Bluetooth Ultra-simple Gamepad

Design of a wireless cooperative gaming device

Floris van der Heijde
Jorn van der Linden
Koen van Remundt

23rd December 2021

TU Delft

Abstract

The design, production and testing of the Bluetooth Ultrasimple Gamepad, or BUG for short, is laid out in this report. In the current market, most of the controllers available are multi-button controllers with analogue sticks. These controllers are expensive and can not be used for many web browser games which rely on a keyboard input to work. In order for these controllers to work, a video game or self developed app can be used, but this can be difficult or expensive for many users. In the implementation of the BUG, the controller was made to be used by a single hand, having a clear ”main” button, an intuitive reconfiguration scheme and to have Bluetooth connection which is present in a lot of laptops and PCs currently in use. With this approach, the user can have a cheap, simple and comfortable alternative to existing gamepads. It can be stated that, although not for everybody, the BUG is a good and fun alternative for users. This can be seen because the general user experience is rated at 5 out of 5 points by 50% of the participants, 4 out of 5 by 46.7% of the participants and only 1, or 3.3% of the participants has given a rating of 2 out of 5.

Preface

The project took place over 10 weeks during which the team members worked towards the development of a Bluetooth Ultra-simple Gamepad. The team consisted of 3 members: Floris van der Heijde, Jorn van der Linden, and Koen van Remundt. All three members have completed the vast majority of the Electrical Engineering Bachelor and the Delft University of Technology.

The project was suggested by prof.dr.ir. Rob Kooij as a way to do research on the differences in collaboration within teams of heterogeneous and homogeneous cultural background. During the project an initial prototype was developed as a proof of concept which happened rather quickly. Thus the choice was made to develop a set of more fully developed prototypes.

We would like to thank prof.dr.ir Rob Kooij for his suggestion of the project and his suggestions throughout the project. We would also like to thank dr.ing. I.E. Lager for his guidance throughout the project.

Floris van der Heijde, Jorn van der Linden, and Koen van Remundt
December 2021

Contents

Chapter 1

Introduction

1.1 Gamepads

In the gaming industry the vast majority of gamepads is designed for one person to control the full input of the game. For multiplayer games it is commonly the case that each person has their own gamepad with all full input possibilities and controls their own part of the game for example their own character. In some cases multiple people have to work together to control one entity but this still is done with full gamepads each and having relatively broad control input.

The proposed idea by Rob Kooij is to make gamepads with only one button each so every person can only control a single input of the game. The gamepads would have to connect to a computer. This would result in multiple people having to work together to control a game, for example solving a maze. The eventual interest would be to see if there is a significant difference in how homogeneous and heterogeneous groups of different cultures work together in completing the game.

Throughout the thesis certain terms and jargon will be used. In some cases specific choices were made to avoid confusion. An overview of the most important terms and jargon can be found in Appendix A.

1.2 State of the art analysis - Gamepads

The current market of gamepads is in vast majority the more complicated x-box and PlayStation style. Simpler gamepads do exist, such as the Nintendo Switch Joy-Cons. However gamepads with only a single button don't really exist. There are some big red button style gamepads for quiz style gameplay but nothing really which is made to work as a team on controlling a game. One of the largest video game digital distributors, Steam, did data analysis on the gamepads used with their platform. From this analysis it was found that PlayStation and Xbox controllers account for 92% of the used gamepads[1]. Furthermore a market research team performed a market analysis on the gamepad market and determined the 6 most important companies in this market[2]. None of these companies offer single button gamepads on their webstore.

Wireless gamepads often use Bluetooth to interface with the computer and control the game. A way to communicate keystrokes to control a game would be to use a wireless keyboard. There are quite a few hobby projects only that discuss making a Bluetooth keyboard, one project uses that principle to make a simple gamepad with direction controls and two buttons. This final project can be simplified to just be essentially a one button Bluetooth keyboard. For a single game 3-5 gamepads would be needed to control a game. Given that most Bluetooth systems support a maximum of 7 simultaneous connections, this would be fine for one team. However it

could be interesting to allow for multiple teams to play simultaneously this would require more than 7 connections. To facilitate more than seven gamepads a Bluetooth mesh network could be a possibility or a tree structure where every set of gamepads has one host which is the interface between the computer and gamepads.

1.3 State of the art analysis - Bluetooth protocol

A common wireless protocol to connect peripherals to computers is the Bluetooth protocol. This protocol was introduced in 1998 to reduce the amount of cables used. Currently the Bluetooth technology could be split into two parts: Bluetooth Classic and Bluetooth Low Energy (BLE) [3]. Bluetooth Classic is based on the original technology and consist of various profiles, such as data transfer, audio streaming or a HID profile. A HID (Human Interface Device) profile is used for communication with a keyboard, a mouse, or a gamepad.

The BLE technology is the newer version, introduced with Bluetooth version 4.0 in 2010. The technology makes very low power communication possible and uses another method of communication. The communication is based on the General Access Profile (GAP) and the General Attribute Profile (GATT) [4]. These profiles can be seen as network layers, each containing multiple other layers. The GAP is used as a framework to discover other devices and establish a connection. GATT provides a framework over BLE to communicate and interchange data. By defining a standard how certain data is formatted, it makes the protocol compatible between different manufacturers.

The GATT is based on attributes, which are grouped in services. An example is an HID service, combining multiple attributes together which are all directly related to the HID. The battery percentage is not directly related, so this would be grouped in another service. Each service contains zero or more characteristics: these contain the user data. The user data itself is send or received using descriptors: each characteristic contains zero or more of them. A schematic view of this can be seen in Figure 1.1.

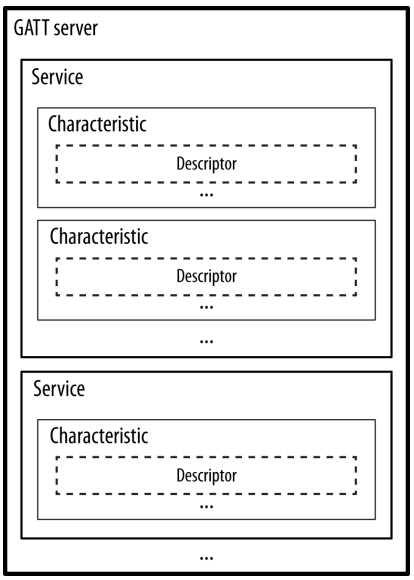


Figure 1.1: Schematic overview of the GATT protocol, as shown in [4].

All characteristics and services are identified using an unique user ID (UUID). Most common data is send using services with a UUID defined by Bluetooth SIG. This concept makes it possible the protocol still works when using devices made by different manufacturers. If data is not defined by the list from Bluetooth SIG, another UUID can be chosen, as long as it does not conflict with a predefined UUID.

Using BLE, is can be unclear which device is the server and which is the client. It is intuitive to call a computer a server, but most of the time this is not the case: with a HID the computer uses the data send from the HID and is thus the client. Thus, a HID is usually a BLE server.

Chapter 2

Programme of Requirements

2.1 Project requirements

- The gamepads need to be safe.
- The gamepads have to emulate a keystroke on the computer.
- Must make it possible to play simple games such as Pac-Man.
- Test multiple different implementations of the technology and controller designs.
- The use of the devices must be user friendly.
  - The device must be wireless
  - It must be clear to the user which key the user is controlling
  - It must not be difficult to connect a BUG to a computer
- The designed gamepad needs to be tested by at least sixteen players, playing in teams of four (not simultaneously).

2.2 Technical requirements

- Device must be able to work wireless.
  - The used protocol for communications must be able to work without additional custom drivers or software.
  - The used protocol must be able to connect easily.
- Device must be rechargeable.
  - It must contains a rechargeable battery.
  - The battery must be charged safely.
  - The user must be able to see if the device is fully charged or not.
  - The charging time is at most the same as the time the device can be used on a single charge.
- Device must be able to run for at least an hour.
  - The battery must have a large enough capacity.

  - The power consumption must be minimised.
- At least four gamepads should be able to work simultaneously.
- The gamepad design must contain at least:
  - A button.
  - A method to switch the game controller off.
  - A form of indication to the user which key bind is currently assigned to the gamepad.

2.3 Additional requirements

The following requirements are not mandatory requirements, but it could be argued these follow the user friendly-requirement from section 2.1.

- A way to reconfigure the key binding from the gamepad itself.
- Reconfiguring the key binding and other settings through optional PC software.
  This gives a wider range of possibilities for key bindings and thus applications for the gamepad.
- Testing with more than eight players simultaneously using the same host PC.
  This makes it clear if it is possible to connect and use more gamepads simultaneously, so other games could be played.

Chapter 3

Prototype Design

3.1 3D Design

While designing the 3D model, two form factors were worked out. Both shapes will be printed using a 3D printer and the preference of the participants will be gathered. These two form factors are the Credit Card and the nunchuck.

3.1.1 Gamepad design

Credit Card Model

The first model, a classic, SNES-like custom box was designed with the dimensions of a credit card (see Figure 3.1a). This model is fairly simple, mainly to get a feeling of the model and to test the Bluetooth connection and software. It was also used to get some preferences for the LED layout from test participants.

The design process is very straight forward, a simple box is designed with filleted corners. On the top side, some mounts for LEDs are made to test some indicator layouts. On the other side of the top surface, a mount for a button is made, which is a small 6mm tactile push button. Some cutouts are added to make it possible to add a wider, more comfortable button cap in the future. Inside, some mounting holes for the used development kit of the ESP32-C3 are added. Why this MCU is picked will be discussed in subsection 3.3.3.

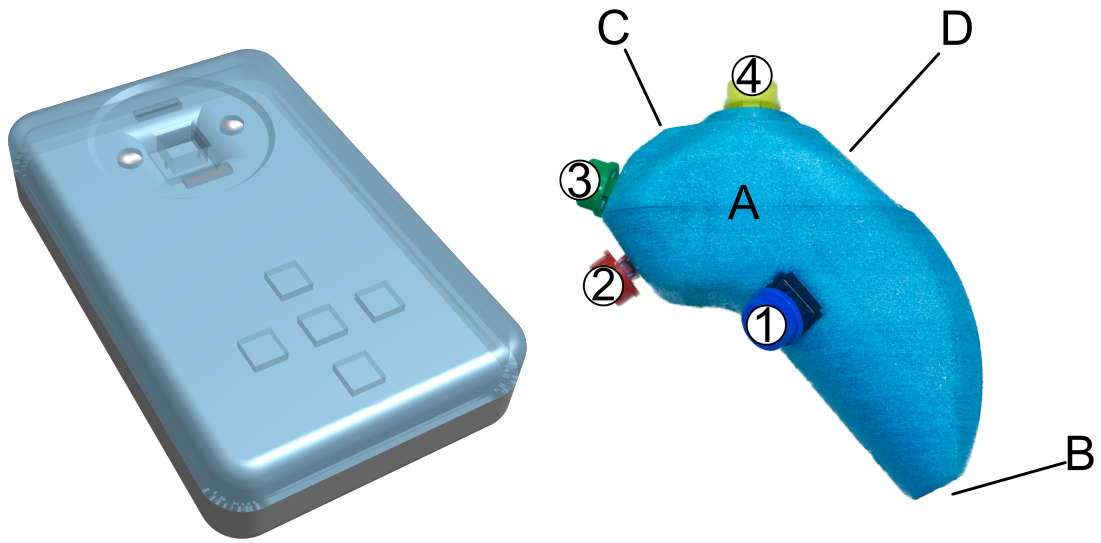
Nunchuck Model

The second model is a more ergonomic model, very comparable to the Nintendo® Wii Nunchuck. This design is used to test several locations of the buttons.

Instead of reinventing the wheel, an online model (licensed with Creative Commons - Attribution) is used for initial testing [5]. This model is made to accompany three (12mm) tactile push buttons and a joystick, but an adapter is designed to fix a fourth push button on top. Inside an Arduino Pro Mini is fitted to read the button states and send them over an UART serial connection to a PC. A picture of this model can be seen in Figure 3.1b.

3.2 User Interface

The way a user interacts with the gamepad is important to take into consideration during design. As every user might hold the device slightly differently and there might be a difference in the way right and left handed people hold the device. The gamepad should also make it clear, at a



(a) The Credit Card design

(b) The nunchuck design

Figure 3.1: The two prototype designs

glance, what the current key binding is and the team the gamepad is assigned to. To accomplish the mentioned goals a list of user interface components was determined and aspects of it will be tested as will be explained in later chapters.

3.2.1 Requirements

The Requirements for the prototype user interface are a button to register a key press, a way to switch the device on and off, an indication of the device's key bind, a charging port, and a charging status indication. Additionally the reconfiguration from the gamepad itself was taken as one optional requirement to address as the added functionality would be rather significant.

3.2.2 Overview and Design Considerations

An overview of all the user interface components and a quick summary of how each will function will be discussed. For each part, the design considerations will be explained.

Wake up

The BUG consumes a significant amount of power when active. To reduce the power consumption when the device is not in use several options were considered. The first option would be to have a physical power switch that disconnects the power source from the rest of the circuit. The second option would be to set the ESP chip to deep sleep and wake up the BUG with a button press. As the preference was to keep the user interface as simple as possible, adding the wake up system to an existing button was preferred. As the configuration button isn't in use during gaming, it would be easier to add the extra functionality to the configuration button. This also avoids accidental triggering of deep sleep during gaming which might happen if the main button was used for the functionality.

Configuration button

The Configuration button will allow the user to set the gamepad to deep sleep mode and wake it up again with a short press. A long press of the Configuration button will reset the gamepad

to factory settings. In factory settings the gamepad has no Bluetooth connections, a default keybinding, a team assignment, and if applicable a right handed configuration. Furthermore, the configuration button will cycle through a number of default keys for easy reconfiguration to the most used keys when the device is turned on and not set to a custom key (see section 4.5).

The Configuration button will have to be in a location on the gamepad which is hard to accidentally touch during common use. This is important to prevent the user to accidently shut down the gamepad during use. For the nunchuck design, the best location meeting this criteria would be on the flat underside, point B in Figure 3.1b. This location is highly unlikely to be touched by the user during use. For the credit card design, the possible locations would be on the front facing in the centre or potentially on one of the sides. As it is important to avoid accidental presses of the button the sides might not be ideal as some users might hold the device touching some of the sides. Thus the proposed location would be centre front facing. Furthermore the button can be mounted flush with the casing to make it more difficult to accidentally press.

The Button

The gamepad's main button will, when the device is paired, send the bound keystroke to the paired device over the Bluetooth connection. The Button is the main input for the gamepad thus it has to be relatively easy to recognise and press. To this end the possibilities are to have it slightly raised and to have the button in a different colour than the gamepad casing. Further the location of the button is important. For the nunchuck design, the button on the initial prototype has four possible locations, as indicated by the numbers in Figure 3.1b. The overall preferred location would be determined during testing. For the credit card design, there is really only one logical location for The Button which is on the front facing as in the prototype design see the circular debossed section in Figure 3.1a.

Key LEDs

The LED array on the gamepad will be used to indicate the current key binding and the current assigned team. This will be done by lighting up the LED(s) corresponding to the bound key and the colour of the LED(s) will indicate the assigned team. The LED array can also be used to indicate the gamepad is in pairing mode by pulsing blue.

The key LED array has two possible designs to indicate the gamepad current key binding. The LED array could be either in the shape of the keyboard arrow keys or a plus shape. Both possibilities have advantages and disadvantages. The arrow key layout clearly shows which side is up but could be slightly more restricting in indicating possible keys. Contrary the plus shape is less clear in what is up but has slightly more flexibility in indicating keys. The LEDs will also be able to indicate the team assignment by using different colours, and by blinking blue they can indicate pairing mode. The decision on which layout will be used will be done using testing data on user preference which will be further explained in section 3.5.

The location for the LEDs has essentially one conceivable spot on the credit card design which is on the front facing as seen in Figure 3.1a. The nunchuck design might have some more options for the LEDs depending on the size, theoretically the best options would probably be A or D in Figure 3.1b. During testing peoples preferences regarding the location will be gathered as well.

Charging port

A port will be present on the device to recharge the internal battery unit so the user doesn't have to deal with replacing batteries. For the charging port several options are possible, both for location and the physical port. For location on the Nunchuck model the most logical place where it won't interfere with user actions is the bottom at point B in Figure 3.1b. This is also an intuitive place as most devices have the charging port at the bottom. Location-wise the same location is logical for the credit card design which would be the short side near the LEDs in Figure 3.1a.

Regarding the options for the physical port the most common two options would be micro-USB or USB-C as these are most universally used on smaller devices. For the prototype a micro-USB port is used as this is what was present on the devkit. For location regardless off which design is decided upon the most logical and intuitive place would be next to the charging port as this is common for most devices that use a light to indicate charging status.

Charging LED

A separate LED will be used to indicate the charging state of the gamepad. The charging LED will burn red when the device is charging and green when the device is done charging. The best way to do this would be an LED with multiple colour options, ideally these colours would also be easily distinguishable by colour blind people. In this case a red-blue charging-done indicator would be great. However the more intuitive case for charging status indicators is red-green, as most humans associate red with stop and green with go. Thus depending on availability the status LED might be red-green.

The most logical location for the charging LED would be near the charging port.

3.3 Electronics

For the product, both a MCU and a Bluetooth module is needed. This can be done with a separate MCU (such as a Arduino-like IC) and a separate Bluetooth module, but also with an integrated solution, such as an ESP32. BLE is preferred over classic Bluetooth due to power considerations, as BLE is more efficient. BLE 5 is preferred over BLE 4 due to it being slightly more power efficient but also to use the newest version, so the BUG will be more future proof.

3.3.1 MCU with separate Bluetooth module

For this prototype, a SAMD21[1] MCU is used since it is easily programmable and has enough power for the initial, rough prototyping. As Bluetooth module, the BT836B module is used: this module is compatible with Bluetooth version 5 and is controllable with AT commands[2].

It was found out quickly that controlling the Bluetooth module is very challenging using the raw AT commands. Configuring the module has been done, but using it and sending the proper commands seemed hard, mainly because the documentation wasn't as clear as it should have been. This option was the more expensive option (the MCU and Bluetooth module are €12,50 combined), and a serial connection is needed between the MCU and Bluetooth module. This conversion of data to an UART protocol is most likely to be less fast than an integrated solution within an IC. Therefore, it was decided to suspend further efforts in making this a possible solution.

3.3.2 Raspberry Pi

Raspberry Pi (RPi) would be overkill since things such as ethernet, (mini)HDMI and an SD card would all be unnecessary. Besides that, the RPi zero, the smallest of the RPi's, has a size of 65mm x 30mm which could barely fit in the controllers. This would not leave enough space for extra electronics such as buttons and a battery. A RPi only comes in development kits, so making a custom PCB for added features will only make it even bigger. This means that it is not an option to add or remove functionalities.

[1] The SAMD21 is a variant of the more known Arduino MCU.
[2] The AT command set (or the Hayes command set) is a low level command set used mainly for communication with modems.[6]

3.3.3 ESP32-C3 (MCU with integrated Bluetooth)

A big advantage of an integrated solution, is that communication between the MCU and the Bluetooth module is completely within the IC: the fastest way possible. Since this is universal, libraries for this integration exist and can be used right away: this frees the developer from writing a Bluetooth driver. Prototyping was done using a so called development kit, a circuit board with this ESP32-C3 module pre-soldered with all the needed peripherals. After a short amount of time, a proof of concept was achieved (Bluetooth communication, sending key strokes to a PC), making this solution very appealing. Combined with the cost of this module (the MCU with flash and the on board antenna), which is only €2,50 per module, makes this pretty much the best possible solution in this case.

3.4 Software

In this section, an overview for the software written for the prototype will be discussed, and how it interacts with a PC.

3.4.1 Requirements

The software would have to meet several requirements. Mainly the prototype would have to be able to connect to a PC over Bluetooth and send a programmed keystroke to the PC. It should also be able to show which key binding is currently active using the WS2812 LEDs on the prototype.

3.4.2 Program

The main requirement is to have the ESP development kit be able to connect to a PC and send a keystroke. To achieve this goal a ESP library was found on Github that allows the ESP32 chip to act as a Bluetooth keyboard. [7] This would allow for the BUG to send a keystroke when the button is pressed. An example provided by this library with a similar concept was found which provided some code to form a clearer understanding of how this library worked.

For the second requirement a library made by the manufacturers of the WS2812 LEDs was used to allow for easy control of the LEDs with intuitive functions. [8] These functions allow for each LED to be set to a specific colour, to set an overall brightness, clear all the LEDs, and to set all the LEDs to one colour.

These libraries together provided what was needed for the prototype to function as desired. The final code combining the libraries power consisted of several parts. First the libraries are included so they can be used in the code, next the name is set under which the BUG can be found over Bluetooth. Following that several definitions are done, and some character initialisations. After this the WS2812 are initialised so they can be called with a name and functions can be used to address them. Consequently the setup code which runs every time the BUG starts. In the setup code the Bluetooth is started, the pixels are started as well, and the button was set as a pulldown button thus being connected to ground as default.

3.5 Testing

Both prototypes were tested for both functionality of the designs but also to acquire potential user data/input.

3.5.1 Functionality

The first testing was to determine whether the prototypes functioned as expected. Only one of the prototypes had the development kit implemented with the Bluetooth connectivity. Thus

the Bluetooth transmission could only be tested with the credit card prototype. The nunchuck design had a MCU implemented which could be connected over a wired serial connection to a PC. From the serial connection the button presses could be emulated as a keyboard press and logged. For initial testing both gamepads were used to play the Google T-rex game[3]. Both gamepads functioned well and no real problems were encountered during this testing. Some initial data was collected by the project team on the preferred buttons on the nunchuck. The initial data on the preferred button showed a strong preference for button 4/the top button.

3.5.2 Potential user testing

The two possible designs for the gamepad and multiple options for the user interface components required outside input as to make less biased decisions. The project team could be biased towards a possible design if it is easier to develop for. To gather potential users opinions some small scale tests were executed. Though for less biased results a larger sample group would have been better due to time constraints a small sample group was used. From the sample group the button press data was collected for the nunchuck and a small survey was done as will be explained further in the next section. The test subjects were found by contacting friends and family and stopping by the ETV (Elektrotechnische Vereeniging) the study association of the Electrical Engineering program at the TU Delft.

3.5.3 Survey and Data acquisition

In this section an overview of the testing procedure and survey and data acquisition.

Testing procedure

The testing was done by having two subjects cooperatively playing a game of Pong against the computer. In this cooperative game of Pong both players controlled one direction of the bar, one user up one user down. To stop users from thinking to much about what button they want to use questions from the survey were asked during playing so the button use becomes subconscious. After about 5 min of playing the users switched gamepad and thus direction control. This allowed the users to try both gamepads and thus give their preference in the survey.

Data acquisition

The button press data logged by the laptop was stored for every two test subjects as this way the users could smoothly keep playing. During testing the time was manually logged when the users switched to be able to separate the key logging by test subject during data analysis. One problem was encountered during testing when a subject accidentally unplugged the gamepad which resulted in the loss of the testing data for that session. Following this incident the data acquisition code was altered to also save data if the serial connection was broken physically. The Python code for data acquisition and key stroke emulation can be seen in section B.1.

3.5.4 Data analysis

The data that was collected during the small scale tests had to be analysed to see whether the predictions were confirmed. To this end the recorded key presses were imported into Excel. The data sets were first split so each subjects data could be individually analysed. Consequently the data was split into 20 second sections and the number of presses per section for each button were determined. The resulting timeline of button presses could be plotted as a stacked graph to show which buttons were pressed in each time slot and how the preferred button might have changed over time. The data and graphs can be seen in Appendix C. After looking at the results individually, an average of button presses per time interval was made for each button for each

[3]A very simple game needing only the space key to be played. See https://www.trex-game.skipser.com/.

user. These averages were added up to create a total of the averages of button presses for each button per time interval. The resulting values for each button were turned into a pie chart to give an overview of the most used button on average. The resulting pie chart with percentages can be seen in Figure 3.2. As evident from the figure button 4 is the most used button during the small scale test, however it is worth noting that button 2 and 3 which are in a similar location on the gamepad together have a larger usage.

3.5.5 *Survey analysis*

The survey was done through Google Forms which automatically creates an overview of the answer data including pie charts where possible. The full results can be seen in Appendix C. The most interesting data from this survey was the preferred gamepad option of the test subjects and the preferred button on the gamepad. The overall preference can be clearly seen in Figure 3.3a as two thirds of people preferred the nunchuck design. Though only a small sample size was used this does confirm the suspicions that the more ergonomic design of the nunchuck design would be favoured by users. Further the results of the button preference are interesting, as can be seen from Figure 3.3b the subjects indicated preference matches the button usage quite closely. The button preference also shows a similar division in that button 2 and 3 together have about the same share as button 4. It could potentially be interesting to see if aspects like handedness or age effect the gamepad preference or button preference however with such a small sample size it would not have any statistical relevance. Thus the decision was made not to do this analysis.

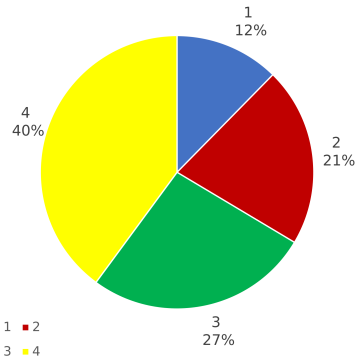
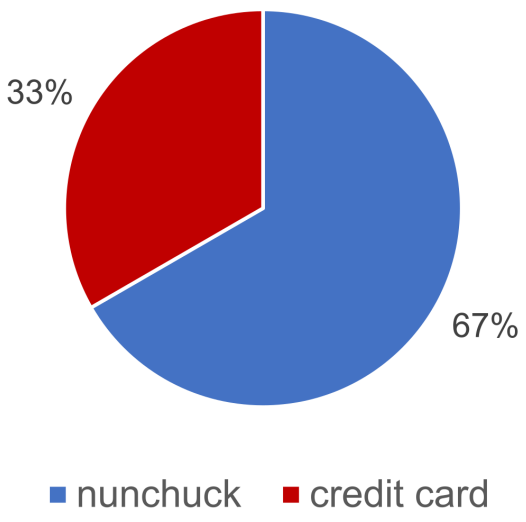
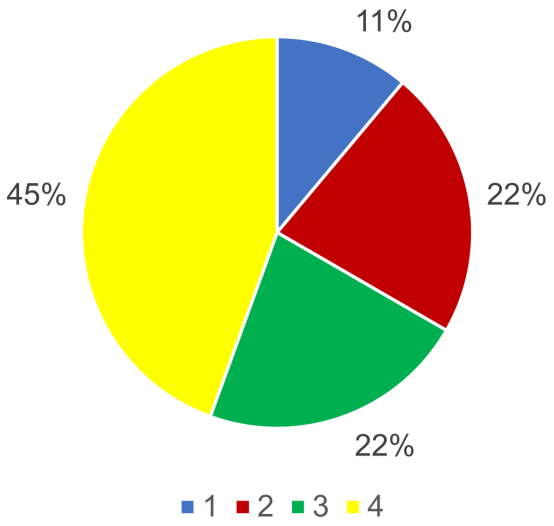


Figure 3.2: Overview of average button usage

The questions regarding the key bind LEDs are also interesting to quickly consider. Regarding the possible layouts, the arrow key style layout received the majority but only just, see Appendix C. Furthermore, the small sample size means this cannot be considered conclusive. Finally the LED location for the nunchuck is interesting, location A in Figure 3.1b was suggested 3 times, location D 5 times, and C 1 time.



