Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Turan Elchuev

Implementation of a universal remote control device

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Turan Elchuev

Implementation of a universal remote control device

Bachelor Thesis based on the examination and study
regulations for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner : Prof. Dr.-Ing. Rainer Schoenen
Second examiner : Prof. Dr.-Ing. Marc Hensel

Day of delivery November 4, 2019

Turan Elchuev

Title of the Bachelor Thesis

Implementation of a universal remote control device

Keywords

Remote control, Wireless, Microcontroller

Abstract

Various communication channels and protocols for data transmission exist. Each protocol serves its purpose and has its advantages, hence, it has the right to exist. On the other hand, due to a high number of available protocols, people are sometimes forced to use different devices for similar applications. This comes at the cost of paying for multiple devices, which serve the single purpose - remote control. For example, a TV remote cannot be used to manipulate a drone, an RC car manipulator in the most of the cases cannot be used to play computer games, or, the more so, to switch the lights in the room.

The goal of this thesis is to implement a universal remote controller. The product should be able to communicate via various channels and using different protocols such as Bluetooth, Wi-Fi, USB, wireless transceivers, and others. Furthermore, the device should provide a variety of input methods such as keys and inertial values (movement and orientation of the device in the 3D space). As a result, it is expected to have a single gadget, that will provide a user with the possibility to manipulate various types of remote devices, such as computers, PC games, home appliances, smart-home devices, RC toys, etc. in an intuitive and user-friendly manner.

Turan Elchuev

Thema der Bachelorarbeit

Implementierung eines universellen Fernsteuerungsgerätes

Stichworte

Fernsteuerung, Drahtlos, Mikrocontroller

Kurzzusammenfassung

Es gibt verschiedene Kommunikationskanäle und Protokolle für die Datenübertragung. Jedes Protokoll dient seinem Zweck und hat seine Vorteile, daher hat es das Recht zu existieren. Andererseits sind die Menschen aufgrund einer hohen Anzahl von verfügbaren Protokollen manchmal gezwungen, verschiedene Geräte für ähnliche Anwendungen zu verwenden. Dies geht zu Lasten der Bezahlung mehrerer Geräte, die dem einen Zweck dienen - der Fernbedienung. So kann beispielsweise eine TV-Fernbedienung nicht verwendet werden, um eine Drohne zu manipulieren; ein RC-Automanipulator kann in den meisten Fällen nicht zum Spielen von Computerspielen oder, umso mehr, zum Schalten der Lichter im Raum verwendet werden.

Das Ziel dieser Arbeit ist die Implementierung eines universellen Fernsteuerungsgerätes. Das Produkt sollte in der Lage sein, über verschiedene Kanäle und unter Verwendung verschiedener Protokolle wie Bluetooth, Wi-Fi, USB, drahtlose Transceiver und andere zu kommunizieren. Darüber hinaus sollte die Vorrichtung eine Vielzahl von Eingabemethoden wie Tasten und Inertialkräfte (Bewegung und Ausrichtung des Gerätes im 3D-Raum). Daher wird erwartet, dass es über ein einziges Gadget verfügt, das es dem Benutzer ermöglicht, verschiedene Arten von Remote-Geräten wie Computer, PC-Spiele, Haushaltsgeräte, Smart-Home-Geräte, RC-Spielzeug usw. auf intuitive und benutzerfreundliche Weise zu bedienen.

Contents

List of Tables **7**

List of Figures **8**

1. Introduction **10**
1.1. Motivation 10
1.2. Outline 11

2. Background **12**
2.1. Theoretical Fundamentals 12
2.2. Involved Resources 16
2.2.1. Hardware 16
2.2.2. Software 21

3. Task Definition **24**
3.1. Overview Of The Goal 24
3.2. Hardware Specification 25
3.3. Functional Requirements 26
3.4. Nonfunctional Requirements 28

4. Requirements Analysis And Design **29**
4.1. Hardware Design 29
4.1.1. Layout 29
4.1.2. Modules Interconnection 30
4.2. Software Design 31
4.2.1. Analysis Of Libraries 32
4.2.2. Impact Of Hardware 32
4.2.3. Architecture 33

5. Hardware Implementation **36**
5.1. Power And Programming Circuit 36
5.2. Mainboard Circuit 38
5.3. Complete Device 39

6. Software Implementation **47**
6.1. Communication Protocol 47
6.2. ESP32 48
6.2.1. System Overview 48
6.2.2. Main Routine 52
6.2.3. System Handler 52
6.2.4. Main Menu 53
6.2.5. Storage Handler 55
6.2.6. Accelerometer, Gyroscope, Magnetometer 57
6.2.7. OLED Display 59
6.2.8. Keys, LEDs 60
6.2.9. Power 61
6.2.10. PC Gamepad Mode 63
6.2.11. Universal Joypad Mode 65
6.2.12. Keyboard Mode 66
6.2.13. Mouse Mode 67
6.2.14. Service Mode 69
6.2.15. USB Serial Helper 69
6.2.16. Wi-Fi 72
6.2.17. Sensors Mode 73
6.2.18. Debug Output 73
6.3. Atmega32u4 73

7. Testing **76**
7.1. Unit Test 76
7.2. System Test 78
7.3. Timing 81
7.4. Universal Joypad Demo 83

8. Summary **87**
8.1. Project Status 87
8.2. Problems 87
8.3. Further Work Outlook 88

9. Conclusion **90**

Bibliography **91**

A. Appendix **93**
A.1. Unit Test output (ESP32) 93

List of Tables

5.1. Main pin connections 40
5.2. Port expander pin connections 44
5.3. Debug interface pin connections 45

6.1. Keyboard key mapping 67
6.2. Mouse function mapping 68

7.1. Demo functions 86

List of Figures

2.1. Roll, pitch, yaw . . . 14
2.2. SPI network . . . 15
2.3. I2C bus . . . 15
2.4. UART connection . . . 16
2.5. ODROID Universal Motion Joypad . . . 16
2.6. Initial state of the wheel . . . 17
2.7. ESP32 pinout . . . 18
2.8. Arduino Micro and its pinout . . . 18
2.9. BEETLE BadUSB Micro . . . 19

3.1. Device concept . . . 24

4.1. Design approach: hardware . . . 31
4.2. Design approach: software concept . . . 34
4.3. Design approach: data flow . . . 35

5.1. Power and programming circuit . . . 41
5.2. Main board circuit . . . 42
5.3. Buttons and LEDs circuit . . . 43
5.4. Debug interface circuit . . . 43
5.5. Final device: outer look . . . 45
5.6. Final device: inside . . . 46

6.1. Communication protocol frame . . . 47
6.2. ESP32 overview class diagram . . . 51
6.3. Wheel key labels . . . 55
6.4. Storage Handler class diagram . . . 56
6.5. RPY filtering . . . 58
6.6. The battery management tool . . . 63
6.7. Gamepad frame . . . 64
6.8. Joypad frame . . . 65
6.9. Keyboard frame . . . 66
6.10. Mouse frame . . . 67
6.11. USB Serial Helper class diagram . . . 70

6.12. USB Serial Helper example 71
6.13. WiFi management tool 72
6.14. The Beetle: class diagram 75

7.1. Unit test menus 77
7.2. Joypad frames: Wi-Fi UDP 78
7.3. PC Gamepad calibration 79
7.4. PC Gamepad RPY mode test 80
7.5. Data transmission intervals 82
7.6. Demo devices 83

1. Introduction

1.1. Motivation

Communication technologies have developed tremendously over the past decades. Today, they enable a vast amount of possibilities ranging from a simple remote control of the home appliances, through the interconnection of the entire world into a global net, to the long-distance data transmission through space. In telecommunications, a very important role is given to the protocols as they facilitate the standardized, coordinated, and relatively reliable ways of data transmission over the communication media. So far, there have been many protocols developed, that operate on various wireless and wired transmission channels. Some basic examples of such protocols and communication technologies, which are widely used in daily life, are the Internet, Wi-Fi, Bluetooth, USB, Infrared Remote Control, etc.

Some protocols and communication channels have found their application in certain domains more than in the others. For example, the infrared communication is more used in the remote controls of the home appliances; the Bluetooth technology is popular in the production of wireless consumer electronics, such as audio headsets, mouses, keyboards, and similar; the RC vehicle industry mainly employs the 2.4 GHz band for wireless data transmission between the joystick and the vehicle, etc. The drawback of such diversity of technologies and protocols is that people are sometimes forced to buy different devices, which serve a single purpose - the remote control. The reason is clear - one usually cannot use an RC car joystick to switch the lights in the room or the channel on the TV; an air-conditioning remote control cannot be used to play computer games, and so on.

The motivation behind this thesis is to implement a concept of a universal remote control device, which is meant to encapsulate the functionality of various manipulators, such as PC gamepads, joysticks, remote controls of home appliances, PC mouses and keyboards, etc. in a single body. A product, that could be equally intuitively and in a user-friendly manner used to manipulate various devices, which are meant to be controlled by differing methods. A user, for example, would be able to play PC games, control RC vehicles, manipulate a smart TV or a stereo system, set air conditioning, or even toggle the lights in the house using a single device. Such remote control would alleviate the need to buy for each application a separate remote control, hence letting the user save money and improve overall user experience.

1.2. Outline

Chapter 2 provides a short theoretical background on relevant technical topics and introduces the hardware and software resources involved along the progression of the work. Chapter 3 defines the goals to be achieved. This includes a general overview of the concept as well as the hardware and software specifications of the target product. Chapter 4 describes the process of the requirements analysis and decision-making concerning the hardware and software design to be followed. Chapters 5 and 6 are dedicated to the process of the implementation of the hardware and software of the device respectively. Here, detailed information is provided about the implementation of each component of the system. Chapter 7 describes the process of testing the developed device. This includes implementation of unit tests, system tests as well as a demo to assess and showcase the performance of the device in the intended scope of its application. Chapter 8 summarizes the performed work by evaluating the compliance of the developed device with the requirements from chapter 3. Furthermore, the chapter describes the major problems faced along the process of implementation as well as suggest potential ways to further improve the device. Finally, the work is concluded by chapter 9.

2. Background

This chapter provides a theoretical background and gives insight into the technical resources involved in the development of the project. First, the reader will gain basic theoretical knowledge about each relevant technology. Later, the hardware components used to build the final product will be shortly introduced. The chapter will be finalized by the section dedicated to the software packages - this includes a range of desktop applications, browser-based tools as well as programming languages used for management, implementation, and documentation of this project.

2.1. Theoretical Fundamentals

In this section, the theoretical background on the selected topics is provided, which will help to gain a better understanding of the major components and technologies involved in this project.

Microcontroller Unit (MCU)

Nowadays, most consumer electronics, industrial machines, vehicles, and other technological devices, as well as their internal components, have got functionality that has to be automated and controlled. To facilitate this process, microcontrollers are used. Ajay V Deshmukh describes microcontrollers as follows: “Microcontrollers are single-chip microcomputers, more suited for control automation of machines and processes. Microcontrollers have central processing unit (CPU), memory, input/output (IO) ports, timers and counters, Analog-to-digital converter (ADC), digital-to-analog converter (DAC), serial ports, interrupt logic, oscillator circuitry and many more functional blocks on chip” [1].

Interrupt and Polling

Interrupts are used to distract the central processing unit (CPU) from the execution of the current task to bring its attention to some event that has high priority and needs to be handled.

Common examples of such events are input/output (IO) events such as key presses, timer timeout events, data ready interrupts, failures of peripheral devices and others. Interrupts remove the need to constantly check the status of a peripheral by the CPU, hence, the processor can focus on the execution of the other tasks. Whenever a peripheral needs to be served by the processor, it will send a signal to one of the processor's interrupt inputs [2].

The opposite of the *Interrupt* is *Polling*. Polling is the process of busy waiting when the processor continuously checks the status of the IO device. Polling in most of the cases leads to wasting of the processor's time but is simpler to implement in a program [2].

Analog-to-digital converter (ADC)

An Analog-to-digital converter (ADC) is a system that converts a continuous signal into the digital domain, where each sample of the continuous signal finds its corresponding digital representation. This allows the signal to be processed by a computer. Common application of the ADCs is measurement of voltages [3].

Moving Average Filter

The moving average is a popular filter in Digital Signal Processing (DSP), which is easy to implement. The filter is used to reduce the random noise of a time-domain sampled signal, or in other words, smooth it. The operational principle of the filter, as the name suggests, is averaging some samples of the input signal to produce an output signal. The output signal is calculated by the following equation:

$$y[i] = \frac{1}{M} \sum_{j=0}^{M-1} x[i+j]$$

Here, x[] stores samples of the input signal, and each value in the y[] stores the arithmetic mean of the previous M samples from the input [3]. For example, in a 3-point filter, the value of the 10^{th} output point will be calculated as follows:

$$y[10] = \frac{x[10] + x[11] + x[12]}{3}$$

Roll, Pitch, Yaw (RPY)

The terms Roll, Pitch and Yaw are used to describe the orientation of an object in the three dimensional (3D) space. They are commonly applied to aerial vehicles, such as planes, drones, etc. The orientation is expressed in terms of rotations of the object along its 3 principal axes, which are perpendicular to one another and intersect at the center of gravity of the object.

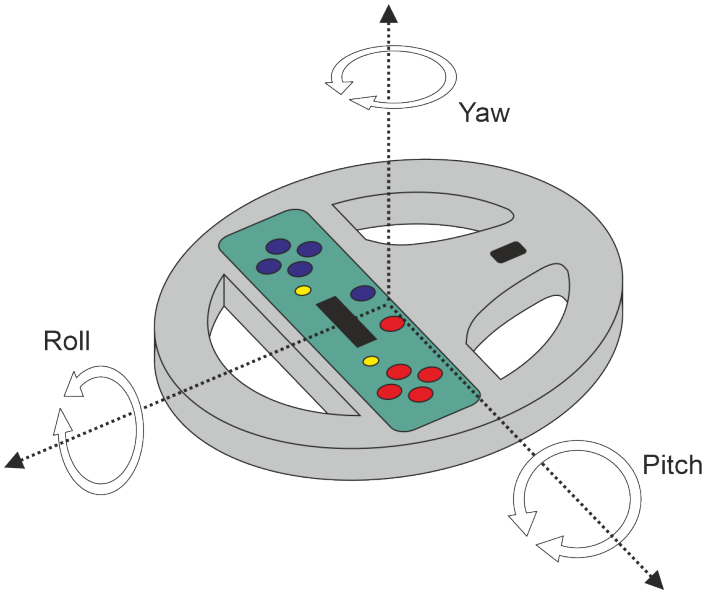

Figure 2.1.: The roll, pitch and yaw rotations of an object

Serial Peripheral Interface (SPI)

Serial Peripheral Interface is a synchronous serial communication protocol, that allows microcontrollers to communicate to peripheral modules or other microcontrollers. In a typical SPI network, there is a single master node and one or more slave nodes (peripherals). Communication is bidirectional and for each direction, there is a dedicated line:

- Master In Slave Out (MISO) line - transmission from the slaves to the master.
- Master Out Slave In (MOSI) line - transmission from the master to the slaves.

Besides, there is a Serial Clock (SCK) line, which is used to drive the clock signal from the master to the slaves. This way the synchronization is achieved. Since communication with SPI protocols is possible to only a single slave at a time, there are additional Slave Select (SS) lines, one for each slave, that are used by the master to enable one slave at a time by

pulling the line low. The main advantage of the SPI protocol is the possibility to transmit data at high rates (usually more than 10 megabits per second). The Fig. 2.2 illustrates a typical SPI network.

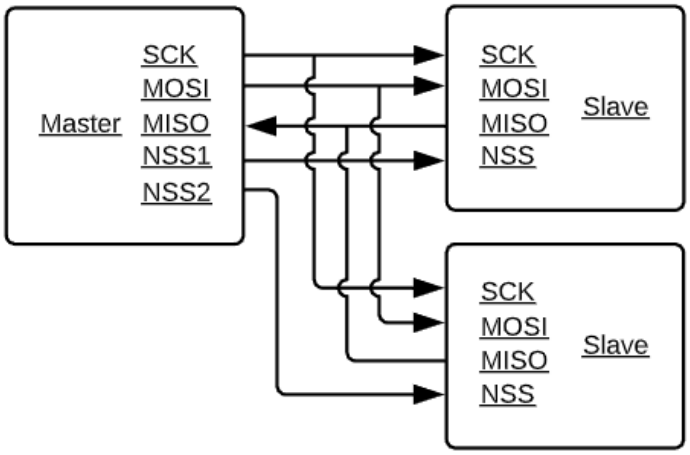

Figure 2.2.: A typical SPI network

Inter-integrated circuit (I^2C)

Inter-integrated circuit (I^2C) is another serial communication protocol. Similar to SPI, the I^2C network consists of a single master (usually a microcontroller) and multiple slave nodes (peripherals or other microcontrollers). However, the major advantage of the I^2C over the SPI is the bus topology and slave selection method, reduces the amount of required lines down to 2, namely the Serial Clock (SCL) and Serial Data (SDA). The former is used to synchronize communication between the nodes, the latter is used to transmit actual data. In the I^2C protocol, each slave node has its 7-bit wide address, which implies that there can be up to 128 slaves in an I^2C network. Selection of the slave is achieved by transmitting its address over the SDA line. The fastest I^2C modules can transmit at a rate up to 3.2 mega bits per second [4]. The Fig. 2.3 illustrates a simplified I^2C network.

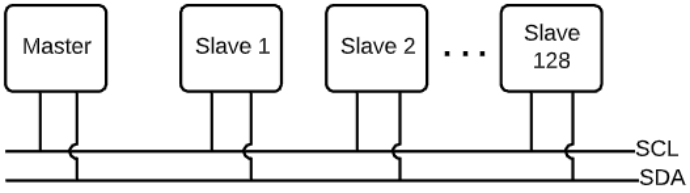

Figure 2.3.: A simplified representation of an I2C network

Universal Asynchronous Receiver/Transmitter (UART)

The Universal Asynchronous Receiver/Transmitter (UART) is yet another serial communication protocol widely used in embedded systems. It is asynchronous, which implies there is no clock line. Besides, it allows only 2 devices to communicate with each other. There is no master or slave in UART communication and data can be transmitted in both directions simultaneously. Each UART device has a Receive (Rx) and a Transmit (Tx) signal. Two UART interfaces should be connected as illustrated in Fig. 2.4.

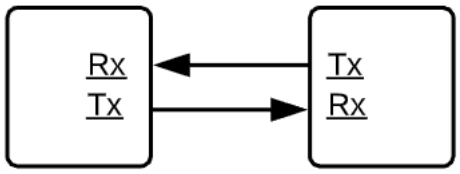

Figure 2.4.: UART signal connection (omitting the ground signal)

2.2. Involved Resources

2.2.1. Hardware

ODROID Universal Motion Joypad

The ODROID Universal Motion Joypad (hereinafter referred to as "the wheel") is a game controller in the shape of a wheel, that comes with a USB input/output (IO) board, a BMA150 motion sensor and 10 push buttons mounted on a detachable PCB board [5].

Figure 2.5.: ODROID Universal Motion Joypad

The wheel was used as the base for the device since it has a user-friendly form factor and can be effectively used in the target scope of the device in the motion-based operation modes.

Besides, it provided a PCB with intuitively aligned buttons that further reduced the necessity to make one.

Before the beginning of this project, the wheel had already been slightly modified. These modifications included a mounted Arduino Micro board, an OLED display and an MPU-9250 motion sensor. The figure 2.6 illustrates the initial state of the wheel.

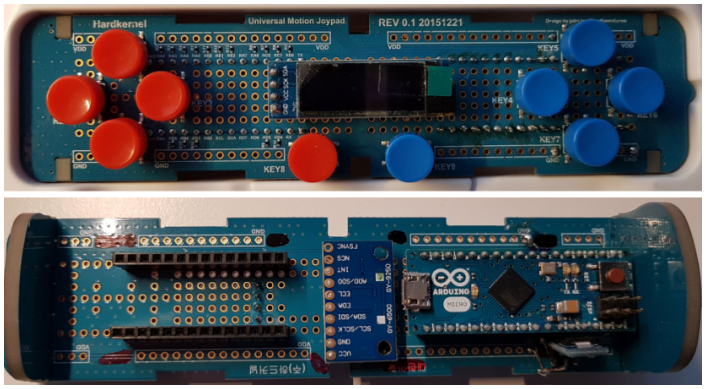

Figure 2.6.: Initial state of the wheel

DOIT ESP32 DEVKIT V1

The DOIT ESP32 DEVKIT V1 (hereinafter referred to as "the ESP32") is a development board based on the ESP32-WROOM-32 module. The ESP32-WROOM-32 is a powerful MCU module which along with 2 CPUs, GPIO pins, PWM and ADC capabilities as well as communication interfaces such as SPI, UART and I2C offers built-in BLE and Wi-Fi functionality, which make the module a perfect choice for the connected applications such as IoT [6]. The ESP32 was intended to be one of the computational modules of the device responsible among other tasks for the Wi-Fi and Bluetooth connectivity. The figure 2.7 illustrates a graphical representation as well as the pinout of the board.

