POLITECNICO DI TORINO

Master's Thesis
in Mechatronic Engineering

Master's Thesis

Development of a Firmware for a Wearable Embedded System based on WI-FI for the Monitoring of Neurological Pathologies

Supervisors
prof. Stefano Pastorelli
prof. Laura Gastaldi
Ing. Silvio Massimino

Candidate
Federico Leone

Anno Accademico 2024-2025

Summary

Neurological pathologies frequently require continuous and reliable monitoring of patient movements, especially during rehabilitation. This thesis will follow the steps of the development of a firmware for a wearable embedded system that uses Wi-Fi networking to send the motion data that has to be analysed. The Zephyr Real-Time Operating System (RTOS), which was selected for the system due to its robust Internet of Things (IoT) capabilities and modular architecture, serves as its foundation.

A low power 9-axis inertial sensor was selected for gathering movements data. The firmware efficiently manages data acquisition, processing, and wireless transmission to external devices or cloud platforms for further analysis. The Zephyr RTOS ensures real-time performance and energy efficiency, essential for wearable applications.

The firmware's functionality includes reading data from the inertial sensors, noise reduction, and the provision of a message broker for bidirectional data transfer. Data validation is performed by comparing computed results with values provided by the sensor's integrated Digital Motion Processor.

By improving remote monitoring of patients undergoing rehabilitation, the developed firmware aims to provide to clinicians a new instrument to gather significant data. The integration of real-time data processing, embedded systems, and IoT creates new opportunities to enhance the treatment of neurological conditions.

Acknowledgements

I wish to express my thanks to Ing. Silvio Massimino of Brain Technologies for giving me the opportunity to work on this project.
A special thanks goes to Ing. Dario Gogliandolo, Saverio Milo, Alessandro Bocchio, and Francesco D'Introno for their support in helping me navigate and overcome the challenges I encountered throughout this thesis.

Contents

1 Introduction 7
1.1 Motivations 7
1.2 Chapters organization 8

2 State of Art 9

I Comparison between various open-source RTOS 10

3 Introduction to the RTOS 11
3.1 Overview of the Three RTOS 11
3.1.1 FreeRTOS 11
3.1.2 ThreadX (Azure RTOS) 12
3.1.3 Zephyr RTOS 13
3.2 Confrontation 14
3.3 Conclusion 14

II ESP32 and ICM20948 15

4 Development Boards Evaluation 16
4.1 ESP32 Family Board 16
4.1.1 ESP32-WROOM-32 17

5 ICM20948 - IMU (Inertial Measurement Unit) 18

III Zephyr Project 20

6 Zephyr Project Analysis 21
6.1 Zephyr GitHub repository 21
6.2 Zephyr Project Structure 21
6.3 Configuration and Build System 22
6.3.1 Kconfig 22
6.3.2 Devicetree 22
6.4 Device and Driver Model 23

6.5 Real-Time Capabilities . . . 23
6.5.1 Priority based scheduler . . . 24
6.5.2 Semaphores, Mutex and Race Condition . . . 25
6.6 Conclusions . . . 25

IV Wi-Fi connection **26**

7 What is the Wi-Fi **27**

8 Connecting to the Wi-Fi with Zephyr **28**
8.1 CMakeLists.txt . . . 28
8.2 Overlay file . . . 28
8.3 prj.conf file . . . 29
8.4 main.c file . . . 30

V MQTT Broker **36**

9 What is the MQTT **37**

10 Connecting to an MQTT broker with Zephyr **38**
10.1 Initialization of the Client . . . 38
10.2 The thread . . . 40
10.3 Publisher Function . . . 40
10.4 Subscriber Function . . . 41

VI IMU implementation **43**

11 The SparkFun Library for ICM20948 **44**
11.1 ICM_20948.c . . . 45
11.1.1 ICM_20948_init() . . . 45
11.2 Raw data Reading and Conversion . . . 48
11.3 Magnetometer Calibration . . . 49
11.3.1 Hard-Iron Calibration . . . 50
11.3.2 Soft-Iron Calibration . . . 50
11.3.3 Calibration code . . . 51
11.4 Data Acquisition . . . 52
11.4.1 Angle Computation . . . 54
11.5 Sensor Fusion . . . 55
11.5.1 Mahony Filter . . . 56

12 Algorithm Outputs **62**
12.1 Raw Data . . . 62
12.2 Acceleration and Magnetometer, angles computation . . . 64

12.3 Gyroscope with Integral Action 65
12.4 Mahony Filter Outputs 66

VII Conclusions **70**

13 Conclusions **71**
13.1 Future Work 71

List of Tables

List of Figures

3.1 FreeRTOS logo 11
3.2 Logo of Azure (on the left) and of ThreadX (on the right) 12
3.3 Zephyr logo 13
3.4 Increse of number of investors for various RTOS since 2016 13
4.1 Espressif Systems logo 16
4.2 ESP32-WROOM-32 17
5.1 ICM20948 produceD by pimaroni.com 18
6.1 Devicetree build flow 23
6.2 Zephyr state transition 24
7.1 Wi-Fi certified logo 27
11.1 calibration of the magnetometer successfully eliminates the hard and soft iron disturbances 50
12.1 Accelerometer raw data 63
12.2 Gyroscope raw data 63
12.3 Magnetometer raw data 64
12.4 Angles computation from gyroscope readings 65
12.5 Angles computation from gyroscope readings 66
12.6 Angles computation from sensor fusion through a Mahony filter 67
12.7 Angles computation from sensor fusion through a Mahony filter 68

12.8 Angles computation from gyroscope readings, same readings as Figure 10.7 68

Chapter 1

Introduction

1.1 Motivations

In the last decade the field of wearable devices has seen significant growth. In healthcare they have gained popularity for their ability to monitor physiological and biomechanical parameters which can provide valuable data for the treatment of the patient. There are different ways in which these devices can support doctors, such as tracking in real-time the muscle activity or other critical physiological signals and contributing to early diagnosis, personalized treatments and improved patient outcomes.

One of the critical applications of these devices is the monitoring of neurological pathologies, that usually require long-term observation and precise data collection. The traditional methods of treating this diseases are through subject evaluation by healthcare professionals and infrequent clinical visits. Wearable technologies provide a means for continuous and objective data collection which enable new possibilities for a remote rehabilitation program.

In this thesis, the development of a firmware for a wearable embedded system will be presented. The system is based on Wi-Fi communication, designed to assist in the rehabilitation of patients with neurological disorders and it is built around the ESP32 microcontroller and integrates the ICM-20948 IMU. The sensor allows us to measure the acceleration, angular velocity and orientation, making the tracking of the patient movements possible with enough precision without requiring additional equipment.

To ensure efficiency and real-time performance, the firmware has been developed using Zephyr, an open-source RTOS that has become increasingly popular in recent years. Zephyr provides a robust and modular environment for embedded applications which allows for a better task scheduling, resource management and integration with various sensors [15]. This project wants to provide a cost-effective, high-performance and adaptable solution for healthcare professionals and researchers working in the field of neurorehabilitation. On the other hand, the Wi-Fi communication facilitates seamless data transmission allowing remote monitoring of patient progress and adjust rehabilitation programs, thereby enhancing the overall effectiveness of treatment strategies whitout significant financial budget.

1.2 Chapters organization

In this chapter the thesis structure will be summarized.0.3cm
Chapter 3: this chapter will be reported a general analyses of various RTOS. That will be followed by a comaprison on the advantages and disadvantage between them.

Chapter 4-5: the reasoning that brought us to choose the board and sensor are explained here.

Chapter 6: We will delve deeper into the Zephyr RTOS and its main features.

Chapter 7-8: The code that let us establish and mantain a Wi-Fi connection is discussed and analyzed.

Chapter 9-10: As for the WI-Fi the same as been done in these chapters with The connection to a MQTT broker.

Chapter 11: this chapter presents the implementation of the sensor and the code utilized for it with all the further with all the further additions.

Chapter 12: the data that were obtained from the sensor are reported and analyzed here.

Chapter 2

State of Art

Wearable technologies in the various field have been applied for the development of various systems aimed at monitoring neurological conditions such as Parkinson's disease, stroke recovery, and other motor disability. IMUs are largely used in these wearable devices, employing sensors such as accelerometers, gyroscopes and magnetometers for movement tracking. Wireless connection are often incorporated in them and between the various communication technologies the most used are Bluetooth and Wi-Fi to transmit the data to external devices that are required to analyze them.

The effectiveness of IMU-based monitoring systems during rehabilitation have been proven by multiple studies. Their utility can vary from tremor detection to motor coordination evaluation. These studies highlight that wearable devices can provide measurements which can be used to improve rehabilitation strategies. In some cases machine learning algorithms are integrated in this devices for enhancing accuracy, detecting abnormalities and predicting disease progression.

However this devices are usually very expensive and not fully customizable while the existing open-source alternatives may lack real-time capabilities or robust firmware support. For this reason this project aims for an affordable and more flexible solution.

The ESP32 microcontroller strands out over other platforms du to its integrated Wi-Fi capabilities, low-power consumption and affordability. The Zephyr RTOS is a young framework for embedded that seeing an exponential increase in support during the past years and it's open-source nature facilitates adaptability and allows developers to modify and expand the system according to their necessity.

Part I

Comparison between various open-source RTOS

Chapter 3

Introduction to the RTOS

Fist there has been the need to select the appropriate RTOS for the firmware that it's going to be developed. Choosing the RTOS more fit for the project necessities is essential considering it will impact performances and efficiency.
Between the various open-source RTOS we selected three possible candidate that could fit our needs:

- FreeRTOS
- Azure RTOS (ThreadX)
- Zephyr

Now, this three RTOS will be compared, highlighting strength and weaknesses and in the end explaining why Zephyr was ultimately chosen.

3.1 Overview of the Three RTOS

Let's start analyzing the three systems singularly.

3.1.1 FreeRTOS

Figure 3.1. FreeRTOS logo

[5]This RTOS is well-known for its simplicity and efficiency that make it a great option for embedded application which require minimal resource constraints. Its structure is based on a microkernel architecture. The supported lightweight scheduler make it a valuable choice for low-power devices running a single and well-defined function. This system can boast a small memory footprint, minimizing overhead.

However, the support it offers related to multi-core processing is basic making it less advanced in this area then its main competitors. It is also supported on less board limiting the freedom of the developer on choosing the hardware. On top of that, the lack of built-in safety certifications limits its use in various fields where they are required, such as the medical field. In fact its safety depends only on simple memory protection, execution privilege level hardware and on the the OS itself.

3.1.2 ThreadX (Azure RTOS)

Figure 3.2. Logo of Azure (on the left) and of ThreadX (on the right)

[13]ThreadX, also known as Azure RTOS in the past, offers high-performances, presenting a great optimization that accounts for incredibly fast context switching. This system features a pikokernel architecture. Such structure means that all its services plug directly into the core avoiding unnecessary layering which helps reducing function call overhead and allows for sub-microsecond context switching which makes Azure RTOS one of the fastest open-source RTOS on the market.

Additionally, ThreadX can display robust safety certifications. Between them IEC-61508 SIL 4 can be highlighted since it gives the guarantee that Azure RTOS is well suited for safety-critical applications. By presenting high scalability, this system is able to efficiently manage up to 200 threads making it more suited for complex embedded systems. On the other hand, it lacks the open-source flexibility of the other two RTOS and the community driven development found in Zephyr since is been bought form Microsoft which however, as promised to keep the project open-source.

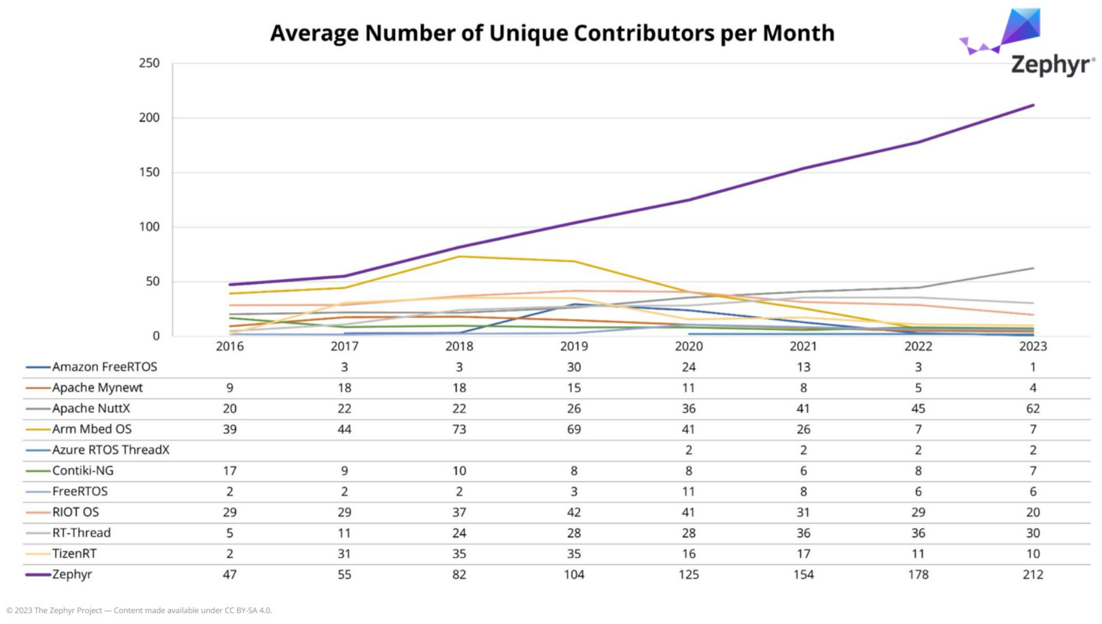
To this day, as today, Azure RTOS remains one of the most used RTOS in the entire world.

