template for the accelerometer and the barometer. As mentioned in that same section, the barometer needed calibration data from a real unit. This required extra work since the data had to be extracted from a real unit. This also means that the emulated sensor represents a unique sensor which is further discussed in Section 5.3. Even though the sensor structure was based of other sensors, the more advanced barometer required more work. The implementation was also hindered by the fact that not all details were disclosed in the sensor manuals. This can also limit how well some peripherals can be implemented without the involvement and support of the manufactures.

The EEPROM had no similar peripheral and thus had to be implemented from the I^2C interface, using functions to read and write data as explained in Subsection "EEPROM" in Section 3.3. The approach to this implementation was for the write function to sequentially write all data received to the storage as described in the manual, with page overflow emulated with modular arithmetic. Since the I^2C peripheral sends one extra byte, explained in Section 3.3, the final byte received is ignored. If the I^2C is fixed, all peripherals communicating with this protocol must be changed accordingly to use the complete message. The I^2C caused more problems when implementing read functionality, since no information is given to the EEPROM how many bytes should be read. Therefore, while all three EEPROM reading modes are implemented, the internal address pointer cannot update correctly. Thus the approach to implement missing peripherals after that peripherals manual can be limited by the existing peripherals in Renode, as in this case, where the I^2C limited what the EEPROM could do.

5.2 Benefits and Expectations

As mentioned in Section 4.2 there are three main usage areas. The first is as a development tool which may be the largest area. The main reason for this is that there are several possible ways to interactively use the emulator. As mentioned it is for example possible to access memory registers and check that individual bits are set correctly.

The emulator runs the Crazyflie firmware, the same as used for real hardware. In order to print messages to the used USART two pre-existing compile flags in `tools/make/config.mk`, described in Appendix A, are required when compiling the firmware. The only functional changes made were a workaround to manually enable the DMA when reading from I^2C peripherals and extra debug messages printing if certain components of the self test fail. These changes have been incorporated in the Crazyflie firmware, with the workaround noted to be removed if obsolete from further development of the emulator.

Development tool

In test case 2 the programmer is assumed to have missed enabling the DMA i.e. line 266 in Listing 4.5 has not been added. It is possible from inspecting the log and Crazyflie debug messages to make an educated guess that the problem occurs due to either the USART used for the syslink communication or the DMA used with said USART. This guess can be made based on the last prints before the loop, where access to those peripherals was done. By using the command `readDoubleWord` in Renode to read the registers of those two peripherals, it is possible to see that the bit that enables transfer has not been set in the configuration register for stream 7 at offset 0xB8.

Test cases 1 and 3 fail in different ways but both are caused by similar errors where the program counter points to illegal values from where code cannot be executed. In the first case the address is zero and automatically caught by FreeRTOS, null pointers are probably common enough for this assertion to always be made. In the latter case the address is faulty due to an index exceeding the boundary of an array. The program counter became non-zero but happened to be outside executable memory space which causes the emulation to fail. Depending on the data stored after the array this could lead to an arbitrary function executing, possibly not cause the system to fail directly and complicate debugging.

In the same way as for the first three cases, test case 4 would be discovered first when flashing to the hardware. For the physical Crazyflie, the debug prints for the self test are visible when connected to the Crazyflie client. The messages for the failed self test would therefore be visible once the firmware has been flashed. However actually finding the error would be much harder since no other log output is given. Figuring out that it is the worker queue that has not been initialised and therefore cause the system fail message is harder.

Since these errors are discovered during the simulation and not compilation of

the firmware it follows that the errors would be discovered after flashing to the hardware. However they would still be discovered in the sense that the Crazyflie would not work. To actually find the error causing the failure would have required more work. Debugging embedded systems is difficult since they do not run a proper operating system and therefore require the use of extra hardware for interaction. In this case the STM debugger link needs to be established with a debugger like OpenOCD[1] on another machine which takes some time to set up. Executing inside an emulator in a multi-purpose computer makes is possible to easily access the state of the processor and detect bugs without the use of extra hardware.

One more example where emulated hardware can be useful is during work with peripherals that are hard to debug, such as the EEPROM with page overflows and I^2C communication which requires additional overhead. In the emulator, it is possible to access and see the entire EEPROM to check that it is written to as expected. An alternative to emulating could be to formally specify the behaviour to prove whether an implementation should work. Being able to try the firmware directly however enables a more interactive development method.

Continuous Integration

As seen in the test cases the simulation is useful in finding different types of errors. Using the simulation in a CI pipeline, described in Section 4.2, would therefore be possible. The main deciding factor would be the actual correctness of the emulation and how well it represents the physical hardware, in other words the accuracy. The accuracy of the emulation is discussed in Section 5.3.