Arduino Micro

The Arduino Micro is a development board based on the ATmega32U4 module. The ATmega32U4 is a high performance low-power 8-bit MCU with 32 KB of program memory, GPIO pins, PWM, ADC capabilities, SPI, UART and I2C as well as a built-in USB controller [7]. The latter enables USB features such as a keyboard, mouse, gamepad emulation and others. The board, as was mentioned before, came mounted on the PCB of the wheel and was intended to be used as another computational module of the device responsible among other tasks for the USB-related functionality. The figure 2.8 shows the board and its pinout.

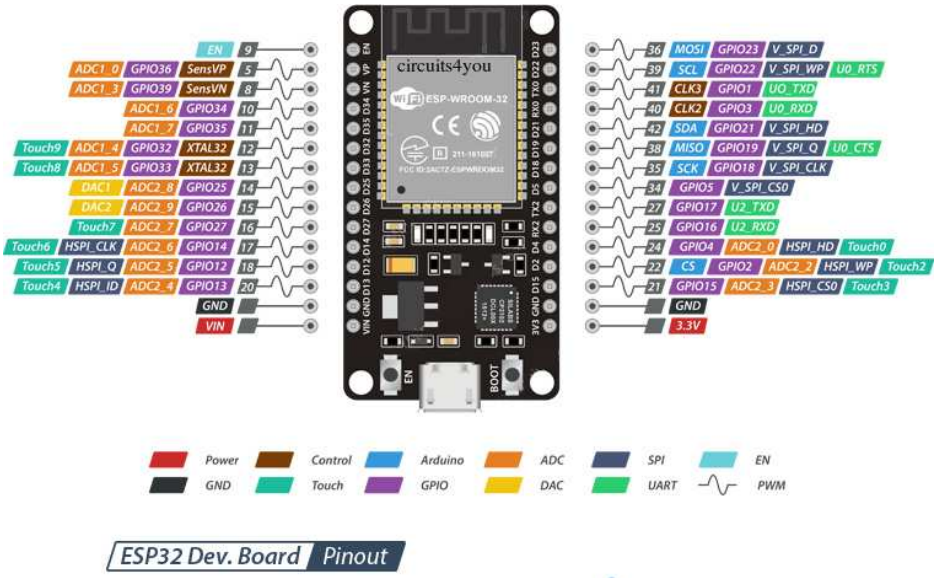

Figure 2.7.: ESP32 pinout

BEETLE BadUSB Micro

The BEETLE BadUSB Micro (hereinafter referred to as "the Beetle") is another development board based on the ATmega32U4 MCU. The main difference between this board and the Arduino Micro is the smaller footprint of the former. This comes, however, with the cost of the reduced amount of pins that can be accessed. As such, the BEETLE BadUSB Micro offers a limited number of pins, yet offering all core features and communication interfaces of the MCU, that were introduced before. Within this project, 2 such boards were used. The figure 2.9 illustrates the board.

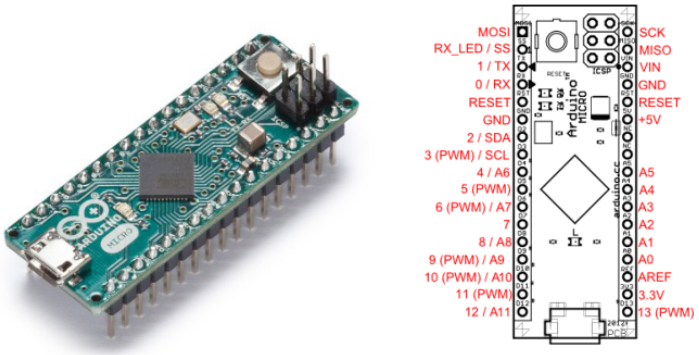

Figure 2.8.: Arduino Micro and its pinout

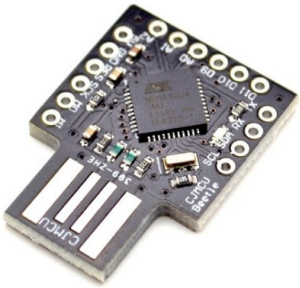

Figure 2.9.: BEETLE BadUSB Micro

IO peripherals

Port expander Port expanders are used in order to extend the number of GPIO pins when a large number of components have to be interfaced by a MCU or simply to keep free as many GPIO pins as possible. In this project, an MCP23017-E/SP port expander was used to interface the keys and the LEDs of the device. The expander provides 16 GPIO pins programmable to act as input or output, interrupt functionality as well as an I2C interface. The disadvantage of this module is the lack of PWM support.

OLED display A 128x32 SSD1306 OLED display that provides an I2C communication interface was used in the project to serve as the main output device.

Buzzer A buzzer is an audio signaling device. In this project, a piezoelectric buzzer was used to serve the purpose of notifying the user.

LEDs In this project, 2 common anode RGB LEDs were used as an additional output device.

Communication modules

Wireless transceiver In order to implement additional means of wireless communication and, correspondingly, enlarge the scope of application of the device, an 868MHz RFM69 transceiver module was used in the project. The module provides SPI for interfacing by a MCU.

Bluetooth module Few HC-05 and HC-06 BT modules were used in the project as an alternative mean of wireless communication, e.g. to control remote devices that have on-board classic BT modules. The modules can be interfaced via the built-in UART.

Sensors

MPU-9250 One of the core components of this project is the MPU-9250 module - a 9-Degrees Of Freedom (DOF) motion tracking device. It consists of three main modules: a Gyroscope, an Accelerometer and a Magnetometer, which make the device a powerful tool for tracking the position and orientation in the 3D space. The module can be interfaced using I2C and SPI communication protocols.

Power

Battery A 3.7V 1280 mAh Lithium-Ion battery was used to power the device. This type of rechargeable batteries is commonly used for portable electronics and hence is suitable for this project.

Charger module In this project, an LTC4056-module based charger board was used. The board encapsulates all the necessary safety circuitry as well as LEDs indicating the “charging” and “charged” states.

DC/DC converter This type of converters is used to convert DC from one voltage level to another. Since some of the components (the ESP32 and all the peripherals) within this project had to be supplied with a 3.3V source and the others (the Beetle) - with 5V, 2 converters were used:

- a step-up converter to convert the battery’s output voltage of 3.7V to 5V for the 5V-powered devices
- a step-down converter to convert 5v to 3.3v correspondingly for the 3.3V-powered devices.

Level shifter is a module that converts one logical voltage level to another depending on the specification and applied reference voltages. In this project, a level shifter was used to interconnect the ESP32 with the Beetle in order to prevent the former from potential damages as it has 3.3V logic level whereas the latter has a 5V level.

Miscellaneous

In addition to the components mentioned above, there were used a number of switches, sockets as well as passive electric components.

2.2.2. Software

The Arduino integrated development environment (IDE)

The Arduino IDE is a light-weight cross-platform open-source application, that is used for writing and compiling code for the genuine Arduino boards, their compatible clones as well as a range of development boards from other vendors. The IDE offers a wide range of features such as serial monitor, serial plotter, sample sketches, and others. The main motivation behind the application was to make microcontroller programming so easy that any inexperienced individual could do it. This strategy led to considerable growth in the popularity of the platform. As a result, nowadays the Arduino Community consists of a huge network of contributors who have produced an enormous amount of materials such as libraries, sample projects, a tutorial for nearly any context in which a microcontroller can be applied. Within this project, the Arduino IDE was selected as the main tool for the programming of the device.

C++

C++ is a general-purpose OOP language. Among other features that make it one of the most popular programming languages of the modern world, it is worth emphasizing its low-level memory manipulation capabilities. This feature combined with the advantages of the OOP model makes the language particularly suitable for programming of the systems with limited resources and performance [8]. Within this project, the entire system excluding minor utility programs was developed in C++.

Java

Java is another popular general-purpose OOP language. What makes the language stand out against the background of the others is the so-called Write Once, Run Anywhere (WORA) concept, which means that the programs developed in Java could be run on any platform (Windows, macOS, Linux, etc.) without being adapted for the specific one. Java is nowadays widely used for the development of web-based, mobile and desktop applications. In contrast to C++, it does not provide the low-level memory access features, which makes it hard to

apply in the context of embedded systems. Within this project, Java was used for the purpose of testing some of the device's functions, such as UDP and TCP/IP.

MATLAB

MATLAB is a multi-paradigm programming language and a development environment used mainly for numerical computations. The language is widely applied in the engineering context. It provides a broad range of features such as matrix calculations, solving differential equations, data visualization, system modeling and simulation, etc. [9] In this project, MATLAB was used for visualization of the orientation of the device in the three dimensional (3D) space.

Git

Git is the most popular version control system, an essential tool in the software development process, yet applied in other contexts where change tracking is needed as well. It provides a wide range of features such as branching, conflicts resolution, etc., and most importantly - a collaboration of multiple developers [10]. There is a variety of console and graphical user interface (GUI)-based applications as well as online tools for handling Git available. In this project, the Git Bash - a Git console for Windows, and the GitLab - a web-based tool providing among other features a Git repository manager and issue tracking were used.

Gantt chart

Gantt chart is a chart used for illustration of the tasks of a project and their duration on a timeline. In a typical Gantt chart, the tasks are specified on the vertical axis, whereas the horizontal axis represents the timeline. Each task has a start and end times and is displayed on the chart by a horizontal bar with its left edge aligned with the start time and the width corresponding to the duration. Modern project management tools are based on the Gantt chart and offer a range of additional features such as dependencies between tasks, resource allocation, and others, that ease the project management process. For this project, the Agantty - a web-based project management tool was used.

Autodesk® EAGLE™

EAGLE™ is a software package used for printed circuit board (PCB) and integrated circuit (IC) design. The program contains a schematic editor tool that is used for designing circuit diagrams, which then can be transformed to e.g. a corresponding PCB. Although the development of a PCB was out of the scope of this project, the tool came in handy for designing all the relevant circuit diagrams.

Lucidchart

Lucidchart is a web-based tool that is used for drawing many types of charts and diagrams in the engineering, financial, project management, and other contexts. The platform allows sharing, revising charts and diagrams as well as user collaboration. The tool was applied in the scope of this project for drawing the UML class diagrams and other charts.

3. Task Definition

The purpose of this project, as the title suggests, is to implement a universal remote control device. This chapter provides a general overview of what has to be achieved followed by the more specific requirements applied to the hardware and software design.

3.1. Overview Of The Goal

The final product should provide a user with the possibility to manipulate a variety of appliances using wireless and wired communication. Manipulation should be motion-based as well as performed through pressing keys. Additionally, the device should have feedback capabilities such as light, sound, and vibration.

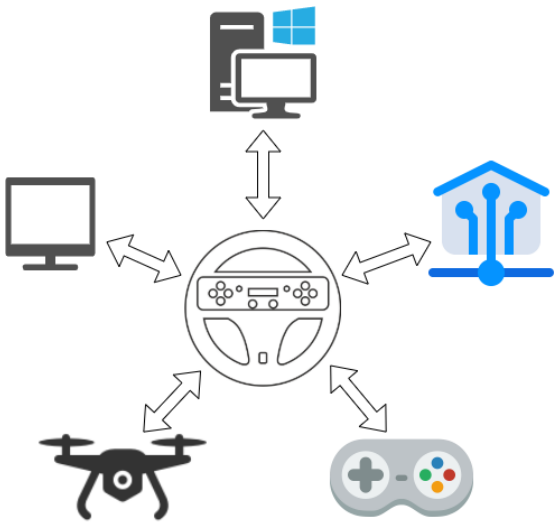

Figure 3.1.: The concept

The potential scope of application of the product is manipulation and interfacing of remote devices such as a Personal Computer (PC), a variety of Radio Control (RC) vehicles (e.g. a drone, a car), PC games, home appliances, media players, Smart Home and Internet of Things (IoT) devices, etc.

The implementation should be based on the Wheel, which was introduced earlier in the chapter 2, by preserving all of its original hardware except the MCU and complementing it with the components listed in the same chapter. Programming should be performed in the Arduino IDE. Unit Tests should be developed to verify the functionality of the hardware. The overall system and its performance should be tested by a System Test. The development process should be finalized by the

implementation of a demo to showcase the device in action in some of the cases of the application scope mentioned before.

3.2. Hardware Specification

- 2 MCUs: the ESP32 and the Arduino.
- An accessible USB programming interface of the ESP32.
- An accessible USB programming interface of the Arduino.
- A chargeable battery as the power source.
- Charging is possible via the USB interfaces of both MCUs.
- The USB interfaces of the MCUs can be safely connected to different charging devices or PCs simultaneously.
- A main switch to power the entire system. When the switch is open, charging is still possible.
- Each MCU has a separate power switch.
- A separate voltage converter for 5V output.
- A separate voltage converter for 3.3V output.
- Voltage dividers connected to the ADCs of both MCUs to measure the 5V and 3.3V outputs as well as the battery's voltage.
- The 10 keys provided on the PCB can be accessed from both MCUs via a port expander.
- The inertial sensor BMA150 provided on the PCB can be accessed from both MCUs.
- An additional inertial sensor that can be accessed from both MCUs.
- A buzzer that can be accessed from both MCUs.
- 2 common anode RGB LEDs that can be accessed from both MCUs via a port expander.
- An OLED display that can be accessed from both MCUs.
- An 868 MHz transceiver that can be accessed from both MCUs.
- Serial communication between the 2 MCUs.

- All modules that provide the I2C interface are connected to the I2C bus.
- All wireless antennas should have minimum obstacles on the path towards the receiving side.
- An external debug interface that provides connection to both MCUs via their serial interfaces, to all voltage sources as well as to the I2C bus.
- It should be possible to unplug each major component to replace it without desoldering.
- The PCB remains detachable from the wheel's body.
- Wires should be as short as possible, yet allowing access to each component for the maintenance purpose.

3.3. Functional Requirements

Modes of operation

- Absolute mouse
- Relative mouse
- Keyboard
- PC Gamepad
- Universal Joypad

Detailed features

- One MCU should act as the primary, the other as the secondary one. Each of the MCUs should be able to act as the primary. When one MCU is switched off, the other should automatically take control over the system.
- Menu navigation using the OLED display and the keys.
- LEDs should indicate the state of the device.
- The inertial module (MPU-9250) should be used to obtain the Roll, Pitch, Yaw (RPY) values.
- The MPU-9250 should be calibrated on demand. The calibration values should be stored in the EEPROM.

- The RPY values should be used to implement the motion-based features of the Mouse, Keyboard, PC Gamepad and Joypad modes.
- Mouse, Keyboard and PC Gamepad commands should be transmitted via the USB interface of the Arduino and via BT (e.g. to a remote Arduino).
- Joypad commands should be sent via the USB serial, BT, Wi-Fi (TCP/IP, UDP), 868Mhz transceiver, and the debug interface. The way of transmission should be selected from the menu.
- The Keyboard mode should provide various configurations (mapping of the keyboard keys to the keys on the device). The configuration should be selected from the menu.
- The PC Gamepad mode should provide various configurations (mapping of the keys and the RPY values of the device to the PC Gamepad commands). The configuration should be selected from the menu.
- Wi-Fi APs should be discovered, selected and stored in the EEPROM.
- On system startup, the device should automatically try to connect to the first available Wi-Fi AP.
- Manual connection to a stored Wi-Fi AP through the menu.
- TCP/IP, UDP client/server configurations should be managed and stored in the EEPROM.
- Received TCP/IP and UDP messages should be immediately displayed on the OLED screen.
- BT devices should be discovered, selected, paired and stored in the EEPROM.
- A specific BT device should be selected for connection from the menu when necessary.
- A standard communication protocol should be designed to transmit all the messages and commands mentioned in this list.
- Debug messages should be transmitted via the dedicated interface during normal operation.

3.4. Nonfunctional Requirements

- The system should respond to a keypress within 10 ms.
- Keyboard press, mouse click, and gamepad click events should be delivered to the remote PC within 30 ms.
- Motion commands (e.g. mouse cursor position or gamepad axis/rotation value) should be delivered to the remote PC within 30 ms.
- Joypad frames should be delivered to the remote devices at a rate of 16 Hz.
- TCP/IP and UDP messages should be sent to the target within 100 ms.
- Keyboard configurations, Gamepad configurations, Wi-Fi Access Points as well as BT devices should be managed (added, deleted, modified) using the UI of the device.
- For the maintainability purpose, the system should be implemented using the official libraries available in the Arduino IDE.

4. Requirements Analysis And Design

Before the implementation, a thorough analysis was performed to find out the optimal ways to approach the goals specified in the previous chapter as close as possible within the time constraints applied to this project. This chapter tells about the decisions, that were made concerning the hardware as well as the software design, and provides the reasoning behind them.

4.1. Hardware Design

4.1.1. Layout

Hardware design started from defining a layout of the components, that would comply with the requirements. When deciding where to place each component, the following requirements were given the highest priority:

- wireless antennas should have minimum obstacles;
- all major components as well as the PCB should be detachable;
- the USB sockets of both MCUs should be easily accessible from the outside;
- wires should be short, but not prevent access to the components.

The remaining requirements were not supposed to affect the layout considerably.

Right from the beginning, one issue had become apparent, namely, the space in the recess on the wheel intended for the PCB was insufficient to fit all the components together. A decision was made to place the components related to powering the device (the battery, the charger module, the voltage converters) inside the body of the wheel, where there was enough free space. This decision would ease the problem of the lack of space as well as introduce an additional feature - the wheel alone, with a detached mainboard, could act as a power source with the outputs of 3.3V, 5V as well as the battery's direct output.

As the next step, the location of the USB sockets for charging the battery as well as programming the MCUs was determined. Since direct access to the on-board USB sockets of the MCUs was limited due to their potential locations, a decision was made to use external micro USB female sockets extended by wires and connected at the other end to other USB connectors, which in turn would be plugged into the MCUs. The hole at the left side of the wheel was considered as the appropriate place for the external USB sockets.

The keys provided with the wheel have their corresponding connection holes on the PCB, and the holes, despite the keys being spread over the entire board, are concentrated at one side of the board. The port expander was placed over the area where the holes are located. This way the overall amount of wiring in the device would be considerably reduced.

Further, the remaining components were arranged on the PCB in various ways to determine the layout that would place each component in an appropriate location according to the high-priority requirements mentioned above as well as the decisions described so far. This, unfortunately, did not lead to success, again due to the lack of space. To work this problem around, the Arduino Micro board was replaced by its smaller analog - the Beetle, which is based on the same MCU (ATmega32u4) and was shortly introduced in the chapter 2. This way, however, the number of accessible pins on the ATmega32u4-based MCU was reduced, further limiting its capabilities within this project.

Finally, the debug interface (a 14 pin ribbon cable socket) had found its place in the rectangular hole with rounded corners at the backside of the wheel. Originally, there was a button in that hole, which was removed leaving a perfectly suitable place for the debug interface available.

Further, in chapter 5, there will be some figures provided that illustrate the layout described in this chapter.

4.1.2. Modules Interconnection

After defining the final set of involved components, the task was to determine how they connect on a high level of abstraction (without going into details about the exact pin mapping). At this stage, the relevant documentation such as pinouts and datasheets of the involved hardware components was analyzed, which led to the following set of conclusions:

- The ATmega32u4 turned out to have an insufficient number of interrupt-capable pins on the Beetle board. This problem would not be fully solved using the Arduino Micro either, because 4 out of 5 interrupt-capable pins of the board coincide with the I2C (SCL, SDA) and the Hardware Serial (Rx, Tx) pins, which are assigned other important functions. Thus, the Beetle could not be connected to all the interrupt pins of the peripherals.

- The specificity of the SPI communication protocol made it hard to enable the master role for both MCUs without additional circuitry. This way, the RFM69 transceiver, which can be accessed only via SPI, could be connected to only one of the MCUs.

The aforementioned conclusions led to the following decisions:

Despite the requirements, the ATMega32u4-based MCU (the Beetle) was limited in its functionality, namely, it would not be connected to the RFM69 transceiver, the buzzer as well as to the interrupt pins of the other peripheral modules. Nevertheless, the Beetle would be connected to the I2C bus enabling access to the vital modules of the system such as the port expander, the inertial sensor and the OLED display as well as to the ESP32 via additional Serial interface. This way the board would have the potential to perform most of the stipulated tasks, although in the polling mode. The ESP32 was assigned the primary MCU role because it could be fully connected to all the peripherals.

