# INTERN REPORT HRO - TI

**February 7, 2020**



Paul Wondel
Stud.nr.: 0947421
Internship period: 1 September 2019 - 7 Februari 2020
**First Report**

Hogeschool Rotterdam
Technische Informatica
Rotterdam University of Applied Sciences
Applied Computer Science

*Teachers*: G.W.M. van Kruining, R. van Doorn
*Intern Supervisor*: A.C. van Rossum

# Glossary

- **BLE** - Bluetooth Low Energy
- **Crownstone** - The smart device created by Crownstone B.V.
- **JSON** - JavaScript Object Notation: a extension file type
- **OS** - Operating System
- **CLI** - CommandLine Interface
- **pip** - Python Package Manager for downloading and installing packages and their dependencies for python
- **git** - distributed version-control system for tracking changes in source code during software development
- **npm** - npm is a package manager for the JavaScript programming language
- **CPU** - Computer Processing Unit
- **Crownstone code | Bluenet code** - Source code for the crownstone devices. On github the crownstone code repository is called bluenet
- **MBR** - Master Boot Record
- **EEPROM** - Electrically Erasable Programmable Read-Only Memory
- **PROM** - Programmable Read-Only Memory
- **SoftDevice** - Is the firmware required for the nRF52 chip that the crownstone uses
- **DFU** - Device Firmware Upgrade
- **IDE** - Integrated Development Environment
- **Object file** - Is a file that contain code
- **Flashing** - Uploading code to a device

# Contents

# 1 Introduction

At the Rotterdam University of Applied Sciences I study applied computer science. In the third year of my study I have done an internship at a small IT company called Crownstone B.V. At Crownstone I worked on an assignment for 6 months. During this time I learned a lot about working in embedded systems. Subjects like memory mapping, linkerscripts, binary files and sections have been surfacing frequently. I also learned a lot about the life of a technician at work at a small IT company. It has been a great experience to work at Crownstone B.V. as an intern. Working with the people at crownstone was a pleasant and wanted experience during the internship. They are willing to teach when you ask and never decline the request for help.
My assignment is been a tough assignment which I regretfully have not been able to finish. In expectation that Crownstone B.V. will pass on this assignment, allong with my work, to another intern or worker, I have written this report and have added my recommendations to the report. More information about the assignment is documented in chapter 2 and about Crownstone B.V. in chapter 3.

# 2 Assignment information

The end goal of this assignment is to make the Crownstone compatible so that Arduino code can be executed on the Crownstone device. The reason for this assignment is to attract the open-source community. At the moment the Crownstone runs its own code called bluenet. The Crownstone has many functions such as dimming lights, turning lights on and off, locating your position in the house. All these functions need to be accessible to the programmer when writing a arduino script.

## 2.1 Requirement list

This assignment has a list of requirements that need to be met before it can be considered as finished.

- DFU (update over the air) flashing/uploading of code intead of OpenOCD/JTAG
- Should be able to run basic arduino code
- Needs to be compatible with PlatformIO
- Needs to be compatible with Arduino IDE
- Automatic download of the toolchain
- Crownstone needs to be added to the Arduino IDE board manager

# 3 Crownstone B.V. background

Crownstone, founded in 2016, is a company that combines indoor localization with building automation for homes and offices. Crownstone went through the Rockstart accelerator (demo day July 2016). Crownstone has support from Almende as research company and from Almende Investments as investor. This guarantees that Crownstone can produce beyond state-of-the-art technology and has enough budget to grow organically. Crownstone's unique selling point is indoor localization. This has not been capitalized upon by any competitor in the home automation market, and only a few in the building automation market. The architecture of the company can be found in appendix B

# 4 Internship goals

During the internship the school has given some requirements that need to be met by the student. The requirements set by the school are listed in chapter 4.1.

## 4.1 Project goals:

1. The student can clearly formulate and clearly define the scope of his own internship assignment.
2. The student can translate the requirements and wishes of the customer into a possible solution.
3. The student can perform a stakeholder analysis.
4. The student can weigh up alternatives and existing solutions, taking into account the stakeholders and / or the technology.
5. The student can map and apply the Software Configuration Management.
6. The student can draw up a risk log, actively keep it up and act accordingly.
7. The student applies version management to documentation.
8. The student can apply issue tracking.
9. The student can appropriately advise the client on the results and conclusions of the analysis.
10. The student can map the system architecture.
11. The student can determine which designs are relevant to their own assignment and prepare these designs as appropriate.
12. The student can clearly demonstrate the correct functioning of the realized prototype.
13. The student has demonstrably tested the realized solution with appropriate test forms and documented this.

**Proof during internship**

| Project goal | Chapter |
|---|---|
| 1 | 2 |
| 2 | 7 |
| 3 | 4.4 |
| 4 | |
| 5 | Usage of Git |
| 6 | 4.2 |
| 7 | On top of the document pages |
| 8 | 4.3 |
| 9 | Phase progression 6.2.1.10 |
| 10 | 3 |
| 11 | C |
| 12 | |
| 13 | E |

## 4.2 Risk register

Every project comes with its own risk factors. The risk factors that influence the assignment during the project are noted in table 1.

**Table 1:** Risk Register Table

| Risk Register | | | | |
|---|---|---|---|---|
| ID | Description | Probability (1 - 5) | Impact (1 - 5) | Risk Score (1 - 25) |
| 1 | Calling in sick | 5 | 5 | 25 |
| 2 | Missing a deadline | 4 | 3 | 12 |
| 3 | Hardware breaks | 3 | 2 | 6 |
| 4 | Supervisor is unavailable | 3 | 2 | 6 |
| 5 | Laptop breaks | 1 | 5 | 5 |
| 6 | Software gets corrupted | 3 | 4 | 12 |
| 7 | Documentation files are lost | 5 | 5 | 25 |
| 8 | Research takes longer than expected | 5 | 3 | 15 |
| 9 | Missing expertise in work area | 5 | 5 | 25 |
| 10 | Wrong approach on assignment (loss of time) | 4 | 3 | 12 |
| 11 | Arduino code doesn't compile | 5 | 2 | 25 |
| 12 | Crownstone Bluenet code doesn't compile | 5 | 3 | 25 |
| 13 | J-link connection with Nordic nRF52-DK doesn't work | 5 | 2 | 25 |
| 14 | Nordic Developer Software doesn't build or execute | 5 | 3 | 25 |
| 15 | GCC compiler doesn't work | 5 | 3 | 25 |
| 16 | Debugging of code takes too long | 5 | 2 | 25 |
| 17 | The linkerscript doesn't link the c program files | 5 | 4 | 25 |

## 4.3 Issue tracking

During the assignment, issues are tracked by using the issue system on Github. Open and closed issues are saved in the repositories on Github that are involved in this project. In figures 1 and 2 the usage of issues on Github is displayed.

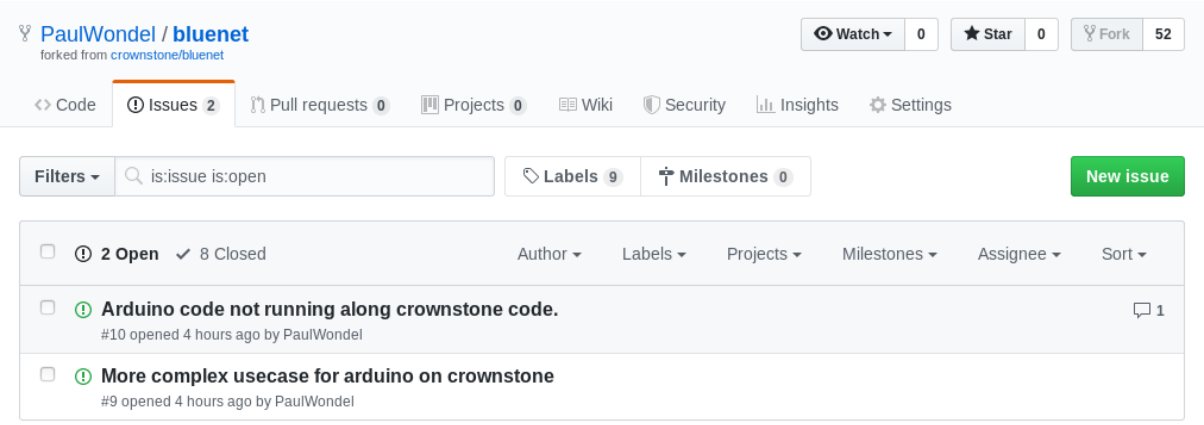



**Figure 1:** Example: Open issues on Github
(dated: 5 November 2019)

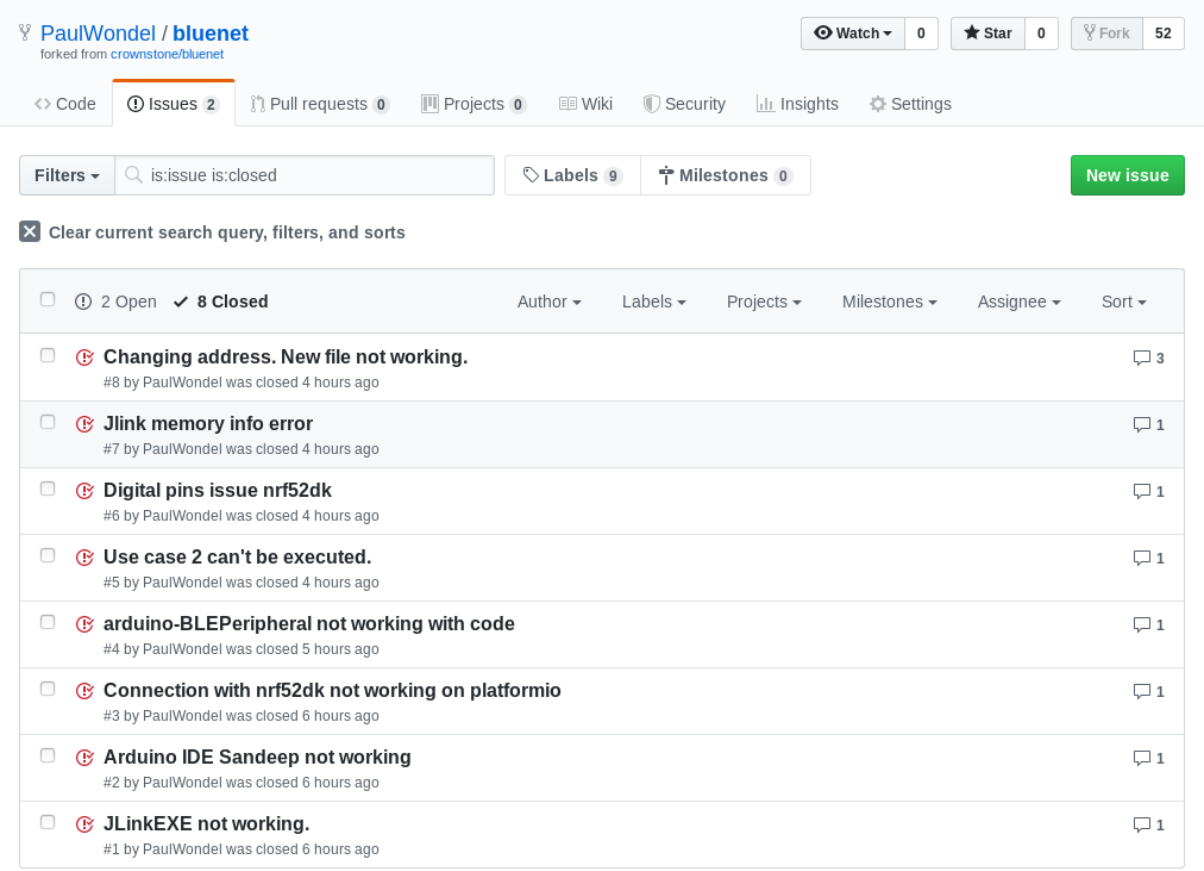



**Figure 2:** Example: Closed issues on Github
(dated: 5 November 2019)

## 4.4 Stakeholder analysis

During this assignment there are stakeholders that have certain interests and influences on the assignment. In table 2 the stakeholders are described along with their level of influence and relation.

**Table 2:** Stakeholder Analysis Table

| Stakeholder | Info | Interest | Influence On Assignment | Relative priority |
|---|---|---|---|---|
| Anne Van Rossum | CEO & Product Owner | Has set the requirements of the end product | High | High |
| Alex de Mulder | Designer & Software developer | Works on the software for that application that is used on the smartphone | Low | Low |
| Bart van Vliet | Software developer | Is the main developer on the firmware for the crownstone | High | Medium |
| Arend de Jonge | Algorithm Designer & Firmware Developer | Works on firmware, which is a heavy influence on the assignment | High | Low |
| Crownstone Customers | - | Is the target group of the assignment | Low | High |
| Arduino Community | Community that is filled with arduino code developers for various boards | After the assignment the product owner will reach out to arduino developers | Medium | High |

### Conclusion

The assignment is depended on 3 people in the company who work on the firmware of the crownstone. The wishes of the product owner are prioritized over the other parties in the company.

# 5 Research

In chapter 5 all the information that is relevant to the project has been documented. Literature and theoretical cases that have been researched during this internship are also included in chapter 5.

## 5.1 Crownstone devices

Crownstones are smart devices that can change your home or office into a smart environment. A crownstone can function as a switch, a dimmer a power monitor and a standby killer. The devices are connected to a smartphone using Bluetooth Low Energie (BLE). The crownstones use indoor positioning. When connected to an owner's smartphone they can calculate his posistion and execute their functions in a room based on the posistion of the owner's smartphone. The crownstones have machine learning capabilities. They have the ability to learn which type of rooms there are in the huis and where they are.

### 5.1.1 Crownstone Plug

The Crownstone plug is a crownstone device that can be easily inserted into an outlet.

### 5.1.2 In-built Crownstone

The in-built crownstone is a crownstone device tat needs to be installed into the electrical circuit of the house, preferably near the outlets. The in-built has the same properties as the plug crownstone.

## 5.2 Crownstone code: Bluenet

The Crownstone code named Bluenet is written by team Crownstone. The Crownstones have their own custom circuitboards on which the firmware is executed. The nRF52 chip from Nordic is the chip that is used by the Crownstone to execute the bluenet code. Bluenet is also opensource and can be accessed on the Github repository created by team Crownstone. The link to the Github repository of bluenet is noted as reference [35].

## 5.3 Arduino Integrated Development Environment

The Arduino Integrated Development Environment or Arduino Software but mostly know as the Arduino IDE, is a development environment that is used for writing and uploading code to an arduino device. The IDE containts a text editor, a message area, a text console and a toolbar with buttons that gives access to multiple menus. It also comes with its own compiler and tools to upload the code to the arduino boards. The arduino IDE also has access to different libaries that can be used in arduino codes. [24]

## 5.4 PlatformIO

PlatformIO[30] is a cross platform code builder and library manager for other platforms like Arduino or MBED support. PlatformIO has toolchains, debuggers and frameworks that work on popular plantforms like Windows, Mac en linux laten werken. There is support for more than 200 developmnent boards along with 15 development platforms and 10 frameworks. Most of the popular boards are supported.

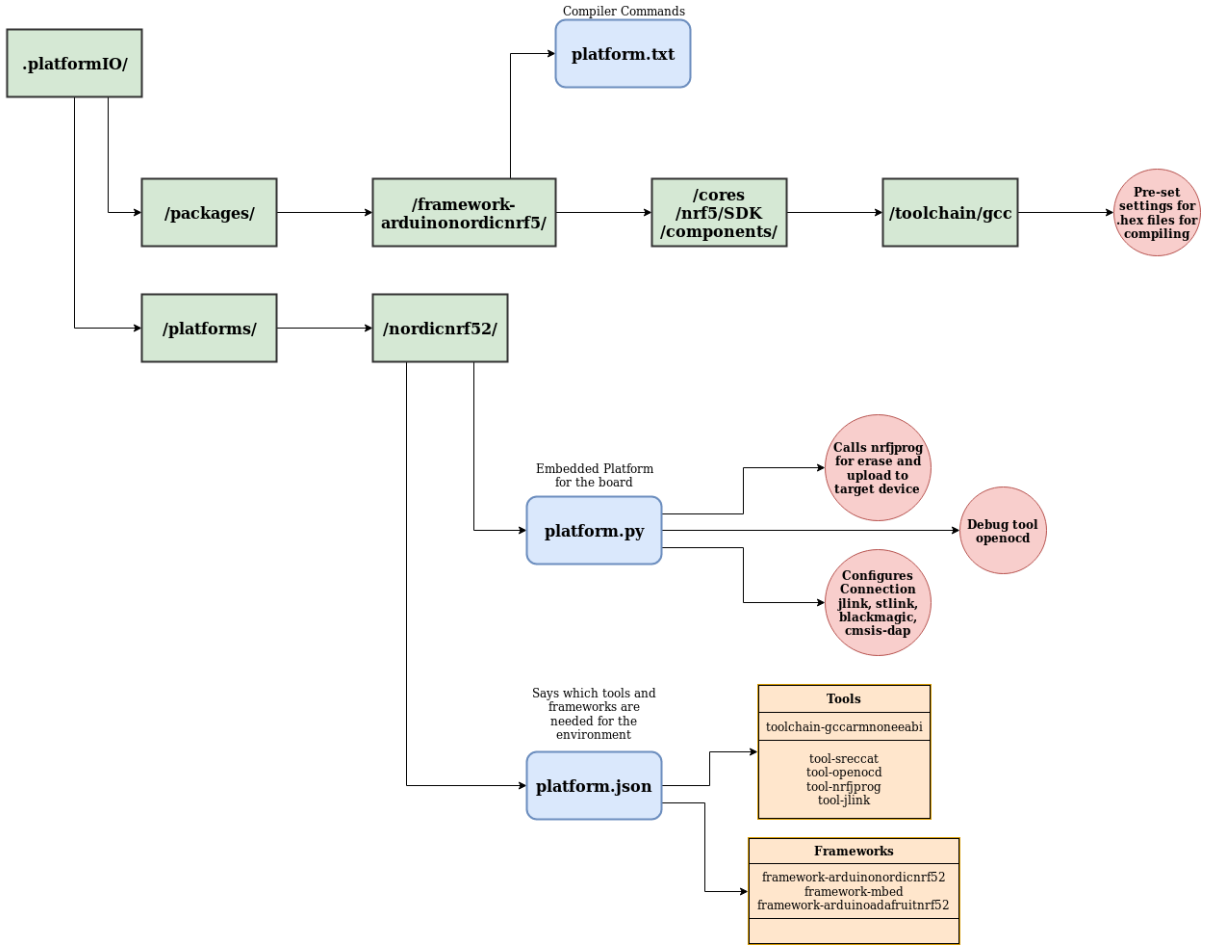



**Figure 3:** File Structure PlatformIO

PlatformIO was actually meant to be used through the command line, but with success it is able to be with other IDE's like Eclipse of Visual Studio. In figure 3 is displayed the files that are most relevant for this project in a flowchart. PlatformIO has the options and possibility to add new boards. For a new board the be added on PlatformIO, there needs to be a **JSON structure (.json)** file made for it. PlatformIO published a installation manual for adding new boards. [28]

### 5.4.1 Development platform

PlatformIO has a development platform that contains the toolchains, buildscripts and settings for each different board. A custom development platform in depended on a certain parts to be able to function, mainly packages, a manifest file (platform.json) and buildscript (main.py). [29]

### 5.4.2 Advanced scripting

PlatformIO Build System allows a programmer to extend the build process with their own custom scripts. Main and framework scripts can be found in the /builder/ folder of the platform board folder. [27]

## 5.5 Pin layout Nordic nRF52-DK

The pin layout of the Nordic nRF52-DK does not have a default layout for arduino use. Depending on the framework you pins are assigned variables and names in the header files. To find out how the pins are setup on the Nordic developer board, I had to read the datasheet and introduction guide of the Nordic nRF52-DK.