Tests can be either successful or result in a failure. The test is successful if it passes the start-up self test. When testing there are two faulty conclusions that may be done, false failure and false success. Neither of the cases should ever occur in a perfect emulator and have different effects during automatic testing. False failures means that working pull requests are rejected. If the proposed changes that fail the simulation are proven to work on hardware and therefore are accepted it will lead to future simulations failing as well. The emulator would require an update based on the failed case to implement the missing logic, which would improve the accuracy of the emulator but need extra work. False successes would cause bugs to pass through as though the simulation was not a part of the CI pipeline. They would hopefully be detected when run on real hardware and then fixed. Although not causing the emulated system to fail, it might be possible to use the emulator as an interactive tool to fix the bug.

If the emulation is accurate it would mean that the rate of false failures and false successes would be low to non-existing. An example of a false failure, where a temporary workaround had to be added to the firmware, has already been seen when the DMA was not enabled automatically during I^2C transfer. Test case 5 is an example of a false success, commenting out the I2C_DMACmd enable has no effect

[1] `http://openocd.org/`

in the emulation since the DMA requests are not emulated properly for the I^2C. However, in the real hardware the line has to be included for the I^2C communication to work properly.

Research of Control Software Testing

The main focus in this area is the possibility the Crazyflie offers as a case study in future work. Some of the reasons for why the Crazyflie was chosen for this project have already been mentioned in Section 1.1. With a working emulation of the hardware, with the possibility to run simulations, the Crazyflie's potential for software testing increases. It is then possible to set up test studies for the different test levels using the same target product. This enables the test levels to systematically be compared and different types of bugs to be classified depending on expected visibility in different layers.

As mentioned before, the Crazyflie is already used in several control research projects. Since the emulation is open source and publicly available it can be used as a research tool in those projects as well and not only for control software testing. The open source nature also means that the emulator can be adjusted and modified to suit different use cases.

5.3 Accuracy

One of the main problems with emulators is how accurate they are, does the emulated system mimic the real system? With the approach used to construct the emulator, this is directly influenced by how well the actual hardware is described by the reference manuals, as an example the STM32F405 reference manual used has been revised 17 times in just over eight years. Undocumented functionality might cause side-effects when real hardware is used. These may cause firmware to function on hardware but not when emulated or vice versa.

Not all parts of the platform are required for the simulation to pass the self test. The current version only uses one bus to connect all peripherals while the real microcontroller has several, physically limiting how e.g. DMA can be used. This means that faulty address configuration of a DMA unit could work in the emulated system but fail on real hardware. There are also differences in maximum frequencies between buses. How the lack of this affects emulation accuracy has not been investigated.

Not implementing unused peripherals can actually be more useful than implementing them. If something goes wrong during the emulation and attempts to access unused peripherals are made, as in test case 3 and shown in Figure 4.1, the warning messages can help to locate the bug. If there was no function call there would be no crash since reads and writes are in most cases non-fatal operations. If that occurred on real hardware the error would possibly not be easily detected. It would be harder to locate the bug on an emulator including all peripherals as well, since the warnings

about non existing peripherals would disappear. For test case 3, the failure would still be discovered because of the fatal error but that may not always be the case.

For the barometer implementation the calibration data was taken from a real Crazyflie. Thus any calibration errors of that drone would be present in the emulation. This only affects the measured values fed to the barometer. If the emulation should be used for project where accuracy of those values are of importance this should be considered, otherwise it does not affect the emulation itself.

The memory peripheral used for the OTP had to be larger than what it should be due to Renode limitations. The extra address space occupied is reserved by the MCU though and should not cause problems since the MCU does not use it in the emulation.

The emulation does not occur in real-time and, as mentioned in Section 2.3, Renode is not cycle-accurate. Choosing between an emulation tool that is cycle-accurate and one that is not, is a trade-off where the user has to prioritise. What is considered more important, fast results or high accuracy and does the extra accuracy really matter? While using Renode, we have not noticed any problems regarding the fact that Renode is not cycle-accurate. On the other hand we have not done anything where it would have been important such as hard real-time constraints at the time resolution of the duration of a single cycle. The user also has to weigh in the increasing complexity of trying to create a cycle-accurate emulation. Although in a more complex proposed model for emulating RISC-V architecture [Herdt et al., 2020], high cycle accuracy could be achieved but with an increased overhead cost. Future and more exact models might make cycle-accurate emulators more common.