Based on the requirements from chapter 3 and the decisions mentioned in this chapter, the design of the interconnections of the modules was concluded as shown in Fig. 4.1.

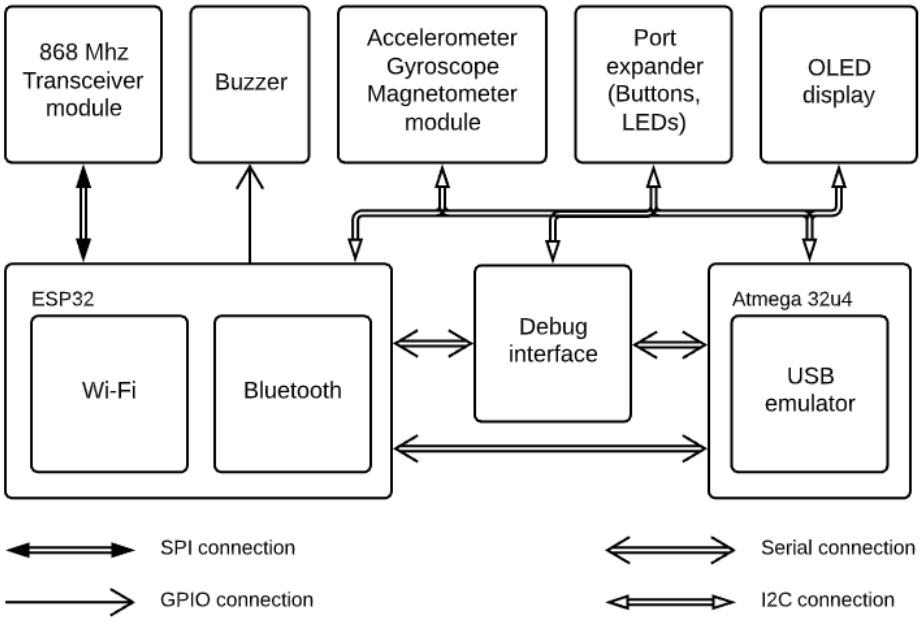

Figure 4.1.: Design approach diagram: hardware interconnection

4.2. Software Design

After the hardware design was determined, the software-related aspects of the yet to be developed system were analyzed. This process included analysis of risks, assessment of

the workload and subsequent determination of the optimal design strategy as well as some implementational decisions.

4.2.1. Analysis Of Libraries

Potential software libraries were examined that could be utilized within this project. The following criteria ordered by priority were considered for the selection of the appropriate ones:

1. Availability in the Library Manager of the Arduino IDE.
2. Coverage of necessary features.
3. The amount of the program memory occupied.

At this, stage the first potential risk arose - the 32KB of the program memory of the ATmeage32u4 MCU turned out to be a tough space constraint. This fact proportionally increased the estimated efforts (correspondingly time and workload) needed to meet all the functional requirements. Besides of that, the standard libraries for handling of the Bluetooth Low Energy (BLE) functionality of the ESP32 required around 70% of the program memory of the ESP32, which made it impossible to use them as they are. For this reason, implementation of the BT-related requirements was postponed until there would be sufficient time left for finding alternative ways of handling it.

The implemented libraries will be introduced in more detail later, in chapter 6.

4.2.2. Impact Of Hardware

The hardware-specific decisions mentioned in the previous section were taken into consideration as they introduced new constraints applied to the software design. To be more specific, the fact that the ATmega32u4-based MCU (the Beetle) would not be fully connected with the rest of the peripherals, implied strong divergence of its potential source code from that of the ESP32. This in turn further increased the estimated time of the implementation and, consequently, the risk to not fit in the time constraints.

4.2.3. Architecture

The roles of the MCUs in the system

The observations mentioned above supported by those from the hardware design section led to the following major architectural decision:

Despite being connected to some of the peripherals via I2C, the Beetle was assigned a limited role. As such, the module would be responsible exclusively for the handling of the USB emulation functionality, namely the Mouse, Keyboard, and PC Gamepad modes. This, however, given enough time, could be improved in the future.

The overall architecture

Following the previous decision, the overall architecture was designed as monocentric, with ESP32 as the primary MCU responsible for coordination of the system and the Beetle acting only as a USB interface to the PC. The next requirements were taken into account:

- Responsiveness - the system should react to the user's input fast enough for a positive user experience.
- Maintainability - the code should be easy to read and modify, implementation of additional features should not lead to a major refactoring of the system.
- Sensors should be sampled with a sufficient rate.
- Other functional and to the extent possible nonfunctional requirements mentioned in the chapter 3.

According to the aforementioned requirements, the following concept was developed:

Each major node in the system (e.g. mode of operation, a hardware module, menu, etc.) has its dedicated handler class. Besides, there is a central node, e.g. a System Handler, that coordinates operation of the system at a high level, namely:

- Initializes the system.
- Initializes and finalizes nodes (e.g. when the mode of operation changes).
- Updates the system:
  - Regularly invokes update of the nodes, which handle the sensors.
  - When needed, invokes update of other nodes depending on the mode of operation.

Furthermore, there is a User Input Handler (e.g. a menu), which changes the state of the system by telling the System Handler what should be the next state (e.g. mode of operation). Finally, the system is updated with a certain rate, triggered e.g. by a timer interrupt or in an infinite loop with the maximum possible speed.

Considering the scale of the system, the OOP paradigm was given the preference as it would ease further maintainability of the code. Since the nodes (correspondingly, the classes) of the system would handle resources such as a sensor, a communication interface, or a mode of operation, and since each resource was present only in one exemplar, the Singletone Pattern was chosen as the appropriate one.

The figures 4.3 and 4.2 illustrate the software design concepts discussed in this section.

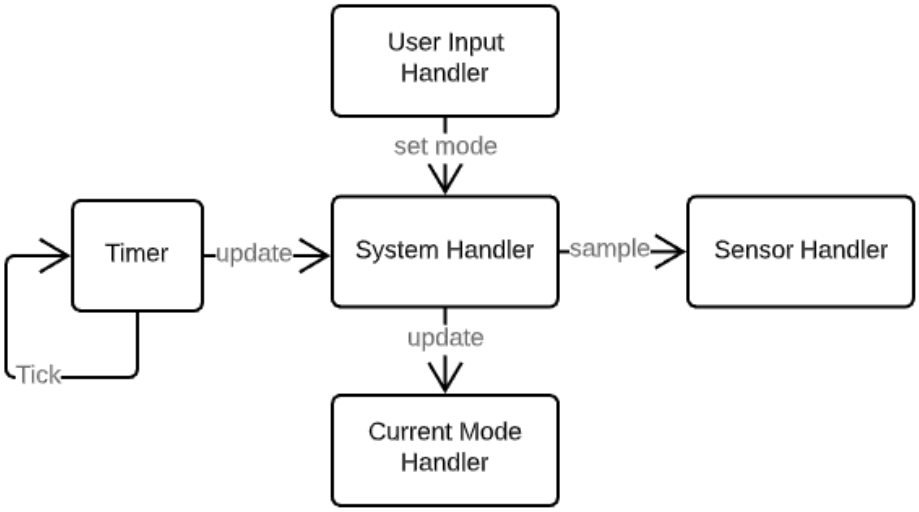

Figure 4.2.: Design approach diagram: software concept at a high level of abstraction

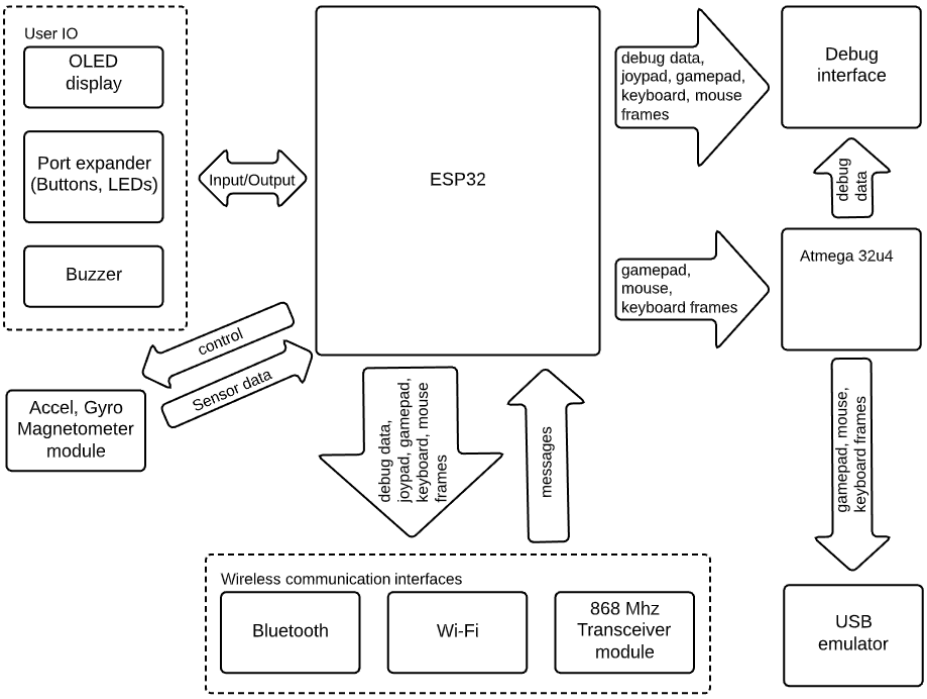

Figure 4.3.: Design approach diagram: data flow

5. Hardware Implementation

This chapter guides the reader through the process of development of hardware, which was based on the requirements listed in the chapter 3 and following the corresponding design described in the chapter 4. This process can be split into two major stages:

1. Circuitry for powering the system, charging the battery and programming the MCUs;
2. Mainboard circuitry (interconnection of the MCUs and the peripheral modules);

The following sections will describe each stage in more detail accompanied by corresponding illustrations. Finally, pictures of the final state of the device will be provided.

5.1. Power And Programming Circuit

Following the hardware specifications, a circuit diagram had to be developed, that would allow the next features:

1. Supplying all the modules with their corresponding voltages (3.3V, 5V);
2. Switching the power of the entire system by a main switch;
3. Switching each MCU's power by a separate switch;
4. Measuring the voltages of the 3.3V and 5V outputs as well as the battery by ADCs of the MCUs;
5. Programming the MCUs via their corresponding external micro USB sockets;
6. Charging the battery via both external micro USB sockets.

The first 2 features among those listed above did not require any particular efforts and were solved straightforwardly:

- The 5V step-up DC-DC converter mentioned in the list of involved hardware (chapter 2) was fed directly from the battery supplying the system with a regulated 5V output;
- The 3.3V step-down converter took 5V as an input providing a regulated 3.3V output;

- The main switch was placed right after the positive terminal of the battery hence being able to switch power of the entire system;

The 3rd feature required some investigation to identify which power input is suitable for each MCU. According to the documentation of the ESP32 [11], the recommended way to power the module is either via its USB socket by a regulated 5V input or, alternatively, via the external power input pin by a voltage source with the output range of 7-12 Volts. The Beetle board can be powered either via its USB input or the 5V pin, in both cases by a regulated 5V power source. Considering these specifications, both MCUs were powered via their USB inputs by the regulated 5V supply, hence their corresponding switches were placed between the supply and the USB sockets of the MCUs.

The feature number 4, namely measurement of the voltages, was initially approached simply and intuitively: three voltage dividers consisting of two 100 Ohm resistors each were connected to the power sources, and their measurement points were shared by both MCUs. As a result, the ADC pins of the MCUs would be electrically connected, e.g. the three ADC pins pin of the ESP32 were connected to their corresponding peers on the Beetle). This setup was tested on a breadboard revealing a problem: when one of the MCU was switched off, its ADC pins caused interference with the ADC pins of the other MCU leading to distortions in voltage measurements. The problem was solved by decoupling the ADC pins of the MCUs by simply implementing 2 sets of voltage dividers. Additionally, the measurement point of the battery's voltage was placed after the main switch, thus preventing potential current leakage through its voltage divider when the main switch is off.

The 5th was solved simply by forwarding the data signals of the external micro USB sockets to their corresponding pins on the MCUs.

The last feature, namely charging the battery via both external micro USB sockets, implied that the VBUS pins of the sockets were supposed to be in a way interconnected allowing them to act as a power source for a single battery charger. This task introduced a potential risk of reverse current when, e.g. 2 different PCs (ar any other devices, power sources) are connected to the USB sockets of the wheel simultaneously. The problem had two potential approaches:

- To use a mechanical switch allowing only a single USB socket to acts as a power source for the battery charger;
- Implementing a protection circuit at the level of electronics.

Despite its simplicity, the first approach would make the device less user-friendly, hence, the second approach was given the preference. To keep the design as simple as possible with minimum additional components involved, a classical reverse current protection approach was considered - placing diodes after the VBUS pins of the USB sockets. This would introduce a forward voltage drop across the diode further reducing the input voltage of the battery

charger. However, the tests on a breadboard revealed that the resulting input voltage of the charging circuit was within its allowed range and the battery was charging successfully.

The figure 5.1 illustrates the circuit diagram that was based on the description above and was implemented in the device.

5.2. Mainboard Circuit

Following the design decisions, namely the roles of each component in the system as well as their layout, the next step was to identify the actual pin connections. At this stage, the following requirements ordered by priority were taken into account:

1. The peripheral devices should be connected to their corresponding MCUs in a way that necessary functionality could be utilized (interrupt pins, communication pins, etc.).
2. The pins of the MCUs should be used optimally such that the functionality of each involved pin is utilized in the best possible way.
3. The pins assigned predefined functions such as UART pins of programming interfaces should be handled with particular care to avoid interference during programming of the boards.
4. Wires should be as short as possible yet allowing access to all components for the maintenance purpose.

ATmega32u4 pins

Since the Beetle (ATmega32u4) was limited in its functionality, its pin connections were straight forward. The I2C-capable pins of the board served their dedicated purpose. The hardware serial pins were assigned the role of the debug interface, which is described in the next section. Two out of the three remaining digital IO pins were used to communicate with the ESP32 and hence were connected to the level converter between the two MCUs. The two pins were selected based exclusively on their locations such that soldering becomes easier. The three available analog input pins were used for voltage measurement, which was described in the previous section.

ESP32 pins

Assignment of functions to the pins of the ESP32 required more investigation to identify capabilities, limitations and best possible ways to utilize the pins. For this purpose, the documentation of the board was studied [11], which led to the following decisions:

- The pins GPIO1 and GPIO3 are connected to the programming UART interface and hence, were left untouched.
- The pins GPIO21 and GPIO22 were used to serve their dedicated purpose - I2C.
- The pins GPIO18, GPIO19, GPIO23, and GPIO32 were used for the SPI according to their intended purpose.
- Since the UART modules of the board can be logically mapped to any of the available pins, the pins used for serial communication with the Beetle as well as for the debug purpose were selected based on their locations to ease soldering.
- Since all other GPIO pins are ADC and interrupt-capable as well as can serve as IO, their roles were assigned, again, based on their locations to make the wires short and optimally aligned.

The circuit diagram of the main board implemented in the device is illustrated by the Fig. 5.2. For simplicity, some parts of the diagram such as powering, interfacing of the LEDs and the keys as well as the debug interface are excluded from the diagram and are presented in the figures 5.1, 5.3 and 5.4 respectively. The tables 5.1, 5.2 and 5.3 list corresponding pin connections.

5.3. Complete Device

After designing the circuit diagrams, the process of assembly had begun. This involved fixing the components in their corresponding places according to the design from the chapter 4 followed by wiring and soldering according to the circuit diagrams. The outer look of the final device is illustrated in Fig. 5.5, whereas the overview of the inner components as well as a closer look at the USB and debug interfaces are provided by the Fig. 5.6.

Atmega 32u4 (5V)	Periphery / function	ESP32 (3.3V)
D9 (SW Serial Tx)	<->Level shifter <->	**Rx2 (D16)**
D10 (SW Serial Rx)	<->Level shifter <->	**Tx2 (D17)**
SDA (3.3V)		**D21**
SCL (3.3V)	I2C SDA bus	**D22**
	5V step-up voltage divider 1	**D34**
	3.3V step-down voltage divider 1	**VN (D39)**
	Battery voltage divider 1	**VP (D36)**
A0	5V step-up voltage divider 2	
A1	3.3V step-down voltage divider 2	
A2	Battery voltage divider 2	
Rx	32u4 Debug Rx	
Tx	32u4 Debug Tx	
	RFM69 SPI MOSI	**D23**
	RFM69 SPI MISO	**D19**
	RFM69 SPI SCK	**D18**
	RFM69 SPI CS	**D32**
	RFM69 Interrupt	**D33**
	Port expander Interrupt B	**D25**
	Port expander Interrupt A	**D26**
	MPU9250 Interrupt	**D14**
	BMA150 Interrupt	**D13**
	ESP32 Debug Rx	**D2**
	ESP32 Debug Tx	**D4**
	Buzzer (BJT Base through 10kOhm)	**D15**