## 5.6 Tools or linux commands

During the project there were some tools or linux commands that were used frequently. They are noted in a list in chapter 5.6.

- ldd - print shared object dependencies [11]
- ld - ld combines a number of object and archive files, relocates their data and ties up symbol references [10]
- nm - Shows list of symbols of object files
- hexdump - display file contents in hexadecimal, decimal, octal, or ascii [7]
- objcopy - copy and translate object files [17] [18]
- objdump - display information from object files. [19]
- readelf: Executable and Linkable Format and it defines the structure for binaries, libraries, and core files. [4] [5]
- Checksum:
  - md5sum - compute and check MD5 message digest [15]
  - cksum - checksum and count the bytes in a file [2]
- nrfconnect - This is a tool with build gui for developing the nRF52832 developer board
- nrfjprog - This tool is used to flash hexfiles to the nRF52832
- srec_cat [22] [21]

## 5.7 Binary file

A binary file is a file stored in binary format. A binary file is computer-readable but not human-readable. All executable programs are stored in binary files, as are most numeric data files. In contrast, text files are stored in a form (usually ASCII) that is human-readable. [1]

## 5.8 HEX file

A HEX file is program file with binary information commonly used for programming microcontrollers. A compiler converts the program's code into machine code or assembly and outputs it into a HEX file. The HEX file is then read by a programmer to write the machine code into a PROM or is transferred to the target system for loading and execution. A hex file is unreadable for a person. Only machines can read it. To read the data in a hex file, the file needs to be converted into another file type. [8]

## 5.9 Random-Access Memory (RAM)

RAM (Random-Access Memory) is a form of computer data storage that is used to store working data and machine code that can be read or changed in any order. And it allows data items to be read or written almost the same amount of time irrespective of the physical location of data inside the memory. Random access memory is volatile, i.e., it requires a steady flow of electricity to maintain its contents, and as soon as the power goes off, whatever data that was in the RAM is lost. It is commonly known as read/write memory, and is an integral part in computers and other devices like printers. [20]

## 5.10 Flash memory

Flash memory is a derivative of the EEPROM memory. It is designed to make storing large amounts of data in a small space possible, allowing reading and writing in multiple memory locations with the same operation. This type of memory, is based on the use of semiconductors. In addition to being non – volatile and rewritable, it possesses almost all the features of RAM, along with the added advantage that it is non-volatile, meaning, what is stored in this type of flash memory, does not get deleted when you disconnect the device from the PC or the apparatus, unlike RAM. Flash memories are extremely important, especially in today's computer world, owing to its low power consumption, portability and size, as well as safety and efficiency; makes them ideal for supporting data and information created with digital cameras, smartphones, audio devices, among other gadgets. Even, they are quite resistant to any blow or fall, which represents a huge improvement over portable mass storage devices of previous generation. [33] [6]

## 5.11 Linker scripts

Linkerscripts are are text files with the commands to make object files compiled by a compiler into executable programs. The linker puts the input files into a single output file. The object files have a list of sections. Sections in object files have names and sizes. Most sections also have an associated block of data, known as the section contents A section may be marked as loadable, which means that the contents should be loaded into memory when the output file is run. Sections with no contents may be allocatable, which means that an area in the memory should be set aside. Nothing in particular should be loaded there. In some cases this memory area should be zeroed out. A section which is neither loadable nor allocatable typically contains some sort of debugging information. [32] In figure 19 is an example of a memory map displayed.

## 5.12 Sections in linkerscripts

The SECTIONS command tells the linker how to map input sections into output sections, and how to place the output sections in memory. This can also make the linker script easier to understand because you can use those commands at meaningful points in the layout of the output file. If a SECTIONS command is not used in the linker script, the linker will place each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file. The first section will be at address zero. [31]

### 5.12.1 Addresses in sections

Every loadable or allocatable output section has two addresses. The first is the VMA (virtual memory address) and the second is the LMA (load memory address). VMA is the address the section will have when the output file is run. LMA is the address at which the section will be loaded.

## 5.13 Interrupt Handler

Interrupts are signals that go to the processor to indicate that there is an event that needs the processor's immediate attention. When an interrupt occurs the processor completes the execution of the current instruction and starts the execution of an Interrupt Service Routine (ISR) or Interrupt Handler. ISR tells the processor what to do when the interrupt occurs. The interrupts can be either hardware interrupts or software interrupts. Interrupt handlers vary based on what triggered the interrupt and the speed at which the interrupt handler completes its task.

When an interrupt occurs, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine, ISR. [34] Interrupts also have a priority system on which they operate. More over interrupt priority can be found in references [9] and [3]

## 5.14 SoC support in SoftDevice Handler

When making handlers for the firmware, the handlers need to be registered so that the softdevice knows what they are and where they are located. Registering the handler goes through using the SoftDevice Handler.

```
#define NRF_SDH_SOC_EVENT_OBSERVERS
        (
            _name,
            _prio,
            _handler,
            _context,
            _cnt
        )
```

**Listing 1:** Macro Registering Event Handler

Macro for registering an array of `nrf_sdh_soc_evt_observer_t`. Modules that want to be notified about SoC events must register the handler using this macro. Each observer's handler will be dispatched

an event with its relative context from _context. This macro places the observer in a section named "sdh_soc_observers". View listing 1.

```
#define NRF_SDH_SOC_OBSERVER
        (
            _name,
            _prio,
            _handler,
            _context
        )
```

**Listing 2:** Macro Registering Handler

Macro for registering `nrf_sdh_soc_evt_observer_t`. Modules that want to be notified about SoC events must register the handler using this macro.
This macro places the observer in a section named "sdh_soc_observers". View listing 2. The complete explanation can be viewed in reference [26].

## 5.15 Function pointers in C/C++

Function pointers are variables that are used to store a address of a function to later be called through that function pointer. It is useful so that not all functions need to be loaded in the main file. [23]

# 6 Phase progression

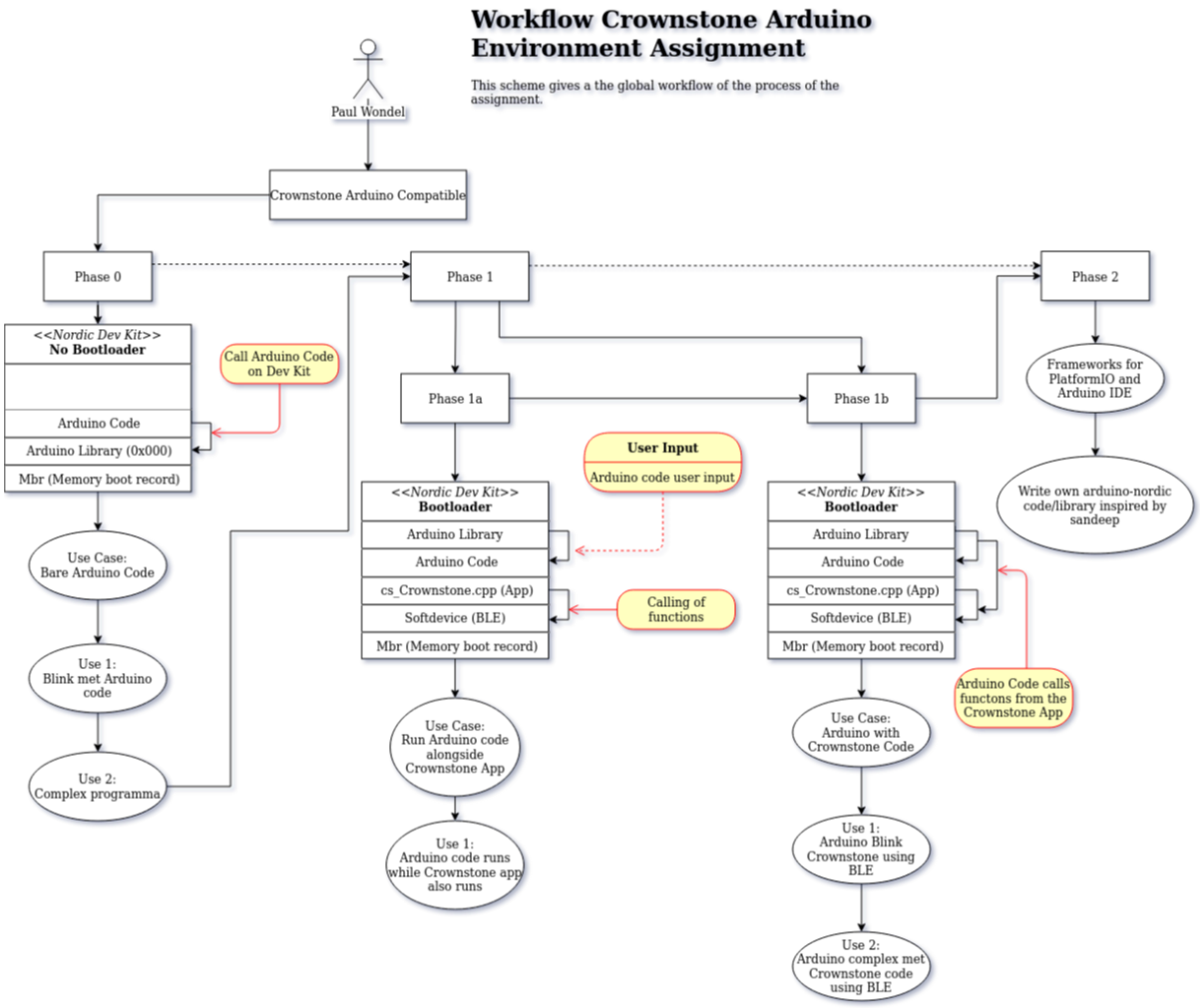



**Figure 4:** Phase Plan Workflow Scheme

## 6.1 Phase 0

The goal of phase 0 is to get arduino code to run on the Nordic nRF52-DK Developer kit without a bootloader and without the crownstone app. Windows 10 is not a recommended OS to be used for this phase. The Microsoft OS does not use the required drivers automatically, which makes it so that the developer kit cannot flash code tot he nRF52-DK. The best option is to use Ubuntu 18.4.1. due to the software and driver support it has.

On the Linux system it is essential to have the right package for the J-Link tool installed. Not all Linux distributions can work with the package. The recommended Linux OS for the package is a Debian based OS such as Ubuntu & Linux Mint. Arch does not have the right architecture for the package and the package is not available in the repository of Arch Linux. The official website of SEGGER has the

best packages available which are downloadable.

PlatformIO has to be installed on the operating system. This is required for programming the Nordic nRF52-DK with arduino code. It can be PlatformIO Core (CLI) or PlatformIO IDE. During this research I have made use of PlatformIO IDE on VSCode. In PlatformIO the right package is needed for the Nordic nRF52-DK board. PlatformIO already has the support for nRF52 Nordic boards. The platform files still need to be installed which can be done at the "platforms" tab in the PlatformIO IDE. After the required packages are installed, a new project needs to be created with the nRF52 as board and Arduino as framework.

Programming arduino code is the same as programming code for an Arduino Uno. The difference is that the pin layout is not the same as with an arduino. In figures 5 and 6 are the pins on the Nordic nRF52 Development Kit displayed along with the assigned arduino signal.

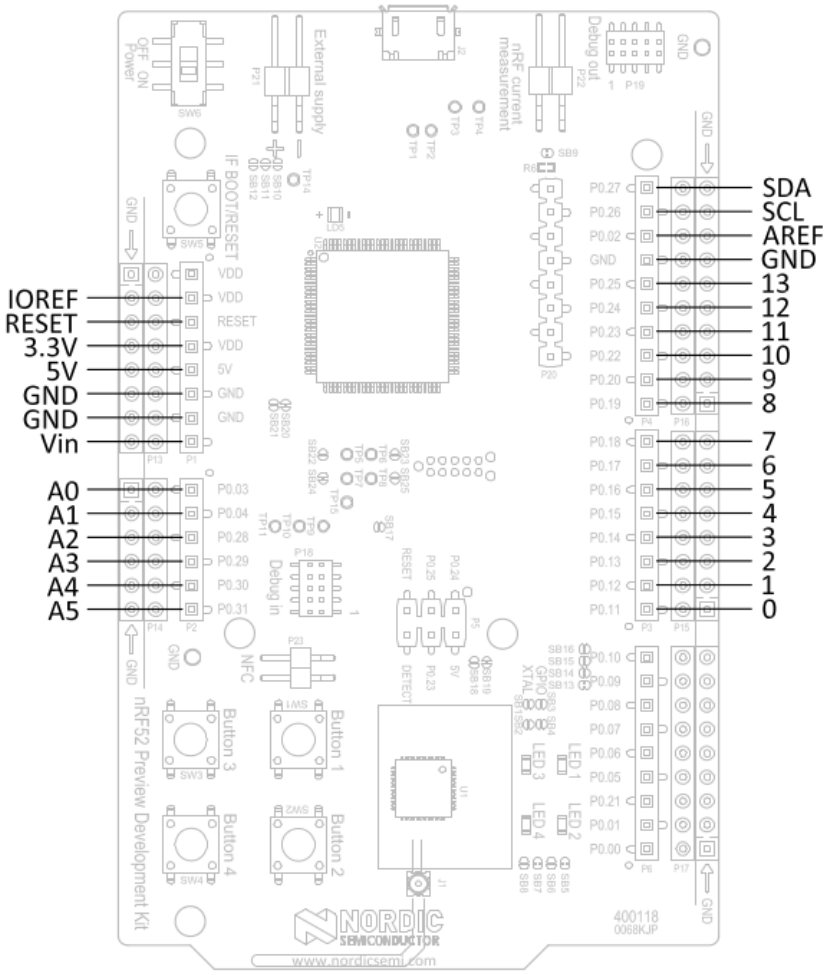



**Figure 5:** nRF52-DK pin layout
Reference [25]

The analog pins are called upon by using them as {A0, A1, A2, A3, A4, A5} and the digital pins as {(0),(1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12),(13)}. Some of the digital pins are already assigned to some buttons and LEDs that are on the developer board.

| GPIO | Part | Arduino signal |
|---|---|---|
| P0.13 | Button 1 | 2 |
| P0.14 | Button 2 | 3 |
| P0.15 | Button 3 | 4 |
| P0.16 | Button 4 | 5 |
| **GPIO** | **Part** | **Arduino signal** |
| P0.17 | LED 1 | 6 |
| P0.18 | LED 2 | 7 |
| P0.19 | LED 3 | 8 |
| P0.20 | LED 4 | 9 |

**Figure 6:** Assigned pins onboard LEDs & buttons nRF52-DK
Reference [25]

In listing 3 is an example of how the pin definition is written in arduino code.

```
#define ledPin (7) //dgital pin 7 on the board (P0.18)
#define irPin PIN_A1 //Analog pin A0 on the board (P0.05)
#define temPin PIN_AREF //AREF pin on the board (P0.02)
```

**Listing 3:** Definition PINS in arduino code

#### Use case: bare arduino code

In this use case the goal is to have bare arduino code run on the Nordic nRF52-DK. This is use case is devided into 2 use cases:

**Use case 1: arduino blink** A simple arduino script to light up the onboard LEDs by using the onboard buttons.

**Use case 2: auto intensity control of power LED** To test the control of the analog pins as well as the digital pins with pwm using sensors for input data.

The testplan for this phase can be found in the appendix E.

### 6.1.1 Results use case: bare arduino code

The results of this use case test have been documented in a test report. To view the results refer to appendix E.

## 6.2 Phase 1

In phase 1 the goal is to execute the arduino code with arduino library simultaneously with bluenet. To achieve the goal in a more secure fashion, this phase has been strategically divided into phase 1a and phase 1b. In phase 1a the goal is to execute the arduino code along side bluenet both independently. Bluenet should be executed and should be able to call functions from the softdevice (BLE) regardless of the arduino code. And the arduino code should be able to call funtions of bluenet so that they can be used in the arduino code.

### 6.2.1 Phase 1a

Installing bluenet on a computer is required to continue in this phase. The recommended system to install bluenet is on Ubuntu 18.4.1 because of the support there is for this OS. The installation guide for bluenet can be found on the github repository (see reference [35]).

**6.2.1.1 Flashing** The bluenet code is put into a binary file that is a hexfile which is flashed to the nRF52 using the `nrfjprog` tool. In the bluenet repository there is a build directory in which you will find a file called CMakeList.txt. In this list you will find the commands that are used to flash the bootloader settings, the crownstone code itself and how to reset the nRF52. The commands are listed in 6.2.1.1

```
$ nrfjprog -f nrf52 --eraseall
$ nrfjprog -f nrf52 --program softdevice_mainpart.hex --sectorerase
$ nrfjprog -f nrf52 --program crownstone.hex --sectorerase
$ nrfjprog -f nrf52 --program bootloader_settings.hex --sectorerase
$ nrfjprog --reset
```

**6.2.1.2 Memory Map** Nordic Semiconductor (the company that makes the Nordic nRF52-DK) has tools available for different uses. The tool for displaying a memory map of nRF52-DK is named nrfconnectprogrammer. To access this tool the nrfconnect_core program has to be installed first. The installation guide can be found on the git repository of bluenet (see references [35] and 22).
After installing the nrfconnect_core the `nrfconnect_programmer` will need to be run. To start the tool, run the command `make nrfconnect_core` in the bluenet directory. The `nrfconnect_core` gui tool is displayed in figure 20. With the `nrfconnect_programmer` tool the memory map of the current device can be displayed in realtime. It also displays the addresses the files are loaded unto. The programmer tool gui is displayed in figure 21.

After displaying the memory map the addresses in the memory had to be found. The programmer tool has the option to show the start address of the hex files. The hex files of the crownstone firmware were placed along with the bootloader settings in the upload section of the tool to display the sizes of the files with their start addresses. This way their addresses in the memory are displayed. The relevant binary files are listed below:

- crownstone.hex
- bootloadersettings.hex

- softdevice_mainpart.hex

The files are available in the /bluenet/build/default/ directory.

For the certainty of having the right start addresses of the files, the files are seperately inspected in commandline. In the .elf files the start address of the program is located. To read a .elf file, you use the command `readelf [option] [inputfile.elf]`. To get the addresses the command `readelf -Sh crownstone.elf` is used. Applying this to each of the files has resulted in displaying an estimation of their sizes in the commandline. See figures 23 and 24.

#### 6.2.1.3 Arduino binary file by PlatformIO

When compiling arduino code in platformio, the .hex files are stored in a directory. This directory is hidden and only used for the .hex file to be stored in and to be flashed from. Because the .hex and .elf files are needed of the arduino code, they were located in /home/$USER/Documents/PlatformIO/Projects/PROJECT_NAME/.pio/build/nrf52dk/. In this directory the files firmware.hex and firmware.elf are used, which contains the arduino.cpp compiled into assembly code for the chip to be read.

