Development of an embedded system for tagging high impact zones in a car accident

André Lopes

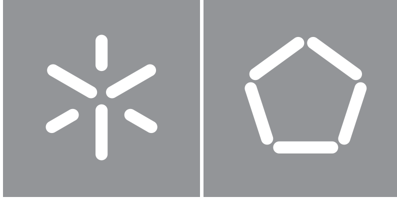
UMinho | 2023

Universidade do Minho
Escola de Engenharia

André Filipe Machado Lopes

Development of an embedded system for tagging high impact zones in a car accident

março de 2023



Universidade do Minho
Escola de Engenharia

André Filipe Machado Lopes

Development of an embedded system for tagging high impact zones in a car accident

Master's Dissertation

Master's in Industrial Eletronics and Computers Engineering

Work supervised by

Rui Pedro Oliveira Machado

June, 2023

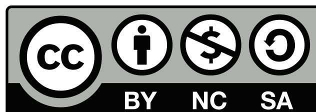
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositóriUM of Universidade do Minho.

License granted to the users of this work



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International
CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en

"If you can't explain it simply, you don't understand it well enough."

Albert Einstein

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador, Professor Doutor Rui Machado, pelo apoio durante toda a dissertação e por todo o conhecimento transmitido. Gostaria também de agradecer ao professor João Carvalho por me ter ajudado e guiado nas questões mais técnicas e pelo seu apoio nesta dissertação. Gostaria de agradecer também ao professor Jorge Cabral pela oportunidade e ajuda neste projeto, bem como a todos os elementos que passaram pelo laboratório utilizado e que de alguma forma me ajudaram a ultrapassar os obstáculos.

Um agradecimento especial a todos os meus amigos que me ajudaram a ultrapassar os momentos mais dificeis da dissertação e pelo suporte e acompanhamento durante este ultimo ano de percurso académico. Gostaria também de agradecer aos meus companheiros de curso por me acompanharem durante todo este percurso e por toda ajuda e motivação.

Por fim e mais importante, quero deixar um agradecimento especial aos meus pais pela motivação que me deram e por acreditarem em mim durante todo o curso.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Abstract

Development of an embedded system for tagging high impact zones in a car accident

In the constantly evolving automotive industry, there is a growing focus on the development of advanced safety mechanisms, such as sensors, algorithms, and devices for collision detection, to create a new generation of highly safe cars. One key safety element in vehicles is the airbag control unit, which detects car crashes and triggers the airbag deployment. However, there is currently a gap in identifying the most severe impact areas in a car seat, which can minimize the diagnostic time by identifying potentially life-threatening injuries. In this research work, we aim to address this gap by developing an embedded system prototype that can detect car collisions and acquire data for real-time analysis and future accident reconstruction. The device, which will be placed in a car seat, is designed to detect high impacts and store data in case of a collision. The system also samples data from acquisition sensors, applies filters, and performs a collision algorithm in a typical operation. The main contribution of this study is the development of an embedded system that can improve the diagnostic process for paramedics and provide more accurate data for accident reconstruction. The research focuses on the reliable use of sensor data, high-impact detection, and data storage. The prototype is tested using a shaker to confirm the accuracy and reliability of the overall system. Also, the performance of the system is analyzed under high-acceleration conditions.

Keywords: automotive industry, safety, collision detection, car seat, embedded system, car collisions, accident reconstruction, high impacts, sensor data, prototype testing, reliability, performance.

Resumo

Desenvolvimento de um sistema embebido para a marcação de zonas de alto impacto num acidente de carro

Na constante evolução da indústria automóvel, existe um crescente foco no desenvolvimento de mecanismos de segurança avançados, tais como, sensores, algoritmos e dispositivos de detecção de colisão. Atualmente, o mecanismo de segurança mais utilizado para a deteção de acidentes é o airbag, que permite uma rápida identificação da colisão através do uso de diversos sensores posicionados em locais estratégicos. Contudo, existe actualmente uma lacuna na identificação das áreas que sofreram um maior impacto num assento automóvel. O preenchimento desta lacuna poderá minimizar o tempo de diagnóstico efetuado pelos paramédicos, através da identificação da zona dos ferimentos. Neste estudo, o objectivo é abordar esta lacuna através do desenvolvimento de um sistema embebido protótipo que detete colisões de automóveis e adquira os dados para uma análise em tempo real e para uma possível reconstrução do acidente. Sendo assim, este dispositivo terá de recolher amostras de dados dos sensores de aquisição, aplicar os filtros necessários e executar o algoritmo de deteção.

Esta investigação centra-se na utilização fiável dos dados dos sensores, detecção de alto impacto, e armazenamento de dados. O protótipo é testado utilizando um shaker para confirmar a exactidão e fiabilidade do sistema no seu todo. Além disso, o desempenho do sistema é analisado em condições de alta aceleração.

Palavras-chave: Industria automóvel, segurança, deteção de colisões, assento automóvel, sistema embebido, reconstrução de acidentes, impactos elevados, protótipo, fiabilidade, desempenho.

Contents

List of Figures x

List of Tables xii

Listings xiii

Acronyms xiv

1 Introduction 1

1.1 Contextualization and Motivation . . . 1
1.2 Objectives . . . 2
1.3 Methodology . . . 2
1.4 Document Structure . . . 3

2 Background and State of the Art 4

2.1 Basic Concepts . . . 4
2.1.1 Embedded System . . . 4
2.1.2 Real-Time System . . . 5
2.1.3 Operating System . . . 6
2.1.4 Multitasking . . . 7
2.1.5 Scheduling . . . 8
2.1.6 Inter-Process Communication . . . 8
2.2 Crash detection sensors . . . 9
2.2.1 Accelerometers . . . 10
2.2.2 IMU . . . 10
2.3 Crash detection algorithms . . . 11
2.3.1 ”Total and Partial Energies”, Diller . . . 12
2.3.2 ”Power-Rate Method”, Allen . . . 13
2.3.3 ”A Predictive Algorithm”, Gioutsos . . . 13
2.3.4 ”Adjustable Velocity Threshold”, Mattes . . . 13

2.3.5 ”Multiple Evaluation and Expertise Algorithms”, Diller . . . 13
2.3.6 ”Multiple Evaluation Circuits and Time Window”, Eigler . . . 14
2.3.7 ”Power Spectrum of Acceleration Signal in a Frequency Range”, Tohbaru . . 14
2.3.8 ”Deployment Method Based on Occupant Displacement and Crash Severity”, Cashler and Kelle . . . 14
2.3.9 ”Method and Apparatus for Sensing a Vehicle CrashCondition Using Velocity Enhanced Acceleration Metrics”, McIver . . . 15
2.3.10 ”Method and Apparatus for Crash Sensing Using Anticipatory Sensor Inputs”, Kosiak . . . 15
2.4 Related Work . . . 16
2.4.1 Airbag . . . 16
2.4.2 High-impact scenarios . . . 19
2.4.3 Bosch Peripheral Acceleration Sensor (PAS) . . . 21
2.4.4 Literature review . . . 22
2.5 Conclusions . . . 23

3 System Specification **25**
3.1 Requirements . . . 25
3.1.1 Functional Requirements: . . . 25
3.1.2 Non-Functional Requirements: . . . 25
3.2 System Overview . . . 26
3.3 Hardware Specification . . . 26
3.3.1 Sensors . . . 27
3.3.2 CPU . . . 28
3.4 Software Specification . . . 29
3.5 Signal Processing . . . 30
3.5.1 Kalman Filter . . . 31
3.6 Conclusions . . . 34

4 Design **35**
4.1 Hardware Considerations . . . 35
4.1.1 LDO (Low-Dropout) . . . 36
4.1.2 ADXL357 . . . 37
4.1.3 MicroSD Card . . . 38
4.1.4 MPU-6881 . . . 39
4.1.5 CPU . . . 40
4.1.6 Micro USB Connector . . . 41

4.1.7 CP2102N 42
4.2 Software Considerations 43
4.2.1 Task Overview 43
4.2.2 Drivers 45
4.2.3 Kalman 47
4.2.4 FreeRTOS Tasks 55

5 Implementation **61**
5.1 Hardware Schematic 61
5.1.1 Main sheet 61
5.1.2 Micro USB 63
5.2 Hardware Layout 65
5.2.1 Layout Process 65
5.3 Software 67
5.3.1 Drivers 68
5.3.2 Tasks 75

6 Tests and Results **81**
6.1 Shaker Tests 81
6.2 Collision Test 87

7 Conclusions **90**
7.1 Future Work 91

References **92**

Appendices **95**

A Appendix 1 **95**
A.1 SPI Driver 95
A.2 SDCard Driver 96
A.3 Matlab Kalman 97
A.4 Python Code to present the results 102

List of Figures

1 Example of an embedded system 5
2 Kernel Role in a computer system 6
3 Scheduling States in FreeRTOS 8
4 Airbag System Overview 17
5 Airbag Control Unit 17
6 Airbag Sensors 18
7 Airbag signal Processing 18
8 Car Coordinates 20
9 Peripheral acceleration sensor 21

10 System Overview 26
11 ADXL357 Accelerometer 27
12 ESP32-Mini-1 28
13 Software Architecture 29
14 Signal Processing Overview 30
15 Kalman Simple Overview 31
16 Kalman Detailed Overview 32
17 Kalman Equations Overview 33

18 PCB Build Process 36
19 PCB Overview 36
20 LDO schematic 37
21 ADXL357 schematic 38
22 MicroSD Card schematic 39
23 MicroSD Pin Number Order 39
24 MPU-6881 schematic 40
25 CPU schematic 41
26 MicroUSB connector schematic 42
27 CP2102N schematic 42

28 Programming logic schematic . . . 43
29 Tasks Overview . . . 44
30 Tasks Synchronization . . . 45
31 SPI Format ADXL357 . . . 46
32 ADXL357 Sensitivity . . . 46
33 MPU-6881 SPI Format . . . 47
34 MPU-6881 sensitivity . . . 47
35 vAccelerometer task . . . 56
36 vIMU task . . . 57
37 vCollisionDetection task . . . 58
38 vSDCard task . . . 59
39 vMain . . . 60

40 Main schematic . . . 62
41 Altium Project Organization . . . 63
42 MicroUSB main schematic . . . 64
43 Layer Stack . . . 65
44 PCB layout LDO . . . 65
45 PCB layout ESP32 . . . 66
46 PCB layout layers . . . 67
47 3D PCB . . . 67
48 ADXL357 acceleration registers organization . . . 70

