BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

HUMAN MO-CAP SYSTEM BASED ON INERTIAL MEASUREMENT UNITS

SYSTÉM PRO SNÍMÁNÍ POHYBU ČLOVĚKA ZALOŽEN NA INERČNÍCH MEŘÍCÍCH JEDNOTKÁCH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR   **Bc. MARTINA GRZYBOWSKÁ**
AUTOR PRÁCE

SUPERVISOR   **Prof. Ing., Dipl.-Ing. MARTIN DRAHANSKÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Department of Intelligent Systems (DITS)   Academic year 2020/2021

Master's Thesis Specification

20602

Student: **Grzybowská Martina, Bc.**
Programme: Information Technology
Field of   Information Technology Security
study:
Title: **Human Mo-cap System Based on Inertial Measurement Units**
Category: Embedded Systems
Assignment:
1. Study the literature on human motion capture - review the presently available methods, primarily focusing on inertial human motion capture, using inertial measurement units (IMUs).
2. Create an overview sumarizing properties of MPU-6050 inertial motion sensor and ESP32 microcontroller, as well as available options for data synchronization of multiple ESP32 modules.
3. Design a system consisting of hardware devices based on ESP32 and MPU-6050 for capturing human motion data and software for data processing.
4. Implement a laboratory sample of the proposed system - create the hardware solution, implement firmware for the sensor device and software for data collection and basic visualization.
5. Test the implemented solution and summarize the achieved results.

Recommended literature:
- YAHYA, Muhammad, et al. Motion capture sensing techniques usedin human upper limb motion: a review. *Sensor Review*, 2019.
- HILMERSSON, Kristoffer; GUMMESSON, Filip. Time Synchronization in Short Range Wireless Networks. 2016.

Requirements for the semestral defence:
- Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Drahanský Martin, prof. Ing., Dipl.-Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: November 11, 2020

Abstract

The aim of this thesis is to design, construct and implement a custom inertial motion capture system. Though multiple techniques have been studied, the primary focus is placed on inertial motion capture itself – its merits and demerits, key properties and components necessary for construction of an inertial-based system. The preliminary information-gathering is followed by the design, implementation and evaluation phases, which deal with presenting the process of developing and testing the solution. The main contribution of the system implementation is the construction of hardware motion capture devices, i.e. small, lightweight, battery-powered wearable bands, which are completely wireless – both in terms of communication with the outside world as well as in their Qi-compliant charging capabilities.

Abstrakt

Cieľom tejto práce je navrhnúť, zhotoviť a implementovať vlastný systém pre zachytávanie pohybu založený na inerciálnych meracích jednotkách. V rámci budovania teoretického základu bolo preskúmaných viacero metód, avšak primárne bola pozornosť venovaná samotnému inerciálnemu snímanu – jeho kladom a nedostatkom, kľúčovým vlastnostiam a jednotlivým komponentom potrebným pre zostrojenie systému na jeho báze. Tento úvodný zber informácií je nasledovaný fázami návrhu, zhotovenia a zhodnotenia, ktoré sa zaoberajú procesom vývoja a testovania daného riešenia. Hlavným prínosom realizácie systému je zostrojenie zariadení pre snímanie pohybu – jedná sa o malé, ľahké, batériovo napájané zariadenia, ktoré sú kompletne bezdrôtové, či už z hľadiska komunikácie s okolitým svetom, alebo vďaka napájaniu kompatibilnému so štandardom Qi.

Keywords

inertial motion capture, inertial measurement units, MPU-6050, microcontroller, ESP32, wireless sensor network, MQTT, ESP-NOW, wireless charging

Klíčová slova

inerciálne snímanie pohybu, inerciálne meracie jednotky, MPU-6050, mikrokontrolér, ESP32, bezdrôtová senzorová sieť, MQTT, ESP-NOW, bezdrôtové nabíjanie

Reference

Rozšířený abstrakt

V prvopočiatkoch tvorby kreslených filmov animátori zápasili so strnulosťou a neprirodzenosťou pohybov kreslených postáv. V roku 1917 si americký výnálezca Max Fleischer nechal patentovať špeciálnu techniku, ktorá mala revolucionalizovať proces animácie – táto metóda dodnes nesie názov *rotoskopia*. Podstata rotoskopie spočívala vo využití zariadenia zvaného *rotoskop* na premietanie hraných filmov na sklenenú dosku a ručného obkresľovania zobrazovaných hercov či zvierat – snímok po snímku. Hoci zdĺhavá a časovo náročná, táto technika položila základy pre myšlienku automatizácie rekonštrukcie snímaného pohybu – súhrnne sa táto oblasť nazýva *motion capture* (skrátene *mo-cap*).

Dnes, viac ako sto rokov od vynálezu rotoskopie, je používaných mnoho konceptov pre zachytávanie pohybu, umožňujúcich snímanie v rôznych merítkach či na rôznych úrovniach komplexnosti – sme rovnako schopní úspešne zachytávať vzdialený pohyb subjektu v priestore, ako aj rozpoznávať gestá rúk či sledovať kontrakcie mimických svalov na tvári. Princípy, na ktorých sú novodobé snímacie metódy založené, sú dvojsečná zbraň, prirodzene prinášajúca svoje výhody aj nevýhody – väčšina techník, ako optické či magnetické zachytávanie, je závislých od nastavenia scény a špeciálneho, mnohokrát finančne náročého vybavenia. Práve inerciálne snímanie pohybu, založené na inerciálnych meracích jednotkách, je technika nezávislá od lokácie snímania – popularita tejto metódy neustále rastie v dôsledku zmenšujúcich sa rozmerov i nákladov na snímacie senzory.

Hlavným cieľom tejto práce je navrhnúť, skonštruovať a otestovať vlastný základ pre inerciálny mo-cap systém založený na MPU-6050 jednotkách, hostovaných mikrokontrolérmi ESP32. Kapitoly 2 až 4 sú venované tvorbe teroretického základu. Kapitola 2 sa zaoberá popisom základných princípov, výhod a nevýhod štyroch najpoužívanejších snímacích techník: optickej, mechanickej, magnetickej a inerciálnej. V závere kapitoly je tento náhľad využitý pre zhrnutie aspektov, ktoré autori mo-cap systémov propagujú na svojich výrobkoch ako priaznivé voči konkurencii – tento zoznam sa stal smerodajným pre celý zvyšok práce. Kapitola 3 pokladá základ pre prácu s MPU-6050 a ESP32. MPU-6050 je zariadenie pre zachytávanie pohybu, kombinujúce 3-osý MEMS gyroskop, 3-osý MEMS akcelerometer a vstavaný proprietárny procesor pre komplexné spracovanie nameraných dát, nazývaný DMP (Digital Motion Processor). Toto zariadenie vyžaduje pripojenie na externý mikrokontrolér, s ktorým komunikuje pomocou I^{2}C rozhrania – pre tento účel je využité práve ESP32 od firmy Espressif systems. Kapitola 4 sa zaoberá komunikačnými technológiami, ktoré ESP32 podporuje natívne (bez prídavných komunikačných modulov), a taktiež predstavuje problematiku synchronizácie viacerých ESP32 zariadení fungujúcich v spoločnej sieti.

Vychádzajúc z tohto teroretického základu, kapitoly 5 až 7 predstavujú proces návrhu, jeho prevod do implementácie a jej následné otestovanie. Výsledný systém pozostáva zo štyroch separátnych častí komunikujúcich pomocou aplikačného protokolu MQTT: senzorických zariadení, procesujúcej aplikácie, vizualizačného užívateľského rozhrania a MQTT brokeru. Senzorické náramky, ktorých úlohou je generovať dáta o pohybových trajektóriách, sú hlavným prínosom tejto práce. Jedná sa o pomerne malé zariadenia o rozmeroch 8.65×30.0×30.0 mm, vážiace 8 gramov, založené na vlastnom návrhu dosky plošných spojov. Tieto náramky boli zostrojené tak, aby spôsobovali čo najmenšiu prekážku v prirodzenom pohybe: komunikácia aj nabíjanie jednobunkovej li-polymér batérie, ktorá slúži ako zdroj energie, sú vykonávané bezdrôtovo. S vonkajším svetom náramky komunikujú pomocou protokolu MQTT, zatiaľ čo synchronizácia medzi zariadeniami navzájom prebieha pomocou protokolu ESP-NOW. Nabíjanie batérie je zabezpečené pomocou prijímača CP2021, kompatibilného so štandardom Qi.

Hlavnou úlohou procesujúcej aplikácie je zber a ukladanie dát generovaných zo senzorických zariadení, zatiaľ čo užívateľské rozhranie, implementované v Javascriptovom frameworku Vue.js, sa stará o zobrazovanie orientácií senzorických zariadení v reálnom čase ako pomocou grafov, tak cez animácie príslušných 3D modelov. Tieto aplikácie boli implementované tak, aby boli jednoducho rozšíriteľné o nové vlastnosti.

Testovanie výsledného riešenia prebiehalo v dvoch fázach, ručne aj automatizovane – komponenty boli najprv testované samostatne, následne bol systém otestovaný ako celok. Za účelom otestovania správnosti interpretácie zachytených pohybov boli vykonané celkovo tri experimenty, pričom séria pohybov vykonaná pre každý experiment bola zopakovaná niekoľkokrát – z takýchto sérií bol za pomoci algoritmu Dynamic Time Warping (DTW) vytvorený jeden reprezentatívny profil pohybu, ktorý bol následne vyhodnocovaný. Pre tento účel bola procesujúca aplikácia rozšírená o funkcionalitu modifikovanej verzie DTW, pracujúcej s multidimenzionálnymi časovými radami, kde je vývoj jednotlivých dimenzií v čase navzájom previazaný. Záver tejto práce sa zaoberá zhrnutím nedostatkov zistených počas fáze testovania, spôsobmi, ako by bolo možné ich dopad minimalizovať, a taktiež predstavením funkcionalít, ktoré by systém v budúcnosti mohol potenciálne obsahovať.

Human mo-cap system based on inertial measurement units

Declaration

Hereby I declare that I have composed this master's thesis autonomously under the guidance of Mr. Drahanský. All the relevant information sources, which were used during the preparation of this thesis, are properly cited and included in the list of references.

....

Martina Grzybowská

May 14, 2021

Acknowledgements

Contents

Chapter 1

Introduction

The meaning of *"to animate"*, stemming from Latin *"animare"*, is *"to give life to"*. *Animation* is a term presently referring to the process of creating an illusion of movement of an otherwise inanimate object, by displaying a sequence of frames, each containing a partial change in the object's pose. It seems to be rather fitting that the early endeavours to accelerate the process of creating these frames – of "bringing objects to life" – inadvertently *gave life* to a cutting-edge technique presently employed across a wide variety of domains: *motion capture*, commonly referred to as *mo-cap*.

We have come a long way from these initial endeavours – at present, there are several approaches to motion capture, enabling movement estimation on various levels of complexity: ranging from determining the position and orientation of a body in a multidimensional space, through recognizing hand gestures, to recording contractions of the mimic muscles forming emotional expressions. Nonetheless, with each method having its own merits and demerits, generally emerging from the very concepts they are built upon, the degree of suitability for certain tasks naturally tends to differ for various mo-cap systems, especially regarding the required scene setting. It is *inertial motion capture*, self-contained and non-invasive, based on *inertial measurement units*, that has been gaining popularity in recent years: both due to the ever-shrinking sensor cost and size, as well as the ability to capture motion in situated environments.

The aim of this thesis is to lay the foundations of a custom inertial mo-cap system by designing and constructing a hardware solution for sensing human movements, as well as implementing a software for gathering the captured data. In Chapters 2 to 4, a preliminary knowledge base is established, including research into the concepts, technologies and methods employed in the later parts of the thesis. More specifically, Chapter 2 deals with the principles of various mo-cap techniques with the primary focus on inertial motion capture, along with determining the desired qualities based on a research into existing inertial motion capture projects. Following the presented findings, Chapter 3 describes the hardware components for the construction of custom inertial mo-cap devices, and Chapter 4 introduces potential candidates for the role of a carrier of internal communication.

Based on the knowledge acquired through studying these preliminaries, a proposal for a custom inertial mo-cap system is created and presented in Chapter 5, whereas the process of transforming this design into an actual realisation is described in Chapter 6. Chapter 7 concludes the thesis with testing and evaluation of the constructed solution, also suggesting potential improvements for its future fine-tuning.

Chapter 2

Methods for motion capture

In the early days of cartoons, animators struggled with improving the stiff and unnatural movements plaguing their creations. Then, in 1915, along came *rotoscoping*, a technique poised to revolutionize the process of animation, patented by Max Fleischer. The essence of this method lies in the employment of a device called a *rotoscope* for projecting a motion picture film onto a glass panel, manually tracing the contents of each frame and copying postures of the projected actors or animals. Although lengthy and time-consuming, this technique became instrumental in creating an illusion of a fluid, life-like movement in illustrations [42].

As it stands, a century ago, rotoscoping laid foundations for the idea of motion reconstruction, nowadays coined as *motion capture* – a data-collecting technique utilized for *measuring an object's position and orientation in physical space* [30], extracting the captured motion into a representation allowing for further processing and analysis [42].

Almost twenty years ago, the authors of article [54] pointed out a staggering number of approaches to capturing human motion, based on a variety of principles and presenting different performance characteristics, yet all developed for the same purpose. This "boom" in available methods allowed for today's systems to become more specialized, varying the complexity of the information being captured. It comes as no surprise that motion capture graduated from the entertainment industry into areas such as computer vision, robotics or medical and sport analysis.

Having briefly described how motion capture came to be, this chapter serves as a primer to mo-cap methods, introducing presently available approaches in Section 2.1, although primarily focusing on the fundamental principles of one method in particular – *inertial motion capture* (Section 2.2).

2.1 Review of available approaches

As the authors of article [54] wrote, *there is no silver bullet* among the available motion tracking technologies, as they all have their own strengths and weaknesses. Following review of available mo-cap approaches is a summary of information comprising multiple primary sources: [56], [42], [54], [30], [43], [29] and [49].

2.1.1 Optical motion capture

Optical motion capture – *video tracking* – is a process of locating the position and orientation of a subject by means of synchronized video cameras [56]. The stream of the

provided frames should be continuous, with each frame analyzed individually. Currently, there are two fundamentally different approaches to optical motion capture: *marker-based* and *markerless.*

Marker-based methods

As the name suggests, marker-based methods utilize *markers* – elements affixed to the strategic points on the performer's body. Several high-resolution cameras are then positioned around the measurement workspace to track the motion of these markers during the subject's performance. The subsequent reconstruction of the movement is based on matching the markers from various vantage points provided by the cameras, using triangulation to compute the markers positions in 3D space for each frame [30]. There are two types of markers: either *active*, battery powered diodes emitting infra-red light (IR LEDs), or *passive*, coated by retro-reflective material designed to reflect incoming light back to its source, in which case the cameras utilized for tracking are responsible for emitting the light themselves[1]. Marker type notwithstanding, to measure the infra-red light emitted/reflected, the cameras must also be fitted with infrared pass filters [54]. The main problem with marker-based optical methods lies in marker obstruction – to tackle this issue, redundant markers are often employed. Even though this approach reduces the probability of error occurrence, it also increases the processing latency [49, 30, 42].

Markerless methods

The traditional marker-based methods, along with mostly being restricted to indoor venues only, are impractical when the markers themselves might hinder the activity to be captured, e.g., sporting games. Due to this, markerless optical methods are a subject of extensive research [49]. As of today, there are multiple approaches being developed, some of them purely software-based (employing various computer vision processing algorithms), while some require special hardware – e.g., a depth-sensitive camera, emitting infrared light to perceive depth, such that each pixel also contains information about the distance to the object [56]. In recent years, usage of the LiDAR (Light Detection and Ranging) technology is also gaining attention – this method is based on calculating the Time of Flight (ToF) of a near infrared light beam released from a laser and reflected off the surface of the surrounding objects. Such 3D or Doppler LiDAR scans might then be further processed for detecting clusters of moving points, detecting objects in motion [41, 27].

2.1.2 Mechanical motion capture

Mechanical motion capture involves tracking the rotation angles of the subject's joints. To this end, the performer is required to wear a skeletal-like structure consisting of rods connected by sensorized joints – referred to as an *exoskeleton*. This device was designed to mimic the structure of a human body – rods for bones and joints for, well, joints – following the performer's movements, ideally having the angles formed between the rods match the respective inter-segmental angles (i.e. angles between two body segments on either side of joints). To measure these angles, each exoskeleton joint is equipped with electromechanical transducers capable of converting displacement to voltage output through changes in resistance (such as potentiometers) [30, 42]. Having collected data from the transducers,

---

[1]usually through the means of infra-red LEDs placed around the camera lens

the relative position and rotation of the individual limbs may be calculated, followed by inferring the pose of the entire body.

The main issue with this approach, apart from the obvious movement limitations caused by the exoskeleton structure, lies within determining the absolute position of the performer in regard to the surrounding environment – to achieve this, an external positioning technology has to be introduced into the system [49].

2.1.3 Electromagnetic motion capture

Electromagnetic motion systems are based on the principles of electromagnetism, employing two main elements for determining the position and orientation of the performer: *sensor units*, affixed to the performer's body (along with a control unit), and stationary antennas, referred to as *transmitters*. The main role of a transmitter is to generate three orthogonal electromagnetic fields by having an electric current (either alternating or direct, based on the system type) flow through a set of metallic coils – a process referred to as *excitation*. Generally, three separate antennas are used – one for each spatial direction – however, it is not unusual to have a transmitter consisting of three orthogonally oriented metal coils enclosed within a single package [49, 30]. The type of the electromagnetic field produced by the transmitter also influences how the sensing devices are to be designed: in AC-based systems, they also consist of three orthogonal coils, yet in DC-based systems, these are substituted with three magnetometers, measuring the magnitude of the magnetic field. The control unit collects the voltage readings in each axis generated by the electromagnetic induction within the devices, determining tridimensional vectors indicating the devices' position and orientation with respect to the excitation (transmitter).

There is no known material able to block an electromagnetic field without being attracted to the magnetic force itself[2] – this is an equally great advantage and disadvantage at the same time. As the human body is incapable of blocking the electromagnetic field, the issue of sensor obstruction, causing much heartache in optical mo-cap systems, seems to no longer be an issue at all. On the other hand, new electromagnetic fields are being generated with the occurrence of metallic objects in the immediate vicinity of the transmitters, and this interference is bound to skew the measurements [49].

2.2 Inertial motion capture

Briefly, one could summarize the inertial mo-cap systems as follows: in inertial motion capture, as the name indicates, recording the motion trajectories is based on a series of *smart* devices containing *inertial measurement units* (IMUs) – self-contained units usually comprising *inertial MEMS sensors* – attached to a performer's body, whereas the performer's position and orientation at a point in time are computed analytically through a process similar to *dead-reckoning* [49]. Since this description contains terms which have yet to be explained, and the goal is to dive deeply into the principles under which inertial motion capture operates, the following sections shall start from the very basic definitions (Section 2.2.1), gradually building up to describing the system as a whole (Sections 2.2.2 and 2.2.3).

---

[2]https://www.uu.edu/dept/physics/scienceguys/2004Feb.cfm

2.2.1 Ingredients of an inertial measurement unit

The following definitions are based on [35] and [37]. The term *sensor* might have already been "thrown around" here and there, although its dictionary definition would be that of *a device which responds to a physical stimulus and transmits a resulting impulse*[3]. As such, a sensor is usually comprised of three chained components: a *sensing section* in direct contact with the monitored environment, a *processing circuitry* responsible for converting the physical property into a measuring (most often electrical) quantity, and an *outputting interface*. A *smart* or *intelligent* sensor generates processed, actionable data instead of raw output, employing a built-in microprocessor, usually transmitting the data through a network to a collecting unit. An *inertial* sensor determines the position and orientation changes with respect to the outer inertial reference system. A *MEMS* (micro-electro-mechanical system) sensor is a microscopic device (generally ranging from 0.02 up to 1 mm) manufactured using one of the microelectronic fabrication methods (such as *etching* or *photolithography*), integrating micromechanical structures and electronic components onto a single substrate.

Having defined the necessary, let us revisit the aforementioned definition: inertial motion capture employs inertial measurement units (IMUs) for capturing motion trajectories. These estimations are subsequently utilized for inferring the current position/orientation of the performer based on their previously known position/orientation – i.e., essentially the process of *dead reckoning*. To acquire these estimates, IMUs usually fuse the output data of at least two (often three) types of MEMS sensors: *accelerometers* and *gyroscopes*, to derive the object's attitude, and *magnetometers* to estimate the heading angle [49]. The following principles' descriptions are based on [37] and [45].

MEMS accelerometer principle

A MEMS accelerometer is a sensor measuring either *static* or *dynamic* acceleration, i.e., the rate of change of the velocity of an object. Static acceleration is measured when no other external forces are applied to the sensor, except for the standard acceleration of a free fall (caused by the force of gravity), while dynamic acceleration is a result of vibrations caused during movement.
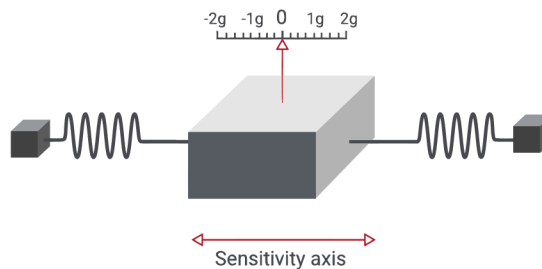
Figure 2.1: MEMS accelerometer – a proof mass suspended on springs, free to move along one axis, denoted as the *sensitivity axis*. The Système International (SI) unit of acceleration is m/s^2, though G-force is also used, where 1 g is approximately equal to $9.8\,\text{m/s}^2$, i.e. the conventional value of gravitational acceleration on Earth. Illustration source vectornav.com.

A capacitive accelerometer operates on the principle of monitoring the position of a *proof mass*, suspended on springs, free to move along a single axis as the device accelerates

---

[3]https://www.merriam-webster.com/dictionary/sensor

(illustrated by Figure 2.1). These springs are attached to a fixed capacitor plate, setting up the capacitive effect. As the proof mass moves under the force acting on the sensor, its distance from the capacitor plate changes, resulting in a change of the capacitance.

MEMS gyroscope principle

A MEMS gyroscope is a sensor measuring the angular velocity of an object (rotating about a specific axis) by means of what is known as *Coriolis*. Also known as the Coriolis effect, this phenomenon refers to the deflection of an object moving within a rotating non-inertial frame, caused by an inertial force called the Coriolis force. Similar to a MEMS accelerometer, a mass suspended by strings attached to a capacitor plate is utilized to measure the displacement resulting from the Coriolis effect (illustrated in Figure 2.2).
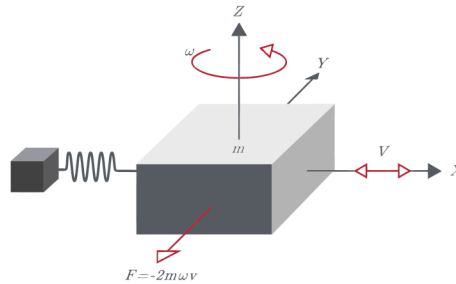

Figure 2.2: MEMS gyroscope – a proof mass (m) suspended on a string, with a driving force on the x axis, forcing the mass to oscillate, with the angular velocity (ω) applied about the z axis. A displacement force (F) is being generated in the y axis as a result of the Coriolis force. Illustration source `vectornav.com`.

MEMS magnetometer principle

A MEMS magnetometer is a sensor measuring magnetic fields operating on the principle of the Hall effect: with an electric current flowing through a conductive plate, electrons flow from one end of the plate to the other. Bringing a magnetic field near the plate causes the electrons to defect from one side to the opposite end. In turn, this influx of electrons induces a voltage, dependent on the strength of the applied magnetic field and its direction.

2.2.2 Interpreting the measurements

A single inertial MEMS sensor measures motion in regard to only one of the three principal axes; in other words, it only has one *Degree of Freedom* (DOF) [55]. Typically, three such sensors are positioned orthogonally to each other within a single package – these devices are then referred to as 3-axis or 3DOF. Combining a 3-axis accelerometer and a 3-axis gyroscope results in a 6-axis (6DOF) IMU, adding a 3-axis magnetometer results in a 9-axis (9DOF) IMU.

What the three MEMS sensors described in the previous section have in common is that they are extremely susceptible to estimation errors. A gyroscope is subject to a time-varying bias due to inherent noise-causing imperfections, and even though it can be countered with initial calibration, these errors tend to accumulate and the estimates will drift over time. On the other hand, accelerometers are sensitive to all vibrations and magnetometers are

naturally sensitive to any rogue magnetic forces. These inaccuracies prove to be an issue for stand-alone sensors, however, one is able to effectively cancel these errors out by combining their individual measurements [40, 55]. This might be taken care of by the IMU's on-board microprocessor – though such a process is usually comprised of multiple steps, often employing proprietary black-box algorithms supplied by the IMU's producers (revisited in 3.2.4) – however, there is also the possibility of employing self-implemented or third-party methods by utilizing the raw data. For instance, in-depth descriptions of these can be found in [55] and [24] – in short, these methods usually entail the following phases:

- **Sensor calibration** – for gyroscopes and accelerometers, this includes the removal of *zero-errors*, e.g., recordings of non-zero values when the device is level, usually through means of calculating an additive *zero-offset* (for example as many-times weighted average of captured estimations) or using low-pass filters. Calibrating a magnetometer requires reducing the jitter and minimizing the interference of surrounding magnetic materials.