(a) The test subjects gamepad preference



(b) The test subjects button preference

Figure 3.3: Survey results

3.5.6 Conclusions

The initial small scale test confirmed the predictions that the nunchuck is indeed favoured over the credit card, and the preferred button is the top button/number 4. Due to time constraints this was only a very small sample size thus the results are not very conclusive. However the preferences were quite significant differences thus it was taken as significant enough to move forward with the nunchuck design.

Chapter 4

Final Design

In his chapter the stages of the final design will be discussed. The design process has been split in several sections: User Interface, 3D Model, Electronics, and Software. The design processes of each of these sections were done simultaneously and not consecutively thus there is a decent amount of overlap in the sections and a large amount of inter-sectional referencing.

4.1 3D Model

Using the results discussed in subsection 3.5.6, the final design is based on the nunchuck model. The steps for the 3D model itself are discussed in Appendix E. Even though the buttons on the front of the initial prototype are evenly popular when joint as the location on the top, it is decided to not use the front to make sure the Bluetooth antenna can be shifted forward as much as possible. This is done so minimal interference with the Bluetooth signal is experienced when using the BUG. This makes it impossible to position a button in the front of the model. The indication LEDs are positioned on the side, since they must be mounted flat on the circuit board. It was mentioned by a decent amount of people testing the prototype this was a desirable location and it is the cleanest method of integrating them into the design. Worth mentioning is that this location is non-ideal for left handed people, but since the indication is not needed during the time the BUG is used, this is considered acceptable. The arrow layout (opposed to the plus layout) is chosen, since it was found the most people preferred this and it fits the gamepad model better. The printed circuit board is placed vertical to make sure all components can be fitted directly to a single board. The printed circuit board is offset to one side to make sure the push buttons are centred in the model. This immediately ensures a lithium polymer battery can be fitted inside. As for the indication LEDs on the side of the model, a small two millimetre thick square of EVA material (hot glue) is added to diffuse the indicator lights. This material is cheap, easy to work with and diffuses the light nicely.

All these decisions are made with a PCB design and orientation in mind. With the decisions above, a vertical placed PCB, as shown in Figure 4.1 could be used to make all buttons, LEDs and other electronics accessible. The main downside is the USB-C connection: it needs to be an angled connection which is difficult to find and slightly more expensive. In section 4.2, it is discussed why USB-C is chosen as power supply. The Li-Po battery is shown in Figure 4.1, but is positioned behind the PCB and thus not clearly visible.

The main shape of the BUG is made with the prototype nunchuck model in mind: it is not in the scope of this project to make the most ergonomic gamepad possible, but a usable, somewhat comfortable gamepad to house the electronics. The model is made with computer aided design software (CAD), creating a rough shape and rounding the edges. A test model was developed using a 3D printer and the ergonomic features were subjectively tested. Inside the 3D model,

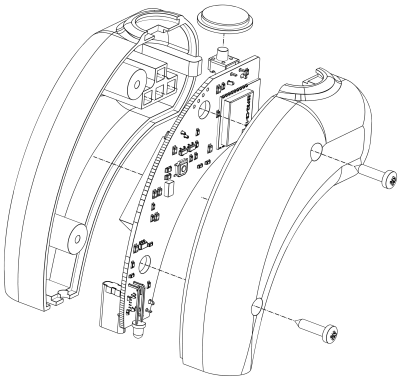


Figure 4.1: Exploded view of the 3D model of the BUG, making the placement and shape of the printed circuit board on the inside clearly visible.

a shape for the printed circuit board is defined and exported to the PCB software for further development.

The model will be closed with a ledge in front to secure the front. This is done because the first models fabricated showed some movement at the front of the case, which is unwanted behaviour of the gamepad. Two screws are used to fit the two halves together, as shown in Figure 4.1. It could be done with only ledges, but this makes debugging a lot more difficult. In a final model (possibly manufactured using injection moulding), ledges could be used to cut some costs, but the use of screws does improve the ability to repair. The ledge in front had to be quite robust, since it was prone to breaking due to the nature of 3D printing[1] and would be solved when using injection moulding.

4.2 User Interface

In this section the final decided User Interface components will be explained, mostly the choices were based on the testing done with the initial prototype design.

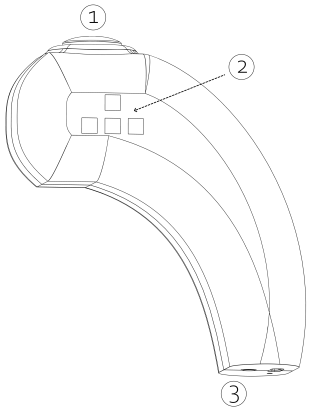


Figure 4.2: Side on view of the BUG model with numbers

4.2.1 Requirements

The requirements the user interface part has to comply with consist of the main interaction button, the method for switching of the device, rechargeability, and the current keybinding indication. As an additional requirement the possibility of PC software for reconfiguration of the BUGs is part of user interface.

4.2.2 User Interface Design

In this section a quick overview will be given of the User interface components as previously described in section 3.2 and the final design decisions made based on the testing with the prototypes.

[1] Due to the way 3D printing works by placing layers onto each other, sections consisting off small surface area are more likely to break in the direction off the layers, thus two layers snapping apart.

Configuration button

As mentioned in section 3.2 a logical and intuitive location for the configuration key would be the underside and the survey showed similar thoughts by the test subjects. Thus the choice was made to place the configuration key on the underside of the device at number 3 in Figure 4.2. As for the functionality the suggested functionality as discussed in section 3.2 was implemented. This allows the configuration key to turn the BUG on and off, cycle the keybinding, and set the BUG to factory settings. In Figure 4.3 an view from the underside can be seen showing the configuration button.

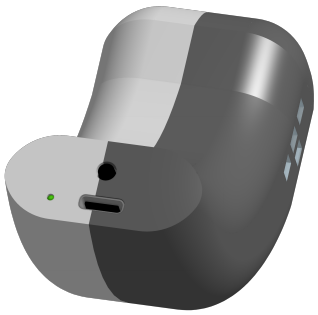


Figure 4.3: Bottom view of the nunchuck model

The Button

The button location was determined during testing with the initial prototype as explained in subsection 3.5.6. The final location for the button was chosen to be the top, location 1 in Figure 4.2, as this was the most preferred. Functionality wise the button lets the ESP know it needs to send a keystroke as is desired. The button can be seen in red in Figure 4.4.

Key LEDs

The Key LEDs were placed on the side, as the chosen design is the nun-chuck design. During the survey, it was found this is a suitable and intuitive position. Furthermore due to the orientation of the PCB not many other places were possible. It was decided to use the left-hand side for the LEDs as this would be visible for most people using it as when the BUG is held with the right hand only the right side is covered. This corresponds to location 2 in Figure 4.2. As the majority of the human population is right handed [9] and the LEDs can still be relatively easily viewed when holding the device left-handed, this seemed like the best option. The LEDs can be seen in Figure 4.4 as the transparent light blue windows.

Figure 4.4: Side top view of the nunchuck model

Charging port

To meet the requirement of rechargeability a charging port is required and as mentioned in section 3.2 the most logical place would be the bottom. With the charging port located at the bottom of the BUG the user can also continue to use the BUG while charging. Thus the charging port would be located next to the configuration switch as can be seen in Figure 4.3. Regarding what type of port the decision was made to go with USB-C as more and more devices are moving toward using USB-C and the EU is hoping to mandate the use of USB-C on smaller electronic device[10]. The new mandate might not apply to the BUGs but by using the universal standard the device becomes more user friendly as user won't require a separate charging cable for the BUG.

Charging LED

The charging status LED is placed next to the charging port as this is an intuitive place to indicate the charging port. The charging LED can be seen in Figure 4.3 to the left of the charging port. During component research the availability of 2in1 red-blue LEDs was not within the budget so the choice was made to go for red-green 2in1 LED this will be further explained in section 4.3.3. The red-green LED will unfortunately not be ideal for colourblind people however a different solution with 2 more easily distinguishable colours could be developed but would be outside the scope of this project.

PC software

The PC software could be partially developed as there was time during the project. The PC software would allow a user to individually change the keybinding, the LED layout, the LED colour, the operation mode, and the sleep timer through a GUI. This will be further explained in section 4.5.

4.3 Electronics

4.3.1 Requirements

In order to design something, one needs to set out what the designed hardware should do. This list of requirements are the following for the BUG:

1. The BUG needs to be safe
2. A button press needs to be detected
3. Communication should be possible using a Bluetooth connection
4. The BUG needs to be able to power itself
5. The BUG needs to be able to recharge without disconnecting any components
6. The BUG can enter and exit a low power mode or the battery can be disconnected without opening up the BUG.
7. The BUG can indicate which keyboard button it is configured to

There are features which can be nice to have, but are not essential for the easy and comfortable usage of the BUG. These are not stated under the Requirements, but will be discussed later on.

4.3.2 Architectural design

With these requirements an architectural design can be made. This means that the entire electronics from this project will be split into blocks with certain tasks. These blocks have a well understood and concrete connection with each other, while the tasks they need to perform are well defined. This allows for the blocks to be easily designable within the project. The initial architecture design can be seen in Figure 4.5.

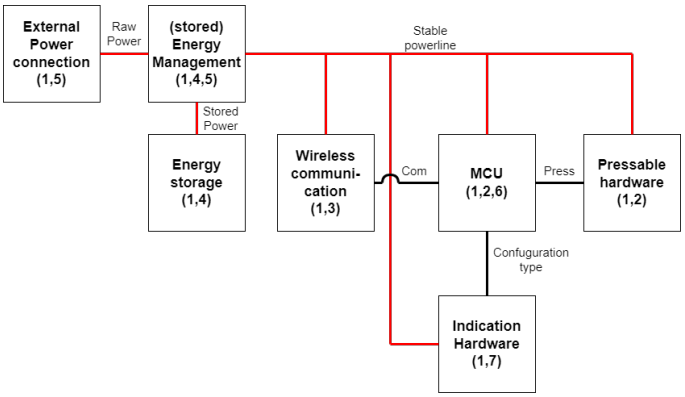


Figure 4.5: The initial Architecture design.

The most important block of the BUG is the micro controller unit (MCU). The MCU will make sure the software can be implemented in the BUG and communicate with the hardware. It reads the state of the buttons (Pressable hardware), controls the indication hardware and Bluetooth

connection. The low-level Bluetooth protocol itself is handled by the dedicated hardware inside the Wireless communication-block.

Another aspect is the power, managed by the Energy management-block. This block connects to the energy storage (a battery, for example) and makes sure this storage is properly charged. It takes an input (the external power connector) and generates a steady voltage output for the rest of the system to work.

4.3.3 Schematics

The block diagram will be further developed in this section about schematics. This means that the block will be worked out and discussed per block. The entire overview of the schematics can be seen in Appendix J.

Wireless communication

When sending data wirelessly, an antenna is an essential component. There are three widely used types of antenna. The first type is the whip antenna. Whip antennas are the type of antennas you would typically see on top of a car. They are rods which extend a certain distance and are generally quite flexible and long. These antennas have a good performance[11] and take up little space on a PCB[12]. They can also be easily replaced since they have a connector connected to the PCB. This reason, combined with the fact that they are already impedance matched, makes them a plug-and-play kind of antenna[12]. Downsides are their size and their cost. Since these antennas are fairly long, they might not even fit in our application. Performance is the main advantage, but since we do not need the best performance (only a range of about 3m) this option will probably be overkill.

Figure 4.6: Two examples of whip antennas.

Chip antennas are made in a package like any other passive component on a PCB. Most of them are already matched to 50 Ohm for a certain frequency, but some extra tuning is often required[11]. These chip antennas are also able to be replaced without ordering an entirely new PCB, simply by reheating the solder and pulling them off or using other tools which are able to do that. Although this is not as easy as unplugging a whip antenna, it is still possible when needed. One of the main advantages is the space it takes up on a PCB. Although a clearance area is required, this one is relatively small compared to other on-PCB applications.

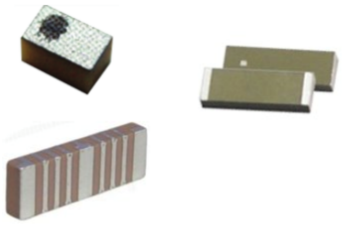


Figure 4.7: Three examples of chip antennas[12]

A PCB or trace Antenna is an antenna created by creating a trace on a PCB which has the shape and functionality of an antenna. This has to be done at all layers and cannot have conductive material near it. This means that there is very little assembly when producing this antenna. This also makes this antenna practically free, since the costs will be in the PCB production cost[12]. When producing the PCB, it is made of copper most often and it is set in "stone" when ordering the PCB. This means that very little tweaking can be done once the PCB has been produced. This means that per setup, you require a new PCB for side-by-side comparisons, making testing more expensive[12]. Because somebody is basically designing their own antenna from scratch, one should possess a lot of knowledge[1] or simulation beforehand about viable options[11]. Although there are some pre-made setups, adjusting these

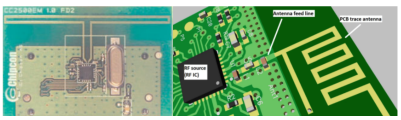


Figure 4.8: Two examples of a trace antenna[11][12]

setups to fit your needs will still require knowledge to make an educated adjustment. In section 4.3.3 the decision for the antenna will be made

MCU

As discussed in section 3.3, the ESP MCU was chosen. Due to the reasons mentioned in the section, the software already written and the experience obtained with this MCU, the decision was made to continue working with the ESP as an MCU.
Another consideration is if a ESP32-C3 chip, module or Dev kit is used. A chip is just the ESP32 C3 core, without flash memory or any other peripherals to make the MCU work. This allows a lot of freedom in PCB layout, specifications and reducing redundant hardware. This comes at the cost of effort. More research needs to be done into antennas, flash memory and computer architecture.

The use of a module makes the design easier. Most modules come with a trace antenna with a matched network, flash memory, crystal oscillator, decoupling capacitors and other protection hardware. This allows the user to focus on making the functional product. This does come at the cost of interchangeability. If the modules do not have enough flash memory, it is very difficult to desolder the shielding, the flash IC and put the new components back on without damaging any of the components. If the antenna is not as functional as one might want, a new one can be connected to most modules, but the trace antenna's space will be wasted. One of the main things one still has to do is make a power supply for the module. The module works on 3.3V, while USB works with 5V and a LiPo battery will work with 3.7V.

Lastly, a dev kit will be the least interchangeable, while having the most subsystems in place. Besides the components discussed in the module, a Devkit most often brings micro usb connectability, an LDO which allows the board to be powered by USB, some reset and boot buttons, LEDs, USB serial converter and a pinout which is breadboard compatible. These extra capabilities come with the cost of a lot of extra space, more power consumption and even less interchangeability.

Since the Dev kit has a lot of capabilities which are not going to be used, while taking a lot of space which can be better used for a battery, the Dev kit was used in the prototype which most often will be fully charged or powered by cable. The chips will be a lot of work and time to implement and mainly test, since a custom antenna should be used. Debugging this antenna may require new printed circuit board designs, which is not possible within the time frame of this research. That is why the decision was made to start with implementing a module on a PCB and, if there is enough time, the chip might also be used to save on space and power.

In terms of modules, the requirements were an onboard antenna, 4MB of Flash memory and at least 4 GPIO pins and 2 ADC channels. With these requirements, 2 modules were considered. The ESP32-C3 12F and the ESP32-C3 13. The differences between these are mainly: Both modules have an antenna on top, pins on the sides, but the 12F also has pins on the bottom and 4 NC (no connection) pins[13]. The 13 has no pins on the bottom, making routing easier, but soldering harder, since the pins are closer to each other (from 2mm distance in the 12F to 1.5mm in the 13)[14]. Another difference is the size. The 12F is a total size of 16mm x 24mm or $384mm^2$. The 13 is 18mm x 20mm or $360mm^2$.

In the end, the ESP32-C3 13 has been chosen as MCU since the MCU will be mounted in the front, with little space available on the sides of the MCU, making routing the PCB nearly impossible if there can be no traces below the MCU. This also means that a trace antenna will be used.

Energy storage

Energy can be stored in multiple ways, but batteries are most common is user appliances. Since the BUG should be rechargeable, NiMH or LiPo batteries are possibilities. Since the limited size available inside the gamepad, a small battery is needed with a decent capacity to be able to

use the BUG for a certain amount of time. Since, as will be discussed in section 4.3.3, a lot of integrated circuit solutions are available for LiPo-battery charging, a LiPo battery is chosen. With the available size inside the gamepad of 35x17x15mm (l x b x h), a 250mAh battery can be used. The LiPo battery has as advantage that is supplies 3.4V (empty) to 4.2V (fully charged), which is all above the minimum voltage for the MCU.
Besides satisfying these requirements, does the battery also have safety protection. The battery has integrated over current, over- and undervoltage prevention build into the LiPo[15]. All this functionality is on the small PCB called the PCM.

Energy management

The energy management is responsible for charging the LiPo battery and providing a steady 3.3 volt to the MCU. The charger should allow an input voltage of 5V, since it will be powered by an USB connector). To safely charge a LiPo battery, it should first be charged with a constant current. When the LiPo is almost finished charging, the charging should swap to be constant voltage. This is usually achieved by a dedicated battery management IC. Three possible power paths are possible, with direct battery supply as the most simple solution and more complex, bypass solutions which don't depend on the battery voltage to work [16].

Since the MCU needs 3.3 volts and a LiPo should at least supply 3.4 volts, the LiPo voltage should always be above MCU voltage. This makes the direct battery supply possible and, since it uses the least components, the cheapest. An IC capable of these restrictions is the MCP73832. This is a simple IC which can be programmed with a resistor with what current the battery should be charged. It also provides a logic output whether the battery is currently charging or not. However, this particular IC is not available at JLCPCB, which company is used to fabricate and assemble the circuit boards for this project. A different IC, which is pin compatible and provides the same function, is found with the name TP4054 [17].

The logic output is either low when not charging or high impedance when charging. With this output pulled up to the USB power, a LED could be used to indicate if the device is currently charging only when the USB power is connected. The easy way of doing this is by adding a single LED which indicates if the device is charging and is turned off when the device is ready. However, it is desirable to be able to differentiate between done charging and not charging. To achieve this, both state of the output of the TP4054 should be detected. By adding two additional resistors and a NPN bipolar transistor, a bipolar LED can be used to indicate with two colours if the device is charging (red) or done (green). A schematic of this circuit can be found in Figure 4.9. The pull up resistors (R5 and R6) are also used as the current limiting resistors for the LED. During testing it was found these resistances were too high to have a well lit LED, so they are swapped for 680Ω resistors.

To provide the MCU a clean 3.3V power input, some sort of voltage regulation is needed. This can either be done using a buck down converter or a voltage regulator. Since the LiPo can be as low as 3.4V, a very low dropout voltage is needed. Since the MCU needs a fairly stable power input [14], a buck converter is not a valid solution, since it introduces a ripple effect. To generate a stable voltage for the MCU, a low dropout regulator (LDO) is chosen. Since the voltage difference between the input and output is dissipated inside the LDO, this introduces some power losses, but are considered acceptable since a wider range of the battery voltage output can be used. The LDO needs to convert an input between 3.4V and 4.2V to an output of 3.3V, so a minimal dropout of 0.1V is needed. Ultimately, an LDO with a dropout voltage of 0,09V is used: the HT7333 [18]. This IC is cheap, is able to provide enough current (up to 250mA) to the system and can handle all voltage ranges.

External Power Connection

In subsection 4.2.2, it is decided to use an USB-C connector. In section 4.1, it is decided to use an angled connector.

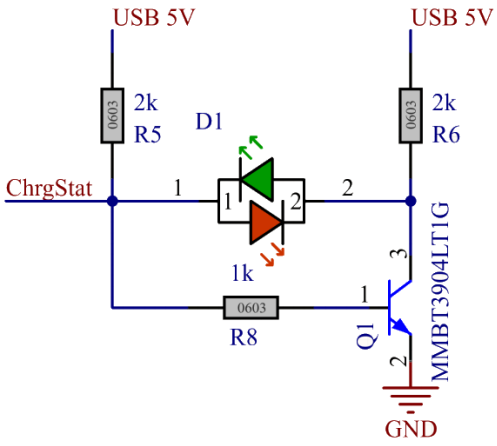


Figure 4.9: The circuit with the bidirectional LED indicating the charging status

Pressable Hardware

The cheapest and easiest to use option is a tactile button, a small button which is directly soldered to the PCB. Since in section 4.1 it is determined an angled button should be used, the button itself is already decided. For the main button, a simple low pass filter is used as a debouncing filter. Without it, the button can repeatedly 'bounce' between its pressed and released state during the transitions.

The same button is used as configuration button, discussed in subsection 4.2.2, but placed at the bottom of the gamepad. This button has various functions defined by the software, but has also two different possible outputs at the hardware level. When the BUG is powered on, a signal ("RDY") is set high and disables the feedback this button has to the enable-pin of the ESP32. When RDY is set low, this button momentarily pulls the enable-pin low when pressed and restarts the BUG. The schematic achieving this behaviour can be seen in Figure 4.10 and a simple simulation can be seen in section I.1.

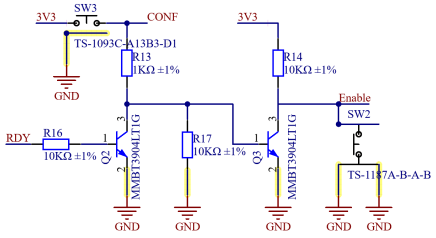


Figure 4.10: The schematics of the wake-up circuitry

Indication Hardware

Task of the Indication Hardware is to indicate to the user which configuration is active in a BUG. Due to positive feedback during the prototype testing, it is decided to use LEDs. To be able to differentiate between different teams, RGB LEDs are preferred. According to subsection 3.5.5, it is decided to use four LEDs in the arrow-keys layout. Conventional RGB leds will use 12 GPIO pins in this case. Another option would be to use WS2812-style LEDs, which require a serial input and can be daisy chained[2]. This requires only a single GPIO and simple routing, while also maintaining full RGB functionality. Each WS2812 LED trims the first 24 bits from the serial data stream and applies it to its (LED) output, and forwards the remainder of the data stream to its data out pin. An example of this can be seen in Figure 4.11. For this design, the WS2812 solution is chosen due to the MCU not having enough available GPIO pins for the conventional method.

[2]Daisy chaining: connecting multiple devices by connecting the output of one to the input of another device.

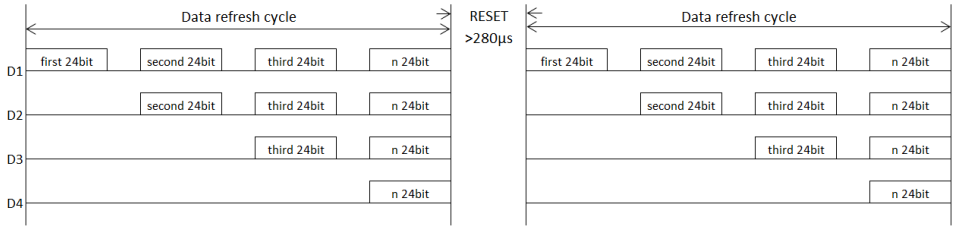
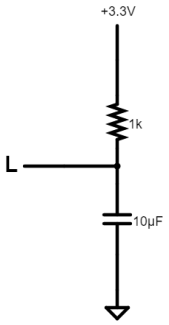


Figure 4.11: A abstract example of a datastream leaving a WS2812

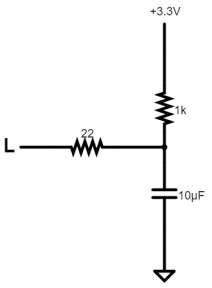
Miscellaneous Hardware

Beside the required hardware, there will be hardware present which is not meant for one of the requirements, but does make the experience for the testers or end-users better. One of the extra features is a voltage measurement of the battery voltage. It consists of a simple voltage divider with a capacitor added to create a buffer when reading the voltage level. The ESP has got an internal analogue-digital converter (ADC), which is used for this case.

Another option would be to set a standard starting button for the factory settings (see subsection 4.2.2) by a resistor divider and the second ADC. This ADC happens to be reserved for the wireless radio [14]. An RC-oscillator with different timing values is also a possibility, but this would require an additional current limiting resistor (22Ω in Figure 4.12b), which is not implemented on the PCB. During the research to this RC-oscillator, the PCB was already ordered as seen in Figure 4.12a so it was decided to handle this issue purely with the firmware of the BUG.



(a) The circuit made on the BUG PCB



(b) The circuit with the added current limiting resistor

Figure 4.12: The two options for ID voltage division

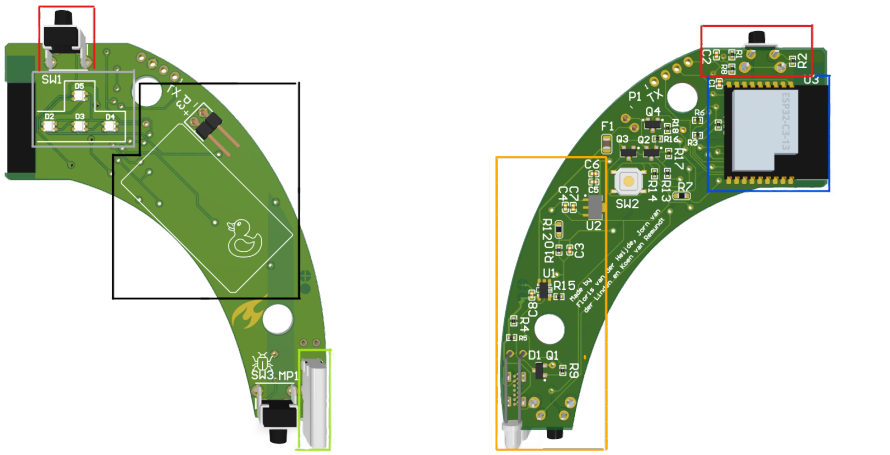
4.3.4 PCB design

When it comes to PCB design, a lot of the constraints to work with were already laid out in section 4.1. This consists of the board shape, the exact location of the MCU, the WS2812, the main button, the configuration button and the external power connector were already predetermined and these can be seen in Figure 4.13a and Figure 4.13b. The battery needed to be at the front, but no specific place was given.