49 Shaker Components . . . 82
50 3D holder model . . . 82
51 PCB Mounting Configuration . . . 83
52 Shaker profile frequency ranges . . . 83
53 Shaker profile . . . 84
54 Kalman Shaker X . . . 84
55 Kalman Shaker Y . . . 85
56 Kalman Shaker Z . . . 85
57 PCB Kalman Shaker Y . . . 86
58 PCB Kalman Shaker X . . . 86
59 PCB Kalman Shaker Z . . . 87
60 Box enclusure . . . 88
61 3D Box model . . . 88
62 Collision Test . . . 89

List of Tables

1 Accelerometer Grade and Typical Application Area 10
2 IMU Grade and Typical Application Area 11
3 Highlights and comparisons of a selected list of sensing algorithms 12
4 Human Tolerance Limits 21
5 MicroSD pinout 39

Listings

1	ADXL Driver main functions	68
2	ADXL Driver read function	70
3	MPU Driver main functions	70
4	MPU Driver read function	71
5	Matrix Module main functions	72
6	Kalman module init function	73
7	Kalman module predict function	73
8	Kalman module setup function	74
9	Kalman module update function	75
10	vMain Task	76
11	vAccelerometer Task	77
12	vIMU Task	78
13	vColisionDetection Task	78
14	vSDCard Task	80
15	SPI Driver main header file functions	95
16	SPI Driver main functions	95
17	SD Card Driver mount function	96
18	SD Card Driver init and close functions	97
19	Matlab Kalman Implementation	97
20	Python script to visualize results	102

Acronyms

ADC Analog to Digital Converter

API Application Programming Interface

BOM Bill of Materials

CAN Controller Area Network

CPU Central Processing Unit

DMA Direct Memory Access

DRC Design Rule Checking

FAT File Allocation Table

FIFO First-In First-Out

GPOS General-Purpose Operating System

GPS Global Positioning System

IMU Inertial Measurement Unit

LDO Low-Dropout

MEMS Microelectrome-chanical systems

MISO Master In Slave Out

MOSI Master Out Slave In

MSB Most Significant Bit

PCB Printed Circuit Board

RAM Random Access Memory

RTC Real-Time Clock

RTOS Real-Time Operating System

SNR Signal to Noise Ratio

UART Universal Asynchronous Receiver/Transmitter

Chapter 1

Introduction

This introductory chapter aims to provide an overview of the scientific work presented in the following chapters. This chapter will begin by contextualizing the research within the automotive industry and discussing the motivation for creating a device that can detect car impacts. The main goals of the investigation, and the intermediate steps needed to achieve those goals, will be explained after. The methodology used during the dissertation, including the research design, research subjects, data collection and analysis methods, and any statistical tests, will be discussed. Finally, the document structure will be presented to provide a clear overview of the dissertation work.

1.1 Contextualization and Motivation

With the evolution of the automotive industry, the development of mechanisms to accomplish vehicle safety is receiving more and more attention. Thus, currently, several projects aim for the latest techniques, including new sensors, algorithms or devices to detect a crash, which will lead to a new generation of highly safe cars. The airbag is one of the most relevant safety elements in a vehicle. Detecting a car crash and triggering the airbag is part of the airbag control unit's job. By understanding how the control unit operates, researchers can design and create devices able to detect and respond to several types of collisions. Some safety vehicle devices reduce the collision impact preventing and saving lives. However, there is a missing gap in identifying the most severe impact areas in a car seat, which can reduce the diagnostic time. This reduction can yield important clues to life-threatening injuries [1]. Since there is a growing demand to perform accident reconstruction using sensor data, a device placed in a car seat can also provide more information about the crash if the data can be saved [2].

The missing gap is the motivation for this dissertation. To fill this research gap becomes relevant to

build a system with detection abilities that can reduce the paramedic's diagnostic time. Also, the device data can be combined with other car elements, such as GPS (Global Positioning System), to obtain a more precise vehicle location or associated with the airbag system for a second opinion. There are a lot of functionalities that can be added, including a data-storing capability. This functionality allows the driver to protect himself against possible scenarios where he is being accused and can also be used for accident testing by industry companies [3].

1.2 Objectives

The goal of this dissertation is the development of an embedded system prototype that performs a data acquisition and a real-time detection in case of car collision. This device will be present in a car seat and should detect if the car seat received a higher impact in the accident. In that case, the embedded system should store the data for a further collision reconstruction or investigation. In a normal operation, the system should sample data from acquisition sensors, apply filters to that data and perform a collision algorithm. Also, this device should be powered by the car and be reliable enough to not be destroyed during the car accident. In order to achieve this goal, there are some objectives to be done:

- Reliable sensor data usage
- High impact detection
- Save information from the crash

1.3 Methodology

The following section outlines the methodology for achieving the defined goals of this dissertation. The process begins with an analysis to identify the most appropriate sensors and algorithms for collision detection based on existing devices such as airbags and previous studies with similar objectives. The system components, including sensors, microcontrollers, and other elements, are then specified to begin the design phase, during which schematics and software diagrams are created. The development of the PCB (Printed Circuit Board) during the implementation phase involves creating a layout, generating Gerber files, creating a Bill of Materials, and soldering it. This is followed by the preparation of firmware and the implementation of all previously designed software. The filtering stage is integrated into the system during this phase, with the optional use of a Kalman filter to improve accuracy and reliability. The performance of the implemented embedded system is tested using a shaker test to confirm the accuracy and reliability of the sensors and the overall system. Finally, the system's overall performance under high-acceleration

conditions is analyzed and evaluated, including the performance of the sensors, the effectiveness of the filtering stage, and the accuracy of the accident detection algorithm.

1.4 Document Structure

The document structure of this dissertation will consist of several chapters. The first chapter, Introduction, will provide an overview of the research, including the motivation for the study and the main goals and objectives. The second chapter, Background and State-of-art will provide background information on the automotive industry and the current state of the art in collision detection equipment. This chapter will also introduce the necessary basic concepts and theories to understand the development of the dissertation. The third chapter, System Specification, will describe the requirements and specifications for the system, including the choice of sensors, microcontroller, and other components. This chapter will also provide a detailed system overview and describe the algorithms and methods used for data processing. The fourth chapter, Design, will describe the design of the system, including the PCB and the software. The fifth chapter, Implementation, will describe the steps taken to implement the system, including the firmware and any hardware modifications. The sixth chapter, Tests, will present the results of the tests conducted to evaluate the performance of the system, including the shaker test to evaluate the filtered data performance. The final chapter, Conclusions, will summarize the main points of the dissertation and provide conclusions and implications for future research in this area. Overall, this document structure will provide a clear and organized presentation of the research and its findings.

Chapter 2

Background and State of the Art

This chapter presents an overview of the technologies and concepts to be addressed during this dissertation. Since the goal is to develop an embedded system for data acquisition, some fundaments about these systems will be included for understanding how they are developed. Because this device will act in high-impact scenarios, a characterization of this environment is necessary to make decisions that may influence the system's performance. Furthermore, this chapter will address what technologies, types of sensors and algorithms are used in accident detection. Finally, a description of similar devices on the market, similar studies and their functionalities will be presented in this chapter.

2.1 Basic Concepts

The development of an embedded system, including the hardware and software, is the core of this dissertation, so it becomes necessary to present some concepts for its understanding. Therefore, the definition and characteristics of this type of system will be described. In an embedded system, an operating system is not always a requirement, but in systems with some deadlines and critical decisions, the use of an operating system becomes imperative. For this reason, some concepts about operating systems and their mechanisms for processing and communication between tasks will be introduced.

2.1.1 Embedded System

An embedded system is a computer system, a combination of hardware, software, and perhaps mechanical or other parts designed to perform a specific function [4]. Most of the time, these systems are

part of a higher system, so users are unfamiliar with their presence. These systems work in several environments and range in size and complexity from personal devices such as digital watches to more complex mechanisms such as a router or printers. Furthermore, depending on how they are designed, they can range from the most powerful, with more computing resources or graphical capabilities, to the simplest ones without any supporting operating system. Figure 1 shows some examples of embedded systems that are present in our daily life, such as the smartphone and the microwave oven.

Figure 1: Example of an embedded system

By fulfilling all the requirements for the execution of a specific task or a restricted set of tasks, these systems are generally made up of the following components:

- CPU (Central Processing Unit)
- Memory
- Inputs and outputs
- Communication Protocol

Moreover, they can have restrictions on energy consumption using batteries or alternative energy sources and are programmed with a built-in software application that controls the hardware used. As mentioned earlier, these devices may have no user interface at all, making most of these systems standalone applications.

2.1.2 Real-Time System

A real-time system is a computational system with time constraints, which is partially specified in terms of its ability to perform certain calculations or decisions in an appropriate time [4]. These calculations or

decisions have time deadlines for the execution of a task. Thus, real-time systems can be divided into two categories, which range according to the importance that a failure has in the execution of a task. These two categories are:

- Hard real-time
- Soft real-time

A hard Real-Time system is a system in which this failure is fatal to the system operation and makes the whole system unusable, for example, a real-time system that is part of a control system in an airplane. On the other hand, we have Soft Real-Time systems in which a failure to meet task deadlines is not catastrophic. Streaming audio or video is an example of this kind of application in which a tolerance for delay and even data loss is allowed. The designer of a real- time system must be careful because they need to guarantee a reliable operation of the software and hardware in all possible conditions.

2.1.3 Operating System

An operating system typically consists of a set of "function calls", or software interrupts, and a periodic clock tick. An operating system is responsible for deciding which task the processor should run at any given time and controlling access to shared resources. The core of an operating system is the Kernel. It manages most of the main details an operating system needs to deal with, including memory, task concurrency, scheduling and I/O events [5], as represented in figure 2. In general, the Kernel is the part of the operating system that interacts directly with hardware, thus acting as a resource manager. These resources can be CPU time allocated to processes, management of existing RAM (Random Access Memory) or even connected hardware devices.