#### 6.2.1.4 Flashing the arduino binary file

PlatformIO uses the same tools that bluenet uses to compile, build & flash code to the nRF52-DK. The tools are listed below:

- `nrfjprog` - For writing files to nrf52
- `jlink` - For connecting to nrf52
- `nrfutil` - A Python package that includes the nrfutil utility and the nordicsemi library
- `gcc-arm-none-eabi` - For compiling and building the .elf and .hex files

After reading the arduino binary file with the command `readelf -e firmware.elf` it is shown that the start address of the firmware.hex is 0x00000000. Before flashing the arduino binary file, there needs to be checked if the file will be in conflict with the addresses of the other binary files from bluenet. This comes in conflict with the softdevice of the crownstone that already has that address reserved for itself. This is shown in figure 7.
A solution was to change the address of the `firmware.hex` to a different address. To edit a hex file its .elf file needs to be edited and then converted to .hex. To write to an .elf file the arm-none-eabi-objcopy tool can be used. The address 0x00026000 is the address on which crownstone.hex is registered to. The purpose was to change the firmware.hex address from 0x00000000 to 0x00026000 so that the arduino code can be tested to run independently. By running the command below the address can be changed of a binary:

```
arm-none-eabi-objcopy --change-addresses 0x00026000 -I elf32-little firmware.elf -O
elf32-little test.elf
```

The command above able to change all the addresses of the sections within the firmware.elf and called the output file test.elf. After changing the address using a test.elf binary file it was converted into a .hex file. by running the command: `arm-none-eabi-objcopy -O ihex test.elf test.hex`. This created the output file test.hex. Next was flashing the test.hex along with the softdevice_mainpart.hex and bootloader_settings.hex by using the following commands:

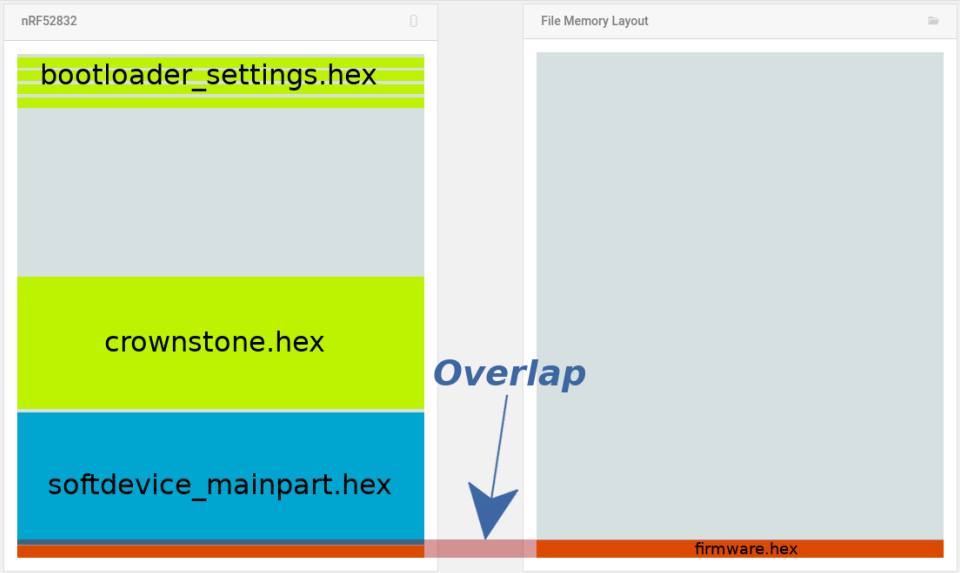



**Figure 7:** Bluenet binary along side the arduino binary

```
$ nrfjprog -f nrf52 --eraseall
$ nrfjprog -f nrf52 --program softdevice_mainpart.hex --sectorerase
$ nrfjprog -f nrf52 --program test.hex --sectorerase
$ nrfjprog -f nrf52 --program bootloader-settings.hex --sectorerase
$ nrfjprog --reset
```

After the successfull flashing of files, the code did not execute. The changing the address of a binary file for the memory map is not effective. This has only changes the addresses but it doesn't link the other functions within the file to the addresses of the sections. During compiling a binary file a linkerscript links all the required and included files, so the change of the address has to happen before the a binary file is compiled.

#### 6.2.1.5 Re-compiling the arduino binary file in PlatformIO

Instead of changing the address of the already compiled hex file, the explicit command or script that assigns the address in the compiling process of PlatformIO needs to be found, so that the start address can be modified. The command for compiling code in platformio is `platformio run`. To find out what platformio executes during this command, the command has to be run with the verbose option enabled so that the output will be displayed in teh commandline. This also displayes some of the processes that are run in the background. View this in figure 25.

In the output that `platformio run` gave, the command `arm-none-eabi-gc++` is shown to be called upon. In the command .platformio/packages/framework-arduinonordicnrf5/cores/nRF5/SDK/components/toolchain is set as input in this command for compiling. The linkerscripts are located in the gcc folder and the script with the start address is named nrf52_xxaa.ld (see figure 3 for reference). The contents of the file are displayed in listing 4

```
/* Linker script to configure memory regions. */

SEARCH_DIR(.)
GROUP(-lgcc -lc -lnosys)

MEMORY
{
  FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x80000
  RAM (rwx) :  ORIGIN = 0x20000000, LENGTH = 0x10000
}


INCLUDE "nrf52_common.ld"
```

**Listing 4:** Arduino linkerscript from PlatformIO

The FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x80000 is the start address for where the firmware.hex is going to be placed in the memory of the nRF52-DK. The reason why platformio puts the address at 0x000000000 is because that is where the CPU starts reading the memory. The address is changed to 0x00026000 and then uploaded the hex file along with the softdevice_mainpart.hex and bootloader_settings.hex. The address is changed because the softdevice is already assigned to 0x000000000. Bluenet is assigned to 0x00026000 and is executed after the softdevice. The purpose is to have the arduino binary on the exact same address. After uploading the files and resetting the device, the arduino binary was successfully executed by the cpu. This was proved by following the test plan in appendix E. The arduino binary is the same code that is made in chapter 6.1.

#### 6.2.1.6 Running Arduino code alongside Crownstone (both independently)

When both bluenet and the arduino code are flashed to the nRF52-DK the cpu only executes bluenet. The arduino code and bluenet function as two different applications. There is no way for one application to just call a function or an object in another application. There is also no function in the crownstone to refer cpu to the arduino code to execute it. There needs to be a reference in the crownstone code to the address of the arduino code. For that we need the use of a handler to refer the cpu to the address of the arduino code. More info about handlers can be found in chapter 5.13. How the cpu currently executes the binary files is visualized in figure 8.

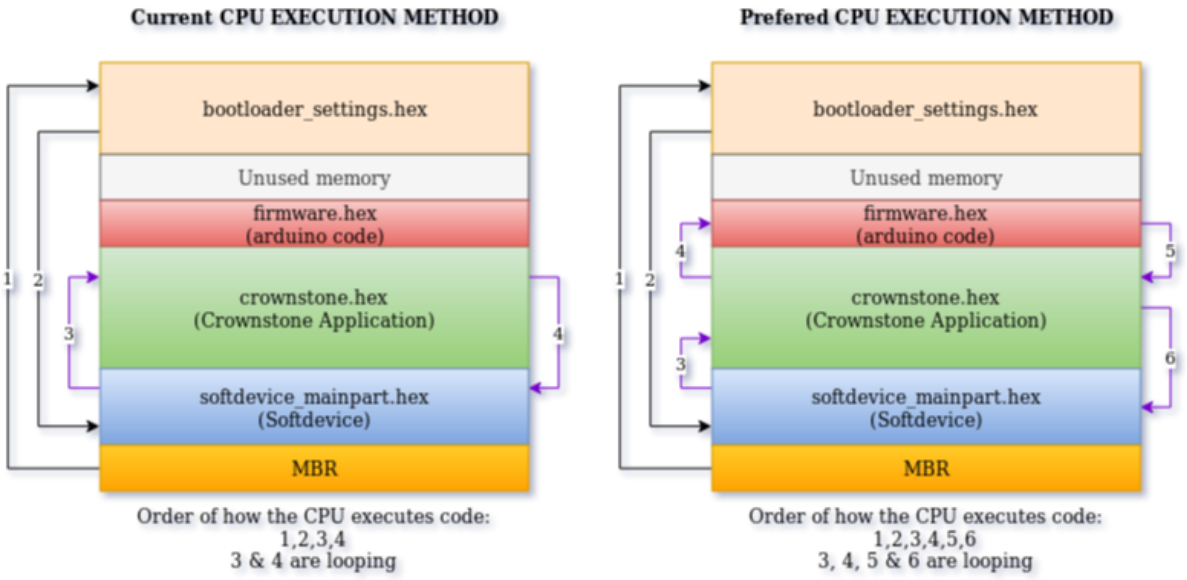



**Figure 8:** CPU Execution of code in the Memory Map Scheme

**6.2.1.7 Handler & linkerscript** A handler for the arduino code needs to be created in bluenet that goes calls the arduino code. All the handlers that the crownstone uses are in bluenet/source/src/common/cs_Handlers.cpp file. In the cs_Handler.cpp file one of the handlers is used as an example to create a

```
71  void crownstone_soc_evt_handler(uint32_t evt_id, void * p_context) {
72          LOGInterruptLevel("soc evt int=%u", BLEutil::getInterruptLevel());
73
74      switch(evt_id) {
75          case NRF_EVT_FLASH_OPERATION_SUCCESS:
76          case NRF_EVT_FLASH_OPERATION_ERROR:
77          case NRF_EVT_POWER_FAILURE_WARNING: {
78  //              uint32_t gpregret_id = 0;
79  //              uint32_t gpregret_msk = GPREGRET_BROWNOUT_RESET;
80  //              // NOTE: do not clear the gpregret register, this way
81  //              //   we can count the number of brownouts in the bootloader.
82  //              sd_power_gpregret_set(gpregret_id, gpregret_msk);
83  //              // Soft reset, because brownout can't be distinguished from hard reset otherwise.
84  //              sd_nvic_SystemReset();
85
86  #if NRF_SDH_DISPATCH_MODEL == NRF_SDH_DISPATCH_MODEL_INTERRUPT
87                  uint32_t retVal = app_sched_event_put(&evt_id, sizeof(evt_id), crownstone_soc_evt_handler_decoupled);
88                  APP_ERROR_CHECK(retVal);
89  #else
90                  crownstone_soc_evt_handler_decoupled(&evt_id, sizeof(evt_id));
91  #endif
92              break;
93          }
94          case NRF_EVT_RADIO_BLOCKED: {
95                  break;
96          }
97          default: {
98              LOGd("Unhandled event: %i", evt_id);
99          }
100     }
101 }
102 NRF_SDH_SOC_OBSERVER(m_crownstone_soc_observer, CROWNSTONE_SOC_OBSERVER_PRIO, crownstone_soc_evt_handler, NULL);
```

**Figure 9:** Crownstone event handler in cs_Handler.cpp

handler for the arduino code. On line 71 in the cs_Handlers.cpp the `void crownstone_soc_evt_handler` has 2 inputs, `uint32_t evt_id & void * p_context`. The `uint_t evt_id` value is input into a switch case method which returns a response if the operation was a success or failure. See figure 9 for reference. On line 182 there is a `NRF_SDH_SOC_OBSERVER` function that is called upon. In chapter 5.14 there is more info about the SoC observer handler.

In the bluenet/source/include/third/nrf/ directory of the bluenet repository is a linkerscript named `generic_gcc_nrf52.ld`. This file containts the information to be compiled with for the `crownstone.elf` and `crownstone.hex`. In the linkerscript the start address for the memory register is specified. Sections are also defined in the linkerscript. Part of the linkerscript is listed in listing 5.

```
SECTIONS
{
  .mem_section_dummy_rom :
  {
  }
  .sdh_ant_observers :
  {
    PROVIDE(__start_sdh_ant_observers = .);
    KEEP(*(SORT(.sdh_ant_observers*)))
    PROVIDE(__stop_sdh_ant_observers = .);
  } > FLASH
  .sdh_soc_observers :
  {
    PROVIDE(__start_sdh_soc_observers = .);
    KEEP(*(SORT(.sdh_soc_observers*)))
    PROVIDE(__stop_sdh_soc_observers = .);
  } > FLASH
  .sdh_ble_observers :
  {
    PROVIDE(__start_sdh_ble_observers = .);
    KEEP(*(SORT(.sdh_ble_observers*)))
    PROVIDE(__stop_sdh_ble_observers = .);
  } > FLASH
  .sdh_state_observers :
  {
    PROVIDE(__start_sdh_state_observers = .);
    KEEP(*(SORT(.sdh_state_observers*)))
    PROVIDE(__stop_sdh_state_observers = .);
  } > FLASH
  .sdh_stack_observers :
  {
    PROVIDE(__start_sdh_stack_observers = .);
    KEEP(*(SORT(.sdh_stack_observers*)))
    PROVIDE(__stop_sdh_stack_observers = .);
  } > FLASH
  .sdh_req_observers :
  {
    PROVIDE(__start_sdh_req_observers = .);
    KEEP(*(SORT(.sdh_req_observers*)))
    PROVIDE(__stop_sdh_req_observers = .);
  } > FLASH
```

**Listing 5:** Crownstone linkerscript generic_gcc_nrf52.ld

First there needs to be a section for the function of the event handler. The sections makes sure that there is enough memory saved for the functions.

```
    .arduino_handler : \\ Name of the section
    {
     PROVIDE(__start_arduino_handler = .); \\ Creates the start for the section
     KEEP(*(SORT(.arduino_handler*)))   \\ Every variable inserted into this section
    recieves this as its type
     PROVIDE(__stop_arduino_handler = .); \\ Creates the end for the section
    } > FLASH \\ Memory location where the section needs to be loaded on
```

After creating a new section for the arduino handler called `.arduino_handler`, it was necessary to know if it compiled compiled successfully. For compiling bluenet code for this "test & prototype phase", there has been made use of a custom build instead of using the default bluenet build. In figure 26 the configuration for creating a custom build is displayed.
The newly custom build is named **arduino**. After following the instructions the bluenet code was compiled with the edited version of the linkerscript. The compiled files of bluenet code of the build are stored in the bin directory of bluenet. The default build uses the bluenet/bin/default where the `bootloader.elf`, `crownstone.bin` `crownstone.elf` and `crownstone.hex` are located. The same goes for the custom **arduino** build; bluenet/bin/arduino.

```
[~/gitfiles/bluenet/build]$ readelf -Sh ../bin/arduino/./crownstone.elf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x35d11
  Start of program headers:          52 (bytes into file)
  Start of section headers:          7972876 (bytes into file)
  Flags:                             0x5000400, Version5 EABI, hard-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         28
  Section header string table index: 27

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00026000 006000 024dec 00  AX  0   0 16
  [ 2] .sdh_soc_observer PROGBITS        0004adec 02adec 000018 00   A  0   0  4
  [ 3] .sdh_ble_observer PROGBITS        0004ae04 02ae04 000008 00   A  0   0  4
  [ 4] .sdh_state_observ PROGBITS        0004ae0c 02ae0c 000010 00   A  0   0  4
  [ 5] .sdh_stack_observ PROGBITS        0004ae1c 02ae1c 000010 00   A  0   0  4
  [ 6] .sdh_req_observer PROGBITS        0004ae2c 02ae2c 000010 00   A  0   0  4
  [ 7] .nrf_mesh_flash   PROGBITS        0004ae3c 02ae3c 0000d4 00  WA  0   0  4
  [ 8] .ARM.exidx        ARM_EXIDX       0004af10 02af10 000008 00  AL  1   0  4
  [ 9] .data             PROGBITS        20002380 032380 0000bc 00  WA  0   0  4
  [10] .fs_data          PROGBITS        2000243c 03243c 000014 00  WA  0   0  4
  [11] .bss              NOBITS          20002450 032450 0042b0 00  WA  0   0  8
  [12] .heap             PROGBITS        20006700 032450 002000 00      0   0  8
  [13] .stack_dummy      PROGBITS        20006700 034450 002000 00      0   0  8
  [14] .ARM.attributes   ARM_ATTRIBUTES  00000000 036450 000030 00      0   0  1
  [15] .comment          PROGBITS        00000000 036480 000076 01  MS  0   0  1
  [16] .debug_info       PROGBITS        00000000 0364f6 405755 00      0   0  1
  [17] .debug_abbrev     PROGBITS        00000000 43bc4b 044da8 00      0   0  1
  [18] .debug_loc        PROGBITS        00000000 4809f3 070fe3 00      0   0  1
  [19] .debug_aranges    PROGBITS        00000000 4f19d8 005f68 00      0   0  8
  [20] .debug_ranges     PROGBITS        00000000 4f7940 0127b0 00      0   0  1
  [21] .debug_macro      PROGBITS        00000000 50a0f0 05e5a4 00      0   0  1
  [22] .debug_line       PROGBITS        00000000 568694 0e01a9 00      0   0  1
  [23] .debug_str        PROGBITS        00000000 64883d 11214e 01  MS  0   0  1
  [24] .debug_frame      PROGBITS        00000000 75a98c 011f40 00      0   0  4
  [25] .symtab           SYMTAB          00000000 76c8cc 018a30 10     26 4436  4
  [26] .strtab           STRTAB          00000000 7852fc 0153bd 00      0   0  1
  [27] .shstrtab         STRTAB          00000000 79a6b9 000152 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
```

**Figure 10:** Section headers in `crownstone.elf`

The command `readelf -Sh ../bin/arduino/./crownstone.elf` is used to see the section headers. The section `.arduino_handler` is not displayed after compiling. This means that just editing the linkerscript `generic_gcc_nrf52.ld` is not enough. See figure 10 for the result.

Running the command `DEBUG=1 make` in the build folder to see what the output was while compiling

and recieved a error output in the listing below.

```
   /home/$USER/gitfiles/bluenet/tools/gcc_arm_none_eabi/bin/../lib/gcc/arm-none-eabi
   /8.3.1/../../../../arm-none-eabi/bin/ld: crownstone section `.arduino_handler' will
   not fit in region `FLASH'
/home/$USER/gitfiles/bluenet/tools/gcc_arm_none_eabi/bin/../lib/gcc/arm-none-eabi
   /8.3.1/../../../../arm-none-eabi/bin/ld: region FLASH overflowed with .data and user
    data
/home/$USER/gitfiles/bluenet/tools/gcc_arm_none_eabi/bin/../lib/gcc/arm-none-eabi
   /8.3.1/../../../../arm-none-eabi/bin/ld: region `FLASH' overflowed by 8202724 bytes
/home/$USER/gitfiles/bluenet/tools/gcc_arm_none_eabi/bin/../lib/gcc/arm-none-eabi
   /8.3.1/../../../../arm-none-eabi/bin/ld: .arduino_handler has both ordered [`.ARM.
   exidx' in /home/$USER/gitfiles/bluenet/tools/gcc_arm_none_eabi/bin/../lib/gcc/arm-
   none-eabi/8.3.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/crt0.o] and
   unordered [`.ARM.extab' in /home/$USER/gitfiles/bluenet/tools/gcc_arm_none_eabi/bin
   /../lib/gcc/arm-none-eabi/8.3.1/../../../../arm-none-eabi/lib/thumb/v7e-m+fp/hard/
   crt0.o] sections
/home/$USER/gitfiles/bluenet/tools/gcc_arm_none_eabi/bin/../lib/gcc/arm-none-eabi
   /8.3.1/../../../../arm-none-eabi/bin/ld: final link failed: bad value
```

The error explains that the `.arduino_handler` section doesn't fit in the FLASH. Because the size of `.arduino_handler` section was not defined, it had to be defined to be compiled. The information in the `.arduino_handler` section was changed by adding the lines to the code:

```
1   .arduino_handler :
2   {
3     __arduino_handler_size = 0x04;
4     __arduino_handler_start = .;
5     . = . + __arduino_handler_size;
6     __arduino_handler_end = .;
7   } > FLASH
```

After running the command `DEBUG=1 make` without recieving errors, it had to be checked again to see if the new section was created and added by the compiler. In the output of the `readelf -Sh ../bin/arduino/./crownstone.elf` the `.arduino_handler` section is dispayed. With this as proof it can be confirmed as a success that the new section is created. Figure 11 shows the proof of the section created.