- **Data fusion** – fusing the angular rate (provided by a gyroscope) with incline (gravity vector, provided by an accelerometer) and, optionally, magnetometer estimation for more reliable heading data, usually through means of Extended Kalman filters or complementary filters.

- **Data representation** – results are to be converted into a suitable representation. There are multiple methods for representing the orientation of an object with respect to a set of three orthogonal coordinate axes, thought the most commonly used are:

  - **Euler angles** – according to the Euler's rotation theorem, *an arbitrary rotation may be described by only three parameters*[4], with the rotation angles usually denoted as (ϕ, θ, ψ) [40]. When changing the orientation of an object, these elemental rotations are applied successively, with the order of axes depending on the axis convention in use. If a convention involves repeatedly rotating about one particular axis, it is called *Eulerian*, using *Euler angles* (e.g., rotations are applied to axes in order xzx). However, if rotating around all three axes of the system, e.g., a permutation of xyz or zyx, the convention is named *Cardanian* and the angles are, rather famously, known as *yaw*, *pitch* and *roll* [55].

  - **Quaternions** – the main issue with Euler or Cardanian angles makes itself known when one of the orientation axes becomes parallel to any of the remaining ones (i.e., when the "gimbals line up") due to the process of rotation. This is referred to as the *Gimbal lock* singularity, causing a loss of one DOF within the rotation system [40, 55]. On the other hand, *quaternions* do not suffer from such shortcomings. A unit-vector quaternion, denoted as (a, b, c, d), is a vector defined in four-dimensional space, composed of two components: a scalar value a corresponding to the angle of rotation, and the vector part (b, c, d) corresponding to the vector about which the rotation is to be performed [40].

2.2.3   Summary of properties

Due to the cost-effectiveness of today's IMUs and the broad possibilities of research, IMU-based motion capture has spanned to multiple spheres: from personal projects [52, 32],

---

[4]

through university-based complex solutions [24] to relatively expensive proprietary systems such as *Notch* by Notch Interfaces[5]. Based on these, one can assume the following: IMU-based mo-cap systems usually take the form of small bands worn on a body (with their number depending on which body part's movement one wishes to capture) and an arbitrary unit for collecting and post-processing the data. These systems are, above all else, self-contained, adaptable and non-invasive, allowing for tracking of human motion in any environment [56]. The following list is a compilation of desirable qualities for any mo-cap system (described in [54] and [42]), along with the aspects authors of the above-mentioned IMU-based systems described as favourable regarding their products:

- **tiny dimensions** – miniature dimensions of today's MEMS sensors and microprocessors allow for bands housing the devices to be light and small-sized,

- **battery-powered** – most IMUs are designed to be low-power, therefore small, preferably rechargeable batteries should be able to sustain them,

- **wireless communication** – with an additional microprocessor module allowing wireless data transmission, the bands do not have to be connected by wires,

- **no hindrance to movement** – due to the previous three characteristics, along with requiring no external structures or suits, the bands pose no restrictions to the performer's movement,

- **location-agnostic** – no measurement workspace is needed, as they do not utilize any additional equipment such as high-resolution cameras or magnetic transmitters.

---

[5]https://wearnotch.com/

Chapter 3

Hardware components

Section 2.2 introduced the basic concepts and principles an inertial mo-cap system operates on. The main objective of this chapter is to follow up and present the physical hardware components required for assembling a smart device capable of capturing the movement trajectory, pre-processing the data and delegating it, in accordance with the desired properties specified in 2.2.3. Naturally, this device is to be built around an IMU – as dictated by the specification of this thesis, this is to be the MPU-6050 by InvenSense (described in 3.2). Since this IMU requires a connection to a hosting microcontroller, the specification further dictates employing an ESP32 (3.3). Finally, this device is to be battery-powered, calling for incorporating a charging circuit along with a battery – this shall be addressed in 3.4.

3.1   Terminology

Prior to commencing the presentation of the components themselves, a couple of terms, which shall later be referenced, need to be defined. The following concepts are quite common in the world of hardware.

System time.   All integrated circuits depend on a *clock source* – an electronic circuit generating an electrical signal with a precise frequency, usually utilizing the mechanical resonance of a vibrating crystal [34]. Built upon the output of a clock source, often preprocessed by a *divider* (also called a *pre-scaler*), are pulse-counting *timers* – counting the steady stream of ticks produced by the source.

Register access management.   The following paragraph is based on [44]. A *register* is a type of fast memory located within a processing unit, utilized for temporarily storing small amounts of data. Should one be interested in continuously receiving the contents of a register upon its update, there are two different methods for finding out whether its value has indeed changed – *polling* and *hardware interrupts*. Polling is a process in which a processor periodically checks for updates, whereas with interrupts it does not need to perform any checks, as it will be notified of the change by a *hardware interrupt signal* – upon receiving such signal, the processor immediately stops the current process and jumps to a function called *interrupt service routine* (ISR) to read the updated contents.

Serial communication.   The following paragraph is based on [7]. Serial communication is a data transmission method, where data between devices is transferred bit by bit

along a single communication bus. There are multiple serial protocols, although the most frequently used are:

- **I^{2}C** – devices employing the synchronous I^{2}C (*Inter-Integrated Circuit*) protocol are in a master-slave relationship (with the master being the connection initiator), where the slave device might receive instructions from multiple masters, just as a single master might command multiple slaves. I^{2}C only requires two lines: a full-duplex connection carrying the communication data (SDA) and a line carrying the clock signal generated by a master to the slave devices (SCL). Data is transferred in the form of *messages* broken into *frames*, with the first 7-bit frame always carrying the receiver's address.

- **SPI** – akin to I^{2}C, devices communicating through SPI (*Serial Peripheral Interface*) are also in a master-slave relationship; a master can control multiple slaves, however, a slave device is permitted to have only one master. SPI requires four lines: Master Output/Slave Input (MOSI) and vice versa (MISO), clock signal (CLK) and Slave Select/Chip Select (SS/CS). As the receiving device is selected through the SS/CS line, no special form of addressing is required and data is transferred without any interruptions in a single continuous stream.

- **UART** – unlike SPI and I^{2}C, UART (standing for *Universal Asynchronous Receiver/-Transmitter*) is not just a protocol, but a specialized hardware circuit. UART communication is asynchronous – there is no synchronization signal transferred from one device to the other, rather each of these devices generates their own. Only two wires are needed for communication, interconnecting the *Tx* (transmitting) pin of one device with the *Rx* (receiving) pin of the second device, and vice versa. Data is organized into *packets*, each containing a *start bit* signalling the start of a packet, *data bits* carrying the payload, *parity bits* for error detection and *stop bits* signaling the end of the packet.

3.2 MPU-6050 motion tracking device

MPU-6050 is a wildly-used 6-axis IMU combining a 3-axis MEMS gyroscope, 3-axis MEMS accelerometer, an embedded temperature sensor and a Digital Motion Processor – all in one tiny QFN-24 package. It was originally designed by InvenSense specifically for integration into low-power devices requiring complex calculations. With its proprietary *MotionApps* software, featuring 6-axis integration, on-board processing of complex fusion algorithms and run-time calibration, it is said to enable manufacturers to eliminate the need for custom complex solutions, guaranteeing optimal motion performance. Figures 3.1, 3.2 and 3.3 show MPU-6050's orientation of axes, an overview of the building blocks it is comprised of, as well as a summary of its internal registers, respectively – these represent the key building blocks and registers mentioned throughout the Sections 3.2.1 to 3.2.6, using the figures as preliminary references. The following sections are based on [11], [12], [14] and [55].
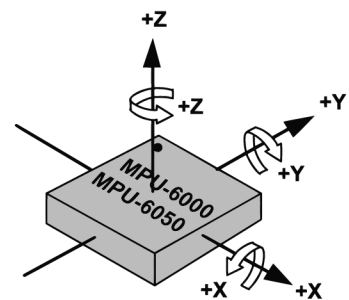

Figure 3.1: MPU-6050's orientation of axes of sensitivity and polarity of rotation. Note the pin 1 identifier (•) in the top left corner. Source [11].
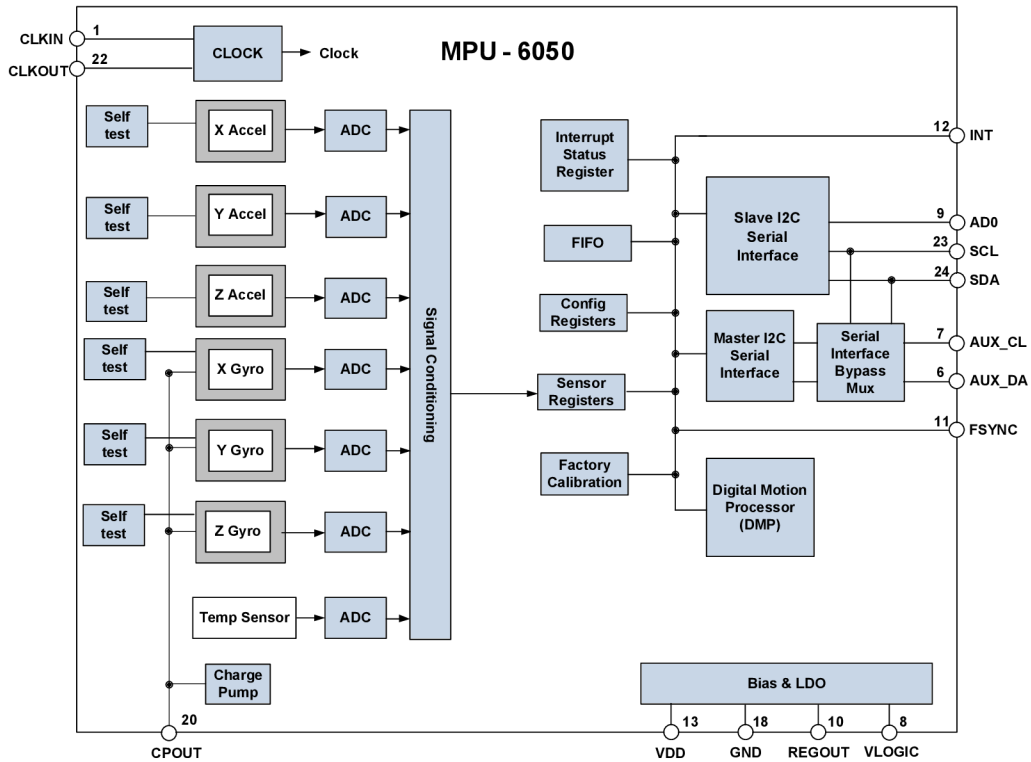
Figure 3.2: A block diagram illustrating MPU-6050 building blocks and basic pin-out: MEMS accelerometer and gyroscope with ADCs, temperature sensor, DMP engine, I^{2}C serial communication interfaces (primary and auxiliary), clocking, self-test/sensor/configuration and interrupt registers, FIFO buffer, Bias and LDO and Charge Pump. Source [11].

Register name	Number	Function
SELF_TEST_[X,Y,A]	13-16	self-test registers
[X,Y,Z][G,A]_OFFS_USR[H,L]	19-24	offset bias registers
SMPRT_DIV	25	sampling rate value
FIFO_EN	35	configuration for the FIFO buffer
INT_PIN_CFG	55	configuration for the interrupt pin
INT_ENABLE	56	flags for interrupt generation
INT_STATUS	58	interrupt statuses of the interrupt sources
ACCEL_[X,Y,Z]OUT_[H,L]	59-64	most recent accelerometer measurements
GYRO_[X,Y,Z]OUT_[H,L]	67-72	most recent gyroscope measurements
FIFO_COUNT_[H,L]	114-115	number of samples in the FIFO buffer
FIFO_R_W	116	r/w data from/to the FIFO buffer
WHO_AM_I	117	upper 6 bits of the I^{2}C address

Figure 3.3: Final overview of the registers, their numbers and functions mentioned in the following sections. A full list of officially accessible MPU-6050 registers can be acquired through combining lists from [12] and [14].

3.2.1   On-board MEMS sensors

As previously explained in 2.2.1, a 3-axis motion sensor measures movements regarding all three principal axes, employing three instances of a measuring system; as shown in Figure 3.2, MPU-6050's accelerometer and gyroscope are no different. Three separate proof masses are used for the accelerometer – a concept also previously mentioned in 2.2.1, as the acceleration along one of the axes causes displacement of the corresponding proof mass, which is then detected by the capacitive sensors – and the output analog values are digitized by three on-board 16-bit analog-to-digital converters (ADCs) enabling simultaneous sampling. The same applies to the gyroscope, which operates on the principle of the Coriolis Effect (explained in 2.2.1). The resulting signal is subsequently amplified, demodulated, and filtered to produce a voltage proportional to the angular rate, and, as with the accelerometer, three dedicated 16-bit ADCs are employed in digitizing the output.

3.2.2   Storing the sensor outputs

The 16-bit sensor readings are generated at a frequency based on the value denoted as the *Sample Rate*, calculated by dividing the gyroscope output rate by the value of `SMPRT_DIV` register. These are then stored, one at a time, in dedicated 8-bit sensor registers (59 to 64 for accelerometer measurements and 67 to 72 for gyroscope measurements). Alternatively, sensor readings might also be stored in the internal FIFO (First-In-First-Out) buffer with a maximum capacity of 1024 bytes, which can be accessed via the `FIFO_R_W` register, provided that the individual sensor's corresponding flag in the `FIFO_EN` register is set to 1.

Following the concept explained in 3.1, there are two ways to manage accessing data written to MPU-6050's registers: either through a polling mechanism or with the help of hardware interrupt signals generated on the `INT` pin (controlled by the `INT_PIN_CFG` register). The interrupt status flags are stored in the `INT_STATUS` register, although in order for the interrupts to be generated, each interrupt source must have its corresponding flag set to 1 in the `INT_ENABLE` register. In particular, the `DATA_RDY_INT` flag is automatically set to 1 whenever a *Data Ready* interrupt is generated, occurring each time the data has been written to all sensor registers.

Regarding the internal FIFO buffer, it is also important to keep track of the number of samples currently buffered, as indicated by the FIFO Count Registers (`FIFO_COUNT_H` and `FIFO_COUNT_L`), mainly for two reasons: should the buffer be found empty or nearing an overflow. If it is empty, reading the `FIFO_R_W` register results in receiving the last byte that was previously read from the buffer. In contrast, when overflown, the status flag `FIFO_OFLOW_INT` located in the `INT_STATUS` register shall be set to 1, the oldest data will be lost and the most recent data shall be written to the buffer.

3.2.3   Device address and communication interface

MPU-6050's registers are accessed via the I^{2}C serial interface with a maximum bus speed of 400 kHz. As a rule, MPU-6050 acts as a slave device (explained in 3.1). The I^{2}C 7-bit address of MPU-6050 is determined as follows: the first six bits are always pre-set to binary value `110100` (stored in register 117, with a rather fitting name `WHO_AM_I`), whereas the value of the least significant bit is determined by the logic level of `AD0` pin: 0 when logical low (resulting in an address `0x68`), and 1 when logical high (`0x69`). This method, enabling two MPU-6050 devices to be connected to the same I^{2}C bus, is customary among many I^{2}C devices [7].

3.2.4 Digital Motion Processor

There are two methods for extracting meaningful data from MPU-6050. As was already mentioned in 3.2.2, one can acquire the raw accelerometer and gyroscope data from the sensor registers or FIFO buffer and carry out all post-processing computations through one's own implementation. Alternatively, one can have the data pass through the sensor's on-board Digital Motion Processor (DMP).

DMP is MPU-6050's internal processing unit, *built with an instruction set designed for very specific and complex mathematical operations necessary for orientation calculation* [47] – its main purpose is generating fused sensor data without intervention from the hosting system's processor. However, the inner workings of DMP are not public – the MPU-6050's manufacturer did not release any information on the proprietary *MotionFusion* and run-time calibration algorithms incorporated into its firmware, nor is there any official resource on what the instruction set is and how it operates [47].

As it stands, DMP acquires digitized data from the on-board accelerometers and gyroscopes. According to the product specification [11], after processing the readings, DMP writes the resulting data in the form of quaternions (explained in 2.2.2) either straight into its dedicated DMP registers or to the internal FIFO buffer – however, such registers are not specifically mentioned in the official register map [12], therefore one can merely assume that the missing registers, such as numbers 109 to 113, are the ones referenced by the documentation. Furthermore, DMP is also supposed to have an access to the `INT` pin, mentioned in 3.2.2, enabling it to generate interrupt signals when processed data becomes available – one must therefore assume that DMP is also one of the possible interrupt sources, even though it is not officially mentioned. These assumptions shall be revisited in section 3.2.6.

3.2.5 Sensors calibration and testing

In 3.2.4, the subject of a MEMS sensor calibration has already been brought up – these sensors, even when stationary, tend to produce false, non-zero readings. Although there are several approaches to counter this issue, the general idea is to compute the average offset biases, which are subsequently taken into account when final readings are produced [24, 55]. These biases can be computed manually and inserted into MPU-6050's built-in offset registers (19 to 24 in Figure 3.3) – they are then automatically applied to the raw data in the sensor registers, before being sent to the FIFO buffer or to DMP for further processing. In addition, per [14], MPU-6050's DMP contains several patented[1] black-box algorithms dealing with further auto-calibration of its integrated sensors.

MPU-6050 also features a built-in method – called a *self-test* – for testing the mechanical portions of the sensors along with the credibility of the produced readings. The self-test can be activated through the self-test registers (13 to 16 in Figure 3.3): upon writing into the self-test register of a sensor, the sensor is being actuated by the on-board electronics, simulating an external force to produce the corresponding readings. These values are then used for obtaining a *self-test response*, which is then compared to the *factory trim* – a manufacturer-provided value incorporated into the MotionApps software, although it can also be obtained manually – whereas in order for the sensor to pass the test, the percentage difference of these values must be within the bounds specified in [11].

---

[1] https://patents.google.com/patent/US20120323520A1/en

3.2.6 Available third-party firmware libraries

As previously mentioned in 3.2.4, very little is known about the inner workings of MPU-6050's DMP. However, through correlating MPU-6050's I^{2}C communication signals against the manufacturer's *MotionApps* MCU software, which comes in a pre-compiled binary form, Jeff Rowberg was able to reverse-engineer enough information to create the very first wrapper over MPU-6050's DMP functionality[2], incorporating it into his open-source library `i2cdev` [47]. Reviewing this library's source code and header files[3], one can summarize that the initial speculations, presented in 3.2.4, were correct: the mystery registers 57 and 109 to 113 are indeed connected to the DMP's functionality, as well as that the first bit in the `INT_ENABLE` register, presumed to be unused, serves as a flag for enabling DMP interrupts, meaning that the DMP truly is another interrupt source.

Nowadays, there are multiple open-source libraries being developed for communicating with MPU-6050, although all are based on Jeff Rowberg's initial endeavours (such as the dedicated library by *ElectronicCats*[4]). Apart from servicing the basic functionality for configuring and controlling DMP, these libraries also provide additional utility functions such as conversion of the output quaternions to obtain values for e.g. yaw-pitch-roll (2.2.2) or gravity-free acceleration.

3.3 ESP32 microcontroller

In 2013, Espressif systems introduced ESP8266 [3], a low-cost *System on a Chip* (SoC) based on a 32-bit RISC architecture capable of effectively embedding Wi-Fi capabilities into other systems. Since its initial purpose was to act as a Serial to Wi-Fi adapter, the original way of communicating with this SoC was to issue AT commands[5] over its UART interface, usually through a USB to UART adapter. However, a dedicated community [3] has seen other firmware options being flashed to ESP8266, turning it into a fully-fledged microcontroller. Noting the ever-increasing popularity of ESP8266, Espressif released its own ESP8266 *software development kit* (SDK) in 2014.

In 2016, ESP8266 was succeeded by ESP32 [19]. In contrast to its predecessor, this SoC was specifically designed for integration into mobile and wearable devices, aiming for ultra-low power consumption. ESP32, running on 3.3 V, housed in a tiny QFN-48 package, comes with a dual-core microprocessor operating at up to 240 MHz (compared to the ESP8266's single-core CPU operating at a maximal frequency of 160 MHz) with 448 ROM and 520 KB SRAM. It has 36 `GPIO` pins, with some of them having direct access to the internal pull-up/pull-down or ADC/DAC circuitry.

There are multiple clock sources (3.1) one can utilize when working with ESP32: be it an external crystal oscillator, an internal phase-locked loop (PLL), or an oscillating circuit. As such, it is possible to have the master clock for both CPU cores – `CPU_CLK`, running as high as 160 MHz in high performance mode – configured to employ any of these sources. Derived from `CPU_CLK` is `APB_CLK`, a 20-bit peripheral clock, which in turn acts as a source for the ESP32's *high resolution timer*, measuring the passage of time at a resolution of 1 μs. This timer provides a utility function for obtaining the number of microseconds passed since its initialization, which happens after the ESP32 start-up [20].

---

[2]http://www.i2cdevlib.com/tools/analyzer/1

[3]https://github.com/jrowberg/i2cdevlib/blob/master/Arduino/MPU6050/MPU6050.h

[4]https://github.com/ElectronicCats/mpu6050

[5]also known as ATtention commands, where every command starts with either `AT` or `at`

3.3.1 Chips, modules and development kits

ESP32 comes in three forms: a bare-bone *chip*, a surface-mountable *module* or a *development board*. Modules are small, shielded, printed circuit boards (PCB), upon which the bare ESP32 chips, along with additional components, are soldered. There are multiple types of modules differing from each other in terms of support for individual components, number of exposed pins or even the number of CPUs, although all of them are ready-made solutions for straightforward integration into final products. Perhaps the most popular among ESP32 modules is *ESP32-WROOM* built on the ESP32-D0WDQ6 chip (Figure 3.4), with an integrated 4 MB flash memory and an on-board printed antenna [21].

Built on top of the modules are the full-featured development boards used for initial prototyping and testing of devices in design. These boards usually feature additional communication interfaces and access to peripherals.
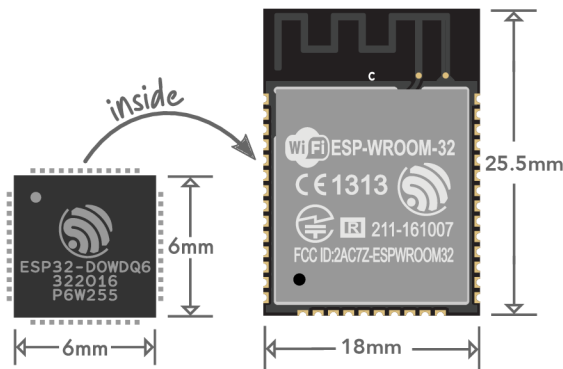

Figure 3.4: ESP32-WROOM module, with an on-board antenna (top part), contains a tiny ESP32-D0WDQ6 chip along with an integrated flash memory, hidden under the shielding. Images taken and edited from esp32.net.

3.3.2 Available communication methods

For wired, serial communication (explained in 3.1), ESP32 contains two I^2C, two I^2S, three UART (in development boards, one is always utilized for a USB-TTL converter) and three SPI bus interfaces (in ESP32-WROOM, one is utilized for the chip's communication with the module's on-board flash memory).

ESP32 features a TCP/IP and full 802.11 b/g/n Wi-Fi MAC protocol implementation for 2.4 GHz band. It can operate in *Access point mode* (AP, sometimes called Soft-AP) or in a *Station mode* (STA) – i.e., either having other devices connected to it or being connected to another device – as well as in a *combined AP-STA mode*, where the device acts as both STA and AP at the same time [19]. Utilizing the Wi-Fi stack, Espressif has also developed two special protocols for wireless communication among ESP devices – ESP-MESH and ESP-NOW – which shall be further described in Section 4.2.3.

Whereas for wireless communication ESP8266 only offers Wi-Fi connectivity, ESP32 also integrates a Bluetooth 4.2 module, supporting both Classic Bluetooth and Bluetooth Low Energy (further described in Section 4.2.4).

3.3.3   Firmware flashing

The following section is based on [38]. Firmware deployment is realized through a process known as *flashing*. ESP32 is reset when `EN` pin is dropped from high to low and back – when `GPIO0` is held low on this reset, ESP32 enters into a *ROM serial bootloader mode*, waiting to receive firmware in binary form through the UART interface. Development boards usually contain a Micro-USB port along with USB-to-UART bridge, automatizing the whole process of flashing – however, since ESP32 modules lack this functionality, an external USB-to-UART converter is often used (provided that one wishes to avoid doing the whole process manually, which is also possible). One such converter, probably the most commonly used, is FTDI FT232R [22].