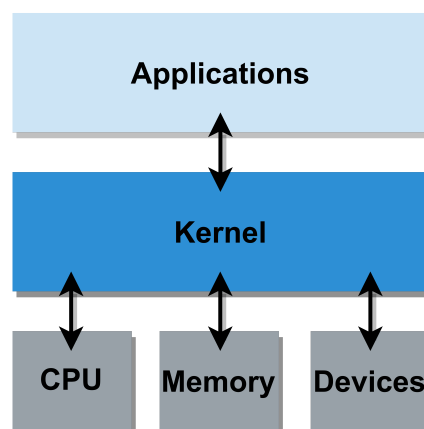


Figure 2: Kernel Role in a computer system

There are many types of operating systems, but in general, they can be divided into two types:

- General-Purpose Operating Systems
- Real-Time Operating Systems

GPOS (General-Purpose Operating Systems) are designed to run generic tasks that don't have time constraints, such as opening a Word document. The scheduling of tasks is done dynamically and fairly to improve overall performance, meaning that sometimes the jobs with the highest priority are not executed first. Such operating systems are typically found in PCs (Personal Computers), servers, tablets and smartphones. Linux, Windows and macOS are some of the most popular general purpose operating systems.

On the other hand, RTOS (Real Time Operating Systems) are designed with the aim of running applications at precise times. Task scheduling, unlike GPOS, is performed according to the highest priority tasks, and all the data processing must obey time constraints. Otherwise, the system will fail [6].

2.1.4 Multitasking

As previously mentioned, the operating systems are responsible for deciding which tasks will execute at a given moment. These threads are executed by the processor, which can switch between them and allow interaction between these several jobs. The ability to process one or more tasks simultaneously in an operating system is called multitasking [8]. This capability represents an advantage over basic systems that execute only one task. This advantage is because every time a thread waits for some signal, the processor can hold that thread and process other task. Therefore, the processor creates an illusion that several tasks are executing simultaneously, even though instructions are being processed one at a time.

Using a multitasking operating system to design a software application has the following advantages:

- Improved throughput
- Execution of several tasks simultaneously
- Structured code that can be easily extended and modified

All these advantages mean that the design of a complex application can be divided into smaller tasks that become more flexible. This divisions simplifies the development of an application, as it allows an easier and faster testing phase. In addition, several teams can work on different parts being less dependent on each other, and the code can be reused increasing productivity. Furthermore, all timing details can be removed from the application code and become operating system's responsibility.

2.1.5 Scheduling

After the multitasking concept explanation is applied at the operating system level, it is still necessary to understand how the system selects the upcoming tasks to run. The scheduler is the part of the Kernel responsible for deciding and selecting which task should execute at a given moment. The scheduler is also responsible for switching between tasks in progress and changing the status of a task when necessary. Figure 3 presents the FreeRTOS several states that a task may have in a multitasking operating system [7].

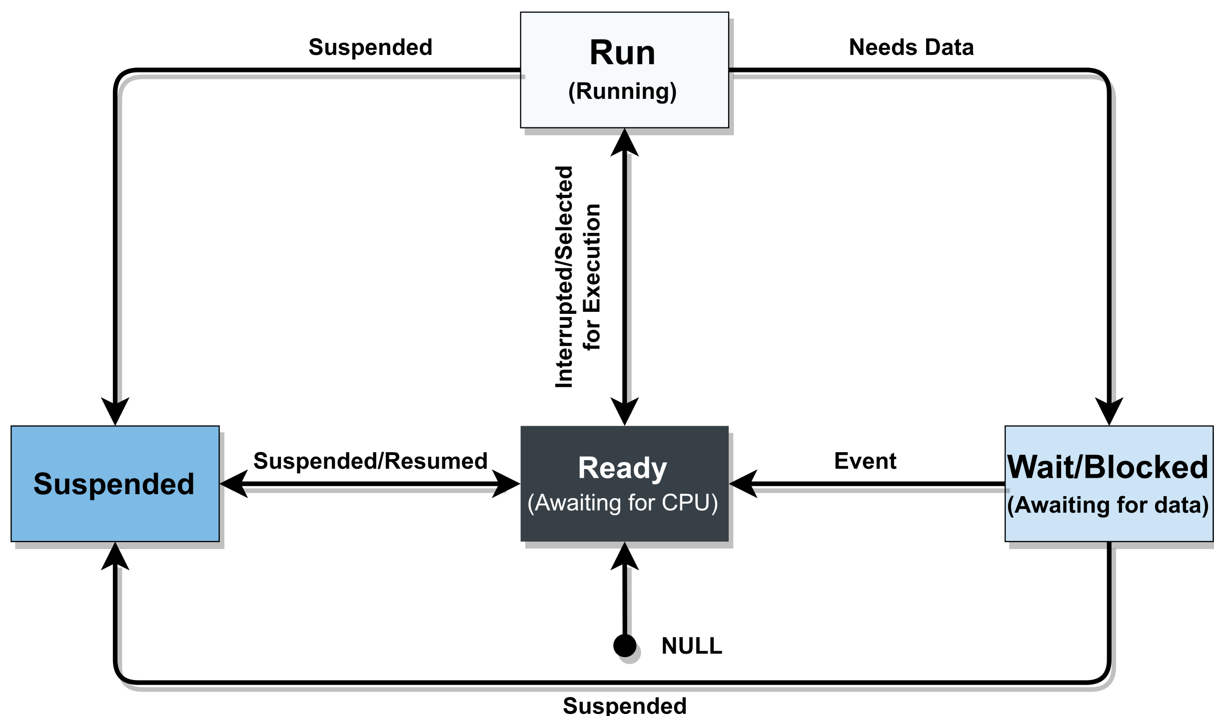


Figure 3: Scheduling States in FreeRTOS [7]

Embedded systems, in particular real-time systems, almost always require a method for sharing the "quantum" of processing between tasks. In most cases, this sharing respect scheduling rules determining that the most critical functions, with higher priority, will have access to the processor when necessary. Therefore, most real-time operating systems use pre-emptive scheduling, which implies that lower-priority tasks will wait for higher-priority tasks.

2.1.6 Inter-Process Communication

In a multitasking system, tasks interact with each other directly through some signalling or indirectly through resource sharing. Thus, the operating system uses synchronization mechanisms to control all these interactions. These interactions should be managed so that several tasks cannot access the same

resource simultaneously. Usually, this phenomenon is called "race condition". To avoid the "race condition", the operating systems offer some synchronization mechanisms that allow the designer to have greater variety in controlling access to shared resources. Some of these mechanisms are:

- Mutex
- Semaphore
- Interlocked Variable Access

Communication between these processes may be seen as a cooperation method between them. Processes can communicate with each other through shared memory and data transfer. Some task communication mechanisms are:

- Pipes
- Message Queues
- Shared Memory
- Remote Procedure Call

2.2 Crash detection sensors

The detection of car accidents requires sensors that are capable of measuring the acceleration forces experienced by the vehicle at the time of the collision. The most commonly used sensors for this purpose are IMUs (Inertial Measurement Units), inertial sensors, and pressure sensors. IMUs combine accelerometers, gyroscopes, and sometimes magnetometers to provide a comprehensive view of the vehicle's orientation and movement, allowing for accurate measurement of acceleration forces. Inertial sensors are specialized accelerometers that are highly sensitive and can detect even small changes in acceleration. Pressure sensors, meanwhile, are used to measure the forces applied to the vehicle during a collision, providing valuable information about the severity and distribution of forces across the vehicle. Together, these sensors provide a complete picture of the forces experienced by the vehicle during a collision, allowing for accurate and reliable detection of car accidents. However, for the proposed system the study of pressure sensors can be discarded because the inteded system must be compact and the operation location is in the car seat. Pressure sensors are very useful in other situations as explained further in this dissertation.

2.2.1 Accelerometers

Accelerometers are electromechanical sensors used to measure acceleration forces acting on an object to determine the object's position in space and monitor the motion of that object. These devices can measure tilt, shock, vibration, and inertial acceleration. The main applications are shown in table 1 with the respective g-Range [8].

Table 1: Accelerometer Grade and Typical Application Area

Accelerometer Grade	Main Application	g-Range
Consumer	Motion, static acceleration	1g
Automotive	Crash/Stability	200g
Industrial	Platform Stability/Tilt	25g
Tactical	Weapon/Craft navigation	8g
Navigation	Submarine/Craft navigation	15g

Acceleration forces applied to an object can be either static or dynamic.

- Static acceleration is a force that is constantly being applied, such as gravity. These forces are generally uniform and predictable
- Dynamic acceleration is a force that is not uniform, short-lived and has unexpected behaviour most of the time. An accident or collision between two masses is an example of this force type.

Therefore, these devices are essential in detecting dynamic forces involved in road accidents. The accelerometers used for crash detection and high-impact applications are typically "High-g" and can detect accelerations up to 200G or more depending on their properties. In recent years, accelerometers based on microelectromechanical systems (MEMS) have been the most widely used due to their attractive characteristics. MEMS devices are low cost, small in size, low power consumption, repeatability, high sensitivity and quite good design flexibility. These types of devices end up being popular due to easy integration, high level of functionality and excellent performance [8].

2.2.2 IMU

An Inertial Measurement Unit (IMU) is a device used to measure specific gravity and angular rate in the object to which it is attached [9]. An IMU generally consists of 3 types of sensors:

- Gyroscopes: providing a measure of angular rate
- Accelerometers: providing a measure of specific force/acceleration
- Magnetometers (optional):measurement of the magnetic field surrounding the system

The main areas of an IMU are presented in table 2, which also shows some relevant applications for each field. In the automotive field, this device is widely used in applications that require a calculation of the vehicle position, estimating the direction and distance travelled through a provided starting point. Usually, this type of device is combined with a GPS, thus ensuring greater accuracy in the vehicle position. The IMU is used in automatic parking as well, where it is also used to calculate the car's location and direction.

Table 2: IMU Grade and Typical Application Area

Accelerometer Grade	Main Application
Consumer	Motion tracker, entertainment systems
Industry	Location of equipment such as antennas
Militar	Maneuver aircraft
Navigation	GPS system
Automotive	Dead Reckoning, Automated Valet Parking

2.3 Crash detection algorithms

Crash detection algorithms are usually implemented in controllers and continuously monitor the sensors used. In addition, they make their decision based on the sensor data and some parameters:

- Derivative of acceleration (called jerk)
- Speed (after integration of acceleration signal).
- Displacement, squeeze, deformation (double integration).
- The signal energy (acceleration or velocity).
- Signal power (acceleration or velocity).

Some of these parameters are used for crash detection algorithms in order to trigger the airbag. Thus, several algorithms have been developed over the years. Table 3 presents different types of algorithms used in airbag control units, as well as their characteristics and the type of approach used.