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00026000 006000 035038 00  AX  0   0 16
  [ 2] .sdh_soc_observer PROGBITS        0005b038 03b038 000018 00   A  0   0  4
  [ 3] .sdh_ble_observer PROGBITS        0005b050 03b050 000008 00   A  0   0  4
  [ 4] .sdh_state_observ PROGBITS        0005b058 03b058 000010 00   A  0   0  4
  [ 5] .sdh_stack_observ PROGBITS        0005b068 03b068 000010 00   A  0   0  4
  [ 6] .sdh_req_observer PROGBITS        0005b078 03b078 000010 00   A  0   0  4
  [ 7] .nrf_mesh_flash   PROGBITS        0005b088 03b088 0000d4 00  WA  0   0  4
  [ 8] .arduino_handler  NOBITS          0005b15c 03b15c 000010 00  WA  0   0  1
  [ 9] .ARM.exidx        ARM_EXIDX       0005b16c 03b16c 000008 00  AL  1   0  4
  [10] .data             PROGBITS        20002380 042380 0000c0 00  WA  0   0  4
  [11] .fs_data          PROGBITS        20002440 042440 000014 00  WA  0   0  4
  [12] .bss              NOBITS          20002458 042458 004b90 00  WA  0   0  8
  [13] .heap             PROGBITS        20006fe8 042458 002000 00      0   0  8
  [14] .stack_dummy      PROGBITS        20006fe8 044458 002000 00      0   0  8
  [15] .ARM.attributes   ARM_ATTRIBUTES  00000000 046458 000030 00      0   0  1
  [16] .comment          PROGBITS        00000000 046488 000076 01  MS  0   0  1
  [17] .debug_info       PROGBITS        00000000 0464fe 42d639 00      0   0  1
  [18] .debug_abbrev     PROGBITS        00000000 473b37 046d12 00      0   0  1
  [19] .debug_loc        PROGBITS        00000000 4ba849 07a667 00      0   0  1
  [20] .debug_aranges    PROGBITS        00000000 534eb0 0061d0 00      0   0  8
  [21] .debug_ranges     PROGBITS        00000000 53b080 012bb0 00      0   0  1
  [22] .debug_macro      PROGBITS        00000000 54dc30 061614 00      0   0  1
  [23] .debug_line       PROGBITS        00000000 5af244 0eb286 00      0   0  1
  [24] .debug_str        PROGBITS        00000000 69a4ca 117575 01  MS  0   0  1
  [25] .debug_frame      PROGBITS        00000000 7b1a40 012b70 00      0   0  4
  [26] .symtab           SYMTAB          00000000 7c45b0 01a3e0 10     27 4794  4
  [27] .strtab           STRTAB          00000000 7de990 015f82 00      0   0  1
  [28] .shstrtab         STRTAB          00000000 7f4912 000163 00      0   0  1
```

**Figure 11:** `.arduino_handler` in section headers in `crownstone.elf`

The next task was to to print the address of the `.arduino_handler` and the contents of the `.arduino_handler`. To do that address of the `__arduino_handler_start` and `__arduino_handler_end` need to be known and printed first. Since the `__arduino_handler_size = 0x10` which is 16bits, there are 16 address value slots that are reserved for the `.arduino_handler section` to assign data to.
To list the sections of `crownstone.elf` run the command `readelf -Sh [file]` and to list the symbols with their addresses run the command `nm [file] | grep [symbol]`.

#### 6.2.1.8 arduino_handler section

The idea is to have a section in the memory where the jumper function is stored. This section needs to be used by both the arduino binary and bluenet binary. First is to print the addresses of the arduino_handler section. when the command nm -a crownstone.elf | grep arduino was executed it gave the addresses as its output. This is noted below.

```
0005b15c b .arduino_handler
0005b160 B __arduino_handler_end
0005b160 B __arduino_handler_size
0005b15c B __arduino_handler_start
```

With the use of the ld tool, the compiler can link the relevant libraries to the involved object files. Next was to create a section in the binary file by using the linkerscript. The code is put in a file called main.c and the linker script is named arduino_handler.ld. All the files for this part of the phase can be found in reference [14]. In listing 6 the contents of the linker script are displayed.

```
1  OUTPUT_FORMAT(elf32-i386)
2  ENTRY(main)
3  OUTPUT(Test)
4
5  SECTIONS
6  {
7      .text : { *(.text) }
8      . = ALIGN(0x10);
9      .data : { *(.data) }
10     . = ALIGN(0x10);
11     .bss : { *(.bss)}
12     . = ALIGN(0X10);
13     .arduino_handler :
14     {
15     __arduino_handler_size = 0x10;
16     __arduino_handler_start = .;
17     . = . + __arduino_handler_size;
18     __arduino_handler_end = .;
19     }
20 }
21 }
```

**Listing 6:** "arduino_handler.ld"

There has also been made use of a makefile which can be viewed in listing 32. After compiling and linking the files was successful, an output was recieved running the readelf command. See listing 7.

```
ELF Header:
```

```
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2 complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          2097564 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         8
  Section header string table index: 7

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 200000 000016 00  AX  0   0  1
  [ 2] .eh_frame         PROGBITS        00000018 200018 000038 00   A  0   0  8
  [ 3] .arduino_handler  NOBITS          00000050 200050 000010 00  WA  0   0  1
  [ 4] .comment          PROGBITS        00000000 200050 00002c 01  MS  0   0  1
  [ 5] .symtab           SYMTAB          00000000 20007c 000090 10      6   5  4
  [ 6] .strtab           STRTAB          00000000 20010c 00004b 00      0   0  1
  [ 7] .shstrtab         STRTAB          00000000 200157 000045 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

**Listing 7:** Output readelf

In the output it is displayed that .arduino_handler is indeed created which proved that creating a new section was a success.

#### 6.2.1.9 Call a function from a different binary

Editing the linkerscript is just one way to create a new section. A new section can also be created in the source code itself without touching the linkerscript. In listing 8 is listed how the `.arduino_handler` section by only using the source code.

```
1  #include<stdio.h>
2  #include<iostream>
3
4  #include<cs_ArduinoHandler.h>
5
6  // Initialize the variable in the arduino_handler section with address,
```

```
7     static void (*jumpToCallback)() __attribute__((section("arduino_handler"))); //
      Variable in the section that functions as a pointer
8     extern unsigned char __start_arduino_handler;
9     extern unsigned char __stop_arduino_handler;
10
11    // Pointers to start and stop
12    static unsigned char * p_arduino_handler_start = &__start_arduino_handler;
13    static unsigned char * p_arduino_handler_end = &__stop_arduino_handler;
14
15    static void (*func)(); //variable to store function address into
16
17    // function to be put in section
18    void callback(void){
19        printf("Go to arduino firmware\n"); // will be replaced with a addresss or
      pointer to address
20    }
21    // Can't call it main() since cs_Crownstone.cpp already has the main. This has to be
       called by the main.
22    int jumpMain(){
23        printf("section 'arduino_handler' starts at %p, ends at %p, and the 1st byte is
      %c\n", p_arduino_handler_start, p_arduino_handler_end, p_arduino_handler_start[0]);
24
25        func = &callback; // assign address to pointer
26        (*func)(); // call function indirectly
27        jumpToCallback = func;
28        (*jumpToCallback)();
29
30        if(jumpToCallback == func){
31            printf("Jump To Callback Address: %p\n", jumpToCallback);
32        }
33        else{
34            printf("jumpToCallback: %p | func: %p\n",jumpToCallback,func);
35        }
36
37        return 0;
38    }
```

**Listing 8:** cs_ArduinoHandler.cpp

```
1     void callback(void);
2
3     int jumpMain();
```

**Listing 9:** cs_ArduinoHandler.h

There is no need to edit the linker-script to create a new section if it can be done in the source code. In the source code a pointer is initialized in the section by using the variable:

```
1     __attribute__((section("arduino_handler")));
```

The variable created in the section by adding the line:

```
1     static unsigned char pling __attribute__((section("arduino_handler"))) = '!';
```

The size of the section can be set by changing the ! symbol for a number that represents the byte size. After creating the variable in the in the `.arduino_handler` section a function pointer had to be declared.

The function pointer is called `jumpToCallback` and it points to the address of the function `callback()`.

Creating a section in the source code will work, but there are no guaranties that the compiler will keep the empty section. So to have the garuantee that the section is created, the section is created in the linkerscript and then the source code puts a variable in the created section. When creating the section by either with the linkerscript or source code it has to be checked that they both refer to the exact same section called. Bluenet creates the sections using both source code and the linkerscript.

cs_ArduinoHandler.cpp is placed in bluenet/source/src/ and cs_ArduinoHandler.h in bluenet/source/src/ directory. The command 'DEBUG=1 make' is used to compile and recieve the output while compiling. After running 'readelf -Sh arduino/crownstone.elf' it was displayed that the `.arduino_handler` section was created. When the nm command was run, the function symbol 'jumpToCallback' was not found. From the verbose output during compiline, it was noted that there was no building process taking place for cs_ArduinoHandler.cpp To fix that the file bluenet/source/conf/cmake/crownstone.src.cmake was edited and the cs_ArduinoHandler.cpp was added to the list of included files to compile. See listing 10.

```
1    # The arduino handler specified to be compiled
2    LIST(APPEND FOLDER_SOURCE "${SOURCE_DIR}/cs_ArduinoHandler.cpp")
```

**Listing 10:** File added to list for compiling

After compiling, the functions were checked to see if they were inserted into the `.arduino_handler` section. The symbols func, jumpToCallback and callback were not found. This means that the compiler did not create the section from source code or deleted the section because it was empty.

What needed to happen was that the variables and functions needed to be registered into the section. By making use of a macro function that automatically registers the function to the next address in the `.arduino_handler` section. In the configuration file of the bluenet build files has been added an option for the arduino macro option. For reference see listing 14.
The macro function `REGISTER_ARDUINO_HANDLER` (see listing 11) registers functions making them a struct. When the function from the section needs to be called, all that needs to be called is the right struct (see listing 13).
The newly created files are cs_Arduino.c and cs_Arduino.h. These files can be found on the github repository bluenet-arduino [13] along with the files that are displayed in the listings 11, 12, 13 & 14.

```
1    #pragma once
2
3    #ifdef __cplusplus
4    extern "C" {
5    #endif
6
7    #include <drivers/cs_Serial.h>
8
9    typedef void (*arduino_func_t)(const char);
10
11   struct arduino_handler {
12       arduino_func_t f;
13       char *name;
```

```
};

#define REGISTER_ARDUINO_HANDLER(func)                          \
    static struct arduino_handler __arduino_handler__ ## func  \
    __attribute((__section__(".arduino_handlers")))             \
    __attribute((__used__)) = {                                 \
        .f = func,                                              \
        .name = #func,                                          \
    }

extern struct arduino_handler __start_arduino_handlers;
extern struct arduino_handler __stop_arduino_handlers;

#ifdef __cplusplus
}
#endif
```

**Listing 11:** cs_Arduino.c

```
#include <arduino/cs_Arduino.h>

#include <drivers/cs_Serial.h>

// Arduino command.
static void arduinoCommand(const char value) {
    LOGd("Arduino command %i", value);
}

REGISTER_ARDUINO_HANDLER(arduinoCommand);
```

**Listing 12:** cs_Arduino.h

```
#include <arduino/cs_Arduino.h>
..
..
Crownstone::Crownstone(boards_config_t& board) :
..
..
#ifdef ARDUINO
    //arduino_init_all();
    struct arduino_handler *iter = &__start_arduino_handlers;
    for (; iter < &__stop_arduino_handlers; ++iter) {
        LOGd("Call handler %s", iter->name);
        iter->f(8);
    }
#endif
..
..
void Crownstone::init(uint16_t step)
```

**Listing 13:** cs_Crownstone.cpp

```
  IF(ARDUINO AND "${ARDUINO}" STRGREATER "0")
LIST(APPEND FOLDER_SOURCE "${SOURCE_DIR}/arduino/cs_Arduino.c")
```

```
    ENDIF()
    ..
    ..
    ADD_DEFINITIONS("-DARDUINO=${ARDUINO}")
```

**Listing 14:** crownstone.src.cmake

In the config of the build `ARDUINO=1` has to be added to the file to enable the use of the arduino handler and section. The config file is located in the config/ directory.

Before implementing actual arduino code into bluenet, an example from Nordic is used. The goal is to have the bluenet binary run along with the example binary, which is called blinky in ble_central. The blinky binary also needs to have the `.arduino_handlers` section created with functions registered in it. The source code can be found on the github repository [12]. The files to help create the .arduino_handlers section are `extra.c` and `extra.h`. To have the binary files work with the exact same section, `.arduino_handlers` section needs to be assigned to the same address in both binary files (see listing 15 & 16).

```
    MEMORY
    {
      FLASH (rx) : ORIGIN = 0x63000, LENGTH = 0x9e000
      RAM (rwx) :  ORIGIN = 0x20001db8, LENGTH = 0xe248
      /* Specific memory space for the arduino section */
      ARDUINO (rx) : ORIGIN = 0x60000, LENGTH = 0x61000
    }
```

**Listing 15:** Lines added in blinky linker files

```
    MEMORY
    {
      FLASH (rx) : ORIGIN = APPLICATION_START_ADDRESS, LENGTH = APPLICATION_LENGTH
      RAM (rwx) :  ORIGIN = RAM_R1_BASE, LENGTH = RAM_APPLICATION_AMOUNT
      CORE_BL_RAM (rw) :     ORIGIN = 0x2000fd00, LENGTH = 0x300
      UICR_BOOTADDR (r) :    ORIGIN = 0x10001014, LENGTH = 0x04
      UICR_MBRPARAMADDR (r): ORIGIN = 0x10001018, LENGTH = 0x04
      /* Specific memory space for the arduino section */
      ARDUINO (rx) : ORIGIN = 0x60000, LENGTH = 0x61000
    }
```

**Listing 16:** Lines added in bluenet linker files

After compiling the section address displayed that they were on the same address. This is display in the listings below.

```
                                    (Address)
[10] .arduino_handlers PROGBITS        00060000 010000 000008 00  WA  0   0  4
```

**Listing 17:** in blinky binary

```
                                    (Address)
[ 8] .arduino_handlers PROGBITS        00060000 010000 000008 00  WA  0   0  4
```

**Listing 18:** In crownstone binary

Calling the function from the blinky binary registered by the bluenet binary is the wrong approach to take. This is because blinky is not being executed by the cpu and what the bluenet binary only registers the function in the `.arduino_handlers` section. Instead the blinky binary needs to register the function in the `.arduino_handlers` section and have bluenet call the function. For debugging it is adviced to enable the serial monitor.

Even with both binaries using the same section, the bluenet binary cannot use the data that has been inserted by the blinky binary. So when the binaries are compiled, they link whatever is in the sections registered by their own code. Whenever the crownstone binary is compiled it doesn't register any function into the arduino section, and so the crownstone binary links that as a empty section. However blinky binary does put data in the exact same section but only links that to its own code. Whenever the function from the bluenet binary registered by blinky binary is called, the bluenet binary cannot execute the function by simply putting `arduinoCommand()` in the code. The bluenet binary does not detect the function registered by the blinky binary. By using an iteration that prints the contents of the addresses of in the arduino section, it is possible to test with the bluenet binary to see if the data on the address is accessible.

Both binaries have the same `__start_arduino_handlers` but the `__stop_arduino_handlers` are not the same. The blinky binary has a different `__stop_arduino_handlers` than from the bleunet binary. To iterate the section the data that is on the addresses after `__start_arduino_handlers` needed to be printed.

The code in listing 19 prints the data that is between `__start_arduino_handlers` and `__stop_arduino_handlers`. Since the bluenet binary has both `__start_arduino_handlers` and `__stop_arduino_handlers` on the the same address `00060000` it doesn't read the `arduinoCommand()` that is placed in the section that is on the address after `00060000`.

```
1    void arduino(){
2        LOGi("Calling handler struct");
3        struct arduino_handler *iter = &__start_arduino_handlers;
4        for (; iter < &__stop_arduino_handlers; ++iter) {
5                LOGd("Call handler %s", iter->name);
6                iter->f(8);
7        }
8    }
```

**Listing 19:** Calling handler iteration function

The result of the print executing the print function is displayed in figure 12.

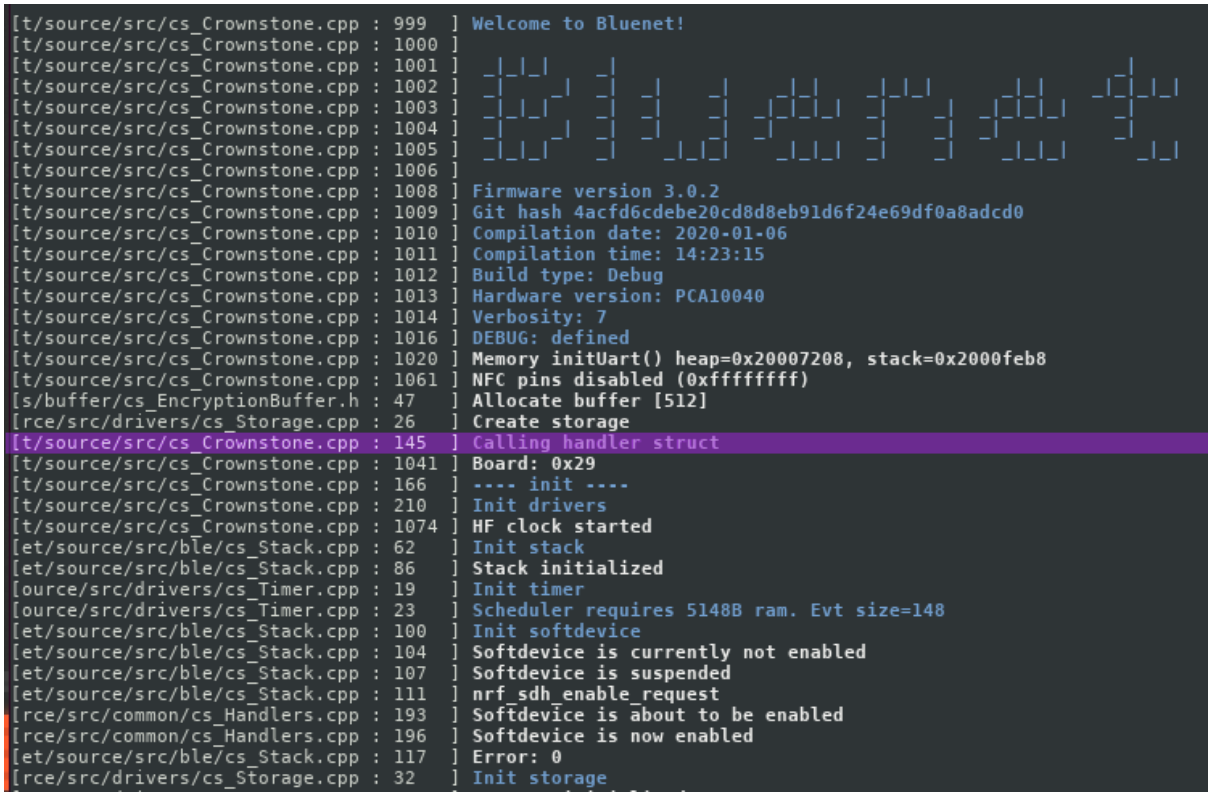



**Figure 12:** Result with Calling handler iteration

The method in the iteration is changed from stopping at `__start_arduino_handlers` to read the next 8 address after `__start_arduino_handlers`. This causes the system to keep looping where the `crownstone.run()` command is never executed. The result is display in figure 13.

```
void arduino(){
    LOGi("Calling handler struct");
    struct arduino_handler *iter = &__start_arduino_handlers;
    for(; iter < &__start_arduino_handlers+8; ++iter) {
        LOGd("Call handler %s", iter->name);
        iter->f(8);
    }
}
```