The PCB was going to be machine produced and a large part of the basic components(ability to be machine soldered) were going to be soldered by machine. This meant that these basic components needed to be on the same side of the PCB. Since the battery, WS2812, two buttons and the USB-C connector were not basic components and they were all at the front, this left the back of the PCB as the only logical place to place these basic components.

With this knowledge, the most logical place to put the Energy management components in the orange outline seen in Figure 4.13b, since this was close to the external power connector.

After this, the place for the battery connector (P3) was selected at a place near the orange outline without taking up a lot of valuable space. That is why the current location was chosen and therefor also the place of the battery in the front. The last components still needing a place were the components for the wake up circuit. Since there is a large part of the PCB still free in between the MCU and the Energy managers, this place was chose for the wake up circuit.



(a) The PCB from the front. The red outline is where the hardware for the architectural block "Pressable Hardware" is present, while the grey outline coincides with the indication hardware, the black outline is where the "Energy storage" is and the lime outline(bottom right) is where the "External Power

(b) The PCB from the back. The red outline is where the hardware for the architectural block "Pressable Hardware" is present, the blue outline coincides with "MCU" and "Wireless Hardware" and the orange outline is where the "(stored) Energy Management is housed.

Figure 4.13: The front and back of the PCB.

4.4 Software - BUG

4.4.1 ESP general software

Process overview

The main software for the ESP went through several iterations. Initially the new iterations were build upon the prototype code added new functionality whenever desired. The first addition being the configuration key. This process led to a complex code which was very difficult to read.

The next big step was to split the code over several files making functions for all the interactions with the library. This significantly improved the flexibility of the code making it much easier to change functionality. It was however still executing a lot of functions repeatedly without any good reason.

The final iteration of the code was split over many headers so each collection of functions could be separated out to further improve legibility. Furthermore some additional functions allowed for a restructuring of the main code so functions were no longer executed unnecessarily.

Code overview

The final code is split over ten files: one main file and nine header files. For each part of the BUG that requires code to operate a header file was made containing the functions needed for that part.

The main code see subsection F.1.1 starts by setting some parameters. Firstly, the use of NIMBLE [7] which ensures more efficient storage of the files on the ESP. Secondly, the debug parameter which can enable serial output for all the functions for easier debugging. After this

the libraries mentioned in subsection 3.4.2 are included and the header files containing all the functions.

Next the setup is run which calls the ”initSystem()” function see section F.1.1. This function starts the serial connection if debug is active. It also starts the bleKeyboard library (section F.2) and neopixel library [8] so their functions can be used. After this the pins are declared according to the pin number variables in the variables header see section F.1.1. Next, the ”memory2lib()” function is called which takes all the variables stored in the memory and writes them to the library, see section F.1.1. Finally, the Identify flag is set to low and the ready pin set to high.

The main loop comes next which is split into two sections one for when the BUG is connected and when it is not connected see Figure 4.14. When the BUG enters the Connected state it enables the LEDs and resets the sleep-timer, see section F.1.1. After this it will enter a while loop which will run continuously as long as the BUG stays connected. During this loop it will check whether any of the events in Figure 4.14 occur and call the associated functions as can be seen in Appendix F. When the BUG is not connected it will enter the ”not connected” while loop where it will blink the LEDs blue, continuously check for activity on the configuration button, and check if the sleep-timer has expired. All the code and functions can be found in Appendix F.

Function/ state overview

The functions in Figure 4.14 each implement some desired functionality from the UI and requirements.

The ”BLE Char updated” function is to ensure library updates send from the PC are implemented immediately. This is done by updating the indication LEDs. As the rest will happen automatically because the other functions pull their information directly from the library. This fulfils the additional requirement for PC reconfiguration see section 2.3. It also helps towards the user friendly nature of the design as changes can be instantly seen, see section 2.1.

The ”Button pressed” function is called on a button press and tells the library to send a keystroke to the PC as long as the button is pressed. This fulfils the requirement for sending keystrokes to the PC.

The ”Conf pressed” and ”Conf released” functions implement the functionality for the configuration button as described in subsection 4.2.2. The key cycling is implemented using an array stored in the library which tells the BUG which directions should be present and whether space should be included. Finally, it tells the BUG whether W, A, S, and D keys are used or if arrow keys are used. This fulfils the mandatory requirement for turning the device on and off, see section 2.2. It also fulfils the additional requirement of allowing the key binding be changed on the BUG itself, see section 2.3.

Finally, the ”Sleeptimer exceeded” function stores all the library variables regarding keybinding, indication LEDs colour, indication LEDs layout, key cycling, and the sleeptimer duration. After all the variables have been stored the BUG is set to deep sleep. This helps towards the requirement of power consumption minimisation.

Battery function

One function that is interesting to discuss a bit more is the battery function. For this function several measurements were taken. First the ESP ADC was measured by connecting a programmable power supply to the battery header and slowly incrementing the supplied voltage while logging both the supplied voltage and the ADC output value. Using these values a formula for converting the ADC readout to battery voltage was made. The second measurement is the voltage curve of the battery taken from full to empty as will be further explained in subsection 5.1.4. The acquired voltage curve will be matched to battery percentages so a formula can be extracted to convert battery voltage to battery percentage.

The battery function starts by taking 20 readings from the ADC and averaging these, this is done to ensure that the effect of potential errors is minimal. Next the formula to convert to battery voltage is applied after which this is converted to battery percentage using the acquired formula. Finally the battery percentage is rounded to the closest 5 for a slightly neater readout.

The ADC on the ESP32C3 can vary from chip to chip [19] due to reference voltage deviation. As due to time constraints the measurement for the ADC to battery voltage transformation was done with only one BUG. This deviation might cause inaccuracies in the battery readings.

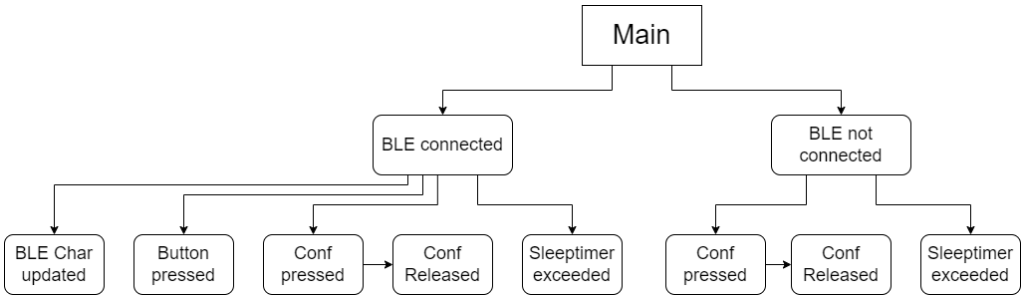


Figure 4.14: Overview of the main loop

4.4.2 Bluetooth communication on ESP

The ESP module must be capable of communicating as a Human Interface Device (HID) (a BLE keyboard), while ideally also accepting some sort of communication to make reconfiguration possible. This makes sure the BUG will work directly when connected to a PC and still have some method for reconfiguration. This can be implemented by creating a custom BLE service (see section 1.3) with multiple characteristics for specific parts of the BUG.

Since it was found in subsection 3.4.2 the BLE-Keyboard library [7] works as intended, it was used in the final implementation for the BLE connection. The library sets up the BLE Server with correct GATT services. The library is then altered to provide an extra GATT service for configuration of the BUG. Four characteristics are added inside this service:

- Key bind containing the char-code of the key bind to be send when the button is pressed.
- Layout containing both the colour of the indication leds and information which leds should be turned on.
- Sleeptimer containing the time in seconds before the BUG should be turned off due to inactivity.
- Mode containing various data. This characteristic can be used to reset the BUG, to identify the BUG and to set which mode it is currently functioning in.

A more comprehensive view of how the data is composed bitwise can be found in Appendix D.

The reveived data is interpret as raw data, thus looking at the binary data instead of the characters this data represents. This is done to minimise the amount of data to be send. The interpretation is done by the modified BLE Keyboard library.

All received data is saved internally inside the scope of the library, since this is more straightforward when dealing with receiving data over BLE. Multiple functions are created to set and read this data from the main program. Aside from the data itself, a flag can be set to notify the main program to restart the inactivity timer. This is done each time when a characteristic is updated using BLE.

The library is also stripped from unused functions to improve readability. To send a key press, the library takes the integer value from the local stored key bind, instead of an input from the main program. All variables are still readable and writable using functions from the main program, since this is needed to store the data to memory on shutdown and restore it on start up.

When one of the variables is updated, either internally or via the BLE connection, the new value is broadcast with the corresponding descriptor. This is done using exactly the same format as is used to set the variables using BLE. When doing this, it is possible for a PC interface to get the variables and display them in a GUI. The variables are always available for a client, even when they are set before the connection is established.

4.5 Software - PC

Although the need for a PC software is completely optional, it is found it could add a lot of functionality. A lot of games use, instead of the arrow keys, the keys W, A, S and D for directions. Making them all rotate with the configuration button would be sub-optimal. It could also be possible the user wants to use the BUG for a specific user case where custom key binds are needed. Instead of reprogramming the BUGs, reconfiguration is possible via the created BLE GATT-service described in subsection 4.4.2 and Appendix D. The PC software provides some graphical user interface (GUI) for controlling these variables instead of using some tool to view, send and receive custom data over this GATT profile.

For this software, multiple options were considered. At first, some communication between a code and the BUGs should be established. Windows is not well known for its compatibility with BLE devices, so a lot of problems emerged. For example, when a library was able to connect to the BUG, show its GATT services and read them, the HID protocol was disabled. This HID protocol is needed for the functionality for the BUG.

The PC software is partly implemented using Qt, a IDE with implemented GUI creator based on C++. Qt has got a working Bluetooth integration which makes it possible to read all GATT services from connected devices, while maintaining the HID profile. The PC software is made by editing an BLE example from the Qt resources [20], mainly because it was used to show a very detailed proof of concept. Many of the design choices of the GUI originated from this example or weren't thoroughly considered but based on intuitive choices.

Looking at the used code for the PC software, it can be clearly seen it is adapted from the used example. The code is based on functions controlled by the GUI and BLE communications. The program itself only connects to the service regarding the reconfiguration of the BUG. When opening the reconfiguration screen of a specific BUG, as shown in Figure 4.15, all relevant data is fetched and processed using the BLE connection. All characteristics (for example the key bind) can independently be set. This proof of concept shows it could also be possible to build a game entirely within this environment, where the controls and indication LEDs of independent BUGs could be adapted to different levels or rooms.

The current state of the PC Software is far from a finished package. It shows configuration is possible using a GUI, but still contains various bugs and workflow issues. A BUG can only be configured when selected the first time, re-selecting it will throw errors. The software also struggles with updating the configuration when multiple BUGs are connected simultaneously. For a final product, the GUI should be fixed or ideally be completely rewritten, making it possible to use the PC software for bulk pairing (and disconnecting) multiple devices at once.

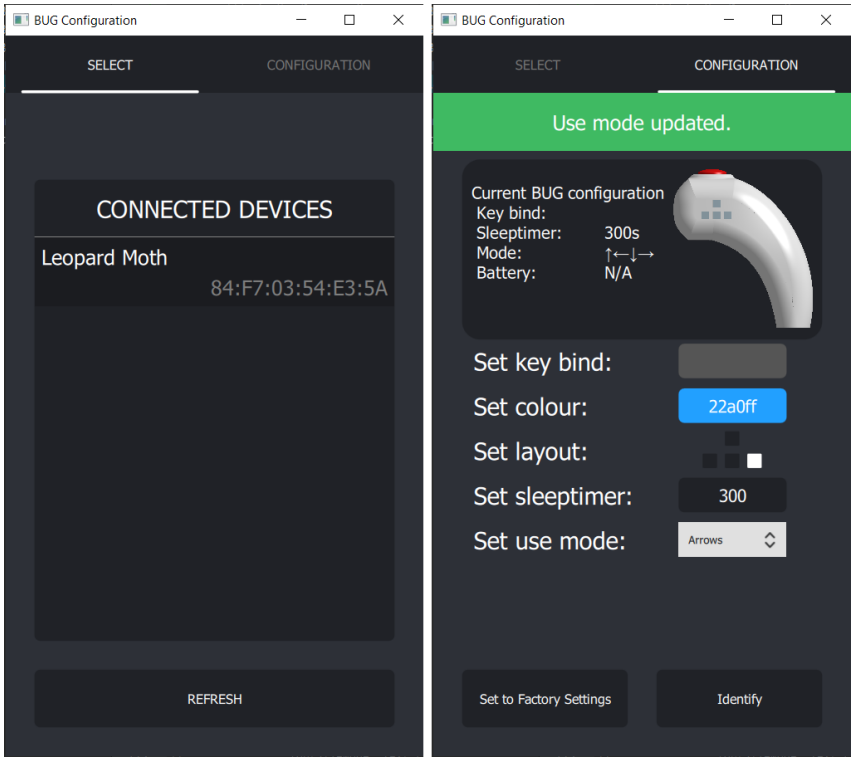


Figure 4.15: Screenshot of the GUI. Left the overview of connected BUGs can be seen (named ”Leopard Moth”). Right, the configuration screen can be seen of a specific BUG. Each characteristic can be set independently from this screen.

Chapter 5

Testing

The final step for the project is testing. Primarily the functionality of the final design needs to be tested. This final testing will include testing range, delay, charging, and more. Besides these more technical tests, user testing needs to be done and the users opinions will need to be gathered. During user testing the aim will also be to determine how many simultaneous connections operate smoothly.

5.1 Final system testing

Some small scale, low level testing has already been done. This consists of some questions like "Does it power on?" or "Can the BUG be seen by a Bluetooth device?". These are done by reversing our design and production process; if soldering was done right was the first test, then the functionality of the separate components was tested, and at last the functionality of blocks. The requirements set out in the beginning are the last to test. Not only is it tested if the requirements are met, but also to what extent. As an example, the BUG does connect wirelessly to a PC, but for what distance can it still have this functionality? Answers to these questions are going to be answered in the tests conducted in this chapter. The parameters (and the requirements) being tested are: Range, accuracy, delay, number of simultaneously functioning BUGs, and the time it takes to charge or discharge the BUGs.

5.1.1 Range

Plan

The range test will test at what range the BUGs Bluetooth connection disconnects. This will be done by connecting a BUG next to a PC, walking back slowly, one pace at a time, and checking whether the button presses are not detected on the PC anymore. If this is the case, the tester should stand still and check for a little longer if the Bluetooth connection will be disconnected. If this is not the case, the tester will walk back very slowly and check when the BUG starts flashing. This will give us 2 parameters to work with: The accurate range and the Bluetooth range: The accurate range is the range where the BUG is unlikely to miss a button press, while the Bluetooth range is the range at which the BUG starts to disconnect and search for a new Bluetooth device to connect to.

To ensure reliable data, ten functional BUGs are going to be tested 3 times. All the BUGs will be tested when they are just disconnected from the external power supply to ensure that all BUGs are in the same state of charge. Once a BUG is done with the first test, it will be disconnected and kept on until the BUG goes into deep sleep. Once this is the case and the next set of tests can be done.

This test will evaluate some of the project requirements, mainly whether the device works wirelessly and as part of user friendly as for party games some range will be desired.

Results

The results for the range test can be seen in full in section H.1. After all the results had been logged for each BUG 3 times, the averages per test were taken over all BUGs. The total average of these averages was taken as the average maximum range. The resulting average maximum range is 71.3 m. It is worth noting that this is the range BUGs on average disconnect so not a practical use max range. Furthermore it seemed that a delay started appearing in the transmission of the BUGs to the PC the further apart the two were. However this delay could also have been due to the way the observation of the keystroke was done.

5.1.2 Accuracy

Plan

The accuracy test is there to test how often a button press is correctly received on the PC when the button is pressed. This will be done by setting a couple of distances and pressing the button of a BUG 50 times. The key bind of the BUG will be set down arrow. By having the PC open on a spreadsheet it will be easy to tell how many times the down key was pressed and thus how many key presses were received. The distances of these tests depend on the outcome of the first test. The proposal is to test the BUG at 50%, 75%, 100% and 125% of the accuracy range obtained from the range test. There are several potential ways for the key presses to be received incorrectly. Firstly, some can just not arrive and there will be no key press registered. Secondly, a key press can arrive but the key release arrives either too slow or not at all, this results in too many key presses being registered. Finally, related to the first if the BUG disconnects during the 50 presses it won't register them all.

In case that at a certain distance the BUG won't connect at all the number of registered keys will be logged at zero. As the test is being done in the hallway of a building on some occasions people will walk in between the BUG and the PC. In such cases an attempt was made to either wait for the people to pass or the test would be redone if the results seem to be affected by it.

Results

The received key strokes per BUG per distance were logged and converted to a percentage of 50. For these percentages the Mean Square Error (MSE) was then calculated. The MSE then represents the number of Key presses that is not received correctly. The percentage of accurately received button presses is plotted against the range in Figure 5.1. The full results can be found in section H.2. As for gaming a very high accuracy would be desired, the functional range is within 40 metres. Since in most cases the user will be within this distance from the PC, this is acceptable. Finally it is important to note that most gaming won't occur in empty 100 metre hallways and thus this test is not necessarily the most accurate for real use situations.

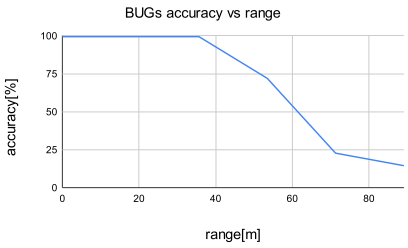


Figure 5.1: The accuracy of the BUG plotted against the range

5.1.3 Delay

Plan

To test the delay of the key press to a PC, the time between the key press and the detection of the key press must be measured. The key press itself is hard to register, but is approached by using the

serial output of a BUG. The received time for the serial message is compensated using the baud rate: the message was 25 characters (400 bits) long over a bit rate of 115200 bit/s. This means a standard delay for the serial message of 400/115200 = 3.47222 milliseconds. The timestamp of the received key press is also registered. This is achieved by a small Python script (section H.3) acting as a serial logger and key logger at the same time. The distance between the gamepad and the PC is constant and for each BUG, the button will be pressed 50 times. The python script will output the average delay from the BLE key press to the compensated serial input in seconds.

Results

The test of the delay was done for the 10 initially functional BUGs as the 11th has a damaged serial connection. The overall average delay of all the BUGs was 1,37 milliseconds. This is considerably less than the average human reaction time [21]. The full test results can be seen in section H.3. It is worth noting that this test was done very close to the computer and delay over greater distance might be larger.

5.1.4 Charging/Discharging

Plan

The BUGs are rechargeable and use a LiPo battery as their power source. This battery will discharge over time and will then need to be charged. This test aims to determine the average charge time from empty to full and discharge time from full to empty. Empty is defined as the point the battery can no longer supply enough power to the BUG. Full is defined as the point when the battery stops charging when connected to USB-C.

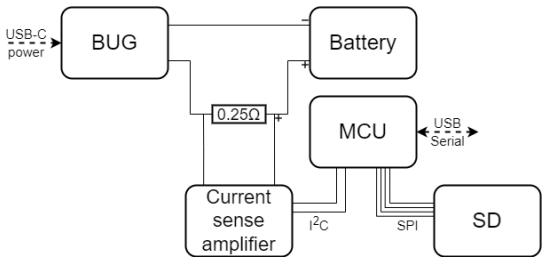


Figure 5.2: The accuracy of the BUG plotted against the range

Both tests were done with a testing circuit consisting of an MCU[1], a Current sense amplifier[2], and an micro SD port. This circuit was build on a breadboard, an overview can be seen in Figure 5.2. The circuit measures the current between the BUG and the battery which shows if the BUG is charging, discharging, on or off. It will also allow the easy detection of brown outs and whether the BUG is fully charged.

The charging test will be done by connecting the BUG to the test circuit as seen in Figure 5.2 and connecting the USB-C to the BUG to charge the battery. The test circuit will then log the current going to the battery and the voltage over the battery. The test will be started when the battery is empty and stopped when the battery is full. As the circuit also logs the timestamp for each value a clear charging timeline will be recorded thus giving a clear charge duration.

The discharging test will be done in a similar way to the charging test only without the USB-C charging cable connected to the BUG. The test will be done with a recently charged BUG and run from full to empty. Again using the logged timestamps a clear indication of discharge time will result. The BUG discharge will be tested while the BUG is on and connected as this will be it's most common state. The BUG does draw slightly more power when sending key presses but this effect is very small mostly because most button presses are very short. Thus the active and connected state will be seen as the average power consumption.

Results

[1] a Seeeduino XIAO
[2] an INA 219

The testing for the charge and discharge cycle was only done for one BUG as the test takes quite a long time to execute and only one test circuit was available. During testing some errors also occurred which unfortunately caused test date to be lost. In the end the cycle recorded was the end of one charge cycle, a full discharge cycle, and the majority of the next charge cycle. The full acquired graph can be seen in Figure 5.3. Looking at the graph we can see that the discharge cycle starts at about 01:52:00 and ends at about 04:16:00. This means that a discharge cycle takes about 2 hours and 21 minutes. Given that the start of the graph is essentially stable which suggests the battery was fully charged and that the end of the graph is almost at the same level. It is relatively safe to say that the charge cycle was nearly complete by the end of the measurement. If the duration is taken for the final charge cycle of the graph it is found to be 1 hour and 52 minutes, to be on the safe side another 30 minutes to one hour can be added to reach the same level as at the start. This would result in a charge time of about two and a half hours.

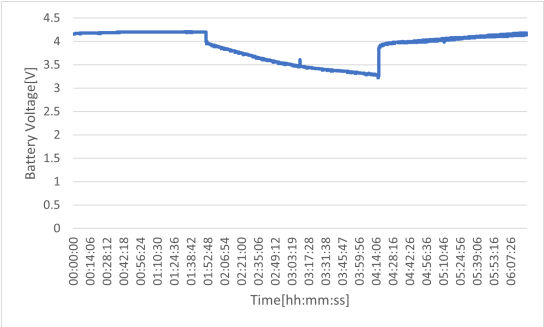


Figure 5.3: The battery voltage over time while charging and discharging

5.1.5 Simultaneous connections

Plan

The BUGs work over BLE 5.0 which theoretically has no limit for the amount of simultaneous connections. However, most hardware has some limit built in, or at least a limit of how many connections can communicate at the same time. It is rather difficult for many devices to find a clear indication of how many connections can be sustained simultaneously. It would be interesting to connect up to ten BUGs simultaneously so that to teams of five could play against each other on one machine. To test the possibility of ten simultaneous connections it was attempted to connect as many BUGs as possible to the PC used for testing[3]. To see if all BUGs could transmit data the key input was checked on the laptop for each BUG.

Results

The testing happened on several occasions. Initially when nine BUGs were functional it was checked whether all nine could connect simultaneously. This went quite smoothly as all BUGs connected and were able to send their button press to the PC. It was unclear whether all BUGs could send a key press at the same time however in most cooperative games their is no need to press nine buttons simultaneously.

On a later occasion before a user test it was again tested if now ten BUGs could be connected simultaneously. Unfortunately on this occasion it was not possible to connect more than eight simultaneously as the BUGs would no longer appear as possible to connect after eight connections were established. It is possible this is due to an update performed on the library to ensure smoother reconnecting on windows and android.

5.2 Public testing

Since the BUGs are designed to be used by people playing games, the BUGs are also tested by people playing a game. To gather data in this test setup, a small survey is taken afterwards. It is seen people need some time to adapt to the controls and feeling of controlling only a part of

[3]HP Zbook studio G4

the game. The results seemed to be better when the test participants were communicating when playing, but this is not measured quantitatively. The games tested are listed in Appendix K, but this is not an exclusive list of compatible games.

The test group consists of 30 people between the ages of 17 and 26 (See Figure H.1a). Most of them mentioned they have played games before, 70% of them are familiar with party games. This genre is known for playing small mini games together (or against each other), similar to the game style created with the BUG. It should be noted that the set of test subjects likes to play these kind of games, thus enjoying the use of the BUG more than the average person might. This could lead to more positive results. See Figure H.1b for an overview of the games played by the test participants.

The test participants were asked about the comfort and size of the gamepad. Most of the participants rated the BUG as comfortable (see Figure H.2a). A big portion of the participants likes the size, however 26% thought it was too small and 13% thought it was too large (Figure H.2b). It could be stated the size is acceptable, since it is preferred by the average user.

The responses to the user interface are good: most people think the key bind indication with the LEDs on the side is clear and rate the overall user experience with a 4.4 out of 5. (See Figure H.3). Most people do however expect a high battery life of the BUG. The current version should be working for more or less two hours, most people expect it to work much longer. This could be because of the common game controllers available now (such as an Xbox controller) having a battery life of up to 30 hours. If this needed to be achieved, a redesign should be done, which is discussed in section 7.1.

Chapter 6

Conclusion

To see if this project was a success, one has to see if all the requirements in the beginning are met. To do this, the requirements were tested in the tests from chapter 5. To take the requirements laid out in chapter 2 from top to bottom; The gamepad is safe. In the 4 weeks working with the BUGs, none have malfunctioned in a dangerous way. This means all the LiPo batteries are intact and working. Some of the other components only malfunctioned by ceasing to function, but not creating a possible dangerous situation. No dangerous sharp edges were found and nobody was harmed when working with or using the BUGs. This is not the best way of testing if a product is safe. There are protocols which test if a product is safe. Since the BUGs are not going to be on the market and there was an insufficient amount of time, these protocol tests are not performed. As can be seen in the Range test, the PC noticed all the key presses until around 35 meters. This means that the keystroke was emulated perfectly in the first 35 meters. Because of this reason, the requirement to work wireless is also met. As can be seen in chapter 3, 2 controller designs were tested. The nunchuck model was rated the best. Both a separate BLE - MCU combination and an integrated BLE and MCU module were tested, the integrated solution being superior. Out of the charge and discharge tests, it can be stated that the BUGs can charge with a USB-C power supply, and charging and discharging takes about the same time.

The devices were found very user friendly according to the 30 test participants, which were able to play games like Pac-Man using the BUGs. At least five BUGs were used simultaneously by playing the game Bomberman, requiring five key inputs.