Table 5.1.: Pin connections: microcontrollers' perspective

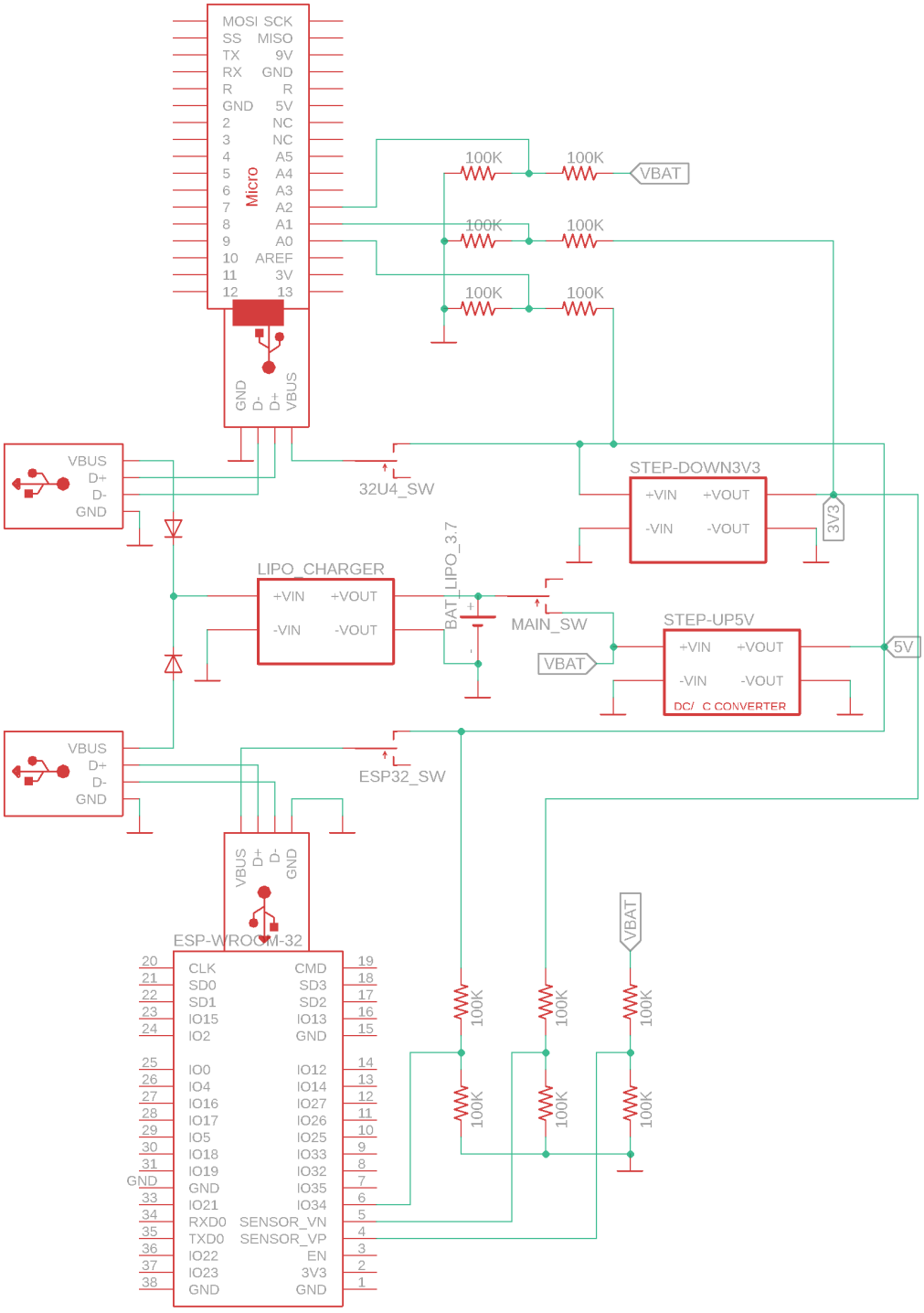

Figure 5.1.: Circuit diagram: power and programming

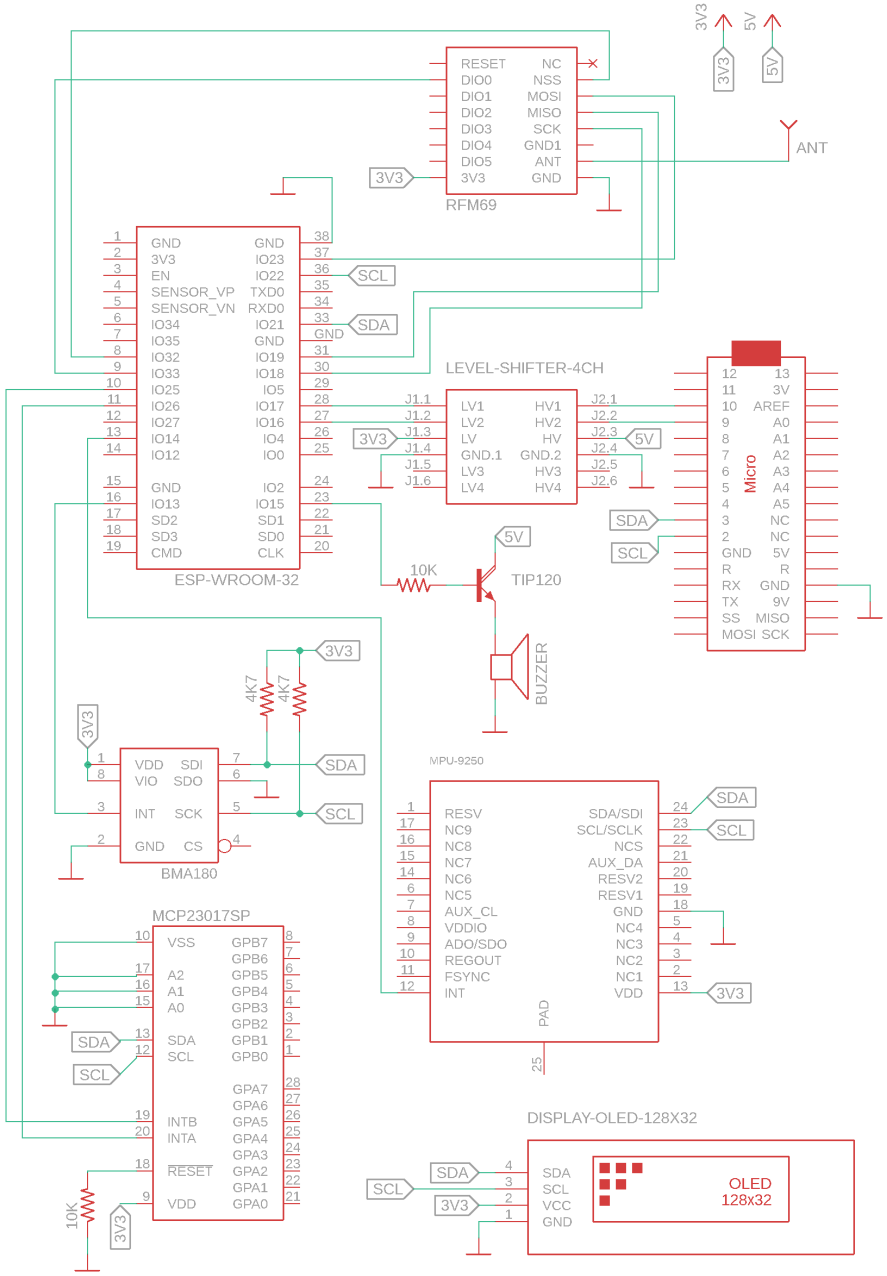

Figure 5.2.: Circuit diagram: main board (incomplete)

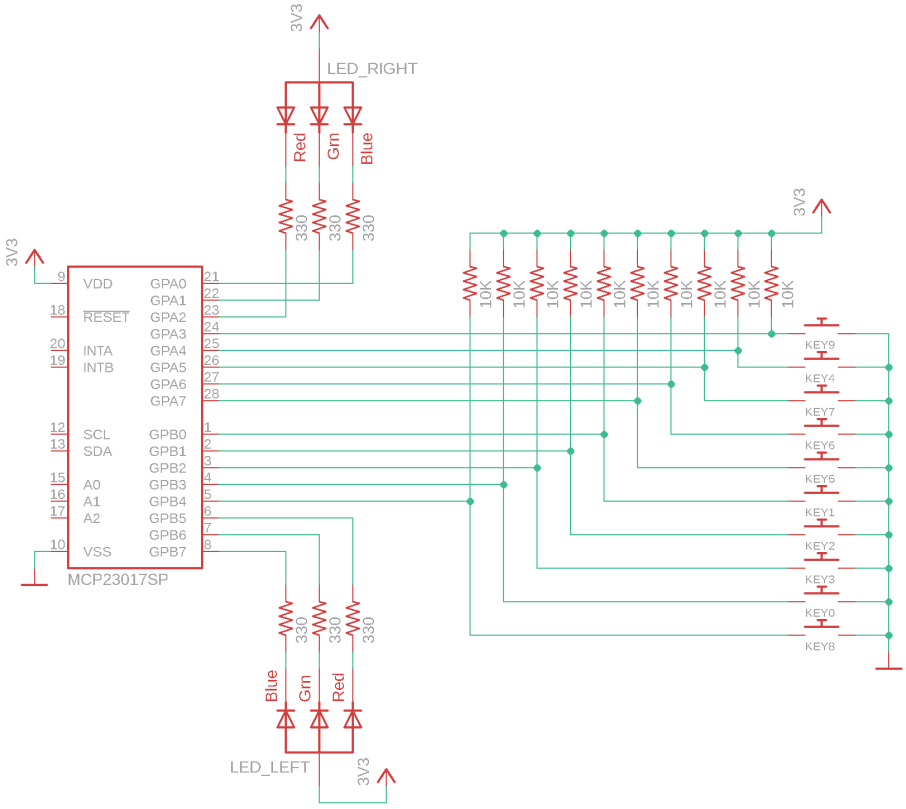

Figure 5.3.: Circuit diagram: buttons and LEDs connection

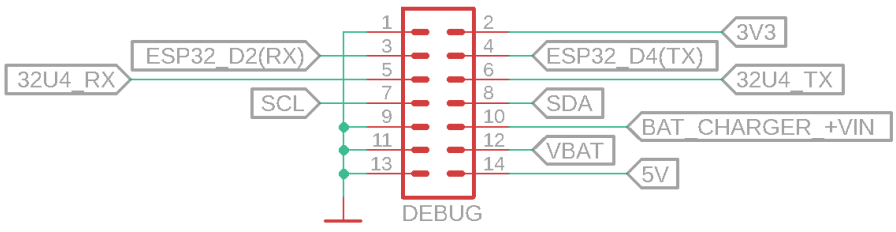

Figure 5.4.: Circuit diagram: 14-pin debug interface

MCP23017 Port expander pin	Where connected
GPB0	Key 1
GPB1	Key 2
GPB2	Key 3
GPB3	Key 0
GPB4	Key 8
GPB5	left LED R through 330Ohm
GPB6	left LED G through 330Ohm
GPB7	left LED B through 330Ohm
GPA0	right LED B through 330Ohm
GPA1	right LED G through 330Ohm
GPA2	right LED R through 330Ohm
GPA3	Key 9
GPA4	Key 4
GPA5	Key 7
GPA6	Key 6
GPA7	Key 5
A0	GND
A1	GND
A2	GND
NReset	3.3V through 10kOhm
VDD	3.3V
VSS	GND
SCK	SCK bus
SDA	SDA bus
INTA	ESP32 D26
INTB	ESP32 D25

Table 5.2.: Pin connections: port expander (keys, LEDs)

Function	Pin		Pin	Function
GND	**1**		**2**	3.3V
ESP32 Debug Rx (D2)	**3**		**4**	ESP32 Debug Tx (D4)
32u4 Debug Rx (Rx)	**5**		**6**	32u4 Debug Tx (Tx)
I2C SCL	**7**		**8**	I2C SDA
GND	**9**		**10**	Charger input voltage (USB_Vbus - Vdiode)
GND	**11**		**12**	Battery
GND	**13**		**14**	5V

Table 5.3.: Pin connections: debug interface perspective

Figure 5.5.: The outer look of the final device

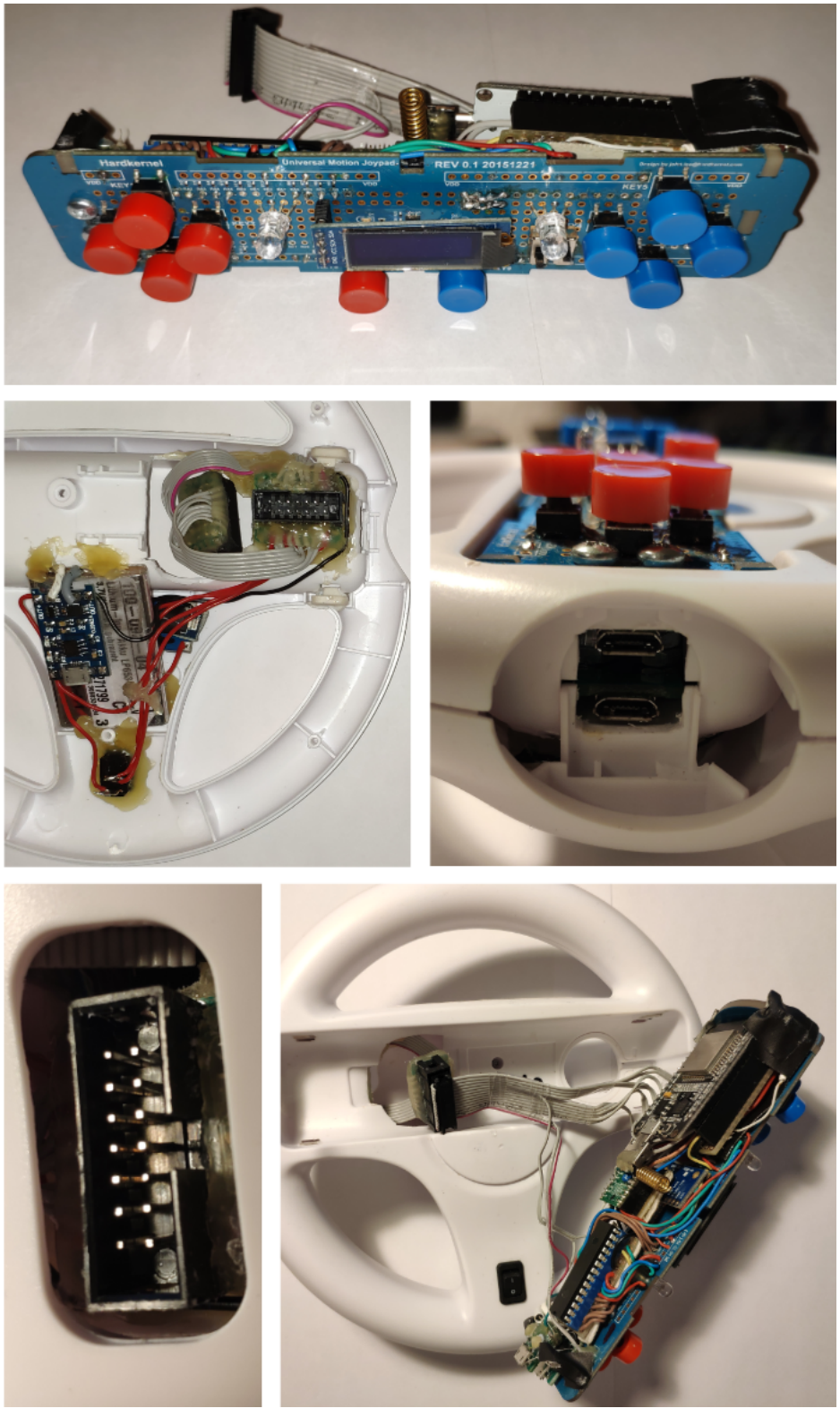

Figure 5.6.: The final device from the inside

6. Software Implementation

This chapter describes the major part of this project - the development of software. This process can be split into three stages:

1. Definition of the protocol for communication with remote devices.
2. Development of the primary system on the ESP32.
3. Implementation of the USB emulation on the Beetle (ATmega32u4).

The following three sections provide details on the development of each stage.

6.1. Communication Protocol

The purpose of the protocol within this system was to provide a structured and standardized way of communication between the wheel and the remote devices. Since there was no specific requirement applied to the structure, a basic protocol with the following frame format was defined and implemented in the Wheel:

Byte 1	Byte 2	Byte 3	Byte 4	Bytes [5 : N]
start of the sequence ('K')	frame length in bytes (N)	primary command	secondary command	payload

Figure 6.1.: Communication protocol: frame format

As can be seen from Fig. 6.1, frames start with a byte with a fixed value 'K', which stands for "KLAB". The second byte indicates the number of bytes in the frame or, in other words, the length of the frame. The following two bytes are used for sending primary and secondary commands. The primary command is mandatory and is used to indicate the purpose of the message, e.g. a mouse command, a debug message, etc. The secondary command is optional and can be used, e.g. to add some supplementary information about the primary command or the payload. The bytes starting from 5 are optional and can be used for passing

some payload, which has the size limit of 252 bytes such that the total size of the frame does not exceed 256 bytes.

6.2. ESP32

This section described the development of the main part of the software system on the Wheel, correspondingly - the system on the ESP32. First, an overview will introduce the major nodes (classes) of the system and their relations as well as briefly explaining their roles. Further, each node will be discussed individually in more detail providing information about their structure and other technical aspects specific to its functionality.

6.2.1. System Overview

Polling vs. interrupt

Before developing the system, the two possible approaches for the system update were considered: the polling and the interrupt based methods. The latter one was initially preferred since it was supposed to guarantee timely reaction to the system update, e.g. when a key is pressed or the inertial sensor has produced a new set of samples. To test this approach, a set of experiments was performed leading to the following results:

1. The inertial sensor module (MPU-9250) was configured to be sampled at the rate of 60 Hz invoking interrupts when new samples were ready. The handling MCU was the ESP32. The interrupts were correctly raised and the Interrupt Service Routine (ISR) was invoked, however, the ISR could not be used for immediate reading of the sensor via the I2C since such attempts were causing the ESP32 to malfunction and reboot. The ISR could handle only setting a Boolean flag indicating that a new sample is ready to be read. This way, the rate of reading the sensor did not depend on the rate of the interrupts, and the sensor could be read only when the main thread reached the corresponding instruction, which in practice did not make this approach considerably different from polling in terms of performance.
2. The port expander (MCP23017) was set up in the interrupt mode similar to the inertial sensor from the previous experiment. The interrupts were raised on keypresses successfully, however, the process of cleaning the interrupt caused problems. Due to the bouncing, the cleared interrupts were repeatedly invoked hence requiring some time after which the interrupt could be finally cleared. During this “waiting” time, no other key presses could be detected making the port expander unresponsive. This

phenomenon made the interrupt-based keypress reading not only lose its advantages in terms of response time but also made this process more error-prone.

The two experiments described above made the interrupt-based implementation of the system too risky considering the complexity of the system and the time constraints.

Besides, due to the target application scope of the device and the inertial sensor being the core of its manipulation method, the system was supposed to operate in the streaming mode most of the time. In this context, streaming implies e.g. transmitting the position of the mouse pointer, PC gamepad inertial commands, universal joypad inertial commands, etc. with a fixed rate. This way, the polling of the sensors could be synchronized with the transmission of the commands leading to the behavior similar to the interrupt-driven update at a certain rate.

Based on the observations described above, a decision was made to follow the polling approach as it would in practice behave similarly to the interrupt-based one but being more stable, less error-prone and straight forward to implement.

The system as it was developed

The system on the ESP32 was developed according to the design described in chapter 4. Each node (peripheral module, mode of operation, communication interface, any other major logical unit) of the system has got its dedicated handler class. All nodes are controlled by the primary node - the System Handler. The system update is invoked at a certain rate. On system update, all sensors are read via polling and the nodes responsible for the current mode of operation are updated.

The system was developed using the OOP paradigm, and specifically, the Singleton pattern, which means each node (class) has only a single static instance that can be accessed from any other node without creating its instance or storing its reference.

Major nodes in the system

The following are the major nodes of the system and their roles in brief:

- **Main** is a symbolic name for the set of the global methods.
- **SystemHandler** is responsible for a high level coordination of the system.
- **Menu** handles navigation in menu and selection of modes.
- **GamepadHandler** handles the PC Gamepad mode of operation.

- **MouseHandler** handles the Mouse mode of operation.
- **KeyboardHandler** handles the Keyboard mode of operation.
- **JoypadHandler** handles the Universal Joypad mode of operation.
- **ServiceHandler** is responsible for service features such as calibration and others.
- **SensorInfoHandler** displays sensor values (inertial, voltages) on the screen.
- **StorageHandler** provides functionality to read and write to the flash memory.
- **WiFiConnectionHandler** handles selection of an AP to connect to.
- **WiFiHelper** provides all functions related to Wi-Fi.
- **USBSerialHelper** provides helper functions for the USB serial communication.
- **Power** handles battery related functionality.
- **MPU** handles the inertial sensor's functionality.
- **Port** handles key presses and LEDs.
- **OLED** provides methods for manipulation of the display.

The Fig. 6.2 illustrates a high-level class diagram of the system. The dashed arrows symbolize static singleton access.

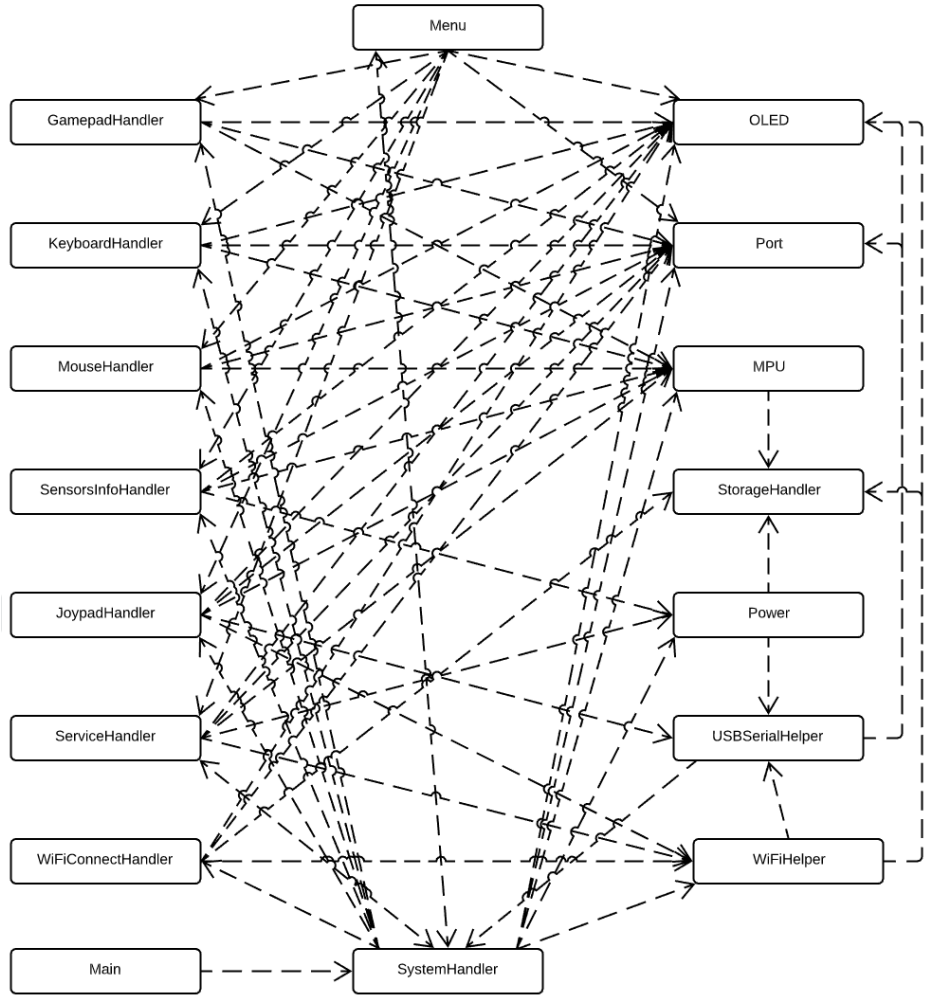

Figure 6.2.: ESP32 class diagram: overview

6.2.2. Main Routine

The main routine is performed by the **Main** node. “Main” is a symbolic name for the global static methods, hence, it is not an actual class. The Main node contains methods for initialization of the system and its regular update. On power-up, the method **setup()** is invoked, which in turn initializes the SystemHandler node. Afterward, the method **loop()** is called in an infinite loop. The later method invokes the update of the SystemHandler node, either with a maximum rate of 100 Hz or with the maximum possible rate depending on the hard-coded configuration. At the time of delivery of this paper, the 100 Hz configuration was selected. Here, the rate of 100 Hz was chosen to fit the sampling rate of the inertial sensors with its default configuration as well as to be able o respond to keypresses within 10 ms. Besides that, such rate allows sufficient time for processing of the routines yet keeping the system responsive. The specified rate is not always the actual one, because depending on the mode of operation some delays can be introduced, which will slightly reduce the rate of update. More details on the timing characteristics of the system will be provided further in this paper.