3.1.3 Zephyr RTOS

Figure 3.3. Zephyr logo

[15]The relatively young Zephyr RTOS is characterized by its modular and salable design, making it a versatile choice for embedded systems. Since its foundation, it aims to grants the best possible supports for IoT and sensor integration. The goal of this RTOS is to ensure that the system can adapt to every possible microcontroller and device with minimum changes. While it's still far away to achieve this goal, it has attracted on itself the attentions of various giants of the industry through which it has ensure continuous and long-term support.

Particular attention as to be given to its device driver model, based on devicetree



	2016	2017	2018	2019	2020	2021	2022	2023
Amazon FreeRTOS		3	3	30	24	13	3	1
Apache Mynewt	9	18	18	15	11	8	5	4
Apache NuttX	20	22	22	26	36	41	45	62
Arm Mbed OS	39	44	73	69	41	26	7	7
Azure RTOS ThreadX					2	2	2	2
Contiki-NG	17	9	10	8	8	6	8	7
FreeRTOS	2	2	2	3	11	8	6	6
RIOT OS	29	29	37	42	41	31	29	20
RT-Thread	5	11	24	28	28	36	36	30
TizenRT	2	31	35	35	16	17	11	10
Zephyr	47	55	82	104	125	154	178	212

Figure 3.4. Increse of number of investors for various RTOS since 2016

description improving hardware abstraction and simplifies development.

Zephyr presents a monolithic kernel but it is however largely scalable making it able to provide a lightweight footprint. This aspect make it possible to enable only the parts of the RTOS that are needed by the developer which make it possible to easily optimize its performance and memory usage. It also provides strong network capabilities including large support for BLE 5.0, BSD sockets and Wi-Fi making it a preferable choice for connected devices.

Another strength of this system is its robust security and safety framework, including IEC61508 certification and PSA-certified IoT security features.

3.2 Confrontation

After careful observation and an analysis of the advantages and disadvantages of various RTOS, the decision was made to use Zephyr for the development of this project. This decision was due to its combination of flexibility, reliability and advanced networking capabilities while also promising a long-term support granted also by the growing interest in it.

Between its standout features there are its scalability and modularity. Unlike FreeRTOS, which is more minimal and requires to integrate external and usually not official parts to compensate, Zephyr possesses a complete system and allows developers to include only the necessary components for their project granting reduction of resources consumption and optimization of performances. This makes it a optimal choice for sensor-driven applications that require a fine balance between processing power and efficiency.

The strong network and IoT support of Zephyr are essential for devices that needs real-time data transmission. For this reason its deep integration with modern IoT protocol give it an advantage in this field over the other two RTOS.

With the need of security and safety certifications, required for medical applications, the ones possessed by Zephyr ensure its compliance with the relative standards. As we said before, while ThreadX also possess some certicate, FreeRTOS is lacking on this aspect. Furthermore, the long-term support and industry backing are also an important advantage for Zephyr that will continue to evolve and grow.

3.3 Conclusion

The reasons discussed above where the motivations that led to the development of the project using the Zephyr RTOS. As far as we know This RTOS seems to be the ideal choice, providing a stable and efficient foundation for real-time data processing and communication through its scalability, advanced networking capabilities, robust safety features, and strong industry support. Thanks to the combination of this features Zephyr can meet both present and future demands of this project making it the best RTOS for this work.

Part II

ESP32 and ICM20948

Chapter 4

Development Boards Evaluation

While evaluating the board supported to this day by Zephyr, the boards we ended up choosing where those of the ESP32 family and the Raspberry Pi Pico W. Both of these platforms offer significant advantages in terms of community support, hardware capabilities, and cost-effectiveness.

The second one is a version of the its basic model (Raspberry Pi Pico) with the Wi-Fi implemented in it. However, the Raspberry Pi Pico W had to be discarded do to the Wi-Fi which, at that moment, was not supported for this board in Zephyr.

4.1 ESP32 Family Board

Figure 4.1. Espressif Systems logo

The ESP32 family is a series of board developed by Espressif Systems, known for its versatility , robust wireless communication features, and extensive support for various development frameworks, including Zephyr. Between the various board of the ESP32 family the board that was chosen to be used is the ESP32 WROOM 32.

4.1.1 ESP32-WROOM-32

This decision was based on various factors:

- **Zephyr Compatibility**: The ESP32-WROOM-32 module [7] has active support within the Zephyr RTOS ecosystem, ensuring a stable development environment.
- **Integrated Wi-Fi and Bluetooth**: The module features built-in Wi-Fi and Bluetooth (Classic and Low Energy), making it well-suited for IoT wireless communication applications.
- **Processing Power**: Equipped with a dual-core Xtensa LX6 processor, the ESP32-WROOM-32 provides sufficient computational resources for real-time applications.
- **Extensive Peripheral Support**: The board includes a wide range of peripherals such as GPIO, I2C, SPI, UART, ADC, and DAC, allowing for seamless integration with sensors and other hardware components.
- **Low Power Consumption**: The ESP32 offers multiple power-saving modes, making it an efficient choice for battery-operated applications.

Figure 4.2. ESP32-WROOM-32

In conclusion this board gave us the insurance that the development process would have aligned with the project's requirement, especially in terms of Wi-Fi communication and Zephyr support. For this reasons this board provided a solid foundation for implementing the firmware in development.

Chapter 5

ICM20948 - IMU (Inertial Measurement Unit)

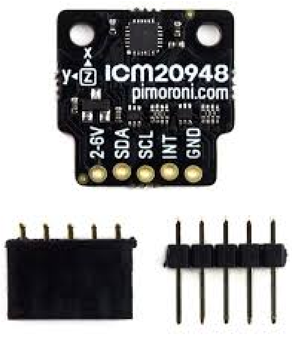


Figure 5.1. ICM20948 produceD by pimaroni.com

The ICM-20948, developed by TDK InvenSense, is a highly versatile Inertial Measurement Unit (IMU) with advanced sensor fusion capabilities while maintaining low power consumption [10]. This IMU integrates:

- a 3-axis accelerometer
- a 3-axis gyroscope
- a 3-axis magnetometer

These sensors grants comprehensive motion tracking and orientation sensing.

Between it's main properties we can find:

- **Low Power Consumption**: The ICM-20948 is designed for low-power applications, fundamental for battery-operated wearable devices.

- **9-Axis Motion Sensing**: The integration of a gyroscope, accelerometer, and magnetometer enables precise motion tracking and orientation estimation.
- **High Sensitivity and Accuracy**: The sensor provides high-resolution measurements with configurable full-scale ranges, making it suitable for applications requiring precise motion analysis.
- **Digital Motion Processor (DMP)**: The built-in DMP offloads complex sensor fusion calculations from the main processor, improving efficiency and reducing computational load.
- **I2C and SPI Communication**: The sensor supports both I2C and SPI interfaces, ensuring easy integration with microcontrollers and embedded systems.
- **Advanced Filtering and Calibration**: The ICM-20948 includes onboard filtering and calibration features to enhance measurement accuracy and reduce noise.

On the other hand this sensor is still not implemented in Zephyr and this has required an adaptation of a valid library for the ICM20948. The library that was implemented in the end was the *"SparkFun_ICM-20948_ArduinoLibrary"*. However this sensor together, with the EPS32-WROOM-32, provides a robust hardware platform for a wearable application which ensure relaible motion tracking and wireless connectivity for fast data transmission.

Part III

Zephyr Project

Chapter 6

Zephyr Project Analysis

I want now to delve deeper into the structure and functionalities of Zephyr. In this chapter the key aspects of its ecosystem will be outlined, starting with the project structure, configuration system and device management to the relevant components which facilitate the development of the firmware.

6.1 Zephyr GitHub repository

Zephyr has been obtained following the Getting Started Guide [14], present on the official website, and through it's GitHub repository [16]. Below the structure of the Zephyr folder with its main components is reported:

```
zephyrproject
├─ .west
├─ bootloader
├─ build
├─ modules
├─ tools
└─ zephyr
```

6.2 Zephyr Project Structure

Zephyr grants to the projects, that are realized using it, to take advantage of its modularity and reusability of components. the typical Zephyr project is composed by:

- **Application Code**: The core logic of the firmware, usually located in the `src/` directory, containing the main program (`main.c`) and additional application-specific modules.

- **CMake and Build Configuration**: Zephyr uses `CMake` and Python-based `west` meta-tool for project management. The `CMakeLists.txt` file in the root directory specifies how the application is built.
- **Kconfig System**: Zephyr relies on `Kconfig` for configuration management, allowing fine-grained control over features and system settings. On the other hand, a `prj.conf` file let the developers select a base of configuration required by the project without the need to tune the project with the `Kconfig` after the build.
- **Devicetree (DTS)**: The device tree provides a hardware description, specifying peripherals and configurations for the target board.
- **Board Support Package (BSP)**: Contains essential files for hardware abstraction and integration with specific microcontrollers.

6.3 Configuration and Build System

The **west-tool** is a command-line used by Zephyr build projects, flash the firmware on the target device and to manage Zephyr repositories. In fact this command can be used to update the libraries and the files system of Zephyr.

On the other hand, this RTOS configuration is made of 2 main components which are the Devicetree (DTS) and the `Kconfig`.

6.3.1 Kconfig

This tool gives the developers the ability to access a configuration menu with the command `menuconfig` or `guiconfig`. In this menu the developer will be able to enable or disable a long list of Zephyr features which, however, have to be implemented in the `Kconfig` file. Thank to this is possible to customize the RTOS without the need of modifying the source code directly.

6.3.2 Devicetree

The DTS is a hierarchical hardware description language as well as a configuration language. It defines all the hardware configuration like boards, buses and communication interfaces. It make possible to enable abstraction which reduce the need for hardware-specific modifications in the application code.

The devicetree input are the *devicetree sources* and the *devicetree bindings*.
The second ones are required to declare all valid structures for the devicetree nodes and without the devicetree sources would loose their meaning and would be unusable. During the configuration phase every node present in the devicetree file used by the project is matched to a binding file. Once all the nodes are validated the information of the bindings is still used by the build system to generate the macros required for those nodes.

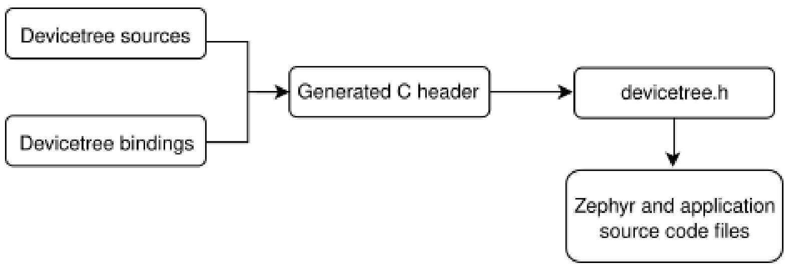


Figure 6.1. Devicetree build flow

Overlay

The Overlay is a file that can be created for requesting the build system to modify or add a part of the devicetree. This is a powerful instrument in the end of developers that are not required to modify each time the devicetree structure of the board on which are working to match their need but can instead, in this file inside the project, request those changes. In this way the correct and general devicetree can be also easily adapt to every possible variation of the boards.
Moreover it makes possible to initialize in the devicetree the non vital components as disable so that only the necessary one will be activated. For example, this is the case for the ESP32 Wi-Fi module for which an overlay file for activating it is required.

6.4 Device and Driver Model

Another advantage that Zephyr offers is represented by its device and driver model. The RTOS provides a standardized API for device driver which ensure compatibility between hardware platforms.Some important features are:

- **Sensor Subsystem**: this subsystem provides generalized API for sensor integration
- **Power Management**: Zephyr includes built-in power management features for the aim of optimizing energy consumption in embedded applications.
- **Networking Stack**: This aspects is really important for this thesis project because Zephyr support multiple networking protocols like the Wi-Fi that will be used for this application.

6.5 Real-Time Capabilities

Being an RTOS, Zephyr provides robust real-time performances which grants:

- **Preemptive Multitasking**: this ensures that the tasks are scheduled based on the assigned priority.

- **Kernel Synchronization Mechanisms**: Includes semaphores, mutex and message queue for efficient inter-task communication.
- **Low Latency Operations**: give the systems deterministic behaviour which is required by real-time data acquisition.

6.5.1 Priority based scheduler

A key part of any RTOS is its scheduler. Zephyr possesses a priority-based preemptive scheduler that ensures that the threads (also known sometimes as tasks) execute based on their assigned priority. This scheduler allows Zephyr to let higher priority tasks to preempt lower priority ones.

The way it handles threads is well illustrated in the pictures below:

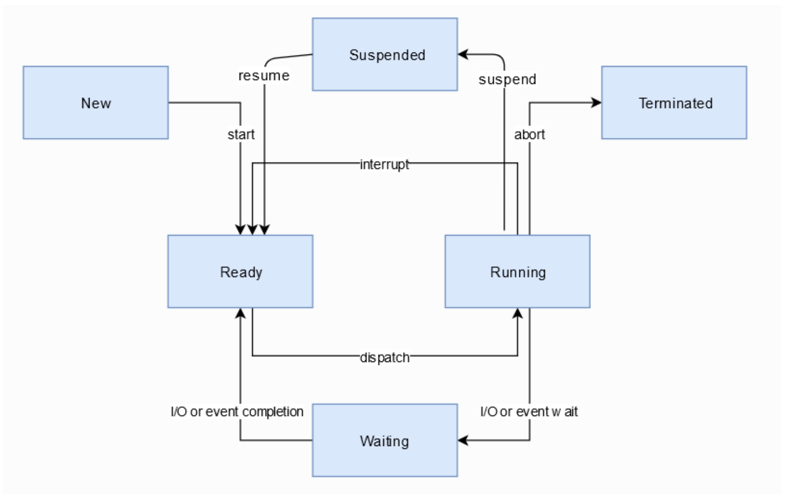


Figure 6.2. Zephyr state transition

the boxes illustrated in the figure 6.5.1 are the execution states between which a thread in a Zephyr project can transition:

- **New**: the thread has been created but it has yet to start.
- **Ready**: the thread is ready but as to wait for a valid CPU to be freed so that it can run.
- **Running**: the thread is actively executing on the CPU.
- **Waiting**: the thread is blocked and it has to wait for a priority level task to finish executing before it can take back the CPU and and finish running.
- **Suspended**: the thread is paused so that it can be resumed later.
- **terminated**: the thread has completed execution or was aborted.