Table 3: Highlights and comparisons of a selected list of sensing algorithms

Individual/Organization	Main Theme or Features	Approach
Diller/TRW	Total and partial energies	Energy in time domain
Allen/ASL	Power rate	Energy + jerk + acceleration + ΔV
Gioutsos/ASL	Waveform recognition	Jerk
Watanabe,Umezawa	Optimal timing	Predicted displacement + acceleration + jerk + energy
Mattes /Robert Bosch GmbG	Adjustable velocity threshold	ΔV
Diller/TRW	Summation of expert circuits	ΔV + jerk + displacement
Eigler and Weber/SIemens	Multiple evaluation circuits and time window	Acceleration recognition + velocity + displacement
Tohbaru/Honda Blackburn /TRW Blackburn and Gentry/TRW	Power of acceleration signal in a frequency range	ΔV + energy in frequency domain
Cashler and Kelly/Delco Electronics	Occupant displacement and crash severity	Jerk + acceleration + ΔV + displacement
McIver/TRW	Crash velocity and crash metrics	ΔV + acceleration + shape function
Sada and Moriyama/STC	Adjustable velocity based on physical quantities	Jerk + acceleration + ΔV + displacement
Kosiak/Delco Electronics	Crash sensing using anticipatory sensor inputs	Acceleration scaling with anticipatory sensors

2.3.1 "Total and Partial Energies", Diller

This algorithm was developed by R.W. Diller and analyzes the energy distributed during an accident. The algorithm uses a sliding window in which it calculates and monitors the amount of energy. The basic principle of the algorithm is based on the assumption that the energy dissipated in an accident (and indirectly recorded by the accelerometers) is divided between a general deceleration of the vehicle and energy dispersed in the structure of the vehicle (bending, vibration, strain, braking that are associated with the individual components of total energy). The length of the time windows is suggested to be 12 milliseconds and updated every 2 milliseconds. The total energy in a crash is related to the root-mean-square of the acceleration value, but the absolute value is substituted in the calculation. The partial energy is calculated by subtracting the mean value.For firing to take place, the rule is that total and partial energies must exceed their thresholds simultaneously prior to the end of a certain time interval. This is an algorithm developed from the perspective of energy transition in a crash event. The calculation of the total energy gives an indication of the overall slowing, while the amount of partial energy keeps track of the fluctuations in the signal [10, 11].

2.3.2 "Power-Rate Method", Allen

In the Power-Rate Method developed by Allen, the first derivative of the designated course of signal power is analysed. To determine the intensity of the crash, it uses measurements of acceleration, velocity, and acceleration derivative [12]. The mathematical equations for calculating power rate are:

Energy:

$$E = \frac{1}{2}mv^2$$

Power:

$$P = \frac{dE}{dt} = mv\frac{dv}{dt} = mva$$

Power-Rate:

$$P = \frac{dP}{dt} = m\left(v\frac{da}{dt} + \frac{dv}{dt}a\right) = m\left(vj + a^2\right)$$

2.3.3 "A Predictive Algorithm", Gioutsos

This algorithm is based on the slope of the acceleration sequence. One particular feature of this approach is the claim of time-invariant criterion. The magnitude of the underlying waveform is correlated with crash severity without the starting point of a crash event being identified [13].

2.3.4 "Adjustable Velocity Threshold", Mattes

The team of B. Mattes (Robert Bosch GmbH) developed an algorithm that initially integrates acceleration signal in the time domain, and then the resulting waveform of velocity is used to activate the system. Essentially, this concept is based on a "varying ΔV" threshold. The threshold ΔV is adjusted according the parameters of the crash pulses, such as acceleration or the variation of the velocity signal. The approach asserts that it is especially advantageous to adjust the release threshold as a function of time for oblique impact collisions [14].

2.3.5 "Multiple Evaluation and Expertise Algorithms", Diller

The approach uses multiple evaluation circuits, including core and supplementary algorithms. A evaluation circuit is a circuit that is responsible for a collision evaluation and is generally composed by a board with some accelerometers. In this board, a alghoritm is implemented to detect the collision and this alghoritm can be a simple or an advanced alghoritm. For example, velocity is used as the core whereas displacement and jerk algorithms are supplementary. Each evaluation circuit conducts an assessment of the signal and provides a vote of "fire" signal if the situation indicates that the restraint should be actuated. This invention suggests the use of multiple algorithms, including velocity, displacement, and jerk. The method of varying weighting factors for individual circuits provides flexibility in adjusting sensor performance [15].

2.3.6 ”Multiple Evaluation Circuits and Time Window”, Eigler

This approach uses multiple evaluation circuits, and each evaluation circuit evaluates the acceleration signal in the time domain with different criteria. This approach utilizes a discrimination circuit to differentiate types of accidents. Thresholds of different crashes are implemented in separate circuits. It also uses the displacement evaluation to determine an ”optimal” timing of the restraint deployment [16].

2.3.7 ”Power Spectrum of Acceleration Signal in a Frequency Range”, Tohbaru

The proposed method involves the following steps:

- Integrating the acceleration signal to determine the passenger’s inertial speed.
- Calculating the power of the filtered acceleration signal within a specific frequency range (100 to 200 Hz).
- Initiating the deployment of restraints when either the passenger’s inertial speed or the power exceeds a predetermined reference value.
- Ensuring prompt and uninterrupted generation of the starting signal, even in oblique collision scenarios.

This method enables timely activation of restraints based on the passenger’s speed and the power of the acceleration signal, ensuring effective protection during various types of collisions [17].

2.3.8 ”Deployment Method Based on Occupant Displacement and Crash Severity”, Cashler and Kelle

The article presents the invention of a two-step deployment control for a supplemental inflatable restraint system in vehicles. The control algorithm consists of a ”time to wait” step and a ”severity” step, which determine when and whether to deploy the restraint, respectively. The time to wait step relies on estimating occupant displacement resulting from the crash, while the severity step relies on estimating the severity of the crash. Notably, the control algorithm is enhanced to trigger earlier deployment when the acceleration data indicates a localized impact, such as an angle or pole impact [18].

2.3.9 ”Method and Apparatus for Sensing a Vehicle CrashCondition Using Velocity Enhanced Acceleration Metrics”, McIver

The invention described in the text involves a method and apparatus for sensing a vehicle crash condition using velocity-enhanced acceleration metrics [19]. The key elements of the invention can be summarized as follows:

- A crash velocity determining circuit calculates the crash velocity value based on accelerometer data.
- Crash metrics determining circuits calculate crash metric values related to acceleration.
- Threshold determining circuits establish threshold values associated with the crash velocity.
- Comparators compare the crash metric values with the corresponding thresholds.
- A shape monitoring circuit checks if the shape of the acceleration signal matches a predetermined pattern.
- If the shape matches, a controller generates a signal to activate a restraining device.

The invention utilizes four measurements performed on the filtered acceleration signal to control the deployment of occupant restraints:

- The first measurement filters the acceleration signal to obtain an acceleration value.
- The second measurement calculates the square of the acceleration signal.
- The third measurement computes the sum of the squared acceleration values over a specific time period.
- The fourth measurement involves monitoring the shape of the crash velocity over a predetermined time period.

2.3.10 ”Method and Apparatus for Crash Sensing Using Anticipatory Sensor Inputs”, Kosiak

One of the suggested approaches is to “prepare” or “arm” the crash sensing system when a crash is about to happen but has not actually happened yet. This example describes the use of anticipatory sensors in crash sensing [20].

The invention is summarized below:

- Anticipatory crash sensors mounted on the front or the sides of a vehicle sense the speed of approach to an obstacle.

- A signal processing algorithm determines the ratio of the speed to a threshold and multiplies the output of an accelerometer by the ratio
- This enhanced acceleration signal is input to a single-point crash sensor algorithm to reduce the time needed to generate a trigger signal.
- An anticipatory sensor capable of very near range sensing would permit calculation of the exact time of impact and would reduce the possible variation in impact speed since last-instant maneuver is accounted for.
- Through the use of two sensors spaced across the front of a vehicle, an indication may be made about whether the impending impact is a full frontal or localized impact, and the point and angle of impact may be determined.

2.4 Related Work

Due to the device characteristics, there are no identical products, so the existing work-study is carried out using similar devices and other devices that perform similar operations, like data acquisition, logging and, collision detection. In this section, an overview of the relevant studies and similar devices will be address.

2.4.1 Airbag

The airbag and its control unit are essential for road safety. In a frontal crash, the airbags help occupants adapt to the vehicle's deceleration with the lowest possible load values and prevent impacts on hard vehicle components like the steering wheel or instrument panel. The airbag is pyrotechnically ignited in a crash. Airbag electronics detect a crash and control the deployment of airbags and seat belt pretensioners. The airbag's algorithm and sensor data determine when it is triggered. The study of this device is important because it uses several methods and techniques to detect a collision, similar to other collision detection systems. Airbag electronics also monitor the entire airbag system, store fault and crash data, and activate the airbag indicator light in the event of system failure. They can notify other system components of a crash via CAN (Controller Area Network) bus and store faults in an error memory. They have a self-sufficient energy supply to ensure their function during a crash [21]. Figure 4 shows a typical airbag system, where analogue sensors are the inputs and analogue actuators are the outputs. In this example, sensors like buckle switches, accelerometers or pressure sensors are measuring the impact parameters. The airbag ECU determine if a crash ocurred and deploy the airbags using the appropriate distribution channels.