Chapter 7

Discussion

7.1 Recommendations

Although the prototype made in this research is received with positive feedback from test participants, some recommendations could be made when further developing this product.

7.1.1 3D Model

Before releasing the final gamepad, the 3D model should be evaluated. The current design is made functional, but with limited attention for ergonomic design. The design should also be adapted to injection moulding, requiring different design steps to create a durable model. Both issues could be resolved with a redesign. An important effect of a redesign is the PCB needs to be redesigned as well, since this is based on the basic shape of the gamepad.

7.1.2 User Interface

Although the user interface itself is good, some small remarks can be noted for a possible redesign. It is found it is very easy to press the configuration button when disconnection the charging cable, starting the BUG. This should be fixed, but this is possible when a charging dock for up to four BUGs is introduced. This makes charging overall a lot easier and user friendly, since it eliminates the hassle with multiple cables. A minor detail which could be fixed to make the BUG more colour blindness friendly is changing the charge indicator led from red-green to red-blue. Finally, the indication LEDs are not ideally located for left handed people, this could again benefit from a redesign.

7.1.3 Electronics

The MCU could be swapped for a more power efficient solution. The complete system currently draws 100mA, which is quite high for a BLE solution. Other solutions integrating an MCU with a BLE module are possible. An example of such a solution is a Nordic Semiconductor IC (NRF-series), which is known for its low power dissipation and wide input voltage range(1.8 up to 3.6V), but these require a custom antenna. When such an IC is implemented, the firmware needs to be rewritten as well since these MCUs use a different architecture. There is a hardware bug in the current BUGs. This bug is the ESP that need to be soldered upside-down. This was because the pinout was a bottom view in the datasheet [14]. This was not picked up on during design and review. It is still possible to solder, but it takes substantially more time to solder compared to soldering them directly onto the PCB

The used WS2812 LEDs for the indication for the key binding might not be the best solution for a final product. With the use of charlieplexing [22], twelve individual LEDs could be controlled by only four GPIO pins. By setting the IO pins to either low, high or high impedance, a specific LED can be controlled. By pulsing every LED with a frequency of at least 50Hz, all LEDs can appear to be on at the same time. This behaviour also has the positive side effect the LEDs will draw less power. This solution will however be more complex with the routing of the PCB and a different approach with the software of the ESP.

7.1.4 ESP Software

It will always be possible to further improve the software, but some remarks were found that could be improved in further versions. The BUG must save the configuration to its memory when received, instead when going to shutdown. This prevents the loss of configuration data when going in to a brownout state when the voltage of the battery is too low. Another remark is regarding recognisability of the BUGs: it should be possible to assign a custom Bluetooth name to the BUGs. Finally, the battery function could be optimised to be more accurate to all BUGs as it is currently based on a single ESP which can cause inaccuracies.

7.1.5 PC Software

The current software is far from perfect. It should be rewritten, but certain functions could be good to implement. One of these functions could be a mass connect function to connect multiple BUGs without the Windows Bluetooth menu. Also a mass configuration, so four or five BUGs could be configured to e.g. WASD plus space mode at the same time. To make interaction with custom games possible, an API could be designed to easily adjust the settings per BUG during gameplay.

7.1.6 Safety

As said in chapter 6, there are protocols to test if a product is safe. Since this can also involve looking for the limits of a system before it fails and since there were a limited amount of BUGs available, these tests were not performed before.

7.1.7 Testing

The testing has mainly been done with the student demographic. This was the demographic which was accessible during the Covid-19 pandemic. These tests could be repeated with a wider range of people.

Appendices

Appendix A

Terminology

To avoid confusion or explain some abbreviations used in this thesis, a table of explanations can be found below:

Term	Definition
BUG	Bluetooth Ultrasimple Gamepad, the prototype as a complete unit.
Gamepad	The complete handheld device, sometimes referred to as 'Controller'. To avoid confusion with the MCU, this is called the gamepad.
Key binding	The key press the gamepad sends to the connected device.
MCU	Micro Controller Unit, the processing power module used.
AT	Hayes Command Set, AT being short for Attention. This command set is used in a very low level communication with various chipsets.
BLE	Bluetooth Low Energy
GATT Profile	Generic Attribute Profile
HID	Human Interface Device, such as a keyboard, mouse or gamepad.
IC	Integrated Circuit, a small chip containing an internal circuit.
PCB	Printed Circuit Board, a board that allows for the mounting of electrical components and has internal connection paths to form a circuit
GUI	Graphical User Interface, a way of giving instructions to a computer using things that can be seen on the screen such as symbols and menus
PWM	Pulse Width Modulation. An alternating signal which is high for a percentage of time which can be used to supply less power then when the signal is always high
LDO	Low DropOut regulator. An electrical component allowing to supply a steady supply voltage

Appendix B

Prototyping Code

In this appendix the code for the project will be listed this includes the versions of the code for the prototype, the data acquisition code, and the GUI code.

B.1 Python data acquisition code

For the data collection, a small python script is written which reads the serial input. It logs the button number which is received and simulates a key press.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Sep 29 21:31:17 2021

@author: Jorn
"""

import keyboard
import serial
import csv  #  https://www.pythontutorial.net/python-basics/python-write-csv-file/
from datetime import datetime
    #https://thispointer.com/python-how-to-get-current-date-and-time-or-timestamp/
import os

serialPort = serial.Serial("COM10", 38400, timeout=2)

header = ["timestamp", "button"]

#serialPort.open()

btn = 0

timestamp = datetime.now()
timestampStr = timestamp.strftime("%Y%m%d-%H%M%S")

f = open(os.path.dirname(os.path.realpath(__file__))+'/'+timestampStr+'.csv', 'w',
    newline='')
writer = csv.writer(f)

```

```
writer.writerow(['time','button'])

serialPort.read_all()

print("Logger and keyboard emulator started. Quit with Ctrl+C to close COM-port")

try:
    while(1):

        # infinite loop to check incoming data on serial port. Convert to int
        if(serialPort.in_waiting>0):
            serialString = serialPort.readline()
            btn = int(serialString.decode('Ascii').split(",")[0])

            # If released, release space. If pressed, press space and log button number
            if (btn == 0):
                keyboard.release('down')
            else:
                keyboard.press('down')
                timestamp = datetime.now()
                timestampStr = timestamp.strftime("%H:%M:%S")
                writer.writerow(["\""+timestampStr+"\"", btn])

except:
    serialPort.close()
    f.close()
    print("Serial port closed, csv saved.")
```

B.2 C++ code for the Prototype

In this section the different versions of the C++ code for the prototype will be listed

B.2.1 Simple Bluetooth connectivity and LED control

The initial code for the ESP development kit that includes the indicator LEDs and basic Bluetooth transmission capabilities. Does not allow for easy reprogramming, a change of key binding requires re-flashing the gamepad.

```
/*
@Title BUG ESP code including indication LEDs Devkit
@Author Floris van der Heijde
@Delft University of Technology
@Date 05-10-2021

@Hardware
ESP-C3-12F kit dev board
neopixles

18 Pulldown button pin
19 Neopixle output pin

*/

#include <BleKeyboard.h>
```

```
17 #include <Adafruit_NeoPixel.h>
18
19 //Set the name of the BUG(Bluetooth Ultrasimple Gamepad
20 BleKeyboard bleKeyboard("BUG-ESP");
21
22 #define buttonPin   18  //set the buttonpin
23 #define NEOPIN      19  //set the pin for the neopixels
24 #define NUMPIXELS    5  //set the amount of neopixels
25 #define WHITE       255, 255, 255 //set the rgb value for white
26 #define BLUE        0, 0, 255     //set the rgb value for blue
27 #define RED         255, 0, 0     //set the rgb value for red
28 #define GREEN       0, 255, 0     //set the rgb value for green
29 #define PURPLE      255, 0, 255   //set the rgb value for purple
30 #define OFF         0, 0, 0       //set the rgb value for OFF
31
32 //Define char for space arrow up, down, left, and right
33 char space = 32;
34 char arrowup = 218;
35 char arrowdown = 217;
36 char arrowleft = 216;
37 char arrowright = 215;
38
39 //select the desired key and colour
40 #define key         arrowup
41 #define colour      GREEN
42
43
44 //NEOPIXEL known numbers button on the top: 0=left, 1=top, 2=centre, 3=bottom, 4=right
45
46 Adafruit_NeoPixel pixels(NUMPIXELS, NEOPIN, NEO_GRB + NEO_KHZ800);
47 #define DELAYVAL 500 // Time (in milliseconds) to pause between pixels
48
49 void setup() {
50   bleKeyboard.begin();                //start ble keyboard
51   pinMode(buttonPin, INPUT_PULLDOWN); //set buttonpin as pulldown so standard low
52   pixels.begin();
53 }
54
55 void loop() {
56
57   //if BUG set to arrowup light up the top led in the selected colour
58   if (key == arrowup) {
59     pixels.clear();                           //reset the pixels
60     pixels.setBrightness(15);                 //set the pixel brightness
61     pixels.setPixelColor(0, pixels.Color(OFF));       //set the pixels corresponding to the key to
         ↪ the selected colour
62     pixels.setPixelColor(1, pixels.Color(colour));
63     pixels.setPixelColor(2, pixels.Color(OFF));
64     pixels.setPixelColor(3, pixels.Color(OFF));
65     pixels.setPixelColor(4, pixels.Color(OFF));
66     pixels.show();                            //display the pixels according to the settings
67
68   }
69   //if BUG set to space light up the middle row of leds in the selected colour
```

```
70    if (key == space) {
71      pixels.clear();
72      pixels.setBrightness(15);
73      pixels.setPixelColor(0, pixels.Color(colour));
74      pixels.setPixelColor(1, pixels.Color(OFF));
75      pixels.setPixelColor(2, pixels.Color(colour));
76      pixels.setPixelColor(3, pixels.Color(OFF));
77      pixels.setPixelColor(4, pixels.Color(colour));
78      pixels.show();
79
80    }
81    //if BUG set to arrowdown light up the botton led in the selected colour
82    if (key == arrowdown) {
83      pixels.clear();
84      pixels.setBrightness(15);
85      pixels.setPixelColor(0, pixels.Color(OFF));
86      pixels.setPixelColor(1, pixels.Color(OFF));
87      pixels.setPixelColor(2, pixels.Color(OFF));
88      pixels.setPixelColor(3, pixels.Color(colour));
89      pixels.setPixelColor(4, pixels.Color(OFF));
90      pixels.show();
91
92    }
93    //if BUG set to arrowright light up the right led in the selected colour
94    if (key == arrowright) {
95      pixels.clear();
96      pixels.setBrightness(15);
97      pixels.setPixelColor(0, pixels.Color(OFF));
98      pixels.setPixelColor(1, pixels.Color(OFF));
99      pixels.setPixelColor(2, pixels.Color(OFF));
100     pixels.setPixelColor(3, pixels.Color(OFF));
101     pixels.setPixelColor(4, pixels.Color(colour));
102     pixels.show();
103
104   }
105   //if BUG set to arrowleft light up the left led in the selected colour
106   if (key == arrowleft) {
107     pixels.clear();
108     pixels.setBrightness(15);
109     pixels.setPixelColor(0, pixels.Color(colour));
110     pixels.setPixelColor(1, pixels.Color(OFF));
111     pixels.setPixelColor(2, pixels.Color(OFF));
112     pixels.setPixelColor(3, pixels.Color(OFF));
113     pixels.setPixelColor(4, pixels.Color(OFF));
114     pixels.show();
115
116   }
117   while (digitalRead(buttonPin) ==HIGH) {
118       bleKeyboard.press(key); //continuously send a spacebar when button is pressed
119   }
120   bleKeyboard.release(key);  //stop sending the spacebar when the button is released
121   delay(5);
122 }
```

Appendix C

Test data and Figures

C.1 Test subjects figures

In this section the figures providing an overview of every test subjects button usage and preference is given.

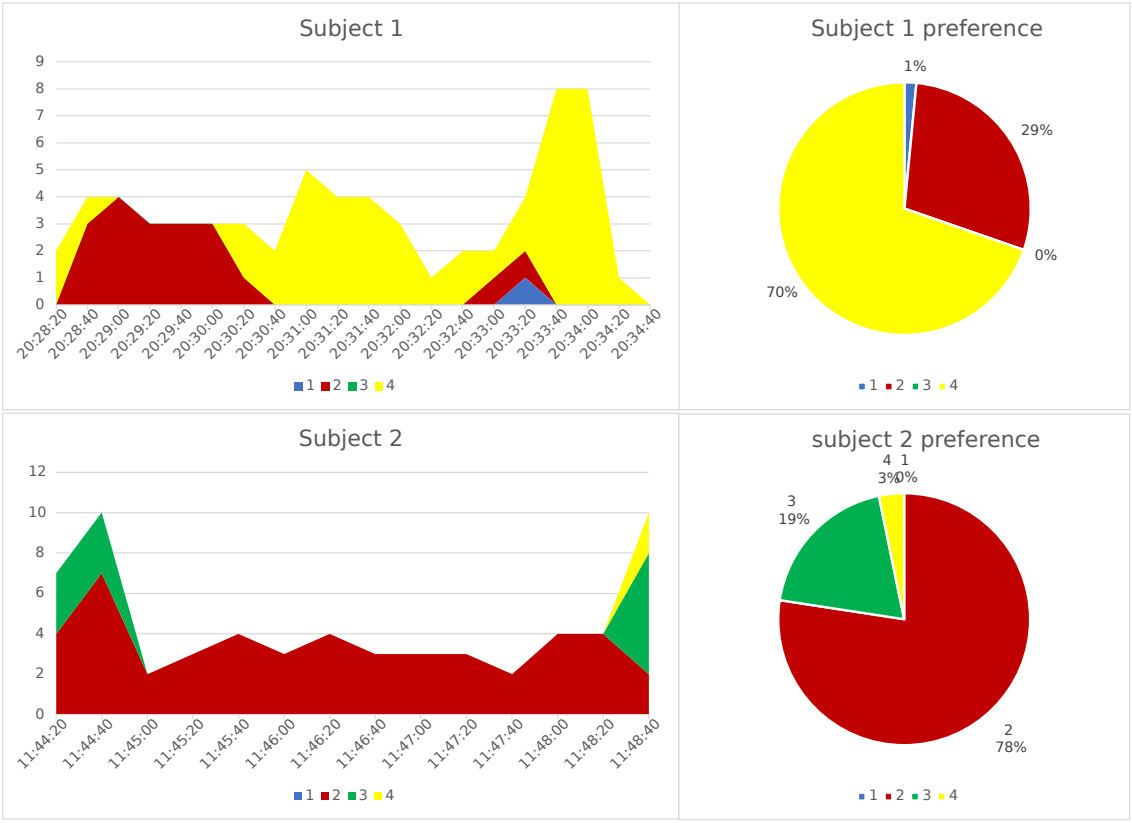


Figure C.1: Button usage and preference of subject 1 and 2

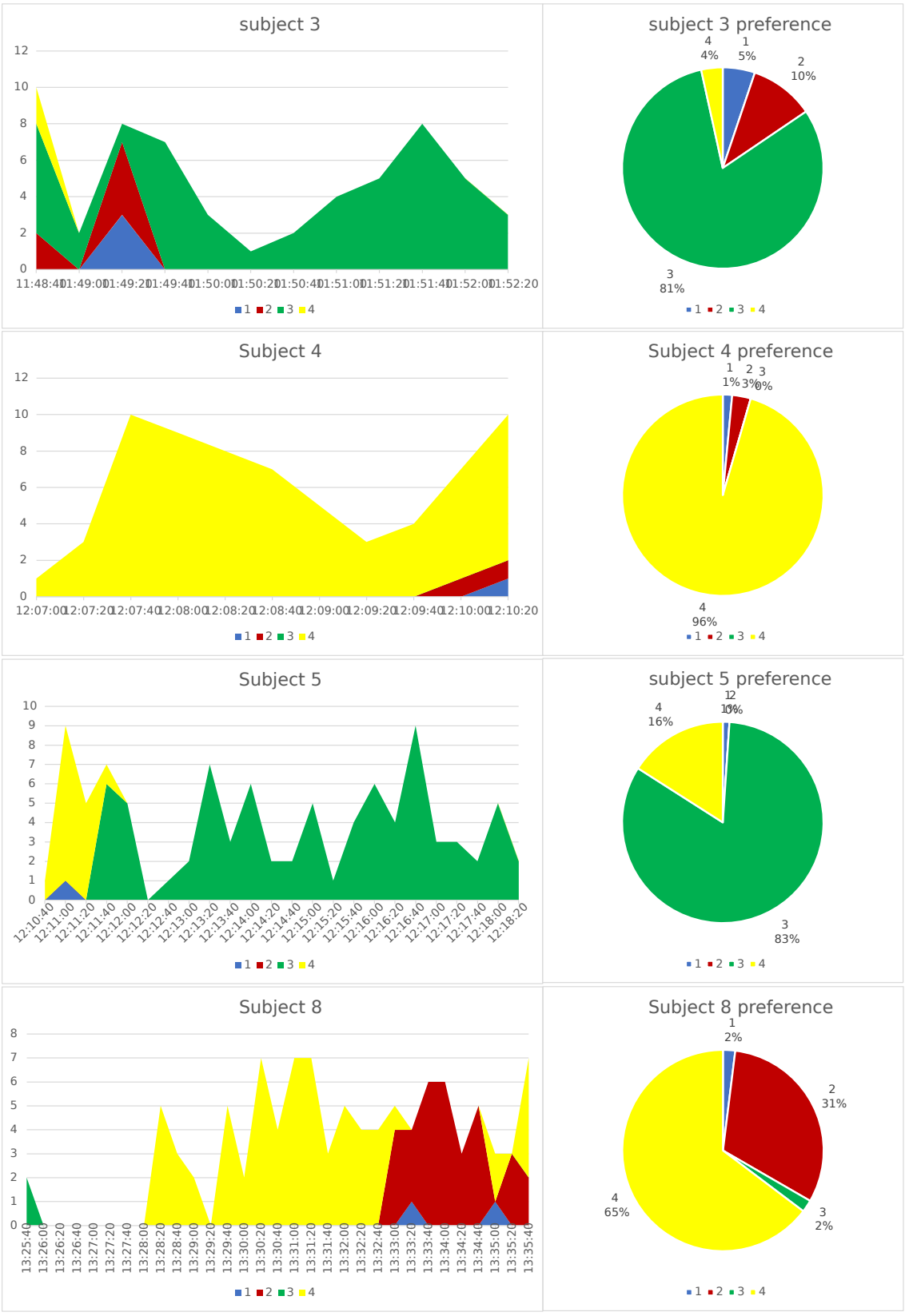


Figure C.2: Button usage and preference of subject 3, 4, 5, and 8

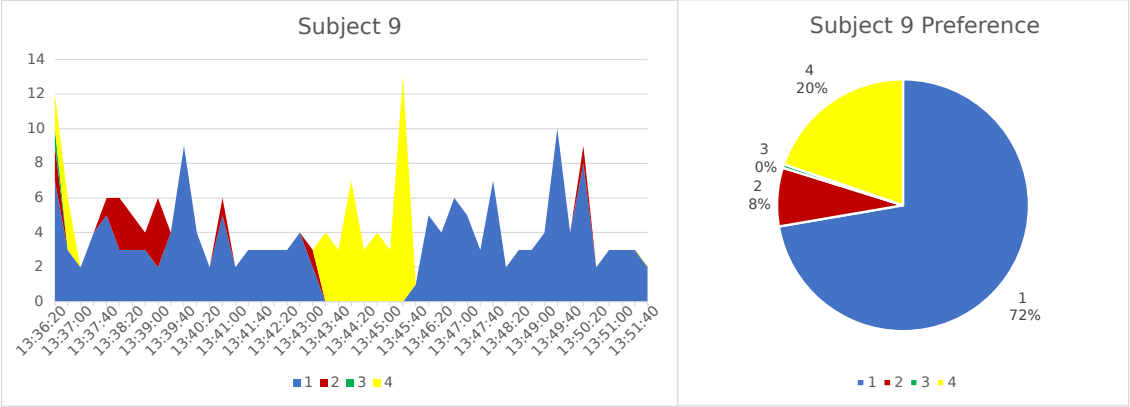
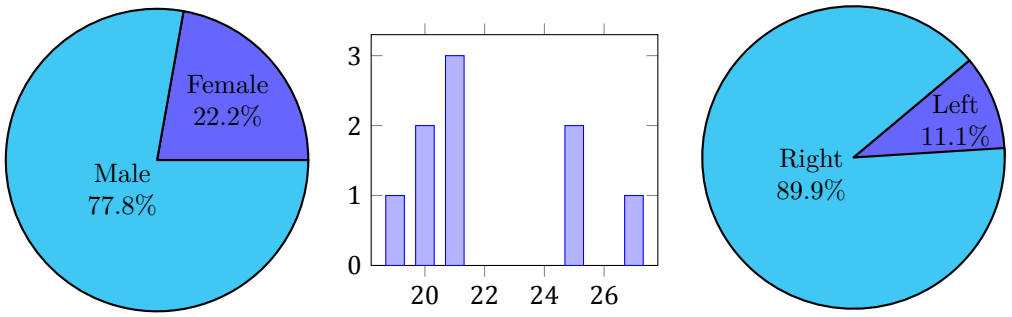


Figure C.3: Button usage and preference of subject 9

C.2 Survey results

In this section the automatically generated results overview from the Google Forms survey will be provided. A total of nine responses was gathered.



(a) Gender distribution among test participants

(b) Age of test participants

(c) Handiness distribution among test participants

Figure C.4: Characteristics of the test participants

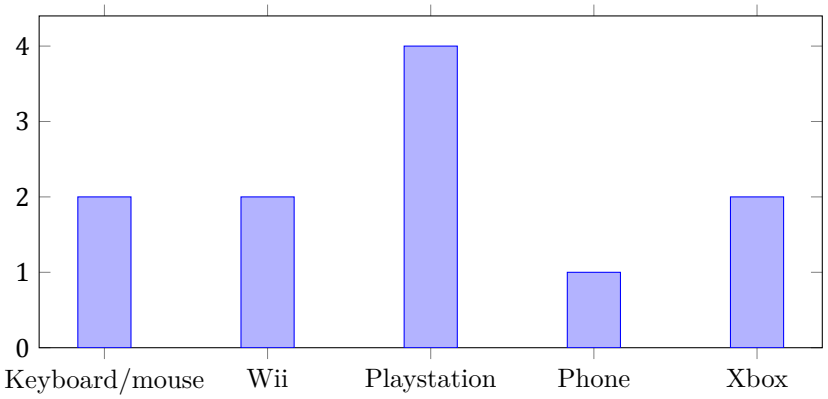
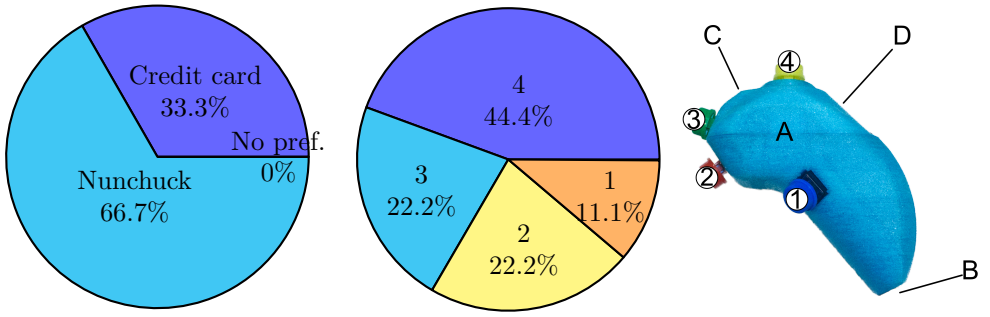


Figure C.5: Preference for gamepads among test participants, each bar representing the number of mentions.

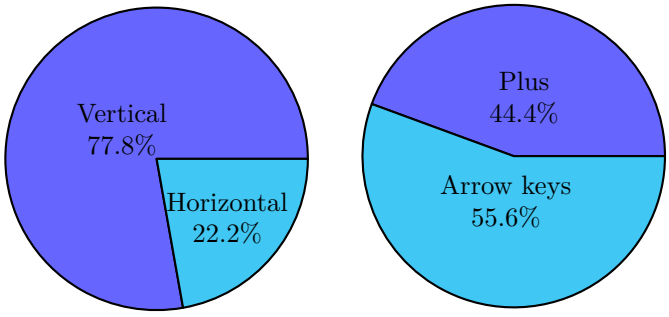


(a) Preference for both prototyping models

(b) Preference for button location on nunchuck model

(c) Location of buttons on the nunchuck itself

Figure C.6: Preference for prototype configurations (1)



(a) Preference for orientation of credit card prototype

(b) Preference for layout of indication LEDs

Figure C.7: Preference for prototype configurations (2)

Table C.1: Survey answers, open answer questions

Which LED location do you prefer? (nunchuck)
De zijkant (of evt bovenop)
Bij blauwe knop
Zijkant misschien midden voor
Onder blauw (handpalm)
Aan de linker zijkant
Onder de blauwe knop
Voorkant of tussen 3 en 4
Bovenkant
Tussen button 3 and 4
What do you think would be a good location for the reset/configuration key for the nunchuck?
Onderkant (cable input)
Bij groene
Out of the way
Bij draden
Aan de linker zijkant
Achterkant onder geel
Bij het draad
Onderkant zijkant
Button 3
What do you think would be a good location for the reset/configuration key for the credit card?
Onderkant (bij usb)
Midden
Midden bovenop
Schakelaar bovenkant, maar combi van knoppen
Bij de led array
Onderste led (bovenkant, bij usb)
Bij de boord
Onder bij de led array
Zijkant
Do you have any comments on either gamepad design you would like to share?
Nunchuck eerst getest.
Nunchuck eerst
11 48 switch
Nunchuck first
Switch 12 11
Nunchuck first
Switch 12 25
Ook knop 3 switch 13.36/37 credit card als afgerond
Nunchuck misschien beter als maar 1 knop

Appendix D

Custom GATT profile overview

D.1 Key binding characteristic

The key binding characteristic contains a two digit hexadecimal number. This integer is an almost direct representation of a key in the ASCII-table representation for keyboard values. The first 128 keys are an exact copy, the last 128 are representations of media and action keys such as the arrow keys. A section of the complete table can be found in Table D.1. This section only contains the values for keys which are potentially usable for use with the BUG.

Table D.1: table with ASCII values and possible key bindings represented by those values

Value	Key	Value	Key	Value	Key	Value	Key
0x20	Space	0x3B	;	0x6C	l	0x82	Left alt
0x2A	*	0x3D	=	0x6D	m	0x84	Right ctrl
0x2B	+	0x5B	[	0x6E	n	0x85	Right shift
0x2C	,	0x5C	\	0x6F	o	0x86	Right alt
0x2D	-	0x5D	]	0x70	p	0xD7	Right arrow
0x2E	.	0x60	‘	0x71	q	0xD8	Left arrow
0x2F	/	0x61	a	0x72	r	0xD9	Down arrow
0x30	0	0x62	b	0x73	s	0xDA	Up arrow
0x31	1	0x63	c	0x74	t	0xB1	Escape
0x32	2	0x64	d	0x75	u	0xB2	Backspace
0x33	3	0x65	e	0x76	v	0xB3	Tab
0x34	4	0x66	f	0x77	w	0xD1	Insert
0x35	5	0x67	g	0x78	x	0xD2	Home
0x36	6	0x68	h	0x79	y	0xD3	Page up
0x37	7	0x69	i	0x7A	z	0xD4	Delete
0x38	8	0x6A	j	0x80	Left ctrl	0xD5	End
0x39	9	0x6B	k	0x81	Left shift	0xD6	Page down

D.2 Indication characteristic

The indicator characteristic contains four sets of two digit hexadecimal numbers. A schematic representation can be found below, where the 0x shows it is in hexadecimal format:

0x AA BB CC 0D

In this example, 0xAA represents the amount of red to be shown. 0xBB and 0xCC represent the amount of green and blue, respectively. 0x0D will always be zero padded: this is a number between 0x00 and 0x0F. When converted to a binary number, each bit represents one of the indicator LEDs in the following order: the upper led followed by the left - middle - right led of the lower row.

D.3 Sleep timer characteristic

This characteristic contains a value up to 4 digits (hexadecimal), which is a direct representation of the amount of seconds before the BUG should shut down automatically. For example, the value 0x012C equals 300 in decimal notation. So if this characteristic is set to 0x012C, the BUG will shutdown after 5 minutes.

D.4 Information characteristic

This characteristic contains a simple input and behaves as the input are some sort of flags. It is used to implement a factory reset, an identification method. It is also used to implement a mode: which keys are implemented in the key array circulated by the configuration button. Also, a flag can be set to use WASD-keys instead of arrows.

Bit number	Functionality
0	Factory reset flag
1	Identification flag
2	Flag to use WASD instead of arrows
3	Include space in key array
4	Include up arrow in key array
5	Include left arrow in key array
6	Include down arrow in key array
7	Include right arrow in key array

The implementation of the key array can be found in subsection 4.4.1.

Appendix E

3D Model: Breakdown of design process

The design choices itself are discussed in section 4.1, the design steps are shown here. The used CAD software is OnShape, an online based editor.

Shape of the gamepad

First, a design was made based on the rough dimensions of the nunchuck model used in section 3.1.1. Since the design of the most ergonomic gamepad is not the scope of this research, not too much effort is put into optimising this design. The model is split vertically into two almost symmetric halves. The right half can be seen in Figure E.1. The two halves join together by some standing edge, creating a (theoretically) dust-proof joint between the two halves. This feature is not tested, since this is not the scope of this research.

Figure E.1: Basic shape of the gamepad

PCB Outline

With the basic shape in place, the outline for the PCB can be constructed. A shape is created with a margin of 0.5mm to all sides, to create a tight but still easy fit of the PCB into the gamepad. At this stage, the screw mounting holes are added, making sure the screws fix the two halves together while also fixating the PCB inside the model. The outline of this PCB is exported to the PCB design software (Altium). A small cutout can be seen in the circuit board. This is the location of the antenna of the used MCU, to prevent possible interfering of the Bluetooth signal.

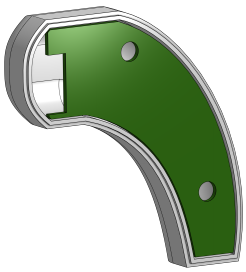


Figure E.2: PCB Outline

Bottom cutouts and LED guards

At the bottom, some room for the USB-c connector and configuration button is made. The cut outs are not clearly visible in Figure E.3. For the indicator LEDs, the light needs to be guided from the PCB to the side of the gamepad. This is done using LED guards, making sure each single LED can be clearly distinguished at the side of the BUG. The introduction of these guards in the 3D model fixes the position of the LEDs on the printed circuit board. The coordinates of these LEDs are used in the PCB layout.

Figure E.3: LED Guards

Button

To finalise the 3D model, the button needs to be added. The button itself used the feedback generated by the tactile button on the PCB, with a 3D-printed model on top of it to create a bigger pushable surface. The design should contain a cutout for the button and some retention for the button, making sure it does not fall out. This is done by placing the button on top of the model and a ledge above it do keep the button in place. See Figure E.4 for the implementation. Also a ledge is added at this stage, to keep the two halves in front together at all times. In Figure E.5, the complete model can be seen, including the designed circuit board.

Figure E.4: Final button

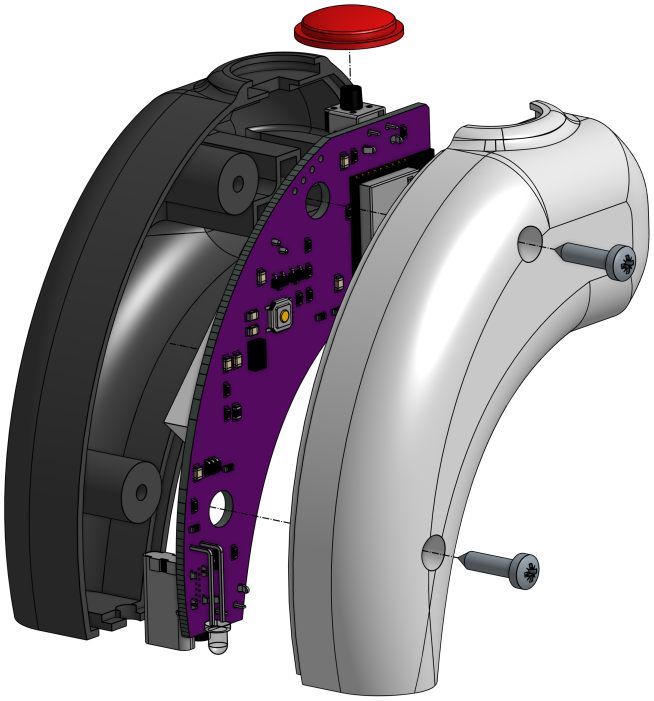


Figure E.5: Exploded view of the complete gamepad.

Appendix F

ESP Code

Below, the code programmed on the BUGs can be found. It can also be found on GitHub: https://github.com/jornvdl/BUG.

F.1 Main code

F.1.1 Main