One obvious problem with the test cases is how they were designed. The test cases were designed to show how bugs can cause visible failures, which causes a bias from the authors. Tools to randomly mutate the source code, as used by [Balasubramaniam et al., 2020], could be used to create a large set of test cases to investigate. Those changes would depend on possible mutations but not be designed to specifically cause certain failures.

5.4 Further Work

In order to improve the emulation, the peripherals mentioned in Section 3.4, which are used by the physical Crazyflie but not included in the emulated version, could be implemented. Notably the independent watchdog, which causes the system to reset unless it receives periodic writes.

The nRF microcontroller did not have the same priority as the STM32F405 where the main firmware is executed. This led to the syslink only providing the necessary functionality for the self test to pass. By connecting to an external program, or another Renode system, more advanced cases could be tested. Apart from the 1-wire connection to expansion decks, the nRF handles the radio connection and should periodically send battery information. With those properties emulated,

the simulation could be connected to the client used for controlling real Crazyflies.

The simulation runs through the firmware start-up and enters the work loop. Since no signal to start flying is sent to the Crazyflie, in-flight states and associated tasks performed are never tested. While this work focused on the firmware passing the start-up self test, the simulation could be extended to test the virtual Crazyflie in flight with an external physics simulator. Generated data can be fed directly to the sensors and, with the syslink extensions above, signals to fly could be sent to the system. This could be used to test extreme conditions and edge cases or how the firmware handles faulty sensor readings. It would also be possible to set breakpoints mid-flight, something impossible on a real system, to check registers and other interesting memory addresses.

As mentioned before the main problem found was the I^2C implementation. To get a better implementation of the I^2C, for which the workaround added to the firmware no longer is needed, is one of the most important things to fix. Apart from requiring a change in the firmware, the I^2C is such an important peripheral in the Crazyflie and used as a communication bus in so many instances. It is for example used by the sensors and the EEPROM.

Although the simulation works and successfully passes the self test, there are still things that can be done to improve the accuracy of the emulation. If in-flight testing during development is of interest then some extra nRF and sensor functionality should be prioritised. For continuous integration, focus should probably be on the I^2C and other peripherals to improve the accuracy.

6

Conclusions

Overall the project was a success, we managed to get the firmware to pass the self test when run on the emulated hardware. Unfortunately a small addition had to be made so the firmware was not unmodified. Fixing the emulation to handle the I^2C and DMA situation would probably require remodelling how both peripherals are implemented.

The emulation can be used as a development tool and work as a research test case. The utility for both cases can be expanded depending on the situation. The simulation has not yet been integrated into the CI pipeline. While ready to be used, some of the additions for extra emulation accuracy discussed in Sections 5.3 and 5.4 might be wise to make before using the results as more than warnings. If false failures occur, an automated test could cause more harm than good.

In Section 1.1 we stated that running tests on the physical process can be a slow and time-consuming task. During this thesis we have seen that once an emulation exist it is easy and fast to use. We have also seen that feedback is given immediately and the information is easy to access.

We concluded that it is possible to emulate hardware in Renode but that the time effort is dependant on what already has been implemented. The accuracy may also vary because of the complexity of some peripherals limiting how well they can be implemented.

A

Readme

Installation

Cloning & Dependencies

1. Clone and install all dependencies for making the firmware.[1]
2. Clone the custom Renode[2] repository.
3. Install dependencies according to Renode build instructions.[3]

Building

1. Compile[4] the Crazyflie firmware. In order to get outputs via the USART, uncomment `CFLAGS += -DDEBUG_PRINT_ON_UART` in `tools/make/config.mk` and add `CFLAGS += -DENABLE_UART1`. These flags are mandatory for continuous integration and provides more information when running the emulator.
2. Build[5] Renode. The first time might take a while.

Starting Renode

To start Renode run

```
mono output/bin/Release/Renode.exe [option] [script]
```

[1] `https://github.com/bitcraze/crazyflie-firmware`
[2] `https://github.com/bitcraze/renode`
[3] `https://renode.readthedocs.io/en/latest/advanced/building_from_sources.html#core-prerequisites`
[4] `https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/building-and-flashing/build/`
[5] `https://renode.readthedocs.io/en/latest/advanced/building_from_sources.html#building-renode`

For a complete list of launch options, call with option `--help`. To automatically load a Renode script file when starting add the `script_name.resc` as script input.

Example Usage

Loading a Renode script file

A .resc file is loaded with `include @script_name.resc`. The .resc file contains Renode commands that also may be run manually. In Renode, paths are prefixed with an `@` and can be printed with the `path` command.

Controlling the emulation

Start the emulation with `start` or `s`. Pause with `pause` or `p`. To reset the Renode instance use `Clear`. Quit with `quit` or `q`.