3.4   Battery and charging

The following paragraph is based on the information acquired from [39]. There are multiple types of rechargeable batteries, although *Lithium-ion polymer* (LiPo), with a typical nominal voltage of 3.7 V per cell [4], seems to be today's most popular choice for wearable devices. LiPo batteries are based on the same principle as *Lithium-ion* (Li-ion) batteries, with cells comprised of positive and negative electrodes separated by a chemical called an *electrolyte*. The most notable difference between Li-ion and LiPo lies in the type of electrolyte material: in LiPo, it is a solid (or semi-solid) polymer, whereas in Li-ion it is liquid. Although not as power-dense as Li-ions, LiPo batteries are extremely lightweight, manufactured in all shapes and sizes.

A LiPo battery must be handled with extra care, as incorrect charging may result in its permanent damage. To this end, these batteries often include a built-in *protection integrated circuit*, preventing charging above the typical maximum safe voltage (4.2 V) as well as discharging below the typical minimum safe voltage (3 V) [4]. Furthermore, it is generally recommended to only charge these batteries through means of specialized *charging circuits*.

Generally, there are two methods employed for interfacing the charging circuit with the external power supply for charging the battery. The first option would be to utilize a *wired connection*, i.e., a connector pair of a *receptacle* and a *plug* (a cylindrical coaxial power connector, for instance), whereas the second option would be *wireless power transfer*. Although there are multiple methods for wireless charging, the most popular standard – Qi, developed by Wireless Power Consortium (WPC) – is based on the phenomenon of *electromagnetic induction*. In this, there is a *power receiver* (PRx) and a *power transmitter* (PTx), both containing a metal coil: PTx runs an AC through its coil, producing an alternating magnetic field, which is then picked up by the PRx's coil and transformed back into an AC [16]. Converted into DC and pre-processed to meet the specified limits, it is then fed to the charging circuit to charge the battery.

Chapter 4

Wireless communication with ESP32

Section 2.2.1 defined what a smart IMU-based device *is*, what it *does*, and *how*. Chapter 3 followed with a presentation of the hardware components such a device might be built of, as well as establishing that ESP32 is to act as the main processing unit. However, in Section 2.2.3, it was also mentioned that a series of such devices is necessary if more than one point on a body is to be tracked, and that the output of all these devices is to be transmitted to a collecting unit to be processed. Hence, what has yet to be discussed are the implications of combining multiple ESP32 devices into a single network, having them synchronize with each other and communicate with the collecting unit.

Let us start with classifying such a network. Based on the properties defined in 2.2.3, one should be dealing with a group of smart devices, attached to one human body, yet communicating wirelessly, each recording physical data. This description fits like a glove to that of a Wireless Body Area Network (WBAN) – a type of Wireless Sensor Network (WSN), operating with devices placed inside, attached to or worn over a human body, executing real-time data recordings of various body parameters. As devices within these networks are referred to as *nodes*, this chapter shall henceforth follow this convention [34].

Since there seem to be next to none set-in-stone standards regarding the communication technologies and dataflows utilized in WBANs, this chapter opens with a brief explanation on when synchronization among ESP32 nodes might be necessary and how it might be achieved (4.1). This is followed by a review of four commonly employed methods for wireless communication with ESP32 devices, presenting their inner workings and underlying architectures (4.2).

4.1   Synchronization of multiple ESP32 nodes

The following section is based on [34]. As it stands, it is quite usual for sensors to record data over periods of time. Such data measurements are only meaningful when ordered by their occurrence; within the context of a single ESP32 node, one only needs to have each measurement accompanied by either a sequence number or a local timestamp to create a series of orientation measurements in time. However, this is not as straightforward when dealing with a network consisting of multiple nodes, each contributing a different aspect of data. In such case, each of the ESP32 nodes provides a single proverbial cog in a wheel of measurements grouped by their common timestamp; one obtains a whole set, *a complete*

motion event, only after aggregating the corresponding orientation measurements recorded by all nodes. This, however, means that data measured by different devices at the same point in time must also be denoted by a timestamp of roughly the same value. Such synchronization might be achieved in two ways: either by having the nodes ignore their local time and create a *common time scale* for the whole network, or simply have each node measure the time relatively to the beginning of a capturing event.

There are multiple methods for time synchronization in short-range WSNs, usually built over common communication technologies such as Wi-Fi or Bluetooth. Authors of thesis [34] focused on determining schemes enabling "as precise as possible" time synchronization among low-energy hardware devices, situated in a short-range network. Their method of choice is *Reference Broadcast Synchronization* (RBS), utilizing broadcast data transmissions for comparing the local times of nodes. In this scheme, timestamps are not transmitted, rather created upon the packet's time of arrival, with the nodes subsequently comparing these values, thus creating a form of a relative timescale within the network.

4.2 Available communication technologies

The following sections describe four representatives of wireless technologies often employed in WSNs, all utilizing different underlying principles – three of them could be described as state-of-the-art protocols specifically designed for low-power WSNs, while the remaining one is very basic in nature. All of these enable short-range communication among multiple ESP32 devices without any additional modules.

4.2.1 UDP-based communication

To start from the very basics, a review on what a communication through plain UDP (User Datagram Protocol) has to offer was conducted, as this method has been successfully utilized in similar projects, for example by Ivo Herzig in his thesis-turned-project *Bewegungsfelder* [32]. UDP [6] operates on the transport layer of the TCP/IP stack, using IP as the underlying protocol. It does not provide any guarantee for successful delivery of messages and there is no mechanism for duplicates detection – as it is, UDP is primarily employed for its small overhead and low latency, since it does not require any kind of acknowledgements from the prospective receiving party before transmitting data. As opposed to TCP (Transport Control Protocol), unicast, broadcast, as well as multicast communication is possible over UDP.

4.2.2 MQTT protocol

MQTT (Message Queuing Telemetry Transport) [17, 33] was introduced by IBM in 1999 and standardized by OASIS in 2013. It is a TCP/IP based[1], application layer, client-server protocol; however, it does not follow your typical request/respond model, where clients initiate communication sessions with servers to request their services. MQTT uses the publish-subscribe messaging pattern – after establishing connections to MQTT servers (called *brokers*), MQTT clients might register for a subscription of multiple information

---

[1]Even though MQTT clients are usually TCP/IP based, it can run over any network protocol providing ordered, full-duplex connections

channels (referred to as *topics*[2]), becoming the topic's *subscribers*, or they might publish messages to these topics, in which case they are called *publishers*. After receiving a message bound to a certain topic, the broker distributes it to the topic's subscribers – this allows for decoupling of client applications, as a client does not need to specify the receivers of its messages. Due to this, and due to its small transport overhead (fixed packet header length of 2 bytes) and protocol exchanges minimized to reduce network traffic, MQTT is very-well suited for communication with constrained devices.

For assessing whether a connection between a client and a broker is active, MQTT allows one to specify a *keep-alive* interval, i.e., the maximum time interval permitted to elapse between the client's transmission of packets. In the absence of meaningful data flowing through the connection, the client transmits a `PINGREQ` packet and the broker responds with `PINGRESP`, confirming that the connection is indeed open and working.

When connecting to a broker, a client can also specify its *last will and testament* (LWT) – a custom message which is to be published to the previously specified topic after the client's untimely demise, e.g, a network failure is detected by the broker, the client fails to communicate within the keep-alive time limit or the connection is closed without first transmitting a `DISCONNECT` packet.

MQTT also provides three built-in QoS (Qualities of Service), bolstering the network reliability:

- **QoS 0** – referred to as *at most once*, where messages are delivered according to the best efforts of the system, meaning that a message loss can occur. The publisher does not store the message and no acknowledgements are sent by the receiver.

- **QoS 1** – referred to as *at least once*, messages are assured to arrive, however, an occurrence of duplicates is a possibility. After the publisher transmits a message, it waits for the receiver's acknowledgement packet (`PUBACK`); if the response is not received within the specified time, the sender re-transmits the message with parameter `DUP` set to 1.

- **QoS 2** – referred to as *exactly once*, messages are assured to arrive exactly once, with no loss or duplicates. This is the slowest method, as it requires four messages in total – `PUBLISH`, `PUBREC`, `PUBREL` and `PUBCOMP`. The sender stores and transmits a message, upon which it awaits a `PUBREC` packet from the receiver. After receiving the acknowledgement, the sender discards the original message, saves the `PUBREC` message and responds with `PUBREL` message, waiting for the receiver to reply with the `PUBCOMP` message. Only after receiving this message, all saved states can be discarded. Whichever stage the process of transmission might be in, if a packet loss occurs, the sender re-transmits the previous message.

In 2007, the first unofficial specification of MQTT-SN (MQTT For Sensor Networks) [13] was introduced. It is specifically adapted for wireless sensor networks, focusing on low bandwidth, and it is also optimized for deployment on battery-constrained devices. The key difference between MQTT and MQTT-SN is that MQTT-SN uses UDP or Bluetooth as an underlying transport protocol, as opposed to MQTT's TCP.

---

[2]The MQTT topics do not need to be initialised prior to being published to, however, their names should follow a certain structure – thus, a topic level separator, a forward slash, was introduced. If present, it separates the topic name into multiple topic *levels*, creating a hierarchical structure of addresses.

4.2.3 ESP-NOW protocol

Published by the Espressif Systems, ESP-NOW [15, 18] is a fast wireless communication method developed for short-packet (with payload up to 250 bytes) transmission in networks consisting of ESP8266 and ESP32 nodes, aiming for low-power consumption. Similar to Wi-Fi, it operates in the 2.4 GHz ISM band, even utilizing the same channels; however, in contrast, it is a peer-to-peer (P2P) protocol (similar to Wi-Fi Direct[3]), requiring a pairing of nodes prior to communicating in unicast or multicast (only optional in broadcast) – once complete, the pairing becomes persistent, i.e. if a node resets, it automatically restarts communication with its peers. ESP-NOW also uses a quite peculiar way for the transmission of packets within its network – in order to comprehend, one must dwell deeply into the workings of the MAC layer frames, as the official ESP-NOW documentation is rather tight-lipped. The following descriptions of MAC layer frames are based on [9].
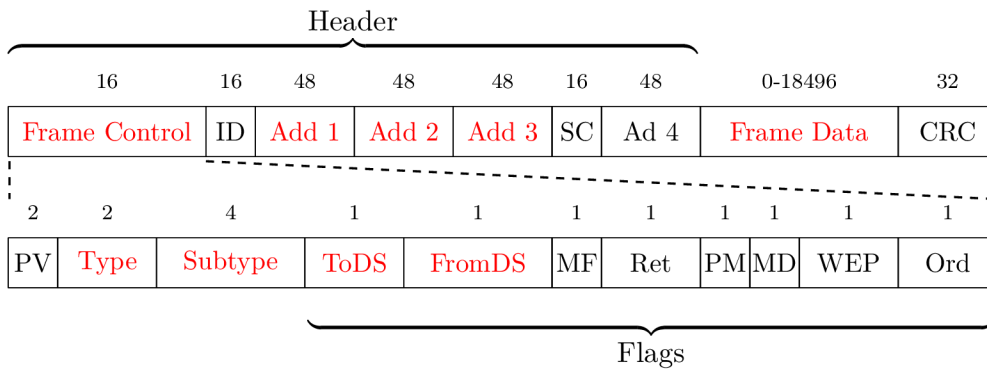

Figure 4.1: Structure of a MAC layer frame, with field lengths in unit of bits, along with detailed structure of the `Frame Control` field. Derived from [9].

As depicted in Figure 4.1, a MAC layer frame consists of 9 fields in total; yet only some of them are particularly interesting:

- **Frame Control** field, perhaps the most important one, is subdivided into further 11 fields (Figure 4.1), with 8 flags:

    - Based on the **Type** field, there are three types of frames specified in IEEE 802.11: *management* (value of 00), *data* (value of 01) and *control* (value of 10). Data frames carry the actual data that is passed down from the higher-layer protocols, with control frames assisting with their delivery. Management frames are used by wireless stations to connect to or disconnect from the Basic Service Set (BSS).

    - **Subtype** field serves for subdividing the `Type` into more categories – there are 12 management frame subtypes, among them the *action* subtype (value of 1110): as their name suggests, action frames are utilized for the triggering of actions.

    - **ToDS** and **FromDS** flags identify whether the frame is a part of a wireless distribution system, or a part of a wireless system, in which case it could simply be passing through, either entering or leaving, or it is a special frame only intended for internal purposes.

---

[3]A standard allowing two devices to establish a Wi-Fi connection directly, not requiring an access point.

- **Address 1** to **Address 3** each contain 48-bit MAC address; their interpretation depends on the setting of the aforementioned `ToDS` and `FromDS` fields – when both flags are set to 0, indicating that it is a special packet for internal purposes, `Address 1` contains the MAC address of the destination, `Address 2` contains the MAC address of the source and `Address 3` contains the MAC address of the BSSID (Basic Service Set Identifier).

- **Frame Body** is a variable-length field containing data specific to individual frame types.

192	8	24	32	56–2040	32
MAC header	Category Code	OUI	RV	Vendor-specific Content	CRC

8	8	24	8	8	0–2000
Element ID	Length	OUI	Type	V	Vendor-specific Body
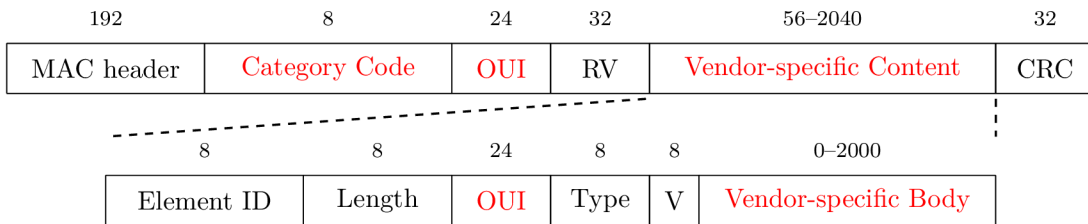
Figure 4.2: Structure of a generic MAC layer vendor-specific action management frame. Derived from [9].

With data frames, the `Frame Body` usually encapsulates headers and payloads of the upper layers; however, as the management frames do not carry any application data, these frames only contain a specialized frame body with a predefined structure. Figure 4.2 depicts the structure of an action management frame:

- **Category Code** indicates the category of the frame, one of them being *vendor-specific* (value of 127).

- **OUI** indicates a public OUI, assigned by the IEEE, of the entity that has defined the context of the vendor-specific action.

- **Vendor-specific Content** is further split into the *Information Element* fields (IE) – variable-length components containing **Element ID**, **Length**, **OUI** and, finally, a **Body**.

Having introduced the basic concepts, let us have a look specifically at the ESP-NOW packet. According to [15], within ESP-NOW packets, the `Type` is set to 00, `ToDS` and `FromDS` flags are set to 0 – a management type. `Subtype` is set to 1110 – an action subtype. `Address 1` is set to the destination address, `Address 2` is set to the source address and `Address 3` is set to the broadcast address (`0xff:0xff:0xff:0xff:0xff:0xff`). `Category Code` is set to 127, making it a vendor-specific category.

To summarize, ESP-NOW "hijacks" the vendor-specific action management frames and encapsulates all user data within their `Body` fields, effectively cutting the overhead of parsing headers of the upper layers of the TCP/IP stack. Combined with not requiring a connection to either an access point or a DHCP server, it is no wonder that the latency of ESP-NOW data transmission is unquestionably lower than with classical Wi-Fi communication.

4.2.4 Bluetooth Low Energy

Bluetooth Low Energy (BLE, also called Bluetooth Smart), is a Wireless Personal Area Network (WPAN) system operating in the unlicensed 2.4 GHz Industrial Scientific Medicinal (ISM) band, integrated into Bluetooth 4.0 *Core Specification* in 2009 along with Classic Bluetooth and Bluetooth high speed. It was designed especially for short-range communication with ultra-low power consumption, intending to replace the wires connecting portable electronic devices – it comes as no surprise that it has become the technology of choice for communication with wearable IoT devices with constraints on battery life. Information presented in the following sections was mainly acquired from [8], [10], [34] and [26].

Architecture, layers and protocols

Similar to the models such as ISO/OSI and TCP/IP, BLE architecture, illustrated in Figure 4.3, also consists of a hierarchy of layers, encapsulated within two major components – *Controller* and *Host* – enabling communication with applications. The format of communication between Host and Controller is standardized by the *Host-Controller Interface* (HCI).
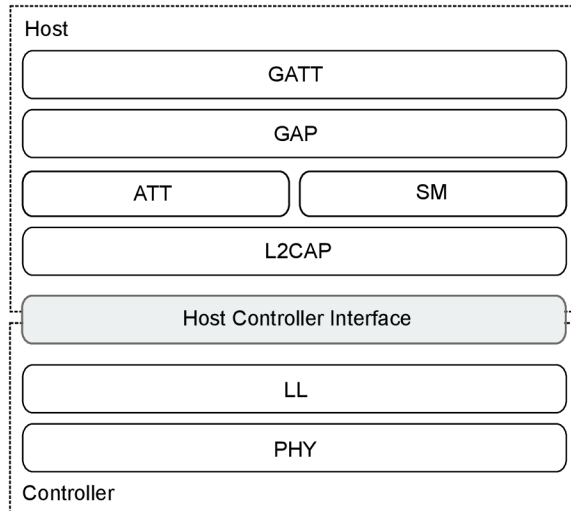

Figure 4.3: Simplified BLE stack architecture. Based on the architecture overview of BLE layers and protocols presented in [10].

As the lowest component of the stack, Controller encapsulates the two lowest layers:

- **PHY** (Physical layer), responsible for modulation and demodulation of analog signals and analog-to-digital conversions,

- **LL** (Link layer), in charge of advertising, scanning, creating and maintaining connections, data encryption and packet filtering.

As a go-between, Host provides a high-level API for applications; implementing a variety of functionalities, it is comprised of multiple network and transport protocols, with GATT (Generic Attribute Profile) and GAP (Generic Access Profile) as the two topmost layers of data and control hierarchy, respectively:

- **GAP protocol** is responsible for overall connection functionality – handling procedures such as device discovery and configuration, link establishment and termination, as well as initiation of security features, utilizing the underlying SM (Security Manager) protocol.

- **GATT protocol.** GATT utilizes ATT (Attribute protocol) as a transport mechanism for exchanging data between devices. Data is structured hierarchically in sections known as *services*; these are collections of conceptually related information called *characteristics*, which, in turn, group together *attributes* (units of generic label-value data). Multiple services can be grouped together to form a *profile*, although these terms are used interchangeably, as it is usual for profiles to only implement a single service. Every service, characteristic and attribute must be uniquely labeled with an UUID (Universally Unique ID) – a 16-bit (predefined) or 128-bit (custom) number.

Roles and connection modes

From a GATT standpoint, a single device can operate in two roles at the same time, even within the context of the same connection: as a *client* or a *server*. A GATT client sends requests to the server, intending to either:

(i) *discover* what services the server has to offer based on their UUIDs (called *service discovery*),

(ii) *receive* the value of an attribute upon a *read* request,

(iii) *update* an attribute's value upon a *write* request.

In contrast, a GATT server contains a database of values – a table of attributes, if you will – with the ability to initiate its own asynchronous operations towards its subscribers:

(i) *notify* the subscribers of a change in an attribute's value,

(ii) *indicate*, which is the same concept as notify, apart from the subscribers also sending acknowledgements of the received notification (making it a blocking operation, since the server waits for the incoming acknowledgements until a timer runs out).

From the standpoint of the GAP protocol, i.e., from the aspect of controlling a connection, the role of a BLE device depends on its attitude to establishing a connection link. If it shuns connections, it is denoted as either a *broadcaster* or an *observer* – in other words, it either has something to *broadcast* to the world (with no care about who might be listening), or it merely wishes to passively *observe* its whereabouts. On the other hand, if a device wishes to establish a connection, it can be either a *peripheral* or a *central* – a peripheral sits at the *peripherals* of the BLE topology, advertising, until a central, usually placed at the very *center*, decides to connect to it. If a connection is successfully established, a central device becomes a *master* and a peripheral becomes a *slave*.

Considering the roles presented by the GAP protocol, there are two main modes devices can communicate in a BLE network: *connection-less* and *connection-oriented*. In the connection-less mode, also called a broadcaster-observer or an advertiser-scanner mode, all transmissions of data packets occur on the advertising PHY channels, in what is called *advertising events* (Figure 4.5) – in this manner, an entire conversation might be carried out without establishing any lasting connection.
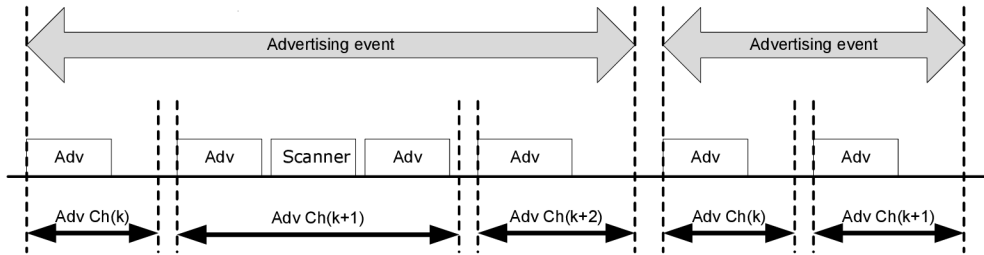
Figure 4.4: At the start of each advertising event, the broadcaster (`Adv`) sends an advertising packet corresponding to the advertising event type to its first advertising channel (`Ch(k)`), and an observer (`Scanner`) might then make a request to the broadcaster, which may be followed by a response. The advertising PHY channel changes (`Ch(k+1)`) on the next advertising packet sent by the broadcaster in the same advertising event. Source [10].

On the other hand, in the connection-oriented mode, centrals, also called *initiators*, listening for connectable advertising packets coming from peripherals, may make a connection request using the same advertising PHY channel on which they received the connectable advertising packet. The *connection event* is considered open while both the master and slave keep on exchanging packets – in the event of being disconnected, both the client and the server allow re-establishment of the connection (upon a trigger operation causing a notification or indication).
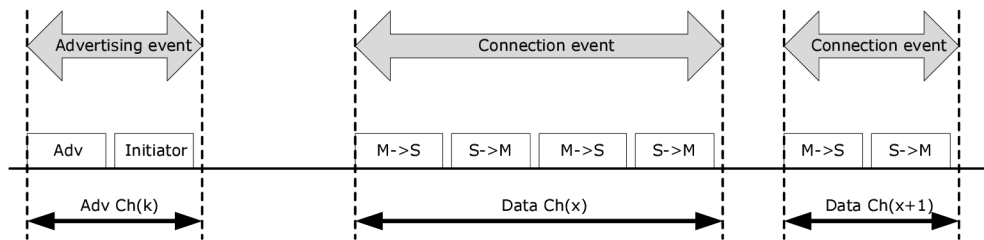

Figure 4.5: If the peripheral (`Adv`) receives and accepts the request for a connection to be initiated coming from the central (`Initiator`), the advertising event is ended and a *connection event* is started by the central, upon which the central becomes a master (`M`) and the peripheral becomes a slave (`S`). Source [10].

Available network topologies

In the Bluetooth 4.0 *Core Specification* [8], a slave node was restricted to a single connection with a single master – it was explicitly forbidden from participating in multiple simultaneous connections with other masters. Hence, originally, the only possible network topology for a BLE network was a *star topology*, with clients individually connected to a central connection point [26]. This changed in Bluetooth 4.1 *Core Specification* [10], released in 2013, where a slave was allowed to be simultaneously connected to more than one master, as well as operate as a slave at certain intervals and as a master at others, effectively keeping parallel communications with surrounding nodes. This allowed for creating advanced network topologies such as a *mesh topology*, where all nodes cooperate to distribute data amongst each other, or a *hybrid topology*, which is, simply described, a freestyle.

Chapter 5

Designing the mo-cap system

Up until now, every chapter served an introductory purpose. By studying the available motion capture techniques in Chapter 2, the greatest advantage an inertial-based system has over the other mo-cap methods was identified: it has the predisposition to be unobtrusive, portable and cause no hindrance to the user's movement. It all comes down to the hardware solution: to obtain these qualities, the sensing devices must be light, small-sized and fully wireless – i.e. battery-powered, as well as delegating the captured data wirelessly. These findings were followed with Chapters 3 and 4: describing the hardware components such devices might potentially be built on, along with presenting wireless technologies ESP32-based devices might utilize for communication with each other, as well as with the outside world. Having studied these preliminaries, a design of a custom inertial motion capture system was created – it is, without further ado, presented in the following sections.

5.1   System architecture as a whole

For the sake of decoupling the individual responsibilities, the system is to be split into four parts, each serving a different purpose (Figure 5.1): ESP32-based sensing devices generating motion data, console-based processing application acting as a wrapper over the storage, browser-based visualising application serving as a graphical user interface (GUI), and a collecting server providing a uniform communication interface among the three modules. Reasons for selecting the mentioned component platforms shall be revisited in their respective design sections.