```
/*
 *
 */

#define USE_NIMBLE
#define debug 1       // Set to 1 to enable serial debug information. Baudrate = 115200

#include <BleKeyboardGATT.h>
#include <Adafruit_NeoPixel.h>

// Import custom libraries
#include "variables.h"
#include "init.h"
#include "factory.h"
#include "led.h"
#include "battery.h"
#include "btn.h"
#include "shutdown.h"
#include "conf.h"

// setup() is run once at start up
void setup() {
  initSystem();
  sleepTimer = millis();
}

// loop() is looped after completion of the setup() function.
//  in this function, the main loop is implemented.
void loop() {
  if (bleKeyboard.isConnected()) {
```

```
31      ledsOn();                 // Enable LEDs
32      sleepTimer = millis();  // Reset sleeptimer
33
34      while(bleKeyboard.isConnected()) {
35        // React to a button press
36        if (digitalRead(btnPin)) {
37          sleepTimer = millis();
38          btnPress();
39        }
40
41        // React to a configuration press
42        if (digitalRead(confPin)) {
43          sleepTimer = millis();
44          confPress();
45        }
46
47        // Sleeptimer restart requested, so BLE char updated
48        if (*bleKeyboard.flgRstTimer()) {
49          if (debug) Serial.println("Sleeptimer reset requested from library");
50          sleepTimer = millis();
51          ledsOn();
52          batterySend();
53          bleKeyboard.flgRstTimer(false);
54        }
55
56        // Factory reset requested over BLE
57        if (*bleKeyboard.flgRstBUG()) {
58          if (debug) Serial.println("Reset BUG from BLE flag.");
59          factory();
60        }
61
62        // Identify BUG
63        if (*bleKeyboard.flgIdentify()) {
64          ledsBlink(true, false);
65        }
66
67        // Shutdown if to long inactivity
68        int timeLived = millis() - sleepTimer;
69        if (timeLived > *bleKeyboard.getTimeout()) {
70          if (debug) Serial.println("Sleeptimer exceeded!");
71          shutdown();
72        }
73      }
74    }
75
76    else { // !bleKeyboard.isConnected()
77
78      while(!bleKeyboard.isConnected()) {
79        // Let the LEDs blink
80        ledsBlink(false, true);
81
82        // React to a configuration press
83        if (digitalRead(confPin)) {
84          sleepTimer = millis();
```

```
85          confPress();
86        }
87
88        // Shutdown if to long inactivity
89        int timeLived = millis() - sleepTimer;
90        if (timeLived > *bleKeyboard.getTimeout()) {
91          if (debug) Serial.println("Sleeptimer exceeded!");
92          shutdown();
93        }
94      }
95    }
96  }
```

Header: Battery

```
1   /*
2    * Function to extract battery percentage out of the ADC pin.
3    */
4
5   #ifndef _BATTERY_H
6   #define _BATTERY_H
7
8   void batterySend() {
9     int batmeasure[20];
10    float total;
11    for(int m = 0; m < 20; m++) {
12      batmeasure[m] = analogRead(batPin);
13      total = total + batmeasure[m];
14    }
15    float avg         = total/20;
16    float bV          = 0.0021*avg - 0.4452;
17    float battPercent = sqrt((bV/a) + (sq(b)/2*a) - (c/a)) - b/(2*a);
18
19    // Round to integer (rounded down to 5%)
20    int batt5         = battPercent/5;
21    int battFin       = batt5*5;
22
23    battFin = min( 100, max( 0, battFin));
24
25    bleKeyboard.setBatteryLevel(battFin);
26
27    if (debug) Serial.println("Battery percentage calculated and sent");
28  }
29
30  #endif // _BATTERY_H
```

Header: Button

```
1   /*
2    * Button header
3    * btnPress() is called when main detects a button press. It sends the event
4    * to the library and waits until the button is released. It then sends that
5    * event to the library and ends the function, returning to the main scope.
6    */
```

```

#ifndef _BTN_H
#define _BTN_H


void btnPress() {
  if (debug) Serial.println("Game button pressed!");

  bleKeyboard.press();
  while(digitalRead(btnPin)) {};
  bleKeyboard.releaseAll();

  batterySend();

  if (debug) Serial.println("Game button released!");
}

#endif // _BTN_H
```

Header: Configuration key

```
/*
 *  This function will handle the configuration button behaviour. It is send here when pressed,
 *  and acts according to the timing values and release of the button. It has a very simple
    ↪ debounce and 3 functions:
 *  - next key, a short press
 *  - shutdown, a middle press
 *  - revert to factory settings, a long press.
 *  Timings of the press durations are set in variables.h
 */

#ifndef _CONF_H
#define _CONF_H

bool modeSelect() {
  bool* ptrMode = bleKeyboard.cirKeys();
  if (*ptrMode || *(ptrMode+1) || *(ptrMode+2) || *(ptrMode+3) || *(ptrMode+4) ) {
    if (debug) {
      Serial.print("modeSelect = true, since ");

        ↪ Serial.print(*ptrMode);Serial.print(*(ptrMode+1));Serial.print(*(ptrMode+2));Serial.print(*(ptrMode+3))
    }
    return 1;
  }
  else {
    if (debug) Serial.println("modeSelect = false");
    return 0;
  }
}

void confRelease(int pressTime) {
  if (debug) Serial.println("conf:released");
  int releaseTime = millis();
```

```
31    int durationTime = releaseTime - pressTime;
32    bool* ptrMode = bleKeyboard.cirKeys();
33
34    if((durationTime < shutdownTime) && modeSelect()) {
35      if (debug) Serial.println("conf:next key");
36      confSelect++;
37      if(confSelect > 4) confSelect = 0;
38
39
40      while (!*(ptrMode + confSelect)) {
41        confSelect++;
42        if(confSelect > 4) confSelect = 0;
43      }
44
45      //Write new values to library
46      if (*bleKeyboard.flgWASD()) {
47        bleKeyboard.setKeybind  ( &keyWASD[confSelect]    );
48      }
49      else {
50        bleKeyboard.setKeybind  ( &keyArrows[confSelect]  );
51      }
52      bleKeyboard.setLayout     ( &keyLayout[confSelect]  );
53
54      //Update LEDs if BLE connected, otherwise it is handled by ledBlink()
55      if (bleKeyboard.isConnected()) ledsOn();
56
57    }
58    else if(durationTime > shutdownTime && durationTime < factoryTime) {
59      if (debug) Serial.println("conf:shutdown");
60      shutdown();
61    }
62    else if(durationTime > factoryTime) {
63      if (debug) Serial.println("conf:factory");
64      factory();
65    }
66
67  }
68
69  void confPress(){
70    if (debug) Serial.println("conf:pressed");
71    int confTimer = millis();
72    while(digitalRead(confPin) || (millis()-confTimer) < debounceTime) {
73      if ((millis() - confTimer) > shutdownTime && (millis()- confTimer) < factoryTime) {
74        ledsOff();
75      }
76      else if ((millis() - confTimer) >= factoryTime) {
77        ledsBlink(true, false);
78      }
79      else if (!bleKeyboard.isConnected()) ledsBlink(false, true);
80    }
81    confRelease(confTimer);
82  }
83
84  #endif // _CONF_H
```

Header: Factory Settings

```
/*
 * Function to reset BUG to the factory settings
 */

#ifndef _FACTORY_H
#define _FACTORY_H
void factory() {
  if (debug) Serial.println("factory: writing factory settings to library");
  if (factWASD) {
    bleKeyboard.setKeybind ( &keyWASD[factConf] );
    bleKeyboard.flgWASD(true);
  }
  else {
    bleKeyboard.setKeybind ( &keyArrows[factConf] );
    bleKeyboard.flgWASD(false);
  }
  bleKeyboard.setColour  ( &factColour[0] );
  bleKeyboard.cirKeys    ( &factMode[0]   );
  bleKeyboard.setLayout  ( &keyLayout[0]  );

  bleKeyboard.flgRstBUG(false);
  bleKeyboard.flgIdentify(false);

  confSelect = factConf;

  ledsOn();
}


#endif // _FACTORY_H
```

Header: Initialisation

```
/*  Initialization header
 *  All initializations will be done here.
 */

#ifndef _INIT_H
#define _INIT_H

#include "memory.h"


void initSystem() {
  // Enable debug output over serial
  if (debug) Serial.begin(115200);
  if (debug) Serial.println("Debug serial started.");

  // Start BLE Keyboard server and LED controller
  bleKeyboard.begin();
  leds.begin();

```

```
  // Configure GPIO pins
  pinMode(btnPin,  INPUT);
  pinMode(confPin, INPUT);
  pinMode(batPin,  INPUT);
  pinMode(rdyPin,  OUTPUT);

  // Get data from memory and set to library
  memory2lib();
  if (debug) Serial.println("Memory read and written to lib.");

  bleKeyboard.flgIdentify(false);

  digitalWrite(rdyPin, HIGH);
}


#endif // _INIT_H
```

Header: LED Control

```
/*
 *  Here, the indicator LED behaviour is managed. Three options are available:
 *  All leds off, a selection (according to a given layout) on or blinking.
 *  All three options have a seperate function and are accompanied by a function
 *  to convert the layout to a more useable format.
 */

#ifndef _LED_H
#define _LED_H

int* layout_hextobin(){
  static int binTemp[]   = {0,0,0,0};
         int layout_main = *bleKeyboard.getLayout();

  binTemp[3] = (layout_main &1);
  binTemp[2] = (layout_main>>1 &1);
  binTemp[1] = (layout_main>>2 &1);
  binTemp[0] = (layout_main>>3 &1);

  //if (debug) Serial.println("Layout hextobin");

  return binTemp;
}


void ledsOn() {
  //Set neopixles according to ledBin top = ledBin[3], left = ledBin[2], down = ledBin[1], right =
    ↪ ledBin[0]
  //neo pixels: top = 0, left = 1, down = 2, right = 3;
  bool ledBin[4] = {0,0,0,0};

  ledBin[0] = *layout_hextobin();
  ledBin[1] = *(layout_hextobin()+1);
  ledBin[2] = *(layout_hextobin()+2);
```

```
34    ledBin[3] = *(layout_hextobin()+3);
35
36    int* ptrColour = bleKeyboard.getColour();
37    long ledColour = leds.Color(*ptrColour, *(ptrColour+1), *(ptrColour+2));
38
39    for(int i = 0; i < 4; i++) {
40      if (ledBin[i]) {
41        leds.setPixelColor((3-i), ledColour);
42      }
43      else {
44        leds.setPixelColor((3-i), leds.Color(0,0,0));
45      }
46      leds.show();
47    }
48
49    if (debug) Serial.println("LEDs On/update");
50
51  }
52
53
54  void ledsOff() {
55    // Turn off all Neopixels, but only if they are currently on to prevent
56    // unneccesary communications
57    bool currentState = 0;
58    for (int j = 0; j < 4; j++) {
59      currentState = currentState || (leds.getPixelColor(j) > 0 );
60    }
61
62    if (currentState) {
63      leds.clear();
64      leds.show();
65      if (debug) Serial.println("LEDs off");
66    }
67  }
68
69  void ledsBlink(bool keepColour, bool keepLayout) {
70  // Function to let the leds blink. The function must be called in a loop, since it only updates is
      ↪ state,
71  // is does not handle the blinking it self.
72
73    long ledColour;
74
75    // Determining the color to show during blinking
76    if (keepColour) {
77      int* ptrColour = bleKeyboard.getColour();
78      ledColour = leds.Color(*ptrColour, *(ptrColour+1), *(ptrColour+2));
79    }
80    else { // If not using current colour, then select blinkColour set in variables.
81      ledColour = leds.Color(blinkColour[0], blinkColour[1], blinkColour[2]);
82    }
83
84    // Determine if LEDs should be on or off. This is done using the system time
85    //  by using the modulo and the millis().
86    int ledPeriod = millis() % (blinkTime * 2);
```

```
87    bool ledEnabled = ledPeriod > blinkTime;
88
89    // Getting current state, to prevent unnecessary updates to the leds
90    bool currentState = 0;
91    for (int j = 0; j < 4; j++) {
92      currentState = currentState || (leds.getPixelColor(j) > 0 );
93    }
94
95    // Debug output
96    if (debug && !currentState &&  ledEnabled) {
97      Serial.print("LED Blink: on  ");
98      if (keepColour) Serial.print("[keepcolour]");
99      if (keepLayout) {
100       Serial.print("[keeplayout=");
101       Serial.print(*bleKeyboard.getLayout());
102       Serial.print("]");
103     }
104     Serial.print("\n");
105   }
106   if (debug &&  currentState && !ledEnabled) Serial.println("LED Blink: off");
107
108   // Set LEDs to corrent state and update
109   for(int i = 0; i < 4; i++) {
110     if( !currentState && ledEnabled) {              // If currently off, but supposed to be on
111       if (*(layout_hextobin()+i) || !keepLayout ) { // and specific LED should be on
112         leds.setPixelColor((3-i), ledColour);       // set LED colour
113       }
114       else {                                        // otherwise set off (layout specific)
115         leds.setPixelColor((3-i), leds.Color(0,0,0));
116       }
117     }
118     else if ( currentState && !ledEnabled) {        // If currently on, but supposed to be off
119       leds.setPixelColor((3-i), leds.Color(0,0,0)); // Turn off
120     }
121   }
122   leds.show();
123 }
124
125 #endif // _LED_H
```

Header: Memory management

```
1  /*
2   *  In this file, saving and retrieving data to the internal EEPROM
3   *  memory will be handled. Except for the (global) confSelect, this
4   *  is stored/saved to the BleKeyboard lib, handling a lot of the data.
5   */
6
7  #ifndef _MEMORY_H
8  #define _MEMORY_H
9
10 #include <Preferences.h>
11
12 Preferences memory;
```

```
13
14  void memory2lib() {
15    int  memKey;
16    int  memColour[3];
17    bool memMode[5];
18    int  memLayout;
19    int  memSleep;
20
21    memory.begin("bug_data",true);
22
23    // Select proper factory settings for first init
24    if (!factWASD) {
25      memKey      = memory.getInt("key",    keyArrows[factConf] );
26    } else {
27      memKey      = memory.getInt("key",    keyWASD[factConf]   );
28    }
29
30    memColour[0]  = memory.getInt("cRed",   factColour[0]       );
31    memColour[1]  = memory.getInt("cGreen", factColour[1]       );
32    memColour[2]  = memory.getInt("cBlue",  factColour[2]       );
33    memMode[0]    = memory.getInt("mode0",  factMode[0]         );
34    memMode[1]    = memory.getInt("mode1",  factMode[1]         );
35    memMode[2]    = memory.getInt("mode2",  factMode[2]         );
36    memMode[3]    = memory.getInt("mode3",  factMode[3]         );
37    memMode[4]    = memory.getInt("mode4",  factMode[4]         );
38    memLayout     = memory.getInt("layout", keyLayout[factConf] );
39    memSleep      = memory.getInt("timeout",factSleep           );
40    confSelect    = memory.getInt("conf",   factConf            );
41    memory.end();
42
43    bleKeyboard.setKeybind ( &memKey       );
44    bleKeyboard.setColour  ( &memColour[0] );
45    bleKeyboard.cirKeys    ( &memMode[0]   );
46    bleKeyboard.setLayout  ( &memLayout    );
47    bleKeyboard.setTimeout ( &memSleep     );
48    bleKeyboard.flgWASD    ( factWASD      );
49    bleKeyboard.flgRstTimer( false         );
50    bleKeyboard.flgIdentify( false         );
51    bleKeyboard.flgRstBUG  ( false         );
52  }
53
54  void lib2memory() {
55    int*  memColour = bleKeyboard.getColour();
56    bool* memMode   = bleKeyboard.cirKeys();
57
58    memory.begin("bug_data", false);
59    memory.putInt("key",    *bleKeyboard.getKeybind() );
60    memory.putInt("cRed",   *(memColour)              );
61    memory.putInt("cGreen", *(memColour+1)            );
62    memory.putInt("cBlue",  *(memColour+2)            );
63    memory.putInt("mode0",  *(memMode)                );
64    memory.putInt("mode1",  *(memMode+1)              );
65    memory.putInt("mode2",  *(memMode+2)              );
66    memory.putInt("mode3",  *(memMode+3)              );
```

```
67    memory.putInt("mode4",  *(memMode+4)              );
68    memory.putInt("layout", *bleKeyboard.getLayout()  );
69    memory.putInt("timeout",*bleKeyboard.getTimeout() );
70    memory.putInt("conf",    confSelect               );
71  }
72
73  #endif // _MEMORY_H
```

Header: Shutdown protocol

```
1   /*
2    *  Function to handle shutdown protocol for a correct shutdown.
3    */
4
5   #ifndef _SHUTDOWN_H
6   #define _SHUTDOWN_H
7   #include "memory.h"
8
9   void shutdown() {
10    if (debug) Serial.println("Shutting down! Turning LEDs off and writing memory...");
11    ledsOff();                  //Turn Leds off
12    lib2memory();                 //Memory storage
13    delay(50);
14    digitalWrite(rdyPin,LOW);   //Set the readyPin low
15    esp_deep_sleep_start();     //Set the ESP to deep sleep
16  }
17
18  #endif // _SHUTDOWN_H
```

Header: Variables

```
1   /*
2    *   Overview of all definitions, global variables. Small configurations can be set here, such
3    *   as default factory settings, pin declarations and timing values
4    */
5
6   #ifndef _VARIABLES_H
7   #define _VARIABLES_H
8
9   ////////////////////////// Definitions and values ////////////////////////////
10  // Device Info
11  #define deviceName    "Leopard Moth"
12  #define manufacturer  "Bluetooth Ultrasimple Gamepad"
13
14  // Pin declarations
15  #define btnPin        19
16  #define confPin       2
17  #define ledPin        1
18  #define rdyPin        4
19  #define batPin        0
20
21  // Led configuration variables
22  #define numLeds       4
23  #define blinkTime     600
```

```
24 int blinkColour[3]  = {0x0, 0x0, 0xFF};
25
26 // Configuration button variables
27 #define debounceTime  250   // in millis
28 #define shutdownTime  3000  // in millis
29 #define factoryTime   7000  // in millis
30
31 // Factory settings variables
32 int  factSleep     = 360;                // in seconds
33 int  factColour[3] = {0x22, 0xA0, 0xFF};   // R,G,B values
34 bool factMode[5]   = {1,1,1,1,0};          // 0 disabled, 1 enabled (order: ↑←↓→␣)
35 int  factConf      = 0;                  // Start value of confSelect. Range [0,4]
36 bool factWASD      = false;                    // Use WASD instead of arrows
37
38 // Keybind arrays and corresponding layout
39 #define keyRight      215
40 #define keyLeft       216
41 #define keyDown       217
42 #define keyUp         218
43 #define keyW          119
44 #define keyA          97
45 #define keyS          115
46 #define keyD          100
47 #define keySpace      32
48
49 int keyArrows[5]  = {keyUp, keyLeft, keyDown, keyRight, keySpace};
50 int keyWASD[5]    = {keyW,  keyA,    keyS,    keyD,     keySpace};
51 int keyLayout[5]  = {0x01,  0x02,    0x04,    0x08,     0x0E};
52
53 // Battery percentage variables equation in battery.h
54 float a = 0.00005;
55 float b = 0.002;
56 float c = 3.1787;
57
58 ////////////////////////////// Global variables ///////////////////////////////
59 // Classes from libraries
60 BleKeyboard      bleKeyboard(deviceName, manufacturer);
61 Adafruit_NeoPixel leds(numLeds, ledPin, NEO_GRB + NEO_KHZ800);
62
63 // Mode select
64 int confSelect;
65
66 // Timers
67 volatile long sleepTimer;
68
69 #endif // _VARIABLES_H
```

F.2 Customised library

Since an existing library is used (see the Github cited in [7]), only the changes and additions are shown here.

Main

```
1  // Updated BLEKeyboard library base to 0.3.1 (1 oct 2021)
2  // Updated Custom part on 24 nov 2021
3  // ESP32-BLE-Keyboard used as base library, written by T-vK
4  //                    https://github.com/T-vK/ESP32-BLE-Keyboard
5  //
6  // Added custom GATT profiles and editted library to use with BUG: Bluetooth Ultrasimple
     ↪ Gamepad
7  // Editten by Jorn van der Linden

       ...