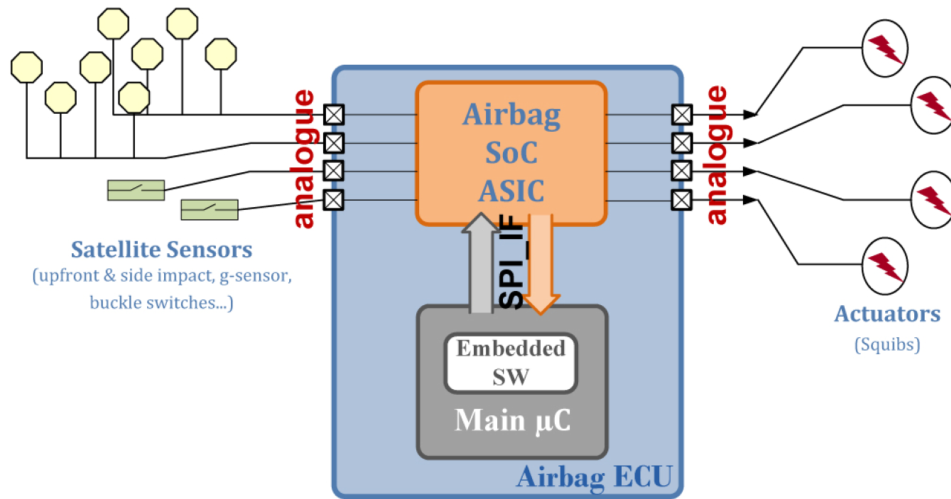


Figure 4: Airbag System Overview [21]

Figure 5 presents an example of an airbag control unit where two independent processors are used. One of the sensors is typically more reliable, so a complex algorithm is applied only to the data from this sensor. The other sensor is used to supplement the collision detection and a simple algorithm is applied to its data. The other processor is a simple processor that is only used to sample and save the data. All these algorithms are combined to determine when to deploy the airbag. Figure 6 shows an example of the sensor system in a car. Typically, four external acceleration sensors and two pressure sensors are used in the front, rear, and sides to detect a collision. In the internal sensor system, some additional sensors are also used to detect a car collision [22].

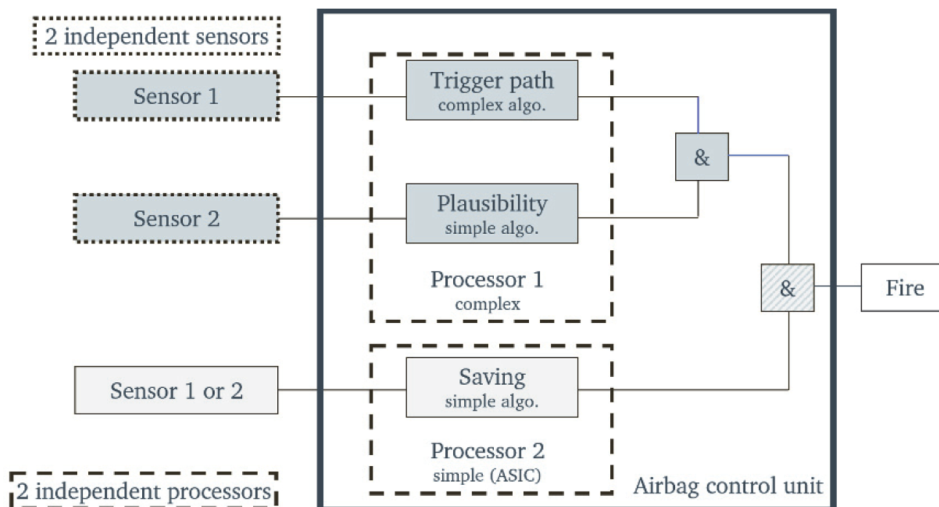


Figure 5: Airbag Control Unit [22]

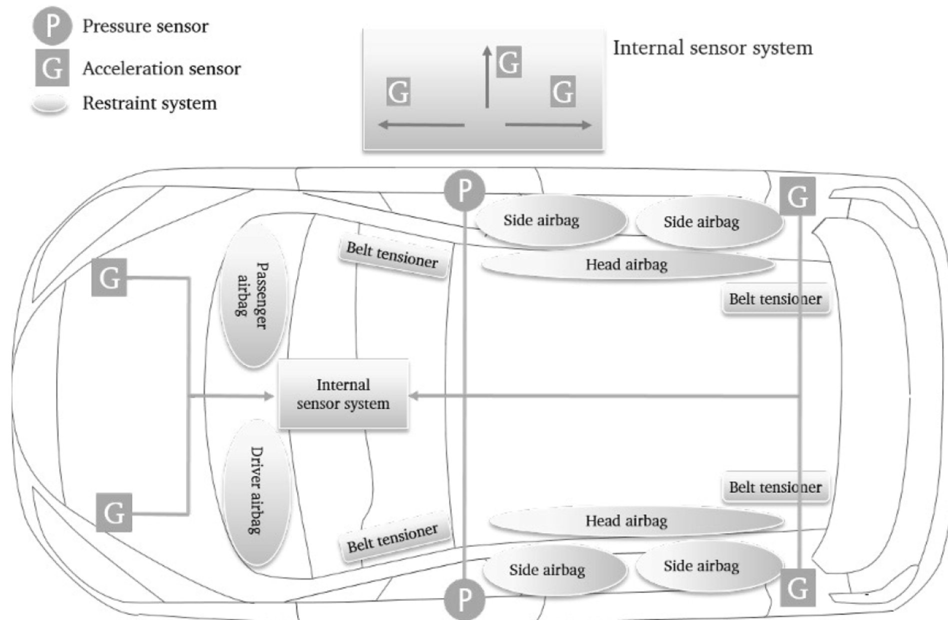


Figure 6: Airbag Sensors [22]

In the airbag control unit, the functional principle of the signal processing chain is shown in Figure 7. Analogue signals from sensors are processed through an analogue-to-digital converter (A/D converter) that converts the continuous input signal into discrete samples, which are then converted into digital values. The parameters of the A/D converter include its bit depth and maximum sampling rate. The input signal is first filtered by a low-pass filter, then sampled by the A/D converter, and finally processed by a digital high-pass filter to eliminate zero-point shifts caused by temperature and ageing effects. This allows for further processing in the crash algorithm. Sampling frequency is determined by the bandwidth of the input signal and must be greater than twice the maximum possible frequency in order to reconstruct the signal without loss. Bessel filters with a cut-off frequency of 400 Hz are commonly used for this purpose. Typical sampling frequencies used in airbag control units are between 1 kHz and 4 kHz. These frequencies have been found to be best suited for further processing in crash algorithms [22].

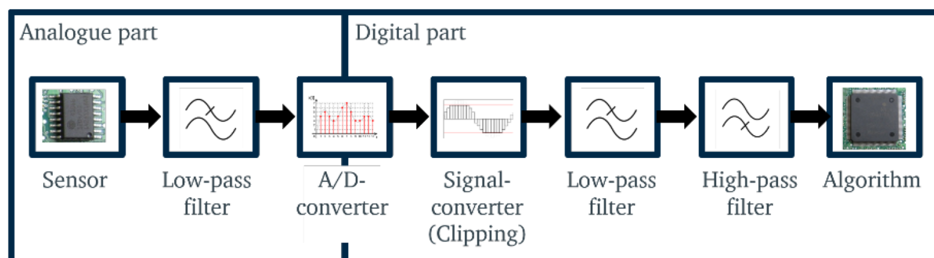


Figure 7: Airbag signal Processing

2.4.2 High-impact scenarios

Considering the conditions in which the equipment will work, it is necessary to analyse the characteristics present in a car crash and attempt to define the limits to which the product should operate. A car collision occurs when a vehicle collides with another object, be it another car, a wall, or a person. Each vehicle has an energy associated called kinetic energy, defined as the work required to accelerate a body of mass at rest to the acquire speed. In a car crash, this energy is transferred somewhere and usually wasted as heat. In addition, much of this energy is absorbed by the body, especially if the seat belt is not attached[23]. Today, as the automotive industry advances, vehicles are already designed to absorb much of a car crash energy through structures such as bumper. Also, systems such as ABS or airbag are fitted to reduce the impact.

In a high-impact scenario, it is usual to evaluate this impact based on the force exerted at the moment of collision and the acceleration experienced by the vehicle. When measuring one of these factors, it is possible to characterize and make decisions about vehicle collisions. Acceleration is one of the parameters that allow an accident assessment and the main factor used in decision-making in a safety device such as the airbag. Acceleration is defined as the rate of change of velocity of a mass, and its SI unit is metre per second squared. However, to distinguish acceleration relative to free fall from simple acceleration (rate of change of velocity), the unit g (or g) is often used. One g is the force per unit mass due to gravity at the Earth's surface and is the standard gravity, defined as 9.80665 metres per second squared [24] [25].

Before understanding and classifying which human tolerances are present in a car accident, it is necessary to establish the vehicle's coordinate axes. Figure 8 represents the coordinate system used as a reference during this dissertation.

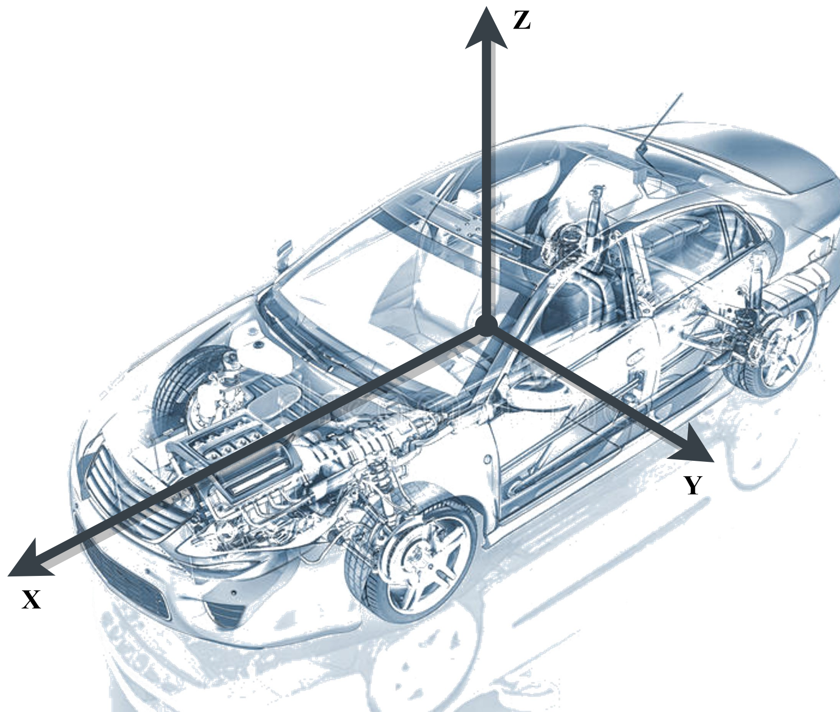


Figure 8: Car Coordinates

Usually, a road accident is described as a "Crash Pulse", which represents the acceleration that has occurred, and which assumes a triangular shape. This description allows an accident classification in terms of its peak or the duration of that acceleration peak [26]. For a better understanding of the human tolerances to acceleration present in such accidents, the following factors should be considered:

- Magnitude
- Direction
- Duration
- Rate of onset
- Position/Restraint/Support

Through the exposed concepts, to design the detection algorithms of the proposed system, it is essential to know the human limits. In table 4, it is possible to observe these limits and see the present range of accelerations on all axes. As mentioned, these are extreme limits that a human can hold for a few milliseconds.