6.5.2 Semaphores, Mutex and Race Condition

It became necessary to evaluate correctly and with attention the priority of the tasks to avoid critical situations. In fact, if the priority and the semaphore are not correctly inserted in the code, the program could end up in a situation were two or more task are locked by each other adn the program can't run anymore.

Semaphores and mutes are powerful instruments that can help handle some delicate situation in real-time ambient. Because some resources are shared between more threads, in case of a multi-core processor or preemption, it's possible that two thread will require simultaneously to access the same resource. In this case the semaphore will prevent this and the last thread that will request it will have to wait for the first to finish using it before being able to access it. This situation is called race condition and can be the cause of a deadlock.

In fact, if a lower priority task takes a resource and is then preempted by a higher priority task, which also need to access it, before releasing it then the the lower priority one will not be able to conlcude and so the resource will never be freed while the higher priority task will await forever for the other one to release it.

This critical event can happen also in the presence of more shared resources. If two task are required to lock 2 resources but this resources are locked by the one each this will cause both the thread to wait forever for the other one to release the resource that it has locked.

These situations can only be avoided through a careful study of the priorities and assignment of semaphores.

6.6 Conclusions

In this thesis Zephyr operates a crucial rule in:

- Managing sensor data to help handling the data take from the ICM20948.
- Handling network communication by ensuring through its networking stack a stable Wi-Fi connection and transmit data via MQTT.
- Optimizing power consumption to ensure a low energy usage.
- Ensuring modularity and scalability so that if the project will require further tuning in the future this will be easily accomplished

Part IV
Wi-Fi connection

Chapter 7

What is the Wi-Fi

Figure 7.1. Wi-Fi certified logo

The Wi-Fi (which stand for *Wireless Fidelity*) is a technology that enables wireless communication between devices over radio waves without the need of physical cables and granting high flexibility eliminating the constraints of wired technologies. It is based on the 802.11 family of standards that defines the protocols for wireless communication [2]. Since its arrival in the late 1990s, Wi-Fi has became a core component of the modern communication [9].

The Wi-Fi can operate in different frequency bands, primarily 2.4 GHz, 5 GHz and 6 GHz [9]. Iin this project the 2.4 GHz Wi-Fi was used which offers longer range. However, at this frequency, Wi-Fi performance can be affected by things like signal strength, interference from other devices, and network congestion.

Chapter 8

Connecting to the Wi-Fi with Zephyr

We will now investigate the code and files that make it possible for the code to connect to the Wi-Fi [6].

8.1 CMakeLists.txt

```
# SPDX-License-Identifier: Apache-2.0

cmake_minimum_required(VERSION 3.20.0)

find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(esp32_wifi)

FILE(GLOB app_sources src/*.c)
target_sources(app PRIVATE ${app_sources})
```

In this case, the CMakeLists file remained unchanged. It simply defined the ZEPHYR_PATH and where the files.c are located in the various folders requested by the project. Having just the \src folder with all the files in it, it reports only that one. The "*" before the.c tells the program to take into consideration all the.c files in the folder. Otherwise this files should be added one by one.

8.2 Overlay file

The first thing to do is to activate the Wi-Fi module of the ESP32. For doing so, you need to activate it through an overlay file of the board. This will be detected by the west tool while building the program and will modify the related node of the devicetree.

Below is reported the code of the overlay file:

```
/*
 * Copyright (c) 2022 Espressif Systems (Shanghai) Co., Ltd.
 *
 * SPDX-License-Identifier: Apache-2.0
 */

&wifi {
  status = "okay";
};
```

Pay attention to the name of the file which is required to precisely point to the exact board for which the west tool is building the project; otherwise it will not be recognized. In this case *esp32.overlay* and also *esp32_devkitc_wroom* will not be taken into account by it and so it would use the standard devicetree for the board without making changes.

8.3 prj.conf file

Through the prj.conf file, we will now request to Zephyr to include the modules required for the connection. In this file, which contains the configuration settings for Zephyr RTOS, the Wi-FI connectivity and parameters for network communication.

```
#
# Copyright (c) 2023 Craig Peacock.
#
# SPDX-License-Identifier: Apache-2.0
#

CONFIG_NETWORKING=y

CONFIG_NET_IPV4=y
CONFIG_NET_IPV6=n

CONFIG_NET_DHCPV4=y
CONFIG_WIFI=y
CONFIG_INIT_STACKS=y

CONFIG_NET_UDP=y
CONFIG_DNS_RESOLVER=y
CONFIG_DNS_RESOLVER_AI_MAX_ENTRIES=10

CONFIG_HTTP_CLIENT=y

# Following line must be enabled, if not WiFi connection fails with -1.
CONFIG_NET_CONFIG_SETTINGS=y
# Add CONFIG_INIT_TIMEOUT to prevent 30 second timeout when starting:
CONFIG_NET_CONFIG_INIT_TIMEOUT=1

CONFIG_NET_TX_STACK_SIZE=2048
CONFIG_NET_RX_STACK_SIZE=2048
```

```
CONFIG_NET_PKT_RX_COUNT=10
CONFIG_NET_PKT_TX_COUNT=10

CONFIG_NET_BUF_RX_COUNT=20
CONFIG_NET_BUF_TX_COUNT=20

CONFIG_NET_MAX_CONTEXTS=10

CONFIG_NET_L2_ETHERNET=y

CONFIG_PRINTK_BUFFER_SIZE=2048
```

Let 's now breakdown the key elements.
First the configurations for networking:

- **CONFIG_NETWORKING** → enables networking features
- **CONFIG_NET_IPV4** and **CONFIG_NET_PIV6** → enables IPv4 support (y) while also take in consideration the possibility of enabling IPv6 that is disable for now (n)
- **CONFIG_NET_DHCPV4** → let the device get an IP adress dynamically from a DHCP server
- **CONFIG_NET_CONFIG_SETTINGS** → enables automatic network configuration

For the Wi-FI, on th other hand, the following option was obviously selected:

- **CONFIG_WIFI** → enables Wi-Fi support in Zephyr

The remaining aspect strictly related to the Wi-Fi will be handled directly by the code. The remaining settings are required for networking stack and buffer management while also optimazing the network stack. The only exception is CONIFG_PRINTK_BUFFER_SIZE which is used for debugging purpose and determine th size of the buffer for the `pritnk` function in bytes.

8.4 main.c file

The next file that will be analyzed is the `main.c`. This section will focus on the function that makes this project able to reach successfully its objective.

```c
void wifi_connect(void)
{
    struct net_if *iface = net_if_get_default();

    struct wifi_connect_req_params wifi_params = {0};

    wifi_params.ssid = SSID;
    wifi_params.psk = PSK;
    wifi_params.ssid_length = strlen(SSID);
    wifi_params.psk_length = strlen(PSK);
    wifi_params.channel = WIFI_CHANNEL_ANY;
    wifi_params.security = WIFI_SECURITY_TYPE_PSK;
    wifi_params.band = WIFI_FREQ_BAND_2_4_GHZ;
    wifi_params.mfp = WIFI_MFP_OPTIONAL;

    printk("Connecting to SSID: %s\n", wifi_params.ssid);

    if (net_mgmt(NET_REQUEST_WIFI_CONNECT, iface, &wifi_params, sizeof(
    struct wifi_connect_req_params)))
    {
        printk("WiFi Connection Request Failed\n");
    }
}
```

The function reported above is the one that handles the request of connection to the Wi-Fi. In the structure wifi_connect_req_params are saved all the data required for the connection. In particular the first two elements: this are the username (SSID) and the password (PSK) which in this case are predefined and kept in a separate file.

Then the program calls `net_mgmt()` for connection using the macro `NET_REQUEST_WIFI_CONNECT` while the security parameters are saved and passed to the function through the pointer `&wifi_param`. The `iface` is define as a pointer to the structure `net_if_get_default()` which is use for obtaining the default network interface.

```
void wifi_disconnect(void)
{
    struct net_if *iface = net_if_get_default();

    if (net_mgmt(NET_REQUEST_WIFI_DISCONNECT, iface, NULL, 0))
    {
        printk("WiFi Disconnection Request Failed\n");
    }
}
```

Here `net_mgmt()` is called again but for requesting the disconnection using the macro `NET_REQUEST_WIFI_DISCONNECT`.

```
void wifi_status(void)
{
    struct net_if *iface = net_if_get_default();

    struct wifi_iface_status status = {0};

    if (net_mgmt(NET_REQUEST_WIFI_IFACE_STATUS, iface, &status, sizeof(
    struct wifi_iface_status)))
    {
        printk("WiFi Status Request Failed\n");
    }

    printk("\n");

    if (status.state >= WIFI_STATE_ASSOCIATED) {
        printk("SSID: %-32s\n", status.ssid);
        printk("Band: %s\n", wifi_band_txt(status.band));
        printk("Channel: %d\n", status.channel);
        printk("Security: %s\n", wifi_security_txt(status.security));
        printk("RSSI: %d\n", status.rssi);
    }
}
```

This function can help verify the status of the Wi-Fi connection. To do so, it once again uses the function `net_mgmt()` with the macro `NET_REQUEST_IFACE_STATUS`. The data that it retrieve are the SSID, band, channel, security type and RSSI.

To make all of this data readable the functions `wifi_band_txt()` and `wifi_security_txt()` are used.

```
static K_SEM_DEFINE(wifi_connected, 0, 1);
static K_SEM_DEFINE(ipv4_address_obtained, 0, 1);

static void handle_wifi_connect_result(struct net_mgmt_event_callback *
    cb)
{
    const struct wifi_status *status = (const struct wifi_status *)cb->
    info;

    if (status->status)
    {
        printk("Connection request failed (%d)\n", status->status);
    }
    else
    {
        printk("Connected\n");
        k_sem_give(&wifi_connected);
    }
}

static void handle_wifi_disconnect_result(struct net_mgmt_event_callback
     *cb)
{
    const struct wifi_status *status = (const struct wifi_status *)cb->
    info;

    if (status->status)
    {
        printk("Disconnection request (%d)\n", status->status);
    }
    else
    {
        printk("Disconnected\n");
        k_sem_take(&wifi_connected, K_NO_WAIT);
    }
}

static void handle_ipv4_result(struct net_if *iface)
{
    int i = 0;

    for (i = 0; i < NET_IF_MAX_IPV4_ADDR; i++) {

        char buf[NET_IPV4_ADDR_LEN];

        printk("IPv4 address: %s\n",
                net_addr_ntop(AF_INET,
                                &iface->config.ip.ipv4->unicast[i].ipv4,
                                buf, sizeof(buf)));
        printk("Subnet: %s\n",
                net_addr_ntop(AF_INET,
                                &iface->config.ip.ipv4->unicast[i].
   netmask,
                                buf, sizeof(buf)));
        printk("Router: %s\n",
                net_addr_ntop(AF_INET,
                                &iface->config.ip.ipv4->gw,
                                buf, sizeof(buf)));
        }

        k_sem_give(&ipv4_address_obtained);
}
```

For this program two semaphores have been used:

- `wifi_connected`
- `ipv4_adress_obtained`

Both of them are defined already as taken and are release by the events handler functions just after certain conditions are met.

The first function is `handle_wifi_connect_result()` and it's required to handle the results of a WI-Fi connection attempt. It extract the connection status from the event structure and if the connection is successful it signals it and releases the semaphore `wifi_connected`.

The second one, `handle_wifi_disconnect_result()`, handles the case of a disconnection. If the disconnection happens it takes the semaphore released by `wifi_connected`.

The last one is `handle_ipv4_result()`. It handles the event related to the assignment of the IPv4 address. It extracts and prints the IP address, subnet mask, and gateway. It signals that the IPv4 address is been obtained and it free the related semaphore

(ipv4_adress_obtained)

```
static void wifi_mgmt_event_handler(struct net_mgmt_event_callback *cb,
    uint32_t mgmt_event, struct net_if *iface)
{
    switch (mgmt_event)
    {

        case NET_EVENT_WIFI_CONNECT_RESULT:
            handle_wifi_connect_result(cb);
            break;

        case NET_EVENT_WIFI_DISCONNECT_RESULT:
            handle_wifi_disconnect_result(cb);
            break;

        case NET_EVENT_IPV4_ADDR_ADD:
            handle_ipv4_result(iface);
            break;

        default:
            break;
    }
}
```

In this case the function is used to dispatch events to the appropriate function:

- Wi-Fi connection results → `handle_wifi_connect_result()`
- Wi-Fi disconnection results → `handle_wifi_disconnect_result()`
- IPv4 address assignment → `handle_ipv4_result()`

This is realized through a `switch` and the result passed through `mgmt_event`.

```
static struct net_mgmt_event_callback wifi_cb;
static struct net_mgmt_event_callback ipv4_cb;

int main(void)
{
    int sock;

    printk("WiFi Example\nBoard: %s\n", CONFIG_BOARD);

    net_mgmt_init_event_callback(&wifi_cb, wifi_mgmt_event_handler,
                                 NET_EVENT_WIFI_CONNECT_RESULT |
    NET_EVENT_WIFI_DISCONNECT_RESULT);

    net_mgmt_init_event_callback(&ipv4_cb, wifi_mgmt_event_handler,
    NET_EVENT_IPV4_ADDR_ADD);

    net_mgmt_add_event_callback(&wifi_cb);
    net_mgmt_add_event_callback(&ipv4_cb);

    wifi_connect();
    k_sem_take(&wifi_connected, K_FOREVER);
    wifi_status();
    k_sem_take(&ipv4_address_obtained, K_FOREVER);
    printk("Ready...\n\n");

    // Ping Google DNS 4 times
    ping("8.8.8.8", 4);

    printk("\nLooking up IP addresses:\n");
    struct zsock_addrinfo *res;
    nslookup("iot.beyondlogic.org", &res);
    print_addrinfo_results(&res);

    printk("\nConnecting to HTTP Server:\n");
    sock = connect_socket(&res, 80);
    http_get(sock, "iot.beyondlogic.org", "/LoremIpsum.txt");
    zsock_close(sock);

    // Stay connected for 30 seconds, then disconnect.
    //k_sleep(K_SECONDS(30));
    //wifi_disconnect();

    return(0);
}
```

This part is the main entry point of the program and in here the callbacks are initialized.