36 /// Flags and values for communication over custom BLE characteristic
37 // KEYBIND CHARACTERISTIC
38 int keystroke;  // integer value corresponding to keybind
39
40 // INDICATOR CHARACTERISTIC
41 int colour[3];  // RGB colour value of the LEDs: each int in range 0x00~0xFF
42 int layout;     // integer representing layout of indicator leds
43
44 // TIMEOUT CHARACTERISTIC
45 int timeout;    // store integer value of timeout before sleep in seconds
46 bool rstTimer;        // general flag to reset timer when ANY characteristic is updated
47
48 // INFO CHARACTERISTIC
49 bool rstBUG;                // flag: reset to factory settings
50 bool identify;         // flag: flash indicators to identify BUG
51 bool wasd;                     // flag: when circulating, use WASD keybinds instead of arrows
52 bool keys[5];               // flag-array: select which keys to circulate. When all are zero, a custom
     ↪ key should be set.
53
54
55 // When new key is received, update global variable and acknowledge.
56 class keyCallbacks: public BLECharacteristicCallbacks {
57   void onWrite(BLECharacteristic *keyCharacteristic) {
58     uint8_t *data = keyCharacteristic->getData();
59
60     if (*data>0 && *data<256) {
61       keystroke = *data;
62                 rstTimer = true;
63       keyCharacteristic->setValue(keystroke);
64       //keyCharacteristic->notify(true); // Something like this to notify, test later.
65     }
66   }
67 };
68 // When new layout is received, update global variable and acknowledge.
69 class indicatorCallbacks: public BLECharacteristicCallbacks {
70   void onWrite(BLECharacteristic *indicatorCharacteristic) {
71     uint8_t *data = indicatorCharacteristic->getData();
72     colour[0] = *data;
73     colour[1] = *(data+1);
74     colour[2] = *(data+2);
75     layout = *(data+3);
76
```

```
77      int returnValue = (layout << 24) + (colour[2] << 16) + (colour[1] << 8) + colour[0];
78      indicatorCharacteristic->setValue(returnValue);
79
80      rstTimer = true;
81    }
82  };
83  // When new timeout is received, update global variable and acknowledge.
84  class timeoutCallbacks: public BLECharacteristicCallbacks {
85    void onWrite(BLECharacteristic *timeoutCharacteristic) {
86      uint8_t *data = timeoutCharacteristic->getData();
87      if (*(data+1)>0 || *data > 0) {
88        timeout = *data * 256 + *(data+1);
89        timeoutCharacteristic->setValue(timeout);
90        rstTimer = true;
91      }
92    }
93  };
94  // When state is received, check is reset (then do this) or call. Reply with (resetted) states.
95  class stateCallbacks: public BLECharacteristicCallbacks {
96    void onWrite(BLECharacteristic *stateCharacteristic) {
97      uint8_t *data = stateCharacteristic->getData();
98
99      if ((*data & 0x80) == 0x80) {                                    // Reset the BUG
100           rstBUG = true;
101     }
102
103     identify      = (*data & 0x40) == 0x40;
104     wasd                = (*data & 0x20) == 0x20;
105     keys[4]           = (*data & 0x10) == 0x10;
106     keys[0]           =      (*data & 0x08) == 0x08;
107     keys[1]           =      (*data & 0x04) == 0x04;
108     keys[2]           =      (*data & 0x02) == 0x02;
109     keys[3]           =      (*data & 0x01) == 0x01;
110
111     rstTimer = true;
112
113     int written = rstBUG * 128 + identify * 64 + wasd * 32 + keys[4] * 16 + keys[0] * 8 +
          ↪ keys[1] * 4 + keys[2] * 2 + keys[1];
114     stateCharacteristic->setValue(written);
115   }
116 };

      ...

201 // Newly added code below
202 // Used UUID's (https://www.uuidgenerator.net/)
203 #define ServiceUUID "0ba682ae-4f1f-4e9b-be2a-809c224540fd"
204 #define KeyUUID     "3e7f5770-d6b7-4709-9b4b-951c63f97aaa"
205 #define IndicUUID   "6f1f3ce2-cb88-4c5a-9ba1-6e19369b8bbb"
206 #define TimeoutUUID "6de0e9a1-7f07-4e64-81b0-a8ca334bcccc"
207 #define StateUUID   "99503c7d-6924-413c-bb7d-db7e913fbddd"

      ...

274 void BleKeyboard::begin(void)
275 {
```

```
276   BLEDevice::init(deviceName);
277   BLEServer *pServer = BLEDevice::createServer();
278
279   BLEService *pService = pServer->createService(ServiceUUID); // Create a new service for the
        ↪ GATT service
280
281       // Creating multiple characteristics on the GATT service.
282   keyCharacteristic = pService->createCharacteristic(
283           KeyUUID,
284           BLECharacteristic::PROPERTY_WRITE |
285           BLECharacteristic::PROPERTY_READ |
286           BLECharacteristic::PROPERTY_NOTIFY
287   );
288   indicatorCharacteristic = pService->createCharacteristic(
289           IndicUUID,
290           BLECharacteristic::PROPERTY_WRITE |
291           BLECharacteristic::PROPERTY_READ |
292           BLECharacteristic::PROPERTY_NOTIFY
293   );
294   timeoutCharacteristic = pService->createCharacteristic(
295           TimeoutUUID,
296           BLECharacteristic::PROPERTY_WRITE |
297           BLECharacteristic::PROPERTY_READ |
298           BLECharacteristic::PROPERTY_NOTIFY
299   );
300   stateCharacteristic = pService->createCharacteristic(
301           StateUUID,
302           BLECharacteristic::PROPERTY_WRITE |
303           BLECharacteristic::PROPERTY_READ |
304           BLECharacteristic::PROPERTY_NOTIFY
305   );
306
307   pService->start();
308
309   // Create callbacks when a new item is received
310   keyCharacteristic->setCallbacks(new keyCallbacks());
311   indicatorCharacteristic->setCallbacks(new indicatorCallbacks());
312   timeoutCharacteristic->setCallbacks(new timeoutCallbacks());
313   stateCharacteristic->setCallbacks(new stateCallbacks());
314 }

      ...

553 size_t BleKeyboard::press()
554 {
555   uint8_t k = keystroke;
556   uint8_t i;
557   // Check if it is possible to hardcode the use of 'keystroke' var
558   if (k >= 136) {     // it's a non-printing key (not a modifier)
559     k = k - 136;
560   } else if (k >= 128) {  // it's a modifier key
561     _keyReport.modifiers |= (1<<(k-128));
562     k = 0;
563   } else {        // it's a printing key
564     k = pgm_read_byte(_asciimap + k);
```

```
565     if (!k) {
566       setWriteError();
567       return 0;
568     }
569     if (k & 0x80) {           // it's a capital letter or other character reached with shift
570       _keyReport.modifiers |= 0x02; // the left shift modifier
571       k &= 0x7F;
572     }
573   }
574
575   // Add k to the key report only if it's not already present
576   // and if there is an empty slot.
577   if (_keyReport.keys[0] != k && _keyReport.keys[1] != k &&
578     _keyReport.keys[2] != k && _keyReport.keys[3] != k &&
579     _keyReport.keys[4] != k && _keyReport.keys[5] != k) {
580
581     for (i=0; i<6; i++) {
582       if (_keyReport.keys[i] == 0x00) {
583         _keyReport.keys[i] = k;
584         break;
585       }
586     }
587     if (i == 6) {
588       setWriteError();
589       return 0;
590     }
591   }
592   sendReport(&_keyReport);
593   return 1;
594 }

    ...

744 int* BleKeyboard::getKeybind() { // not needed when using hardcoded keys in lib
745       return &keystroke;
746 }
747
748 int* BleKeyboard::getTimeout() {
749       return &timeout;
750 }
751
752 int* BleKeyboard::getColour() {
753       return &colour[0];
754 }
755
756 int* BleKeyboard::getLayout() {
757       return &layout;
758 }
759
760 void BleKeyboard::setKeybind(int* k) {
761       keystroke = *k;
762       keyCharacteristic->setValue(keystroke);
763 }
764
765 void BleKeyboard::setTimeout(int* t) {
```

```
766         timeout = *t;
767   timeoutCharacteristic->setValue(timeout);
768 }
769
770 void BleKeyboard::setColour(int* c) {
771         for (int i = 0; i < 3; i++) {
772                 colour[i] = *(c+i);
773         }
774   int returnValue = (layout << 24) + (colour[2] << 16) + (colour[1] << 8) + colour[0];
775   indicatorCharacteristic->setValue(returnValue);
776 }
777
778 void BleKeyboard::setLayout(int* l) {
779         layout = *l;
780   int returnValue = (layout << 24) + (colour[2] << 16) + (colour[1] << 8) + colour[0];
781   indicatorCharacteristic->setValue(returnValue);
782 }
783
784 // Share and reset timer flag
785 bool* BleKeyboard::flgRstTimer() {
786         return &rstTimer;
787 }
788
789 void BleKeyboard::flgRstTimer(bool flg) {
790         rstTimer = flg;
791 }
792
793 // Share and reset factory flag
794 bool* BleKeyboard::flgRstBUG() {
795         return &rstBUG;
796 }
797
798 void BleKeyboard::flgRstBUG(bool flg) {
799         rstBUG = flg;
800
801         int written = rstBUG * 128 + identify * 64 + wasd * 32 + keys[4] * 16 + keys[0] * 8 +
              ↪  keys[1] * 4 + keys[2] * 2 + keys[1];
802   stateCharacteristic->setValue(written);
803 }
804
805 // Share mode flags
806 bool* BleKeyboard::flgIdentify() {
807         return &identify;
808 }
809
810 void BleKeyboard::flgIdentify(bool flg) {
811         identify = flg;
812
813         int written = rstBUG * 128 + identify * 64 + wasd * 32 + keys[4] * 16 + keys[0] * 8 +
              ↪  keys[1] * 4 + keys[2] * 2 + keys[1];
814   stateCharacteristic->setValue(written);
815 }
816
817 bool* BleKeyboard::flgWASD() {
```

```
818         return &wasd;
819 }
820
821 void BleKeyboard::flgWASD(bool flg) {
822         wasd = flg;
823
824         int written = rstBUG * 128 + identify * 64 + wasd * 32 + keys[4] * 16 + keys[0] * 8 +
              ↪  keys[1] * 4 + keys[2] * 2 + keys[1];
825   stateCharacteristic->setValue(written);
826 }
827
828 bool* BleKeyboard::cirKeys() {
829         return &keys[0];
830 }
831
832 void BleKeyboard::cirKeys(bool* k) {
833         for (int i = 0; i < 5; i++) {
834                 keys[i] = *(k+i);
835         }
836
837         int written = rstBUG * 128 + identify * 64 + wasd * 32 + keys[4] * 16 + keys[0] * 8 +
              ↪  keys[1] * 4 + keys[2] * 2 + keys[1];
838   stateCharacteristic->setValue(written);
839 }
```

Header

```
106 class BleKeyboard : public Print, public BLEServerCallbacks, public BLECharacteristicCallbacks
107 {
108 private:
109   BLEHIDDevice* hid;
110   BLECharacteristic* inputKeyboard;
111   BLECharacteristic* outputKeyboard;
112   BLECharacteristic* inputMediaKeys;
113   BLECharacteristic* keyCharacteristic;
114   BLECharacteristic* indicatorCharacteristic;
115   BLECharacteristic* timeoutCharacteristic;
116   BLECharacteristic* stateCharacteristic;

      ...

132 public:

      ...

154   int* getKeybind();
155   int* getTimeout();
156   int* getColour();
157   int* getLayout();
158   void setKeybind(int* k);
159   void setTimeout(int* t);
160   void setColour(int* c);
161   void setLayout(int* l);
162   bool* flgRstTimer();
163   void flgRstTimer(bool flg);
164   bool* flgRstBUG();
```

```
165     void flgRstBUG(bool flg);
166     bool* flgIdentify();
167     void flgIdentify(bool flg);
168     bool* flgWASD();
169     void flgWASD(bool flg);
170     bool* cirKeys();
171     void cirKeys(bool* k);
```

Appendix G

PC Software code

Below, the code can be found for the PC Software. It can be split into two parts: the backend written in C++, and the frontend / GUI design written with QML. The program was designed and compiled using Qt (version 5.15.2 with the WinGW compiler). The code isn't clean, since it is just used as a proof of concept for the BUG configuration over BLE. Since the main program is practically the same as the example provided by Qt, only the files changed are included in the Appendix. All other files can be found in the Github of this project: https://github.com/jornvdl/BUG.

G.1 Backend

These files (Devicehandler.cpp and its header) are responsible for providing and processing the data to the GUI.

Devicehandler.cpp

```
1  /****************************************************************************
2  **
3  ** Copyright (C) 2017 The Qt Company Ltd.
4  ** Contact: https://www.qt.io/licensing/
5  **
6  ** This file is part of the examples of the QtBluetooth module of the Qt Toolkit.
7  **
8  ** $QT_BEGIN_LICENSE:BSD$
9  ** Commercial License Usage
10 ** Licensees holding valid commercial Qt licenses may use this file in
11 ** accordance with the commercial license agreement provided with the
12 ** Software or, alternatively, in accordance with the terms contained in
13 ** a written agreement between you and The Qt Company. For licensing terms
14 ** and conditions see https://www.qt.io/terms-conditions. For further
15 ** information use the contact form at https://www.qt.io/contact-us.
16 **
17 ** BSD License Usage
18 ** Alternatively, you may use this file under the terms of the BSD license
19 ** as follows:
20 **
21 ** "Redistribution and use in source and binary forms, with or without
22 ** modification, are permitted provided that the following conditions are
```

```
23  ** met:
24  **   * Redistributions of source code must retain the above copyright
25  **     notice, this list of conditions and the following disclaimer.
26  **   * Redistributions in binary form must reproduce the above copyright
27  **     notice, this list of conditions and the following disclaimer in
28  **     the documentation and/or other materials provided with the
29  **     distribution.
30  **   * Neither the name of The Qt Company Ltd nor the names of its
31  **     contributors may be used to endorse or promote products derived
32  **     from this software without specific prior written permission.
33  **
34  **
35  ** THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
      ↪ CONTRIBUTORS
36  ** "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
37  ** LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
      ↪ FOR
38  ** A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
      ↪ COPYRIGHT
39  ** OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
      ↪ INCIDENTAL,
40  ** SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
41  ** LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
      ↪ USE,
42  ** DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
      ↪ ANY
43  ** THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
44  ** (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
      ↪ USE
45  ** OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
46  **
47  ** $QT_END_LICENSE$
48  **
49  ****************************************************************************/
50
51  #include "devicehandler.h"
52  #include "deviceinfo.h"
53  #include <QtEndian>
54  #include <QRandomGenerator>
55  #include <QTextStream>
56  #include <QKeyEvent>
57  #include <QList>
58
59
60  QString gattUUID = "0ba682ae-4f1f-4e9b-be2a-809c224540fd";
61  QString bindUUID = "3e7f5770-d6b7-4709-9b4b-951c63f97aaa";
62  QString indcUUID = "6f1f3Ce2-cb88-4c5a-9ba1-6e19369b8bbb";
63  QString timeUUID = "6de0e9a1-7f07-4e64-81b0-a8ca334bcccc";
64  QString settUUID = "99503c7d-6924-413c-bb7d-db7e913fbddd";
65
66  QLowEnergyCharacteristic bindChar;
67  QLowEnergyCharacteristic layoutChar;
68  QLowEnergyCharacteristic timeoutChar;
69  QLowEnergyCharacteristic infoChar;
```

```
70
71  QTextStream out(stdout);
72
73  int ___timeout;
74  bool ___layout[4];
75  int ___mode;
76  int ___colour;
77
78  bool flgID;
79
80  bool BLEUpdated = true;
81
82  void DeviceHandler::fetchBLEData() const
83  {
84      // layout
85      QString tempresult;
86      QByteArray a = layoutChar.value();
87      int result;
88      if (a.isEmpty()) {
89          out << "None" << Qt::endl;
90          ___layout[0] = 0;
91          ___layout[1] = 0;
92          ___layout[2] = 0;
93          ___layout[3] = 0;
94      }
95
96      bool valid;
97      QString templayout;
98      tempresult += a.toHex();
99      out << "Tempresult layout: " << tempresult << Qt::endl;
100     templayout = tempresult.right(1);
101     result = templayout.toInt(&valid,16);
102     out << "            layout:" << result << Qt::endl;
103     ___layout[0] = (result & 0b0001) == 0b0001;
104     ___layout[1] = (result & 0b0010) == 0b0010;
105     ___layout[2] = (result & 0b0100) == 0b0100;
106     ___layout[3] = (result & 0b1000) == 0b1000;
107
108     // colour
109     QString tempcolour;
110     tempcolour = tempresult.right(8);
111     tempcolour = tempcolour.left(6);
112     ___colour = tempcolour.toInt(&valid, 16);
113
114     a.clear();
115     a += timeoutChar.value();
116     if (a.isEmpty()) {
117         out << "None" << Qt::endl;
118         ___timeout = 0 ;
119     }
120
121     tempresult.clear();
122     tempresult += a.toHex();
123     tempresult = tempresult.left(4);
```

```
124     out << "Tempresult timeout: " << tempresult << Qt::endl;
125     ___timeout = tempresult.toInt(&valid,16);
126 
127     out << " timeout: " << ___timeout << Qt::endl;
128 
129     // Add mode read-out
130     a.clear();
131     a += infoChar.value();
132     if (a.isEmpty()) {
133         out << "None" << Qt::endl;
134         ___mode = 0 ;
135     }
136 
137     tempresult.clear();
138     tempresult += a.toHex();
139     tempresult = tempresult.left(2);
140     out << "Tempresult mode: " << tempresult << Qt::endl;
141     ___mode = tempresult.toInt(&valid,16);
142 
143 }
144 
145 QString DeviceHandler::getKeybind() const
146 {
147     QByteArray a = bindChar.value();
148     int result;
149     QString tempresult;
150     if (a.isEmpty()) {
151         out << "None" << Qt::endl;
152         return "None";
153     }
154 
155     tempresult = a;
156     bool valid;
157     tempresult += a.toHex();
158     tempresult.remove(0,1);
159     tempresult.remove(2,6);
160     result = tempresult.toInt(&valid,16);
161 
162     // Lookup table to convert some values to icons (space, arrows, etc)
163     QString strResult;
164     switch (result) {
165     case 32:
166         strResult = "Space";
167         break;
168     case 80:
169         strResult = "l Ctrl";
170         break;
171     case 81:
172         strResult = "l Shift";
173         break;
174     case 82:
175         strResult = "l Alt";
176         break;
177     case 84:
```

```
178         strResult = "r Ctrl";
179         break;
180     case 85:
181         strResult = "r Shift";
182         break;
183     case 86:
184         strResult = "r Alt";
185         break;
186     case 215:
187         strResult = "→";
188         break;
189     case 216:
190         strResult = "←";
191         break;
192     case 217:
193         strResult = "↓";
194         break;
195     case 218:
196         strResult = "↑";
197         break;
198     case 177:
199         strResult = "Escape";
200         break;
201     case 178:
202         strResult = "Bckspc";
203         break;
204     case 0xB3:
205         strResult = "Tab";
206         break;
207     case 0xD1:
208         strResult = "Insert";
209         break;
210     case 0xD2:
211         strResult = "Tab";
212         break;
213     case 0xD3:
214         strResult = "PgUp";
215         break;
216     case 0xD4:
217         strResult = "Delete";
218         break;
219     case 0xD5:
220         strResult = "End";
221         break;
222     case 0xD6:
223         strResult = "PgDown";
224         break;
225     default:
226         char tmpchar = result;
227         strResult = tmpchar;
228         break;
229     }
230
231     out << "Keybind: int=" << result << " char=" << strResult << Qt::endl;
```

```
232
233     return strResult;
234 }
235
236 int DeviceHandler::getColour() const
237 {
238     // if value updated in BLE, fetch it.
239     if (BLEUpdated) {
240         fetchBLEData();
241         BLEUpdated=false;
242     }
243
244     out << "Colour = " << ___colour << Qt::endl;
245
246     return ___colour;
247 }
248
249 bool DeviceHandler::getLayout0() const
250 {
251     // if value updated in BLE, fetch it.
252     if (BLEUpdated) {
253         fetchBLEData();
254         BLEUpdated=false;
255     }
256
257     out << "Layout[0] = " << ___layout[0] << Qt::endl;
258
259     return ___layout[0];
260 }
261
262 bool DeviceHandler::getLayout1() const
263 {
264     // if value updated in BLE, fetch it.
265     if (BLEUpdated) {
266         fetchBLEData();
267         BLEUpdated=false;
268     }
269
270     out << "Layout[1] = " << ___layout[1] << Qt::endl;
271
272     return ___layout[1];
273 }
274
275 bool DeviceHandler::getLayout2() const
276 {
277     // if value updated in BLE, fetch it.
278     if (BLEUpdated) {
279         fetchBLEData();
280         BLEUpdated=false;
281     }
282
283     out << "Layout[2] = " << ___layout[2] << Qt::endl;
284
285     return ___layout[2];
```

```
286 }
287
288 bool DeviceHandler::getLayout3() const
289 {
290     // if value updated in BLE, fetch it.
291     if (BLEUpdated) {
292         fetchBLEData();
293         BLEUpdated=false;
294     }
295
296     out << "Layout[3] = " << ___layout[3] << Qt::endl;
297
298     return ___layout[3];
299 }
300
301 int DeviceHandler::getTimeout() const
302 {
303     if (BLEUpdated) {
304         fetchBLEData();
305         BLEUpdated=false;
306     }
307
308     out << "Timeout = " << ___timeout << Qt::endl;
309     return ___timeout;
310 }
311
312 int DeviceHandler::getMode() const
313 {
314     if (BLEUpdated) {
315         fetchBLEData();
316         BLEUpdated=false;
317     }
318
319     out << "Mode = " << ___mode << Qt::endl;
320     return ___mode;
321 }
322
323 QString DeviceHandler::getTextMode() const
324 {
325     if (BLEUpdated) {
326         fetchBLEData();
327         BLEUpdated=false;
328     }
329
330     QString output;
331
332     switch (___mode) {
333     case 0:
334         output = "↑←↓→";
335         break;
336     case 1:
337         output = "↑←↓→␣";
338         break;
339     case 2:
```

```
340             output = "WASD";
341             break;
342         case 3:
343             output = "WASD␣";
344             break;
345         case 4:
346             output = "Presenter";
347             break;
348         case 5:
349             output = "Custom key";
350             break;
351         case 6:
352             output = "Identify";
353             break;
354         default:
355             output = "other";
356             break;
357         }
358
359         return output;
360     }
361
362     bool DeviceHandler::getIDflg() const
363     {
364         return flgID;
365     }
366
367     void DeviceHandler::updateLayout(int i)
368     {
369         ___layout[i] = !___layout[i];
370
371         emit dataChanged();
372         out << "layout" << i << " set to " << ___layout[i] << Qt::endl;
373
374         sendLayout();
375     }
376
377     void DeviceHandler::updateColour(QString from_gui)
378     {
379         out << "from_gui: " << from_gui << Qt::endl;
380         ___colour = from_gui.toInt(NULL, 16);
381
382         sendLayout();
383
384         emit dataChanged();
385     }
386
387     void DeviceHandler::updateMode(int k)
388     {
389         char mout[1];
390
391         if (k < 6) ___mode= k;
392
393
```

```
394     switch (___mode) {
395     case 0: // Arrows
396        out << "mode: arrows" << Qt::endl;
397        mout[0] = 0x0F;
398         break;
399     case 1: // Arrows with space
400         out << "mode: arrows w/ space" << Qt::endl;
401        mout[0] = 0x1F;
402         break;
403     case 2: // WASD
404         out << "mode: WASD" << Qt::endl;
405        mout[0] = 0x2F;
406         break;
407     case 3: // WASD with space
408         out << "mode: WASD w/ space" << Qt::endl;
409        mout[0] = 0x3F;
410         break;
411     case 4: // Arrow left and right (presenter)
412         out << "mode: presenter" << Qt::endl;
413        mout[0] = 0x05;
414         break;
415     case 7: // Indicate, use current settings
416         // stuff
417         mout[0] = ___mode;
418         break;
419     default: // All other options (eg custom key): no flags
420         out << "mode: custom" << Qt::endl;
421         mout[0] = 0x00;
422     }
423
424     if (k == 7) { // If indicate, flip flag and update value
425         out << "mode: identify" << Qt::endl;
426         flgID = !flgID;
427         mout[0] =+ flgID * 0x40;
428         out << "Identify BUG! ID=" << flgID << Qt::endl;
429     }
430     else if (k == 6) {
431         mout[0] = 0x80;
432         out << "Reset BUG! to factory settings" << Qt::endl;
433     }
434
435
436
437     QByteArray output = QByteArray::fromRawData(mout,1);
438     m_service->writeCharacteristic(infoChar, output, QLowEnergyService::WriteWithResponse);
439
440     setInfo("Use mode updated.");
441     emit dataChanged();
442
443 }
444
445
446 DeviceHandler::DeviceHandler(QObject *parent) :
447     BluetoothBaseClass(parent)
```

```
448 {
449
450 }
451
452 void DeviceHandler::setAddressType(AddressType type)
453 {
454     switch (type) {
455     case DeviceHandler::AddressType::PublicAddress:
456         m_addressType = QLowEnergyController::PublicAddress;
457         break;
458     case DeviceHandler::AddressType::RandomAddress:
459         m_addressType = QLowEnergyController::RandomAddress;
460         break;
461     }
462 }
463
464 DeviceHandler::AddressType DeviceHandler::addressType() const
465 {
466     if (m_addressType == QLowEnergyController::RandomAddress)
467         return DeviceHandler::AddressType::RandomAddress;
468
469     return DeviceHandler::AddressType::PublicAddress;
470 }
471
472 void DeviceHandler::setDevice(DeviceInfo *device)
473 {
474     clearMessages();
475     m_currentDevice = device;
476
477
478
479     // Disconnect and delete old connection
480     if (m_control) {
481         m_control->disconnectFromDevice();
482         delete m_control;
483         m_control = nullptr;
484     }
485
486     // Create new controller and connect it if device available
487     if (m_currentDevice) {
488
489         // Make connections
490         //! [Connect-Signals-1]
491         m_control = QLowEnergyController::createCentral(m_currentDevice->getDevice(), this);
492         //! [Connect-Signals-1]
493         m_control->setRemoteAddressType(m_addressType);
494         //! [Connect-Signals-2]
495         connect(m_control, &QLowEnergyController::serviceDiscovered,
496                 this, &DeviceHandler::serviceDiscovered);
497         connect(m_control, &QLowEnergyController::discoveryFinished,
498                 this, &DeviceHandler::serviceScanDone);
499
500         connect(m_control, static_cast<void (QLowEnergyControl-
            ↪ ler::*)(QLowEnergyController::Error)>(&QLowEnergyController::error),
```