An introduction to the system workflow would be as follows: when powered up, each of the devices establishes a connection with the collecting server and starts generating motion data (B). Collecting server sends the data, unprocessed, to both applications (C, E) – in this case, the GUI renders the stream in real-time and the processing application is in a stand-by mode. Once one of the devices is physically activated by the user, synchronization (A) of the devices is commenced (explained in 4.1). When finished, the capturing of the motion tracking is activated – the processing application listens to the incoming stream and stores the captured representation in the data storage (F). Using the GUI, these captures can be replayed – in this case, the GUI asks (C, E) the processing application to pull the specified file from the storage (F) and simulate the band, by re-transmitting the measurements from the parsed file (E). Although this description is brief, it provides sufficient insight into the inner workings of the system, enabling us to move on to designing the form the actual communication between the components should take.
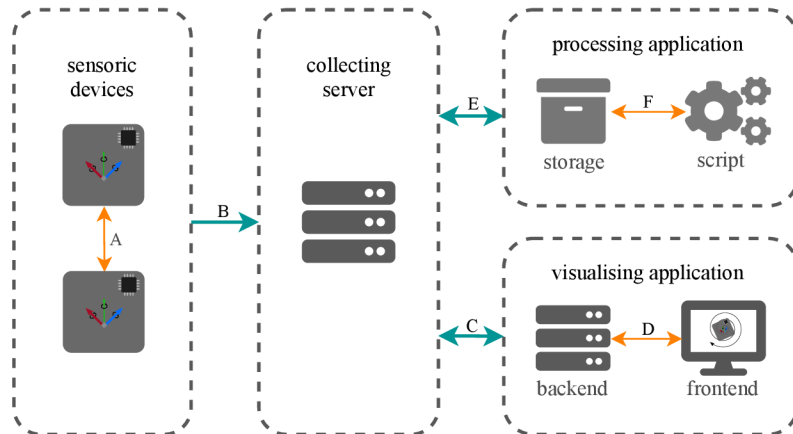
Figure 5.1: Draft of the design of the system components. The synchronised sensing devices (A) act as data generators, delegating the stream to the collecting server (B), in order to be distributed for further processing by the processing application (E), or visualised by the GUI (C).

5.1.1 Inter-component communication design

One can easily identify two different levels of communication streams in Figure 5.1: within the parts of the components and between the components themselves, flowing through the collecting server. Since communication within the components is, once again, discussed in their respective sections, this section mainly focuses on selecting suitable communication methods for inter-component data transmissions, based on the technologies reviewed in Chapter 4. To keep the design as clean as possible, the preference lies in technologies able to function on all three different component systems. As the collecting server is to delegate data streams to applications – unprompted, upon receiving them from the ESP32 devices – this technology should also enable bi-directional communication and provide quality of service settings enabling packet-delivery guarantees. Considering these characteristics, MQTT appears to be the best candidate. As mentioned in Section 4.2.2, apart from being widely used in communication with smart, energy-constrained devices, it also implements publish-subscribe pattern for full-duplex data transmissions between MQTT broker and its clients. Furthermore, MQTT over Websockets – i.e., inserting MQTT packets into Websocket packets – brings support for direct MQTT functionality to browser-based applications.

Establishing the MQTT roles and topic structures

The collecting server will act as an MQTT broker, while the sensing devices, along with both applications, shall take on the role of MQTT clients. Following the proposed workflow, three areas of topics, which need to be covered, can be identified: motion data measurements and capturing commands generated by the sensing devices, commands triggered by GUI, as well as the processing application's responses. Mirroring these, nine topics are listed (Figure 5.2), adhering to the best practices of MQTT topic structuring (4.2.2). The first part of the topic name – `m_t`, short for "motion tracking" – is common for each of the areas, signalizing the topic's affiliation with the motion capture system. The second part delimits the origin of the transmitted message – `band` for data from sensing devices[1], `v_app`

---

[1]ID of the device sending the message shall be located inside payload

for the visualising application and `v_app` for the processing application – whereas the last part describes the content of the payload being transmitted.

I	Origin	Topic	Transmits
0.	sensing device	`m_t/band/status`	band status (on/off)
1.	sensing device	`m_t/band/capturing`	capturing status (on/off)
2.	sensing device	`m_t/band/data`	generated motion data
3.	visualising app	`m_t/v_app/get_status`	trigger for 6.
4.	visualising app	`m_t/v_app/get_filenames`	trigger for 7.
5.	visualising app	`m_t/v_app/get_replay`	trigger for 8. and 9.
6.	processing app	`m_t/p_app/status`	processing app status (on/off)
7.	processing app	`m_t/p_app/filenames`	names of stored filenames
8.	processing app	`m_t/p_app/replay/data`	stored data for replaying
9.	processing app	`m_t/p_app/replay/status`	replaying status (on/off)

Figure 5.2: A list of MQTT topics designed for inter-component communication. Topics 0 to 2 are employed by the sensing devices for indicating their connection status, transmitting the generated motion data, as well as triggering the start of their capturing for storage. Topics 3 to 5, reserved for the visualising application, serve as user-triggered commands for polling responses from the processing application (topics 6 to 9), such as its connection status or names of the stored motion data files.

Designing the communication sequences

Although some of the messages are stand-alone, most serve as triggers, designed to prompt the receivers to execute an action – as such, these request-reply pairs create sequences, each starting its own context within both of the applications. The following paragraphs describe three such sequences.

Initialization of GUI. Upon start-up, GUI polls the topics the processing application subscribes to, to determine its connection status, as well as to request a list of filenames stored in the storage. If active, the processing application responds immediately; if inactive, the GUI shall receive these messages as soon as the processing application establishes connection with the broker.
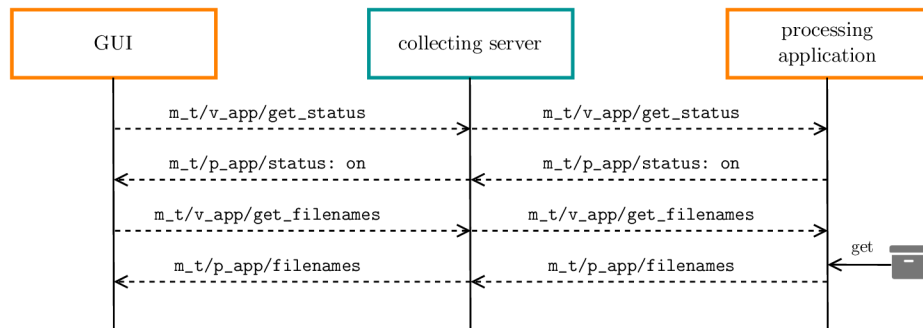

Figure 5.3: Sequence of messages transmitted after the GUI successfully connects to the broker, with the processing application in an active state.

Capturing. Triggering one of the sensing devices starts a chain of actions within the system. As illustrated by Figures 5.4 and 5.5, the device transmits a change of the capturing status. If toggled from inactive to active, the processing application is alerted to start collecting the incoming data measurements; if vice versa, the processing application closes the recorded collection, stores it within the storage and dispatches a message for the visualising application to update the list of filenames.
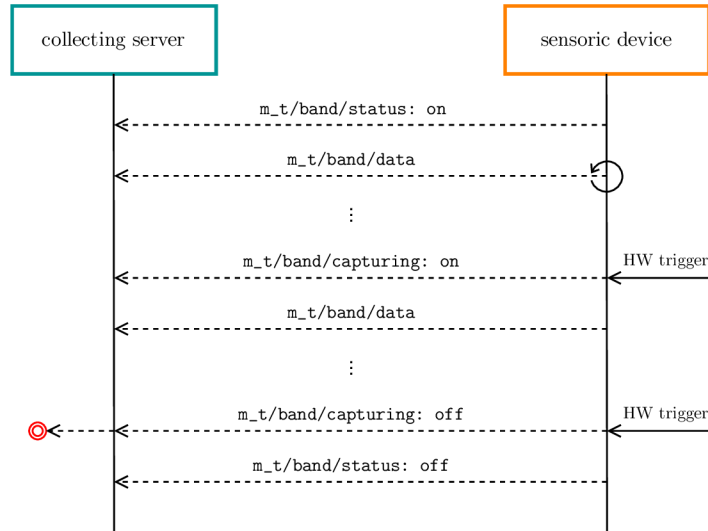

Figure 5.4: Sequence of messages covering the communication between the broker and a sensing device. Upon establishing the connection, the band makes an announcement about its connection status change and starts transmitting the motion data. The red circle serves as a reference – this message triggers events depicted in Figure 5.5.
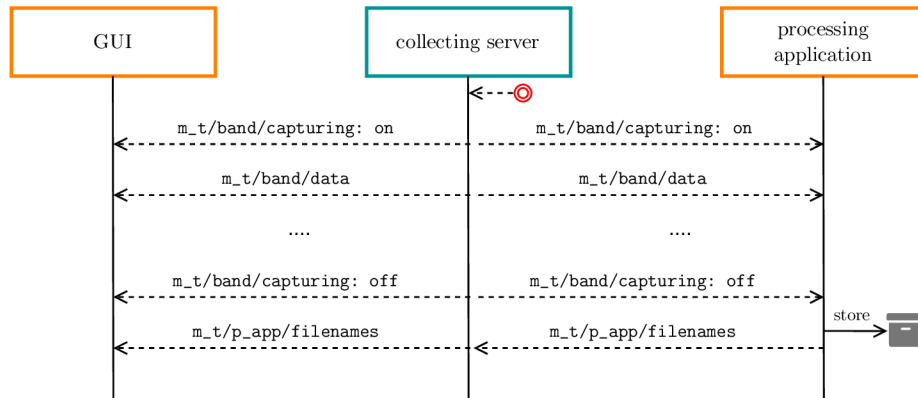

Figure 5.5: Triggered by finishing the capturing of motion data (depicted in Figure 5.4), the processing application stores the data in a file and transmits the filename to the GUI.

Replaying stored events. Provided that there are any existing pre-recorded measurement events, the user can trigger their replay through the GUI (5.6). The processing application responds with a sequence of messages: announcing the change in replaying status, transmitting the data itself, and closing the replaying session.
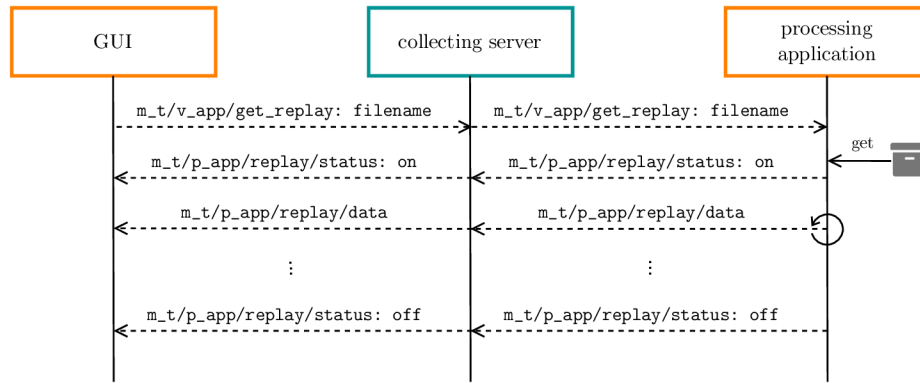
Figure 5.6: Sequence of messages covering the replaying functionality. After GUI requesting a file to be replayed, the processing application reads the file from the storage and re-transmits its contents.

Having introduced the main structure of the system, along with resolving how individual components are to communicate with each other, a presentation on the components themselves can be commenced.

5.2   Designing the application components

This section deals with further introducing the workflows the two application components are to follow, in order to reveal more information on the tasks they are intended to process.

5.2.1   Processing application and storage proposal

Figure B.1 depicts the design of the workflow the processing application is to follow. As illustrated, it is mainly built around facilitating sequences of messages designed in the previous section. The algorithm runs in a loop, listening to the incoming MQTT messages, whereas the original sender of the message is pivotal to the direction the flow is to take – i.e., to the left side of the flowchart, dedicated to the band data processing, or to the right, responding to the messages from GUI.

As for the storage itself, it was decided that the data shall be stored in actual separate files – i.e., one file per one capturing session – in the underlying filesystem. The other possible approach – although intended for more robust systems – would be using one of the time-series database technologies, such as InfluxDB[2], as the motion trajectories could potentially be treated as temporal sequences.

5.2.2   Graphical user interface design

For the sake of being cross-platform, the GUI shall be implemented as a web application. Traditionally, these applications are comprised of two separate parts, communicating using HTTP protocol: a backend server, carrying the computing load, and a frontend graphical interface. However, since the objective is to render data in real-time, it makes little sense for the stream to be first sent to the server, only to be delegated to the frontend. Therefore, it was decided that the communication responsibility is to be delegated to the client, thus

---

[2]https://www.influxdata.com/

reducing the backend to a single purpose only – serve the frontend application and its static files when requested by a web browser.

Akin to the processing application, the workflow the frontend is intended to follow – depicted in Figure B.2 – is a loop designed to process incoming MQTT messages. In contrast, whereas the processing application can only be interacted with through the MQTT interface, this application also listens to the user input. The draft for this user interface, illustrated in Figure 5.7, is designed as a simple dashboard, allowing the user to view the actual real-time orientation of the devices as 3D models.
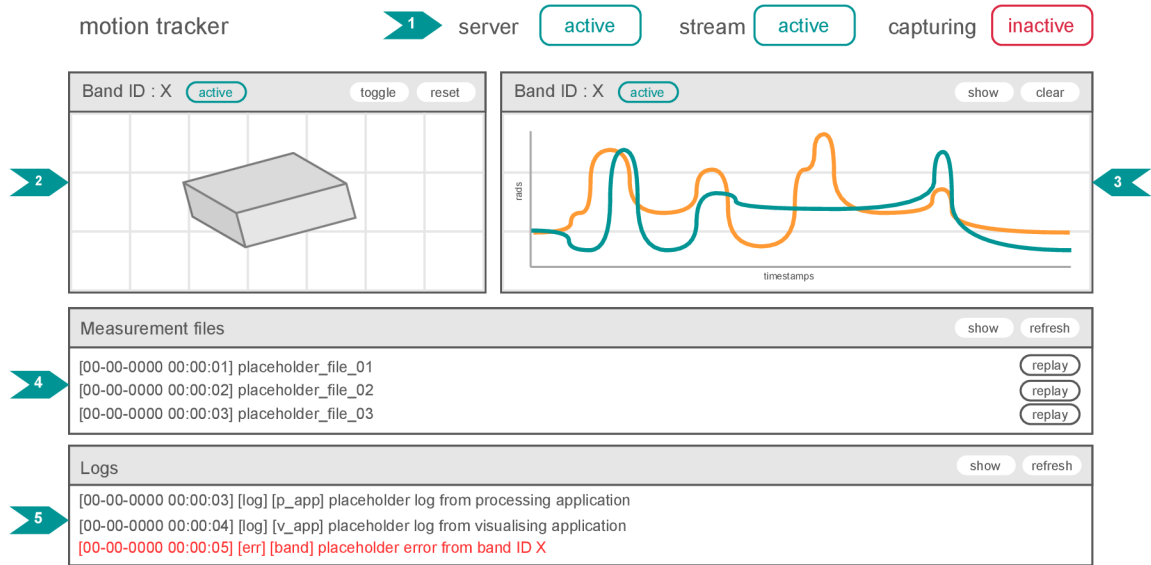

Figure 5.7: Draft of the initial design of the graphical user interface comprised of the following components: (1) status and control panel, (2) 3D model of the device mirroring its current orientation, (3) chart displaying orientation progress of a sensing device, (4) measurements panel for replay triggering and (5) logging panel for tracking the logs.

5.3   Sensing devices design

This section encompasses an all-in proposal for the ESP32/MPU-6050-based sensing devices, describing the hardware solution design (5.3.1) along with proposing an algorithm for the firmware realisation (5.3.2).

5.3.1   Hardware solution design

The following sections (referencing Sections 3.2 to 3.4) shall introduce the plans for interfacing ESP32 with MPU-6050 and a power source in terms of circuit and PCB design. This is commenced by presenting the powering and charging solution, along with the additional supporting components it requires, gradually building up to the communication between ESP32 and MPU-6050 itself. All diagrams for circuit and PCB design were assembled in EasyEda[3] online tool and a complete circuit diagram can be found in Appendix C.

---

[3]https://easyeda.com/

Battery and charging circuit

Not beating around the bush, it was decided that a rechargeable single-cell LiPo battery with a built-in protection circuit shall be used, as they come small-sized and lightweight. Li-ion and LiPo batteries are usually charged with the help of a specialized charging circuit – one of the most frequently used appears to be TP4056 [5]. Housed in 8-pin SOP package, it supports 8 V as the maximum voltage input while outputting a constant voltage of 4.2 V. It guards the battery from over-discharging (by cutting itself from the battery output in the event of the battery being discharged below 2.4 V) and overcharging, also offering a short-circuit protection. Figure 5.8 illustrates the circuit design for integrating TP4056 into the device – thought inspired by [5], some of the pins are left unused or grounded, as the functionality they had to offer was redundant. `PROG` pin is used for indicating the maximum amount of constant current the battery is to be charged at, as determined by the resistance of the resistor connected from this pin to ground (`R9`). A 2 kΩ resistor was selected, which, according to [5], results in a current of 0.58 A at 4.2 V – the reason for this choice shall be further explained in Section 5.3.1.
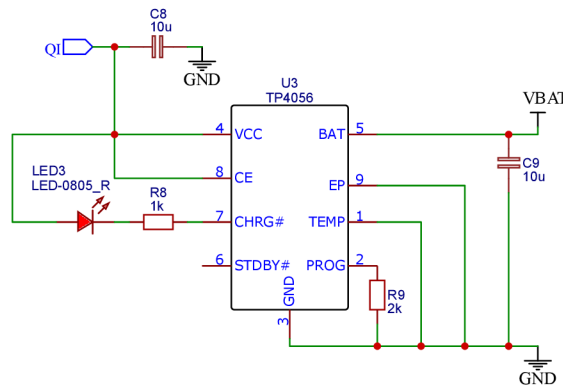

Figure 5.8: Diagram of TP4056 circuit application, inspired by [5]. `TEMP` pin, usually connected to the battery's thermistor output, is not used, therefore it is grounded. `STDBY` and `CHRG` pins indicate the state of the process of charging, finished or ongoing, respectively – it was decided to only indicate the process of charging, therefore `STDBY` is not connected. The `QI` label, acting as a power source, shall be further explained in Section 5.3.1, along with the value of resistor `R9`.

Wireless charging

As far as the interface between the charging circuit and the external power supply goes, a wireless power transfer was opted for, as it was found prudent to avoid dependency on a single connector type – this, however, requires the integration of a secondary coil and a Qi-compliant wireless power receiver with all supporting circuitry. It was decided to use CP2021 [2], a single-chip wireless receiver housed in a QFN-16 package, with built-in rectifier for AC-DC transformation and 5 V power output. As this chip often comes in the form of a stand-alone PCB module with all external components already included (along with a receiver coil), this variant was opted for rather than integrating the bare chip into the circuit. Revisiting Section 5.3.1, the `QI` label indicates that the power input of TP4056 originates from CP2021. As CP2021 outputs 2.5 W, the TP4056's output power should be similar – this was achieved by using a 2 kΩ resistor for `PROG` pin, as TP4056's constant voltage output is 4.2 V (4.2 V × 0.58 A = 2.436 W).

Powering ESP32 and MPU-6050

Having resolved what battery to use and how it is to be charged, let us now have a look at powering ESP32 and MPU-6050 themselves – as the LiPo batteries have a typical maximum voltage of 4.2 V [39], ESP32 runs at 3.3 V [19] and MPU-6050's typical operating range is between 2.375 V and 3.46 V [11], a single 3.3 V regulator should be sufficient for regulating the voltage for the whole circuit. It was opted to use MCP1702 [1] in a 3-Pin SOT-23A package. This is a CMOS low-dropout (LDO) regulator with a typical input voltage ranging between 2.7 and 13.2 V, also offering short-circuit, overcurrent and overtemperature protection – with 2.7 V, this regulator also serves as a overdischarge protection for the battery. The design for integration of MCP1702 is illustrated in Figure 5.9. In order for the user to be able to switch the device on and off at will, i.e., cut the regulator and the rest of the components from the battery source, a DPDT (*double pole double throw*, controlling two independent circuits) switch was added (S1)[4].
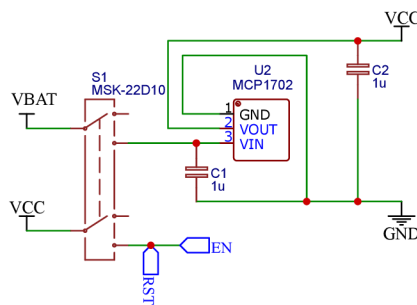

Figure 5.9: Diagram of MCP1702 circuit application, inspired by [1]. With only three pins (VIN, VOUT, GND) and two decoupling capacitors, integration of MCP1702 into any circuit is rather straightforward. The S1 switch shall serve as the main device switch, cutting the components from power source. Label EN indicates a connection to ESP32's EN pin, label RST shall be explained in 5.3.1.

ESP32 and MPU-6050 circuit integration

Having finalized the charging and powering solution, let us now have a look at the external components required by MPU-6050 and ESP32 – their design for circuit integration is illustrated by Figures 5.10 (inspired by Sparkfun breakboard design[5]) and 5.11. As these two devices are to communicate through an I²C interface (3.2.3), this is realized through labels MPU_SDA and MPU_SCL, connecting MPU-6050's SDA and SCL pins with ESP32's IO21 and IO22 respectively. Furthermore, utilizing hardware interrupts was opted for instead of polling (explained in 3.1), for letting ESP32 know when to read data from MPU-6050, therefore MPU-6050's INT pin is connected directly to ESP32's IO25 pin.

As for ESP32, it was decided that the WROOM module (introduced in 3.3.1) would be used, since it does most of the heavy lifting in terms of providing the external components necessary for the ESP32-D0WDQ6 chip to function properly. As described in Section 3.3.3, ESP32 modules lack a USB-to-UART bridge – therefore the external FTDI FT232R (Figure 5.12) would be used, as indicated by labels FTDI_RX and FTDI_TX connecting TX and RX pins on ESP32 and FTDI connector. The DTR and CTS pins are connected to

---

[4]http://www.hqdz123.com/en/product/toggleswitch/mini/7940.html

[5]https://cdn.sparkfun.com/datasheets/Sensors/IMU/Triple_Axis_Accelerometer-Gyro_Breakout_-_MPU-6050_v12.pdf)

ESP32's `EN` (through label `RST`, mentioned in 5.3.1) and `GPIO0` pins respectively, enabling an automatic flashing routine. In order for this to work, the `S1` switch (Figure 5.9) must be in the up position, cutting the `EN` pin from `VCC`.
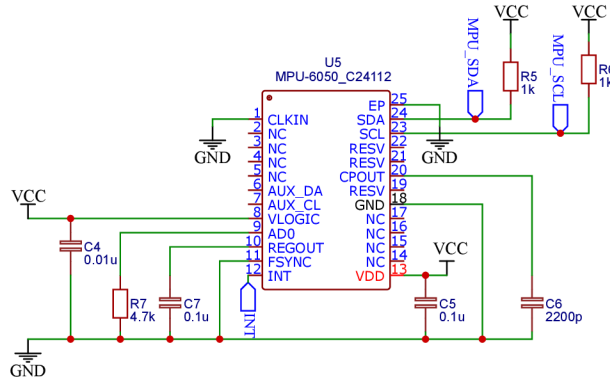

Figure 5.10: Diagram of MPU-6050 circuit application inspired by the Sparkfun breakout board design. Pins `AD0`, `AUX_DA` and `AUX_CL` are not used, it was opted to employ the device's default I²C address and no auxiliary sensors are used.
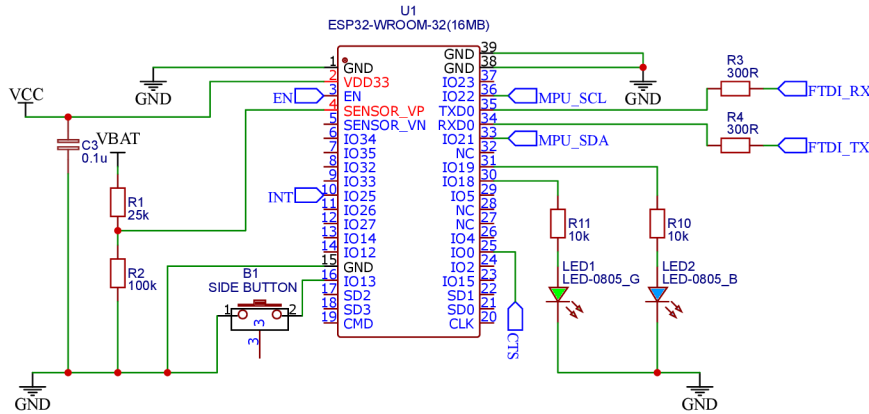

Figure 5.11: Diagram of ESP32 circuit integration. Resistor voltage divider (the resistor pair of `R1` and `R2`, connected in series) on the ADC pin `SENSOR_VP` is used for determining the battery level. The SMD side button (`B1`) serves as an interface for controlling MPU-6050, which is connected through I²C.
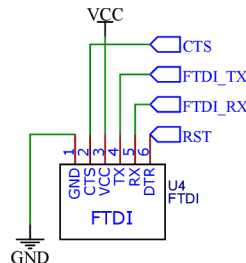

Figure 5.12: Diagram of standard FTDI FT232R connector layout, described in [22]. `CTS` and `DTR` flow control pins (connected to `GPIO0` and EN respectively) are generated in order to simulate conditions needed for driving ESP32 into the bootloader mode for flashing.