Then, the program try to connect and wait for the connection to success. after this happens successfully, a few action as a ping or getting a http address are tested to verify it.Because we wanted to verify its ability to maintain the connection for prolunged periods, the line that request the disconnection are commented but in case they are requested they are available and work correctly.

Part V
MQTT Broker

Chapter 9

What is the MQTT

The MQTT (Message Queuing Telemetry Transport) protocol is a lightweight and efficient messaging protocol designed for constrained devices and low-bandwidth, high-latency networks. This protocol is widely used in IoT due to its simplicity, reliability, and low overhead [4]. Due to its efficiency and flexibility, MQTT has been adopted in various domains, including industrial automation, healthcare monitoring, and smart home applications and protocols like the tls ensures secure and robust communication which makes it a preferred choice for IoT.

One of MQTT's key features is its Quality of Service (QoS) levels, which define the reliability of message delivery. The three QoS levels are:

- QoS 0 (at most once)
- QoS 1 (at least once)
- QoS 2 (exactly once)

Offer flexibility in balancing network performance and message reliability [8].

Chapter 10

Connecting to an MQTT broker with Zephyr

In the code the connection to the broker is handled as a thread that, after the device has successfully connected, sends messages that can be received by those connected and subscribed to it. The starting point was the mqtt sample present in the zephyr repo that was adapted and to which a function for the subscription to the broker was added so that it could also recieve messages sent by other applications. This structured initialization ensures that the MQTT client is correctly configured to establish a reliable connection with the broker while maintaining flexibility to support different transport mechanisms and security levels.

10.1 Initialization of the Client

```
static void client_init(struct mqtt_client *client)
{
  mqtt_client_init(client);
  // printk("mqtt_client_init test\n");
  broker_init();
  // printk("broker_init test\n");
  /* MQTT client configuration */
  client->broker = &broker;
  client->evt_cb = mqtt_evt_handler;
  client->client_id.utf8 = (uint8_t *)MQTT_CLIENTID;
  client->client_id.size = strlen(MQTT_CLIENTID);
  client->password = MQTT_PASS;
  client->user_name = MQTT_USER;
  client->protocol_version = MQTT_VERSION_3_1_1;

  /* MQTT buffers configuration */
  client->rx_buf = rx_buffer;
  client->rx_buf_size = sizeof(rx_buffer);
  client->tx_buf = tx_buffer;
  client->tx_buf_size = sizeof(tx_buffer);
  /* MQTT transport configuration */
#if defined(CONFIG_MQTT_LIB_TLS)
#if defined(CONFIG_MQTT_LIB_WEBSOCKET)
  client->transport.type = MQTT_TRANSPORT_SECURE_WEBSOCKET;
#else
  client->transport.type = MQTT_TRANSPORT_SECURE;
#endif

  struct mqtt_sec_config *tls_config = &client->transport.tls.config;

  tls_config->peer_verify = TLS_PEER_VERIFY_REQUIRED;
  tls_config->cipher_list = NULL;
  tls_config->sec_tag_list = m_sec_tags;
  tls_config->sec_tag_count = ARRAY_SIZE(m_sec_tags);
#if defined(MBEDTLS_X509_CRT_PARSE_C) || defined(
   CONFIG_NET_SOCKETS_OFFLOAD)
  tls_config->hostname = TLS_SNI_HOSTNAME;
#else
  tls_config->hostname = NULL;
#endif

#else
#if defined(CONFIG_MQTT_LIB_WEBSOCKET)
  client->transport.type = MQTT_TRANSPORT_NON_SECURE_WEBSOCKET;
#else
  client->transport.type = MQTT_TRANSPORT_NON_SECURE;
#endif
#endif

#if defined(CONFIG_MQTT_LIB_WEBSOCKET)
  client->transport.websocket.config.host = SERVER_ADDR;
  client->transport.websocket.config.url = "/mqtt";
  client->transport.websocket.config.tmp_buf = temp_ws_rx_buf;
  client->transport.websocket.config.tmp_buf_len =
            sizeof(temp_ws_rx_buf);
  client->transport.websocket.timeout = 50 * MSEC_PER_SEC;
#endif

#if defined(CONFIG_SOCKS)
  mqtt_client_set_proxy(client, &socks5_proxy,
            socks5_proxy.sa_family == AF_INET ?
            sizeof(struct sockaddr_in) :
            sizeof(struct sockaddr_in6));
#endif
}
```

The `client_init` function is responsible for configuring and initializing an MQTT client. As for the Wi-Fi, also in this case an event handle and callback functions are recalled and used to handle the various situations in which could occur.

10.2 The thread

```
static int start_app(void)
{
  int r = 0, i = 0;

  while (!CONFIG_NET_SAMPLE_APP_MAX_CONNECTIONS ||
         i++ < CONFIG_NET_SAMPLE_APP_MAX_CONNECTIONS) {
    r = publisher();

    if (!CONFIG_NET_SAMPLE_APP_MAX_CONNECTIONS) {
      k_sleep(K_MSEC(5000));
    }
  }

  return r;
}

#if defined(CONFIG_USERSPACE)
#define STACK_SIZE 2048

#if defined(CONFIG_NET_TC_THREAD_COOPERATIVE)
#define THREAD_PRIORITY K_PRIO_COOP(CONFIG_NUM_COOP_PRIORITIES - 1)
#else
#define THREAD_PRIORITY K_PRIO_PREEMPT(8)
#endif

K_THREAD_DEFINE(app_thread, STACK_SIZE,
    start_app, NULL, NULL, NULL,
    THREAD_PRIORITY, K_USER, -1);

static K_HEAP_DEFINE(app_mem_pool, 1024 * 2);
#endif
```

In this portion of the code the thread that handles the thread is defined. It can be seen that first we define the function and then the function is connected to a thread through the zephyr macro `K_THREAD_DEFINE{}`. The request to start the thread is created inside the main:

```
k_thread_heap_assign(app_thread, app_mem_pool);
k_thread_start(app_thread);
k_thread_join(app_thread, K_FOREVER);
```

Here, the thread is assigned its heap memory and becomes operative.

10.3 Publisher Function

```
  while (i++ < CONFIG_NET_SAMPLE_APP_MAX_ITERATIONS && connected) {
    r = -1;

    rc = mqtt_ping(&client_ctx);
    PRINT_RESULT("mqtt_ping", rc);
    SUCCESS_OR_BREAK(rc);
    // printk("i'm here in the while");
    rc = process_mqtt_and_sleep(&client_ctx, APP_CONNECT_TIMEOUT_MS);
    SUCCESS_OR_BREAK(rc);

    rc = publish(&client_ctx, MQTT_QOS_0_AT_MOST_ONCE);
    PRINT_RESULT("mqtt_publish", rc);
    SUCCESS_OR_BREAK(rc);

    rc = process_mqtt_and_sleep(&client_ctx, APP_CONNECT_TIMEOUT_MS);
    SUCCESS_OR_BREAK(rc);

    rc = publish(&client_ctx, MQTT_QOS_1_AT_LEAST_ONCE);
    PRINT_RESULT("mqtt_publish", rc);
    SUCCESS_OR_BREAK(rc);

    rc = process_mqtt_and_sleep(&client_ctx, APP_CONNECT_TIMEOUT_MS);
    SUCCESS_OR_BREAK(rc);

    rc = publish(&client_ctx, MQTT_QOS_2_EXACTLY_ONCE);
    PRINT_RESULT("mqtt_publish", rc);
    SUCCESS_OR_BREAK(rc);

    rc = process_mqtt_and_sleep(&client_ctx, APP_CONNECT_TIMEOUT_MS);
    SUCCESS_OR_BREAK(rc);
  }
```

In this portion of the publisher the type of messages to send was selected and then sent to the broker

10.4 Subscriber Function

```
int app_mqtt_subscribe(struct mqtt_client *client)
{
  int rc;
  struct mqtt_topic sub_topics[] = {
    {
      .topic = {
        .utf8 = MQTT_TOPIC,
        .size = strlen(sub_topics->topic.utf8)
      },
      .qos = IS_ENABLED(CONFIG_NET_SAMPLE_MQTT_QOS_0_AT_MOST_ONCE) ? 0 :
        (IS_ENABLED(CONFIG_NET_SAMPLE_MQTT_QOS_1_AT_LEAST_ONCE) ? 1 : 2)
    }
  };
  const struct mqtt_subscription_list sub_list = {
    .list = sub_topics,
    .list_count = ARRAY_SIZE(sub_topics),
    .message_id = 5841u
  };

  LOG_INF("Subscribing to %d topic(s)", sub_list.list_count);

  rc = mqtt_subscribe(client, &sub_list);
  if (rc != 0) {
    LOG_ERR("MQTT Subscribe failed [%d]", rc);
  }
  else
  {
    LOG_INF("MQTT Subscribe success [%d]", rc);
  }

  return rc;
}
```

The `app_mqtt_subscribe` function is responsible for subscribing an MQTT client to a specified topic and it enables message reception from the broker by defining the subscription parameters invoking the `mqtt_subscribe` API.

The QoS level is dynamically determined based on compile-time configuration options (`CONFIG_NET_SAMPLE_MQTT_QOS_0_AT_MOST_ONCE`, `CONFIG_NET_SAMPLE_MQTT_QOS_1_AT_LEAST_ONCE`), allowing flexibility in message delivery reliability.

Part VI

IMU implementation

Chapter 11

The SparkFun Library for ICM20948

Having the ICM20948 not implemented in Zephyr brings about the need of finding a valid library to use and adapt for our necessity. The library of SparkFun for the IMU was selected. This is a library for the IMU ICM20948 realized for Arduino available at on its GitHub at [12]. Below the structure of the library is reported:

```
SparkFun_ICM-20948_ArduinoLibrary
 |
 ├─ src
 |   |
 |   ├─ util
 |   |   |
 |   |   ├─ ICM20948_eMD_nucleo_1.0
 |   |   |
 |   |   ├─ eMD-SmartMotion-ICM20948-1.1.0-MP
 |   |   |
 |   |   ├─ ICM_20948_C.c
 |   |   |
 |   |   ├─ ICM_20948_C.h
 |   |   |
 |   |   ├─ ICM_20948_DMP.h
 |   |   |
 |   |   └─ various header file...
 |   |
 |   ├─ ICM_20948.cpp
 |   |
 |   └─ ICM_20948.h
 |
 └─ other files...
```

The library provides functionalities for initializing the ICM-20948 sensor, reading accelerometer, gyroscope, and magnetometer data, and configuring various settings as the

DMP. This APIs are written in C++ but Zephyr uses the C language (it could support C++ but some complication and problem could arose if enabled) and also some other functionality that will be required. However, it must be acknowledged that these library backbone is already written in C, so only the function that will then be recalled in the main have to be adapted.

The I2C communication are treated in a different way in Arduino than Zephyr. In Arduino it is initialized through the library `wire` with the `Wire.begin()` and `Wire.read()` commands while Zephyr does it through its drivers and the devicetree and so an overlay file was created where the I2C port of the board where defined.

One of the challenges, while converting the `ICM20948.cpp` file into `ICM20948.c` file, is to replace the class method functions with standalone functions and also converting private member variables into elements of a `struct`. Moreover the function for the initialization of the sensor was completely rewritten.

11.1 ICM_20948.c

I will now analyze a few function that have been written in this file.