```
501             this, [this](QLowEnergyController::Error error) {
502         Q_UNUSED(error);
503         setError("Cannot connect to remote device.");
504     });
505     connect(m_control, &QLowEnergyController::connected, this, [this]() {
506         setInfo("Controller connected. Search services...");
507         m_control->discoverServices();
508     });
509     connect(m_control, &QLowEnergyController::disconnected, this, [this]() {
510         setError("LowEnergy controller disconnected");
511     });
512
513     // Connect
514     m_control->connectToDevice();
515     //! [Connect-Signals-2]
516 }
517 }
518
519 void DeviceHandler::sendKeybind(QString value)
520 {
521     out << "  Send keybind: " << value << Qt::endl;
522     int output = 0x00;
523     if (value == "Enter") output = 0xE0;
524     else if (value == "←") output = 0xD8;
525     else if (value == "↑") output = 0xDA;
526     else if (value == "→") output = 0xD7;
527     else if (value == "↓") output = 0xD9;
528     else if (value == "Ctrl") output = 0x84;
529     else if (value == "Alt") output = 0x86;
530     else if (value == "Shift") output = 0x81;
531     else if (value == "Escape") output = 0xB1;
532     else if (value == "Space") output = 0x20;
533     else { //convert to char and then to int
534         QChar tc = value.at(0);
535         tc = tc.toLower();
536         out << "  Converted character: " << tc << Qt::endl;
537         output = tc.toLatin1();
538     }
539
540     out << "  Converted integer:   " << output << Qt::endl;
541
542     char chartmp[1];
543     chartmp[0] = output;
544     QByteArray number = QByteArray::fromRawData(chartmp,1);
545     m_service->writeCharacteristic(bindChar, number,
        ↪ QLowEnergyService::WriteWithResponse);
546
547     out << "Written keybind (sent: " << output << ")" << Qt::endl;
548     setInfo("Key bind updated.");
549     emit dataChanged();
550 }
551
552 void DeviceHandler::sendTimeout(QString value)
553 {
```

```
554     // Convert value to int
555     int toSend = value.toInt(NULL, 10);
556     out << "Timer to send: " << toSend << Qt::endl;
557 
558     char charSend[2];
559     charSend[0] = (toSend >> 8);
560     out << "charSend[0] = " << int(charSend[0]) << Qt::endl;
561     charSend[1] = (toSend);
562     out << "charSend[1] = " << int(charSend[1]) << Qt::endl;
563 
564     QByteArray number;
565     number = QByteArray::fromRawData(charSend,2);
566     m_service->writeCharacteristic(timeoutChar, number,
        ↪ QLowEnergyService::WriteWithResponse);
567 
568     out << "Raw timeout send: " << number.toHex() << Qt::endl;
569     setInfo("Timeout value updated.");
570 
571     ___timeout = toSend;
572 
573     emit dataChanged();
574 }
575 
576 void DeviceHandler::sendLayout()
577 {
578     // fetch colour
579     //int tmpcol;
580     //tmpcol = 0xff0000; // temporary
581 
582     // fetch layout
583     int tmplay = 0x00;
584     tmplay = 8*___layout[3] + 4*___layout[2] + 2*___layout[1] + ___layout[0];
585 
586     //uint tmp = (tmpcol << 8) + tmplay;
587     QByteArray number;
588     //number.setNum(tmp,16);
589 
590     // Split colour into R,G,B value
591 
592     char chartmp[4];
593     chartmp[0] = (___colour & 0xff0000) >> 16;
594     chartmp[1] = (___colour & 0x00ff00) >> 8;
595     chartmp[2] = (___colour & 0x0000ff);
596     chartmp[3] = tmplay;
597 
598     number = QByteArray::fromRawData(chartmp,4);
599 
600     //number.append(tmpcol);
601     //number.fromRawData(tmp,2);
602     m_service->writeCharacteristic(layoutChar, number,
        ↪ QLowEnergyService::WriteWithResponse);
603     out << "Written layout (sent: " << number << ")" << Qt::endl;
604     out << "     layout tmp (set: " << tmplay << ")" << Qt::endl;
605     setInfo("Layout & colour updated.");
```

```
606     emit dataChanged();
607 }
608
609 //! [Filter HeartRate service 1]
610 void DeviceHandler::serviceDiscovered(const QBluetoothUuid &gatt)
611 {
612     //if (gatt == QBluetoothUuid(QBluetoothUuid::HeartRate)) {
613     if (gatt == QBluetoothUuid(gattUUID)) {
614         setInfo("Key bind service discovered. Waiting for service scan to be done...");
615         m_foundGATTService = true;
616         printf("UUID found! gatt == QBluetoothUuid(gattUUID)\n");
617     }
618 }
619 //! [Filter HeartRate service 1]
620
621 void DeviceHandler::serviceScanDone()
622 {
623     setInfo("Service scan done.");
624
625     // Delete old service if available
626     if (m_service) {
627         delete m_service;
628         m_service = nullptr;
629     }
630
631 //! [Filter HeartRate service 2]
632     // If heartRateService found, create new service
633     if (m_foundGATTService)
634         m_service = m_control->createServiceObject(QBluetoothUuid(gattUUID), this);
635
636     if (m_service) {
637         connect(m_service, &QLowEnergyService::stateChanged, this,
            ↪ &DeviceHandler::serviceStateChanged);
638         //connect(m_service, &QLowEnergyService::characteristicChanged, this,
            ↪ &DeviceHandler::updateHeartRateValue);
639         connect(m_service, &QLowEnergyService::descriptorWritten, this,
            ↪ &DeviceHandler::confirmedDescriptorWrite);
640         m_service->discoverDetails();
641         out << "m_service found!" << Qt::endl;
642     } else {
643         setError("Heart Rate Service not found.");
644     }
645 //! [Filter HeartRate service 2]
646 }
647
648 // Service functions
649 //! [Find HRM characteristic]
650 void DeviceHandler::serviceStateChanged(QLowEnergyService::ServiceState s)
651 {
652     out << "Service changed!" << Qt::endl;
653     switch (s) {
654     case QLowEnergyService::DiscoveringServices:
655         setInfo(tr("Discovering services...\n"));
656         break;
```

```
657     case QLowEnergyService::ServiceDiscovered:
658     {
659         setInfo(tr("Settings loaded."));
660         bindChar = m_service->characteristic(QBluetoothUuid(bindUUID));
661         layoutChar = m_service->characteristic(QBluetoothUuid(indcUUID));
662         timeoutChar = m_service->characteristic(QBluetoothUuid(timeUUID));
663         infoChar = m_service->characteristic(QBluetoothUuid(settUUID));
664         if (!bindChar.isValid()) {
665             printf("Binding characteristic not found.\n");
666             break;
667         }
668         else {
669             printf("Binding characteristic valid.\n");
670         }
671         if (!layoutChar.isValid()) {
672             printf("Layout characteristic not found.\n");
673             break;
674         }
675         else {
676             printf("Layout characteristic valid.\n");
677         }
678         if (!timeoutChar.isValid()) {
679             printf("Timeout characteristic not found.\n");
680             break;
681         }
682         else {
683             printf("Timeout characteristic valid.\n");
684         }
685         if (!infoChar.isValid()) {
686             printf("Settings characteristic not found.\n");
687             break;
688         }
689         else {
690             printf("Settings characteristic valid.\n");
691         }
692 
693         // This below is weird, but also key
694 //         m_bindingDesc =
    ↪ bindChar.descriptor(QBluetoothUuid::ClientCharacteristicConfiguration);
695 //         if (m_bindingDesc.isValid())
696 //             printf("Binding data valid!\n");
697 ////             m_service->writeDescriptor(m_bindingDesc, QByteArray::fromHex("0100"));
698 //         else
699 //             printf("Binding data invalid!\n");
700 
701 //         QByteArray a = bindChar.value();
702 //         QString result;
703 //         if (a.isEmpty()) {
704 //             result = QStringLiteral("<none>");
705 //             out << "None" << Qt::endl;
706 //             break;
707 //         }
708 //         result = a;
709 //         result += QLatin1Char('\n');
```

```
710 //        result += a.toHex();
711
712 //        out << result << Qt::endl;
713
714         break;
715     }
716     default:
717         //nothing for now
718         break;
719     }
720
721     emit aliveChanged();
722 }
723 //! [Find HRM characteristic]
724
725
726 void DeviceHandler::confirmedDescriptorWrite(const QLowEnergyDescriptor &d, const
      ↪ QByteArray &value)
727 {
728     out << "Descriptor changed!" << Qt::endl;
729     if (d.isValid() && d == m_bindingDesc && value == QByteArray::fromHex("0000")) {
730         //disabled notifications -> assume disconnect intent
731         m_control->disconnectFromDevice();
732         delete m_service;
733         m_service = nullptr;
734     }
735 }
736
737 void DeviceHandler::disconnectService()
738 {
739     m_foundGATTService = false;
740
741     //disable notifications
742     if (m_bindingDesc.isValid() && m_service
743             && m_bindingDesc.value() == QByteArray::fromHex("0100")) {
744         m_service->writeDescriptor(m_bindingDesc, QByteArray::fromHex("0000"));
745     } else {
746         if (m_control)
747             m_control->disconnectFromDevice();
748
749         delete m_service;
750         m_service = nullptr;
751     }
752 }
753
754 bool DeviceHandler::alive() const
755 {
756
757     if (m_service)
758         return m_service->state() == QLowEnergyService::ServiceDiscovered;
759
760     return false;
761 }
```

Devicehandler.h

```
/****************************************************************************
**
** Copyright (C) 2017 The Qt Company Ltd.
** Contact: https://www.qt.io/licensing/
**
** This file is part of the examples of the QtBluetooth module of the Qt Toolkit.
**
** $QT_BEGIN_LICENSE:BSD$
** Commercial License Usage
** Licensees holding valid commercial Qt licenses may use this file in
** accordance with the commercial license agreement provided with the
** Software or, alternatively, in accordance with the terms contained in
** a written agreement between you and The Qt Company. For licensing terms
** and conditions see https://www.qt.io/terms-conditions. For further
** information use the contact form at https://www.qt.io/contact-us.
**
** BSD License Usage
** Alternatively, you may use this file under the terms of the BSD license
** as follows:
**
** "Redistribution and use in source and binary forms, with or without
** modification, are permitted provided that the following conditions are
** met:
**   * Redistributions of source code must retain the above copyright
**     notice, this list of conditions and the following disclaimer.
**   * Redistributions in binary form must reproduce the above copyright
**     notice, this list of conditions and the following disclaimer in
**     the documentation and/or other materials provided with the
**     distribution.
**   * Neither the name of The Qt Company Ltd nor the names of its
**     contributors may be used to endorse or promote products derived
**     from this software without specific prior written permission.
**
**
** THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
** "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
** LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
** A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
** OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
** SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
** LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
** DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
** THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
** (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
** OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
```

```
46  **
47  ** $QT_END_LICENSE$
48  **
49  ****************************************************************************/
50
51  #ifndef DEVICEHANDLER_H
52  #define DEVICEHANDLER_H
53
54  #include "bluetoothbaseclass.h"
55
56  #include <QDateTime>
57  #include <QTimer>
58  #include <QVector>
59  #include <QKeyEvent>
60  #include <QWidget>
61  #include <QtGui>
62
63  #include <QLowEnergyController>
64  #include <QLowEnergyService>
65
66  class DeviceInfo;
67
68  class DeviceHandler : public BluetoothBaseClass
69  {
70      Q_OBJECT
71      Q_PROPERTY(QString keybind READ getKeybind NOTIFY dataChanged)
72      Q_PROPERTY(bool getIDflg READ getIDflg NOTIFY dataChanged)
73      Q_PROPERTY(int colour READ getColour NOTIFY dataChanged)
74      Q_PROPERTY(int layout0 READ getLayout0 NOTIFY dataChanged)
75      Q_PROPERTY(int layout1 READ getLayout1 NOTIFY dataChanged)
76      Q_PROPERTY(int layout2 READ getLayout2 NOTIFY dataChanged)
77      Q_PROPERTY(int layout3 READ getLayout3 NOTIFY dataChanged)
78      Q_PROPERTY(int timeout READ getTimeout NOTIFY dataChanged)
79      Q_PROPERTY(QString mode READ getTextMode NOTIFY dataChanged)
80
81      Q_PROPERTY(AddressType addressType READ addressType WRITE setAddressType)
82
83  public:
84      enum class AddressType {
85          PublicAddress,
86          RandomAddress
87      };
88      Q_ENUM(AddressType)
89
90      DeviceHandler(QObject *parent = nullptr);
91
92      void setDevice(DeviceInfo *device);
93      void setAddressType(AddressType type);
94      AddressType addressType() const;
95
96      bool alive() const;
97
98
99      // Data
```

```
100     QString getKeybind() const;
101     int getColour() const;
102     bool getLayout0() const;
103     bool getLayout1() const;
104     bool getLayout2() const;
105     bool getLayout3() const;
106     int getMode() const;
107     QString getTextMode() const;
108     int getTimeout() const;
109     bool getIDflg() const;
110
111 protected:
112
113 signals:
114     void aliveChanged();
115     void statsChanged();
116     void dataChanged();
117
118 public slots:
119     void sendKeybind(QString value);
120     void sendTimeout(QString value);
121     void sendLayout();
122     void disconnectService();
123     void fetchBLEData() const;
124     void updateLayout(int i);
125     void updateColour(QString from_gui);
126     void updateMode(int k);
127
128 private:
129     //QLowEnergyController
130     void serviceDiscovered(const QBluetoothUuid &);
131     void serviceScanDone();
132
133     bool m_foundGATTService;
134
135     //QLowEnergyService
136     void serviceStateChanged(QLowEnergyService::ServiceState s);
137     void confirmedDescriptorWrite(const QLowEnergyDescriptor &d,
138                                   const QByteArray &value);
139
140 private:
141
142     QLowEnergyController *m_control = nullptr;
143     QLowEnergyService *m_service = nullptr;
144     QLowEnergyDescriptor m_bindingDesc;
145     DeviceInfo *m_currentDevice = nullptr;
146
147     QLowEnergyController::RemoteAddressType m_addressType =
        ↪ QLowEnergyController::PublicAddress;
148
149 };
150
151 #endif // DEVICEHANDLER_H
```

G.2 Frontend / GUI

Measure.qml

This code generates the configuration page per BUG.

```
1  /****************************************************************************
2  **
3  ** Copyright (C) 2017 The Qt Company Ltd.
4  ** Contact: https://www.qt.io/licensing/
5  **
6  ** This file is part of the examples of the QtBluetooth module of the Qt Toolkit.
7  **
8  ** $QT_BEGIN_LICENSE:BSD$
9  ** Commercial License Usage
10 ** Licensees holding valid commercial Qt licenses may use this file in
11 ** accordance with the commercial license agreement provided with the
12 ** Software or, alternatively, in accordance with the terms contained in
13 ** a written agreement between you and The Qt Company. For licensing terms
14 ** and conditions see https://www.qt.io/terms-conditions. For further
15 ** information use the contact form at https://www.qt.io/contact-us.
16 **
17 ** BSD License Usage
18 ** Alternatively, you may use this file under the terms of the BSD license
19 ** as follows:
20 **
21 ** "Redistribution and use in source and binary forms, with or without
22 ** modification, are permitted provided that the following conditions are
23 ** met:
24 **   * Redistributions of source code must retain the above copyright
25 **     notice, this list of conditions and the following disclaimer.
26 **   * Redistributions in binary form must reproduce the above copyright
27 **     notice, this list of conditions and the following disclaimer in
28 **     the documentation and/or other materials provided with the
29 **     distribution.
30 **   * Neither the name of The Qt Company Ltd nor the names of its
31 **     contributors may be used to endorse or promote products derived
32 **     from this software without specific prior written permission.
33 **
34 **
35 ** THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   ↪ CONTRIBUTORS
36 ** "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
37 ** LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
   ↪ FOR
38 ** A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
   ↪ COPYRIGHT
39 ** OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
   ↪ INCIDENTAL,
40 ** SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
41 ** LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
   ↪ USE,
42 ** DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
   ↪ ANY
```

```
43  ** THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
44  ** (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
      ↪ USE
45  ** OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
46  **
47  ** $QT_END_LICENSE$
48  **
49  ****************************************************************************/
50
51  import QtQuick 2.11
52  import QtQuick.Controls 2.4
53  GamePage {
54      id: measurePage
55
56      errorMessage: deviceHandler.error
57      infoMessage: deviceHandler.info
58
59      property real ___timeCounter: 0
60      property real ___maxTimeCount: 60
61
62      property real ___rawbatt: 100
63      property real ___rawtimer: 300
64      property real ___newkeybind: 216
65
66      property string ___sleeptimer: qsTr("%1s").arg(___rawtimer)
67      property real minHR: 0
68      property string ___modeview: qsTr("↑←↓→␣")
69      property string ___batteryperc: qsTr("N/A")
70
71      function close() {
72          //deviceHandler.stopMeasurement();
73          deviceHandler.disconnectService();
74          app.prevPage();
75      }
76
77
78      function updateLayout() {
79          deviceHandler.sendLayout();
80      }
81
82      function layout0() {
83          deviceHandler.updateLayout(0)
84      }
85
86      function layout1() {
87          deviceHandler.updateLayout(1)
88      }
89
90      function layout2() {
91          deviceHandler.updateLayout(2)
92      }
93
94      function layout3() {
95          deviceHandler.updateLayout(3)
```

```
96      }
97
98
99      Column {
100         anchors.horizontalCenter: parent.horizontalCenter
101         anchors.verticalCenter: parent.verticalCenter
102         anchors.verticalCenterOffset: -parent.height*0.4
103         spacing: GameSettings.fieldHeight * 0.2
104
105         Rectangle {
106             id: infobox
107             anchors.horizontalCenter: parent.horizontalCenter
108             width: Math.min(measurePage.width, measurePage.height-GameSettings.fieldHeight*4)
                ↪  - 2*GameSettings.fieldMargin
109             height: width*0.5
110             radius: width*0.05
111             color: GameSettings.viewColor
112
113             Text {
114                 id: conftext
115                 anchors.centerIn: parent
116                 anchors.horizontalCenterOffset: -parent.width*0.20
117                 horizontalAlignment: Text.AlignHLeft
118                 verticalAlignment: Text.AlignTop
119                 width: parent.width * 0.55
120                 text: "\nCurrent BUG configuration\n Key bind:
                    ↪  \t"+deviceHandler.keybind.toUpperCase()+"\n
                    ↪  Sleeptimer:\t"+deviceHandler.timeout+"s\n Mode:
                    ↪  \t"+deviceHandler.mode+"\n Battery:     \t"+___batteryperc
121                 color: GameSettings.textColor
122                 fontSizeMode: Text.Fit
123                 minimumPixelSize: 10
124                 font.pixelSize: GameSettings.mediumFontSize
125             }
126
127             Image {
128                 id: bugfront
129                 anchors.centerIn: parent
130                 anchors.horizontalCenterOffset: parent.width*0.28
131                 height: parent.height * 0.9
132                 width: height / 1.143
133                 source: "images/frontview.png"
134                 smooth: true
135                 antialiasing: true
136
137                 }
138         }
139
140         Text {
141             id: keybindinfotext
142             anchors.left: infobox.left
143             anchors.leftMargin: GameSettings.fieldMargin * 0.4
144             anchors.top: infobox.bottom
145             anchors.topMargin: GameSettings.fieldMargin * 0.25
```

```
146             height: GameSettings.fieldHeight * 0.75
147             text: qsTr("Set key bind:")
148             horizontalAlignment: Text.AlignLeft
149             color: GameSettings.textColor
150             fontSizeMode: Text.Fit
151             minimumPixelSize: 10
152             font.pixelSize: GameSettings.mediumFontSize
153         }
154
155         Text {
156             id: colourinfotext
157             anchors.left: keybindinfotext.left
158             anchors.top: keybindinfotext.bottom
159             anchors.topMargin: GameSettings.fieldMargin * 0.05
160             height: GameSettings.fieldHeight * 0.75
161             text: qsTr("Set colour:")
162             horizontalAlignment: Text.AlignLeft
163             color: GameSettings.textColor
164             fontSizeMode: Text.Fit
165             minimumPixelSize: 10
166             font.pixelSize: GameSettings.mediumFontSize
167         }
168
169         Text {
170             id: layoutinfotext
171             anchors.left: keybindinfotext.left
172             anchors.top: colourinfotext.bottom
173             anchors.topMargin: GameSettings.fieldMargin * 0.05
174             height: GameSettings.fieldHeight * 0.75
175             text: qsTr("Set layout:")
176             horizontalAlignment: Text.AlignLeft
177             color: GameSettings.textColor
178             fontSizeMode: Text.Fit
179             minimumPixelSize: 10
180             font.pixelSize: GameSettings.mediumFontSize
181         }
182
183         Text {
184             id: sleeptimerinfotext
185             anchors.left: keybindinfotext.left
186             anchors.top: layoutinfotext.bottom
187             anchors.topMargin: GameSettings.fieldMargin * 0.05
188             height: GameSettings.fieldHeight * 0.75
189             text: qsTr("Set sleeptimer:")
190             horizontalAlignment: Text.AlignLeft
191             color: GameSettings.textColor
192             fontSizeMode: Text.Fit
193             minimumPixelSize: 10
194             font.pixelSize: GameSettings.mediumFontSize
195         }
196
197         Text {
198             id: modeinfotext
199             anchors.left: keybindinfotext.left
```

```
200             anchors.top: sleeptimerinfotext.bottom
201             anchors.topMargin: GameSettings.fieldMargin * 0.05
202             height: GameSettings.fieldHeight * 0.75
203             text: qsTr("Set use mode:")
204             horizontalAlignment: Text.AlignLeft
205             color: GameSettings.textColor
206             fontSizeMode: Text.Fit
207             minimumPixelSize: 10
208             font.pixelSize: GameSettings.mediumFontSize
209         }
210
211         Rectangle {
212             id: keybindbtn
213             width: 0.3 * infobox.width
214             height: GameSettings.fieldHeight * 0.6
215             radius: GameSettings.buttonRadius
216             anchors.horizontalCenter: parent.horizontalCenter
217             anchors.horizontalCenterOffset: 0.25*infobox.width
218             anchors.verticalCenter: keybindinfotext.verticalCenter
219             anchors.verticalCenterOffset: -0.2 * height
220             color: modeselect.currentIndex === 5 ? GameSettings.buttonColor :
                ↪ GameSettings.disabledButtonColor
221         }
222
223         TextInput {
224             id: keybindbtntxt
225             width: keybindbtn.width
226             height: keybindbtn.height
227             //radius: GameSettings.buttonRadius
228             anchors.horizontalCenter: keybindbtn.horizontalCenter
229             anchors.verticalCenter: keybindbtn.verticalCenter
230             color: modeselect.currentIndex === 5 ? GameSettings.textColor :
                ↪ GameSettings.disabledTextColor
231             horizontalAlignment: Text.AlignHCenter
232             verticalAlignment: Text.AlignVCenter
233             font.pixelSize: height / 2
234             text: deviceHandler.keybind
235
236             activeFocusOnPress: modeselect.currentIndex === 5 ? true : false
237
238             onFocusChanged:
239                 if (keybindbtntxt.focus) keybindbtntxt.text = "";
240             onTextChanged: {
241                 if (text != "") {
242                     console.log("Escaped keybind by key", keybindbtntxt.text);
243                     keybindbtntxt.focus = false;
244                     if (keybindbtntxt.text.length === 1) keybindbtntxt.text =
                        ↪ keybindbtntxt.text.toUpperCase();
245                     if (keybindbtntxt.text === " ") keybindbtntxt.text = "Space";
246                     deviceHandler.sendKeybind(keybindbtntxt.text);
247                 }
248             }
249
250             }
```

```
251
252         Rectangle {
253             id: colourbtn
254             width: 0.3 * infobox.width
255             height: GameSettings.fieldHeight * 0.6
256             radius: GameSettings.buttonRadius
257             anchors.horizontalCenter: keybindbtn.horizontalCenter
258             anchors.verticalCenter: colourinfotext.verticalCenter
259             anchors.verticalCenterOffset: -0.2 * height
260             color: Qt.rgba((deviceHandler.colour >> 16 & 0xFF) / 255,
261                            (deviceHandler.colour >> 8  & 0xFF) / 255,
262                            (deviceHandler.colour       & 0xFF) / 255);
263         }
264
265         TextInput {
266             id: colourbtntxt
267             width: keybindbtn.width
268             height: keybindbtn.height
269             //radius: GameSettings.buttonRadius
270             anchors.horizontalCenter: colourbtn.horizontalCenter
271             anchors.verticalCenter: colourbtn.verticalCenter
272             color: "white"
273             horizontalAlignment: Text.AlignHCenter
274             verticalAlignment: Text.AlignVCenter
275             font.pixelSize: height / 2
276             text: deviceHandler.colour.toString(16)
277
278
279             validator: RegExpValidator { regExp: /[0-9A-Fa-f]+/ }
280
281             onFocusChanged:
282                 if (colourbtntxt.focus) colourbtntxt.text = "";
283             onTextChanged: {
284                 if (text != "") {
285                     if (text.length === 6) {
286                         colourbtntxt.focus = false;
287                         // TOFIX: Convert to int first
288                         console.log("Color set to","#"+text);
289                         colourbtn.color = "#"+text;
290                         console.log("Lightness:");
291                         deviceHandler.updateColour(text);
292                     }
293                 }
294
295             }
296
297             }
298
299         Layout0Button {
300             id: layoutbtn1
301             width: 0.04 * infobox.width
302             height: width
303             radius: GameSettings.buttonRadius * 0.2
304             anchors.horizontalCenter: colourbtn.horizontalCenter
```