6.2.3. System Handler

The system is handled at a high level by the **SystemHandler** class, which is the master node of the system. It is initialized once at the system start and regularly updated by the main routine as was described before. The node has a state machine, whose states are represented by the enumeration class **Mode**.

On initialization, the SystemHandler initializes the nodes **OLED**, **Port**, **MPU**, **Power** and **WiFiHelper**, displays an introductory animation on the OLED screen, attempts to connect to the first available Wi-Fi AP among those stored in the memory, and finally enters the state “MENU” by initializing the **Menu** node. Any further states are set by the Menu node based on the user’s input (selection of modes from the menu).

On update, the SystemHandler first updates the **Port**, **MPU**, **Power** and **WiFiHelper** nodes, and then the node, which corresponds to the current state (mode of operation).

On state change, it finalizes the node that corresponds to the previous state and initialized the new one. The role of initialization, update, and finalization of the nodes will be explained further, individually for each node.

The two additional roles of the SystemHandler are handling the commands **quit** and **show status**, which are issued by the user via pressing the corresponding keys. The former command is used to quit the current mode of operation and the latter shows on the OLED screen some information about the status of the system such as Wi-Fi connection, IP address and other. When the “quit” is issued, the SystemHandler changes its state to the “MENU”.

6.2.4. Main Menu

The menu functionality of the device is handled by the **Menu** class, which is responsible for the navigation in the menu of the device and selection of the entries. Similar to the SystemHandler, it has a state machine, whose states are represented by the items of the enumeration class **MenuEntry**. The states are arranged in a two-level hierarchy, where the upper level (the parent) states correspond to the modes of operation of the system, whereas the lower level (the child) states correspond to the sub-modes of operation. For example, in the context of this system, when a mode of operation is the **Mouse**, the sub-modes can be the *Relative* or *Absolute*, which stand for the **Relative Mouse** and the **Absoule Mouse** modes respectively.

The Structure

The structure of the entire menu with a short description of the items is provided below:

- **CONNECT** - connectivity functions
  - **WIFI** - connect to a Wi-Fi AP
- **GAMEPAD** - mode of operation: PC Gamepad
  - **X-Y MODE** - sub-mode: keys and X,Y axes
  - **RPY MODE** - sub-mode: keys and X,Y,Z rotations
- **MOUSE** - mode of operation: Mouse
  - **ABSOLUTE** - sub-mode: absolute
  - **RELATIVE** - sub-mode: relative
- **KEYBOARD** - mode of operatiot: Keyboard
  - **NAV. MODE** - sub-mode: navigational key set
  - **GAME MODE** - sub-mode: gaming key set
- **JOYPAD** - mode of operation: Universal Joypad
  - **WI-FI** - sub-mode: transmission via Wi-Fi
  - **RADIO** - sub-mode: transmission via the wireless transceiver
  - **USB INT.** - sub-mode: transmission via the USB interface
  - **DEBUG INT.** - sub-mode: transmission via the Debug interface

- **SERVICE** - service functionality
  - **CALIBR. MPU** - calibrate the inertial sensor
  - **WI-FI** - manage the Wi-Fi related settings
  - **BATTERY** - manage the power related settings
- **SENSORS** - sensors mode
  - **RPY** - sub-mode: displaying RPY on the OLED screen
  - **VOLTAGES** - sub-mode: displaying the voltages of the 3.3V, 5V output and the battery on the OLED screen

Here, the items in bold match the texts that appear on the OLED screen when the corresponding items are being displayed.

Navigation

Navigation in the menu is performed using the keys. The following list shows the commands involved in menu navigation, their assigned keys as well as description.

- **Next** ***(key 7)*** - is used to switch to the next item within the same level of the menu's hierarchy (switching among the parent states, or among the child states of the same parent).
- **Previous** ***(key 5)*** - is used to switch to the previous item within the same level of the menu's hierarchy.
- **Return** ***(key 4)*** - is used to navigate up the hierarchy (from a child state to its parent).
- **Ok** ***(key 6)*** - is used to navigate down the hierarchy (from a parent state to its first child). If the current item does not have a child, then **Selection** of this item will be issued, which is described further.

The mapping of the labels to the keys is illustrated by Fig. 6.3.

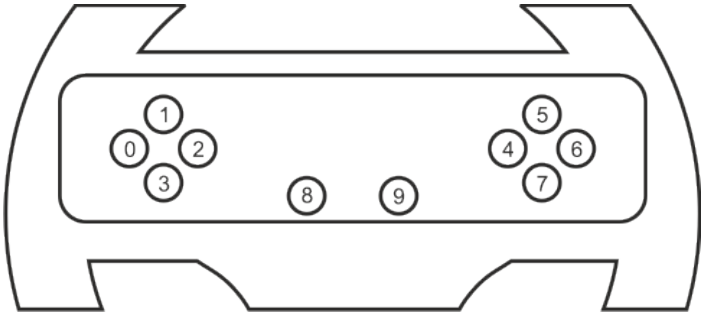

Figure 6.3.: The labels of the keys

Selection

As was mentioned before, when the command **Ok** is issued from a sub-state (child menu item), this will lead to a selection of the mode of operation assigned to that item. For example, if current menu entry is **Absolute Mouse**, then pressing the **Ok** key will lead to the following behavior:

1. The **Menu** node will set the state of the **Mouse** node to the **Absolute**.
2. The **Menu** node will notify the **SystemHandler** node about the next mode being the **Mouse**.
3. At the following system update, the **SystemHandler** will finalize the **Menu** mode and initialize the **Mouse** mode.
4. The device will begin to operate in the **Absolute Mouse** mode.

6.2.5. Storage Handler

In the Wheel's system, the settings and configurations, such as Wi-Fi APs, sensor calibration values, and others are stored in the flash memory of the ESP32. This memory can be accessed using the EEPROM library provided by the Arduino IDE. The library provides functions for reading and writing bytes to specified addresses. The advantage of such capabilities is the low-level access to the flash memory, which gives freedom and flexibility to the developer. However, such freedom requires special care when it comes to specifying the correct addresses of the bytes to be read or written since wrong addresses may cause data loss or reading wrong values which in turn may lead to incorrect operation of the system. Besides, storing and reading multi-byte data types (e.g. float, int, etc.) byte by byte requires additional care as well. This implies a higher risk of errors and more sophisticated software, which increases the overall complexity and reduces the maintainability of the system.

The **StorageHandler** class solves the problems stated above by providing methods for simplified and structured access to the underlying flash memory without the need to care about the correct addresses and data types. The StorageHandler stores a structure **Storage**, which in turn consists of further structures called **Segments**. Each segment stores a set of values that serve a certain purpose, e.g. calibration values of the inertial sensor. The data of a segment of the Storage can be accessed directly. However, before reading the content of the segment, it has to be loaded from the flash memory at least once. Loading, erasing and saving changes in the segments are done through an instance of the **Segment** class that can be obtained using the method **getSegment()** of the StorageHandler by passing the identity of the required segment as an argument. The identities of the segments are represented by the enumeration class **StorageSegment**.

The StorageHandler uses the aforementioned EEPROM library to access the flash memory. The structure of the Storage is illustrated in Fig. 6.4.

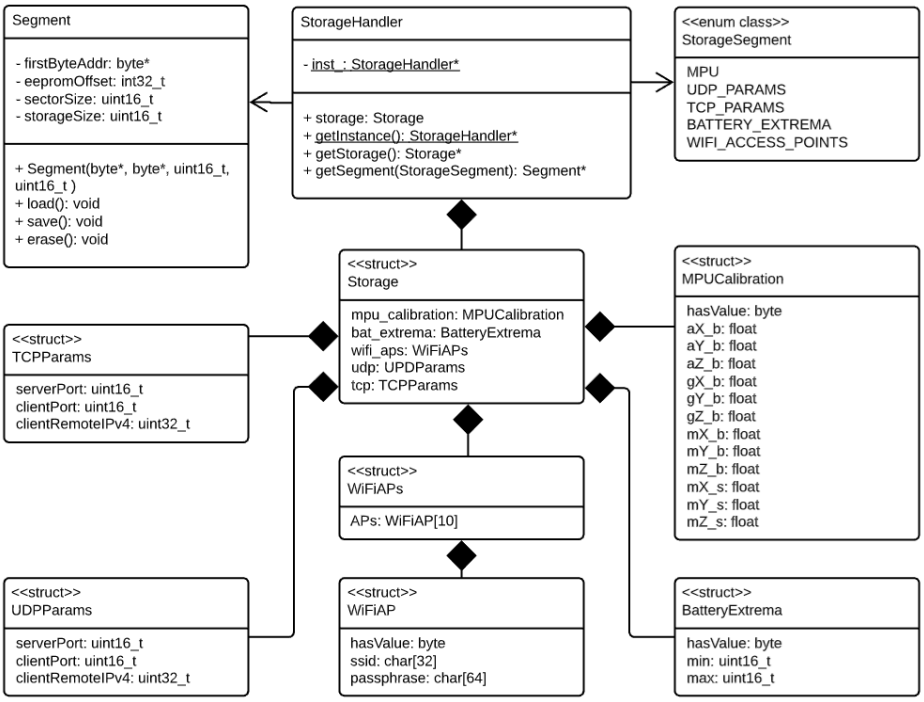

Figure 6.4.: ESP32 class diagram: Storage Handler perspective

6.2.6. Accelerometer, Gyroscope, Magnetometer

The sensor and the library

The inertial manipulation features of the device are based on the MPU-9250 module introduced in the chapter 2, which has built-in Accelerometer, Gyroscope and Magnetometer modules, each providing values in 3 axes. There are several libraries for interfacing the module available in the Arduino IDE. After analyzing them for the provided functionality, preference was given to the one which is called “hideakitai MPU9250”. The library, along with standard features such as configuring the module and obtaining sensor values, provides functions for calibration and even out-of-the-box calculation of the RPY values. The presence of these features would save a considerable amount of time that could be directed to the implementation of the other functionality of the Device.

The MPU class

In this project, the library was not used directly because an additional layer was required that would provide missing functionality. The role of such a layer was assigned to the **MPU** class. The class provides functions for handling the calibration process, saving calibration data in memory, loading and applying calibration data to the sensor using the library. The major role of the MPU class, however, is to implement additional filtering of the Roll, Pitch and Yaw values obtained using the library. The class provides the functions to get the filtered as well as non-filtered values both in floating-point and integer formats. The MPU node is being initialized and regularly updated by the **SystemHandler**. On initialization, the node configures the sensor. On update, it updates the sensor, reads the values of RPY and updates the filters.

The encoded yaw

In addition to the basic RPY, there was a function developed that calculates the value of the encoded Yaw. The difference between the normal Yaw and the encoded Yaw is that the former one takes values in the range of -180 to 180 degrees, whereas the latter stores the absolute degrees of rotation around the Z-axis starting from the initialization of the system. This means the value range of the encoded Yaw is limited only by its data type, which is float in this case. With this feature, the Wheel can be used as a rotary encoder. The name “encoded Yaw” is symbolic and means that the value was obtained by tracking the Yaw angle.

```
void MPU::encode()
{
  float Y_new = getY_f(); // read current yaw in float
  // if overflow through -180 to positive
  if (Y_prev < -90.0f && Y_new > 90.0f)
    Y_encoder += (Y_new - Y_prev - 360.0f);
  // else if overflow through +180 to negative
  else if (Y_prev > 90.0f && Y_new < -90.0f)
    Y_encoder += (Y_new - Y_prev + 360.0f);
  // else no overflow
  else
    Y_encoder += (Y_new - Y_prev);
  Y_prev = Y_new;
}
```

Listing 6.1: Calculation of the encoded Yaw value

Filtering

The filtering functionality is achieved using the Moving Average Filter with the widow size of 10 implemented on a circular buffer. Filtering helped to reduce the noise which was affecting first of all the Absolute Mouse mode of operation. The window size of 10 was selected as an optimal to reach sufficient noise reduction yet keeping the lag within the acceptable limit, which was determined experimentally. The Fig. 6.5 illustrates the time response of the implemented filter to the input of the values of the Roll in a steady state of the device.

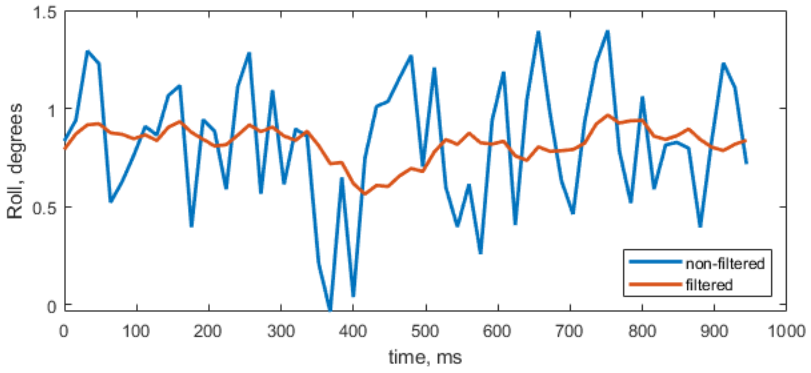

Figure 6.5.: Time response: Moving Average Filter, window size 10

As can be seen from Fig. 6.5, the noise of the original signal has the amplitude range of nearly 1.5 degrees, which in turn is translated to a large distance that, e.g. a mouse pointer would move following those values. The filtering, in turn, reduces the noise considerably.

6.2.7. OLED Display

The OLED display introduced in chapter 2 of this paper is the main output method of the device. It is used for the menu navigation, displaying the status of the device (battery percentage, the current mode of operation, enabled wireless modules, etc.) as well as important messages that arrive from the remote devices.

The library

There are various libraries provided in the Arduino IDE for interfacing such displays. In this project, the library “Adafruit_SSD1306” was selected. It provides all necessary functions to manipulate the display such as printing texts of variable size in any location, drawing bitmaps, pixel manipulation, etc. One of the handiest features is the local buffer which stores each pixel’s value. This way, the content of the entire screen can be first composed and then sent in a single packet to the display improving the overall performance of the system.

The OLED class

The OLED class was implemented to act as an additional layer to access the functions of the library. The class provides functions specific to the device, such as showing the introductory animation as well as displaying menu entries, the current mode of operation, system status, icons, etc. in their dedicated locations and size. Besides that, some functions simplify usage of the standard functions provided by the library, such as clearing a region.

The locking mechanism

The OLED class offers an additional feature - it is possible to lock and unlock the display by calling corresponding functions. If the display is locked, it means that the most recent content is being displayed and cannot be overwritten until the display is unlocked. However, the locked display does not imply the loss of the new content that was supposed to be displayed. During the locked state, the new content is stored in the local buffer of the library and is immediately displayed as soon as the display is unlocked.

Locking the display is used when an important message is displayed that necessarily has to be acknowledged by the user. Within this project, the locking feature was used to show the UDP and TCP/IP messages on the screen according to the corresponding functional requirements as well as to show the system status (Wi-Fi AP, IP address, etc.) when the user issues the **show status** command. In both use cases, the acknowledgment (unlocking)

is performed by issuing the **quit** command. Both mentioned commands were described earlier in the section 6.2.3.

6.2.8. Keys, LEDs

The library

The keys and LEDs of the device are accessed through the port expander module, which was introduced in the chapter 2. The module was interfaced using the library "Adafruit_MCP23017", which provides convenient methods to setup each pin either as an input or as an output, to read and write values from/to the ports as well as to configure interrupts.

The Port class

As in the cases of the previously described modules, there is an additional layer created to access the library's functionality, namely the class **Port**. The Port provides device-specific functions that ease manipulation of the LEDs and access to the states of the keys. As such, an LED can be switched providing only its number and desired color and the state of a key can be checked by providing its number only. Additionally, the class provides the debouncing feature with a configurable debounce period as well as the handling of some predefined key combination clicks. The Port node is being initialized and regularly updated by the **SystemHandler**. On initialization, the node configures the port expander's pins. On update, it reads the port expander's registers, derives the state of each key and stores it in the local variable for further quick access.

Debouncing

Debouncing is implemented simply - if the key is clicked, its next click can be registered only after the debounce period expires. This way, the debouncing functionality works slightly different from its definition, namely, it limits the rate of repeated clicks of the same key and does not work as would be expected if the key has to be pressed and held. This is done intentionally. Since the whole system is polling-based and is updated with sufficiently large intervals, actual debouncing does not appear to be a problem. On the other hand, the polling approach raises another problem - when a key is supposed to be pressed only once, it might be polled several times until the user releases the key. This, in turn, will lead to several registered clicks. The problem of pressing and holding a key is solved by allowing to set the

debouncing duration to zero (or to any other value) by invoking the corresponding method of the Port class. Examples of the use case of this feature:

- In menu navigation, when the user presses and holds a navigation button (e.g. next item), the rate of switching the items should be limited (correspondingly the rate of repeated clicks). This can be achieved by setting a sufficiently large debouncing period.
- In the mouse mode, the user needs to be able to press and hold e.g. the left mouse key to drag an item on the screen of the PC. This can be achieved by setting the debouncing period to zero.

Each node of the system has its own suitable debounce period, which is set when the node is initialized. Debouncing of each key and even some key combinations is tracked individually.

Quit and Status commands

The two commands mentioned in section 6.2.3 have their assigned key combinations. These combinations were defined based on some analysis. Considering the application scope of the device, assigning single keys to those functions was inappropriate since, in such modes as Keyboard, each key would have been assigned some keyboard command. Hence, a combination of 2 keys was needed for each command (3 and more would be less user-friendly).

The decision was made to assign the combination **key8 + key9** to the **quit** function since these keys, due to the inconvenience of pressing them simultaneously, were unlikely to be assigned any other function in the modes of operation. As such, in the system, this combination is used exclusively for the quitting purpose.

The combination **key1 + key5** was assigned the **show status** command. However, due to the location of the keys, they are likely to be used simultaneously in such modes as Keyboard, PC Gamepad and Universal Joypad. For this reason, this combination works for showing the system status only in modes of operation, in which pressing those keys simultaneously is not needed for any other purpose, namely in the Menu, Sensors Info, and Service modes.

The location of each key is illustrated in Fig. 6.3.

6.2.9. Power

The class **Power** was developed to handle the power-related functionality of the system. This includes:

- Sampling and filtering the battery's voltage as well as calculation of its charging level in percents every 500 ms.
- Sampling and filtering of the 3.3V and 5V outputs on demand.
- The battery management tool.

The Power node is being initialized and regularly updated by the **SystemHandler**. On initialization, the node configures corresponding pins of the ESP32 and performs initial sampling of the ADCs until the filters are full. On update, it checks whether sufficient time has passed since the previous sampling and if yes, it samples the battery's ADC, updates the corresponding filter and recalculates the battery's charge in percents. In case the mode of operation of the system is the **Sensors** and the sub-mode **Voltages** is selected, on system update, the node additionally samples the 3.3V and 5V sources with further filtering and calculates the voltages of all three sources.

Filtering

Filtering was implemented in the same manner as was described in section 6.2.6, but with a different filter windows size of 30. Such windows size was considered appropriate due to the high sensitivity of the ADCs on the ESP32 to noise. Admittedly, a larger window size introduced the larger lag of the filtered value, however, this is not a problem when applied to the slowly changing battery voltage.

Charge in percents

The battery percentage is implemented straightforwardly. There is an ADC measurement level that corresponds to the minimum charge of the battery, which is called the **min**, and there is an ADC measurement level that corresponds to a fully charged battery, which is called **max**. Let the **cur** be the current measurement of the ADC, then the battery's charge in percents is calculated using the following equation:

$$percents = (cur - min)/(max - min) * 100 \tag{6.1}$$

The **min** and **max** values were obtained through the time series of the ADC measurements starting from the fully charged battery until the device was off. During that process, to prevent the battery from being charged, the USB cables were not connected to the device. The samples of the time series, therefore, were transmitted via Wi-Fi to the host PC, where they were recorded. For this purpose, a simple UDP server was programmed in Java.