11.1.1 ICM_20948_init()

```c
ICM_20948_Status_e ICM_20948_init(ICM_20948_Device_t *_ICM, struct
    device *_i2c_dev, uint8_t _imu_i2c_addr)
{
    ICM_20948_Status_e ret = ICM_20948_Stat_Err;

    ICM = _ICM;

    if (NULL != _i2c_dev)
    {
        i2c_dev = _i2c_dev;
    }

    i2c_configure(i2c_dev, I2C_SPEED_SET(I2C_SPEED_STANDARD));

    if (!i2c_dev) {
        LOG_ERR("I2C device not valid.");
        return ICM_20948_Stat_Err;
    }

    if (0 != _imu_i2c_addr)
    {
        imu_i2c_addr = _imu_i2c_addr;
    }

  if(ICM_20948_init_struct(ICM) == ICM_20948_Stat_Ok)
```

```
        LOG_INF("ICM_20948_init_struct success\n");

    if(ICM_20948_link_serif(ICM, &_Serif)== ICM_20948_Stat_Ok)
        LOG_INF("ICM_20948_link_serif success\n");

    while (ICM_20948_check_id(ICM) != ICM_20948_Stat_Ok)
    {
      LOG_INF("whoami does not match. Halting... ");
      k_sleep(K_SECONDS(1));
    }

    // Here we are doing a SW reset to make sure the device starts in a
      known state
    ICM_20948_sw_reset(ICM);

      k_sleep(K_MSEC(500));

    // Now wake the sensor up
    ICM_20948_sleep(ICM, false);
    ICM_20948_low_power(ICM, false);

    startupMagnetometer(false);

      // Set Gyro and Accelerometer to a particular sample mode
    ICM_20948_set_sample_mode(ICM, (ICM_20948_InternalSensorID_bm)(
      ICM_20948_Internal_Acc | ICM_20948_Internal_Gyr),
      ICM_20948_Sample_Mode_Continuous); // optiona:
      ICM_20948_Sample_Mode_Continuous. ICM_20948_Sample_Mode_Cycled

    ICM_20948_set_full_scale(ICM, (ICM_20948_InternalSensorID_bm)(
      ICM_20948_Internal_Acc | ICM_20948_Internal_Gyr), ICM_20948_fss);

    // Set up DLPF configuration
    ICM_20948_dlpcfg_t myDLPcfg;
    myDLPcfg.a = acc_d111bw4_n136bw;
    myDLPcfg.g = gyr_d119bw5_n154bw3;
    ICM_20948_set_dlpf_cfg(ICM, (ICM_20948_InternalSensorID_bm)(
      ICM_20948_Internal_Acc | ICM_20948_Internal_Gyr), myDLPcfg);

    ICM_20948_smplrt_t mySMPLRTcfg;
    mySMPLRTcfg.a = 1;
    mySMPLRTcfg.g = 1;
    ICM_20948_set_sample_rate(ICM, (ICM_20948_InternalSensorID_bm)(
      ICM_20948_Internal_Acc | ICM_20948_Internal_Gyr), mySMPLRTcfg);

    // Choose whether or not to use DLPF
    ICM_20948_enable_dlpf(ICM, ICM_20948_Internal_Acc, false);
    ICM_20948_enable_dlpf(ICM, ICM_20948_Internal_Gyr, false);

    #if defined(ICM_20948_USE_DMP)
      ICM->_dmp_firmware_available = true;
      printf("Initialized dmp_firmware_available = true\n");
#else
```

```
    ICM->_dmp_firmware_available = false; // Initialize
   dmp_firmware_available
    printf("Initialized dmp_firmware_available = false\n");
#endif

  ret = ICM_20948_Stat_Ok;

  return ret;
}
```

In the first part, the I2C communication information are saved in the device structure `_ICM`. This value are taken directly from the devicetree through the functions reported below:

```
#define I2C_PORT          DEVICE_DT_GET(DT_NODELABEL(i2c0)) // I2C port 0
#define I2C_PORT_1        DEVICE_DT_GET(DT_NODELABEL(i2c1)) // I2C port 1
```

```
#define ICM20948_I2C_ADDR     0x68
```

This line are taken from the `config.h` file where various macro are defined.
In the rest of the code there are function which initialize or configure the three sensors of the IMU. In the end its also implemented an if with the objective of initialize the DMP but,it i snot been use in this thesis. The reason for this decision is due to the fact that the DMP firmware is private and so there are no possibilities in seeing what there is inside of it. However the function of the DMP have also been converted form C++ to C so that it can be initialized and used if needed.

statupMagnetometer()

This function, used in the `ICM_20948_init()`, is used to activate and initialize the magnetometer. While seeing the code you could think that `startupMagnetometer(false)` means that it isn't going to be used but if true it will activate a "minimal" configuration while setting this bool variable at false will let you configure it yourself inside this function as it can see below:

```
ICM_20948_Status_e startupMagnetometer(bool minimal)
{
  ICM_20948_Status_e retval = ICM_20948_Stat_Ok;

  i2cMasterPassthrough(false);
  i2cMasterEnable(true);

  resetMag();

  ... // Other code

  if (minimal) // Return now if minimal is true
  {
    printf("Magnetomiter minal success\n");

    return status;
  }

  //Set up magnetometer
  AK09916_CNTL2_Reg_t reg;
  reg.MODE = AK09916_mode_cont_100hz;
  reg.reserved_0 = 0;
  retval = writeMag(AK09916_REG_CNTL2, (uint8_t *)&reg);
  if (retval != ICM_20948_Stat_Ok)
  {
    status = retval;
    return status;
  }

  retval = i2cControllerConfigurePeripheral(0, MAG_AK09916_I2C_ADDR,
   AK09916_REG_ST1, 9, true, true, false, false, false, 0);
  if (retval != ICM_20948_Stat_Ok)
  {
    status = retval;
    return status;
  }

  return status;
}
```

11.2 Raw data Reading and Conversion

The raw data obtained from the sensor need to be converted and the following function do so:

```
ICM_20948_AGMT_t getAGMT(void)
{
  status = ICM_20948_get_agmt(ICM, &agmt);

  return agmt;
}

float getMagUT(int16_t axis_val)
{
  return (((float)axis_val) * 0.15);
}

float getAccMG(int16_t axis_val, uint8_t fss)
{
  switch (fss)
  {
  case 0:
    return (((float)axis_val) / 16.384);
    break;
  case 1:
    return (((float)axis_val) / 8.192);
    break;
  case 2:
    return (((float)axis_val) / 4.096);
    break;
  case 3:
    return (((float)axis_val) / 2.048);
    break;
  default:
    return 0;
    break;
  }
}

float getGyrDPS(int16_t axis_val, uint8_t fss)
{
  switch (fss)
  {
  case 0:
    return (((float)axis_val) / 131);
    break;
  case 1:
    return (((float)axis_val) / 65.5);
    break;
  case 2:
    return (((float)axis_val) / 32.8);
    break;
  case 3:
    return (((float)axis_val) / 16.4);
    break;
  default:
    return 0;
    break;
  }
}
```

This function convert the value so that they will acquire a physical meaning. The unit of measure used are the mg (1 mg $\approx$ 9.81 mm/s^2) for the accelerometer, rad/s for the gyroscope and μT for the magnetometer. This operation must be repeated for all axes of each sensor.

The raw readings are obtained using `ICM20948_get_agmt()` and stored in the `struct agmt`. Assigning this data to the functions mentioned above returns values in the corresponding measurement units. However, this data is not yet ready for direct use, as further conversions may be required depending on specific needs.

11.3 Magnetometer Calibration

Among the three sensors, particular attention must be paid to the magnetometer. Since this sensor measures the Earth's magnetic field, it is highly susceptible to external interferences and distortions caused by nearby ferromagnetic materials or electronic components. Therefore, proper calibration is essential to obtain accurate and reliable readings.

There are two type of calibrations:

- **Hard-Iron Calibration**: it is caused by nearby permanent magnetic fields generated by nearby ferromagnetic materials.
- **Soft-Iron Calibration**: for take in consideration the distortion of Earth magnetic field caused by material that do not generate a magnetic field them self but possess an high magnetic permeability.

Both this calibration are crucial when using a magnetometer and without proper calibration magnetometer readings could lead to significant errors and inaccurate results. The result of the calibration are reported below:

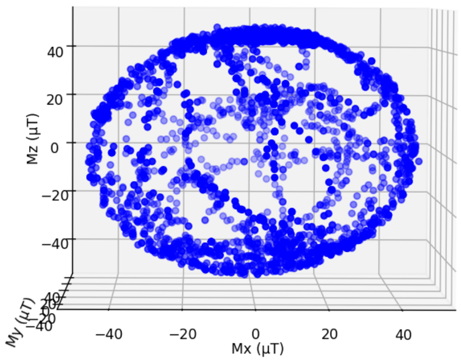


Figure 11.1. calibration of the magnetometer successfully eliminates the hard and soft iron disturbances

As can been seen the value on the axis of the radius is around 40 μT and the sphere results well centered.

11.3.1 Hard-Iron Calibration

The effect of the hard-iron calibration consist in a constant offset in the measured magnetic field which cause a displacement of the magnetic field data from the center of the expected spherical distribution.

For the correction of this offset, the magnetometer data is corrected while rotating the sensor in as many direction as possible. Then the calculated offset are subtracted from the raw magnetometer readings.

11.3.2 Soft-Iron Calibration

Unlike hard-iron calibration, soft-iron distortions affect the shape of the response by turning the expected spherical distribution of the readings in an ellipse.

To compensate this distortion a scale value for each axis is applied to reshape the distorted elliptical data back into a sphere. As for hard-iron calibration, also in this case a lot of data must be collected while rotating the sensor in various direction.

11.3.3 Calibration code

The hard-iron calibration is realized by taking into account the maximum and minimum values between the acquired data for each axis and, next, by simply computing the offset as follows:

$$M^i_{max} = \max(M^i), \quad m^i_{min} = \min(M^i), \quad i \in x, y, z$$

$$\text{Offset}^i = \frac{M^i_{max} - m^i_{min}}{2}, \quad i \in x, y, z$$

For the soft-iron calibration once again the maximum and minimum were used this time to calculate the axes radius.

$$\text{Radius}^i = Mmax^i - M^i_{min}, \quad i \in x, y, z$$

Then the maximum radius was found and then the scale factors for the magnetometer where found as follow:

$$\text{Radius}_{max} = \max(\text{Radius}^x, \text{Radius}^y, \text{Radius}^z)$$

$$\text{scale}^i = \frac{\text{Radius}_{max}}{\text{Radius}^i}, \quad i \in x, y, z$$

Transposing these formulae in C the result was the following:

```
// Searching for Maimum and minimum values of the magnetic field data

prt = 0;
while (prt < NSAMPLS) {
    if(prt == 0) {
        max_m[0] = Mdata[prt*3];
        max_m[1] = Mdata[prt*3 + 1];
        max_m[2] = Mdata[prt*3 + 2];
        min_m[0] = Mdata[prt*3];
        min_m[1] = Mdata[prt*3 + 1];
        min_m[2] = Mdata[prt*3 + 2];
    }
    if(prt < (NSAMPLES-1)) {
        if(Mdata[prt*3] > max_m[0]) max_m[0] = Mdata[prt*3];
        if(Mdata[prt*3 + 1] > max_m[1]) max_m[1] = Mdata[prt*3 + 1];
        if(Mdata[prt*3 + 2] > max_m[2]) max_m[2] = Mdata[prt*3 + 2];
        if(Mdata[prt*3] < min_m[0]) min_m[0] = Mdata[prt*3];
        if(Mdata[prt*3 + 1] < min_m[1]) min_m[1] = Mdata[prt*3 + 1];
        if(Mdata[prt*3 + 2] < min_m[2]) min_m[2] = Mdata[prt*3 + 2];
    }
    prt++;
}

// Computation of Hard Iron Offset

MagnOffsetX = (max_m[0] + min_m[0])/2;
MagnOffsetY = (max_m[1] + min_m[1])/2;
MagnOffsetZ = (max_m[2] + min_m[2])/2;

// Computation of  Soft Iron Scaling Factors

for(int i = 0; i < 3; i++) {
    axis_rad[i] = max_m[i] - min_m[i];
}
float max_radius = fmax(axis_rad[0], fmax(axis_rad[1], axis_rad[2]));

scale_x = max_radius / axis_rad[0];
scale_y = max_radius / axis_rad[1];
scale_z = max_radius / axis_rad[2];
```

Because an offset is present to distort the data, setting the starting value of the $max_m[i]$ and $min_m[i]$ to a predefined value (0 for example) could create a false maximum or minimum. For this reason I decided to assign the first read as maximum and minimum and then confront the following readings (all saved in the vector $Mdata[]$).
In addition, the number of data points collected for calibration can be easily adjusted by modifying the previously defined macro `NSAMPLES`.

The offset and the scaling will then be considered while collecting data as follow:

$$M^i = (M^i - \text{Offset}^i) * \text{scale}^i, \quad i \in x, y, z0$$

By applying these calibrations, the orientation angles obtained have been highly improved especially, as will be later seen, the Yaw angle which strongly depends on the magnetometer for its calculation.

11.4 Data Acquisition

In the code there are two moments when the data are acquired. The first time during calibration where just the magnetometer data are retrieved:

```
printf("Magn. Hard and Soft Iron Calibration...\n");
  k_msleep(500);
  while (prt < 2000) {
    if (ICM_20948_get_agmt(&myICM, &agmt) == ICM_20948_Stat_Ok)
    {
      Mdata[prt*3] = getMagUT(agmt.mag.axes.x);
      Mdata[prt*3 + 1] = getMagUT(agmt.mag.axes.y);
      Mdata[prt*3 + 2] = getMagUT(agmt.mag.axes.z);
    } else {
      printf("Error reading data\n");
      if(ICM_20948_get_agmt(&myICM, &agmt) != ICM_20948_Stat_Ok)
        printf("sensor 1 not getting data\n");
    }
    if(prt%100 == 0) printf("n. data collected: %d\n", prt);
    prt++;
    k_msleep(SLEEP_TIME_MS);
  }
  printf("Data collected\n");
```