Printed circuit board design

First of all, making the device as small as possible in size was sought after, by determining its minimum possible dimensions regarding the components described in Sections 5.3.1 to 5.3.1. This was started by taking into account the dimensions of ESP32 WROOM module – at 25.5 mm in length, it is by far the largest component to be integrated. Having discovered this length limit, a Qi coil and a LiPo battery with dimensions as close to this as possible had to be found – a 28×28 mm coil was chosen, which, rounding up, brought the device to the minimum of 30×30 mm in size. As such, this also established the length and width of a PCB to be striven for. Figure 5.13 illustrates a list containing the dimensions of all components – based on this, it was endeavoured to determine the optimal placement for the components which need to be soldered onto the PCB, along with pads for connecting the external components (LiPo battery, COPO2021 and the coil), with respect to the best-practices of PCB designing. After designing the PCB itself, the external components had to be stacked underneath the PCB, to have the device as thin as possible – this is also why it was opted for to use simple pads instead of connectors such as JST. The final PCB design, based on the circuits displayed in the previous sections, is illustrated in Figure 5.14, whereas the result of the stacking endeavour, the "component hamburger", can be admired in Figure 5.15.

component (package/model)	length (mm)	width (mm)	height (mm)
ESP32 WROOM	25.5	18.0	3.0
LiPo battery (301525)	25.0	15.0	3.0
COPO2021	15.0	10.0	1.3
Qi coil	28.0	28.0	0.4
DPDT switch (MSK-22D10)	9.1	3.5	3.5
SMD button (TS-018)	4.7	2.4	1.9
MCP1702 (3-Pin SOT-23A)	3.0	2.5	1.1
TP4056 (SOP-8)	4.9	6.0	1.52
capacitors (0805)	2.0	1.25	0.60 - 1.25
resistors (0805)	2.0	1.25	0.50
leds (0805)	2.0	1.25	0.8 - 1.0

Figure 5.13: A list of all components and their dimensions comprising the sensing device.
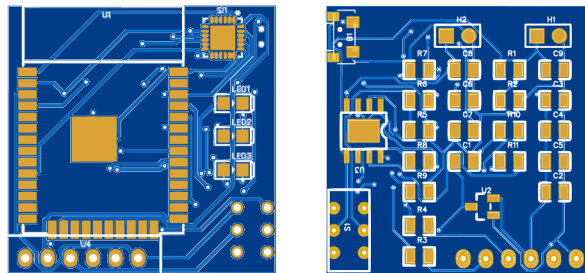

Figure 5.14: Final PCB design, with the top side (left) containing pads for ESP32 WROOM (`U1`), MPU-6050 (`U5`) and leds (`LED1` to `LED3`). The bottom side (right) contains pads for resistors (`R1` to `R11`) and capacitors (`C1` to `C9`), along with TP4056 charging circuit (`U3`), MCP1702 regulator (`U2`), FTDI connector (`U4`), battery (`H1`), COPO2021 (`H2`), THT switch (`S1`) and SMD button (`B1`).
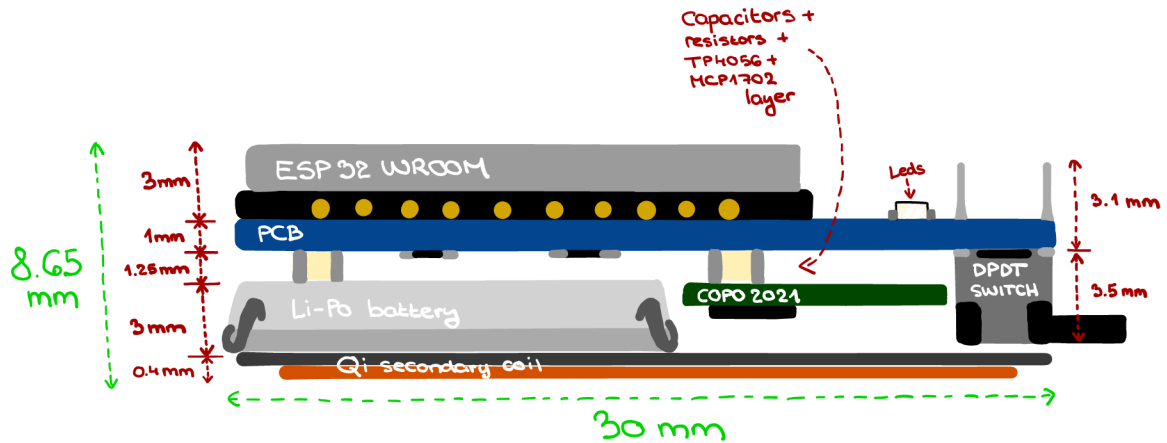
Figure 5.15: A sideways view of the final components arrangement. Soldered on top of the PCB, there is a layer containing ESP32, MPU-6050 and all three leds. Soldered on the bottom side, there is a layer containing all capacitors, resistors, TP4056, MCP1702 and both the SMD button and the THT switch. Stacked beneath this layer is the LiPo battery with COPO2021 PCB, with all of this covered by the secondary coil's ferrite sheet, as well as the coil itself. Altogether, this amounts to dimensions of circa $8.65 \times 30.0 \times 30.0\,\text{mm}$.

5.3.2 Firmware design

Having dealt with the hardware part, this section focuses on describing the firmware proposal. It begins with explaining the reasons behind the device synchronization, moving on to introducing the workflow itself.

Device synchronization design

As described in Section 5.1, triggering any of the devices is to commence the process of storing the motion data – however, prior to this, the devices need to be re-synchronised. In Section 4.1, the subject of synchronization in WSNs was briefly approached – to correctly interpret time-dependent data, timestamping has to be introduced into the system. Data collected by individual devices at the same point in time should be labeled by a timestamp of approximately the same value – with a large enough timestamp offset, measurements could be skewed irreparably.

Because of the capturing trigger, the synchronization strategy can be at-demand rather than automatic. Thus, it was decided that the bands are not going to be required to share any form of time scale, be it absolute or relative to each other; however, the RBS's "timestamping upon the time of arrival" served as an inspiration for the following scheme. Upon reception of the synchronization message, each device shall read and store the time elapsed since its initialization, provided by the ESP32's high resolution timer (3.3). Then, until another re-synchronization is triggered, every motion measurement's timestamp shall be calculated as the difference between the time of measurement creation and the stored time value. This makes for what one could possibly call an "event-relative" synchronization strategy.

Workflow proposal

Having established that a cooperation of devices in the form of synchronization must be incorporated into the firmware, this section proceeds with designing the firmware workflow algorithm itself. To further aid the description, a flowchart illustrating this algorithm is shown in Figure B.3. Taking into account previously written sections on the design of the sensing devices, four functionalities need to be configured within the initial setup: connection to Wi-Fi, initialization of the synchronization technology, connection to MQTT broker and MPU6050 initialization. To establish both Wi-Fi and MQTT communication, one has to provide the microcontroller with additional information such as SSID, broker address and credentials. Since hard-coding these into the firmware requires re-flashing a new version every time a credentials update is needed, it was opted to design a solution employing a Wi-Fi captive portal with ESP32 turning into an access point. As a result, sign-in information can be updated dynamically and every time a sign-in fails, this portal is activated, allowing the user to check the entered information.

After successful configuration, the main program loop is run. Every iteration starts with checking for synchronization packets – if received, the timestamping process is reset. Next, state of the SMD button is checked: if it is not pressed, the algorithm continues along the main branch. In contrast, if pressed, and it can be categorized as a long press, the device resets back to the setup mode – however, with a short press, device synchronization is activated. Once the devices have been synchronized and MQTT message announcing a change in the capturing status is sent to the broker, the flow is redirected back to the main branch. Next follows a DMP interrupt check – if there is an interrupt, i.e., MPU-6050 announces motion measurements were generated, these are read from the FIFO using the I^{2}C interface, parsed and sent in an MQTT data message.

Chapter 6

Implementation

Having finalized the system design in Chapter 5, the current chapter follows by introducing the implementation of the individual system components. Mirroring the presented workflow designs, the previously glanced-over details are now filled in with actual technologies utilized for realizing the components.

6.1 MQTT topic payloads

In Section 5.1.1, it was established that MQTT protocol shall be used for the inter-component communication. Since the only responsibility of the collecting server is to delegate MQTT messages, any existing implementation of an MQTT broker might be used – for the purposes of this thesis, the terminal `mosquitto` broker by Eclipse was utilized.

Following the design of the structure of MQTT topics from Section 5.1.1, this section introduces the actual structure of their payloads. It was decided that the JSON (JavaScript Object Notation) format would be used – it is both human-readable and machine-friendly and most of the modern programming languages support parsing of JSON files, either natively (JavaScript, Python) or with the aid of third-party libraries. These structures are listed in Listing 6.1 in their true form, as they appear within the implementations of MQTT clients of the components themselves.

```
1  // band – topics, v_app – GUI, p_app – processing
2  'm_t/band/status' : {'id': str, 'status': bool}
3  'm_t/band/capturing' : {'status': bool}
4  'm_t/band/data' : {'ts': str, 'id': str, 'ypr': [float], 'q': [float]}
5  'm_t/v_app/get_status' : ''
6  'm_t/v_app/get_filenames' : {'filter': str}
7  'm_t/v_app/get_replay' : {'filename': str}
8  'm_t/p_app/status' : {'status': bool}
9  'm_t/p_app/filenames' : {'filenames': [str]}
10 'm_t/p_app/replay/status' : {'status': bool}
11 'm_t/p_app/replay/data' : {'ts': str, 'id': str, 'ypr': [float], 'q': [float]}
```

Listing 6.1: Structures of payloads being carried within the individual MQTT topics – JSONs, usually in key-value pairs, though the actual values are substituted with their expected data types. Although simple **string** and **boolean** values dominate across the payloads, keys **ypr** (standing for yaw-pitch-roll) and **q** (quaternion) are both represented with lists of floats, whereas filenames contain a list of strings.

6.2 Implementing the applications

Mirroring Section 5.2, more details on the implementation of the tasks the application components are to process are revealed.

6.2.1 Implementation the processing application

This section references 5.2.1. The processing application was implemented in Python 3.7, using the `typing` module – due to introducing function and variable type annotations into the implementation, the services of `mypy` static type checker were regularly employed, thus minimizing the run-time error encounters. Inspired by the *Notch* motion capture system (mentioned in 2.2.2), the motion measurements are stored inside `.csv` files, parsed with the help of `Pandas` library.
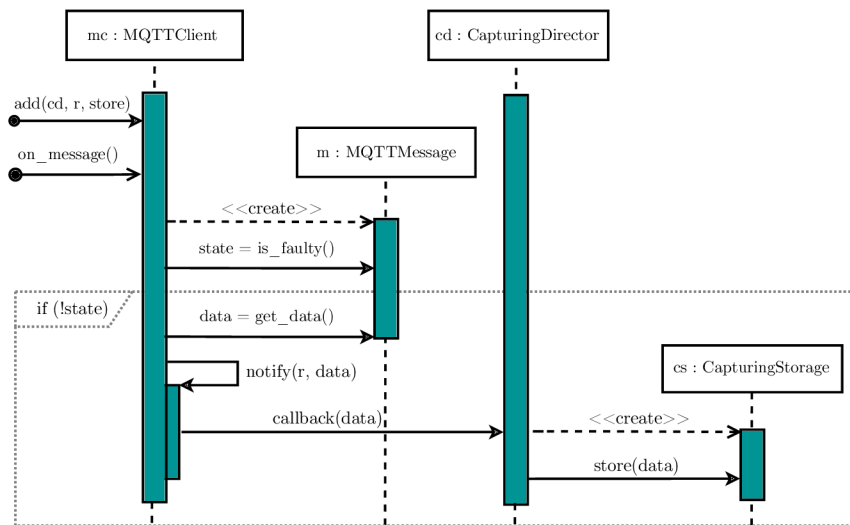

Figure 6.1: Sequence diagram illustrating registration of `CapturingDirector` instance's method `store` for subscription of event `r`. After the `on_message()` callback is evoked, `MQTTMessage` instance is created, parsing the incoming data. If not faulty, `cd` is notified and an instance of `CapturingStorage` is created, storing the message.

MQTT functionality is central to the whole application: living in `client.py` in the `mqtt` module, it is realized through the Eclipse Paho MQTT client library, implementing version 5.0 of the MQTT protocol. Since this client is asynchronous – i.e., processing and maintaining the network connection is performed in the background, whereas notifications of status changes and message receptions are provided to the application through employing callbacks – a custom Observer-like functionality is incorporated into this scheme, allowing for complete separation of the MQTT from the rest of the application. Although the `MQTTClient` serves as a simple Paho client wrapper – registering a callback for the `_on_message` method – it also acts as the Observer pattern's *subject*. Upon receiving a message and instantiating the `MQTTMessage` class, it delegates the notification to all its *observers*, by calling the previously specified callback methods (Figure 6.1). Through this interface, `CaptureDirector` class, contained within the `capturing` module, binds its methods to `MQTTClient` callbacks – these house the actual implementations for responding to the contents of the MQTT messages, allowing the processing application to execute the defined tasks. It uses the services of `CaptureStorage` class, which acts as a wrapper over the storage files.

40

6.2.2 Implementation of the graphical user interface

This section references 5.2.2. Since the plan was to delegate all communication and rendering responsibilities to the application frontend, the backend only needs to serve the HTML and JavaScript files – consequently, the server-side was implemented in Flask[1], a lightweight Python framework running on WSGI[2]. For the client-side, Vue.js [25, 31] was opted for – this is a Model-View-View-Model (MVVM) JavaScript framework specialised for building single-page applications. In Vue, each application is comprised of a hierarchy of nested, self-contained components, whereas each such component lives in its own `.vue` file, having its own separate state and a HTML-based template.
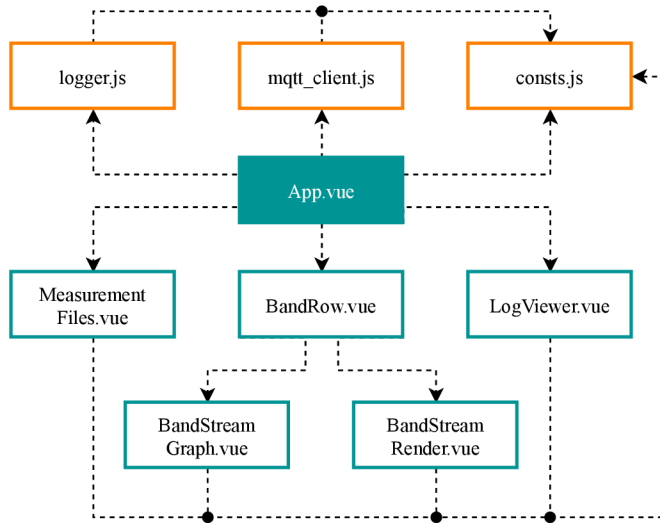

Figure 6.2: Scheme of Vue components and modules hierarchy found in the frontend implementation. These mirror the draft design of the user interface illustrated in Figure 5.7. `App.vue` is at the top level, whereas `MeasurementFile.vue` and `LogViewer.vue` encapsulate measurement and log panels, respectively. `BandRow.vue` serves as a wrapper over `BandStreamGraph.vue` (which employs Highcharts library for rendering the interactive graphs) and `BandStreamRender.vue`, which renders the 3D representation of measurements orientation using Three.js. Module `consts.js`, containing configuration data, is imported by each of these components, whereas `logger.js` and `mqtt.js` are injected into the application instance itself.

Following the graphical mock design, the application was split into six components, being serviced by three custom-made modules – Figure 6.2 illustrates the hierarchy of the components, as well as their dependency on the modules. For rendering the 3D device model and its animations, the services of the Three.js library based on WebGL were employed [28], whereas Highcharts[3], a SVG-based library, was used for rendering the interactive charts. Finally, since Vue.js modules need to be minimized to a single file along with all their dependencies prior to being deployed to the server, the Webpack[4] bundling tool was used. Final look of the graphical user interface is depicted in Appendix D.

---

[1] https://flask.palletsprojects.com/en/1.1.x/
[2] an interface forwarding requests from servers such as Apache to Python web applications
[3] https://www.highcharts.com/
[4] https://webpack.js.org/

Client-side MQTT functionality

Since Vue in version 3.0 was used – a relatively new release of the framework – it proved to be almost impossible to find compatible, ready-to-use modules facilitating MQTT functionality. Thus, a custom implementation was created, with the help of Eclipse' Paho library[5]. This library, akin to its Python-based counterpart, is also asynchronous – incorporating such behaviour into a Vue application is, however, not as straightforward, since these callbacks can only be registered once and the events need to be further delegated to multiple Vue components. There are several approaches to solving this, although it all comes down to choosing the correct location for the implementation, as well as the method for notifying the components upon change. For one, these callbacks can be registered at the top level component, with it passing the data to all its children. Passing could be done through `props` – this could, however, be considered an anti-pattern, since such realisation becomes unmaintainable with deeply nested components [31]. Should one wish to avoid implementing MQTT functionality at the top-level component altogether, this responsibility can be claimed by a dedicated `mixin`[6], passing the messages with the help of Vue's `event bus`, or even employing the services of Vuex[7].

```
1   //store/index.js
2   import { reactive, inject } from 'vue';
3   import { mqtt_client } from '@/js/mqtt_client'
4
5   //create a symbol primitive
6   const MQTTStateSymbol = Symbol('mqtt_client');
7   const createMQTTState = () => reactive({mqtt_client: mqtt_client});
8   const useMQTTState = () => inject(MQTTStateSymbol);
9
10  export { MQTTStateSymbol, createMQTTState, useMQTTState }
```

Listing 6.2: An abstract from the `store` module. Importing `mqtt_client`, this instance is registered to Vue's reactivity system using a symbol primitive. The `useMQTTState` function is defined, facilitating the injection of this symbol as a dependency.

Finally, it was decided that the same approach as with the processing application would be used. A custom `mqtt` module was created, containing a class with methods registered to MQTT callbacks, along with implementing its own Observer pattern. An instance of this class, called `mqtt_client`, is created and directly exported from the module. In the module `store`, this instance is imported and, using a `symbol` primitive [8], two functions are implemented: one enabling the injection of this symbol as a dependency, and one utilizing the symbol for binding the imported `mqtt_client` instance to Vue's `reactivity` system (Listing 6.2). Using the latter, the main application instance serves as a dependency provider (Listing 6.3), creating a common state. Using the first, this dependency is injected into the components, employing their `setup` options (executed before the components are even created) – thus, each component gains access to a single `mqtt_client` instance. Upon being mounted (inside the `mounted` lifecycle hook), each of the components then creates

---

[5]https://www.eclipse.org/paho/index.php?page=clients/js/index.php
[6]essentially a component without HTML template
[7]state management library for Vue applications, creating a global singleton containing a shared state
[8]an instance of the Symbol class, guarantying a return value to be unique

subscriptions to the MQTT-related events it needs to be notified of – usually corresponding to the reception of messages to specific topics – also specifying a callback method bound to the instance itself, which is to be run when the event is observed (Listing 6.4).

```
//main.js
import { createApp } from 'vue';
import App from '@/App.vue';
import { MQTTStateSymbol, createMQTTState } from '@/store';

const app = createApp(App);
app.provide(MQTTStateSymbol, createMQTTState()); //provide to app instance
app.mount('#app');
```

Listing 6.3: Importing symbol primitive along with the `createMQTTState` function from the custom `store` module, the main application instance provides the `mqtt_client` instance as a dependency.

```
//App.vue
import { useMQTTState } from '@/store';

export default {
    setup () {
        return {
            mqtt_client: useMQTTState().mqtt_client
        }
    },
    mounted() {
        this.mqtt_client.addMQTTListener(
            MQTT_TOPICS.MT_BAND_STATUS, //event for subscription
            this.updateBandStatus.bind(this) //bind to the component
        );
    },
}
```

Listing 6.4: Within the component option `setup`, the `useMQTTState` function is utilized, injecting the `mqtt_client` instance into the component. In `mounted`, this instance can be accessed through `this`, and the component can register its method to be called upon receiving a message to a specific topic.

6.3 Sensing device realisation

Although this section is based on the design proposed in Section 5.3, it also heavily leans on the information summarized in Chapters 3 and 4. Here, the process of transforming the PCB design into a fully functional, battery-powered device is described, followed by a presentation of the firmware implementation.

6.3.1 Hardware solution

This section references 5.3.1. Having generated a *gerber* file (an open vector format file containing information on each physical board layer of a PCB design) based on the PCB design created in EasyEDA, the double-sided PCBs were manufactured by JL-CPCB. After assessing the resulting boards, it was concluded that soldering the SMD components with a simple soldering iron – without using a paste – should be sufficient, as none of the packages appear small enough for this to pose an issue. This assumption was, however, proven incorrect while soldering MPU-6050's QFN-24 package with dimensions of $4 \times 4 \times 0.9\,\mathrm{mm}$ (further mentioned in Section 7.1.2) – luckily, the complications were far from severe and Figure 6.3 depicts a complete assembly of a sensing device's PCBs, with all SMD and THT components attached. Following up, the next step was to attach the $25 \times 15 \times 3\,\mathrm{mm}$ LiPo battery and CP2021 wireless receiver (stand-alone components can be found in Figure 6.4), to pads `H1` and `H2`, respectively, both located at the bottom side of the PCB. Stacking these underneath the PCB made for a real-live interpretation of the component "hamburger" (Figure 5.15). Finally, to protect the components, a simple $32 \times 32 \times 10\,\mathrm{mm}$ case was 3D-printed – final set-up of the device is shown in Figure 6.4.
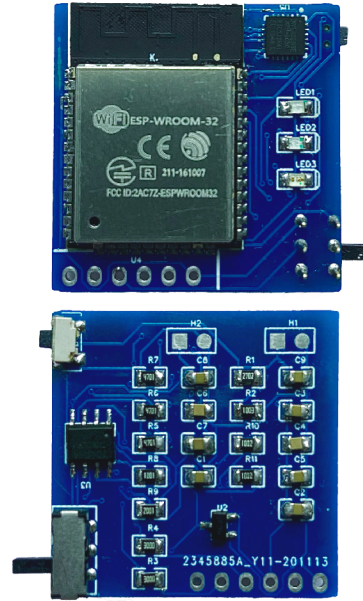

Figure 6.3: A final PCB assembly with all SMD and THT components soldered on, directly mirroring the PCB design illustrated by Figure 5.14.
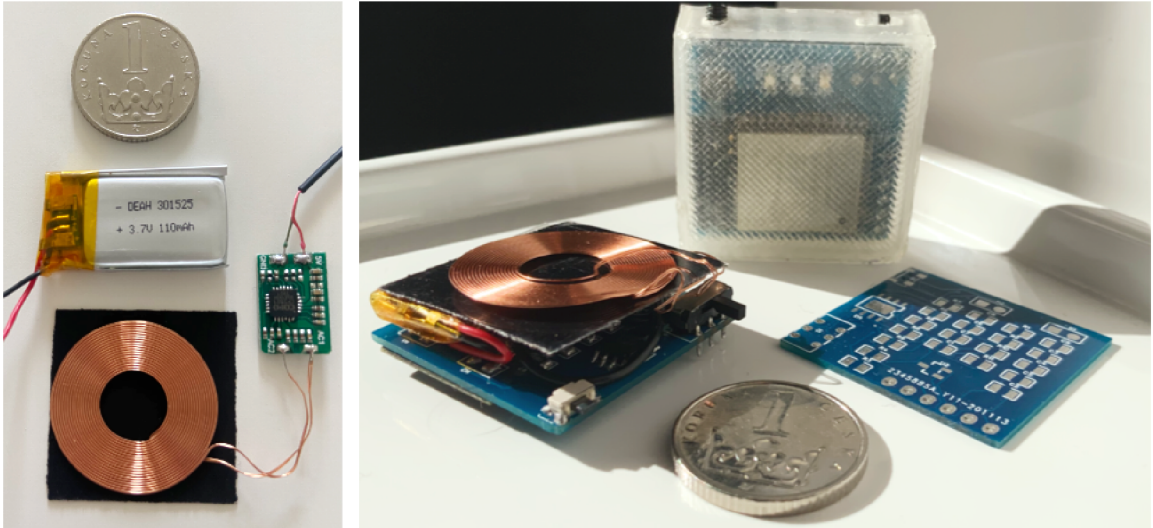

Figure 6.4: On the left there are the external components – copper wire coil attached to the CP2021 module and a LiPo battery. On the right, there is the sensing device realisation in various stages – a bare PCB (right), a component "hamburger" with LiPo battery and CP2021 attached, covered by the ferrite shield and the coil (left) and the fully-operational device contained within a 3D-printed box (top). The coins are for scale.

6.3.2 Firmware implementation

The description of the firmware, which was implemented in C++ using PlatformIO, is split into two parts: the first focuses on selecting an appropriate communication technology for sharing short-range synchronization messages among multiple sensing devices, whereas the second contains information on the implementation of Wi-Fi, MQTT and MPU-6050 functionalities described in the firmware design in Section 5.3.2.