The major disadvantage of using the equation 6.1 is the non-linear relationship between the voltage and the state of the charge of the battery. This makes the percents change slowly when the battery is nearly full and faster as long as the battery is discharged. This issue, however, was considered of minor importance since it would not affect the performance of the device and hence, postponed in case there was enough additional time for it or omitted otherwise.

The battery management tool

The battery management tool is a utility that provides service features such as showing the status of the battery (voltage, percents, the min and max values) and setting the *min* and *max* values used in the equation 6.1 with further storage in the flash memory. The reason behind allowing to set these values is the aging of the battery [12]. As time passes and the number of charging/discharging cycles increases, the battery loses its capacity and hence, the maximum charge it can store (correspondingly, its maximum voltage) decreases. Due to this phenomenon, an earlier defined *max* value may later become invalid (too high), which means after some time the equation 6.1 would never be able to result in 100%. To avoid the latter case, the *max* can be set using the battery management tool.

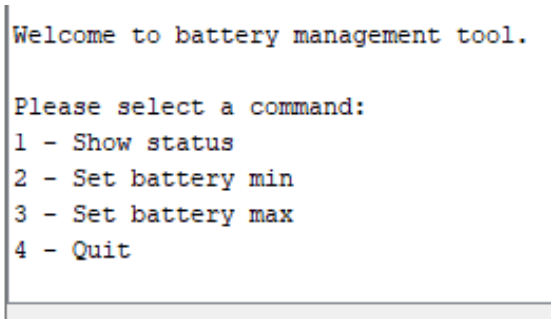

Figure 6.6.: The main menu of the battery management tool

This tool operates via the USB interface of the ESP32 using a serial terminal on a PC. The tool is initialized from the **ServiceHandler** node when the corresponding mode is selected in the **Menu** node.

6.2.10. PC Gamepad Mode

The PC Gamepad Mode is the mode of operation in which the device can emulate the behavior of a generic USB Gamepad. In this mode, it is possible to use the Wheel, e.g. to play PC games. This is achieved because of the USB emulation feature of the ATmega32u4 MCU (the Beetle board).

The GamepadHandler class

At the ESP32 side of the system, the PC Gamepad mode is handled by the GamepadHandler class. Its main function is to convert the RPY values obtained from the inertial sensor as well as the key states to corresponding PC Gamepad commands depending on the selected configuration (sub-mode) and to transmit them to the Beetle board for further processing. The commands are transmitted using the protocol described in section 6.1 and the exact frame format is illustrated in Fig. 6.7. Transmission is performed with the minimum interval of 25 ms to allow sufficient time to transmit the entire frame with the baud rate of 9600, which is set for the serial communication channel between the ESP32 and the Beetle.

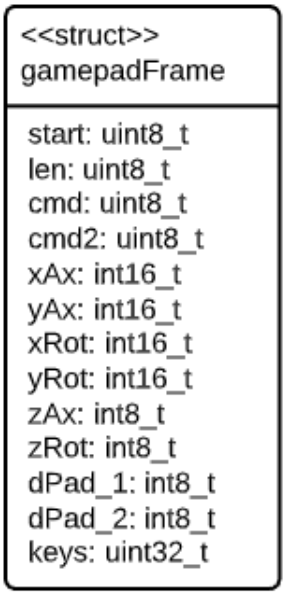

Figure 6.7.: Gamepad frame

The GamepadHandler node is initialized and regularly updated by the **SystemHandler** node in case the **Gamepad** mode of operation is selected. On update, the node checks whether sufficient time has passed since the last transmission and if yes, it performs its main function described before. When the user quits the Gamepad mode, the SystemHandler finalizes the GamepadHandler. On finalization, the latter node sends the corresponding message to the Beetle board (ATmega32u4). This is required to perform clean finalization of the USB emulation at the Beetle's side.

Sub-modes

As can be seen from Fig. 6.7, there are two sets of Gamepad values, to which the RPY values of the inertial sensor could be mapped, namely X, Y and Z axes as well as X, Y and Z rotations. For this reason, two corresponding sub-modes for the PC Gamepad mode are implemented:

- **X-Y** - maps pitch and roll of the device to X and Y axes of the PC gamepad respectively, living the Z-axis as well as X, Y and Z rotations zero.
- **RPY** - maps the RPY of the device to X, Y and Z rotations of the PC gamepad leaving the X, Y and Z-axis values zero.

In both modes the 10 keys of the Wheel are mapped to the first 10 keys of the PC gamepad out of possible 32. The commands *dPad_1* and *dPad_2*, which stand for the directional pads,

are left unused. However, they have their corresponding entries in the gamepad frame for the future implementation of those functions.

6.2.11. Universal Joypad Mode

The Universal Joypad Mode is the mode in which the Wheel can be used to control a variety of remote devices via multiple possible communication channels

The JoypadHandler class

The Universal Joypad mode is handled by the JoypadHandler node. Its main role is to take the RPY values and the encoded Yaw value introduced in section 6.2.6 as well as the key states and to transmit them to a remote device. The commands are transmitted using the protocol described in section 6.1 and the exact frame format is illustrated in Fig. 6.8. Transmission is performed with a minimum interval of 60 ms. This rate is selected as sufficiently high to make the control responsive and sufficiently low to allow successful delivery of the UDP messages.

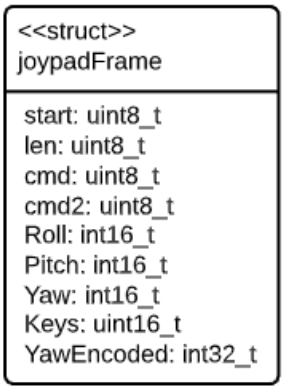

Figure 6.8.: Joypad frame

The JoypadHandler node is initialized and regularly updated by the **SystemHandler** node in case the **Joypad** mode of operation is selected. On update, the node checks whether sufficient time has passed since the last transmission and if yes, it performs its main function described before.

Sub-modes

The sub-modes of the Universal Joypad mode do not change the behavior of the mode in general. They rather define the communication channel, which is used for transmission of the Joypad frames. The following modes can be selected:

- **Wi-Fi** - transmission is performed via the UDP protocol.
- **Radio** - transmission via the 868 Mhz wireless transceiver. This mode was not implemented due to time constraints. However, it has all the infrastructure ready (e.g. menu entry) up to the point where the frame has to be transmitted.

- **USB interface** - transmission via the USB interface of the ESP32. Unlike the other sub-modes, in this sub-mode the bytes of the frame are not written, instead, the members of the frames structure are printed to the USB serial. This allows to see the actual joypad values in the serial terminal in a human-readable form.
- **Debug interface** - transmission via the debug interface of the device.

6.2.12. Keyboard Mode

The PC Keyboard Mode is the mode of operation in which the Wheel can be used to emulate the behavior of a USB Keyboard. Here, the role of actual emulations is played, again, by the Beetle board. The ESP32, in turn, notifies the Beetle about the keys that have to be pressed or released.

The KeyboardHandler class

Within the ESP32, this mode of operation is handled by the KeyboardHandler class. Its main task is to map the key states of the Wheel to keycodes and states of a PC Keyboard and to transmit them to the Beetle board for further processing. Transmission is performed in the same manner as was described in the section 6.2.10 (GamepadHandler), with the same interval of 25 ms, but with a payload containing the keyboard specific data. The used frame format is shown in Fig. 6.9. The KeyboardHandler node is initialized, updated and finalized in the same manner as was described in the section 6.2.10 (PC Gamepad mode).

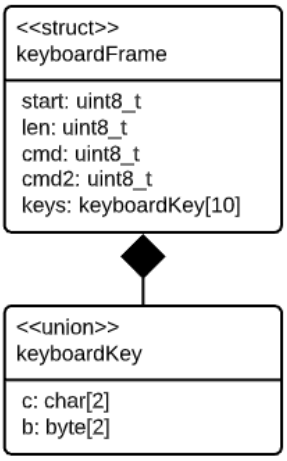

Figure 6.9.: Keyboard messages frame

Sub-modes

The sub-modes of the Keyboard mode define the mapping of the keys of the Wheel to the key codes of a PC Keyboard. There are two sub-modes (correspondingly, key - key code mappings) implemented in the system:

- **Navigation** - implements the keyboard keys, which can be used for navigation in the OS, e.g. opening/closing files, switching among the open windows, invoking the start menu, selecting items, switching the PC off, etc.

- **Gaming** - implements the keyboard keys which are commonly used in gaming.

The mapping of the keys of the Wheel to their corresponding keyboard keys for each sub-mode is presented in the table 6.1. The location and the labels of the keys on the device are illustrated in Fig. 6.3. The sub-modes were successfully tested on the Windows 10 OS.

Device key	Keyboard key, navigation mode	Keyboard key, gaming mode
0	KEY_TAB	KEY_A
1	KEY_ESC	KEY_W
2	KEY_LEFT_CTRL	KEY_D
3	KEY_LEFT_ALT	KEY_S
4	KEY_LEFT_ARROW	KEY_LEFT_ARROW
5	KEY_UP_ARROW	KEY_UP_ARROW
6	KEY_RIGHT_ARROW	KEY_RIGHT_ARROW
7	KEY_DOWN_ARROW	KEY_DOWN_ARROW
8	KEY_F4	KEY_SPACE
9	KEY_RETURN	KEY_RETURN

Table 6.1.: Mapping of the device's keys to keyboard keys

6.2.13. Mouse Mode

The Mouse Mode is the mode of operation in which the Device can be used to emulate the behavior of a USB Mouse. This functionality, as was mentioned earlier, is achieved due to the USB emulation capability of the Beetle board. The role of the ESP32 in this mode, as in the cases of the Gamepad and the Keyboard modes, is to send mouse-related commands to the Beetle.

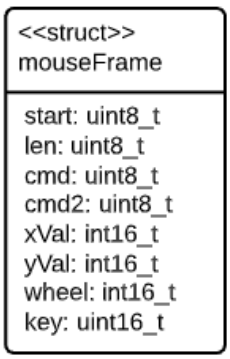

Figure 6.10.: Mouse frame

The MouseHandler class

The Mouse mode functionality is handled by the MouseHandler class. The MouseHandler converts the Roll and Pitch of the Device into the mouse pointer coordinates or velocities depending on the selected sub-mode as well as maps the on-board keys to mouse key commands. This data is then packed into mouse frames (Fig. 6.10) and transmitted to the Beetle board in the same manner as it

was described for the KeyboardHandler and the GamepadHandler earlier. However, the transmission interval is shorter than that of the Keyboard and Gamepad modes, and is set to 15 ms. This is possible due to a smaller length of the transmitted frames.

Initialization, update, and finalization of the MouseHandler node are similar to those of the KeyboardHandler and the GamepadHandler and were described in section 6.2.10.

Sub-modes

There are two sub-modes of the Mouse mode implemented:

- **Absolute** - the *Pitch* of the device is mapped to the *X* coordniate of the mouse pointer, the *Roll* - to the *Y* coordinate.
- **Relative** - the *Pitch* and the *Roll* of the device are translated to the velocities of the mouse pointer along the *X* and *Y* axes respectively.

In both sub-modes, the mapping of the keys of the device to mouse functions is identical and is provided in the table 6.2.

Device key	Mouse function
1	scroll 1 position up
3	scroll 1 position down
4	left key press
5	left key double click
6	right key press
7	middle key press

Table 6.2.: Mapping of the device's keys to mouse functions

Usability

One of the main issues in the development of the Mouse mode was to ensure the stabilization of the mouse pointer at the desired location on the screen.

For the Relative mode, this problem was solved simply by thresholding the values of Roll and Pitch. For example, if the absolute value of the Roll is less than $1.5°$, the corresponding velocity along the Y-axis is 0, which means the pointer does not move vertically with respect to its current location

For the Absolute mode, the problem of instability was to a high extent reduced by filtering of the inertial values as was described in section 6.2.6. The remaining instability did not cause any usability problems except for the difficulty to perform a mouse double click, because, for that, the pointer has to fully stop. The problem was solved by implementing the mouse double click command, which is issued by a single click of the corresponding key on the Wheel.

6.2.14. Service Mode

The purpose of the Service mode, as the name suggests, is to perform service operation such as configuration, calibration of the Device, etc. This mode is handled by the Service-Handler class. Currently, the ServiceHandler provides three functions (sub-modes):

- **Calibration** - initiates and coordinates the process of calibration of the inertial sensor. The process is conducted interactively: before performing each calibration step, the user is asked to place the device in a certain way and to initiate the step by pressing a key. Corresponding messages are displayed on the OLED screen and completion is indicated by the LEDs. This sub-mode is the only one, which is fully coordinated by the ServiceHandler node.
- **Wi-Fi management** - calls the corresponding routine of the WiFiHelper node, which in turn initializes the **Wi-Fi management tool**. The tool is used to perform a variety of Wi-Fi related operations and configurations vie the USB serial interface of the ESP32. Further details about the tool are provided in section 6.2.16.
- **Battery management** - initializes the **Battery management tool**, which was described earlier, in the section 6.2.9.

6.2.15. USB Serial Helper

The USB Serial Helper functionality is nothing that the user can make use of. It is rather a set of utility functions provided by the USBSerialHelper class, that are meant to ease further development and maintenance of the system. The class offers functions that come in handy in the implementation of serial communication based interactive tools, such as data pickers, dialogs of the types “YesNo”, “YesNoCancel”, “Confirm”, etc. The main advantage of these functions is the handling of invalid input from the user and ensuring that either the system receives a valid input or the user cancels the operation. Besides, the functions offer standardized messages for the dialogs yet allowing to set a custom message. The exact set of functions implemented in the class is provided in Fig. 6.11.

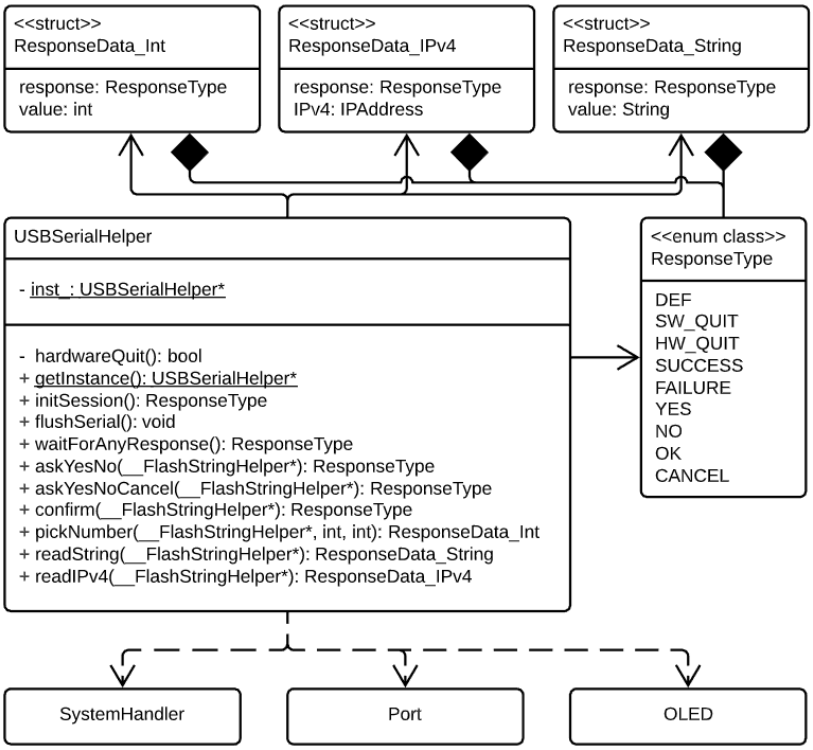

Figure 6.11.: ESP32 class diagram: USB Serial Helper perspective

Here, the function **initSession()** is worth particular attention. It is used to initialize a serial communication session by telling that the user has to connect the Device to a PC via the USB interface, open a serial terminal and send any message to the Device. This is done by showing the corresponding message on the OLED screen. Only after the user has performed those steps, the function returns to the calling routine. This way, it is ensured that all further messages sent via the USB serial interface will appear in the Serial terminal, hence, will not be lost, because the terminal is already open and connected.

The *Battery management tool* and the *Wi-Fi management tool* mentioned in the section 6.2.14 were implemented using the USBSerialHelper class. A sample dialog from the *Wi-Fi management tool* is illustrated by the Fig. 6.11. The example demonstrates handling of wrong inputs.

```
Selected command: Set TCP Client IP and Port.

Please enter an IP V4 address in the format X.X.X.X using decimal numbers.
Entered value: abc

Please enter a valid IP V4 address in the format X.X.X.X
Entered value: 123

Please enter a valid IP V4 address in the format X.X.X.X
Entered value: 256.256.256.256

Please enter a valid IP V4 address in the format X.X.X.X
Entered value: 192.168.0.104


Please enter a port number between 1024 and 65535
Entered value: 1023
Please enter a valid integer number between 1024 and 65535.
Entered value: 65536
Please enter a valid integer number between 1024 and 65535.
Entered value: 1024

Saved new TCP client IP:Port: 192.168.0.104:1024
```

Figure 6.12.: USB Serial Helper sample use case

6.2.16. Wi-Fi

In the system of the Wheel, two classes share functionality related to communication over Wi-Fi, namely the **WiFiConnectHandler** and the **WiFiHelper**. Details about the purpose and functionality of each class are provided further.

The WiFiHelper node

The WiFiHelper node is responsible for handling the Wi-Fi related functionality. It provides functions to connect/disconnect to/from a selected Wi-Fi AP, to start/stop TCP/IP and UDP servers, to send messages as a TCP/IP or UDP client and others.

Furthermore, the WiFiHelper node implements the **Wi-Fi management tool**, which was shortly introduced in section 6.2.14. The tool operates via the USB serial interface using a serial terminal on a PC. It provides a wide range of commands to configure, manipulate and test Wi-Fi-related functionality of the Device such as discovering and saving APs in the flash memory, testing connection, setting IP addresses and ports, etc. The full list of commands is illustrated in Fig. 6.13, where the main menu of the tool can be seen.

```
Welcome to Wi-Fi management tool.

Please select a command:
1 - Show status
2 - Scan for Access Points
3 - List saved Access Points
4 - Add or replace Access Point
5 - Delete Access Point
6 - Connect to a saved Access Point
7 - Disconnect from Access Point
8 - Start UDP Server
9 - Stop UDP Server
10 - Set UDP Server Port
11 - Start UDP Client
12 - Stop UDP Client
13 - Set UDP Client IP and Port
14 - Send UDP message
15 - Start TCP Server
16 - Set TCP Server Port
17 - Set TCP Client IP and Port
18 - Send TCP message
19 - Quit
```

Figure 6.13.: Wi-Fi tool commands

The node is being initialized on the system start and updated at each system update by the SystemHandler node. On initialization, the node performs an attempt to connect to the first available AP from those stored in the flash memory. A successful connection is followed by bringing up the TCP/IP and UDP servers. From that moment on and as long as the servers are running, the device can receive messages via the Wi-Fi network. On update, it checks the connection status as well as incoming TCP/IP and UDP messages. If any message is available, the node shows it on the OLED screen and locks the screen. Unlocking the screen requires the user to acknowledge the message by issuing the **quit** command as was described in section 6.2.7.

The WiFiConnectHandler class

There is a special menu option implemented that is called **CONNECT**, was introduced in section 6.2.4 and serves as a directory for connectivity-related functions. Currently, there is only one entry in that directory, which is called **WIFI**. When the entry is selected, the system initializes the **WiFiConnectHandler** node, which in turn coordinates the interactive process of connecting to a Wi-Fi AP. In this process, the user manually navigates through the list of the Wi-Fi APs stored in the flash memory and selects an AP to connect to. Navigation and selection is performed using the same keys as for the menu navigation. The APs, as well as connection success or failure messages, are displayed on the OLED screen. If the connection succeeds, the system brings up the TCP/IP and UDP servers. The WiFiConnectHandler node uses the functions provided by the **WiFiHelper** class for connection and bringing the servers up.

6.2.17. Sensors Mode

The simplest mode of operation of the Device is the Sensors mode, which is handled by the **SensorInfoHandler** node. The purpose of this mode is to display sensor values on the OLED screen. There are two sub-modes implemented: the **RPY** and the **Voltages**. The former sub-mode displays live values of the RPY, whereas the latter one displays voltages of the 3.3V, 5V outputs as well as the battery.

6.2.18. Debug Output

During operation, the system on the ESP32 produces debug output. This contains messages about the functions being called, changes in the modes of operation, some important operations being performed and their results being produced, the data received and transmitted via the communication interfaces, etc. The USB interface of the ESP32 is selected as the debug output. This is done due to the simplicity of its connection to a PC and its baud rate set to 115200. This, however, can be easily changed to the dedicated 14-pin debug output of the Device by changing the corresponding definition in the code.