In the code above are also reported some debugging print for the cases in which the connection with the sensor is lost or corrupted. Moreover, a print was added as a reference on the number of data collected while rotating the sensor. Once the calibration is completed it restart to collect data this time from all the three sensor:

```
while (1) {
    if (ICM_20948_get_agmt(&myICM, &agmt) == ICM_20948_Stat_Ok)
    {
        ax = /*Acc[i*3]*/getAccMG(agmt.acc.axes.x, myfss.a);
        ay = /*Acc[i*3+1]*/ getAccMG(agmt.acc.axes.y, myfss.a);
        az = /*Acc[i*3+2]*/getAccMG(agmt.acc.axes.z, myfss.a);
        ax = ax * 9.81f / 1000.0f;
        ay = ay * 9.81f / 1000.0f;
        az = az * 9.81f / 1000.0f;
        // r = atan(ay/ax)/M_PI*180.0f;
        // p = atan(-ax/sqrt(pow(ay, 2) + pow(az, 2)))/M_PI*180.0f;

        gx = /*Gyr[i*3]*/ getGyrDPS(agmt.gyr.axes.x, myfss.g);
        gy = /*Gyr[i*3+1]*/ getGyrDPS(agmt.gyr.axes.y, myfss.g);
        gz = /*Gyr[i*3+2]*/ getGyrDPS(agmt.gyr.axes.z, myfss.g);
        // t_curr = k_uptime_get();
        // r = r + gx * (t_curr - t_pre) / 1000.0f;
        //p = p + gy * (t_curr - t_pre) / 1000.0f;
        //y = y + gz * (t_curr - t_pre) / 1000.0f;
        gx = gx * (M_PI / 180.0f);
        gy = gy * (M_PI / 180.0f);
        gz = gz * (M_PI / 180.0f);
        // t_pre = t_curr;
        printf("Gyr: %f, %f, %f\n", gx, gy, gz);

        mx = /*Mag[i*3]*/ (getMagUT(agmt.mag.axes.x) - MagnOffsetX) *
    scale_x;//*/;
        my = /*Mag[i*3+1]*/ (getMagUT(agmt.mag.axes.y) - MagnOffsetY) *
    scale_y;//*/;
        mz = /*Mag[i*3+2]*/ (getMagUT(agmt.mag.axes.z) - MagnOffsetZ) *
    scale_z;//*/;
        //bx = mx*cos(p) + mz*sin(p);
        //by = mx*sin(r)*sin(p) + my*cos(r) - mz*sin(r)*cos(p);
        //y = atan2(-by, bx);
        printf("Acc: %5.3f, %5.3f, %5.3f  ", ax, ay, az);
        printf("Gyr: %5.3f, %5.3f, %5.3f  ", gx, gy, gz);
        printf("Mag: %5.3f, %5.3f, %5.3f\n", mx, my, mz);
        #ifdef MADGWICK_AHRS
        MadgwickAHRSupdate(ax, ay, az, mx, my, mz, gx, gy, gz);
        #endif
        #ifdef MAHONY_AHRS
        MahonyAHRSupdate(ax, ay, az, mx, -my, mz, gx, gy, gz, 1);
        #endif
        //printf("INTEGRATED: Roll: %f, Pitch: %f, Yaw: %f\n", r, p, y);

    }
    else
    {
        printf("Error reading data\n");
        if(ICM_20948_data_ready(&myICM) != ICM_20948_Stat_Ok)
            printf("No data\n");
    }

}
```

As you can see the magnetometer data are immediately corrected by subtracting the offset and multiplying the result for the axis scale.
Moreover, the gyroscope and accelerometer are converted to different unit of measure:

- the accelerometer is converted form milli g to m/s^2
- the gyroscope from rad/s to $°/s$

Also in this case a print for debugging as been inserted.

11.4.1 Angle Computation

Then as can be seen have also been implemented methods to use the converted data of the single sensor for obtaining the angle. However, in the case of the magnetometer it can only be used to compute the Yaw angle while with the accelerometer it can only be used to compute the Roll and Pitch angles.

Roll ans Pitch from Accelerometer

As said before we can use the accelerometer to calculate just two of the three angles: the Roll and the Pitch angles. The reason for this is that the Yaw that does not depend on gravity and so there is no solution to obtain it using just the accelerometer.

The formulas used where:

$$\text{Roll} = arctan(\frac{a_y}{a_x})$$

$$\text{Pitch} = arcsin(\frac{a_x}{g}) = arcsin(\frac{-a_x}{\sqrt{a_x^2 + a_y^2 + a_z^2}})$$

Yaw from magnetometer

On the other hand, the magnetometer can only be use to calculate the Yaw and gyroscope or accelerometer will be required to compute the other two angles.

To correctly calculate the yaw angle Roll and Pitch must be taken in consideration so that the magnetic field component can be corrected in case the board is turned.
For this reason, also the accelerometer (Roll and Pitch obtained from its readings) has to be used.

$$B_x^{'} = B_x * cos(\theta) + B_z * sin(\theta)$$
$$B_y^{'} = B_x * sin(\phi) * sin(\theta) + B_y * cos(\phi) - B_z * sin(\phi) * cos(\theta)$$

From these results the Yaw can be obtained through:

$$\text{Yaw} = arctan2(-B_y^{'}, B_x^{'})$$

Angles from Gyroscope

The gyroscope let you instead compute all the three angles through integration. To achieve it the function `k_uptime_get()` from the zephyr kernel is used first to initialized `t_pre` and then at each cycle to obtain the current time `t_curr`. Then, the following calculation is computed:

$$\text{Roll}(t_{curr}) = \text{Roll}(t_{pre}) + G^x * (t_{curr} - t_{pre})$$
$$\text{Pitch}(t_{curr}) = \text{Pitch}(t_{pre}) + G^y * (t_{curr} - t_{pre})$$
$$\text{Yaw}(t_{curr}) = \text{Yaw}(t_{pre}) + G^z * (t_{curr} - t_{pre})$$

11.5 Sensor Fusion

To increase the precision and also solve the drift problem a possible solution is represented by sensor fusion algorithms.

The sensor fusion is a process that take the readings of more sensors and analyzing them together it's able to provide a precise estimation of the angles. Because each sensor as it's strengths and weaknesses and sensor fusion is a valid solution to overcome those limitation

This process was included in the filter that was selected and utilized for this thesis. The filter taken in consideration was the Mahony filter described by R. Mahony et al. in [11].

11.5.1 Mahony Filter

This filter is a lightweight and simple one which makes it a popular choice for embedded systems for which this properties are very important. The code used was obtain converting in C the code from the AHRS library for Arduino found at [3]. The code is the following:

```
#include "MahonyAHRS.h"
#include <math.h>
#include <stdbool.h>

#define sampleFreq  100.0f      // sample frequency in Hz
#define twoKpDef  (2.0f * 8.7f) // 2 * proportional gain
#define twoKiDef  (2.0f * 0.02f)  // 2 * integral gain
#define M_PI   3.14159265358979323846264338327950288

volatile float twoKp = twoKpDef;  // 2 * proportional gain (Kp)
volatile float twoKi = twoKiDef;  // 2 * integral gain (Ki)
volatile float q0[2] = {1.0f, 1.0f}, q1[2] = {0.0f, 0.0f}, q2[2] = {0.0f
    , 0.0f}, q3[2] = {0.0f, 0.0f};
volatile float integralFBx[2] = {0.0f, 0.0f},  integralFBy[2] = {0.0f,
    0.0f}, integralFBz[2] = {0.0f, 0.0f};

static float roll, pitch, yaw;

//Inverse square root function
float invSqrt(float x) {
  float halfx = 0.5f * x;
  float y = x;
  long i = *(long*)&y;
  i = 0x5f3759df - (i>>1);
  y = *(float*)&i;
  y = y * (1.5f - (halfx * y * y));
  return y;
}
```

In this part of the code the global variable that either are recalled by more function or need to be saved for the next computation are defined as well as the function for the inverse square root. Unlike in the reference case, here some variable are defined as vector and not simple float. These is because in the end the program should be able to read and compute the angles of more than one sensor. this structure is so easily adaptable in case the need of adding other sensors arose. In that case, just q_0, q_1, q_2, q_3, $integralFBx$, $integralFBy$, $integralFBz$ have to be modify by adding the number of sensors.

Here the proportional gain k_p and the integral gain K_i are also defined. This two gain are used to tuning the filter giving more or less importance to certain sensors or actions of the sensor fusion process.

- **Proportional Gain**: it decides how much aggressive the filter will correcter orientation errors but it can introduce noise if it's set too high. Increasing this value will means an higher response from the filter. Lowering it the response will be slower but smoother.

- **Integral Gain**: this is used to compensate the gyro bias. If the value is too high it could cause instability while if too low it could be not able to correct the bias.

```
void MahonyAHRSupdate(float ax, float ay, float az, float mx, float my,
    float mz, float gx, float gy, float gz, int idx) {
  float recipNorm;
    float q0q0, q0q1, q0q2, q0q3, q1q1, q1q2, q1q3, q2q2, q2q3, q3q3;
  float hx, hy, bx, bz;
  float halfvx, halfvy, halfvz, halfwx, halfwy, halfwz;
  float halfex, halfey, halfez;
  float qa, qb, qc;

  // Use IMU algorithm if magnetometer measurement invalid (avoids NaN
    in magnetometer normalisation)
  if((mx == 0.0f) && (my == 0.0f) && (mz == 0.0f)) {
    MahonyAHRSupdateIMU(gx, gy, gz, ax, ay, az, idx);
    return;
  }

  // Compute feedback only if accelerometer measurement valid (avoids
    NaN in accelerometer normalisation)
  if(!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f))) {

    // Normalise accelerometer measurement
    recipNorm = invSqrt(ax * ax + ay * ay + az * az);
    ax *= recipNorm;
    ay *= recipNorm;
    az *= recipNorm;

    // Normalise magnetometer measurement
    recipNorm = invSqrt(mx * mx + my * my + mz * mz);
    mx *= recipNorm;
    my *= recipNorm;
    mz *= recipNorm;

        // Auxiliary variables to avoid repeated arithmetic
        q0q0 = q0[idx] * q0[idx];
    q0q1 = q0[idx] * q1[idx];
    q0q2 = q0[idx] * q2[idx];
    q0q3 = q0[idx] * q3[idx];
    q1q1 = q1[idx] * q1[idx];
    q1q2 = q1[idx] * q2[idx];
    q1q3 = q1[idx] * q3[idx];
    q2q2 = q2[idx] * q2[idx];
    q2q3 = q2[idx] * q3[idx];
    q3q3 = q3[idx] * q3[idx];

        // Reference direction of Earth's magnetic field
        hx = 2.0f * (mx * (0.5f - q2q2 - q3q3) + my * (q1q2 - q0q3) + mz
    * (q1q3 + q0q2));
        hy = 2.0f * (mx * (q1q2 + q0q3) + my * (0.5f - q1q1 - q3q3) + mz
    * (q2q3 - q0q1));
    bx = sqrt(hx * hx + hy * hy);
```

```
      bz = 2.0f * (mx * (q1q3 - q0q2) + my * (q2q3 + q0q1) + mz * (0.5f
    - q1q1 - q2q2));

    // Estimated direction of gravity and magnetic field
    halfvx = q1q3 - q0q2;
    halfvy = q0q1 + q2q3;
    halfvz = q0q0 - 0.5f + q3q3;
    halfwx = bx * (0.5f - q2q2 - q3q3) + bz * (q1q3 - q0q2);
    halfwy = bx * (q1q2 - q0q3) + bz * (q0q1 + q2q3);
    halfwz = bx * (q0q2 + q1q3) + bz * (0.5f - q1q1 - q2q2);

    // Error is sum of cross product between estimated direction and
    measured direction of field vectors
    halfex = (ay * halfvz - az * halfvy) + (my * halfwz - mz * halfwy);
    halfey = (az * halfvx - ax * halfvz) + (mz * halfwx - mx * halfwz);
    halfez = (ax * halfvy - ay * halfvx) + (mx * halfwy - my * halfwx);

    // Compute and apply integral feedback if enabled
    if(twoKi > 0.0f) {
      integralFBx[idx] += twoKi * halfex * (1.0f / sampleFreq);
      integralFBy[idx] += twoKi * halfey * (1.0f / sampleFreq);
      integralFBz[idx] += twoKi * halfez * (1.0f / sampleFreq);
      gx += integralFBx[idx];
      gy += integralFBy[idx];
      gz += integralFBz[idx];
    }
    else {
      integralFBx[idx] = 0.0f;
      integralFBy[idx] = 0.0f;
      integralFBz[idx] = 0.0f;
    }

    // Apply proportional feedback
    gx += twoKp * halfex;
    gy += twoKp * halfey;
    gz += twoKp * halfez;
  }
```

Here, the function responsible for the sensor fusion process and angle computation is defined. After initializing some local variables the another function that will be seen later on is recalled and used just in case thee magnetometer is not been used. To do so it check if the reading from it are all 0.0. After this check, the same happens to the accelerometer readings. In case they are equal to 0.0 the angle will not be computed: on the other hand, if the conditions are met the program start to calculate to execute the required operations.

After it normalize the data from accelerometer and magnetometer, it compute the square and pairwise product of the quaternion obtained with the previous data. This is followed by an estimation of the gravity and magnetic field and computes the error on it.