Read a register

To check the content of a register of a certain peripheral run for example `sysbus.dma1 ReadDoubleWord 0x10` where 0x10 is the offset.

Show analyzers

Output from the USART buses, here the usart6 peripheral, can be shown with `showAnalyzer sysbus.usart6`.

Logging

To change logLevel (verbosity of the log) run
`logLevel [level] [option] [peripheral]`.
To create a logFile run `logFile @some_file_nam`.
For more information see documentation on Using the logger.[6]

Sensor data

Data input values can be added to the sensors directly or from files with
`sysbus.i2c3.bmi_gyro FeedGyroSample x y z [repeat]` or
`sysbus.i2c3.bmi_gyro FeedGyroSample 'filename'`
`sysbus.i2c3.bmi_accel FeedAccSample x y z [repeat]` or
`sysbus.i2c3.bmi_accel FeedAccSample 'filename'`
`sysbus.i2c3.bmp_baro FeedPTSample press tmp [repeat]` or
`sysbus.i2c3.bmp_baro FeedPTSample 'filename'`

[6] `https://renode.readthedocs.io/en/latest/basic/logger.html`

Connect to GDB server

The running emulation can be debugged using GDB. Assuming the provided script file is run, start the appropriate GDB version and connect with `target remote :3333` where the port can be configured in `board.resc`. Start the emulation and continue via the debugger as usual. For more information see the documentation on Debugging with GDB.[7]

Connect via network socket

The provided script starts a socket terminal on port `3456` which can be used to signal an interrupt from the gyroscope by sending a byte. This causes the firmware to read data from the sensors.
By enabling the terminal on port `3457` messages sent to the nRF can be read. This could be used to further improve the emulation and how the syslink is handled. For more information see the documentation on UART integration.[8]

[7] `https://renode.readthedocs.io/en/latest/debugging/gdb.html`
[8] `https://renode.readthedocs.io/en/latest/host-integration/uart.html#socket-terminal`

B

crazyflie.resc

```
:description: This script emulates the Bitcraze
          Crazyflie 2.1 platform and runs the Crazyflie
           firmware.

mach create "CF2.1"
machine LoadPlatformDescription @platforms/boards/cf2.
          repl

# Should be written during production
sysbus.otp WriteString 0x7800 "0;CF21"


# Uncomment/comment to toggle open Analyzer windows
showAnalyzer sysbus.usart3
# showAnalyzer sysbus.usart6
# showAnalyzer sysbus.nrf

sysbus LoadELF @cf2.elf

emulation CreateUARTHub "syslink"
connector Connect sysbus.usart6 syslink
connector Connect sysbus.nrf syslink
syslink Start

machine StartGdbServer 3333
emulation CreateServerSocketTerminal 3456 "sensors"
connector Connect sysbus.i2c3.bmi_gyro sensors
# emulation CreateServerSocketTerminal 3457 "syslink"
# connector Connect sysbus.usart6 syslink
```

```
# Uncomment/comment to toggle noisy LEDs logLevel
logLevel -1 console sysbus.gpioPortD.ledB
logLevel -1 console sysbus.gpioPortC.ledRL
logLevel -1 console sysbus.gpioPortC.ledGL
logLevel -1 console sysbus.gpioPortC.ledGR
logLevel -1 console sysbus.gpioPortC.ledRR

# Uncomment/comment to toggle how much information the
        sensors log
# logLevel -1 console sysbus.i2c3.bmi_gyro
# logLevel -1 console sysbus.i2c3.bmi_accel
# logLevel -1 console sysbus.i2c3.bmp_baro

# Two initial data samples required to pass
        initialisation and test
sysbus.i2c3.bmi_gyro FeedGyroSample 10 20 30 2
```

C

crazyflie_test.resc

```
:description: Runs the firmware (compiled with uart
          debug flag set) and saves the log+uart output
           to a logfile.

logFile @logfile.log true
logLevel 3 file

include @scripts/single-node/crazyflie.resc

sysbus.usart3 AddLineHook "" "Antmicro.Renode.Logging.
          Logger.Log(LogLevel.Error, line)"
sysbus.usart3 AddLineHook "Free heap" "Antmicro.Renode.
          Emulator.Exit()"

start
```

D

Test shell script

```
set -e

timeout --foreground 120 mono output/bin/Release/Renode
          .exe --hide-analyzers --console scripts/
          single-node/crazyflie_test.resc
grep -q "Self test passed!" logfile.log
```

Bibliography

Balasubramaniam, B., H. Bagheri, S. Elbaum, and J. Bradley (2020). "Investigating controller evolution and divergence through mining and mutation*". In: *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 151–161. DOI: 10.1109/ICCPS48487.2020.00022.