Table 4: Human Tolerance Limits [26]

Direction of Accelerating Force	Occupant's Inertial Response	Tolerance Level
Headward (+Gz)	Eyeballs Down	20-25 G
Tailward (-Gz)	Eyeballs Up	15 G
Lateral Right (+Gy)	Eyeballs Left	20 G
Lateral Left (-Gy)	Eyeballs Right	20 G
Back to Chest (+Gx)	Eyeballs Out	45 G
Chest to Back (-Gx)	Eyeballs In	45 G

2.4.3 Bosch Peripheral Acceleration Sensor (PAS)

As meantion earlier in the airbag section, the airbag uses external and internal sensors present in specific car areas. One example of a external sensor is the peripheral acceleration sensor (PAS) present in figure 9.

Figure 9: Peripheral acceleration sensor

The Bosch Peripheral Acceleration Sensor (PAS) [27] provides information on steering and impact levels by measuring accelerations during a crash. In addition, Bosch provides a two-channel peripheral acceleration sensor, the enhanced PAS, which can record longitudinal and lateral accelerations of the vehicle. The optional installation of this sensor can further improve the performance of the airbag system as it provides additional acceleration information about the vehicle's longitudinal direction. Special acceleration sensors can be fitted in the "crumple zone" at the front of the car to improve frontal impact detection.

Bosch also provides acceleration sensors mounted in the front bumper to detect whether a pedestrian is involved in the accident protecting pedestrians during a frontal collision.

2.4.4 Literature review

In this section, three studies will be presented that demonstrate the use of collision detection systems. These studies highlight the effectiveness of these systems in several contexts and provide valuable insights into their capabilities and limitations. The first study examines the use of a multi-sensor decision fusion approach for reliable automotive crash detection, while the second study looks at the application of a low-cost collision impact analyser based on MEMS accelerometers. The third study evaluates the performance of an IMU sensor-based crash detection system. Through these examples, the versatility and potential of collision detection systems in promoting safety and preventing accidents will be illustrated.

Reliable Automotive Crash Detection using Multi Sensor Decision Fusion

The aim of this study was to investigate the potential of multi-sensor decision fusion for reliable airbag deployment in automobiles. The researchers used measurements from an accelerometer located on the vehicle engine and a loadcell on the driver shoulder-belt to make the airbag deployment decision. They developed and tuned Kalman filters for state estimation from the sensor data and used a cumulative sum control chart (CUSUM) for crash detection at the individual sensor level. The resulting algorithm was implemented in MATLAB and tested using US National Highway Traffic Safety Administration (NHTSA) automobile crash data sets.

The results of the study showed that the use of multi-sensor decision fusion provided reliable crash detection compared to single-sensor-based methods. The time taken for the worst-case scenario was within the acceptable bounds of airbag deployment time (50 ms), even with the unoptimized MATLAB implementation. In conclusion, the study demonstrated the potential of multi-sensor decision fusion for reliable airbag deployment in automobiles and suggested that further research could investigate the use of more than two sensors and signal- level fusion [28].

MEMS Accelerometer based Low-Cost Collision Impact Analyzer

The aim of this study was to investigate the use of low-cost MEMS accelerometers for accident impact analysis. The researchers developed a hardware and software system that was able to quickly collect, process, and store large amounts of data from the accelerometers. The system was tested by simulating an accident by dropping the system from a certain height, and the recorded data was later analysed on a computer. The results of the study showed that the low-cost impact analyser could track the system movement with a logging rate of 28 samples per second. However, the researchers also identified several limitations of the system, including the slow data logging speed using a microSD card and the FAT (File Allocation Table) file system. The researchers suggested that using wireless data logging with BLE devices or GSM/GPRS modules could improve the throughput of data transmission.

In conclusion, the study demonstrated the potential of low-cost MEMS accelerometers for accident impact analysis. Further research could investigate ways to improve the speed and accuracy of the system, as well as its applications in different fields such as military, aerospace, sports, and automotive engineering. The statistical analysis of g-force values in different crash scenarios could also provide valuable insights for engineers designing more effective products [29].

Crash detection using IMU sensor

The aim of this study was to investigate the use of an MPU6050 IMU sensor for detecting motorcycle accidents. The researchers used the sensor to measure the Euler angles (roll, pitch, yaw) and the G forces on different axes. They also developed an algorithm that used data fusion between acceleration, deceleration, and tilt angles to detect crashes. The results of the study showed that the MPU6050 sensor was able to provide accurate and reliable data, even in the presence of vibrations from the motorcycle engine. The crash detection algorithm had a high success rate, with no false positives or missed crashes recorded in the tests.

In conclusion, the study demonstrated the potential of using an MPU6050 IMU sensor for detecting motorcycle accidents. The low cost and high accuracy of the sensor make it a good choice for this application. The use of the sensor in a system with E-call could decrease the time between the accident and the arrival of emergency services by 50%, potentially saving lives. Further research could investigate the use of automotive sensors, which are designed to reduce vibrations and improve accuracy, as well as the integration of the crash detection system with other safety features on motorcycles [30].

2.5 Conclusions

In conclusion, this chapter presents an overview of the technologies and concepts that will be addressed in the dissertation. The goal is to develop an embedded system for data acquisition, so some

fundamentals of these systems are discussed. The characterization of the high-impact environment in which the system will be used is also presented.

Additionally, the chapter discusses the technologies, types of sensors, and algorithms used in accident detection, as well as a description of similar devices on the market and similar studies. This information provides the foundation for the development of an effective and reliable collision detection system. Further research could investigate the use of more sensors and signal-level fusion, ways to improve the speed and accuracy of systems, and the integration of crash detection systems with other safety features.

Chapter 3

System Specification

The previous chapter introduced some basic concepts and theoretical backgrounds in the embedded systems field. In addition, it showed the characteristics present in a high-impact scenario and the most used sensors and algorithms to detect a car crash. The theoretical concepts and the presented application scenario are relevant to understand the construction of an embedded system and the needs and choices to be made. In this chapter, the system requirements will be presented, together with the component selection and a system overview, which allows an understanding of the hardware and software to develop.

3.1 Requirements

The project requirements for the proposed system include the following functional and non- functional elements:

3.1.1 Functional Requirements:

- Detect a car collision.
- Save collision data.
- Perform a real-time operation

3.1.2 Non-Functional Requirements:

- Compact Size
- Blackbox

3.2 System Overview

In order to fulfill the requirements of the project, Figure 10 presents the system overview. The main component is the microcontroller, which will be responsible for detecting collisions. This detection is based on data from the accelerometer and IMU sensors, which provide input to the system. In the event of a collision, the proposed system will save the data to an SD card and provide a trigger signal to one of the microcontroller's pins. This will enable the system to respond to the collision and take appropriate action, such as sending an alert or activating safety measures. The use of an SD card allows for the storage of large amounts of data, which can be used for further analysis and investigation.

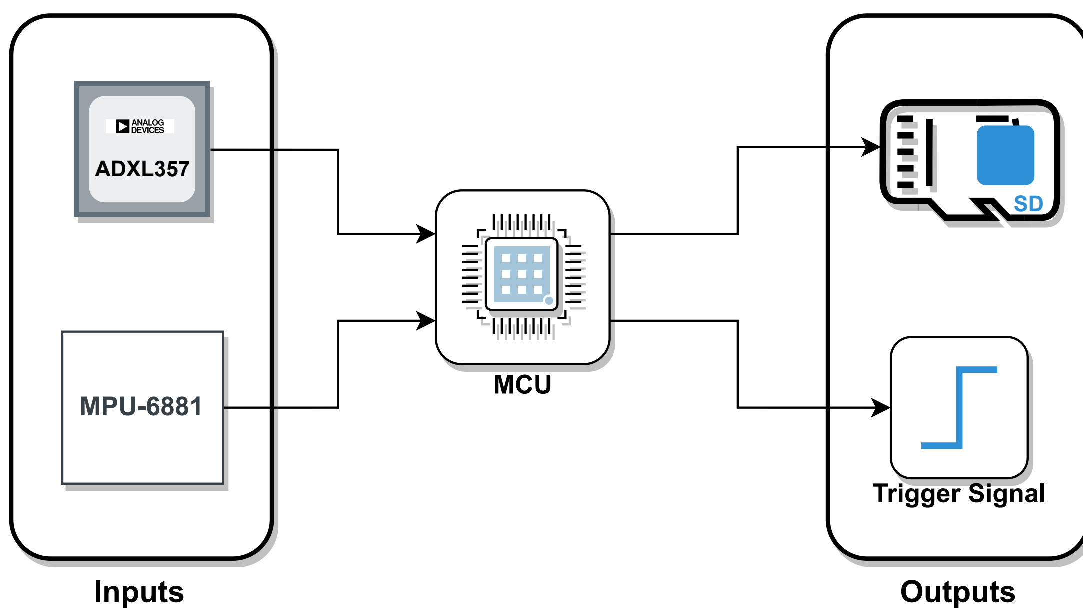


Figure 10: System Overview

3.3 Hardware Specification

This section will provide detailed information on the components required for the assembly, including their technical specifications and justifications. It will follow the system overview and provide a comprehensive overview of the hardware components used.

Programming an embedded system platform usually involves the use of hardware, so the PCB must be able to be programmed by micro-USB and must have support to allow data to be saved on an SD card. There are other ways of storing the necessary data, which are widely used and allow this task to be performed faster and more efficiently. However, as the intention is to build a prototype, a microSD card was selected for its practical use. Moreover, when making a prototype, there are some errors associated. Either at the level of component soldering or even in the "footprint" of some components. Better observation of

some signals allows solving or helping to solve possible errors, so the project board should include LEDs and test points to check and debug this errors easily.

3.3.1 Sensors

The sensors selection is based on the need to measure high accelerations and obtain accurate accelerations during a partial period of the accident. Thus, the choice fell on using an accelerometer capable of detecting values above 30G and a second accelerometer to be combined. The first device allows the storage of all the high acceleration peaks and uses an accelerometer with higher precision. The second accelerometer is embedded in an inertial measurement unit that also contains a built-in gyroscope, thus allowing the inclusion of this data for future analysis.