**Listing 20:** Calling handler iteration function

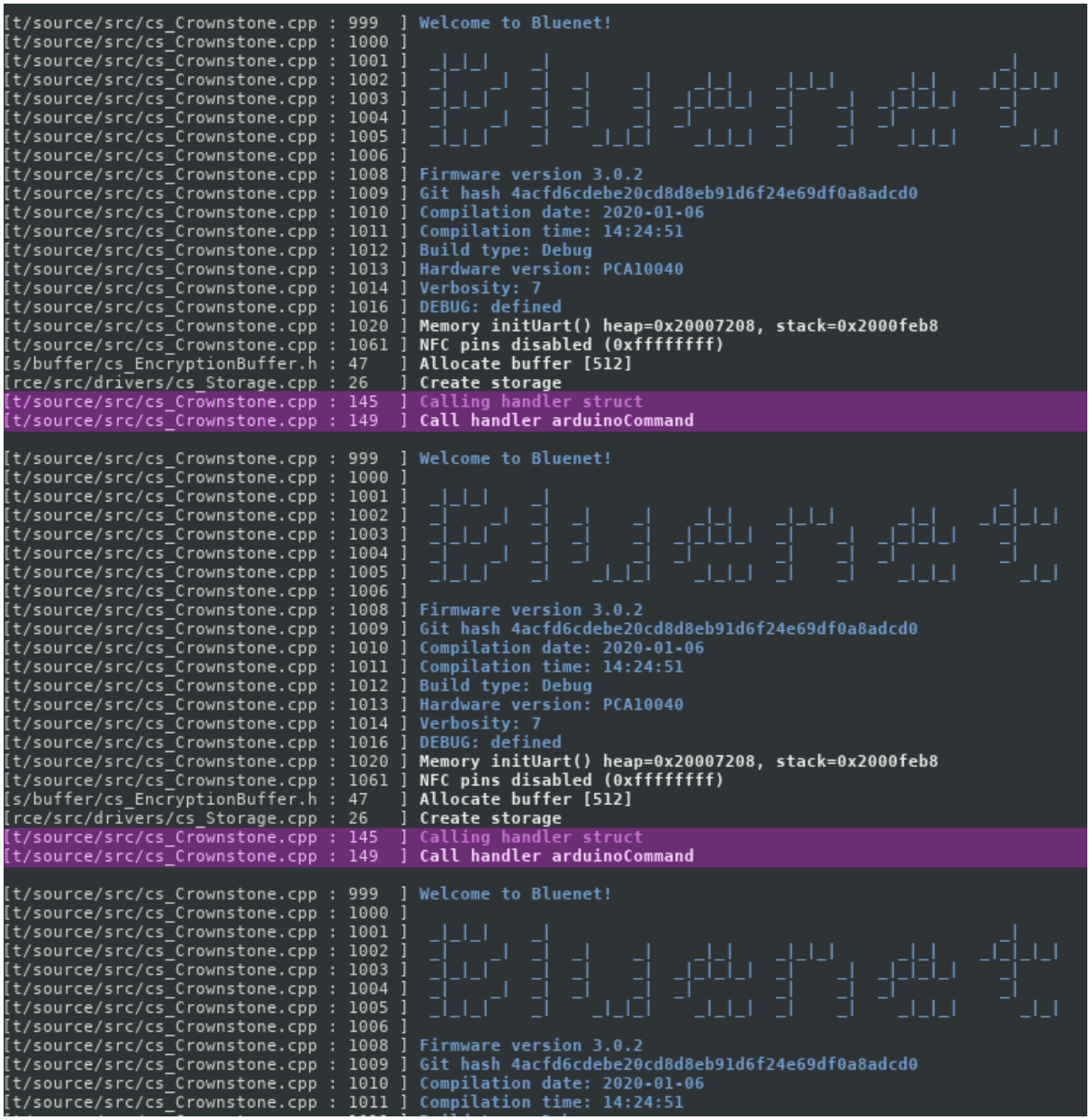



**Figure 13:** Result with Calling handler iteration: FAILED

To fix the error the amount of addresses that the cpu needs to read past the `__start_arduino_handlers` had to be reduced. It has been tested from 8 to 1. From 8 to 2 gave the same loop conflict and at 1 it went successfully. It is concluded that if the cpu reads an address that is empty, it starts the main loop from the beginning again instead of going to the next command. The code displayed in listing 21 has successfully called `arduinoCommand()`. The result of the code's execution is dispayed figure 14.

```
1 void arduino(){
2     LOGi("Calling handler struct"); //Print serial line
3     // Initialize iter pointer to address __start_arduino_handlers
4     struct arduino_handler *iter = &__start_arduino_handlers;
```

```
    for(; iter < &__start_arduino_handlers+1; ++iter) {
        LOGd("Call handler %s", iter->name); // prints name of function
        iter->f(8); // Calls function and puts value 8 in function
    }
}
```

**Listing 21:** Calling handler iteration function

```
[t/source/src/cs_Crownstone.cpp : 999  ] Welcome to Bluenet!
[t/source/src/cs_Crownstone.cpp : 1000 ]
[t/source/src/cs_Crownstone.cpp : 1001 ]  _|_|_|    _|                                                  _|
[t/source/src/cs_Crownstone.cpp : 1002 ]  _|    _|  _|  _|    _|    _|_|    _|_|_|      _|_|    _|_|_|_|
[t/source/src/cs_Crownstone.cpp : 1003 ]  _|_|_|    _|  _|    _|  _|_|_|_|  _|    _|  _|_|_|_|    _|
[t/source/src/cs_Crownstone.cpp : 1004 ]  _|    _|  _|  _|    _|  _|        _|    _|  _|          _|
[t/source/src/cs_Crownstone.cpp : 1005 ]  _|_|_|    _|    _|_|_|    _|_|_|  _|    _|    _|_|_|      _|_|
[t/source/src/cs_Crownstone.cpp : 1006 ]
[t/source/src/cs_Crownstone.cpp : 1008 ] Firmware version 3.0.2
[t/source/src/cs_Crownstone.cpp : 1009 ] Git hash 4acfd6cdebe20cd8d8eb91d6f24e69df0a8adcd0
[t/source/src/cs_Crownstone.cpp : 1010 ] Compilation date: 2020-01-06
[t/source/src/cs_Crownstone.cpp : 1011 ] Compilation time: 13:59:02
[t/source/src/cs_Crownstone.cpp : 1012 ] Build type: Debug
[t/source/src/cs_Crownstone.cpp : 1013 ] Hardware version: PCA10040
[t/source/src/cs_Crownstone.cpp : 1014 ] Verbosity: 7
[t/source/src/cs_Crownstone.cpp : 1016 ] DEBUG: defined
[t/source/src/cs_Crownstone.cpp : 1020 ] Memory initUart() heap=0x20007208, stack=0x2000feb8
[t/source/src/cs_Crownstone.cpp : 1061 ] NFC pins disabled (0xffffffff)
[s/buffer/cs_EncryptionBuffer.h : 47   ] Allocate buffer [512]
[rce/src/drivers/cs_Storage.cpp : 26   ] Create storage
[t/source/src/cs_Crownstone.cpp : 145  ] Calling handler struct
[t/source/src/cs_Crownstone.cpp : 149  ] Call handler arduinoCommand
[t/source/src/cs_Crownstone.cpp : 1041 ] Board: 0x29
[t/source/src/cs_Crownstone.cpp : 166  ] ---- init ----
[t/source/src/cs_Crownstone.cpp : 210  ] Init drivers
[t/source/src/cs_Crownstone.cpp : 1074 ] HF clock started
[et/source/src/ble/cs_Stack.cpp : 62   ] Init stack
[et/source/src/ble/cs_Stack.cpp : 86   ] Stack initialized
[ource/src/drivers/cs_Timer.cpp : 19   ] Init timer
[ource/src/drivers/cs_Timer.cpp : 23   ] Scheduler requires 5148B ram. Evt size=148
[et/source/src/ble/cs_Stack.cpp : 100  ] Init softdevice
[et/source/src/ble/cs_Stack.cpp : 104  ] Softdevice is currently not enabled
[et/source/src/ble/cs_Stack.cpp : 107  ] Softdevice is suspended
[et/source/src/ble/cs_Stack.cpp : 111  ] nrf_sdh_enable_request
[rce/src/common/cs_Handlers.cpp : 193  ] Softdevice is about to be enabled
[rce/src/common/cs_Handlers.cpp : 196  ] Softdevice is now enabled
[et/source/src/ble/cs_Stack.cpp : 117  ] Error: 0
[rce/src/drivers/cs_Storage.cpp : 32   ] Init storage
```

**Figure 14:** Result with Calling handler iteration: SUCCESS

Now that bluenet binary can print the name of the function that is registered by the blinky binary, the function can be called and executed in by the bluenet binary itself. A simple function that returns a value has been written to test the function. It will print the value which can then be seen in the serial monitor. The code is displayed in listings 22, 23, 24.

```
#include <extra.h>
#include <nrf_log.h>

static int giveValue(int value){
return value;
}

REGISTER_ARDUINO_HANDLER(giveValue);
```

**Listing 22:** blinky source code extra.c

```
#pragma once
```

```
2
3     #ifdef __cplusplus
4     extern "C" {
5     #endif
6
7     typedef int (*arduino_func_t)(int);
8
9     struct arduino_handler {
10        arduino_func_t f;
11        char *name;
12    };
13
14    #define REGISTER_ARDUINO_HANDLER(func)                          \
15        static struct arduino_handler __arduino_handler__ ## func  \
16        __attribute((__section__(".arduino_handlers")))             \
17        __attribute((__used__)) = {                                 \
18            .f = func,                                              \
19            .name = #func,                                          \
20        }
21
22    extern struct arduino_handler __start_arduino_handlers;
23    extern struct arduino_handler __stop_arduino_handlers;
24
25    #ifdef __cplusplus
26    }
27    #endif
```

**Listing 23:** blinky source code extra.h

```
1     ..
2     void arduino(){
3         LOGi("Calling handler struct");
4         struct arduino_handler *iter = &__start_arduino_handlers;
5         for (; iter < &__stop_arduino_handlers+1; ++iter) {
6             LOGd("Call handler %s", iter->name);
7             LOGd("Execution handler result %x", iter->f(8));
8             iter->f(8);
9         }
10    }
11    ..
```

**Listing 24:** Code lines added to cs_Crownstone.cpp

In source/include/arduino/cs_Arduino.h the line 16 needs to be changed from

```
typedef void (*arduino_func_t)(const char);
```

to

```
typedef int (*arduino_func_t)(int);
```

Now there is a successfull metod to call a function from one binary in the other. In figure 15 is the result of the executed function.

```
[t/source/src/cs_Crownstone.cpp : 1002 ]
[t/source/src/cs_Crownstone.cpp : 1003 ]  _|_|_|    _|                                                  _|
[t/source/src/cs_Crownstone.cpp : 1004 ]  _|     _|  _|  _|    _|    _|_|    _|_|_|      _|_|    _|_|_|_|
[t/source/src/cs_Crownstone.cpp : 1005 ]  _|_|_|    _|  _|    _|  _|_|_|_|  _|    _|  _|_|_|_|    _|
[t/source/src/cs_Crownstone.cpp : 1006 ]  _|     _|  _|  _|    _|  _|        _|    _|  _|          _|
[t/source/src/cs_Crownstone.cpp : 1007 ]  _|_|_|    _|    _|_|_|    _|_|_|  _|    _|    _|_|_|      _|_|
[t/source/src/cs_Crownstone.cpp : 1008 ]
[t/source/src/cs_Crownstone.cpp : 1010 ] Firmware version 3.0.2
[t/source/src/cs_Crownstone.cpp : 1011 ] Git hash 4ae4509aea7def0d6cf2815e97469ecc04df6f7c
[t/source/src/cs_Crownstone.cpp : 1012 ] Compilation date: 2020-01-08
[t/source/src/cs_Crownstone.cpp : 1013 ] Compilation time: 11:52:29
[t/source/src/cs_Crownstone.cpp : 1014 ] Build type: Debug
[t/source/src/cs_Crownstone.cpp : 1015 ] Hardware version: PCA10040
[t/source/src/cs_Crownstone.cpp : 1016 ] Verbosity: 7
[t/source/src/cs_Crownstone.cpp : 1018 ] DEBUG: defined
[t/source/src/cs_Crownstone.cpp : 1022 ] Memory initUart() heap=0x20007208, stack=0x2000feb8
[t/source/src/cs_Crownstone.cpp : 1063 ] NFC pins disabled (0xffffffff)
[s/buffer/cs_EncryptionBuffer.h : 47   ] Allocate buffer [512]
[rce/src/drivers/cs_Storage.cpp : 26   ] Create storage
[t/source/src/cs_Crownstone.cpp : 145  ] Calling handler struct
[t/source/src/cs_Crownstone.cpp : 148  ] Call handler giveValue
[t/source/src/cs_Crownstone.cpp : 151  ] Execution handler result 8
[t/source/src/cs_Crownstone.cpp : 1043 ] Board: 0x29
[t/source/src/cs_Crownstone.cpp : 168  ] ---- init ----
[t/source/src/cs_Crownstone.cpp : 212  ] Init drivers
[t/source/src/cs_Crownstone.cpp : 1076 ] HF clock started
[et/source/src/ble/cs_Stack.cpp : 62   ] Init stack
[et/source/src/ble/cs_Stack.cpp : 86   ] Stack initialized
[ource/src/drivers/cs_Timer.cpp : 19   ] Init timer
[ource/src/drivers/cs_Timer.cpp : 23   ] Scheduler requires 5148B ram. Evt size=148
[et/source/src/ble/cs_Stack.cpp : 100  ] Init softdevice
[et/source/src/ble/cs_Stack.cpp : 104  ] Softdevice is currently not enabled
[et/source/src/ble/cs_Stack.cpp : 107  ] Softdevice is suspended
[et/source/src/ble/cs_Stack.cpp : 111  ] nrf_sdh_enable_request
[rce/src/common/cs_Handlers.cpp : 193  ] Softdevice is about to be enabled
[rce/src/common/cs_Handlers.cpp : 196  ] Softdevice is now enabled
[et/source/src/ble/cs_Stack.cpp : 117  ] Error: 0
[rce/src/drivers/cs_Storage.cpp : 32   ] Init storage
```

**Figure 15:** Result calling handler struct test

It is recommended to use a switch case method to be able to call multiple functions in one function. In listing 25 is an example on how the function is coded. In listing 26 is an example on how the function is used.

```
1  static int giveValue(int value){
2    return value;
3  }
4
5  static int sumValue(int value){
6    value=value+value;
7    return value;
8  }
9
10 static int arduinoSelect(int option, int value){
11   switch (option){
```

```
12      case 1: return giveValue(value); break;
13      case 2: return sumValue(value); break;
14      default: return 0;
15    }
16  }
17
18  REGISTER_ARDUINO_HANDLER(arduinoSelect);
```

**Listing 25:** Switch case method in extra.c

```
1  void arduino(){
2      LOGi("Calling handler struct");
3      struct arduino_handler *iter = &__start_arduino_handlers;
4      for (; iter < &__stop_arduino_handlers+1; ++iter) {
5          LOGd("Call handler %s", iter->name);
6          LOGd("Execution handler case 1 %i", iter->f(1,5));
7          LOGd("Execution handler case 2 %i", iter->f(2,7));
8      }
9  }
```

**Listing 26:** Calling function using switch case in cs_Crownstone.cpp

#### 6.2.1.10 Executing both binaries simultaneously

Two different binaries cannot be executed at the same time. The bluenet binary is being executed and while that code is running, the loop function of the blinky binary has to be called. In the blinky binary there currently is only the `main()` function. What needs to be created is a `init()` function and a `loop()` function. This is shown in listings 27, 28 and 29. How the new functions are used is shown in listing 30.

```
1      void init(void){
2
3          // Initialize.
4          log_init();
5          timer_init();
6          leds_init();
7          buttons_init();
8          power_management_init();
9          ble_stack_init();
10         scan_init();
11         gatt_init();
12         db_discovery_init();
13         lbs_c_init();
14
15
16     }
17
18     void loop(void){
19         // Start execution.
20         NRF_LOG_INFO("Blinky CENTRAL example started.");
21         //arduino();
22         scan_start();
23
24         // Turn on the LED to signal scanning.
```

```
25          bsp_board_led_on(CENTRAL_SCANNING_LED);
26
27          // Enter main loop.
28          for (;;)
29          {
30              idle_state_handle();
31          }
32      }
33
34      int main(void)
35      {
36
37          init();
38          loop();
39
40      }
```

**Listing 27:** Blinky main.c file

```
1       #include <stdint.h>
2       #include <stdio.h>
3       #include <string.h>
4       #include "nrf_sdh.h"
5       #include "nrf_sdh_ble.h"
6       #include "nrf_sdh_soc.h"
7       #include "nrf_pwr_mgmt.h"
8       #include "app_timer.h"
9       #include "boards.h"
10      #include "bsp.h"
11      #include "bsp_btn_ble.h"
12      #include "ble.h"
13      #include "ble_hci.h"
14      #include "ble_advertising.h"
15      #include "ble_conn_params.h"
16      #include "ble_db_discovery.h"
17      #include "ble_lbs_c.h"
18      #include "nrf_ble_gatt.h"
19      #include "nrf_ble_scan.h"
20      #include "nrf_log.h"
21      #include "nrf_log_ctrl.h"
22      #include "nrf_log_default_backends.h"
23
24      void init(void);
25      void loop(void);
```

**Listing 28:** Blinky main.c file

```
1       #include <extra.h>
2       #include <nrf_log.h>
3       #include <main.h>
4
5       static int init_blinky(void){
6           init();
7           return 0;
```

```
}

static int mainLoop(){
    loop();
    return 0;
}

/*
* Error code:
* 0 - No Errors
* 1 - No case executed in switch case
*/

static int arduinoSelect(int option){
    switch (option){
        case 1: return init_blinky(); break;
        case 2: return mainLoop(); break;
        default: return 1;
    }
}
```

**Listing 29:** Blinky extra.c file

```
void arduino(){
    LOGi("Calling handler struct");
    struct arduino_handler *iter = &__start_arduino_handlers;
    for (; iter < &__stop_arduino_handlers+1; ++iter) {
        LOGd("Call handler %s", iter->name); // print name of function
        LOGd("init() running %i", iter->f(1));
        LOGd("loop() running %i", iter->f(2));
    }
}
```

**Listing 30:** Bluenet cs_Crownstone.cpp file

The new code resulted in an error with the bluenet code. The bluenet code kept restarting itself in a loop. After investigating it was concluded that the peripheral code must not be called from main.c by the blinky binary. Only bluenet should be able to call periperal code. The code lines in listing 31 are in conflict bluenet code that cause the error to take place.

```
void init(){
    log_init(); // conflict with bluenet
    timer_init(); // conflict with bluenet
    leds_init(); // no conflict
    buttons_init(); // no conflict
    power_management_init(); // no conflict
    ble_stack_init(); // conflict with bluenet
    scan_init(); // no conflict with bluenet
    nrf_pwr_mgmt_run(); // no conflict with bluenet
    gatt_init(); // no conflict with bluenet
    db_discovery_init(); // no conflict with bluenet
    lbs_c_init(); // conflict with bluenet
}

```

```
15  void loop(){
16      idle_state_handle(); // no conflict with bluenet
17      scan_start(); // conflict with bluenet
18  }
```

**Listing 31:** Blinky main.c

It is recommended to create a macro function, registered by the bluenet binary in the `.arduino_handlers` section, that calls the peripheral functions from the bluenet binary to be used. This will make the blinky binary dependent on the bluenet binary for the peripheral and will not cause the code to get an error. An example on how the function could be build up is listed below. To register the function it would be:

```
REGISTER(bluArdHandler);
```

And then to create the function body itself:

```
bluArdHandler(int func_id) {
    switch(func_id){
        // Here is where the functions need to be defined that was meant to called
in the blinky code
        // BLE_SCAN_START for example
        // Find the right code in bluenet and call that here instead.
    }
}
```

# 7 Recommendation

Due to the time and difficulty of this assignment, the project could not be finished in time. The project is still in phase 1 and has yet to be completed and moved over to phase 2. All the information that is provided in this report is enough to continue from where the project has been temporarily halted.

The recommendation to on how to finish the project is goes as follows. The methods to run two binaries, that is discussed in chapters 6.2.1.9 and 6.2.1.10, has to be implemented for multiple functions. After it is possible to successfully execute blinky alongside bluenet, the blinky binary needs to be replaced by the arduino code. Running the arduino code, as discussed in chapter 6.1, requires to use its own library written by Sandeep Mistry. This is frequently used in the PlatformIO. The library is to used as inspiration and an example. It can be used to test and experiment during this project. Eventually the library that Sandeep created for the arduino code cannot be used for bluenet as a final product. Bluenet will require a custom library for its usage of arduino code.