```
305             anchors.verticalCenter: layoutbtn2.verticalCenter
306             anchors.verticalCenterOffset: -1.55 * height
307 
308             onClicked: layout0()
309 
310 
311         }
312 
313         Layout1Button {
314             id: layoutbtn2
315             width: 0.04 * infobox.width
316             height: width
317             radius: GameSettings.buttonRadius * 0.2
318             anchors.right: layoutbtn1.left
319             anchors.rightMargin: 0.55*width
320             anchors.verticalCenter: layoutinfotext.verticalCenter
321             anchors.verticalCenterOffset: 0.25 * height
322 
323             onClicked: layout1()
324         }
325 
326         Layout2Button {
327             id: layoutbtn3
328             width: 0.04 * infobox.width
329             height: width
330             radius: GameSettings.buttonRadius * 0.2
331             anchors.verticalCenter: layoutbtn2.verticalCenter
332             anchors.left: layoutbtn2.right
333             anchors.leftMargin: 0.55*width
334 
335             onClicked: layout2()
336         }
337 
338         Layout3Button {
339             id: layoutbtn4
340             width: 0.04 * infobox.width
341             height: width
342             radius: GameSettings.buttonRadius * 0.2
343             anchors.verticalCenter: layoutbtn2.verticalCenter
344             anchors.left: layoutbtn3.right
345             anchors.leftMargin: 0.55*width
346 
347             onClicked: layout3()
348         }
349 
350         Rectangle {
351             width: 0.3 * infobox.width
352             height: GameSettings.fieldHeight * 0.6
353             radius: GameSettings.buttonRadius
354             anchors.horizontalCenter: keybindbtn.horizontalCenter
355             anchors.verticalCenter: sleeptimerinfotext.verticalCenter
356             anchors.verticalCenterOffset: -0.2 * height
357             color: GameSettings.buttonColor
358         }
```

```
359
360         TextInput {
361             id: timerbtn
362             width: 0.3 * infobox.width
363             height: GameSettings.fieldHeight * 0.6
364             //radius: GameSettings.buttonRadius
365             anchors.horizontalCenter: keybindbtn.horizontalCenter
366             anchors.verticalCenter: sleeptimerinfotext.verticalCenter
367             anchors.verticalCenterOffset: -0.2 * height
368             color: GameSettings.textColor
369             horizontalAlignment: Text.AlignHCenter
370             verticalAlignment: Text.AlignVCenter
371             validator: IntValidator { bottom: 0; top: 65535 }
372             font.pixelSize: height / 2
373             text: deviceHandler.timeout
374         }
375
376         ComboBox {
377             id: modeselect
378             width: keybindbtn.width
379             height: GameSettings.fieldHeight * 0.6
380             anchors.horizontalCenter: keybindbtn.horizontalCenter
381             anchors.verticalCenter: modeinfotext.verticalCenter
382             anchors.verticalCenterOffset: -0.2 * height
383
384             model: ListModel {
385                 id: modeItems
386                 ListElement { text: "Arrows"}
387                 ListElement { text: "Arrows + space"}
388                 ListElement { text: "WASD"}
389                 ListElement { text: "WASD + space"}
390                 ListElement { text: "Presenter mode"}
391                 ListElement { text: "Custom key"}
392             }
393             onCurrentIndexChanged: {
394                 deviceHandler.updateMode(currentIndex);
395                 console.log("Current index: ",currentIndex);
396             }
397         }
398
399         Keys.onPressed: {
400             if ((event.key === 16777221 | event.key === 16777220) & timerbtn.activeFocus ===
                ↪ true) {
401                 timerbtn.focus = false;
402                 console.log("Escaped timer. Timer = ", timerbtn.text);
403                 onClicked: deviceHandler.sendTimeout(timerbtn.text);
404             }
405             if ((event.key === 16777216) & timerbtn.activeFocus === true) {
406                 timerbtn.focus = false;
407                 timerbtn.text = deviceHandler.timeout;
408                 console.log("Escaped timer. Timer = ", timerbtn.text);
409             }
410             if (keybindbtntxt.activeFocus) {
411                 switch (event.key) {
```

```
412                 case 16777220:
413                     console.log("Escaped keybind by enter");
414                     keybindbtntxt.text = "Enter";
415                     keybindbtntxt.focus = false;
416                     break;
417                 case 16777221:
418                     console.log("Escaped keybind by enter");
419                     keybindbtntxt.text = "Enter";
420                     keybindbtntxt.focus = false;
421                     break;
422                 case 16777234:
423                     console.log("Escaped keybind by arrow left");
424                     keybindbtntxt.text = "←";
425                     keybindbtntxt.focus = false;
426                     break;
427                 case 16777235:
428                     console.log("Escaped keybind by arrow up");
429                     keybindbtntxt.text = "↑";
430                     keybindbtntxt.focus = false;
431                     break;
432                 case 16777236:
433                     console.log("Escaped keybind by arrow right");
434                     keybindbtntxt.text = "→";
435                     keybindbtntxt.focus = false;
436                     break;
437                 case 16777237:
438                     console.log("Escaped keybind by arrow down");
439                     keybindbtntxt.text = "↓";
440                     keybindbtntxt.focus = false;
441                     break;
442                 case 16777249:
443                     console.log("Escaped keybind by ctrl");
444                     keybindbtntxt.text = "Ctrl";
445                     keybindbtntxt.focus = false;
446                     break;
447                 case 16777251:
448                     console.log("Escaped keybind by alt");
449                     keybindbtntxt.text = "Alt";
450                     keybindbtntxt.focus = false;
451                     break;
452                 case 16777248:
453                     console.log("Escaped keybind by shift");
454                     keybindbtntxt.text = "Shift";
455                     keybindbtntxt.focus = false;
456                     break;
457                 case 16777216:
458                     console.log("Escaped keybind by escape");
459                     keybindbtntxt.text = "Escape";
460                     keybindbtntxt.focus = false;
461                     break;
462                 }
463             }
464         }
465     }
```

```
466
467     GameButton {
468         id: rstButton
469         anchors.horizontalCenter: parent.horizontalCenter
470         anchors.bottom: parent.bottom
471         anchors.bottomMargin: GameSettings.fieldMargin
472         width: parent.width*0.5 - 1.5 * GameSettings.fieldMargin
473         anchors.horizontalCenterOffset: -0.5*GameSettings.fieldMargin - 0.5*width
474         height: GameSettings.fieldHeight
475         enabled: true
476         radius: GameSettings.buttonRadius
477
478         onClicked: {
479             deviceHandler.updateMode(6);
480         }
481
482         Text {
483             anchors.centerIn: parent
484             font.pixelSize: GameSettings.tinyFontSize
485             text: qsTr("Set to Factory Settings")
486             horizontalAlignment: Text.AlignHCenter
487             wrapMode: Text.WordWrap
488             color: rstButton.enabled ? GameSettings.textColor : GameSettings.disabledTextColor
489         }
490     }
491
492     GameButton {
493         id: idButton
494         anchors.horizontalCenter: parent.horizontalCenter
495         anchors.bottom: parent.bottom
496         anchors.bottomMargin: GameSettings.fieldMargin
497         width: parent.width*0.5 - 1.5 * GameSettings.fieldMargin
498         anchors.horizontalCenterOffset: 0.5*GameSettings.fieldMargin + 0.5*width
499         height: GameSettings.fieldHeight
500         radius: GameSettings.buttonRadius
501
502         onClicked: deviceHandler.updateMode(7);
503
504         Text {
505             anchors.centerIn: parent
506             font.pixelSize: GameSettings.tinyFontSize
507             text: qsTr("Identify")
508             wrapMode: Text.WordWrap
509             color: deviceHandler.getIDflg ? GameSettings.buttonPressedColor :
                ↪ GameSettings.textColor
510         }
511
512
513     }
514 }
```

Appendix H

Results final testing

H.1 Range test

Here the all the data collected during the range test can be found. For each BUG the range at which the keys stopped correctly coming through to the PC or the range at which the BUG disconnected was logged. Afterwards the average per range test was calculated. This can all be seen in Table H.1

Table H.1: Results of the range test for the tested BUGs

BUG #	range test 1[m]	range test 2[m]	range test 3[m]
1	43,81	85,52	70,55
2	99,57	72	75,5
3	92,6	69,45	73,36
4	85,34	68,36	54,06
5	86,3	79,85	70
6	90,07	77,9	78,75
7	58,46	68,73	66,64
8	71,54	68	68,8
9	70,57	66,4	57,8
10	43,14	67,64	57,82
Average	74,14	72,385	67,33

H.2 Accuracy

As explained in section 5.1 for each BUG at each distance 50 key presses were sent. The actual received presses were then logged as can be seen in Table H.2. Consequently the number of received presses was calculated as a percentage of the 50 sent presses. After that the deviation from 100 was calculated for each of the percentages. Finally the Mean Square Error (MSE) was determined for each range. Thus resulting in the percentage of incorrectly received key presses. All the mentioned results can be seen in Table H.2.

Table H.2: Results of the accuracy test for the tested BUGs

BUG #	Received presses[#]				Received presses[%]				Deviation[%]			
range[m]→	35.6	53.5	71.3	89.1	35.6	53.5	71.3	89.1	35.6	53.5	71.3	89.1
1	50	50	0	0	100	100	0	0	0	0	-100	-100
2	50	50	50	0	100	100	100	0	0	0	0	-100
3	50	44	105	0	100	88	210	0	0	-12	110	-100
4	50	50	52	65	100	100	104	130	0	0	4	30
5	50	50	50	130	100	100	100	260	0	0	0	160
6	50	50	91	30	100	100	182	60	0	0	82	-40
7	50	50	81	44	100	100	162	88	0	0	62	-12
8	51	92	22	0	102	184	44	0	2	84	-56	-100
9	50	50	41	59	100	100	82	118	0	0	-18	18
10	50	31	0	0	100	62	0	0	0	-38	-100	-100
11	50	51	120	72	100	102	240	144	0	2	140	44
MSE									0,6	28,0	77,3	85,5

H.3 Delay

Used python code

```
# -*- coding: utf-8 -*-

"Keylogger vs Serial read"

import time
import keyboard
import serial

COMPort = "COM4"
BtnStr = "button released"

serialPort = serial.Serial(COMPort, 115200, timeout=2)

serialPort.read_all()

"""
Set timeA when serial
Set timeB when keypress
Show timeA - timeB
Keep averaging until event, then restart
"""
try:
    while(1):
        nextBUG = False
        n=0
        delay_average = 0
        print("New BUG!")

        while(not nextBUG):
            time_key = 0
            time_serial = 0
            print("Detecting BUG input")
```

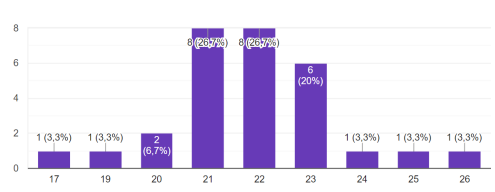
```
33
34          while((time_key == 0) or (time_serial == 0)) :
35
36              if (keyboard.is_pressed(" ")):
37                  while(keyboard.is_pressed(" ")):
38                      {}
39                  time_key = time.time()
40                  print("space pressed")
41
42              if(serialPort.in_waiting>0):
43                  serialString = serialPort.readline()
44                  if (serialString.decode('Ascii').find(BtnStr) > 0):
45                      time_serial = time.time()
46                      print("serial btn rcvd")
47              if (keyboard.is_pressed("n")):
48                  while(keyboard.is_pressed("n")):
49                      {}
50                  nextBUG = True
51                  break
52
53          """ Compute average delay with new sample, compensate serial delay"""
54          """ Serial delay -> 25 chars = 400 bits. 115200 bit/s => 0.0034..sec"""
55          delay_average = delay_average * n + (time_key - time_serial + 0.003472222)
56          n = n + 1
57          delay_average = delay_average / n
58          print("average =",delay_average,"sec ( n =",n,")")
59
60  except:
61      serialPort.close()
```

H.3.1 Results

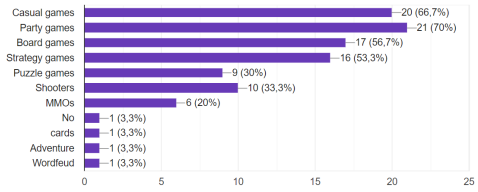
Table H.3: The results of the delay test for all the BUGs

Leopard Moth #	delay[ms] (n=50)
1	1,3604
2	1,4448
3	1,2451
4	1,4360
5	1,0805
6	1,4359
7	1,3851
8	1,4787
9	1,4988
10	1,3999
Average	1,3765

H.4 Public testing

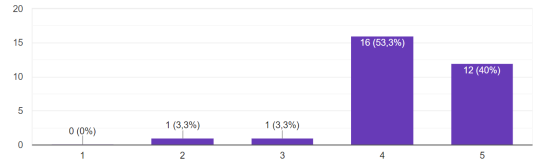


(a) View of the age of the test participants.

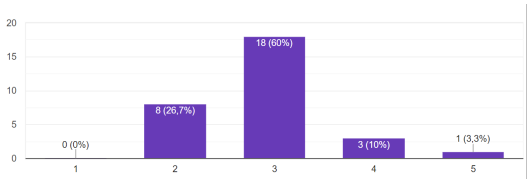


(b) Overview of game genres played by the test participants.

Figure H.1: Age and game preference of set of test participants

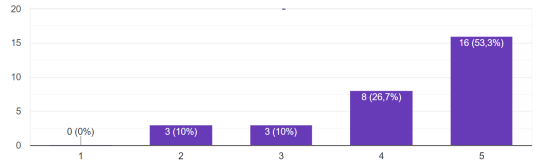


(a) Responses regarding the comfort level of the BUG from one (bad) to 5 (good).

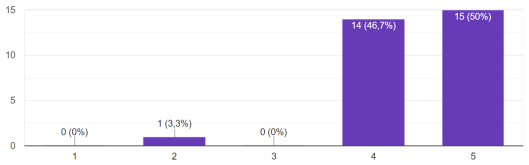


(b) Responses regarding the size of the BUG from one (too small) to five (too big). Three states it is a good size.

Figure H.2: Responses of the test panel regarding the 3D model



(a) Responses regarding the clarity of the current key bind from one (bad) to 5 (good).



(b) Responses regarding the general user experience of the BUG from one (bad) to five (good).

Figure H.3: Responses of the test panel regarding the user interface

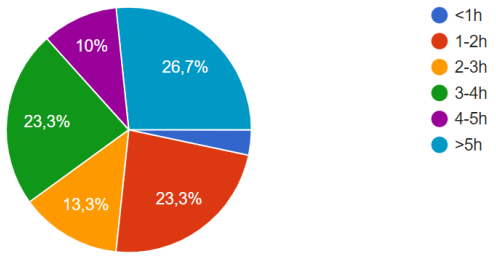


Figure H.4: Responses regarding the expected battery life of a BUG.

Appendix I

Falstad simulations: Pictures and URLs

I.1 Bidirectional LED

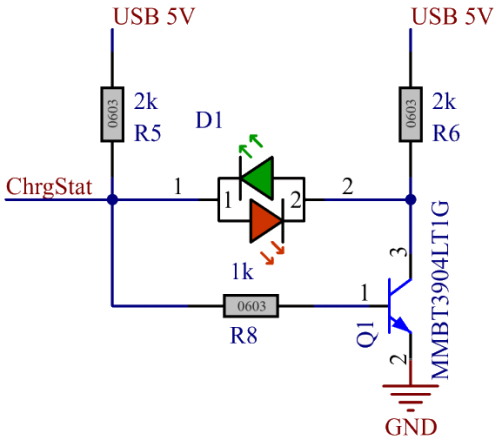


Figure I.1: The final circuit with the bidirectional LED indicating the charging status. The simulation from this model can be found at bit.ly/bugthesis_chargeled.

I.2 Wake-up circuit

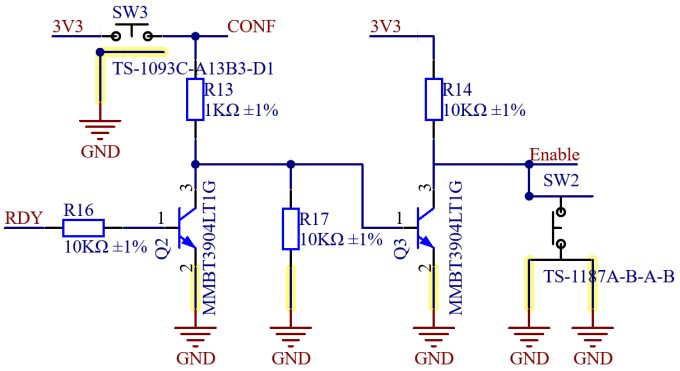
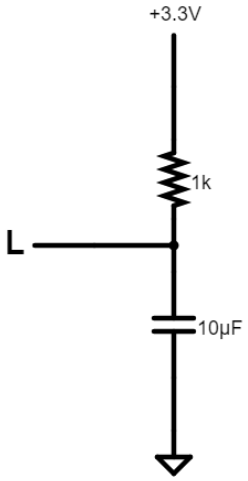
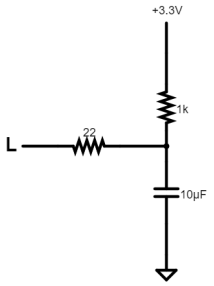


Figure I.2: The schematics of the wake-up circuitry. The simulation can be found at bit.ly/bugthesis_wakeup.

I.3 ID RC circuit



(a) The circuit made on the BUG PCB



(b) The circuit with the soldered on resistor

Figure I.3: The two options for ID voltage division. The link to the model can be found at bit.ly/bugthesis_idrc.

Appendix J

Schematics of the BUG

On the next page, the complete schematics of the BUG can be found with brief descriptions with each part.

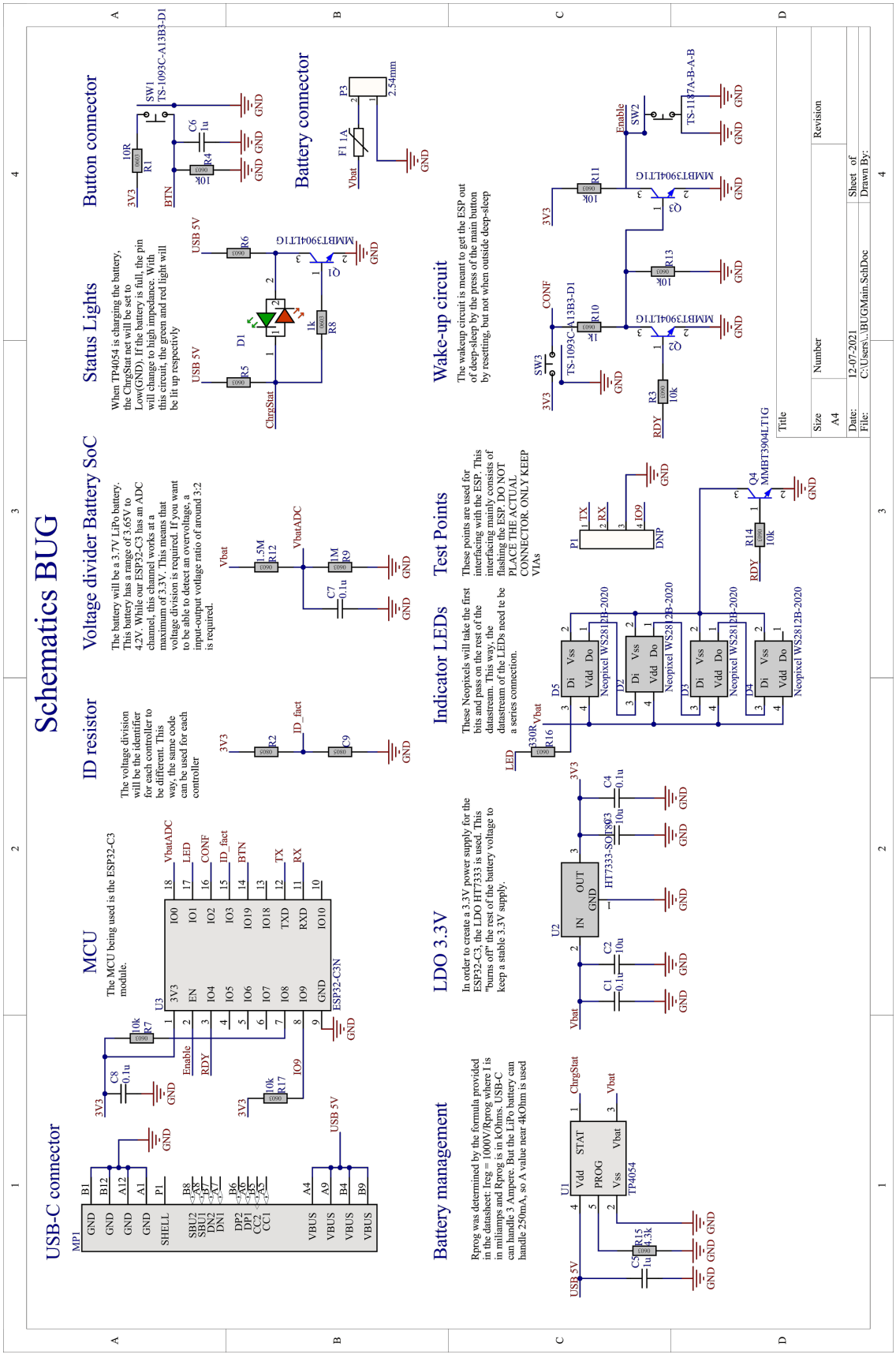


Figure J.1: Complete schematics of the BUG

Appendix K

Tested games

Below a list can be found of games which are found suitable when using the BUG. It is also shown if a score is present, if a versus mode between teams is possible and which key binds are used. It will also be noted if the game could be played without reconfiguration.

Title	Key binds	Default	Score	Versus	Link
Pacman	Arrows	Yes	Yes	No	bit.ly/pacmanbug
Bomberman	Arrows with space	No	Yes	Yes[1]	bit.ly/bomberbug
Snake	Arrows	Yes	Yes	No	bit.ly/snakebug
Pong	Arrow up/down	Yes	Yes	Yes[2]	bit.ly/pongbug
Temple Run	Arrows[3]	Yes	Yes	No	bit.ly/templebug
Tetris	Arrows[4]	Yes	Yes	No	bit.ly/tetrisbug

1) Bomberman multiplayer is possible in another version (BombTag). It is also possible on a single PC if more than 8 BUGs could be connected simultaneously.
2) For multiplayer, a non-default configuration is needed.
3) The space key could also be used for boosts, but is not necessary to play the game.
4) Although it is completely playable with the arrow keys, some more keys can be used to add functionality to the game.

Bibliography

[1] Steam. "Controller gaming on pc." (2018), [Online]. Available: https://steamcommunity.com/games/593110/announcements/detail/1712946892833213377 (visited on 03/12/2021).

[2] Fact.MR. "Gaming controller market." (2021), [Online]. Available: https://www.factmr.com/report/gaming-controller-market (visited on 03/12/2021).

[3] Bluetooth SIG, Inc. "Bluetooth® wireless technology." (2021), [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/ (visited on 29/11/2021).

[4] K. Townsend, C. Cufí, Akiba and R. Davidson, Getting Started with Bluetooth Low Energy. O'Reilly Media, Inc., May 2014, Chapter four, ISBN: 9781491949511. [Online]. Available: https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html.

[5] Ferjerez. "Diy controller." licensed under Creative Commons - Attribution license, Makerbot Thingiverse. (2017), [Online]. Available: https://www.thingiverse.com/thing:2669820 (visited on 27/09/2021).

[6] ETSI, At command set for user equipment (ue), version 15.7.0, 2019. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/127000_127099/127007/15.07.00_60/ts_127007v150700p.pdf (visited on 02/12/2021).

[7] T-vK. "Esp32-ble-keyboard." Used release: v0.3.1-beta, GitHub. (2021), [Online]. Available: https://github.com/T-vK/ESP32-BLE-Keyboard (visited on 29/11/2021).

[8] Adafruit. "Adafruit$_n$*eopixel*." Used release: v1.10.0, GitHub. (2021), [Online]. Available: https://github.com/adafruit/Adafruit_NeoPixel (visited on 12/10/2021).

[9] M. Papadatou-Pastou, E. Ntolka, J. Schmitz et al., "Human handedness: A meta-analysis," Psychological Bulletin, vol. 146, p. 481, 6 Jun. 2020, ISSN: 0033-2909. DOI: 10.1037/bul0000229. [Online]. Available: https://www.proquest.com/scholarly-journals/human-handedness-meta-analysis/docview/2406640810/se-2.

[10] "Pulling the plug on consumer frustration and e-waste: Commission proposes a common charger for electronic devices," European Commission, Brussels, Aug. 2021. [Online]. Available: https://ec.europa.eu/commission/presscorner/detail/en/ip_21_4613 (visited on 18/11/2021).

[11] Texas Instruments. "Selecting antennas for low-power wireless applications." (2008), [Online]. Available: https://www.ti.com/lit/an/slyt296/slyt296.pdf?ts=1632820973946& (visited on 22/11/2021).

[12] Symmetry Electronics. "Internal antennas: Different types and advantages." (2021), [Online]. Available: https://www.semiconductorstore.com/blog/2021/Internal-Antennas-Different-Types-and-Advantages-Symmetry-Blog/4357/ (visited on 24/11/2021).

[13] AI-thinker, Esp-c3-12f specification, Version 1.0, 2021. [Online]. Available: https://docs.ai-thinker.com/_media/esp32/docs/esp-c3-12f_specification.pdf (visited on 02/12/2021).

[14] ——, Esp-c3-13 specification, Version 1.0, 2021. [Online]. Available: https://docs.ai-thinker.com/_media/esp32/docs/esp-c3-13_specification.pdf (visited on 18/11/2021).

[15] Cellevia Batteries. "Specification approval sheet." (2016), [Online]. Available: https://www.tme.eu/Document/aa593083f76c72af8796398caaac30a8/cel0016.pdf (visited on 02/12/2021).

[16] Monolithic Power. "How to select a lithium-ion battery charge management ic." (2021), [Online]. Available: https://www.monolithicpower.com/en/how-to-select-lithium-ion-battery-charge-management-ic (visited on 01/12/2021).
[17] NanJing Top Power, Tp4054 standalone linear li-ion battery charger with thermal regulation in sot.
[18] Holtek Semiconductors, Ht73xx low power consumption ldo, Rev 1.3, 25th Jan. 2005.
[19] Espressif. "Analog to digital converter (adc)." (2021), [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/api-reference/peripherals/adc.html (visited on 03/12/2021).
[20] Qt. "Bluetooth low energy heart rate game." (2017), [Online]. Available: https://doc.qt.io/qt-5/qtbluetooth-heartrate-game-example.html (visited on 29/11/2021).
[21] Human Benchmark. "Statistics." (2021), [Online]. Available: https://humanbenchmark.com/tests/reactiontime/statistics (visited on 13/12/2021).
[22] S. Gupta and D. V. Gadre, "Multiplexing technique yields a reduced-pin-count led display," European Documentary Network, 2008.