ADXL357

The ADXL357 is a high-performance accelerometer that is suitable for applications that require high accuracy and precision measurements of acceleration. It offers a choice of three different ranges up to 40G, making it suitable for a wide range of applications. The low drift, low noise, and low power of the ADXL357 enable accurate tilt measurement in high-vibration environments. The device also features good signal-to-noise ratio, minimal drift over temperature, and long-term stability, making it ideal for precision applications that require minimal calibration. Additionally, the ADXL357 contains antialiasing filters and user-selected output data rates, making it easy to integrate into a variety of systems. Overall, the ADXL357 is a versatile and reliable accelerometer that is well-suited for applications that require high-performance measurement of acceleration.

Figure 11: ADXL357 Accelerometer

MPU-6881

The inclusion of an IMU in the system is important because it allows for the combination of data from multiple sensors, including accelerometers and gyroscopes. This can provide a more accurate and comprehensive view of the device's orientation and movement.

The IMU chosen for this application is the MPU-6881, which contains both a 3-axis accelerometer and a 3-axis gyroscope. This allows for the measurement of both linear acceleration and angular velocity, providing a complete picture of the device's movement and orientation. The MPU-6881 accelerometer is a MEMS accelerometer, which means that it uses tiny mechanical structures to detect acceleration. The accelerometer has ADCs (Analog-to-Digital Converters) with 16-bit resolution, which allows for highly accurate measurement of acceleration. The accelerometer uses different proof masses for each axis, which means that an acceleration in a given direction will cause a displacement of the corresponding proof mass. This displacement is then detected by capacitive sensors, which output a differential signal that is converted to digital form by the ADCs. The digital output of the ADCs can be adjusted to a range of ±2g, ±4g, ±8g, or ±16g, depending on the specific application.

3.3.2 CPU

The central processing unit is the main component of this system because is responsible for running and performing all the software functions.

The microprocessor selected for this application is the ESP32, presented in figure 12. This series of chips are small, low-cost, and also low-power systems. Although there is no requirement for the application to be low power, this is always an advantage for future modifications. These chips are integrated with Wi-Fi and Bluetooth, which allows them to be included in a wide range of IoT systems. The module chosen is the ESP32-MINI-1 which has only one core. ESP32, as the name implies, is 32bit and has a clock frequency of up to 240Mhz.

Figure 12: ESP32-Mini-1

In terms of memory, the specifications are as follows:

- 448 KB ROM.
- 520 KB SRAM.
- 16 KB SRAM in (Real-Time Clock).
- 4 MB SPI flash.

3.4 Software Specification

The two main applications in the project are the detection and recording of collisions. For these programs to operate, data from both sensors must be collected. Therefore, drivers that can write and read data are essential. Additionally, to control the SPI transactions between the sensors, a module known as the SPI driver must be created to manage independent transactions between the sensors. This module is responsible for managing all bus movements, such as insertion, removal, reading, and writing.

As shown in the Figure 13, after establishing connectivity and communication with the sensors, an interface must be created to allow for the necessary configurations to start and stop sensor use, as well as to structure the data from the drivers and provide it in a rearranged form for the application. In the final phase of system operation, data recording is stored on a microSD card through the use of a driver specifically created for this purpose. This driver uses the functions of the SPI driver to communicate with the SD card present in the PCB connector.

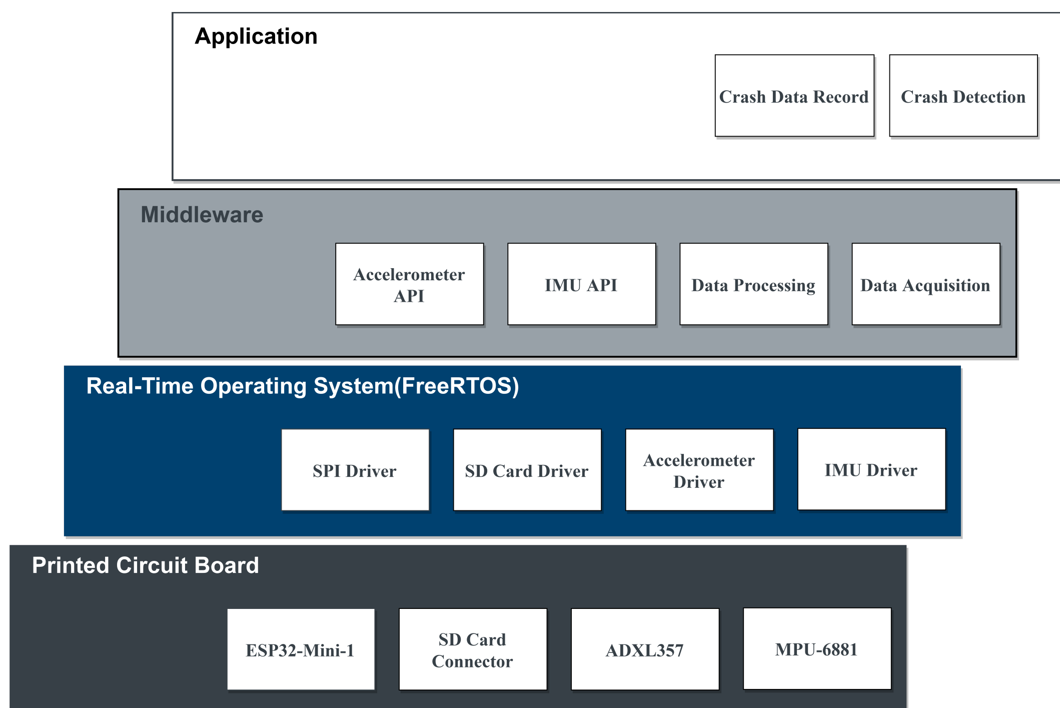


Figure 13: Software Architecture

3.5 Signal Processing

An accident detection is an event that depends not only on the sensors used but also on the treatment and processing that data goes through (as seen in the airbag review). So, it is necessary to study what kind of processing will be applied in the system and how to combine the data from the sensors to obtain a more accurate acceleration. This section will present all filters used and, in some cases, an explanation of their selection or use.

When analysing the data flow in figure 14, it is possible to see that the accelerometer ADXL357 has analogue and digital filters before and after the ADCs. These ADCs have a resolution of 20 bits, and the sampling frequency used in this device is 4kHz. The analogue filters are low pass filters that are used for antialiasing, which reduces the distortion in signals caused by aliasing. This type of filter is also present in the digital part and performs the same function. In addition to the antialiasing filters, the ADXL357 also has low pass and high pass filters that are customizable by the user. The second device only has a digital low-pass custom filter.

For the application in question, the limitation to a frequency band becomes relevant because it allows the elimination of some unnecessary data for a collision evaluation. Therefore, the range chosen was 10Hz-500Hz, which eliminates the DC component that, in this case, is the influence of gravity. As it is possible to predict, the data from MPU-6881 were not in the same frequency range as the accelerometer ADXL357 since it has no high-pass filter, so the use of a digital high-pass filter will be necessary at a later stage.

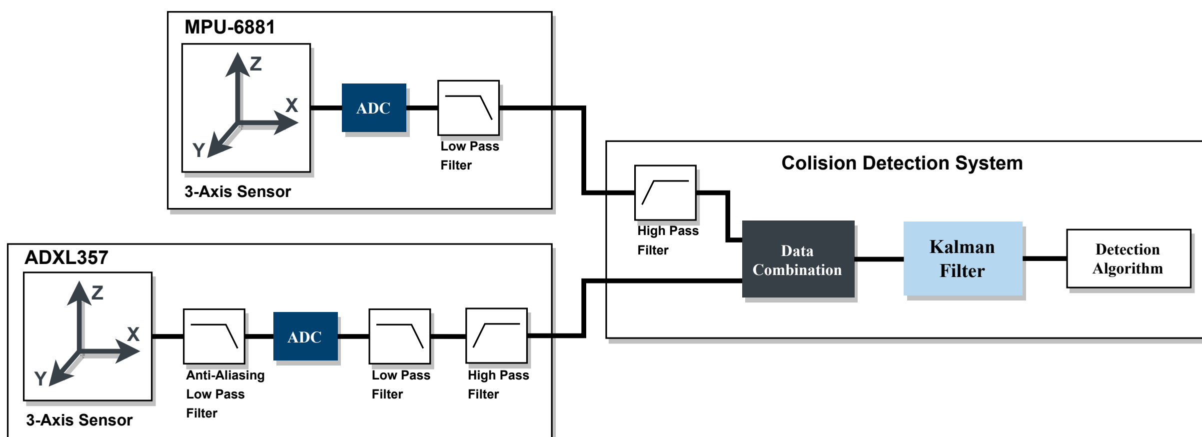


Figure 14: Signal Processing Overview

One of the system requirements is the ability to obtain an accurate acceleration. This requirement can be obtained by combining the data from the sensors. One of the most used filters for this situation is the Kalman filter. As mention earlier, this filter was not a requirement and was included in the system to archieve a better precision and accuracy.

3.5.1 Kalman Filter

Nowadays, most of modern systems use multiple sensors to estimate a given variable. It is because one of the biggest challenges in control systems is achieving great precision and accuracy when there are some uncertainties. These uncertainties usually come from disturbances, such as noise or atmospheric effects, making the sensor's values not 100% reliable. The values uncertainty can be reduced by using several sensors and combining their data to obtain an "Optimized" value. One of the most commonly used filters in this situation is the Kalman filter. The Kalman filter originated in 1960 when its creator Rudolf E. Kálmán published a scientific paper that has become famous, where he presented a recursive solution to a linear data filtering problem [31]. Nowadays, this filter is one of the most relevant and most used estimation algorithms.

The Kalman filter can be divided into two phases, a prediction phase and a correction phase. Initially, the system state is unknown, so the next state prediction is not reliable. Therefore, the system state is initialized with an initial guess and calculate the predicted state for the next iteration using the system dynamics model. The correction phase is the core of the filter and uses the predicted and measured values to obtain a weighted and optimized value to correct the system state.

The diagram in Figure 15 illustrates a simplified version of the Kalman filter, showing the different blocks that make up the filter and their inputs and outputs. The Kalman filter inputs include initial values based on an initial guess and measurements taken by sensors, along with the uncertainties in these measurements. The filter's outputs consist of estimates of the current state and the uncertainty of these estimates. It is common for the initial few iterations to not be precise, depending on the initial values, but the Kalman filter will eventually converge to a more accurate value.