When the running arduino code with bluenet code is a success, it needs to be added as a board option in PlatformIO. In chapter 5 PlatformIO has been briefly discussed. There it is explained on how to add new boards and create frameworks for custom embedded boards. Eventually it is the goal to also add the board to the Arduino IDE, but it is adviced to add the Crownstone board to PlatformIO because PlatformIO has made it easier to add custom boards. Als PlatformIO is used more by developers which is the target audience for the end product.

# References


[1] Binary file wiki. https://en.wikipedia.org/wiki/Binary_file.

[2] cksum manual. https://linux.die.net/man/1/cksum.

[3] Computer architecture: Interrupts. https://www.studytonight.com/computer-architecture/priority-interrupt.

[4] Elf binaries on linux. https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/.

[5] Elf wiki. https://wiki.osdev.org/ELF.

[6] Flash memory wiki. https://en.wikipedia.org/wiki/Flash_memory.

[7] hexdump manual. http://man7.org/linux/man-pages/man1/hexdump.1.html.

[8] Intel hex wiki. https://en.wikipedia.org/wiki/Intel_HEX.

[9] Interrupt point levels wiki. https://en.wikipedia.org/wiki/Interrupt_priority_level.

[10] ld manual. https://linux.die.net/man/1/ld.

[11] ldd manual. http://man7.org/linux/man-pages/man1/ldd.1.html.

[12] Link to project blinky files in bluenet-arduino. https://github.com/PaulWondel/bluenet-arduino/tree/master/arduinoCrownstone/version0.3/ble_app_blinky_c.

[13] Link to project bluenet files, the arduino version. https://github.com/PaulWondel/bluenet-arduino/tree/master/source.

[14] Link to project github repository bluenet-arduino. https://github.com/PaulWondel/bluenet-arduino/tree/master/arduinoCrownstone/version0.2/linkerscript_testing/DROPPED_address_testing.

[15] md5sum manual. https://linux.die.net/man/1/md5sum.

[16] Memory map wiki. https://en.wikipedia.org/wiki/Memory_map.

[17] objcopy manual. https://linux.die.net/man/1/objcopy.

[18] objcopy manual. https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_5.html.

[19] objdump manual. https://linux.die.net/man/1/objdump.

[20] Random-access memory wiki. https://en.wikipedia.org/wiki/Random-access_memory.

[21] srec_cat examples. https://manned.org/srec_examples/d510d1bb.

[22] srec_cat manual. http://srecord.sourceforge.net/man/man1/srec_cat.html.

[23] A. Allain. Function pointers in c and c++. https://www.cprogramming.com/tutorial/function-pointers.html.

[24] Arduino. Arduino software (ide). https://www.arduino.cc/en/Guide/Environment.

[25] Nordic Semiconductor. *nRF52 Preview Development Kit User Guide*. Nordic Semiconductor, v1.2 edition, 2 2017. Doc. ID 4397_497 v1.2.

[26] Nordic Semiconductors. Soc support in softdevice handle. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v14.2.0%2Fgroup__nrf__sdh__soc.html. v14.2.0.

[27] PlatformIO Revision. Platformio: advanced scripting. https://docs.platformio.org/en/latest/projectconf/advanced_scripting.html.

[28] PlatformIO Revision. Platformio: creating boards. https://docs.platformio.org/en/latest/platforms/creating_board.html.

[29] PlatformIO Revision. Platformio: custom platform and board. https://docs.platformio.org/en/latest/platforms/custom_platform_and_board.html.

[30] PlatformIO Revision. What is platformio? https://docs.platformio.org/en/latest/what-is-platformio.html.

[31] Red Hat. Sections command. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Using_ld_the_GNU_Linker/sections.html.

[32] Red Hat Developers. Linker scripts. https://sourceware.org/binutils/docs/ld/Scripts.html#Scripts.

[33] K. Reddy. What is flash memory? https://www.quora.com/What-is-flash-memory, October 2016.

[34] Team @ Tutorial Point. Embedded systems - interrupts. https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm.

[35] Team Crownstone. Bluenet. https://github.com/crownstone/bluenet.

[36] Team Crownstone. Configuration build. https://github.com/crownstone/bluenet/blob/master/docs/INSTALL.md.

[37] Team Crownstone. NRFConnect install guide. https://github.com/crownstone/bluenet/blob/master/docs/BUILD_SYSTEM.md.

# A Lists

## List of Tables

# List of Figures



# Listings

# B Company architecture

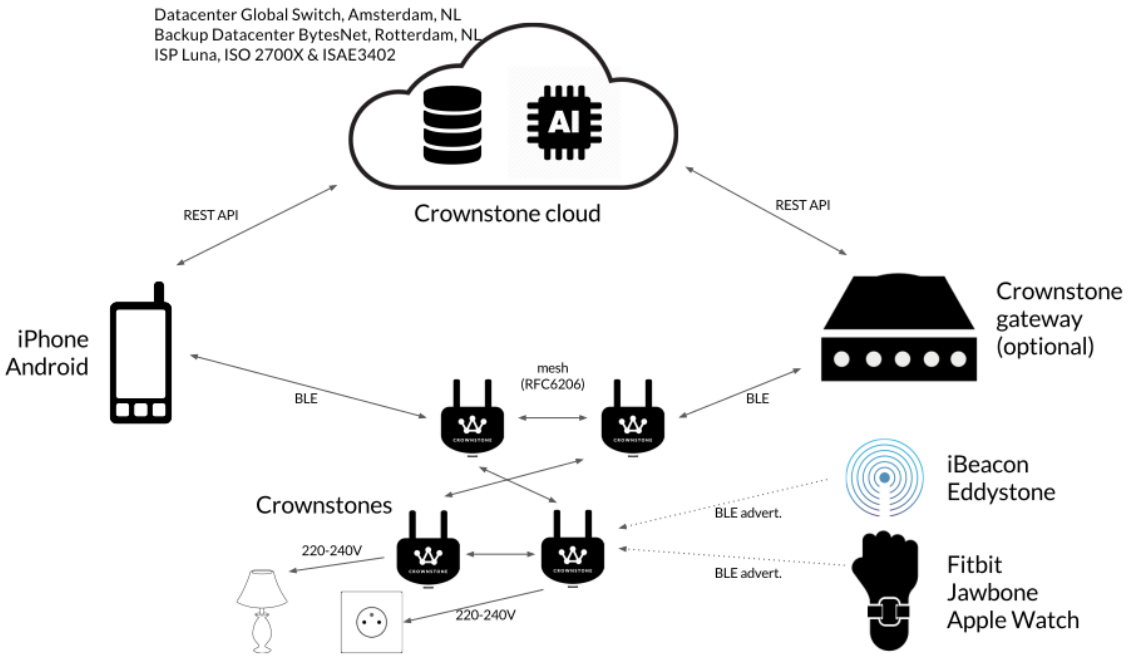



**Figure 16:** Original Crownstone Architecture

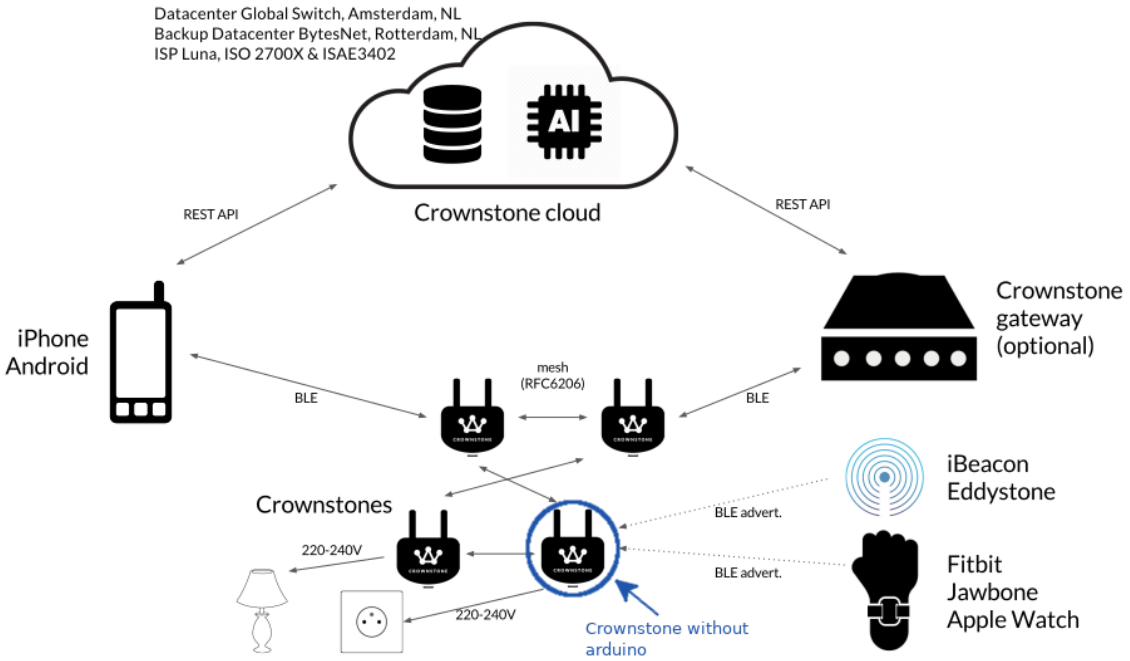



**Figure 17:** Current Crownstone Architecture

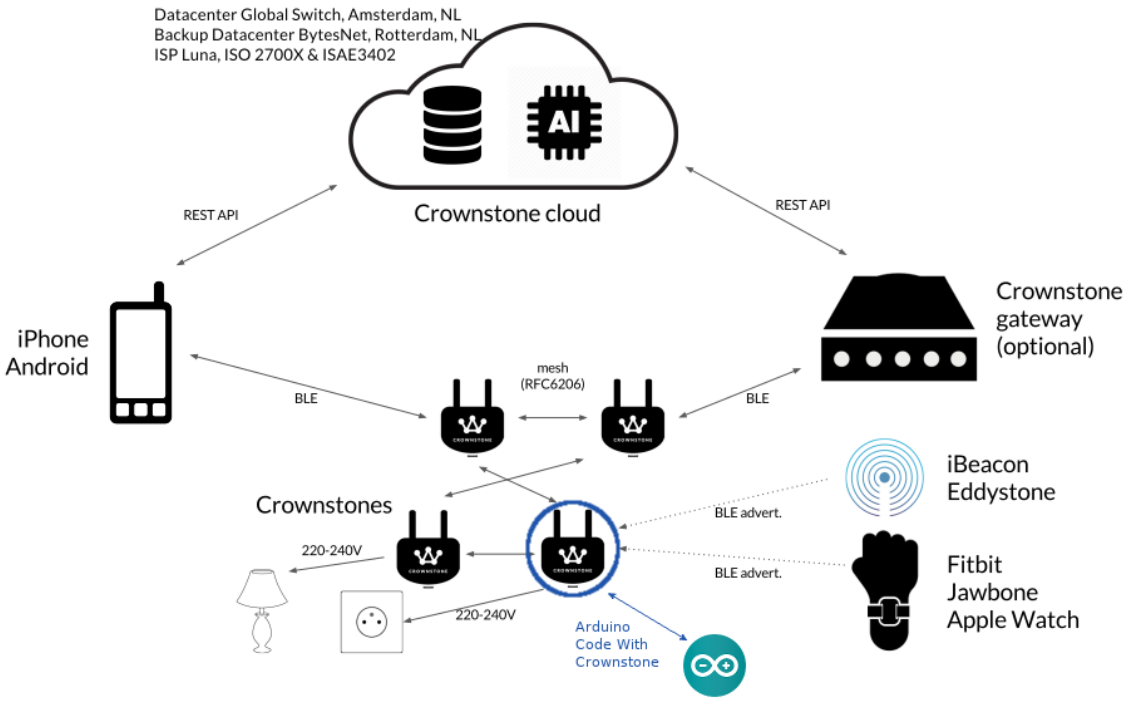



**Figure 18:** Desired Crownstone Architecture

# C Figures

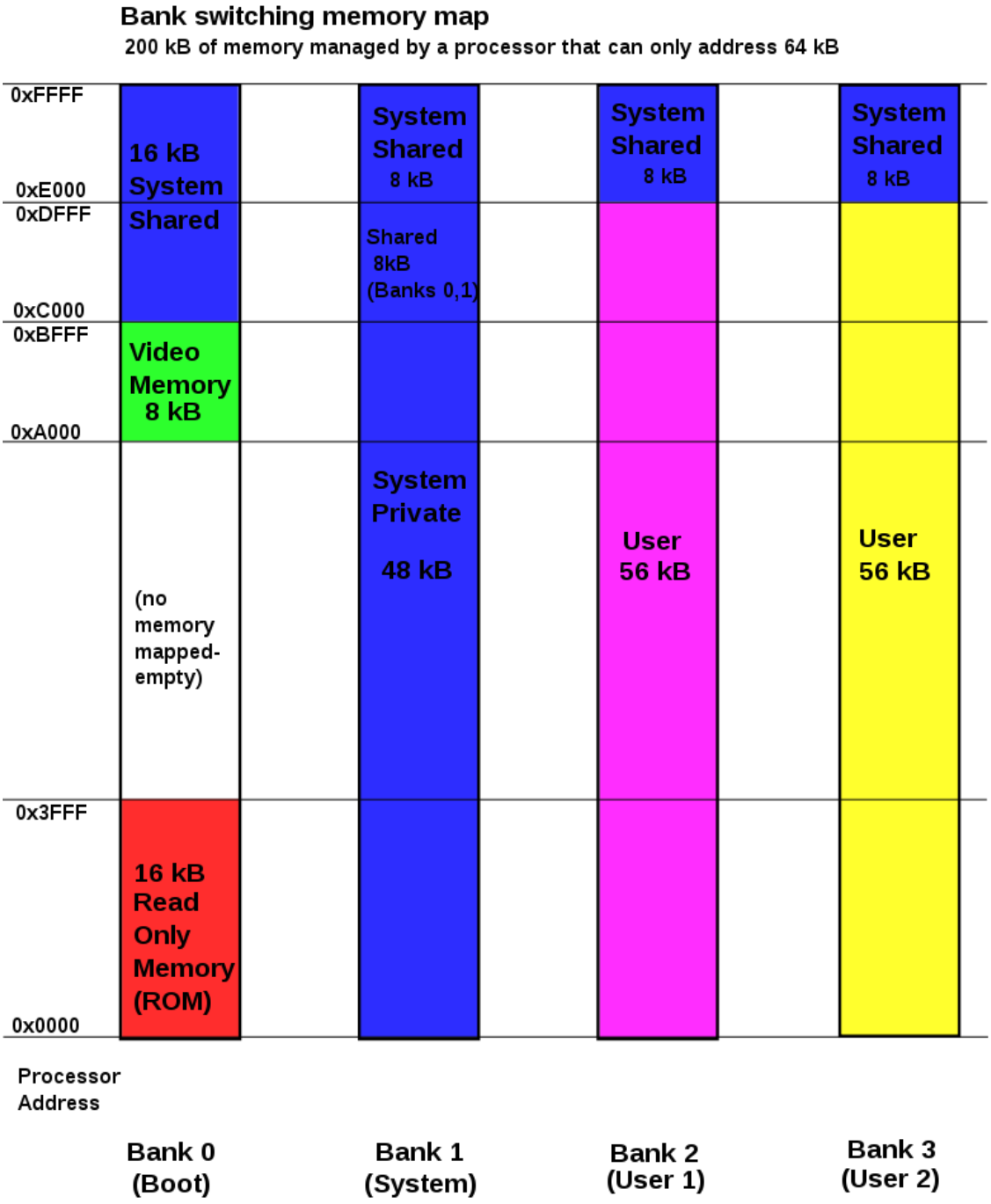



**Figure 19:** Example of a memory map [16]

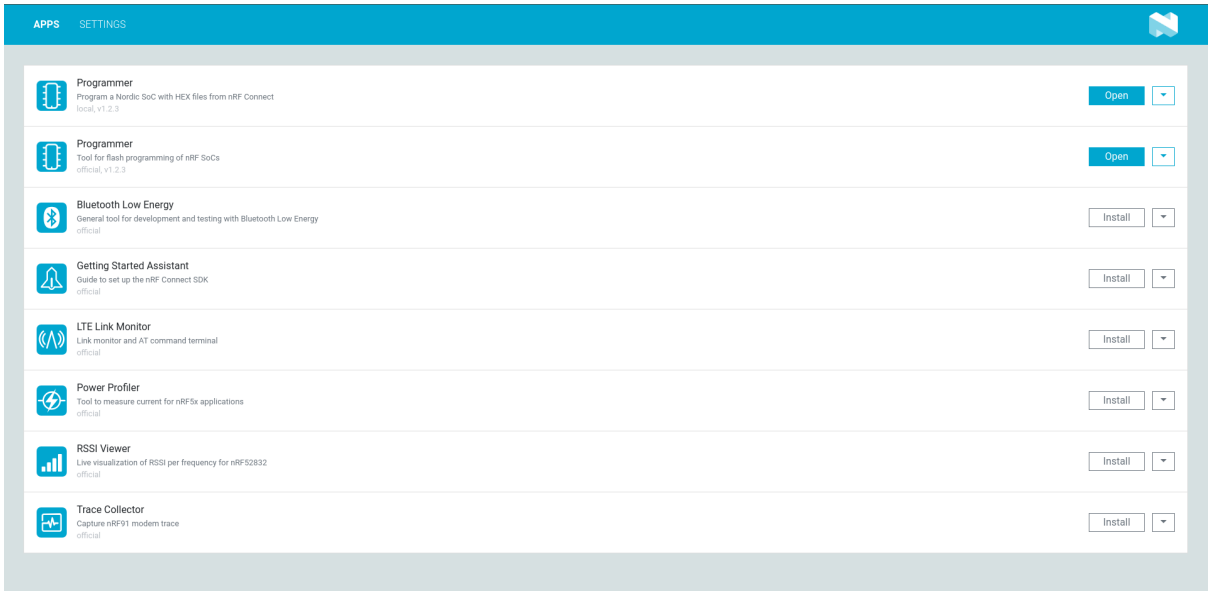



**Figure 20:** NRFCONNECT_CORE GUI

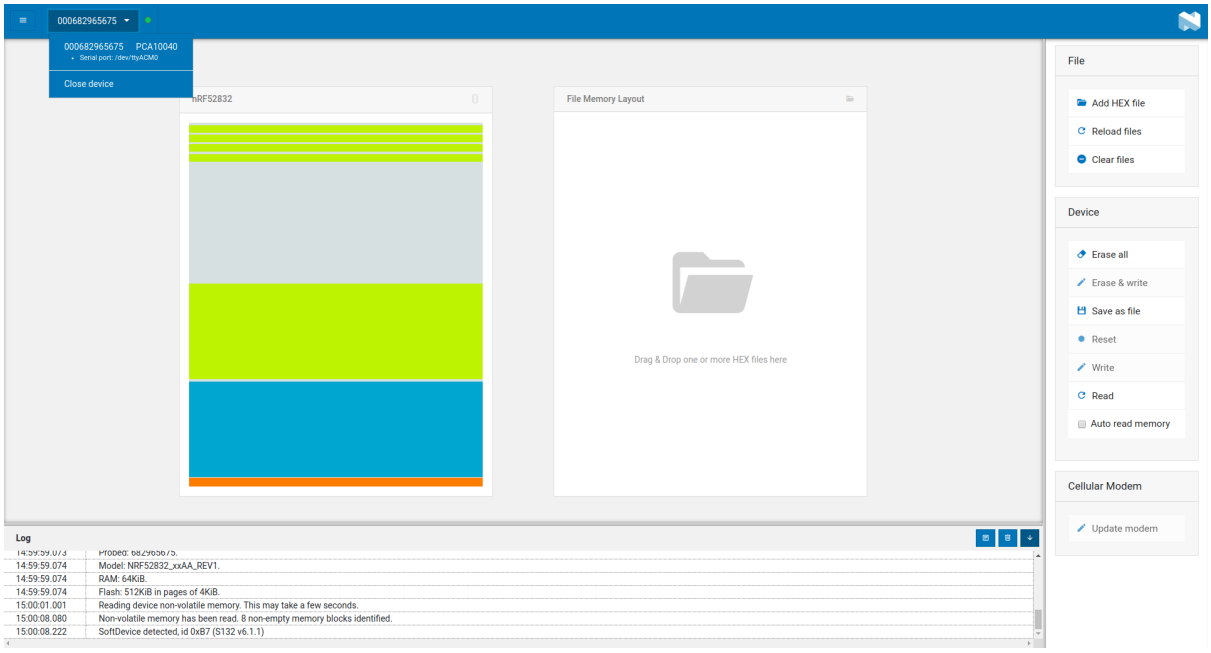



**Figure 21:** NRFCONNECT_PROGRAMMER GUI

You can enable the download of `nrfconnect` by:

```
cmake .. -DDOWNLOAD_NRFCONNECT=ON
make
```

This particular tool requires `npm`. Install it through something like `sudo apt install npm`. Subsequently, it downloads a lot of stuff, amongst which also `nrfjprog` it it cannot find it. Make sure it does not lead to version conflicts. You can run these by:

```
make nrfconnect_core_setup
make nrfconnect_core
```

These run in separate shells. The `_setup` you at least have to run once. After that it can do continuous rebuilds. You can select the tool to use from the list of apps. By default there are now quite a few apps there. The programmer can also be downloaded separately by setting the `-DDOWNLOAD_NRFCONNECT_PROGRAMMER` flag at `CMake`.