Synchronizing multiple sensing devices

In order for the synchronization scheme proposed in Section 5.3.2 to function, the communication technology distributing the command must be predictable, fast and not, so to say, "go around" losing packets needlessly. In Section 4.2, four possible candidates for wireless communication with ESP32 modules were presented, although considering these requirements, ESP-NOW (4.2.3) is theoretically bound to outperform both MQTT (4.2.2) and UDP (4.2.1) protocols in terms of speed, due to running at a lower TCP/IP layer (MAC, in contrast to UDP's transport and MQTT's application layer), as well as facilitating a point-to-point connection. Thus, the decision falls between ESP-NOW and BLE (4.2.4). To resolve which of these to pick, it was decided that their performance is to be evaluated by conducting an experimental comparison, regarding an average round-trip time (RTT)[9] and mean time between failures (MTBF)[10]. The methodology for experimenting with each of the technologies is to be as follows: a minimal functioning sample, enabling communication between two ESP32 devices at a distance of two meters, shall be proposed, implemented and deployed. For each such implementation, 1010 packets carrying a sequence number are to be transmitted (intervals of 250 ms) and acknowledged. A truncated mean (discarding the 10 highest and 10 lowest latencies) of these RTTs is to be calculated, along with MTBF, noting the time between packets missing from the sequence. Finally, the results of the individual technologies shall be compared to each other. Given the above, we start by describing the proposed communication workflows for each of the technologies:

ESP-NOW sample. After the initial pairing of the master and slave nodes, the master broadcasts a message, noting the time. After receiving the message, the slave replies with an acknowledgement carrying the same number – upon receiving this acknowledgement packet, the master checks whether the received sequence number is equal to the expected number and notes the time; if the numbers do not match, a warning is printed.

BLE sample. Here the connection-oriented mode is used, employing the non-blocking asynchronous *notify* operation. A peripheral node, possessing a service with two distinct characteristics, advertises itself. When creating a connection to the peripheral node, the central node makes a subscription to be notified upon a change of a characteristic. Once the connection is established, the peripheral updates its characteristic's value and notes the time. Upon receiving the update notification, the central responds by writing to the second one of the peripheral's characteristics – after receiving this message, the same process as with ESP-NOW is repeated, i.e. checking the sequence number and noting the time.

---

[9]a truncated mean of times elapsed between transmitting a packet and receiving an acknowledgement
[10]predicted elapsed time between two failures, i.e. how often is a packet lost in this case

Results of runs with each of the technologies are shown by the box plot (with the x axis in logarithmic scale) in Figure 6.5 and summarized by the table in Figure 6.7. Although unexpected for both protocols, within the sample of 1010 packets containing a sequence number, each was acknowledged with the correct number – this means that no packet was lost, therefore there is no need for the MTBF metric to be calculated. In contrast, the difference between the average RTTs of these protocols is vast – at 3,564.93 µs, ESP-NOW clearly outperforms BLE with its 31,238.16 µs. As indicated by the upper quartiles, 75% of all ESP-NOW messages were acknowledged before the mark of 4,223 µs, as opposed to BLE's 37,199 µs. In short, ESP-NOW runs circles around BLE at the proposed setting and implementation – at their worst for both, two ESP-NOW round trips could be taken for a single BLE round trip to be finished.
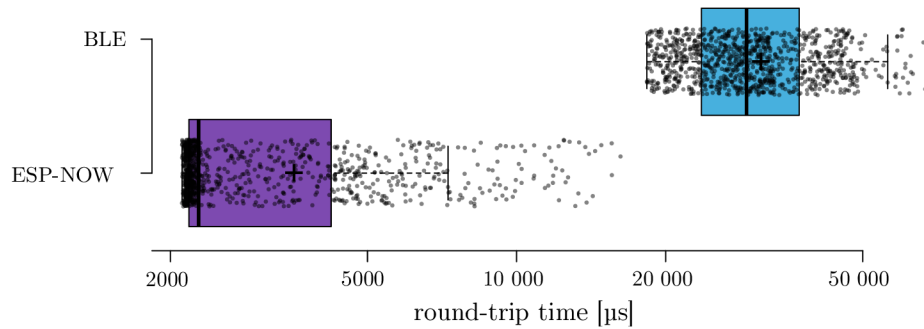

Figure 6.5: Box plot for comparison of 990 round-trip times (in microseconds, in logarithmic scale) for packets transmitted over ESP-NOW (purple) and BLE (blue). ESP-NOW outperforms BLE. The exact values of whiskers, quartiles and the median are listed in Figure 6.7.
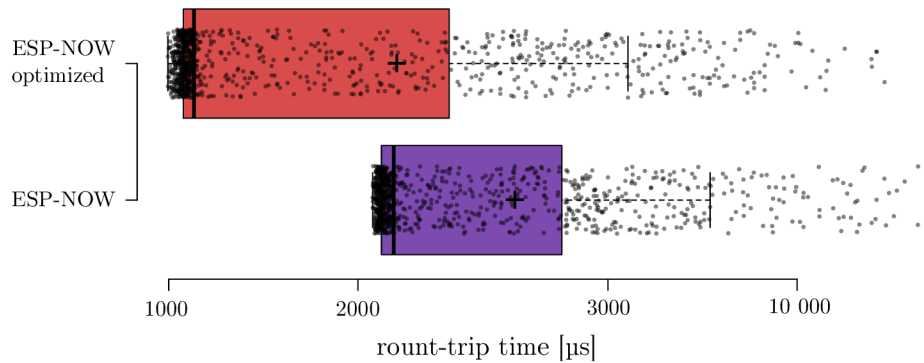

Figure 6.6: Box plot for comparison of 990 round-trips times (in microseconds, in logarithmic scale) for the original packets transmitted over ESP-NOW (purple) and optimized ESP-NOW (red). The exact values of whiskers, quartiles and the median are listed in Figure 6.7.

Initially, it was expected that BLE would become the technology of choice – however, after proving the contrary, ESP-NOW was studied even more extensively. Consequently, an issue opened in 2019 came up at the official ESP-IDF repository[11], theorizing that the ESP-NOW default setting for a bit rate of 1 Mbps can be bypassed by editing the data

---

[11] https://github.com/espressif/esp-idf/issues/3238

rate directly at the underlying Wi-Fi level. The suggested procedure was to enable the transmission of frames with a fixed bit rate (72.2 Mbps for 20 MHz, 150 Mbps for 40 MHz, rather than the default 1 Mbps) and disable AMPDU (Aggregated MAC Protocol Data Unit), which is responsible for aggregating multiple short MAC frames, naturally causing a delay. Even though this method has not yet been officially incorporated into the library, it is said to significantly reduce the ESP-NOW latency in data-transmissions – thus, look into how much of an improvement this fix might make was in order. Incorporating the suggested editions into the sample, the same procedure was followed as in the previous two experiments – even this time, there were no packets lost. As shown in Figure 6.6, when compared to the original ESP-NOW values, this optimization makes for even shorter RTTs, with the average being 2,312.31 µs – considering that this is a round-trip time, one could speculate that a single trip might be around staggering 1150 µs, probably getting close to what a wired connection might take. Since 75% of all RTTs were below the 2,764 µs mark, it was decided that these suggestions shall be incorporated into the final form of the ESP-NOW synchronization.

	BLE	ESP-NOW	ESP-NOW optimized
upper whisker	56,135.00	7,270.00	5,380.00
3rd quartile	37,199.00	4,223.00	2,794.00
median	29,127.00	2,280.50	1,097.00
1st quartile	23,608.00	2,180.00	1,055.00
lower whisker	18,327.00	2,111.00	997
lost packets	0	0	0
MTBF	—	—	—
average RTT	31,238.16	3,564.93	2,312.31

Figure 6.7: Table summarizing statistics for the plots in Figures 6.5 and 6.6, in s. The number of packets for each of the protocols used for the computations was 990. As there were no packets lost, MTBF is not calculated.

Implementing the firmware workflow

Following the flowchart, the first thing to be dealt with is the implementation of Wi-Fi captive portal and establishing Wi-Fi connection. This was a prerequisite for three other technologies, as they all require Wi-Fi communication – OTA, ESP-NOW and MQTT – and afterwards, MPU-6050 itself could be initialized. While the libraries used for the implementation of these functionalities shall be described in the paragraphs below, there is one more library, quite stand-alone from the firmware event chain, that needs to be mentioned – for distinguishing the length of a button press, related to the SMD button mapped to ESP32's `GPIO13`, the `OneButton` library[12] was employed.

Wi-Fi and captive portal. To this end, `ESP-WiFiSettings` library[13] was used, which facilitates an out-of-the-box captive portal functionality, along with storing multiple Wi-Fi credentials and dynamic custom parameters in ESP32's flash memory with the aid of `SPIFFS.h` library. Although the captive portal is programmed to start when credentials could not be read, or whenever the connection to Wi-Fi is lost, it can also be triggered

---

[12]https://github.com/mathertel/OneButton
[13]https://github.com/Juerd/ESP-WiFiSettings

artificially, through method `resetAndEnterConfigPortal()`. Figure 6.8 depicts the final implementation of the captive portal, using a personal cell phone.

Over The Air updates. Even though the FTDI FT232R's services were employed for initial firmware flashing, it was decided to also add the Over The Air (OTA) updates functionality. The OTA routines were incorporated using the `ArduinoOTA` library[14] – although not previously planned in design, enabling the upload of new firmware wirelessly, instead of having to connect through the USB-to-UART converter, is extremely practical.

ESP-NOW synchronization. Although the implementation of ESP-NOW synchronization was described in Section 6.3.2, this realisation was standalone, thus it had to be integrated into the the existing firmware. To initialize ESP-NOW, the Wi-Fi mode has to be configured to `WIFI_AP_STA`, having the device act as both an ESP-NOW access point and Wi-Fi station, simultaneously (Section 4.2.3). This is followed by commanding the device to enter the promiscuous mode and change Wi-Fi `AP` channel to match the `STA` channel, along with setting the internal bit rate, through function `esp_wifi_internal_set_fix_rate`, to a fixed predefined value (`WIFI_PHY_RATE_MCS7_SGI`, representing 72.2 Mbps for 20 MHz, 150 Mbps for 40 MHz). After this, ESP-NOW can be initialized using the `esp_now.h` header file included in ESP-IDF[15]. ESP-NOW handling within the main program loop requires checking for packets addressed to the broadcast MAC address, upon which re-synchronization of timestamping is executed.

Figure 6.8: Captive portal for entering Wi-Fi credentials, hosted on ESP32's webserver.

MQTT client. For implementation of the MQTT client, the services of the `PubSubClient` library[16] were employed, and, to avoid hard-coding MQTT configuration details, custom parameters for the MQTT broker address, port and optional sign-in credentials were added to the captive portal implementation.

MPU-6050 control. As the very last step in the setup part, I^2C a communication with MPU-6050 has to be established: MEMS sensors had to be re-calibrated and DMP initialized. Both of these functionalities, as well as all related data processings, were covered by

---

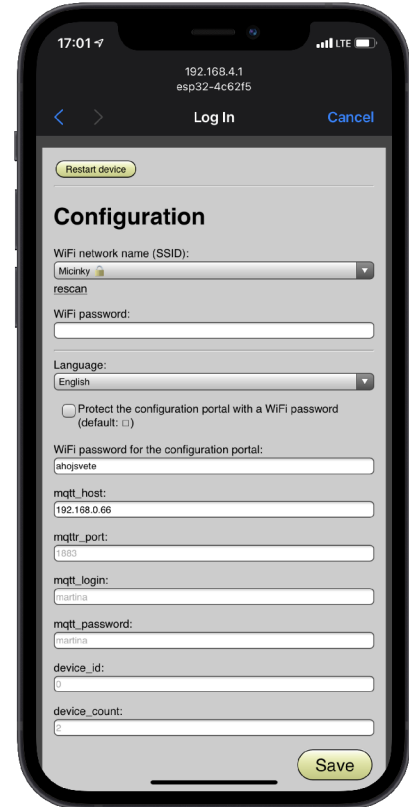[14]https://www.arduino.cc/reference/en/libraries/arduinoota/
[15]https://github.com/espressif/esp-idf
[16]https://www.arduinolibraries.info/libraries/pub-sub-client

the open-source ElectronicCats `MPU6050` library[17]. Prior to starting the library's offset calibration process (explained in Section 3.2.5), the initial offset values had to be determined – to this end, the procedure specified in `i2cdev` forums[18] was followed. The PCB was laid on a flat, horizontal surface and flashed with the suggested firmware, recording 1000 raw measurements for each of the sensors and calculating their mean values. These measurements were then incorporated into the final firmware implementation. Finishing the setup, in the main program loop, upon an interrupt on ESP32's `GPIO25`, measurements in a form of quaternions are read from MPU-6050's FIFO. Along with a corresponding timestamp, these are then formatted as JSON with the help of `ArduinoJson.h` and sent over MQTT for further processing.

---

[17]https://github.com/ElectronicCats/mpu6050
[18]https://www.i2cdevlib.com/forums/topic/96-arduino-sketch-to-automatically-calculate-mpu6050-offsets/

Chapter 7

Solution testing and evaluation

This chapter focuses on describing the procedures conducted for testing the functionality of the implemented mo-cap system: both manual and automated, continuous and post-development, component-based as well as system-wide. Creating some of these tests also inspired the addition of features previously unaccounted for in the initial design: the sensing device self-test procedure described in Section 7.1.2, MQTT-based logging mentioned in section 7.2, and the sequence fusion functionality presented in Section 7.2.2. The very end of the chapter is dedicated to summarizing the properties of the realized mo-cap system, proposing possible improvements, as well as presenting ideas for the future development.

7.1 Testing the system components

This section covers the implementation of automated tests used during the applications development, as well as the manual procedures for testing the hardware of sensing devices. The automated tests are all located within their own `tests` module.

7.1.1 Testing the applications

Since the whole system revolves around the components communicating using the MQTT protocol, the testing initiative was primarily focused on this functionality. The initial testing endeavours were achieved through Eclipse's `mosquitto` terminal clients, although this manual approach became tedious as the system started to grow. To decouple the individual components, a testing Python 3.7 module was built, containing a mock MQTT client class based on the Paho library. By itself, this client only provided the basic methods for connecting to the broker and receiving messages – in short, it was made to be extended with additional functionality.

Since the applications were being developed in parallel with the sensing devices, the very first automated testing endeavours were focused on implementing a device simulator capable of generating random motion measurements. The resulting script, based on the mock MQTT client class, is living in `bands_simulator.py` file. It is capable of generating data seemingly incoming from multiple devices using the `threading` library, also providing messages to start or stop the capturing process. This script handles continuous testing of both the processing application's storing capabilities, as well as the GUI's model rendering and chart plotting – consequently, it covers all applications' interactions reliant on the output of the sensing devices. This served as an inspiration for implementing analogous scripts simulating the applications themselves, thus further decoupling their development.

Two separate scripts were created, both extending the mock MQTT client's functionality: `processing_app_asserter.py`, testing the processing app's responds to the GUI requests, and `gui_asserter.py`, performing the same for the GUI. The first is comprised of a sequence of tests mirroring the app's flowchart shown in Appendix B.1, whereas each test depends on the assert result of its predecessor. Shown in the left part of Figure 7.1, the sequence might be compared to a finite-state machine with a sink error state: actions performed by the the script act as states and the application's responses serve as transitions. The same approach was taken for the GUI – apart from conducting manual and visual inspections of the interface, a sequence of tests following the same flow (shown in the right part of Figure 7.1) was implemented. In contrast to the processing app test, this sequence is commenced and controlled through simulating the user input using Selenium WebDriver bindings to Python[1].
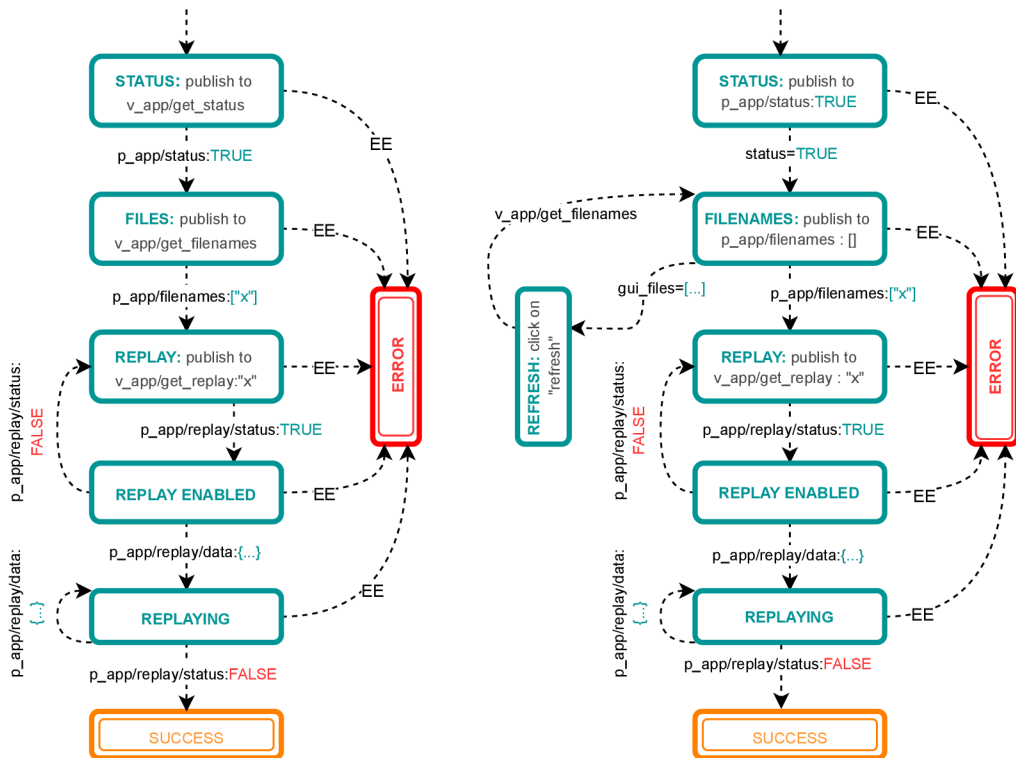

Figure 7.1: FSM-like diagrams illustrating the procession of automated tests for the processing application (left) and the graphical user interface (right).

7.1.2 Testing the sensing devices

Testing the sensing device included inspection of the actual hardware realisation, as well as firmware, mainly focusing on the MPU-6050 functionality.

Solder joins inspection

Having soldered every component onto the PCB, basic inspections were carried out, such as ruling out the presence of any obvious short circuits and cold solder joints, using a mul-

---

[1]https://pypi.org/project/selenium/

timeter in the continuity setting. Confirming that the soldering endeavours seem to be in order, a basic LED sample firmware was flashed into the device (a process mentioned in 3.3.3), using FTDI FT232R USB-to-UART converter, along with an additional power supply provided by a DC-DC converter (12 V step down to 3.3 V). With the LEDs blinking merrily (Figure 7.2), thus verifying that ESP32 is indeed working as intended, the basic firmware was modified to communicate with MPU-6050, initializing its DMP and printing the generated motion data to Serial Monitor. This phase required some cosmetic fixes, as at first MPU-6050 was not detected by ESP32 – since MPU-6050's pins are quite close to each other, applying too much tin could result in bridging, therefore being rather stingy with the solder did not pay off. Once all pins were properly soldered, MPU-6050 was detected by ESP32.
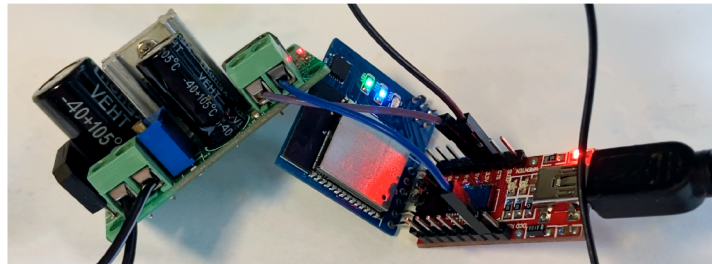

Figure 7.2: Flashing a simple LED blink firmware into the PCB in order to test whether ESP32 is working properly, using FTDI FT232R (right) with external power supply provided by DC-DC converter (left).

Charging solution inspection

Following up, the next step was to inspect the $25 \times 15 \times 3$ mm LiPo battery and CP2021 wireless receiver. To test whether the charging branch of the design is functional, the voltage of the battery cell was measured at first, with the readings noted. Afterwards, the device was put on a Qi-compliant wireless charger – the red LED on the top side of the PCB turned on and held, indicating that TP4056 circuit was working and charging (Figure 7.3). Letting a couple of minutes pass by, the device was removed from the charger and the battery's voltage was measured once more – it increased, therefore the charging branch of the hardware realisation was also deemed functional.
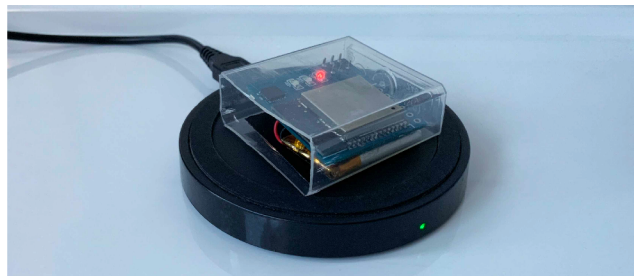

Figure 7.3: Testing the charging functionality of the sensing device, through having it charge using a Qi-compliant wireless transmitter. The green led on the wireless charger indicates that a wireless device was detected, whereas the red led on the sensing device, mapped to TP4056's `chrg` pin, indicates that the charging circuit is working and the battery is being charged.

MEMS sensor testing

To test the mechanical and electrical portions of MPU-6050's accelerometers and gyroscopes within MPU-6050 itself, self-test procedure (explained in Section 3.2.5) was executed. To this end, an extension of the firmware was implemented, inspired by Kris Winer's self-test example[2] – before initializing the DMP, the factory trim values are computed manually, and then the device's self-test registers (13-16) are used for triggering the simulation. Reading the self-test response, a percentage deviation from the factory trim values is calculated, following the formula specified in [12]. Since the results for each of the axes were within the tolerance range of 14% specified in [11], it was concluded that MPU-6050's internal accelerometer and gyroscope should be functioning as expected.

Device communication testing

As the last inspection, the final implementation of the firmware was flashed into two devices, to be able to test their communication capabilities, both with the MQTT broker and with each other. Turning the devices on with the THT switch, first the blue LEDs lighted up, indicating a successful connection to Wi-Fi. As the first band messages got delivered to the broker, communication with the rest of the system was validated. Thus, ESP-NOW synchronization was the last functionality to test – upon pushing the SMD button on one of the devices, both of their green LEDs lighted up, indicating ESP-NOW synchronization (and data capturing) was active, as illustrated by Figure 7.4. After this, the timestamps of the messages delegated to the broker from both devices got reset, marking the end of the device testing.

Figure 7.4: Two mo-cap bands positioned on a subject's arm. The blue LEDs indicate an active Wi-Fi connection, the green ESP-NOW synchronization.

7.2 Testing the system as a whole

This section describes testing of the system as a whole, focusing on monitoring the components interaction, as well as the overall motion capture capabilities.

7.2.1 Debugging the inter-component communication

While debugging the interaction messages between the individual components, a decision was made to aggregate all debugging logs at one place. For this purpose, a new MQTT topic

---

[2]https://github.com/kriswiner/MPU6050/blob/master/MPU6050BasicExample.ino

– `m_t/debug`, shown in Listing 7.1 – was established and shared across the components. Within a log message delegated to the MQTT broker under this topic, each component notes itself as a log source, also stating the timestamp, type of the log and the payload. Although with this setting it was possible to assign the log-aggregation responsibility to any of the components, the GUI seemed as the best candidate for their visualisation.

```
1  // single debug topic
2  'm_t/debug' : {'source': str, 'type': str, 'message': str, 'ts': str}
```

Listing 7.1: A new MQTT topic, shared by all components.

7.2.2 Experimenting with motion capture

This section describes the experiments conducted to test the system's overall motion capture functionality – each of these consisted of generating motion data with the help of sensing devices, storing them with the processing application and visualising the plots with the help of the GUI. Prior to presenting the experiment methodology itself, two preliminary things need to be explained: interpretation of the captured measurements and captured sequence fusion.

Interpretation of the captured measurements. Although the visualisation application uses quaternions for 3D renderings, within this section the angle rotations shall be expressed in the form of Cardanian angles, i.e., yaw-pitch-roll (explained in Section 2.2.2). Looking back to MPU-6050's orientation of the axes and polarity of rotation shown in Figure 3.1 in Section 3.2, as well as at the implementation for obtaining yaw-pitch-roll values by the ElectronicCats library[3], it was determined that yaw represents the rotation angle about the z axis, pitch about the y axis and roll about x axis. Although DMP generates measurements with the frequency of $100\,\text{Hz}$, the sensing devices are pre-set to delegate the data over MQTT with the frequency of $5\,\text{Hz}$ – this means that every timestamp on the plot line represents a step of $0.2\,\text{s}$.