The integral feedback is then computed an, as said before, it depends on K_i. If the gain is set to 0 then the program will not compute it. Then also the K_p is applied.

```
  // Integrate rate of change of quaternion
  gx *= (0.5f * (1.0f / sampleFreq));
  gy *= (0.5f * (1.0f / sampleFreq));
  gz *= (0.5f * (1.0f / sampleFreq));
  qa = q0[idx];
  qb = q1[idx];
  qc = q2[idx];
  q0[idx] += (-qb * gx - qc * gy - q3[idx] * gz);
  q1[idx] += (qa * gx + qc * gz - q3[idx] * gy);
  q2[idx] += (qa * gy - qb * gz + q3[idx] * gx);
  q3[idx] += (qa * gz + qb * gy - qc * gx);

  // Normalise quaternion
  recipNorm = invSqrt(q0[idx] * q0[idx] + q1[idx] * q1[idx] + q2[idx] *
    q2[idx] + q3[idx] * q3[idx]);
  q0[idx] *= recipNorm;
  q1[idx] *= recipNorm;
  q2[idx] *= recipNorm;
  q3[idx] *= recipNorm;

  // roll (x-axis rotation)
  float sinr_cosp = 2 * (q0[idx] * q1[idx] + q2[idx] * q3[idx]);
  float cosr_cosp = 1 - 2 * (q1[idx] * q1[idx] + q2[idx] * q2[idx]);
  roll = atan2(sinr_cosp, cosr_cosp)*180/M_PI;

  // pitch (y-axis rotation)
  float sinp = sqrt(1 + 2 * (q0[idx] * q2[idx] - q1[idx] * q3[idx]));
  float cosp = sqrt(1 - 2 * (q0[idx] * q2[idx] - q1[idx] * q3[idx]));
  pitch = (2 * atan2(sinp, cosp) - M_PI / 2)*180/M_PI;

  // yaw (z-axis rotation)
  float siny_cosp = 2 * (q0[idx] * q3[idx] + q1[idx] * q2[idx]);
  float cosy_cosp = 1 - 2 * (q2[idx] * q2[idx] + q3[idx] * q3[idx]);
  yaw = atan2(siny_cosp, cosy_cosp)*180/M_PI;
  if (yaw > 180.0) yaw -= 360.0;
  if (yaw < -180.0) yaw += 360.0;

  printf("FILTER sensor %d: Roll: %f, Pitch: %f, Yaw: %f\n", idx, roll,
    pitch, yaw);

}
```

In this portion of the function the new quaternions are computed followed by the computation of the Euler angles that are then printed for been able to check them and save them for a more in depth analysis.

```
void MahonyAHRSupdateIMU(float gx, float gy, float gz, float ax, float
    ay, float az, int idx) {
  float recipNorm;
  float halfvx, halfvy, halfvz;
float halfex, halfey, halfez;
float qa, qb, qc;

// Compute feedback only if accelerometer measurement valid (avoids
 NaN in accelerometer normalisation)
if(!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f))) {

  // Normalise accelerometer measurement
  recipNorm = invSqrt(ax * ax + ay * ay + az * az);
  ax *= recipNorm;
  ay *= recipNorm;
  az *= recipNorm;

  // Estimated direction of gravity and vector perpendicular to
 magnetic flux
  halfvx = q1[idx] * q3[idx] - q0[idx] * q2[idx];
  halfvy = q0[idx] * q1[idx] + q2[idx] * q3[idx];
  halfvz = q0[idx] * q0[idx] - 0.5f + q3[idx] * q3[idx];

  // Error is sum of cross product between estimated and measured
 direction of gravity
  halfex = (ay * halfvz - az * halfvy);
  halfey = (az * halfvx - ax * halfvz);
  halfez = (ax * halfvy - ay * halfvx);

  // Compute and apply integral feedback if enabled
  if(twoKi > 0.0f) {
    integralFBx[idx] += twoKi * halfex * (1.0f / sampleFreq);
    integralFBy[idx] += twoKi * halfey * (1.0f / sampleFreq);
    integralFBz[idx] += twoKi * halfez * (1.0f / sampleFreq);
    gx += integralFBx[idx]; // apply integral feedback
    gy += integralFBy[idx];
    gz += integralFBz[idx];
  }
  else {
    integralFBx[idx] = 0.0f;
    integralFBy[idx] = 0.0f;
    integralFBz[idx] = 0.0f;
  }

  // Apply proportional feedback
  gx += twoKp * halfex;
  gy += twoKp * halfey;
  gz += twoKp * halfez;
}

// Integrate rate of change of quaternion
gx *= (0.5f * (1.0f / sampleFreq));
gy *= (0.5f * (1.0f / sampleFreq));
gz *= (0.5f * (1.0f / sampleFreq));
qa = q0[idx];
qb = q1[idx];
qc = q2[idx];
```

```
    q0[idx] += (-qb * gx - qc * gy - q3[idx] * gz);
    q1[idx] += (qa * gx + qc * gz - q3[idx] * gy);
    q2[idx] += (qa * gy - qb * gz + q3[idx] * gx);
    q3[idx] += (qa * gz + qb * gy - qc * gx);

    // Normalise quaternion
    recipNorm = invSqrt(q0[idx] * q0[idx] + q1[idx] * q1[idx] + q2[idx] *
      q2[idx] + q3[idx] * q3[idx]);
    q0[idx] *= recipNorm;
    q1[idx] *= recipNorm;
    q2[idx] *= recipNorm;
    q3[idx] *= recipNorm;

    float sinr_cosp = 2 * (q0[idx] * q1[idx] + q2[idx] * q3[idx]);
    float cosr_cosp = 1 - 2 * (q1[idx] * q1[idx] + q2[idx] * q2[idx]);
    roll = atan2(sinr_cosp, cosr_cosp)*180/M_PI;

    // pitch (y-axis rotation)
    float sinp = sqrt(1 + 2 * (q0[idx] * q2[idx] - q1[idx] * q3[idx]));
    float cosp = sqrt(1 - 2 * (q0[idx] * q2[idx] - q1[idx] * q3[idx]));
    pitch = (2 * atan2(sinp, cosp) - M_PI / 2)*180/M_PI;

    // yaw (z-axis rotation)
    float siny_cosp = 2 * (q0[idx] * q3[idx] + q1[idx] * q2[idx]);
    float cosy_cosp = 1 - 2 * (q2[idx] * q2[idx] + q3[idx] * q3[idx]);
    yaw = atan2(siny_cosp, cosy_cosp)*180/M_PI;

    printf("Roll: %f, Pitch: %f, Yaw: %f\n", roll, pitch, yaw);
}
```

In the end we have the function with the algorithm that is used in case the magnetometer is disable or isn't sending data.

Chapter 12

Algorithm Outputs

The simple angles computation methods reported above have brought us the following results that will now be analyzed. The graphics are been obtained by using a Python program and passing to it a file where all the readings had been saved. More version of this program where realized to adapt to the various cases (Raw Data, Gyroscope integral action, ...).

12.1 Raw Data

Here below the raw data from the 3 sensor of the IMU are reported singularly. Each graphics reports the readings of each axis (x, y, z) of a sensor (accelerometer, gyroscope, magnetometer) with the number of data taken on the abscissa. Here the data are reported in mg. As can been seen there always a single axis which reports the value of 1000 mg which means 1 g. The rumors, which is higher in proximity of the beginning and the end of movements has to be attributed to human error because the sensor was moved manually and also to the fact that while moving the sensor it is subject to an acceleration which of course is detected and counted in the data. Still, the sensor appears to be quite precise in reading the value of the acceleration granting us a great base for the calculation of the angles.

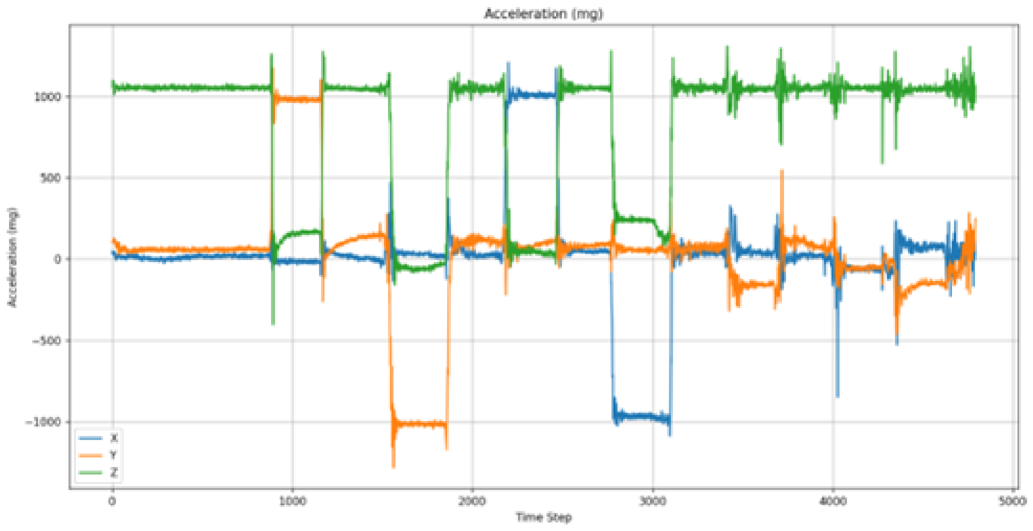


Figure 12.1. Accelerometer raw data

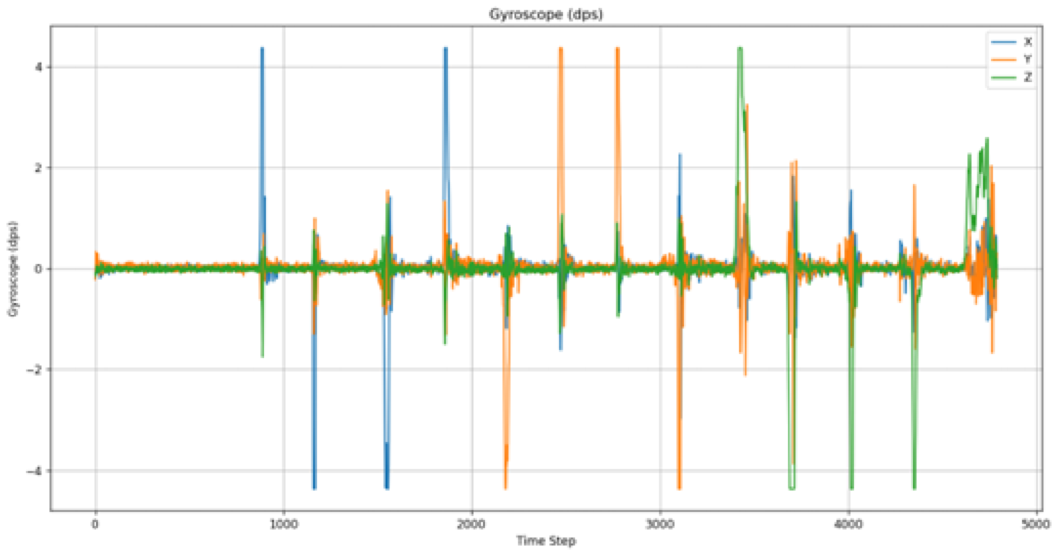


Figure 12.2. Gyroscope raw data

On this second graphic are reported instead the gyroscope readings. as can be seen all the readings are around 0 while the sensor is not moving and are subjected to high speed while rotating. Even more than in the previous graphic the human error can be seen in proximity of the movements.

By putting the two graphics together a coincidence between the accelerometer variation and the gyroscope once was individuated which prove the coherence of this readings.

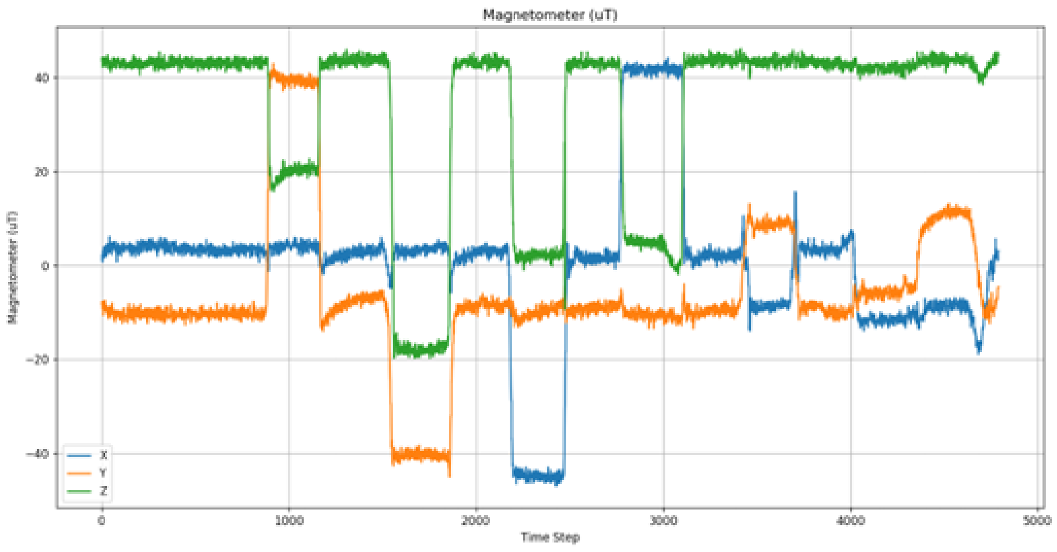


Figure 12.3. Magnetometer raw data

Last but not least, the magnetometer readings are reported. These readings are taken after the calibration The values that i expect in Turin, Italy where this thesis was developed where [1]:

- vertical component: around 42000 nT
- Horizontal component: around 22800 nT

As can been seen in the graphic (where the magnetic field is reported in μT)the vertical component (green) report exactly the expected value. On the other hand the value read by the magnetometer on the x and y axis is a little bit lower than the expected one at 14 μT. However, despite the lower value then the one reported online, the value on both x and y axes was coherent when turning and switching between thee two.

12.2 Acceleration and Magnetometer, angles computation

This method didn't bring the expected results. The outputs were particularly distorted by the fast movements of the IMU. ANd didn't give us results that coul dbe confronted with the others obtained through the rmaining methods.