**Figure 22:** NRFCONNECT Install Guide Reference [37]

```
[~/gitfiles/bluenet-release/firmwares/crownstone_3.0.2/bin]$ readelf -hS crownstone.elf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x34ac1
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5323768 (bytes into file)
  Flags:                             0x5000400, Version5 EABI, hard-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         4
  Size of section headers:           40 (bytes)
  Number of section headers:         28
  Section header string table index: 27

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00026000 006000 023e98 00  AX  0   0 16
  [ 2] .sdh_soc_observer PROGBITS        00049e98 029e98 000018 00   A  0   0  4
  [ 3] .sdh_ble_observer PROGBITS        00049eb0 029eb0 000008 00   A  0   0  4
  [ 4] .sdh_state_observ PROGBITS        00049eb8 029eb8 000010 00   A  0   0  4
  [ 5] .sdh_stack_observ PROGBITS        00049ec8 029ec8 000010 00   A  0   0  4
  [ 6] .sdh_req_observer PROGBITS        00049ed8 029ed8 000010 00   A  0   0  4
  [ 7] .nrf_mesh_flash   PROGBITS        00049ee8 029ee8 0000d4 00  WA  0   0  4
  [ 8] .ARM.exidx        ARM_EXIDX       00049fbc 029fbc 000008 00  AL  1   0  4
  [ 9] .data             PROGBITS        20002380 032380 0000b8 00  WA  0   0  4
  [10] .fs_data          PROGBITS        20002438 032438 000014 00  WA  0   0  4
  [11] .bss              NOBITS          20002450 032450 004098 00  WA  0   0  8
  [12] .heap             PROGBITS        200064e8 032450 002000 00      0   0  8
  [13] .stack_dummy      PROGBITS        200064e8 034450 002000 00      0   0  8
  [14] .ARM.attributes   ARM_ATTRIBUTES  00000000 036450 000030 00      0   0  1
  [15] .comment          PROGBITS        00000000 036480 00007f 01  MS  0   0  1
  [16] .debug_info       PROGBITS        00000000 0364ff 22d2d2 00      0   0  1
  [17] .debug_abbrev     PROGBITS        00000000 2637d1 030828 00      0   0  1
  [18] .debug_loc        PROGBITS        00000000 293ff9 0492e9 00      0   0  1
  [19] .debug_aranges    PROGBITS        00000000 2dd2e8 0052f8 00      0   0  8
  [20] .debug_ranges     PROGBITS        00000000 2e25e0 00c1e0 00      0   0  1
  [21] .debug_macro      PROGBITS        00000000 2ee7c0 052f6a 00      0   0  1
  [22] .debug_line       PROGBITS        00000000 34172a 08e9ba 00      0   0  1
  [23] .debug_str        PROGBITS        00000000 3d00e4 106118 01  MS  0   0  1
  [24] .debug_frame      PROGBITS        00000000 4d61fc 011258 00      0   0  4
  [25] .symtab           SYMTAB          00000000 4e7454 017950 10     26 4304  4
  [26] .strtab           STRTAB          00000000 4feda4 014d02 00      0   0  1
  [27] .shstrtab         STRTAB          00000000 513aa6 000152 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
[~/gitfiles/bluenet-release/firmwares/crownstone_3.0.2/bin]$
```

**Figure 23:** Addresses of sections in the crownstone.hex file

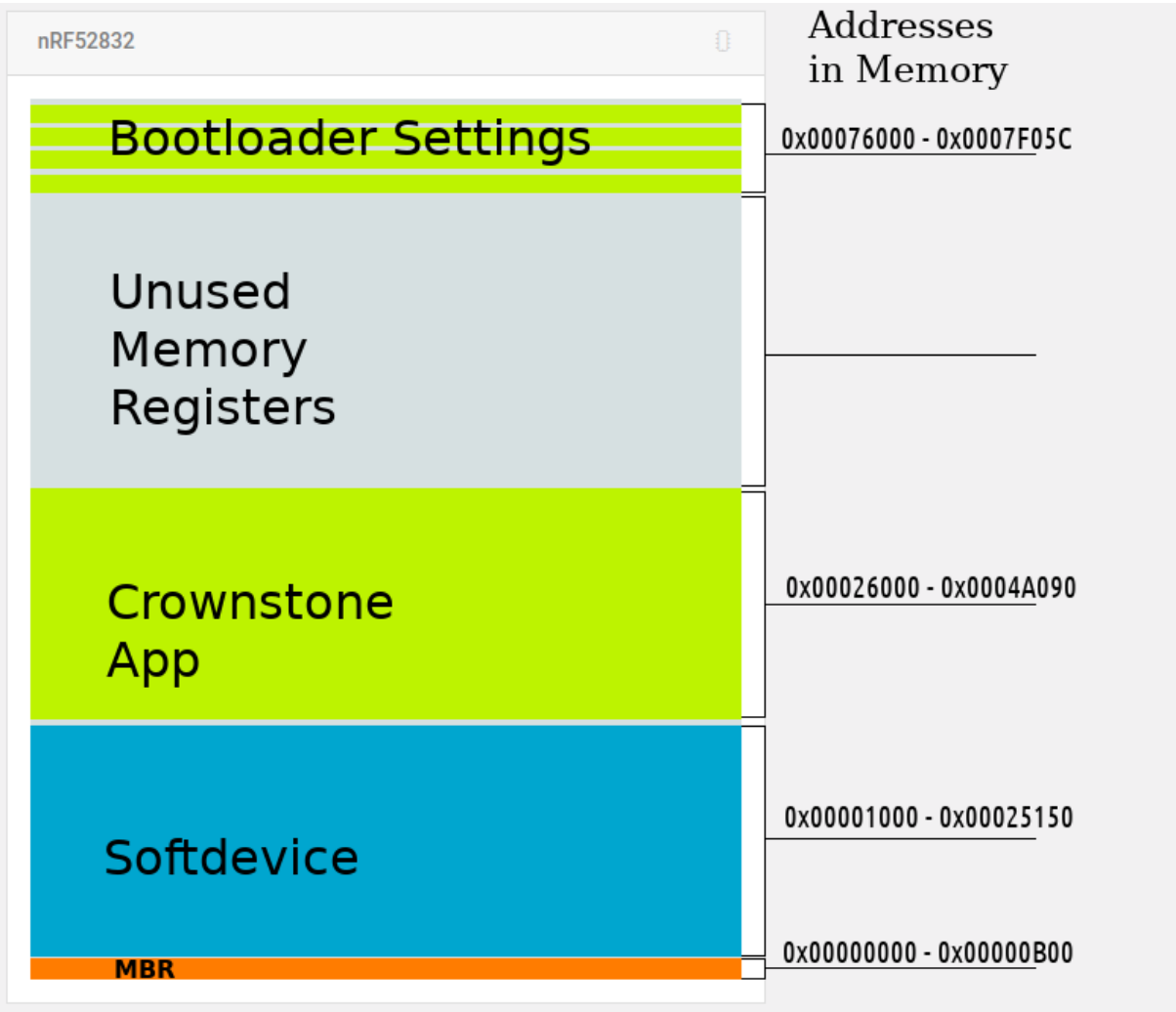



**Figure 24:** Memory map of the binary files on the flash memory of nRF52

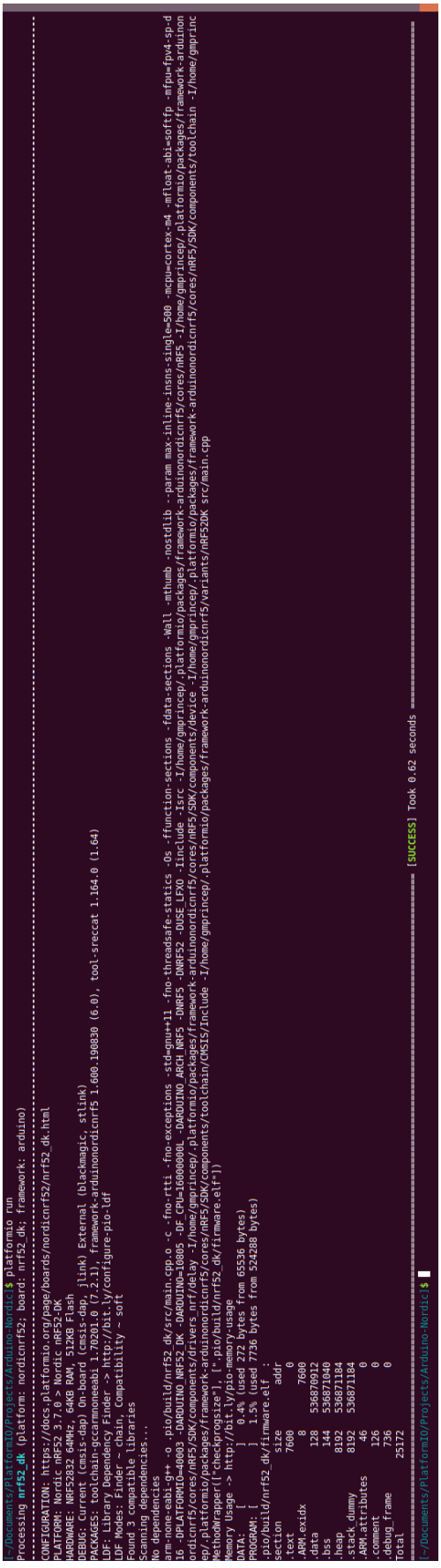



**Figure 25:** Verbose output platformio run command

## Configuration

If you want to configure for a different target, go to the `config` directory in the workspace and copy the default configuration files to a new directory, say `board0`.

```
cd config
cp -r default board0
```

Go to the build

```
cd build
cmake .. -DBOARD_TARGET=board0
make
```

The other commands are as in the usual setting.

Note, now, if you change a configuration setting you will also need to run `CMake` again. It picks up the configuration settings at configuration time and then passes them as defines to `CMake` in the bluenet directory.

```
cd build
cmake ..
make
```

Only after this you can assume that the `make` targets in `build/default` or any other target are up to date.