Fusing the captured measurement sequences. As the results of a single run for each of the experiments cannot be considered representative of the device's capabilities, it was decided that each run shall be repeated at least five times, with the resulting sequences fused into a single profile. Consequently, a fusion method needs to be selected – here, the problem lies in the measurements being time-dependent, i.e. forming an ordered set of observations each recorded at a specific time – a *time series* [23]. Although the same hand motions shall be repeated, the individual stages can potentially be performed faster or slower within the repetitions. Authors of the article [53], dealing with a similar issue, experimented with two possible techniques (Figure 7.5): the first was a simple mean of values matched on common timestamps (also called *Euclidean* matching); the second, the one they determined to be more suitable, was matching the values to compute the mean of with the help of Dynamic Time Warping (DTW). DTW is a distance measure commonly used for finding the optimal alignment of sequences along with their similarity estimation, due to its ability to cope with time deformations [48, 46]. According to the authors of article [50], there are two ways of extending the DTW to the multidimensional space – they

---

[3] https://github.com/ElectronicCats/mpu6050/blob/master/src/MPU6050_6Axis_MotionApps_V6_12.h

coined these as *dependent* and *independent* warping. A decisive factor when opting between these two is the degree of dependency of the respective dimensions regarding their evolution in time. If the events recorded in the time series effect all dimensions simultaneously – i.e., they are *tightly-coupled* – it is recommended to employ the dependent variant. Since recording x and y axis accelerations of a pen's writing tip is set as a tightly-coupled example by the authors, it was decided that this method shall be used for the combination of the multivariate measurement sequences generated for the experiments. For this purpose, the services of the `tslearn` library[4] were employed, as the authors claim that it contains an implementation for dependent warping[5]. The alignment path of two sequences, computed through the means of `tslearn`'s `dtw_path` method, shall serve as a mapping for fusing the sequences into a single profile.
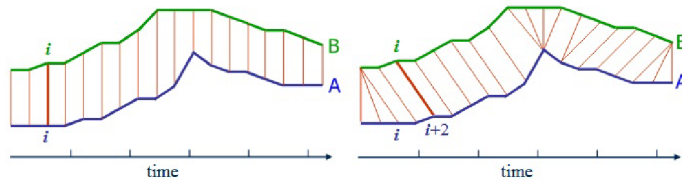

Figure 7.5: Comparison of Eudlidean matching (left) to DTW matching (right).

Without further ado, two types of experiments were conducted: the first (Experiment A) focused on the evolution of measurements with the device staying level, whereas the second type (Experiments B to D) inspected how the devices are affected in relation to rotation about each of the axes individually. The shared methodology for the latter was loosely inspired by [24] – originally, the authors of this article conducted these experiments to verify their approach to fusing the raw data of MPU9150's sensors, as they did not use its DMP. Likewise to the methodology, the ideal results for the latter-type experiments should also match. The angle of the inspected axis of rotation should start at the initial position of $0\,°$, rotating up to a degree dependent on the type of rotation ($\approx 60°$ to $90°$). At this point, the angle should stop increasing and start rotating in the opposing direction instead, reaching the same degree on the opposite side. Changing the direction once more, it should be rotated back to the initial rotation – all the while, the rest of the axes should stay at $0°$. However, due to the nature of human wrist movements, these angles are not going to be precise, and some degree of displacement on all axes is bound to occur. Consequently, what shall be evaluated with this type of experiments is how close the actual results come to the ideal case.

Experiment A: testing the device's drift

At first, the device's behaviour in a static position was tested – as MPU-6050 does not include a magnetometer, the expected behaviour for this IMU would be an ever-present drift in the yaw angle. In Section 3.2.5, it was mentioned that DMP incorporates patented auto-calibration algorithms, balancing the inherent drift of the MEMS sensors to some degree. Laying the device on an even surface, it was noted that this calibration process takes 26 seconds at an average (shown in Figure 7.6) of five runs – i.e., roughly after

---

[4]https://github.com/tslearn-team/tslearn
[5]https://github.com/tslearn-team/tslearn/issues/142

26 seconds, the readings seemingly stabilize, though this process might also take longer[6]. Although the pitch and roll angles truly hold their values, the drift in the yaw angle is still present, albeit minimized in comparison to the pre-balance readings.
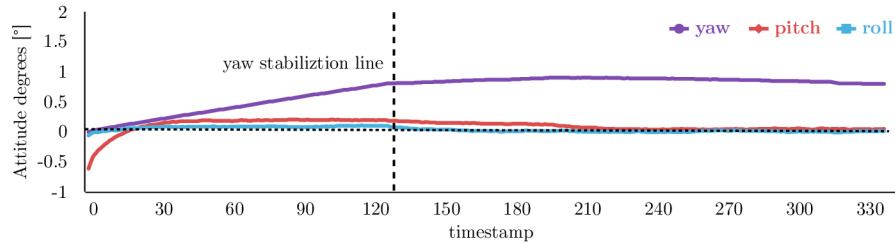

Figure 7.6: Plot showing the profile of five runs of angle measurements evolution generated from the very start of the device activation up to 330 timestamps, i.e. 66 s. The stabilization of the yaw angle appears to occur around the point of 26 s (130th timestamp).

Experiment B: testing the yaw rotation

The steps for testing the yaw rotation, also illustrated in Figure 7.7, were as follows:

Step 1: The arm with the band reaches forwards, with the palm facing down and fingers stretched; once level with the ground, motion capturing is to be triggered.

Step 2: Slowly, the hand is bent at the wrist inwards, pointing the fingers to the left, with the rest of the arm staying as static as possible, the palm still facing the ground.

Step 3: The wrist is bent outwards, passing the initial position, with the fingers pointing as far right as possible.

Step 4: The hand is returned to the original position and motion capturing is stopped.
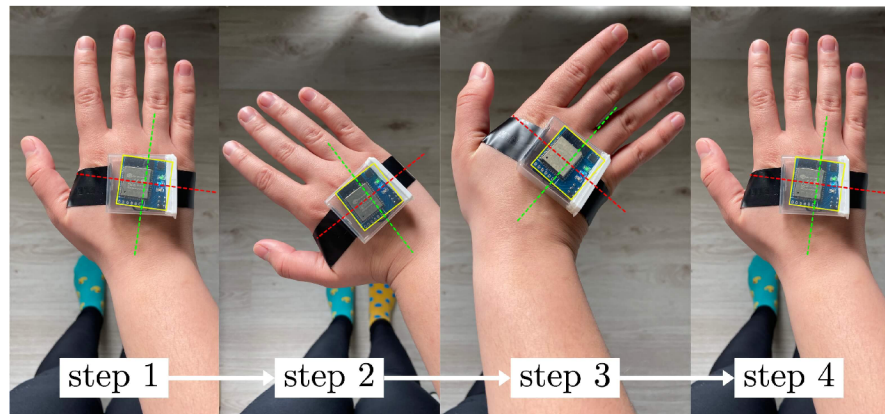

Figure 7.7: Approximate illustration of steps for the yaw experiment.

The resulting plot is shown in Figure 7.8. Although the curve of the inward bend in Step 2 seems to correspond with the expectations, i.e., it follows the $\approx 60\,°$ wrist bend depicted in

---

[6]https://wired.chillibasket.com/2015/01/calibrating-mpu6050

Figure 7.7, the same can no be said for Step 3, representing an outward bend. Here, the fault is put on the structure of a human hand – due to the muscles, bones, and tendons making up the human wrist, it was, naturally, impossible to bend the wrist to the same degree as with the inward turn. It also appears that, inadvertently, the arm was not completely static, allowing the wrist a slight diversion from the original pitch values – this is primarily visible in Step 3. Finally, the noise in Step 4 is most probably a result of an attempt to stop the motion capturing – the slight hand movements are caused by a force being applied to the SMD button, deviating the hand from the original position. Regardless of the outliers, it appears that the device's reactions to yaw rotation correspond with the expectations.
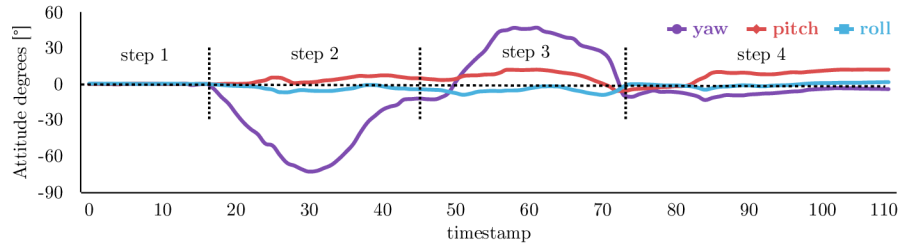

Figure 7.8: Sequence of orientations of a human hand. The steps within the plot are to correspond with the yaw experiment steps also shown in Figure 7.7.

Experiment C: testing the pitch rotation

The steps for testing the pitch rotation, also shown in Figure 7.7, were as follows:

Step 1: Identical to Step 1 of Experiment B.

Step 2: With the fingers still pointing forwards, the whole arm is rotated clockwise, so that the palm becomes perpendicular to the ground – the thumb is pointing up.

Step 3: The arm is rotated counterclockwise, passing the initial position and stopping with the palm once again perpendicular to the ground, the thumb pointing down.

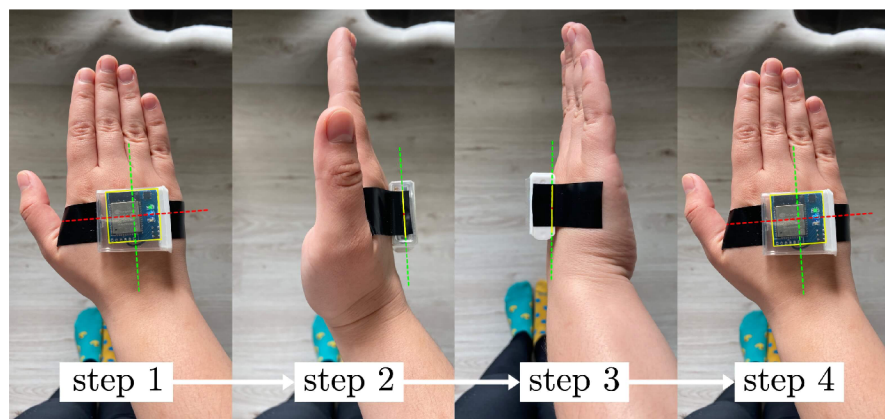**Step 4:** Identical to Step 4 of Experiment B.

Figure 7.9: Approximate illustration of steps for the pitch experiment.

The resulting plot is shown in Figure 7.9. This time, the yaw and roll axes seem almost unaffected, but for tiny tremors in Step 3 – rotating the wrist inwards requires the whole arm

to rotate, which might cause a slight diversion from the original position. All in all, as this rotation comes more naturally to the human hand, there are far less tremors present within the curves, compared to the previous experiment. Learning from the previous experience, the hand was steadied before motion capturing was turned off, which resulted in an almost ideal sequence of measurements in Step 4. Consequently, the course of the pitch rotation seems to have followed the rotations performed in Figure 7.10.
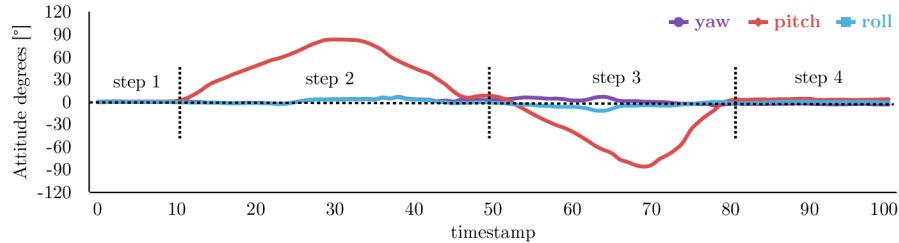

Figure 7.10: Sequence of orientations of a human hand. The steps within the plot are to correspond with the pitch experiment steps also shown in Figure 7.9.

Experiment D: testing the roll rotation

The steps for testing the roll rotation, also illustrated in Figure 7.11, were as follows:

Step 1: Identical to Step 1 of Experiments B and C.

Step 2: Slowly, the hand is bent downwards at the wrist, performing a wrist flexion with the fingers stretched towards the ground.

Step 3: The hand is bent upwards, performing a wrist extension – passing the initial position and moving on, stopping with the fingers pointing to the ceiling.

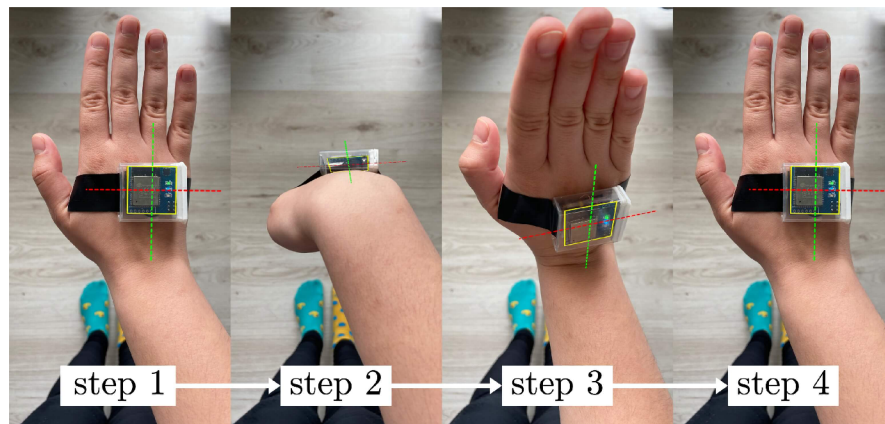**Step 4:** Identical to Step 4 of Experiments B and C.

Figure 7.11: Approximate illustration of steps for the roll experiment.

The plot of the corresponding measurements is shown in Figure 7.11. Since flexions and extensions seem, once again, like a more natural movement for the human wrist, the yaw and pitch axes are almost unaffected, but for Step 2 – it is suspected that during the wrist flexion step, the elbow joint was inadvertently being bent, thus allowing the roll angle to

58

pass 90° mark, as well as diverting the yaw and pitch angles. Nevertheless, the roll rotation seems to have followed the rotations performed in Figure 7.12 as well.
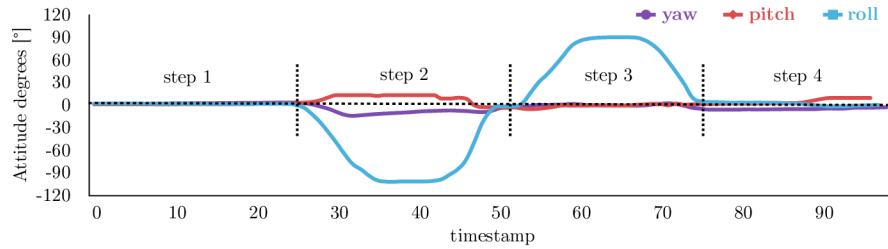

Figure 7.12: Sequence of orientations of a human hand. The steps within the plot are to correspond with the roll experiment steps also shown in Figure 7.11.

7.3 Solution evaluation, possible usage and improvements

To summarize, the previous chapters described the proposal for the inertial mo-cap system, as well as the processes required for converting those ideas into actual hardware and software realisations. Up until now, this chapter described the steps taken for verifying that the solution is indeed functioning as expected. Since the system was deemed functional based on the results of the conducted experiments, the following sections deal with the discoveries made during these testing endeavours, suggesting potential solutions for improvements, along with presenting the possible directions the future development of the system might take, building on the foundations created as part of this thesis.

7.3.1 Improving the measurements accuracy

Although partially improved through the DMP's patented motion recognition algorithms, yaw's drift remains an issue should the devices be required to function for extended periods of time. Consequently, such a case would call for incorporating a magnetometer into the device, configured as a slave device on MPU-6050's `AUX SDA/SCL` lines – however, according to [47], this approach requires some of the computations to be carried out on the host processor, as the MPU-6050's DMP lacks the computational power required for 9-axis fusion. A better approach would be to substitute MPU-6050 with MPU-9250[7], which contains a 3-axis digital compass – this would also solve the problem of the yaw's orientation being relative to the initial position of the device, rather than absolute akin to pitch and roll. The main issues with this motion tracking device are that it is three times more expensive than MPU-6050, and it also seems to lack the community's support in terms of the number of open-source third-party libraries working with its DMP.

Another approach, perhaps worth experimenting on, would be bypassing the DMP altogether, executing the whole process of accelerometer, gyroscope and magnetometer data fusion on the host processor. To this end, either Kalman or Magdwick AHRS filters could be employed, although a preliminary comparative study would probably be required. To further enhance the accuracy of the measurements, multiple motion tracking devices could be incorporated onto the PCB, with their results being averaged, thus also minimizing the impact of measurement outliers.

---

[7]https://invensense.tdk.com/products/motion-tracking/9-axis/mpu-9250/

7.3.2 Reducing the sensing device dimensions

Although already small, the device could be made even smaller, perhaps allowing for a battery of greater dimensions and capacity. Initially, the dimensions of the device were derived from the ESP32 WROOM module's length (Section 5.3.1) – however, this module encompasses many technologies redundant to the proposed mo-cap system's purposes. This could be remedied by utilizing a bare ESP32-D0WDQ6 chip, incorporating it, along with its minimal circuit scheme, straight into the device's custom PCB design. Furthermore, the most obvious optimization would be substituting the 0805 SMD packages used for the complementary components with smaller variants, such as 0603.

7.3.3 Extending the functionality

Although the processing application's functionality is now basic in nature, there are many ways in which it can be extended. The DTW-based sequence fusion, initially implemented for experimental purposes, could easily be incorporated into the application as a new feature for creating representative profiles of specific movements. Since DTW primarily determines a similarity measure of two sequences, a gesture recognition system could potentially be built on top of a database of such profiles. These classification tasks could further be improved by substituting the multivariate DTW method with QDTW – Quaternion Dynamic Time Warping, making use of the specific properties of the quaternion space [36] – thus utilizing the quaternions produced by DMP, instead of the processed yaw-pitch-roll values. It is not unusual for DTW to be employed in such tasks – for instance, the authors of article [51] experimented with combining these two methods, developing a two-level hierarchical classifier for tennis-shot recognition. Although simply listing a couple of new features, this suggests that there are many directions the future development of the proposed mo-cap system may take.

Chapter 8

Conclusion

The main objective of this thesis was to propose and construct foundations for a custom motion capture system, using the MPU-6050 inertial motion devices hosted by the ESP32 microcontrollers. Chapter 2 served as a primer on motion capture in general: introducing various motion capture techniques, their principles, merits and demerits. Chapter 3 presented the properties of ESP32 and MPU-6050 themselves, whereas Chapter 4 dealt with the communication technologies supported by ESP32. Drawing from this information base, Chapter 5 introduced the design of the custom mo-cap system – its components, their individual roles and responsibilities, as well as the interaction schemes they take part in. The process of transforming this proposal into an actual hardware and software realisation was described in Chapter 6 – mirroring the design, the implementations of each of the components were described separately, filling in the details about the technologies employed for this purpose. Finally, Chapter 7 focused on describing the procedures taken for testing and validating the mo-cap system: both continuous and post-development, on the component as well as the system level, using manual and automated procedures alike.

Having reviewed the structure of the thesis, let us now proceed with summarizing the developed solution. The constructed mo-cap system is comprised of four separate components: the sensing devices, the processing application, the graphical user interface and the MQTT broker facilitating all inter-component communication. The sensing bands are based on a custom PCB, powered by a tiny single-cell LiPo battery – due to this, they are reasonably lightweight, weighing $8\,\mathrm{g}$ at just $8.65 \times 30.0 \times 30.0\,\mathrm{mm}$ in size. To be unobtrusive, they were made completely wireless – communicating through MQTT, synchronizing through ESP-NOW, with Qi-compliant wireless charging facilitated through the CP2021 module with a receiver coil. The motion data generated by these devices are delegated to the MQTT broker, stored by the Python 3.7 processing application and visualised by the Vue.js graphical interface, using a 3D band model.

On the whole, the system was made to be modular and easily extendable with new features – as discussed in Chapter 7, one such feature, originally employed for experimenting purposes, was the integration of motion sequence fusion based on the multivariate Dynamic Time Warping method, for creating representative profiles of the captured, time-dependent motion series. Although there are many possible directions for future development of the implemented mo-cap system, this functionality, along with the ability to store and replay the captured movements, might be utilized for laying the foundations for a gesture recognition system, which seems to be the most engaging outcome.

Bibliography

[1] *250 mA Low Quiescent Current LDO Regulator.* Microchip. Available at: https://ww1.microchip.com/downloads/en/DeviceDoc/22008E.pdf.

[2] *CP2021 2.5W Qi SoC.* COPO Microelectronics (NanJing) Co., LTD. Available at: http://co-po.cn/index.php?a=cms&b=index&c=news&cid=53&id=47.

[3] *Getting Started with ESP8266* [online]. [cit. 2020-30-11]. Available at: esp8266.com/wiki/doku.php?id=getting-started-with-the-esp8266.

[4] *Li-Ion and LiPoly Batteries* [online]. [cit. 2020-28-12]. Available at: https://learn.adafruit.com/li-ion-and-lipoly-batteries/voltages.

[5] *TP4056 1A Standalone Linear Li-lon Battery Charger with Thermal Regulation in SOP-8.* NanJing Top Power ASIC Corp. Available at: https://dlnmh9ip6v2uc.cloudfront.net/datasheets/Prototyping/TP4056.pdf.

[6] *User Datagram Protocol.* 1980. Available at: https://tools.ietf.org/html/rfc768.

[7] *AN10216-01 I2C manual.* Philips Semiconductors, March 2003. Available at: https://www.nxp.com/docs/en/application-note/AN10216.pdf.

[8] *Bluetooth Core Specification.* 4.0. Bluetooth SIG (Special Interest Group), December 2009. Available at: https://www.bluetooth.com/specifications/archived-specifications.

[9] *Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.* IEEE Computer Society, 2012. Available at: https://mrncciew.files.wordpress.com/2014/10/ieee-802-11-2012.pdf.

[10] *Bluetooth Core Specification.* 4.2. Bluetooth SIG (Special Interest Group), January 2013. Available at: https://www.bluetooth.com/specifications/archived-specifications/.

[11] *MPU-6000 and MPU-6050 Product Specification.* 3.4. InvenSense, August 2013. Available at: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf.

[12] *MPU-6000 and MPU-6050 Register Map and Descriptions.* 4.2. InvenSense, August 2013. Available at: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf.

[13] *MQTT For Sensor Networks (MQTT-SN) Protocol Specification.* 1.2. IBM, November 2013. Available at: https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf.

[14] *MPU Hardware Offset Registers Application Note.* 1.0. InvenSense, February 2014. Available at: https://invensense.tdk.com/developers/download/emd-6-12/?wpdmdl=45.

[15] *ESP-NOW User Guide.* 1.0. Espressif, 2016. Available at: https://www.espressif.com/sites/default/files/documentation/esp-now_user_guide_en.pdf.

[16] *Introduction to the Power Class 0 Specification.* 1.2.3. Wireless Power Consortium, February 2017. Available at: https://faculty-web.msoe.edu/johnsontimoj/EE4980/files4980/Qi-PC0-introduction-v1.2.3.pdf.

[17] *MQTT Specification.* 5.0. OASIS, March 2019. Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdf.

[18] *ESP-IDF Programming Guide.* 4.1. Espressif, 2020. Available at: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html.

[19] *ESP32 Series.* 3.4. Espressif Systems, November 2020. Available at: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[20] *ESP32 Technical Reference Manual.* 4.3. Espressif Systems, 2020. Available at: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

[21] *ESP32-WROOM-32.* 3.4. Espressif Systems, November 2020. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf.

[22] *FT232R USB UART IC Datasheet.* 2.16. Future Technology Devices International Ltd., May 2020. Available at: https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf.

[23] BROCKWELL, P. and DAVIS, R. *Time Series: Theory and Methods.* Springer New York, 2013. Springer Series in Statistics. ISBN 9781489900043. Available at: https://books.google.cz/books?id=DJ_lBwAAQBAJ.

[24] CHEN, P.-z., LI, J., LUO, M. and ZHU, N.-h. Real-Time Human Motion Capture Driven by a Wireless Sensor Network. *International Journal of Computer Games Technology.* Hindawi Publishing Corporation. February 2015, vol. 2015, p. 695874. DOI: 10.1155/2015/695874. ISSN 1687-7047. Available at: https://doi.org/10.1155/2015/695874.

[25] COPES, F. *Vue.js Design Patterns and Best Practices.* 2018. Available at: https://vuehandbook.com/?ref=madewithvuejs.com.

[26] DARROUDI, S. and GOMEZ, C. Bluetooth Low Energy Mesh Networks: A Survey. *Sensors.* June 2017, vol. 17. DOI: 10.3390/s17071467.

[27] DEWAN, A., CASELITZ, T., TIPALDI, G. D. and BURGARD, W. Motion-based detection and tracking in 3D LiDAR scans. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, p. 4508–4513. DOI: 10.1109/ICRA.2016.7487649.

[28] DIRKSEN, J. *Learn Three.js - Third Edition*. Packt, 2018. 528 p. ISBN 9781788833288. Available at: https://www.packtpub.com/product/learn-three-js-third-edition/9781788833288.