6.3. Atmega32u4

The role of the ATmega32u4-based MCU board (the Beetle) was limited to acting as the USB interface from the ESP32 to a PC for emulation of the USB-based functionality. The behavior of the software on the board is implemented in the following manner:

There are 4 modes of operation: the Keyboard, the Gamepad, the Absolute Mouse and the Relative Mouse. Each mode has its dedicated handler class. The main routine reads the messages arriving from the ESP32 via the serial interface and validates them. When a valid Keyboard, Absolute Mouse, Relative Mouse or Gamepad message is received, the corresponding mode is initialized (if was not initialized before) and its dedicated handler is notified about the new data to be processed. The handler, in turn, processes the data by converting the arrived commands (e.g. mouse pointer coordinates) to the corresponding format to be forwarded to the PC. The “conversion” is performed using dedicated libraries. The Keyboard and Relative Mouse modes are implemented using the standard keyboard and mouse libraries provided by Arduino, whereas for implementation of the Gamepad and Absolute Mouse modes, the “Arduino HID Project” library was required.

If the newly arrived message corresponds to a mode that is different from the previous one, the mode is changed by finalizing the previous mode and initializing the new one. Here, finalization of a mode means, e.g. releasing all the keys of the keyboard and sending a clean report to the PC. There is a special type of message - the default message. This message has its dedicated command and does not correspond to any mode of operation. It is used by the ESP32 to notify the Beetle when the user quits any USB emulation based mode. When the Beetle receives a default message, it finalized the current mode of operation.

The Beetle is programmed to act only as a translator of the commands arriving from the ESP32 to their appropriate USB messages and to forward them to the PC. This means the Beetle is not aware of, e.g. which Keyboard mode configuration (sub-mode) is selected (which mapping of the onboard keys to the keyboard keycodes is used). This, in turn, implies that if a new sub-mode of e.g. the Keyboard or the Gamepad mode needs to be implemented, the Beetle does not need any changes in the code to be able to handle it. Hence, the implementation of new configurations of those modes needs to be performed on the ESP32’s side only, which improves the overall maintainability of the system on the Wheel. Fig. 6.14 describes the overall structure of the system implemented on the Beetle board.

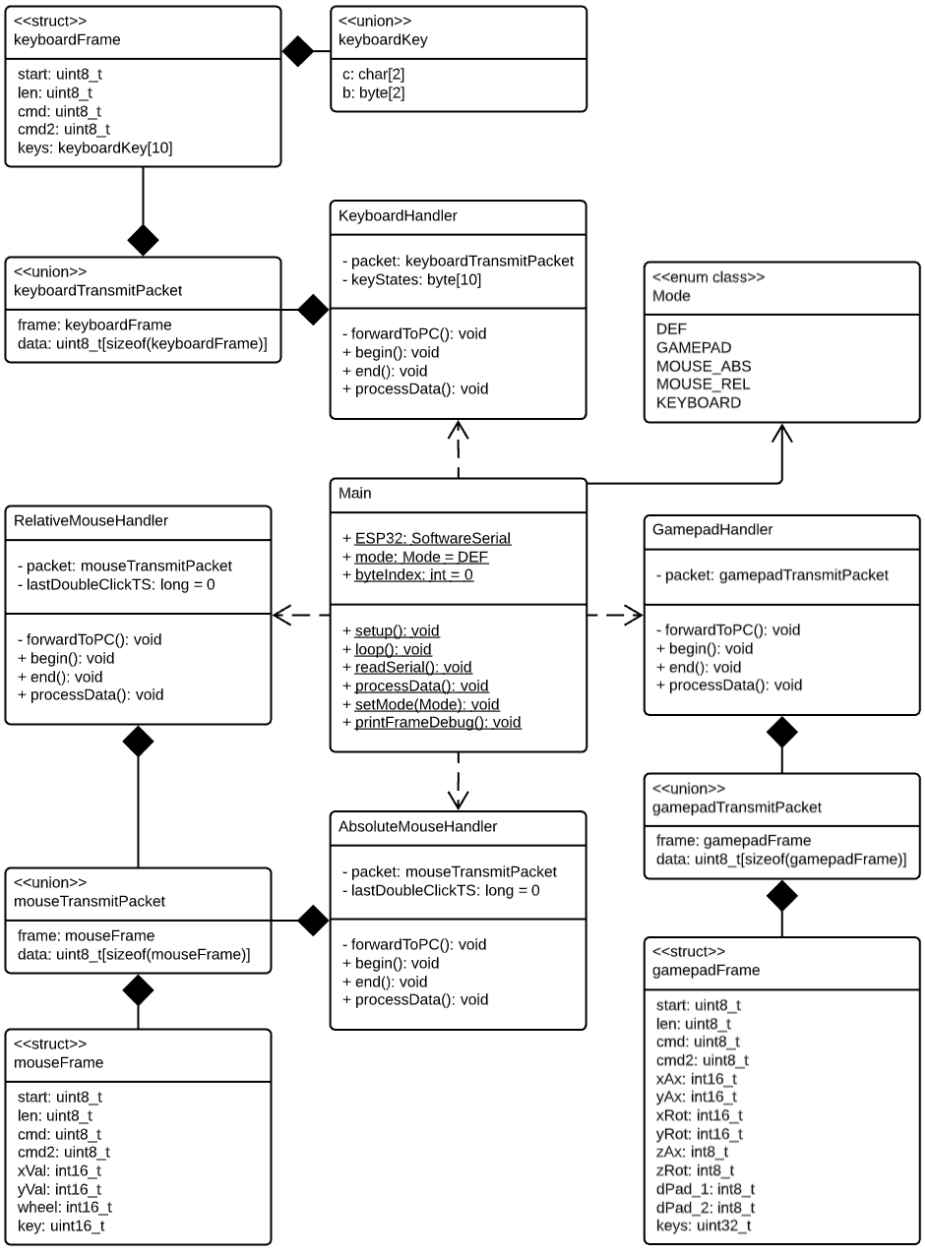

Figure 6.14.: The class diagram of the system on the Beetle board (ATmega32u4)

7. Testing

This chapter describes the process of testing of the Device. First, the implemented unit tests will be introduced, followed by the description of the system and some of its timing characteristics. The chapter will be finalized describing a demo that was developed to test and showcase the Wheel's capabilities of intuitive remote control.

7.1. Unit Test

The unit tests within this project served the purpose of testing the basic functionality of each peripheral module as well as the state of electrical connections (wiring). There were 2 unit tests developed: the **primary** and the **secondary**. The primary unit test covers a larger amount of peripherals and is intended for being run on the ESP32 because the MCU is connected to all hardware modules. The secondary test, correspondingly, covers a limited number of hardware modules, namely those which are interfaced with the Beetle (ATmega32u4), and hence is designed for that MCU.

Source code

The source code of the unit tests is separated from the main system. As such, each unit test has a dedicated project.

Although the primary test is performed by the ESP32, the Beetle is also involved as one of the tested modules. For this reason, each of the two MCUs has its dedicated program, that has to be loaded to perform the primary test. However, without the Beetle, the primary test still can be partially performed.

For the secondary unit test, only the Beetle needs to be programmed since the ESP32 does not take part in this test at all.

Running and the output

The unit tests can be performed by following the next steps:

1. The programs have to be loaded onto the MCUs depending on which unit test is to be run.
2. The Device has to be connected to a PC via the USB interface of either the ESP32 or the Beetle depending on which unit test is to be run.
3. A serial terminal has to be opened on the PC and any message needs to be sent to the Device to show the main menu of the unit test in the terminal.
4. The test can be started by selecting a corresponding command from the menu.
5. During the test, some commands may appear in the terminal, which need to be performed by the user, such as pressing keys, moving the device, looking at it, etc.
6. After the selected unit test is completed, the main menu will appear again allowing us to select another command. This behavior is repeated until the device is off.

```
Select unit test:
1 - Power
2 - OLED
3 - Software Serial
4 - USB Mouse
5 - RFM69
6 - MPU9250
7 - BMA180
8 - Port exapander
9 - I2C scan
10 - I2C ESP32 <-> 32u4
11 - Scan Wi-Fi
12 - Buzzer
13 - All unit tests
```

(a) Primary test

```
Select unit test:
1 - Power
2 - OLED
3 - MPU9250
4 - BMA180
5 - Port exapander
6 - I2C scan
7 - Mouse
8 - All unit tests
```

(b) Secondary test

Figure 7.1.: Unit test main menus

The output of the primary unit test is listed in the appendix A.1. The output of the secondary test is a subset of the output of the primary one containing entries corresponding to its commands.

7.2. System Test

During the system test, functionality of the device was tested using various methods, software tools as well as by practical application of each mode in its intended context.

Universal Joypad mode

Since the purpose of the Universal Joypad mode is to stream joypad frames (inertial values and keys) via various transmission channels, testing of the mode was performed correspondingly. The **Wi-Fi** sub-mode was tested using a UDP server programmed in Java and running on the PC, where the Joypad frames were sent. The Fig. 7.2 illustrates the incoming messages at the server's side. The **Debug** and **USB** sub-modes were tested in a similar manner, but using a serial terminal on the PC.

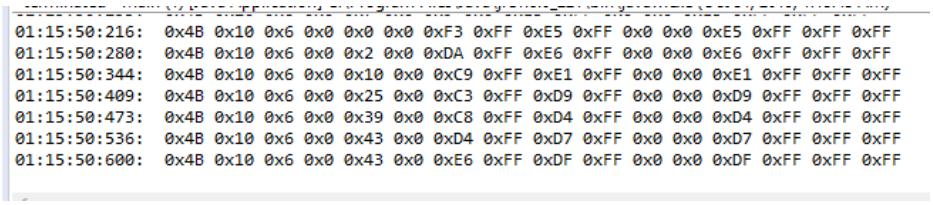

Figure 7.2.: Universal Joypad mode: sub-mode Wi-Fi, frames

Mouse mode

The Mouse mode was tested without any specific software by direct application of the device in the role of a mouse. The **Absolute** sub-mode displayed higher usability compared to the **Relative** due to higher intuitiveness of navigation as well as rapidness of positioning of the pointer at the desired point on the screen. Button clicks in both sub-modes behaved as intended. The only notable problem was that the pointer does not move very smoothly due to the interval of transmission of mouse frames.

Keyboard mode

The keyboard mode was tested similarly to the Mouse mode - by direct application. All keys were functioning as intended and delays between key presses on the device and corresponding actions at the PC's side were barely noticeable.

PC Gamepad mode

The PC Gamepad mode of operation was tested using the **calibration tool** of Windows 10 as well as through practical application by playing browser-based games. The gaming experience met the overall expectation.

The Fig. 7.3 illustrates the process of calibration of the Device. Here, the sub-mode **X-Y** is selected, which was described in section 6.2.10. As can be seen from the picture, the X and the Y-axis values are successfully recognized by the tool.

The Fig. 7.4 demonstrates operation of the Device in the **RPY** sub-mode of the PC Gamepad mode. The horizontal bars represent the X, Y and Z rotations of the Device. Besides, the are 9 button presses successfully recognized by the calibration tool. The 10^{th} button is not pressed intentionally, because it would lead to quitting the PC Gamepad mode (the combination key8 + key9 of the Device, which correspond to the buttons 9 and 10 of the PC Gamepad, is reserved for quitting).

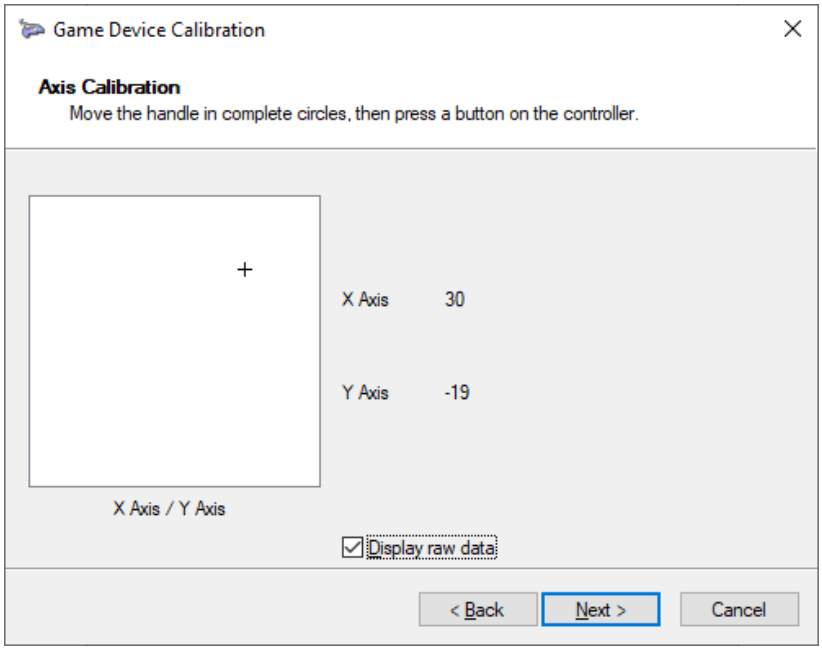

Figure 7.3.: PC Gamepad mode: sub-mode X-Y, axis calibration

Other functions

All the remaining modes of operation as well as features of the device were tested and appeared to be fully functional.

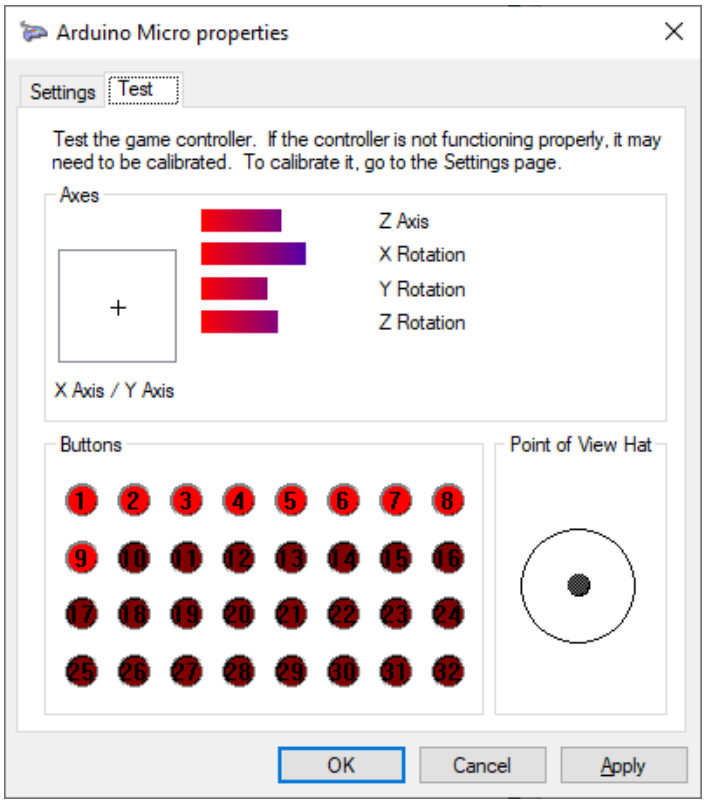

Figure 7.4.: PC Gamepad mode: sub-mode RPY, testing

7.3. Timing

To test compliance of the system with the timing-related non-functional requirements specified in the chapter 3, timing characteristics of the system were measured. This includes measurement of the rate of update of the system in time-critical modes of operation, as well as rates of delivery of the commands via the transmission channels.

System update rate

The system update rate in the modes of operation *PC Gamepad*, *Mouse*, *Keyboard* and *Universal Joypad* is stable at **100 Hz**. In the mode *Sensors*, the rate drops to **50 Hz** due to regular update of the OLED screen involved in the process. The values were obtained by printing the time intervals of calling the system update routine in the debug. The rate of update of the system defines the sampling rate of the inertial sensor as well as the rate of detecting key states because those operations are performed on each system update.

Data transmission rate

For the modes: *PC Gamepad*, *Mouse* and *Keyboard*, the values were obtained by printing the time intervals of receiving corresponding commands at the ATmega32u4 side. Those intervals match the intervals of further transmission of the commands to the PC. The values for the *Wi-Fi* sub-mode of the *Universal Joypad* mode were obtained by measuring the intervals of reception of the commands at the UDP server on the PC. For the *USB* and *Debug* sub-mode of the *Universal Joypad* mode, the values were obtained by measuring transmission intervals of corresponding frames at the ESP32. The Fig. 7.5 presents the results of the measurements. The X-axes show the numbers of the samples, which correspond to the numbers of the transmitted data frames. The Y-axes show for each data frame the elapsed time from the previous data frame.

From the measurements, it can be concluded that the timing behavior of the system to a high extent meets the requirements. However, there are some overshoots in the intervals of reception of frames at the Beetle's side in the USB-emulation based modes (PC Gamepad, Keyboard, Mouse). This phenomenon needs further investigation.

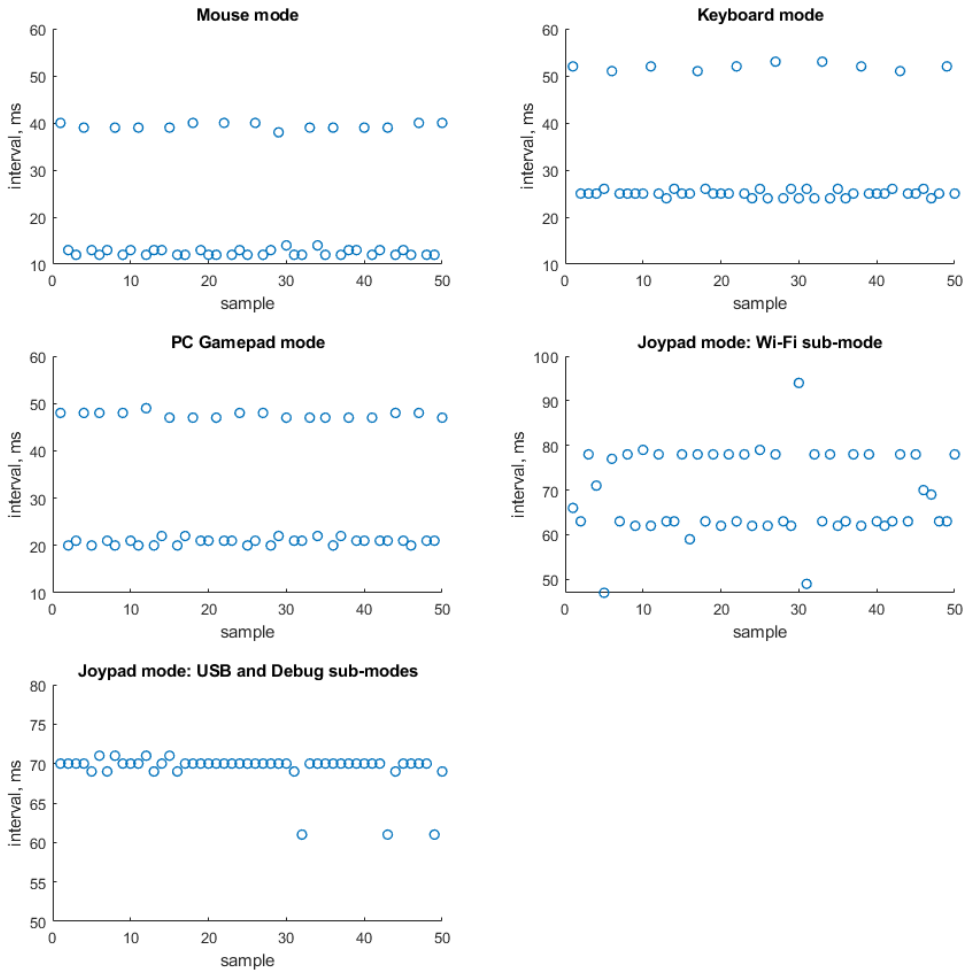

Figure 7.5.: Data transmission intervals

7.4. Universal Joypad Demo

While testing of the USB emulation based functionality in action was straightforward and required only a PC, testing of the *Universal Joypad* mode required additional resources, namely some remote devices that could be controlled by the Wheel in that mode. This would allow in addition to the testing of the general functionality of the mode (e.g. timely transmission of correct Joypad data frames) also to asses usability of the Wheel in this role, e.g. how responsive and intuitive is the process of controlling a remote device. Besides, such a test would serve as a good demo showcase of the final product's capabilities.

To test the Wheel in the Universal Joypad mode, an RC drone and an RC toy excavator were programmed to act as the remote devices to be controlled. Each device has onboard an MCU which is interfaced to its actuators as well as an external (detachable) **HC-06** Bluetooth module for wireless communication.

Figure 7.6.: The RC devices for the demo of the Universal Joypad mode

The general work to be performed included:

- Establishing Bluetooth communication between the Wheel and the remote devices.
- Transmission of Joypad frames from the Wheel and correct sampling of them by the remote device.
- Applying the received Joypad data to control the remote devices in a way that would make the overall experience intuitive and user-friendly.