**Figure 26:** Configuration instructions on github
Reference [36]

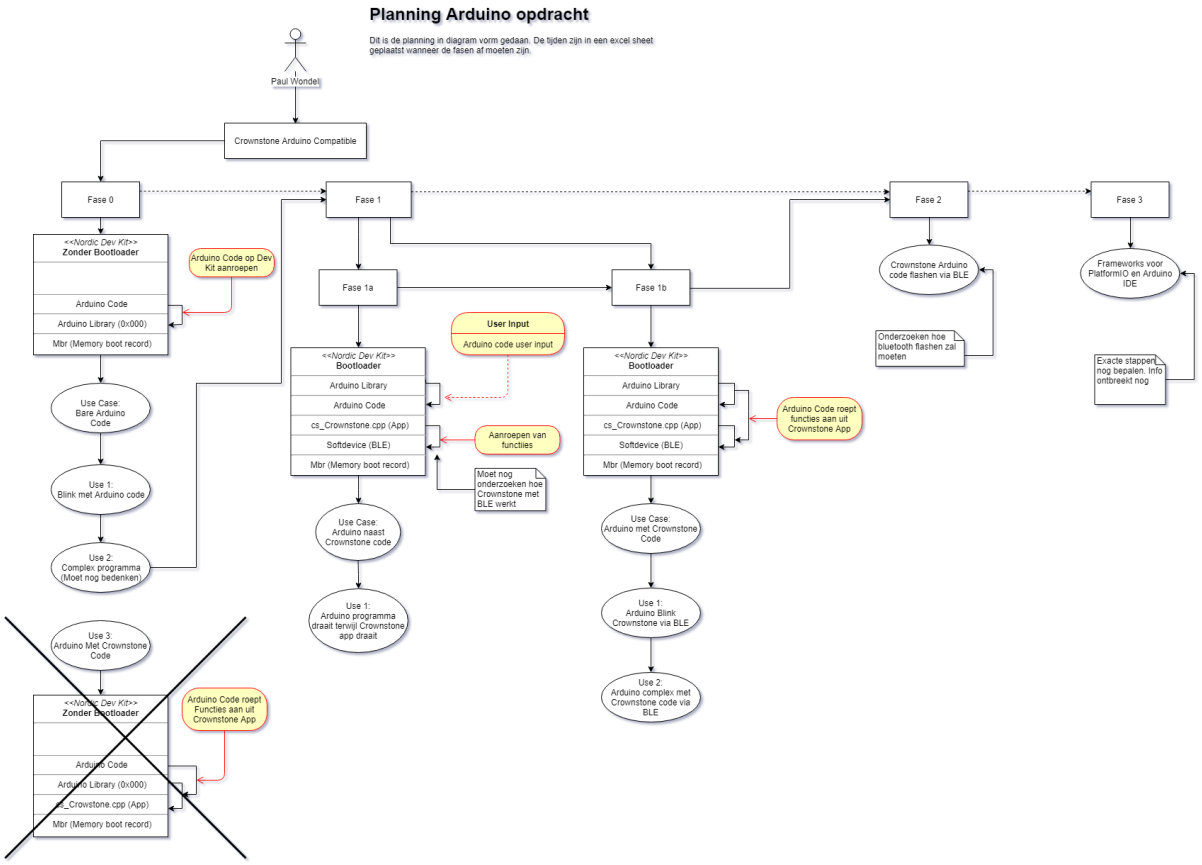



**Figure 27:** Phase Plan Workflow Scheme version 1

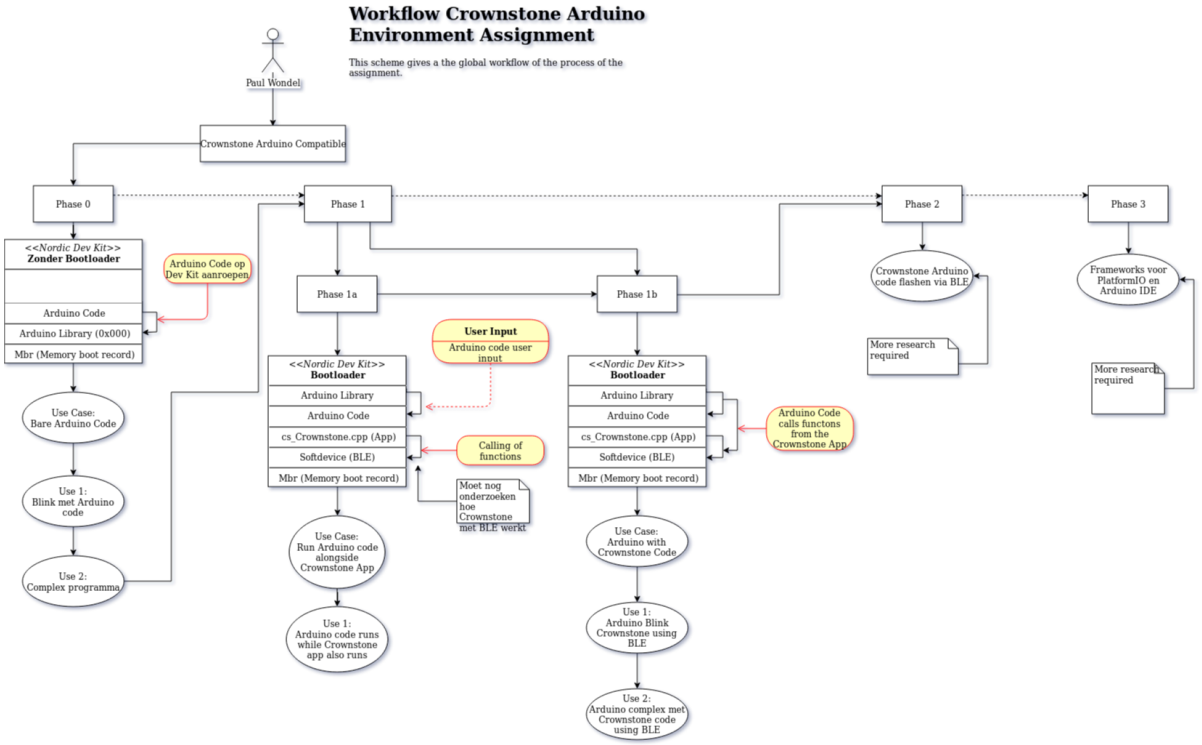



**Figure 28:** Phase Plan Workflow Scheme version 2

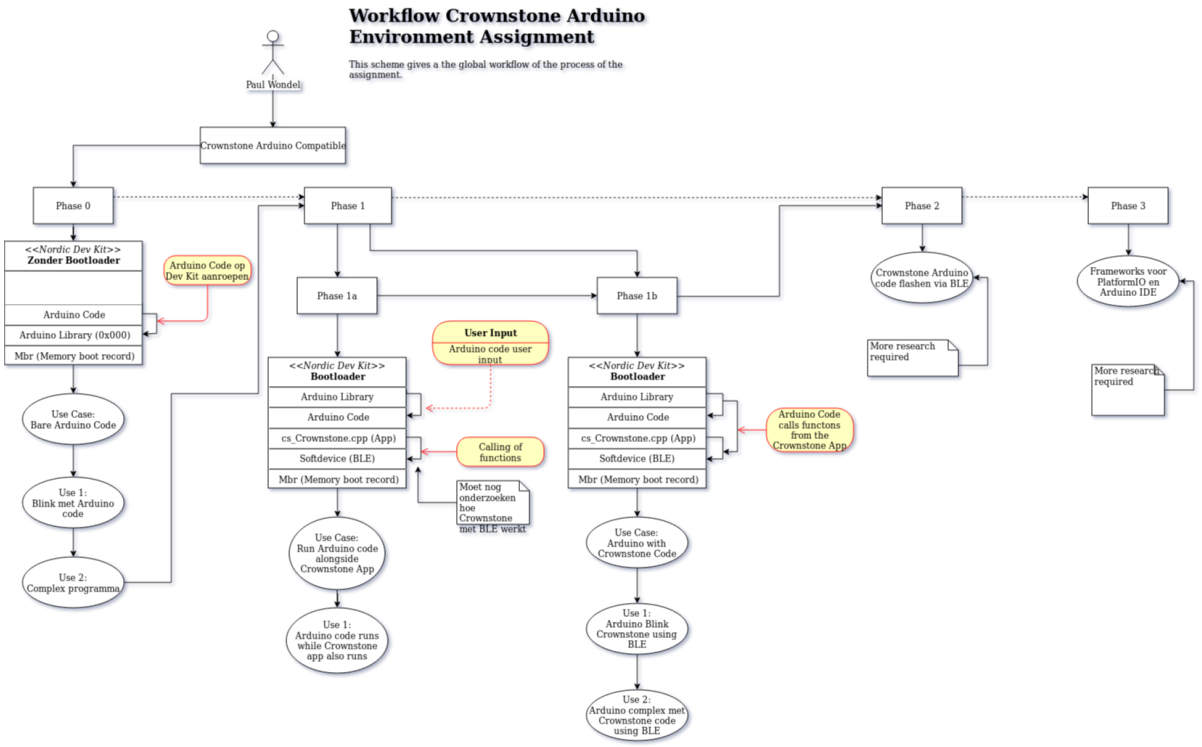



**Figure 29:** Phase Plan Workflow Scheme version 3

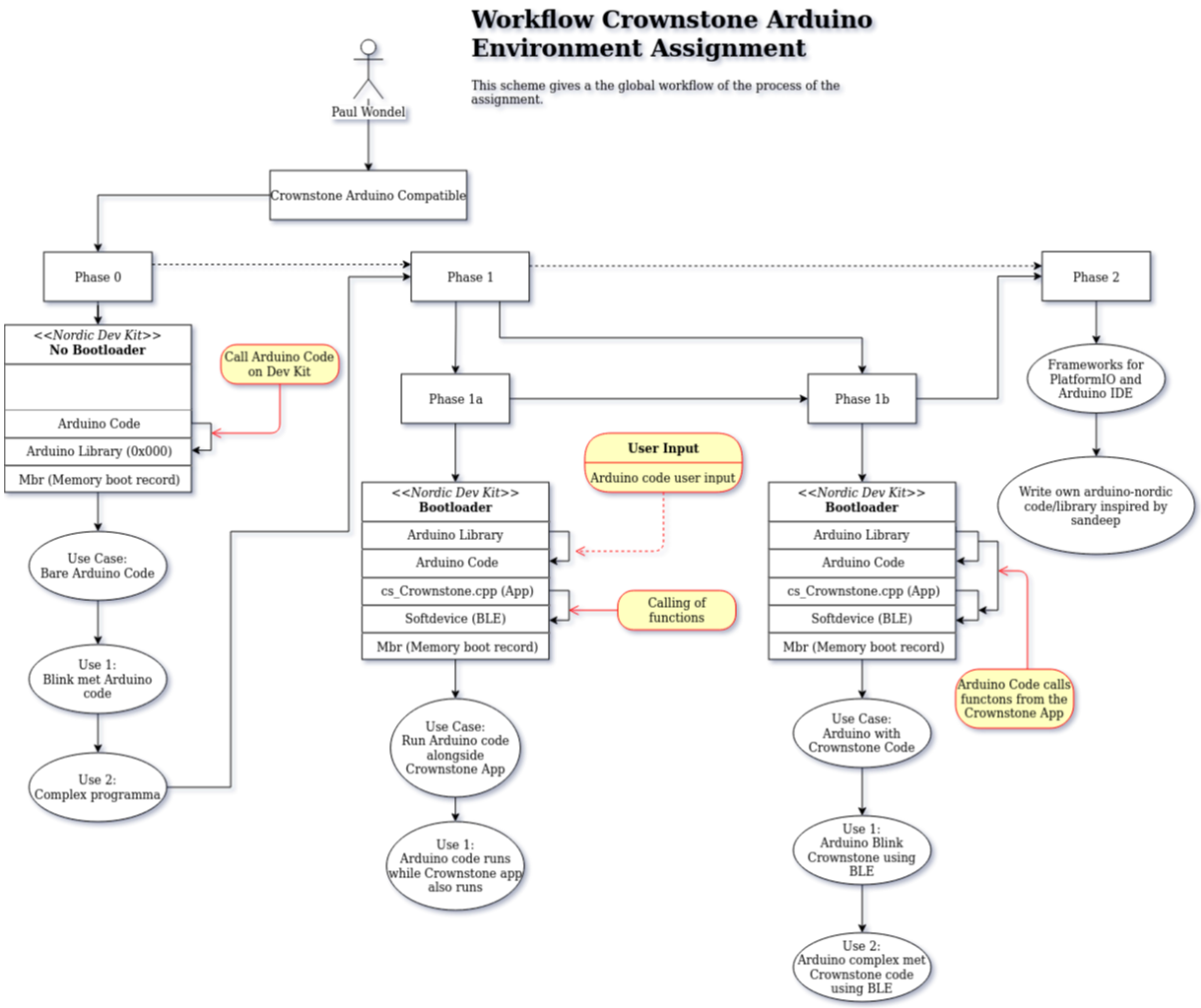



**Figure 30:** Phase Plan Workflow Scheme version Final

# D Listings

```
#This keeps the echo for every command silent that is executed in this makefile
ifndef VERBOSE
.SILENT:
endif

IDIR = /include #Input directory
CC = gcc #Compiler
DEPS = stdio.h main.h #Dependency
OBJ = main.o #Object files
CFLAGS = -I. # Compiler flags

#This is the command for suppressing the echo per command.
#.SILENT: main link_main link_test readelf_main readelf_test symbols_test clean
compile

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

main: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS)

compile:
    $(CC) -c *.c #-I$(IDIR)

link_main:
    ld main.o -e main -T arduino_handler.ld

link_test:
    ld -T arduino_handler.ld add.o

readelf_test:
    readelf -Sh Test

readelf_main:
    readelf -Sh main

symbols_test:
    nm Test

clean:
    rm -f *.o Test main
```

**Listing 32:** Makefile

# E Documents

# Testplan
Phase 0

Paul Wondel
*0947421*
Technische Informatica

HRo - Crownstone B.V. — November 15, 2019

## Project Information

In phase 0 the goal is to run raw arduino code on the nordic develepor kit (Nordic nRF52-DK). Phase 0 has been broken down to 2 use cases.

- Use case 1: Run Arduino Blink
- Use case 2: Auto Intensity Control of Power LED

This test is for use case 1. The goal is to test the basic functions of the arduino library on the Nordic nRF52 DK. The functions digitalWrite(), digitalRead(), analogRead() & analogWrite() are subjected to the test in this plan. Also the posibility to use the analog and digital pins on the Nordic nRF52-DK is to be tested. In order to test this there are a few requirements that need to be met.

## 1 Use Case 1

### 1.1 Requirements

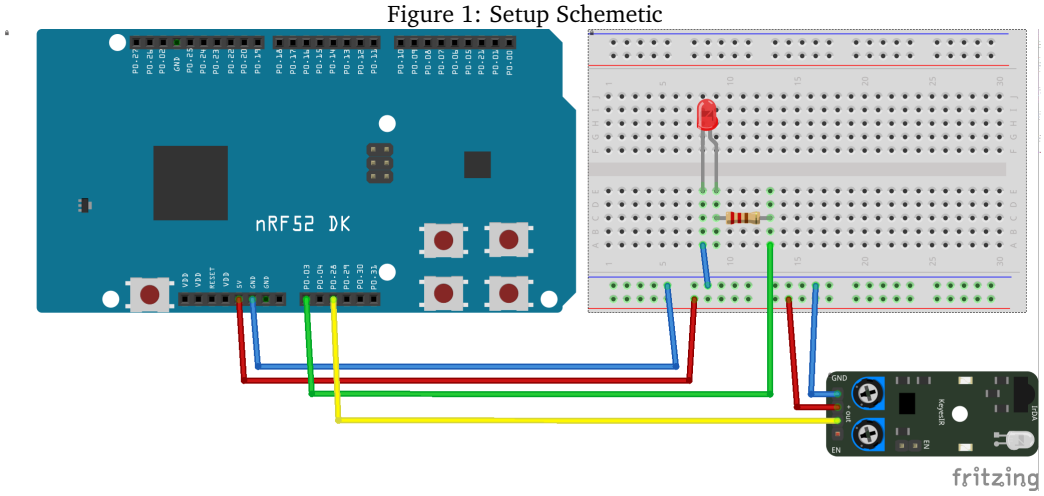
In preparations of this test these are the requirements that should be met:

- An Editor (Visual Code, Atom, eclipse, etc)
- Platformio IDE installed.
- Segger J-Link & tool-jlink (in Platformio) installed
- 1x Nordic nRF52 Developer Kit
- 1x Small LED
- 2x Circuit building wires
- 1x 220K Resistor
- Connecting Wires
- 1x Breadboard
- 1x Infrared Sensor
- Arduino Script: *'Blink'*

## 1.2 Constructing

These are the schematics and code for the test setup. Setup the parts in the same positions as in the schematics. Then upload the code to the board.

Figure 1: Setup Schemetic




Listing 1: Test Code

```
#include <Arduino.h>

#define ledPin PIN_A0
#define irPin PIN_A1

void setup() {
  Serial.begin(9600);
  pinMode(ledPin,OUTPUT);
  pinMode(irPin,INPUT);
}

void loop() {
  Serial.println(analogRead(irPin)); // Shows the values from the sensor
  if(analogRead(irPin) < 600){
    digitalWrite(ledPin, HIGH); // set the LED on
    Serial.println("Led On");
  }
  else {
    digitalWrite(ledPin, LOW); // set the LED off
    Serial.println("Led Off");
  }
}
```

## 1.3 Testing

**Instructions:** After constructing the circuit and uploading the script, the Infrared Sensor has to be configured. The Infrared Sensor has it's own resistor on the circuitboard which can be adjusted with a screwdriver. When the sensor has been configured, test the object detection. The values should appear in the serial monitor. The low values (when there is no object detected) should be between 0-200 and the high

values should be higher than 600. To test the setup place an object in front of the sensor. The led should light up when the object is detected by the sensor.

**Questions during the performance** For performing the test itself there are a few questions that need to be answered during the test.

Question 1

Does the LED turn on when an object is placed in front the sensor?

a) Yes b) No

Question 2

Does the Infrared Sensor detect the object when placed in front of the sensor? (The light on the sensor itself wil light up when detecting an object)

a) Yes b) No

Question 3

Do the printed lines 'Led on' or 'Led off' appear in the serial monitor?

a) Yes b) No

# 2 Use Case 2

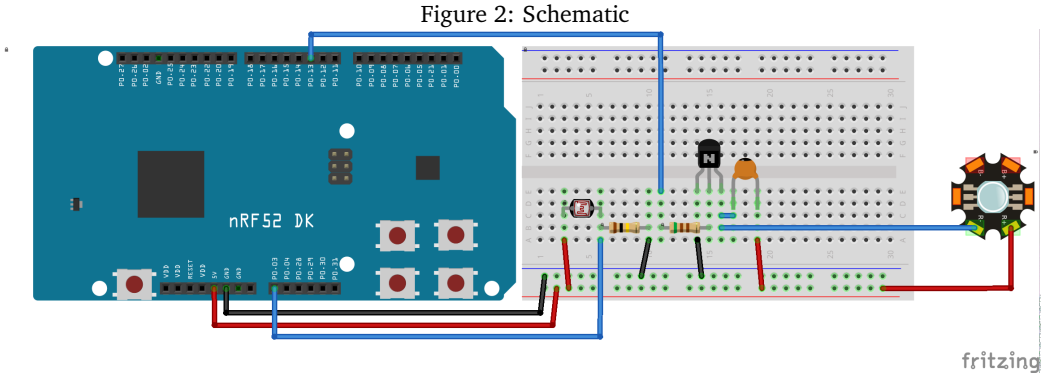
## 2.1 Requirements

- A Editor (Visual Code, Atom, eclipse, etc)
- Platformio IDE installed.
- Segger J-Link & tool-jlink (in Platformio) installed
- 1x Nordic nRF52 Developor Kit
- 1x LDR (Light Density Resistor) Sensor
- 1x Resistor (510, 100k ohm)
- 1x Capacitor (0.1uF)
- 1x Transistor 2N2222
- 1x 1 watt Power LED
- Connecting Wires
- 1x Breadboard
- Flashlight or mobile light source

## 2.2 Constructing

These are the schematics and code for the test setup. Setup the parts in the same positions as in the schematics. Then upload the code to the board.

Figure 2: Schematic




Listing 2: Test Code

```
#include <Arduino.h>

int pwmPin = (2); // assigns pin 12 to variable pwm
int pot = A0; // assigns analog input A0 to variable pot
int c1 = 0;   // declares variable c1
int c2 = 0;   // declares variable c2

void setup()  // setup loop
{
  pinMode(pwmPin, OUTPUT);
  pinMode(pot, INPUT);
  Serial.begin(9600);
```

```
}

void loop()
{
  int value = analogRead(pot);
  Serial.println(value);
  c1= value;
  c2= 500-c1;          // subtracts c2 from 1000 ans saves the result in c1

  if (value < 500)
  {
  digitalWrite(pwmPin, HIGH);
  delayMicroseconds(c2);
  digitalWrite(pwmPin, LOW);
  delayMicroseconds(c1);
  }
  if (value > 500)
  {
    digitalWrite(pwmPin,LOW);
  }
}
```

## 2.3 Testing

**Instructions:** After constructing the circuit and uploading the script, the setup is ready to be tested. To test the setup you must aim the light source in straight unto the LDR Sensor. This should cause the LED (that is turned on by default) to turn off.

**Questions during the performance** For performing the test itself there are a few questions that need to be answered during the test.

Question 4

Does the LED turn off when you shine light unto the LDR?

a) Yes b) No

Question 5

Does the serial monitor show the values given by the LDR?

a) Yes b) No

# Testplan
Phase 1

Paul Wondel
*0947421*
Technische Informatica

HRo - Crownstone B.V. — February 7, 2020

## Project Information

The goal in phase 1a is to run the arduino code that is created in phase 0 use case 2, at the same time that the crownstone code is running.

## 1 Use Case 1

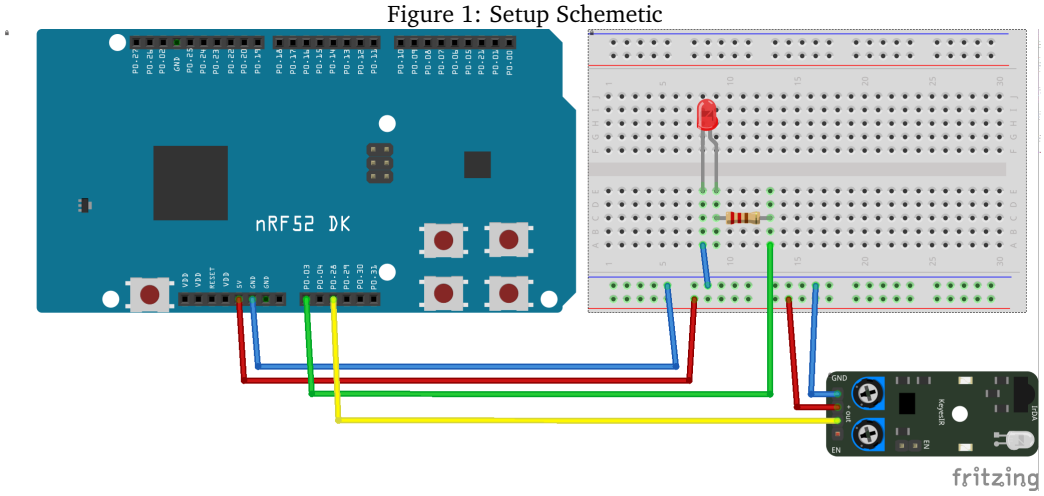
### 1.1 Requirements

- Crownstone bluenet cloned git repository
- An Editor (Visual Code, Atom, eclipse, etc)
- Platformio IDE installed.
- Segger J-Link & tool-jlink (in Platformio) installed
- 1x Nordic nRF52 Developer Kit
- 1x Small LED
- 2x Circuit building wires
- 1x 220K Resistor
- Connecting Wires
- 1x Breadboard
- 1x Infrared Sensor
- Arduino Script: *'Blink with use of Infrared Sensor'*
- Crownstone Smartphone App (Iphone or Andriod)

## 1.2 Constructing

These are the schematics and code for the test setup. Setup the parts in the same positions as in the schematics.

Figure 1: Setup Schemetic




Listing 1: Test Code

```
#include <Arduino.h>

#define ledPin PIN_A0
#define irPin PIN_A1

void setup() {
  Serial.begin(9600);
  pinMode(ledPin,OUTPUT);
  pinMode(irPin,INPUT);
}

void loop() {
  Serial.println(analogRead(irPin)); // Shows the values from the sensor
  if(analogRead(irPin) < 600){
    digitalWrite(ledPin, HIGH); // set the LED on
    Serial.println("Led On");
  }
  else {
    digitalWrite(ledPin, LOW); // set the LED off
    Serial.println("Led Off");
  }
}
```

## 1.3 Testing

**Instructions:** After constructing the circuit and uploading the bluenet code along with the crownstone code, the Infrared Sensor has to be configured. The Infrared Sensor has it's own resistor on the circuitboard which can be adjusted with a screwdriver. When the sensor has been configured, test the object detection. The values should appear in the serial monitor. The low values (when there is no object detected) should

be between 0-200 and the high values should be higher than 600. To test the setup place an object in front of the sensor. The led should light up when the object is detected by the sensor. During testing the arduino code has to function normally as it would on a arduino uno. You place an object in front of the sensor and the LED should turn on, remove the object and the LED should turn off. While that is being tested, the Crownstone App on the smartphone should detect and connect normally to the nordic device as if it was a released Crownstone plug.
If the nordic device cannot be detected, then it means that the bluenet code is not being executed. If the LED are not working but the nordic is being detected, it means that the arduino code is not being executed.

**Questions during the performance** For performing the test itself there are a few questions that need to be answered during the test.

Question 1

Does the LED turn on when an object is placed in front the sensor?

a) Yes b) No

Question 2

Does the Infrared Sensor detect the object when placed in front of the sensor? (The light on the sensor itself wil light up when detecting an object)

a) Yes b) No

Open the crownstone app on a smartphone.

Question 3

Can your smartphone detect the nordic device with bluetooth using the crownstone app?

a) Yes b) No

Question 4

Can your smartphone connect the nordic device with bluetooth using the crownstone app?

a) Yes b) No

# Test Report
Phase 0

Paul Wondel
*0947421*
Technische Informatica

HRo - Crownstone B.V. — February 7, 2020

Preformed on: October 7, 2019
Preformed by: Paul Wondel

## Project Information

In phase 0 the goal is to run raw arduino code on the nordic develepor kit (Nordic nRF52-DK). Phase 0 has been broken down to 2 use cases.

- Use case 1: Run Arduino Blink
- Use case 2: Auto Intensity Control of Power LED

The goal is to test the basic functions of the arduino library on the Nordic nRF52 DK. The functions digitalWrite(), digitalRead(), analogRead() & analogWrite() are subjected to the test in this plan. Also the posibility to use the analog and digital pins on the Nordic nRF52-DK is to be tested. In order to test this there are a few requirements that need to be met.

## 1 Use Case 1

### 1.1 Execution

Constructing the setup is has been fairly simple. By simply following the schematic, the prototype has been build. Using PlatformIO, uploading the code went successfully without error.

### 1.2 Results

The results after following the steps from testplan phase 0 for use case 1 met its criteria. The code had been executed as expected. Every time there was a object placed in front of the sensor, the LED turned on. Is was consistant with the small onboard LED of the infrared sensor. Whenever there had been an object placed infront of the sensor and the LED turned on and off, the code also printed the phrase "Led on" and "Led off". There had not been any errors during the execution of this test.

## 2 Use Case 2

### 2.1 Execution

Constructing the setup is has been fairly simple. By simply following the schematic, the prototype has been build. Using PlatformIO, uploading the code went successfully without error.

## 2.2 Results

The results after following the steps from testplan phase 0 for use case 2 met its criteria. The code had been executed as expected. With the use of the flashlight from the phone, the LDR sensor (Light Dependent Resistor) reacted everytime a the flashlight was directed towards it. The LED is turned on by default and turns off whenever the LDR detects a strong light source or beam. Whenever the flashlight was directed to the LDR, the Led turned off. The serial monitor also printed the values that were read by the LDR. This test had been successfully executed without any errors.