[29] FILIPPESCHI, A., SCHMITZ, N., MIEZAL, M., BLESER, G., RUFFALDI, E. et al. Survey of Motion Tracking Methods Based on Inertial Sensors: A Focus on Upper Limb Human Motion. *Sensors (Basel, Switzerland)*. MDPI. Jun 2017, vol. 17, no. 6, p. 1257. DOI: 10.3390/s17061257. ISSN 1424-8220. 28587178[pmid]. Available at: https://pubmed.ncbi.nlm.nih.gov/28587178.

[30] FURNISS, M. *Motion capture* [online]. [cit. 2020-14-11]. Available at: http://web.mit.edu/comm-forum/legacy/papers/furniss.html.

[31] HALLIDAY, P. *Vue.js Design Patterns and Best Practices*. Packt, 2018. ISBN 9781788839792. Available at: https://www.packtpub.com/free-ebook/vuejs-2-design-patterns-and-best-practices/9781788839792.

[32] HERZIG, I. *Bewegungsfelder Inertial Motion Capture* [online]. [cit. 2020-22-11]. Available at: http://herrzig.ch/work/bewegungsfelder/.

[33] HILLAR, G. C. *MQTT Essentials - A Lightweight IoT Protocol*. Packt, 2017. 280 p. ISBN 9781787287815. Available at: https://www.packtpub.com/product/mqtt-essentials-a-lightweight-iot-protocol/9781787287815.

[34] HILMERSSON, K. and GUMMESSON, F. Time Synchronization in Short Range Wireless Networks. *Department of Electrical and Information Technology*. 2016.

[35] HUNTER, G., STETTER, J., HESKETH, P. and LIU, C. C. Smart Sensor Systems. *Nanodevices and Nanomaterials for Ecological Security*. January 2012, vol. 20.

[36] JABLONSKI, B. Quaternion Dynamic Time Warping. *IEEE Transactions on Signal Processing*. 2012, vol. 60, no. 3, p. 1174–1183. DOI: 10.1109/TSP.2011.2177832.

[37] KEMPE, V. *Inertial MEMS: Principles and Practice*. Cambridge University Press, 2011. ISBN 9781139494823. Available at: https://books.google.cz/books?id=XzdvDGbLZ8EC.

[38] LAI, M. *ESP32 Boot Mode Selection* [online]. [cit. 2020-13-01]. Available at: https://github.com/espressif/esptool/wiki/ESP32-Boot-Mode-Selection.

[39] LIANG, Y., ZHAO, C.-Z., YUAN, H., CHEN, Y., ZHANG, W. et al. A review of rechargeable batteries for portable electronic devices. *InfoMat*. 2019, vol. 1, no. 1, p. 6–32. DOI: https://doi.org/10.1002/inf2.12000. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/inf2.12000.

[40] LLC, C. *CH Robotics content library* [online]. [cit. 2020-25-11]. Available at: http://www.chrobotics.com/library.

[41] MA, Y., ANDERSON, CROUCH and SHAN, J. Moving Object Detection and Tracking with Doppler LiDAR. *Remote Sensing.* may 2019, vol. 11, p. 1154. DOI: 10.3390/rs11101154.

[42] MENACHE, A. *Understanding Motion Capture for Computer Animation and Video Games.* Morgan Kaufmann, 2000. ISBN 9780124906303. Available at: https://books.google.sk/books?id=9njZ482OYfwC.

[43] MILOSEVIC, B., LEARDINI, A. and FARELLA, E. Kinect and wearable inertial sensors for motor rehabilitation programs at home: state of the art and an experimental comparison. *BioMedical Engineering OnLine.* Apr 2020, vol. 19, no. 1, p. 25. DOI: 10.1186/s12938-020-00762-7. ISSN 1475-925X. Available at: https://doi.org/10.1186/s12938-020-00762-7.

[44] MORRIS, N. M. Interrupts and Polling. In: *Microprocessor and Microcomputer Technology.* London: Macmillan Education UK, 1981, p. 173–186. ISBN 978-1-349-16651-0. Available at: https://doi.org/10.1007/978-1-349-16651-0_9.

[45] NIHTIANOV, S. and LUQUE, A. *Smart Sensors and MEMS: Intelligent Sensing Devices and Microsystems for Industrial Applications.* Elsevier Science, 2018. Woodhead Publishing Series in Electronic and Optical Materials. ISBN 97 80081020562. Available at: https://books.google.cz/books?id=-YI2DwAAQBAJ.

[46] RATANAMAHATANA, C. and KEOGH, E. Everything you know about Dynamic Time Warping is wrong. In:. January 2004.

[47] ROWBERG, J. *MPU-6050 6-axis accelerometer/gyroscope* [online]. [cit. 2020-21-12]. Available at: http://www.i2cdevlib.com/devices/mpu6050#source.

[48] SENIN, P. Dynamic Time Warping Algorithm Review. January 2009.

[49] SHI, G., WANG, Y. and LI, S. Human Motion Capture System and its Sensor Analysis. *Sensors and Transducers.* June 2014, vol. 172, p. 206–212.

[50] SHOKOOHI YEKTA, M., HU, B., JIN, H., WANG, J. and KEOGH, E. Generalizing DTW to the multi-dimensional case requires an adaptive approach. *Data Mining and Knowledge Discovery.* february 2016, vol. 31. DOI: 10.1007/s10618-016-0455-0.

[51] SRIVASTAVA, R. and SINHA, P. Hand Movements and Gestures Characterization Using Quaternion Dynamic Time Warping Technique. *IEEE Sensors Journal.* january 2015, vol. 16, p. 1–1. DOI: 10.1109/JSEN.2015.2482759.

[52] STENGEL, M. *DIY Project - Wearable IMU tracking sensor* [online]. [cit. 2020-22-11]. Available at: https://inmagicwetrust.wordpress.com/2015/11/04/diy-project-wearable-imu-tracking-sensor/.

[53] VAUGHAN, N. and GABRYS, B. Comparing and Combining Time Series Trajectories Using Dynamic Time Warping. *Procedia Computer Science.* december 2016, vol. 96, p. 465–474. DOI: 10.1016/j.procs.2016.08.106.

[54] WELCH, G. and FOXLIN, E. Motion Tracking: No Silver Bullet, but a Respectable Arsenal. *Computer Graphics and Applications, IEEE.* December 2002, vol. 22, p. 24 – 38. DOI: 10.1109/MCG.2002.1046626.

[55] WHITE, M. *The MPU6050 Explained* [online]. [cit. 2020-21-12]. Available at: https://mjwhite8119.github.io/Robots/mpu6050.

[56] YAHYA, M., SHAH, J., KADIR, K., YUSOF, Z., KHAN, S. et al. Motion capture sensing techniques used in human upper limb motion: a review. *Sensor Review*. June 2019, vol. 39. DOI: 10.1108/SR-10-2018-0270.

Appendix A

CD contents

```
CD
├── demonstration
│   ├── HD_video_links.txt   - links to the videos in HD quality
│   ├── videos               - video demonstrations of the solution
│   └── images               - screens and photos of the solution
├── docs
│   ├── circuits             - the sensing device circuit schematic
│   ├── flowcharts           - design flowcharts of the system components
│   └── thesis               - thesis text and its source codes
├── README.md                - map of the CD contents
└── source
    ├── firmware             - source codes for the firmware
    ├── processing_app       - source codes for the processing app
    ├── visualising_app
    │   ├── server           - source codes for the backend
    │   └── client           - source codes for the frontend
    └── tests                - source codes for the automated tests
```

Appendix B
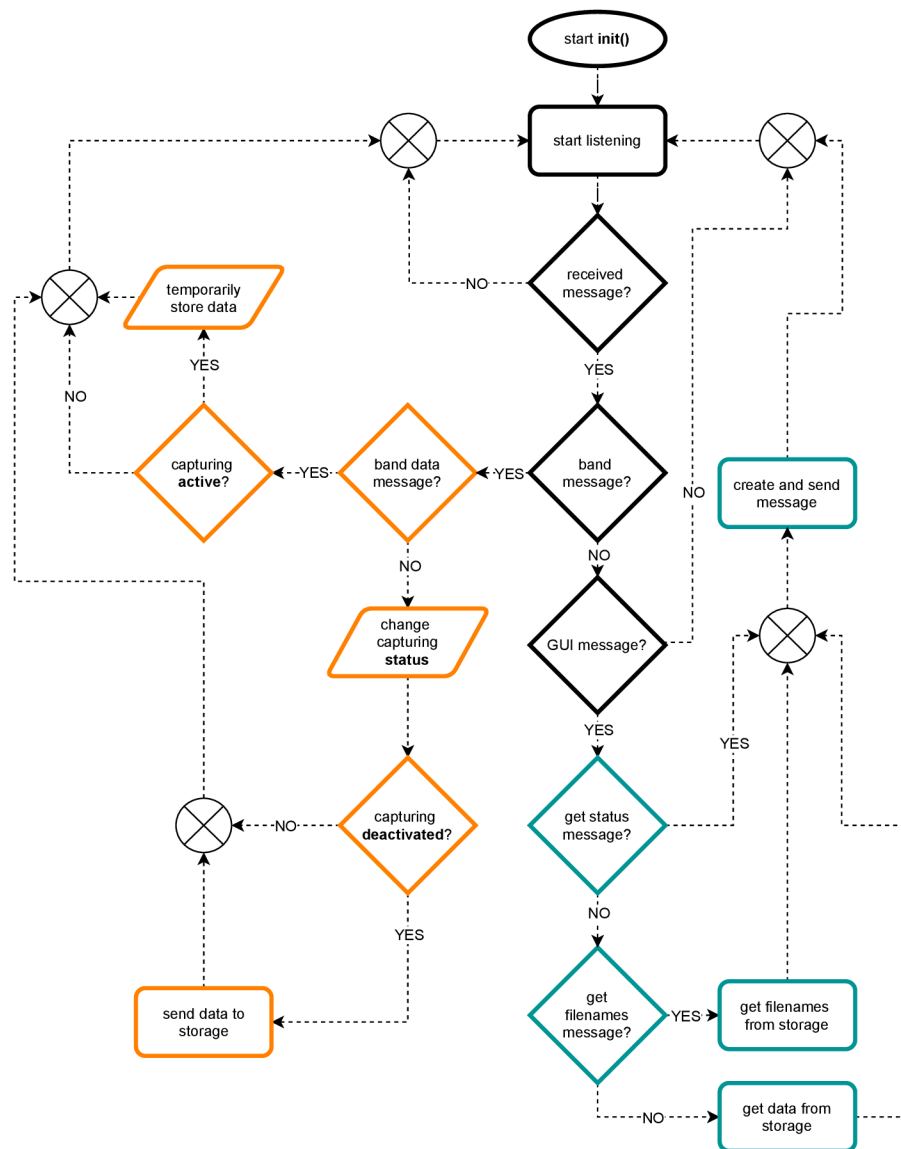
Component flow designs

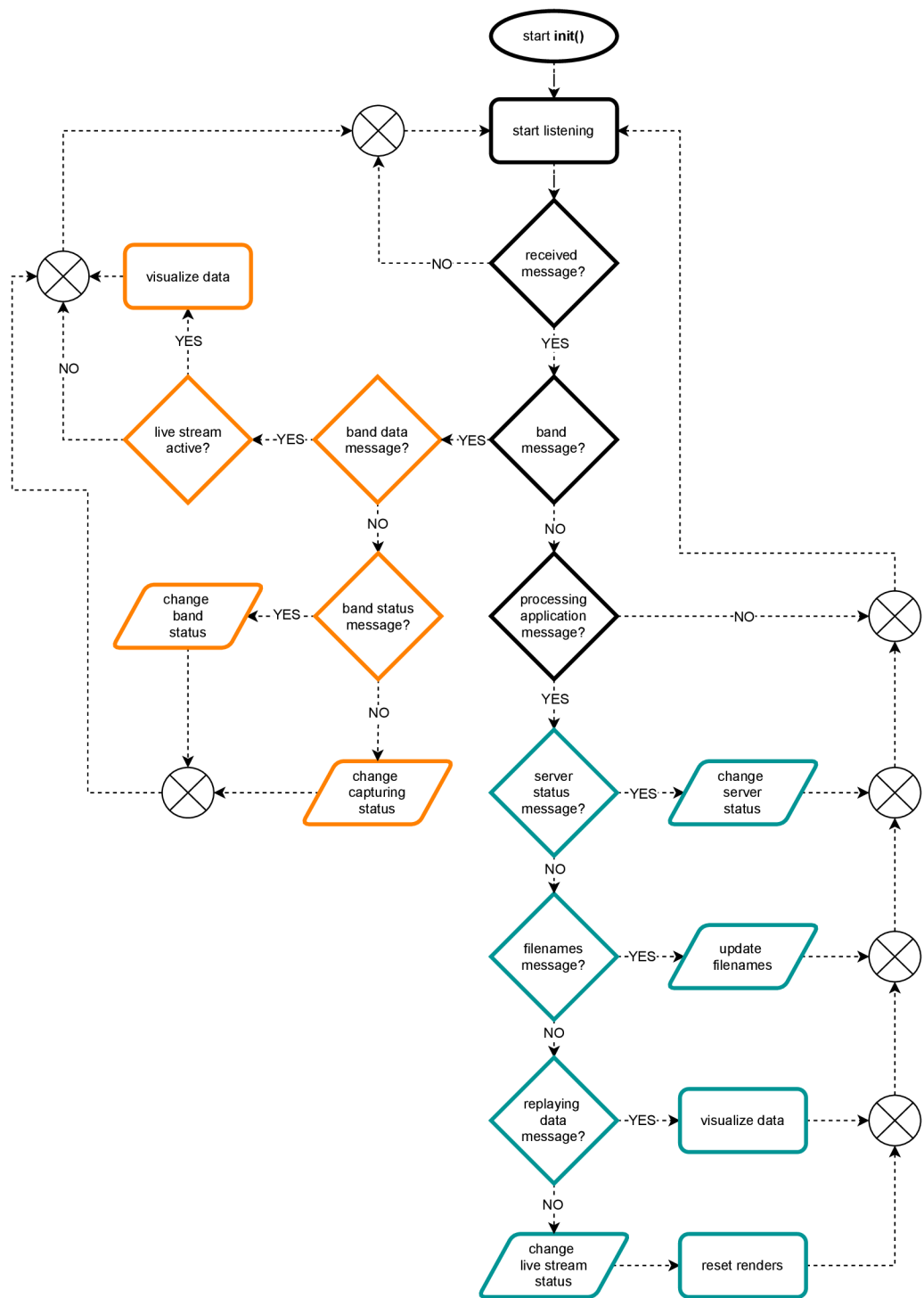Figure B.1: Workflow algorithm design for the processing application.

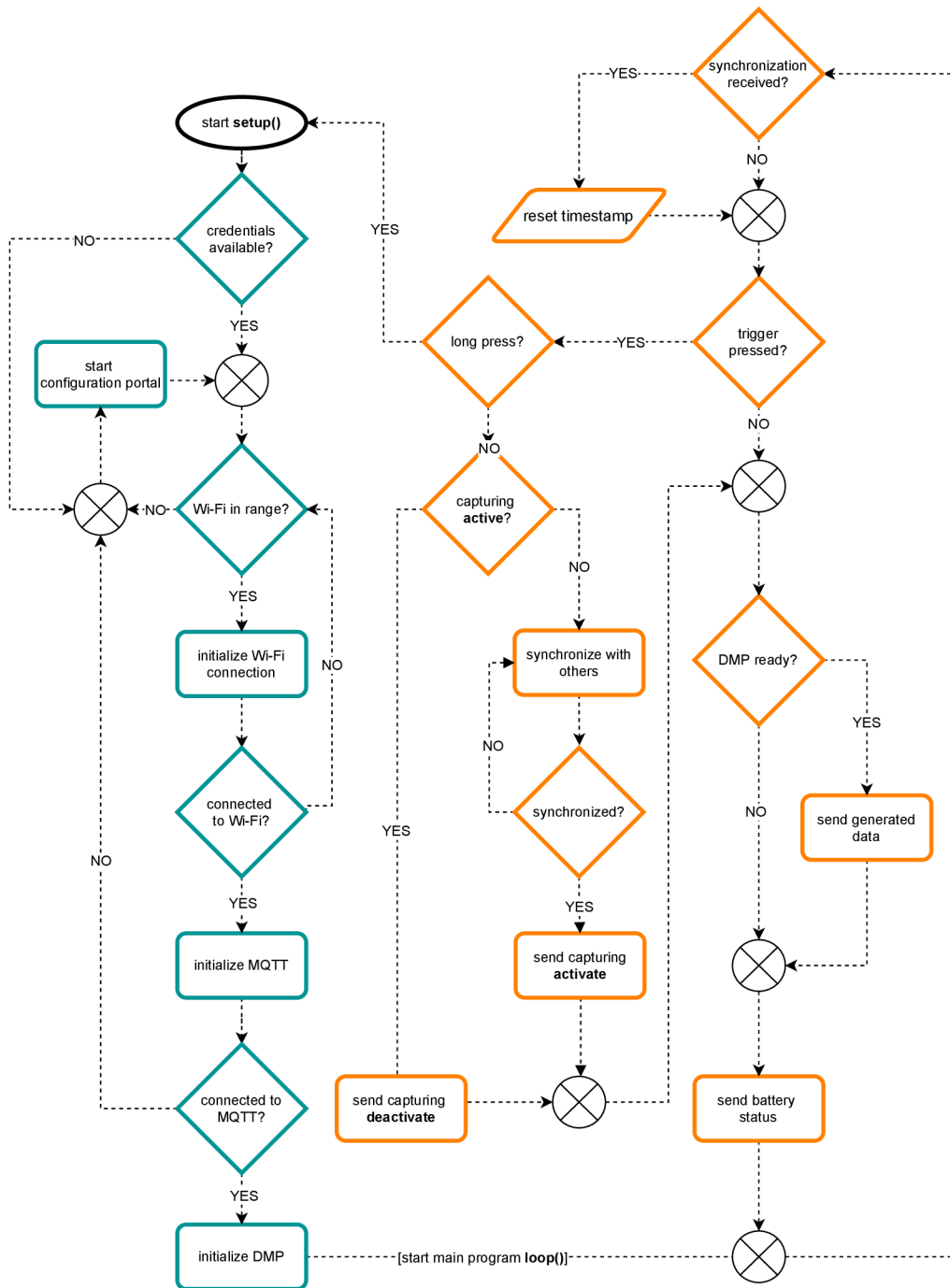Figure B.2: Workflow algorithm design for the visualising application.

Figure B.3: Workflow algorithm design for the sensoric device.
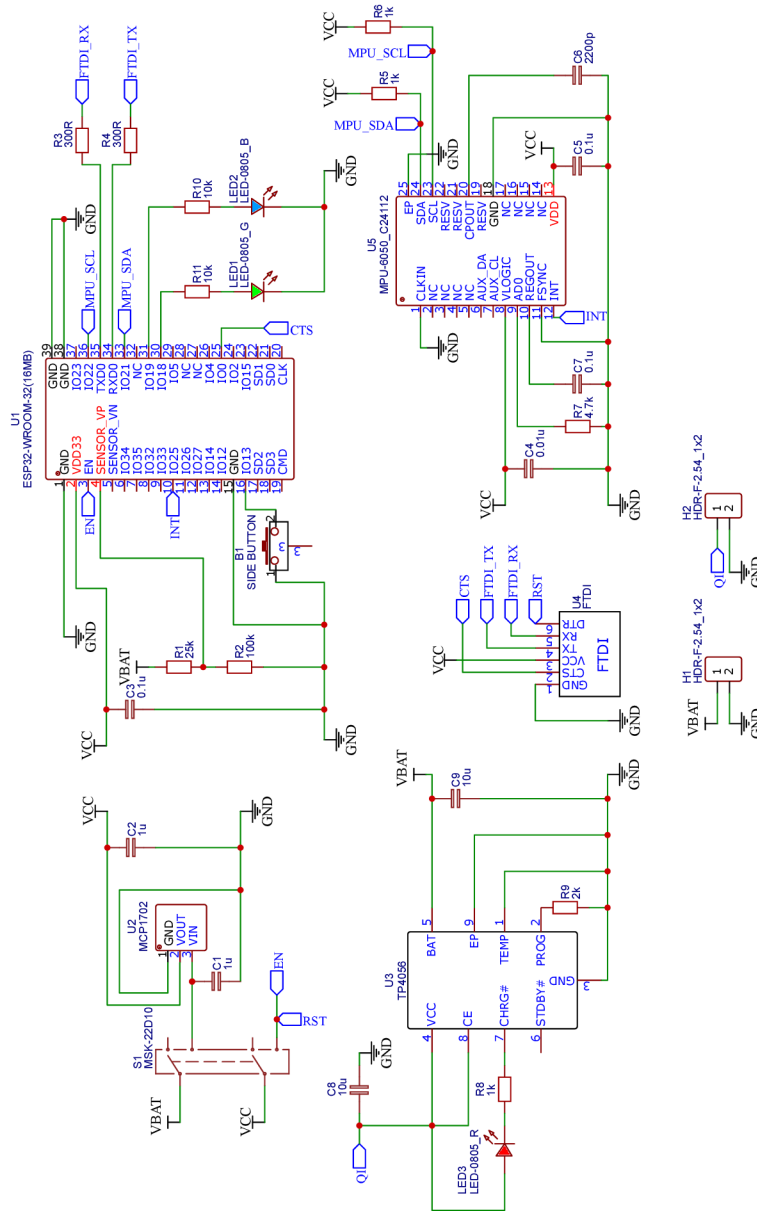
Appendix C

Device circuit schematic

Figure C.1: Diagram of a complete circuit design for a custom motion capturing device.
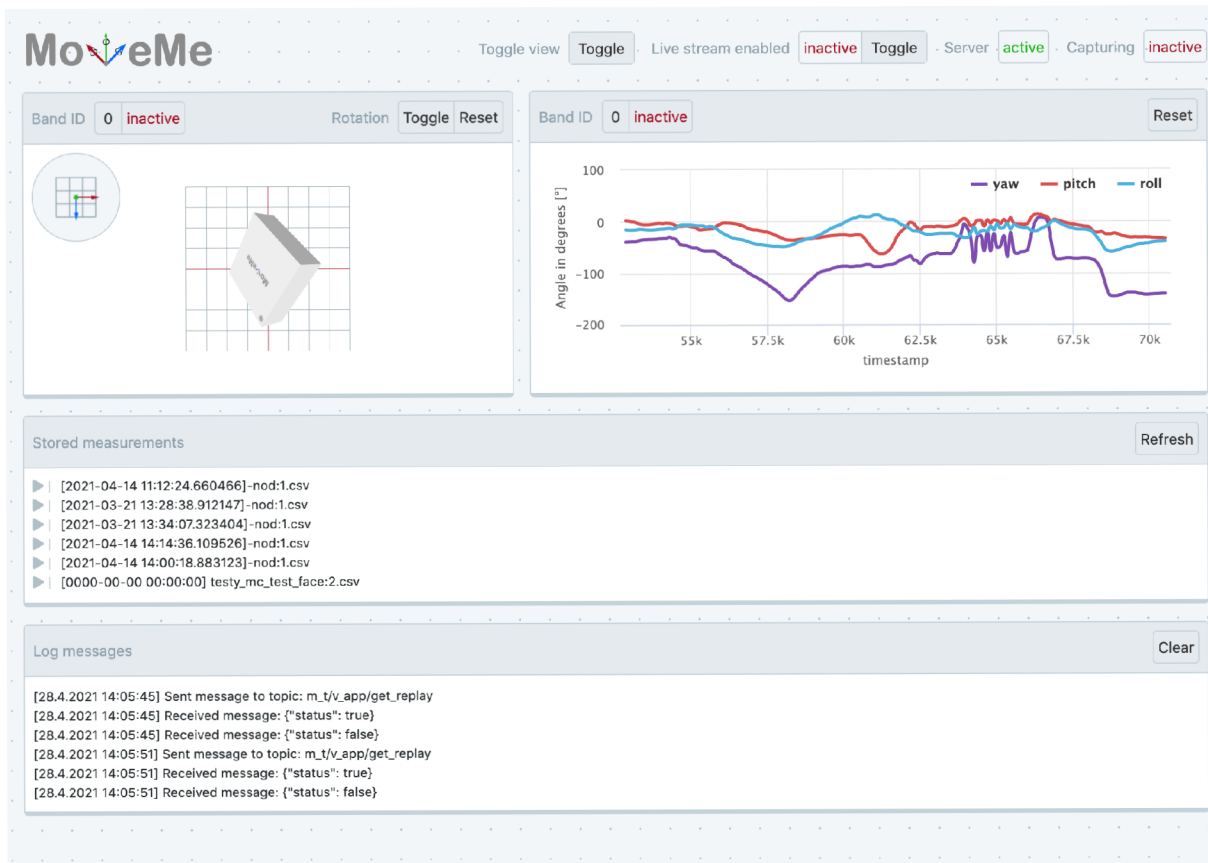
Appendix D

Implementation of GUI

Figure D.1: Final overview of the graphical user interface implementation.