Bluetooth communication

As was mentioned in section 4.2 (Software Design), the BT-related functionality of the Device was not implemented due to the extremely large footprint of corresponding libraries. However, since one of the sub-modes of the Universal Joypad mode of the device is the **Debug**,

any external communication module having an on-board UART interface can be connected to the debug interface of the Wheel to receive the Joypad frames and transmit them further to the remote devices. Thus, the problem of BT communication was solved by using an external BT module **HC-05**. The module provides a serial communication interface, and hence, can be connected to the Wheel's debug interface.

Due to the time constraints, no user interface was developed that would provide functionality such as discovering BT devices and connecting to them. To keep the development process simple, the **HC-05** module was configured to automatically connect to a predefined remote BT module - one of those provided with the drone and the excavator. Thus, in the context of this test, only a single HC-06 BT module could be used by attaching it either to the drone or to the excavator depending on which device is to be controlled.

Connection status

The drone and the excavator have each an on-board RGB LED. The LEDs are programmed to act as the indicators of connection status. The behavior is as follows: when the remote device receives Joypad frames, the LED is blinking with a blue color. After 500 ms of silence, the LED turns red indicating connection loss. In this context, silence means that no valid Joypad frames are received, e.g. no frames are received at all or the received frames are incorrect. Both the drone and the excavator are programmed to stop their motors whenever silence is detected. The excavator stops immediately, whereas the drone holds the current state for 5 seconds.

The Drone

Due to lack of time, a decision was made to omit the development of sophisticated features such as absolute positioning and orientation in the 3D space. Instead, the relative approach was chosen. Thus, the joypad frames arriving at the drone are converted to raw Roll, Pitch, Yaw and Thrust commands of its onboard flight controller and not to position and orientation.

Roll and Pitch The Roll and the Pich commands of the drone are linearly mapped to those from the Joypad frame (Fig. 6.8). However, the values that are sent to the flight controller by the drone's MCU are limited such that the drone does not make highly dynamic maneuvers. This way, it is easier to control the drone without special piloting skills.

Yaw In the case of the Yaw, since there is no tracking of absolute orientation on the drone, mapping of the Yaw from the Joypad frame to the Yaw command of the flight controller would lead to the drone rotating around its Z-axis whenever the Wheel's Yaw is non-zero. Thus, the Yaw command of the drone was mapped to the keys of the Wheel. As such, when a key pressed, the Yaw commands with a predefined value will be issued to the flight controller and the drone will rotate with a fixed rate until the key is released.

Thrust Since the Roll, Pitch and Yaw from the Joypad frame cannot be used for the Thrust commands and there is no additional suitable variable-state input method (e.g. a potentiometer-joystick) on the Wheel, again, the keys of the device were assigned this role. As such, pressing one key will increase the Thrust by a predefined step, whereas the other will decrease it. Relatively smooth behavior can be achieved by pressing and holding the keys.

Other The Drone has an additional onboard LED that can be toggled by pressing a key.

The Excavator

Forward motors The forward motors are controlled by the Pitch and Roll from the Joypad frame. The Pitch defines the target forward velocity of the left and the right motor. The value of the Pitch is biased such that the 0-velocity corresponds to the angle of $-55°$. This is done to improve the intuitiveness of manipulation. The Roll is used to proportionally reduce the target velocity of either the left or the right motor depending on the sign of the Roll's value. Besides, there are commands to perform a rotation, e.g. left motor forward, right motor reverse, or vice-versa. These commands are issued by pressing corresponding keys. Finally, to initiate movement of the forward motors, the key7 has to be pressed and held. This feature prevents the excavator from unintended movement caused by tilting the Wheel.

The tower and the claw motors The tower and the claw motors are controlled using the keys. When the key is pressed, the corresponding motor gets a predefined fixed target velocity.

Smooth movement To achieve smooth maneuvering, the target velocities mentioned so far are not applied to the motors immediately. Instead, the actual velocities of the motors approach their target velocities by predefined value steps (acceleration) at the rate of receiving the Joypad frames. There are individual acceleration and deceleration values as well as

minimum and maximum velocities defined in the program for each motor, which makes the behavior highly tunable.

Other The Excavator has taillights that can be toggled by a keypress.

The table 7.1 summarizes the mapping of the inertial values as well as keys of the Wheel to the functions of the Drone and the Excavator.

Device function	Drone function	Excavator function
key0	Yaw left	Tower left
key1	Toggle LED	Claw up
key2	Yaw right	Tower right
key3		Claw down
key4		Rotate left
key5	Increase thrust	Toggle tail lights
key6		Rotate right
key7	Decrease thrust	Move
key8		
key9		
Roll	Roll	Turn left/right
Pitch	Pitch	Forward/reverse
Yaw		

Table 7.1.: Mapping of functions for the demo

8. Summary

This chapter briefly summarizes the state of the project, what has been achieved and remains unresolved as well as the problems faced along the progression of this project, which could be addressed in the future. Additionally, there will be further ideas introduced on how to improve the device.

8.1. Project Status

The final product has approached its intended state very closely. The developed hardware to high extent complies with the specifications with one exception that the ATmega32u4-based MCU board (the Beetle) is not connected to all the required peripherals, which was a necessary measure. All functional requirements were met except for implementation of the Bluetooth and 868 Mhz wireless transceiver related functionality as well as the possibility to assign the Beetle the master role. The performance-related non-functional requirements are mainly met with minor timing issues that need further investigation. The requirement to implement the management of Wi-Fi APs and configurations using the UI of the Device (keys, OLED display), due to its complexity and strict time constraints, was implemented alternatively - using serial terminal.

Overall, each major mode of operation is implemented and functions well, which was proven by testing.

8.2. Problems

Program memory One of the main problems faced in this project was the lack of program memory on both MCUs. This problem has not turned out to be critical in the case of the Beetle (ATmega32u4), because the functionality of the system can mostly rely on the ESP32 assigning only limited tasks to the Beetle. Lack of memory on the ESP32, on the other hand, made a strong impact on the overall functionality of the system - the Bluetooth Low Energy (BLE) functionality, which was meant to considerably increase the scope of application of

the device, was not implemented. The problem can be potentially solved by avoiding using libraries and accessing the BLE modules functionality at a lower level.

Interrupts Another major problem was caused by the interrupt-based approach on the ESP32. If the Interrupt Service Routine (ISR) contains a certain unsupported set of instructions, this leads to a crash of the program and the MCU reboots. An example of such instructions is performing floating-point operations within an ISR. Such operations are performed using the Floating-point unit (FPU) of the ESP32's processor, whereas the FPU functionality is not supported within the ISRs. This problem is specific to the ESP32 boards.

8.3. Further Work Outlook

Further work could potentially be focused on solving the problems mentioned earlier as well as further development of the device to extend the scope of its application and improve overall user experience. Here are some ideas about possible improvements:

- An Infrared transceiver would allow controlling a wide of home appliances such as TV, stereo system, air conditioning, etc. The device could be programmed to sample and duplicate the frames of existing remote controls as well as implement the existing protocols of popular manufacturers.
- Two potentiometer joysticks would be very user-friendly when controlling RC toys, scrolling through items with variable speed (e.g. menu items or alphanumeric characters) or even as an alternative mouse manipulator.
- The Wheel could be integrated with smartphones to benefit from the flexibility of the mobile Apps. The Apps could be used, e.g. to test or configure the device.
- Additional RC vehicles with different types of manipulation and degrees of freedom could be implemented such as boats, cars, robots, etc. to find out the best ways to achieve intuitive control of each type.
- A vibration motor would add more responsiveness to the device, e.g. when a message is received or the remote device is connected/disconnected, etc.
- The 868 MHz radio transceiver could be complemented by a 2.4 GHz transceiver to extend the range of potential remote devices. The 2.4GHz band is commonly used in the RC toy industry.

- Common Bluetooth protocols could be implemented, such as the Advanced Audio Distribution Protocol (A2DP) and the Audio/Video Remote Control Protocol. This would allow us to control a wide range of multimedia devices that are already offered in the market.
- Implementing a custom PCB and design of a new body.

9. Conclusion

An extensive work has been performed within this thesis. This included the stages of product development starting from the conception and design, through the implementation of the hardware and software, to the testing and assessment of the performance of the device in its intended scope of application. The final product has got capabilities to use the input from a user, such as keypresses and inertia, to manipulate remote devices over various wired and wireless communication channels. The device has to a high extent met the expectations applied to the amount and quality of its functionality. As such, it can be comfortably used to control a range of appliance which are meant to be controlled by differing methods. For example, an aerial vehicle can be equally well manipulated as a tracked vehicle by the same device, which can be used as a PC mouse, keyboard and gamepad as well as act as an input/output node in a Smart Home system. These results have justified the motivation behind this work and proved the overall concept to have good potential for further development and production.

Bibliography

[1] A. Deshmukh, *Microcontrollers: Theory and Applications*, ser. Computer engineering series. Tata McGraw-Hill, 2005, ISBN: 9780070585959. [Online]. Available: https://books.google.de/books?id=5PDx2Q9Ea%5C_YC.

[2] J. Catsoulis, *Designing Embedded Hardware*, ser. O'Reilly Series. O'Reilly, 2002, ISBN: 9780596003623. [Online]. Available: https://books.google.de/books?id=KoR1FxOOmO8C.

[3] S. Smith, *Digital Signal Processing: A Practical Guide for Engineers and Scientists*, ser. Demystifying technology series : by engineers, for engineers. Elsevier Science, 2003, ISBN: 9780750674447. [Online]. Available: https://books.google.de/books?id=PCrcintuzAgC.

[4] *I2c info – i2c bus, interface and protocol*, https://i2c.info/, Accessed: 2019-06-25.

[5] *Odroid universal motion joypad documentation*, https://wiki.odroid.com/accessory/add-on_boards/universal_motion_joypad/universal_motion_joypad, Accessed: 2019-06-20.

[6] *Esp32-wroom-32 datasheet*, https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf, Accessed: 2019-07-20.

[7] *Atmega16u4/atmega32u4 8-bit microcontroller with 16/32k bytes of isp flash and usb controller datasheet*, http://ww1.microchip.com/downloads/en/devicedoc/atmel-7766-8-bit-avr-atmega16u4-32u4_datasheet.pdf, Accessed: 2019-07-20.

[8] B. Stroustrup, *The C++ Programming Language: The C++ Programm Lang_p4*. Pearson Education, 2013, ISBN: 9780133522853. [Online]. Available: https://books.google.de/books?id=PSUNAAAAQBAJ.

[9] S. Chapman, *Essentials of MATLAB Programming*, ser. Essentials of MATLAB programming v. 10. Cengage Learning, 2008, ISBN: 9780495295686. [Online]. Available: https://books.google.de/books?id=63T7XpfMH38C.

[10] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, 2012, ISBN: 9781449345044. [Online]. Available: https://books.google.de/books?id=qIucp61eqAwC.

[11] *Esp32 documentation*, https://docs.espressif.com/projects/esp-idf/en/latest/get-started/, Accessed: 2019-07-23.

[12] B. Kai, *Modern Battery Engineering: A Comprehensive Introduction*. World Scientific Publishing Company, 2019, ISBN: 9789813272170. [Online]. Available: https://books.google.de/books?id=heaWDwAAQBAJ.

[13] R. Pecinovsky, *OOP - Learn Object Oriented Thinking & Programming*, ser. Academic series. Tomas Bruckner, 2013, ISBN: 9788090466180. [Online]. Available: https://books.google.de/books?id=xb-sAQAAQBAJ.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Pearson Education, 1994, ISBN: 9780321700698. [Online]. Available: https://books.google.de/books?id=6oHuKQe3TjQC.

[15] K. Townsend, C. Cufi, and R. Davidson, *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*, ser. EBSCOhost ebooks online. O'Reilly Media, 2014, ISBN: 9781491900598. [Online]. Available: https://books.google.de/books?id=24N7AwAAQBAJ.

[16] *A brief introduction to the serial peripheral interface (spi)*, https://www.arduino.cc/en/reference/SPI, Accessed: 2019-06-25.

[17] L. Frenzel, *Handbook of Serial Communications Interfaces: A Comprehensive Compendium of Serial Digital Input/Output (I/O) Standards*. Elsevier Science, 2015, ISBN: 9780128006719. [Online]. Available: https://books.google.de/books?id=wnGDBAAAQBAJ.

[18] *Basics of uart communication*, http://www.circuitbasics.com/basics-uart-communication/, Accessed: 2019-06-25.

[19] E. Harold, *Java Network Programming*. O'Reilly Media, 2004, ISBN: 9780596552589. [Online]. Available: https://books.google.de/books?id=NyxObrhTv5oC.

A. Appendix

A.1. Unit Test output (ESP32)

```
   Select unit test:
1 - Power
2 - OLED
3 - Software Serial
4 - USB Mouse
5 - RFM69
6 - MPU9250
7 - BMA180
8 - Port exapander
9 - I2C scan
10 - I2C ESP32 <-> 32u4
11 - Scan Wi-Fi
12 - Buzzer
13 - All unit tests

Selected command: 13

Scanning I2C...
I2C device found at address: 0x20
I2C device found at address: 0x38
I2C device found at address: 0x3C
I2C device found at address: 0x64
I2C device found at address: 0x68
done


START unit test: Power
Collecting 30 samples...done
u3V3: 3.33V  u5V: 4.92V  uBat: 4.14V  (99%)
END unit test: Power

START unit test: OLED
Initializing display...done
Drawing lines (please look at the display)...done
END unit test: OLED
```

```
START unit test: Software Serial
Initializing serial communication...done
Sending a command to Atmega32u4: 0xC8
Received response from Atmega32u4: 0xC8 - OK
END unit test: Software Serial

START unit test: ESP32 <-> 32u4 I2C communication
Initializing I2C...done
I2C address: 0x64
Register: 0x01
Sending the value: 0xC8
Sent successfully
Requesting the value: 0xC8
Received value: 0xC8 - OK
END unit test: ESP32 <-> 32u4 I2C communication

START unit test: USB Mouse
Initializing serial communication...done
Sending a "mouse" command to Atmega32u4
Received response from Atmega32u4
After 2 seconds the pointer of the muse should start moving (Atmega32u4
    must be connected to the PC via a USB cable). Please do not touch the
     mouse.
END unit test: USB Mouse

START unit test: RFM69
Initializing...done
Reading all registers:

reg 0x01: 0x04
...
reg 0x7F: 0x00
END unit test: RFM69

START unit test: MPU9250
Initializing MPU9250...MPU9250 WHO AM I = 71
MPU9250 is online...
AK8963  WHO AM I = 48
Calibration values:
X-Axis sensitivity adjustment value 1.17
Y-Axis sensitivity adjustment value 1.18
Z-Axis sensitivity adjustment value 1.13
X-Axis sensitivity offset value 0.00
Y-Axis sensitivity offset value 0.00
Z-Axis sensitivity offset value 0.00
done
Sampling the sensor (please move the device):
```

```
aX: 0.01 g
aY: -0.61 g
aZ: 0.80 g
gX: 3.66 deg/s
gY: 0.55 deg/s
gZ: -5.68 deg/s
mX: 332.74 mG
mY: 293.62 mG
mZ: 346.50 mG
Roll: -175.24
Pitch: 18.72
Yaw: 45.63

...

END unit test: MPU9250

START unit test: BMA180
Initializing BMA180...done
Sampling the sensor (please move the device):

aX: -0.26 g  aY: 0.42 g  aZ: 0.90 g
aX: 0.22 g  aY: -0.09 g  aZ: 1.02 g
...
aX: 0.06 g  aY: -0.34 g  aZ: 1.02 g
aX: -0.50 g  aY: -0.22 g  aZ: 0.75 g
END unit test: BMA180

START unit test: Port expander
Initializing...done
Detecting 10 key presses. Press each key only once.
Key pressed: 0
Key pressed: 1
Key pressed: 2
Key pressed: 3
Key pressed: 4
Key pressed: 5
Key pressed: 6
Key pressed: 7
Key pressed: 8
Key pressed: 9

Testing LEDs
LEFT: RED
LEFT: GREEN
LEFT: BLUE
LEFT: OFF
```

```
RIGHT: RED
RIGHT: GREEN
RIGHT: BLUE
RIGHT: OFF
done

END unit test: Port expander

START unit test: WiFi scan
Initializing...done
scan start
scan done
15 networks found
1: FRITZ!Box 7590 UE (-72)*
2: Tenda_F2BD78 (-74)*
3: TP-LINK_65E2 (-74)*
4: FRITZ!Box 7590 UE (-77)*
5: Yanni (-78)*
6: TP-Link_FDAA (-84)*
7: joes apple time cap (-85)*
8: link-link (-86)*
9: TP-Link_37EC (-87)*
10: TP-LINK_8808 (-87)*
11: FRITZ!Box Fon WLAN 7360 (-89)*
12: NETGEAR24 (-90)*
13: TP-Link_B78M (-91)*
14: Wavlink-N (-93)*
15: NETGEAR56 (-93)*

END unit test: WiFi scan

START unit test: Buzzer
Initializing...done
Duty cycle 50%
Frequency:
500Hz
Frequency:
1000Hz
Frequency:
1500Hz
Frequency:
2000Hz
Frequency:
2500Hz
Frequency:
3000Hz
Stop
END unit test: Buzzer
```

Glossary

3D three dimensional. 14, 20, 22, 84

A2DP Advanced Audio Distribution Protocol. 89

ADC Analog-to-digital converter. 12, 13, 17, 25, 36, 37, 39, 62

AP Access Point. 27, 28, 50, 52, 53, 55, 59, 72, 73, 87

AVRCP Audio/Video Remote Control Protocol. 89

BLE Bluetooth Low Energy. 17, 32, 87, 88

BT Bluetooth. 20, 27, 28, 32, 83, 84, 87, 89

CPU central processing unit. 12, 13, 17

DAC digital-to-analog converter. 12

DC Direct Current. 20

DOF Degrees Of Freedom. 20

DSP Digital Signal Processing. 13

EEPROM Electrically Erasable Programmable Read-Only Memory. 26, 27

FPU Floating-point unit. 88

GPIO General-Purpose Input/Output. 17, 19, 39

GUI graphical user interface. 22

I^2C Inter-integrated circuit. 15

IC integrated circuit. 23

IDE integrated development environment. 21, 24, 28, 32, 55, 57, 59

IO input/output. 12, 13, 16, 19, 38, 39

IoT Internet of Things. 17, 24

IP Internet Protocol. 52, 59, 72

ISR Interrupt Service Routine. 48, 88

LED Light-Emitting Diode. 19, 20, 25, 26, 39, 50, 60, 69, 84, 85

MCU Microcontroller Unit. 12, 17, 18, 19, 24, 25, 26, 29, 30, 31, 32, 33, 36, 37, 38, 48, 63, 73, 76, 77, 83, 84, 87, 88

MISO Master In Slave Out. 14

MOSI Master Out Slave In. 14

OLED Organic Light-Emitting Diode. 17, 19, 25, 26, 27, 31, 52, 54, 59, 69, 71, 72, 73, 81, 87

OOP Object-Oriented Programming. 21, 34, 49

OS Operating System. 66, 67

PC Personal Computer. 24, 25, 26, 27, 28, 33, 37, 49, 53, 61, 62, 63, 64, 66, 71, 72, 73, 74, 77, 78, 79, 81, 83

PCB printed circuit board. 16, 17, 23, 25, 26, 29, 30

PWM Pulse Width Modulation. 17, 19

RC Radio Control. 24, 83, 88

RGB Red Green Blue. 19, 25, 84

RPY Roll, Pitch, Yaw. 8, 14, 26, 27, 54, 57, 64, 65, 73

Rx Receive. 16

SCK Serial Clock. 14

SCL Serial Clock. 15

SDA Serial Data. 15

SPI Serial Peripheral Interface. 14, 15, 17, 19, 20, 31, 39

SS Slave Select. 14

TCP/IP Transmission Control Protocol/Internet Protocol. 22, 27, 28, 59, 72, 73

Tx Transmit. 16

UART Universal Asynchronous Receiver/Transmitter. 16, 17, 20, 38, 39, 84

UDP User Datagram Protocol. 22, 27, 28, 59, 62, 65, 72, 73, 78, 81

UI User Interface. 28, 87

UML Unified Modeling Language. 23

USB Universal Serial Bus. 16, 17, 25, 27, 29, 30, 33, 36, 37, 39, 47, 50, 53, 62, 63, 64, 66, 67, 69, 71, 72, 73, 74, 77, 81, 83

WORA Write Once, Run Anywhere. 21

Declaration

I declare within the meaning of part 16(5) of the General Examination and Study Regulations for Bachelor and Master Study Degree Programmes at the Faculty of Engineering and Computer Science and the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, November 4, 2019

City, Date

Signature