12.3 Gyroscope with Integral Action

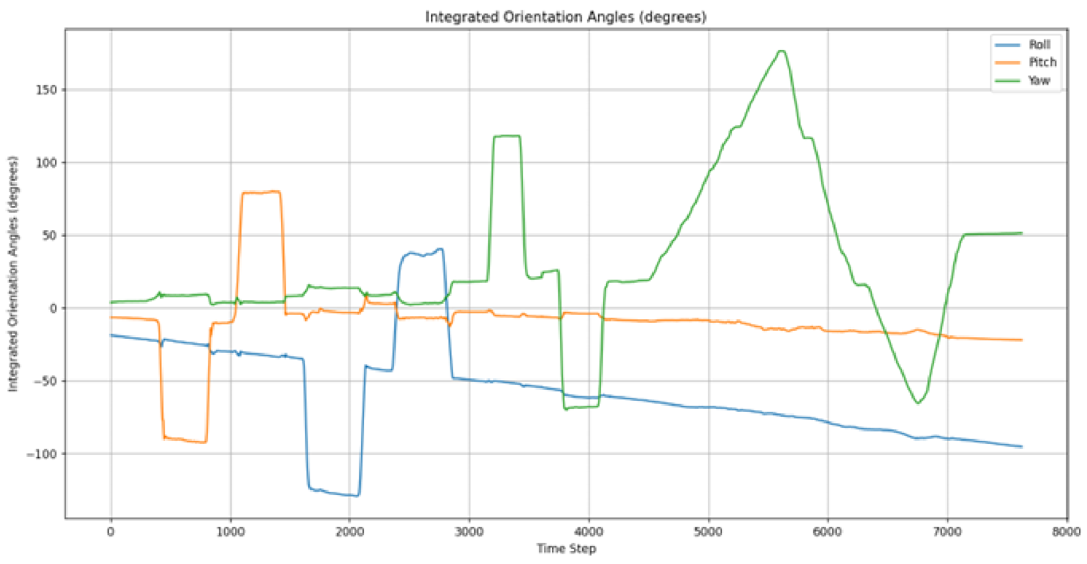


Figure 12.4. Angles computation from gyroscope readings

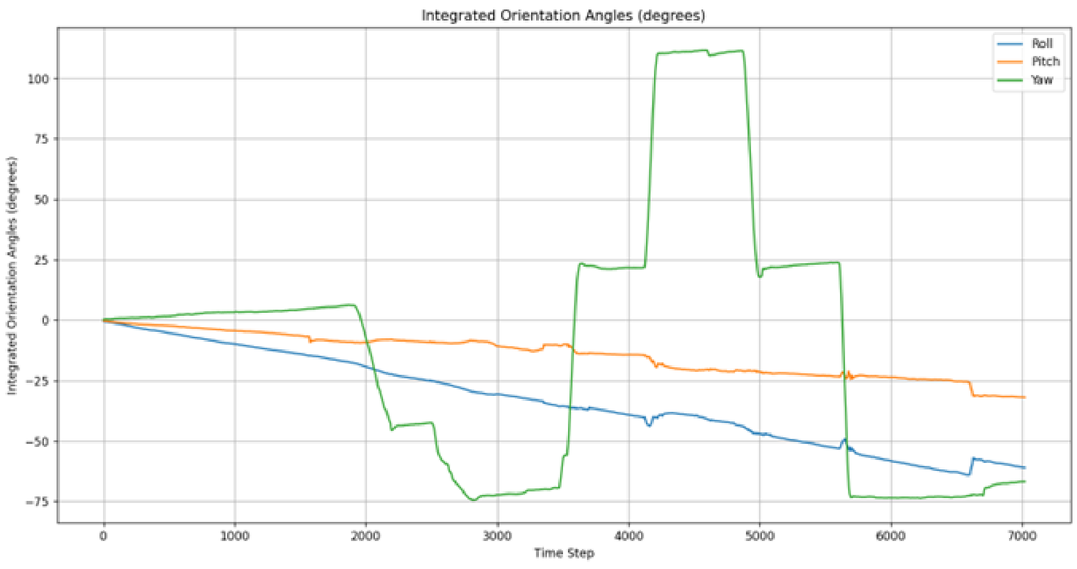


Figure 12.5. Angles computation from gyroscope readings

Above,the angles computes through integration action on the gyroscope have been reported and analyzed.

In both cases a drift bias is found which for the x and the y is negative while for the z axis is less evident and positive. This can be seen especially in Figure 9.5 where around 2000 samples are taken at the beginning remaining still but the z still grow regularly while for the other two in both cases a slow negative drift can be noticed.

12.4 Mahony Filter Outputs

Now a few outputs for the mahony filter will be presented:

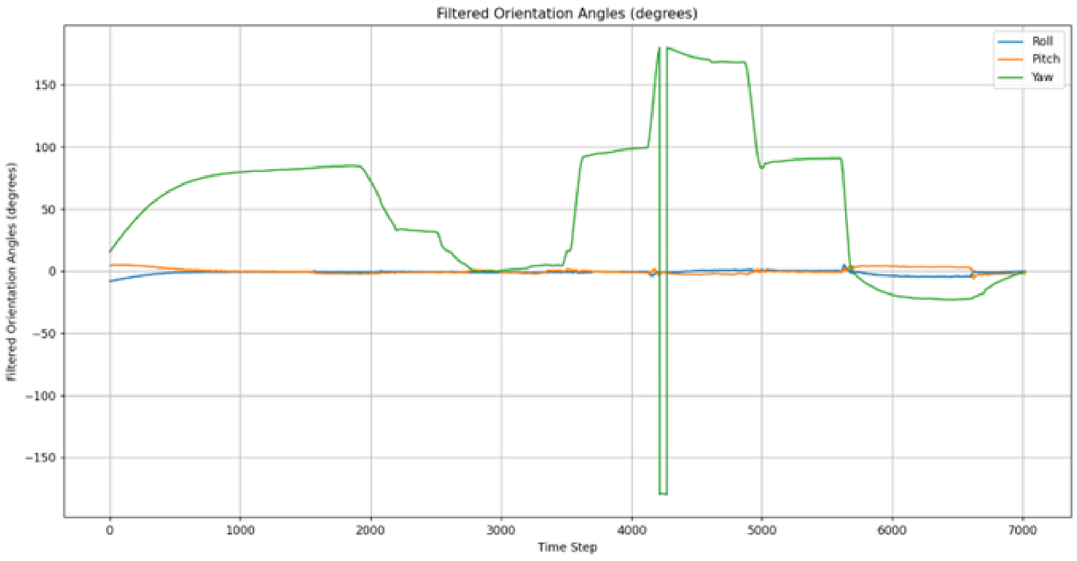


Figure 12.6. Angles computation from sensor fusion through a Mahony filter

In htis figure the data have been taken through a slow rotation of the sensor and then with rapid rotations of around 90° each to test the speed response from the filter.

In the end the sensor is been returned the sensor to the initial position. to test also the presence of a drift.

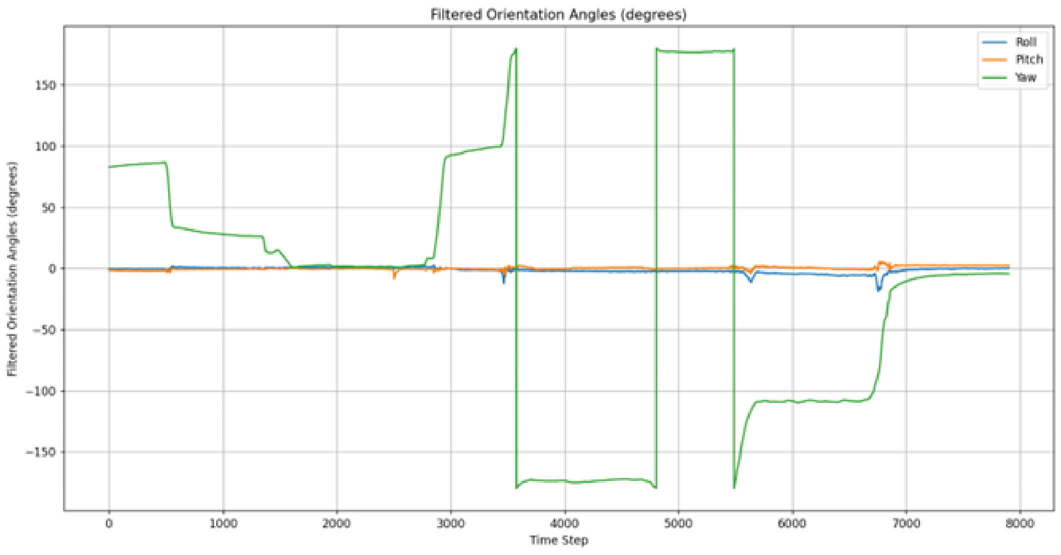


Figure 12.7. Angles computation from sensor fusion through a Mahony filter

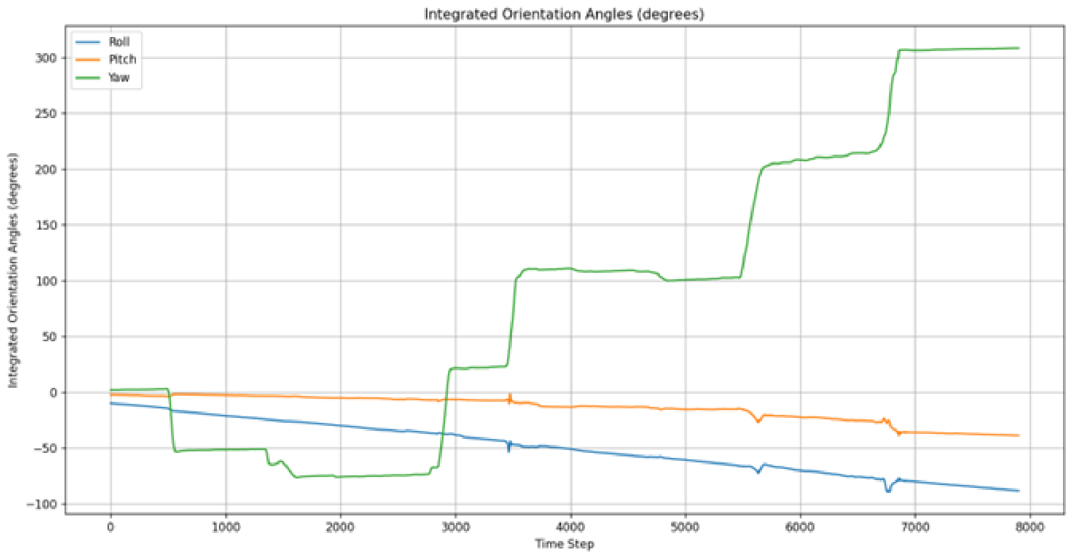


Figure 12.8. Angles computation from gyroscope readings, same readings as Figure 10.7

In this case we have taken both the gyroscope and the Mahony filter data to verify that both methods returned the same output.

In this case when we started taking data the sensor was not pointing north and so the

Mahony doesn't start at 0° but at 90° then I tried to turn the sensor of around 90° each time and completing a 360° rotation. As can been seen the sensor shows a good behavior while the gyroscope, after performing well in the first rotation, is not able to produce a good estimate of the angles.

Part VII

Conclusions

Chapter 13

Conclusions

This thesis explored the development of firmware for a wearable embedded system designed to monitor neurological pathologies using an ESP32 microcontroller and the Zephyr RTOS. The system successfully integrates the ICM-20948 IMU, establishing communication via I2C to acquire motion data. Additionally, the firmware enables Wi-Fi connectivity and supports MQTT communication, allowing the device to transmit sensor data to a broker for further processing. These achievements demonstrate the feasibility of using this embedded system for real-time motion tracking and wireless data transmission in a rehabilitation context.
Despite these accomplishments, the current implementation presents a critical limitation: the program crashes after successfully transmitting a few initial messages to the MQTT broker. This issue, likely related to either the interaction between Zephyr's timing mechanisms and the ESP32's internal components or the a problem with the interaction between the threads of the project and those that are created by Zephyr. This issues requires further investigation and resolution because ensuring the stability and reliability of the system is essential for its practical deployment in medical and rehabilitation applications.

13.1 Future Work

Future work should focus on resolving this instability to achieve continuous data transmission without unexpected failures. Additionally, the development of a dedicated application is necessary to receive, store, and further analyze the transmitted movement data. Such an application would enable advanced processing techniques, including filtering, motion reconstruction, and real-time visualization, providing valuable insights for healthcare professionals and patients undergoing rehabilitation without the need of a direct interaction between them.
By addressing these challenges and expanding the system's capabilities, this project can contribute to the development of a robust and efficient tool for neurological monitoring. With further refinement, the proposed system has the potential to support medical research, improve rehabilitation strategies, and enhance patient care through real-time

motion tracking and wireless data analysis.

Bibliography

[1] Noaa national centers for environmental information, geomagnetism calculator. URL https://www.ngdc.noaa.gov/geomag/calculators/magcalc.shtml?#igrfwmm.

[2] IEEE 802.11. Ieee 802.11, vlan standards. URL https://www.ieee802.org/11/.

[3] Adafruit. Adafruit_ahrs library on github. URL https://github.com/adafruit/Adafruit_AHRS.

[4] A. Banks and R. Gupta. *MQTT Version 3.1.1 Plus Errata 01*. 2015. URL https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/.

[5] Richard Barry and The FreeRTOS Team. *Mastering the FreeRTOS™ Real-Time Kernel*. Amazon Web Services, 1.1.0 edition, 2023. URL https://www.FreeRTOS.org.

[6] craigpeacock. Zephyr wi-fi sample repository. URL https://github.com/craigpeacock/Zephyr_WiFi.

[7] Espressif. *ESP32-WROOM-32 datasheet*. 2023. URL https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf.

[8] J. Dollimore G. F. Coulouris and T. Kindberg. *Distributed Systems: Concepts and Design*. 2012.

[9] IEEE. The evolution of wi-fi technology and standards. URL https://standards.ieee.org/beyond-standards/the-evolution-of-wi-fi-technology-and-standards/.

[10] TDK invenSense. *World's Lowest Power 9-Axis MEMS MotionTracking™ Device*. 1.6 edition, 2024. URL https://invensense.tdk.com/download-pdf/icm-20948-datasheet/.

[11] Robert Mahony, Tarek Hamel, and Jean-Michel Pflimlin. Nonlinear complementary filters on the special orthogonal group. *IEEE Transactions on Automatic Control*, 53 (5):1203–1218, 2008. doi: 10.1109/TAC.2008.923738.

[12] SparkFun. Sparkfun_icm-20948_arduinolibrary, github repository. URL SparkFun_ICM-20948_ArduinoLibrary.

[13] ThreadX. Threadx rtos documentation. URL https://github.com/eclipse-threadx/rtos-docs/blob/main/rtos-docs/threadx/chapter1.md.

[14] Zephyr. Zephyr project getting started guide, . URL https://docs.zephyrproject.org/latest/develop/getting_started/index.html.

[15] Zephyr. Zephyr project documentation, . URL https://docs.zephyrproject.org/latest/index.html.

[16] Zephyr. Zephyr project repository on github, . URL https://github.com/zephyrproject-rtos/zephyr.