Bellard, F. (2005). "Qemu, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA, p. 46.

Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood (2011). "The gem5 simulator". *SIGARCH Comput. Archit. News* **39**:2, pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718.

Bosch (2020a). *BMI088 - DataSheet*. Revision 1.6.

Bosch (2020b). *BMP388 - DataSheet*. Revision 1.7.

García, S., D. Strüber, D. Brugali, T. Berger, and P. Pelliccione (2020). "Robotics software engineering: a perspective from the service robotics domain". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Association for Computing Machinery, Virtual Event, USA, pp. 593–604. ISBN: 9781450370431. DOI: 10.1145/3368089.3409743. URL: https://doi.org/10.1145/3368089.3409743.

Herdt, V., D. Große, and R. Drechsler (2020). "Fast and accurate performance evaluation for risc-v using virtual prototypes*". In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 618–621. DOI: 10.23919/DATE48585.2020.9116522.

Leveson, N. and C. Turner (1993). "An investigation of the therac-25 accidents". *Computer* **26**:7, pp. 18–41. DOI: 10.1109/MC.1993.274940.

Microchip (2012). *24AA64/24LC64/24FC64 Data Sheet*.

Preiss, J. A., W. Hönig, G. S. Sukhatme, and N. Ayanian (2017). "Crazyswarm: a large nano-quadcopter swarm". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3299–3304. DOI: 10.1109/ICRA.2017.7989376.

Sarhadi, P. and S. Yousefpour (2014). "State of the art: hardware in the loop modeling and simulation with its applications in design, development and implementation of system and control software". *International Journal of Dynamics and Control* **3**. DOI: 10.1007/s40435-014-0108-3.

Silano, G., E. Aucone, and L. Iannelli (2018). "Crazys: a software-in-the-loop platform for the crazyflie 2.0 nano-quadcopter". In: *2018 26th Mediterranean Conference on Control and Automation (MED)*, pp. 1–6. DOI: 10.1109/MED.2018.8442759.

Silano, G. and L. Iannelli (2020). "Crazys: a software-in-the-loop simulation platform for the crazyflie 2.0 nano-quadcopter". In: Koubaa, A. (Ed.). *Robot Operating System (ROS): The Complete Reference (Volume 4)*. Springer International Publishing, Cham, pp. 81–115. ISBN: 978-3-030-20190-6. DOI: 10.1007/978-3-030-20190-6_4. URL: https://doi.org/10.1007/978-3-030-20190-6_4.

Silano, G., P. Oppido, and L. Iannelli (2019). "Software-in-the-loop simulation for improving flight control system design: a quadrotor case study". In: *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pp. 466–471. DOI: 10.1109/SMC.2019.8914154.

STMicroelectronics (2019). *STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs - Reference manual*. 18th ed.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2021

Document Number
TFRT-6135

Author(s)
Josefine Möllerström
Max Nyberg Carlsson

Supervisor
Marcus Eliasson, Bitcraze AB, Sweden
Claudio Mandrioli, Dept. of Automatic Control, Lund University, Sweden
Martina Maggio, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
Emulation of the Crazyflie 2.1 Hardware for Embedded Control System Testing

Abstract

Embedded systems are hard to debug and the field of control software testing is relatively unexplored. Progress in these areas could provide better testing heuristics and safer systems. More specifically, hardware emulation is a potentially powerful tool that can help improve the speed and quality of the development cycle. Therefore, this study developed a hardware emulator for supporting embedded testing, debugging and development. As target hardware, the Bitcraze Crazyflie 2.1 quadcopter was used. The emulation was done in the open source framework Renode. The development of the emulator is accompanied by a discussion on the uses both in the industry and research environments. In order to set up the emulation, we extended Renode by implementing different peripherals such as sensors, the EEPROM and a basic timer. These, together with the created platforms were pushed to the forked Bitcraze Renode repositories. The repositories are open source and available for future research projects to use. The emulator allows interactive use to debug and explore the virtual system without extra hardware. It can also be setup to automatically test proposed firmware changes. The usage is showcased in several test cases, where different bugs have been injected into the firmware and then found using the emulation. The main goal for the thesis was to run the same firmware as used in real Crazyflies and pass a built-in, start-up, self test. Accuracy of hardware emulators is also an open research problem. As such, the thesis provides a thorough discussion on the accuracy of the proposed tool. The discussion also includes possible future work to improve it.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-57

Recipient's notes

Security classification