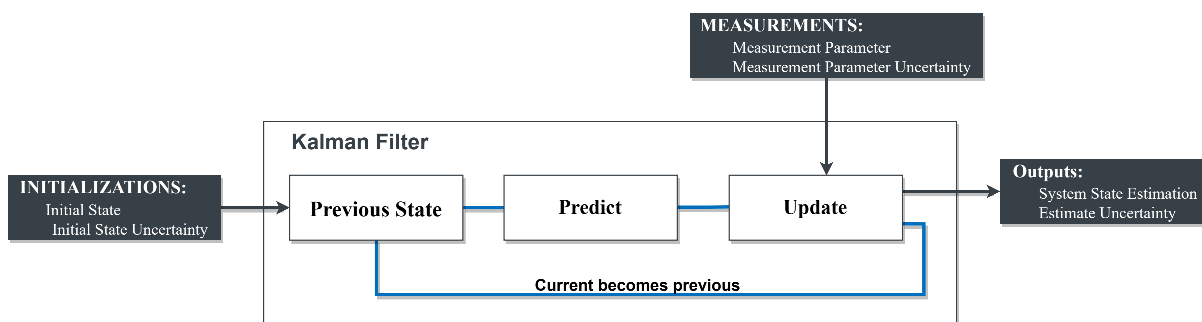


Figure 15: Kalman Simple Overview

To better understand the Kalman filter, it is necessary to explain the entire data flow and their respective equations. Figure 16 presents a detailed Kalman filter view where it is possible to observe the several variables implied. The first step is the filter initialization which only occurs at system start-up and provides two parameters:

- Initial system state (X_0)

- Initial system uncertainty (P_0)

Then comes the system prediction phase, where the system dynamics are used to predict the future values of the system state and the uncertainty associated. In the first iteration, the filter uses the values provided by the initialization to predict the next system state. From this iteration on, the final values of the correction phase of this filter are used.

The following step is the measurements that served as input for the correction phase. The measurement process involves the use of two variables:

- State of the measuring system (Y_k)
- Uncertainty of the measurement

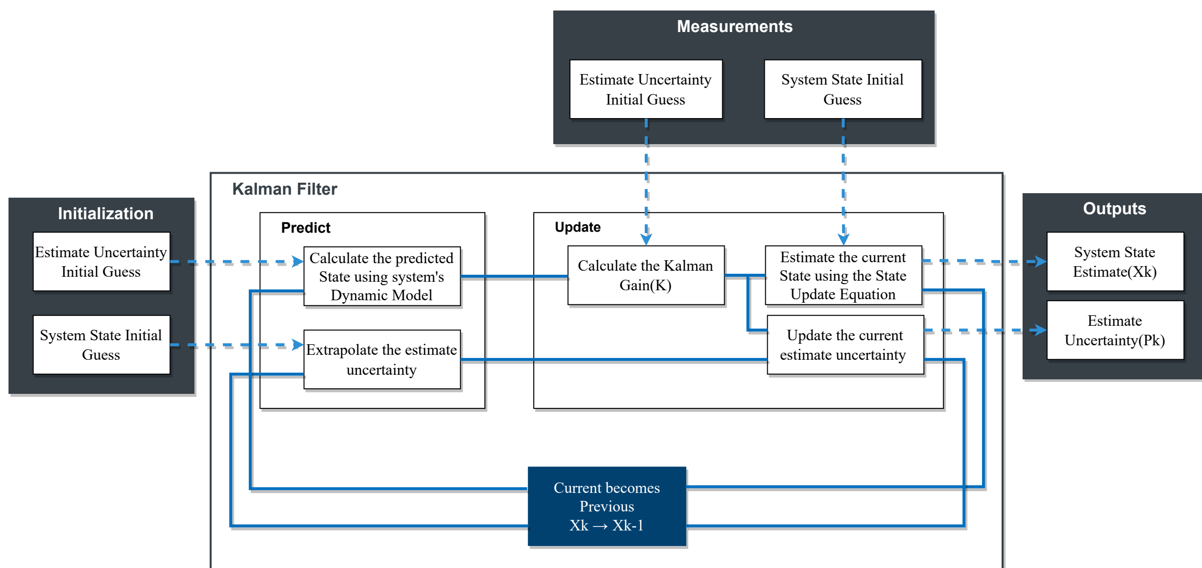


Figure 16: Kalman Detailed Overview

The state of the measuring system is the measured values from the sensors. The measurement uncertainty is the uncertainty that those sensors have in the variable they are measuring and are usually described in their datasheet. Thus, the correction phase presents the following inputs:

- Measured Values (Y_k)
- Measurement Uncertainty
- Previous System State (X_{k-1})
- Previous System Uncertainty (P_{k-1})

Kalman Filter Equations

After a detailed description of the variables in the Kalman Filter, lets describe the main equations. It is composed by five equations, two in the predict phase and three in the update phase, as shown in figure 17.

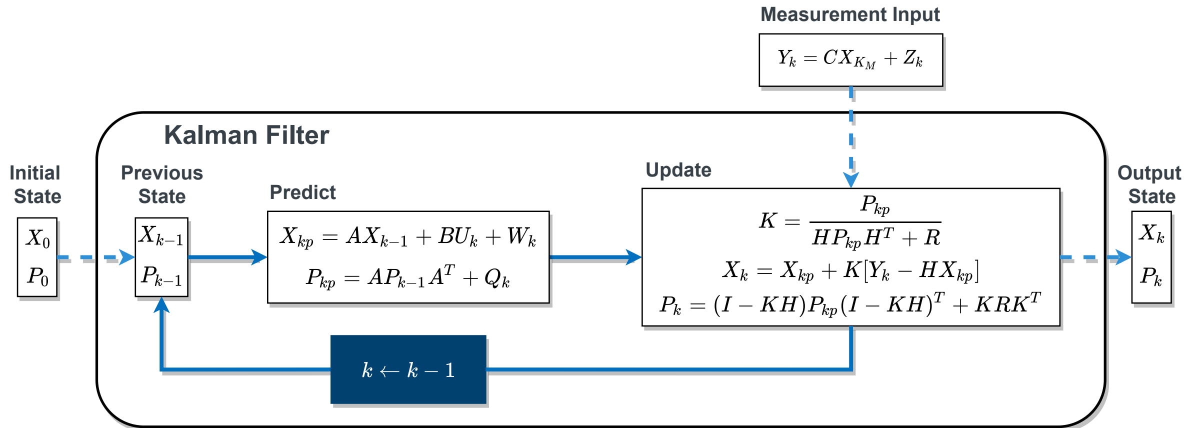


Figure 17: Kalman Equations Overview

The predict phase begins with the State Extrapolation Equation, which is used to predict the next state of the system based on the system dynamics that are already known. This equation takes the form of:

$$X_{\text{kp}} = AX_{\text{k-1}} + BU_{\text{k}} + W_{\text{k}}$$

X_{kp} is the predicted state, X_{k-1} is the previous state, A is the state transition matrix, B is the input matrix, and U is the current input. The second equation in the predict phase is the covariance extrapolation equation, which is used to predict the covariance of the next state based on the covariance of the previous state and the system dynamics. This equation generally takes the form of:

$$P_{\text{kp}} = AP_{\text{k-1}}A^T + Q_{\text{k}}$$

P_{kp} is the covariance of the current state, $AP_{\text{k-1}}A^T$ is the expected covariance of the current state given the previous state and system dynamics, and Q is the process noise covariance matrix representing the uncertainty in the system dynamics. After the predicted state and covariance have been calculated, the Kalman filter moves on to the update phase. The first step in this phase is to calculate the Kalman gain using the following equation:

$$K = \frac{P_{kp}H}{HP_{kp}H^T + R}$$

K is the Kalman gain, P_{kp} is the predicted state covariance, H is the measurement matrix, and R is the measurement noise covariance. The Kalman gain determines how much weight to give to the predicted

state and the actual measurement in the final estimate. The second step in the update phase is to update the predicted state estimate (X_k) using the Kalman gain and the measurement. This is done using the following equation:

$$X_k = X_{kp} + K(Y_k - HX_{kp})$$

X_k is the updated predicted state estimate, X_{kp} is the previous state estimate, Y_k is the current measurement, and H is the measurement matrix.The third and final step in the update phase is to update the predicted state covariance. This is done using the following equation:

$$P_k = (I - KH)P_{kp}(I - KH)^T + KRK^T$$

P_k is the updated predicted state covariance, I is the identity matrix, K is the Kalman gain, and H is the measurement matrix. The result of the update phase is a new, improved estimate of the system's state and its associated covariance. These updated estimates are then used in the next iteration of the Kalman filter to further improve the accuracy of the state estimation.

3.6 Conclusions

This chapter outlines the system specification process, starting by establishing the requirements to fulfil all system functionalities. To achieve this, the entire system and its components are specified, including hardware, software, and signal processing, with detailed descriptions provided for quick comprehension of the system's operation. Following this analysis and problem-solving approach, the system design will begin detailing all the integration of the selected components and explaining how the different software and hardware parts will be combined to achieve a working system.

Chapter 4

Design

The design of a system is carried out through a set of tools that allow explanations for the problems and specifications presented in the system analysis. Tools like diagrams and flowcharts were used to design a viable solution by showing and justifying all the choices. Therefore, in this chapter, the hardware and software used will be specified, together with some considerations and precautions. Regarding filters used, the Kalman filter parameters and other digital filters shall be described.

4.1 Hardware Considerations

After the main components have been selected in the previous phase, the requirements for designing a PCB are fulfilled. At this stage, it is relevant to design several schematics and connect all the components to get a view of the several blocks from a theoretical perspective.

Before starting to design the schematics, let's understand what's the necessary steps to make a PCB. Figure 18 shows the build process, that starts exactly by design all the schematics with the main components previously selected. After that, its necessary to make a schematic error verification in order to proceed to the layout phase. In this layout phase, first is defined the design rules that affects the way how the designer will route and place the components. After that, the designer proceeds to implement the PCB layout making several DRC (Design Rule Checking) verifications until the board is ready. The PCB fabrication process begins after generating the required output files, including Gerber files, drill files, and a bill of materials. These files are used to manufacture the PCB, which involves the application of various processes such as drilling, plating, and solder masking to create a good quality board. Once the board is fabricated, the components can be mounted and soldered onto the board, followed by thorough testing to ensure proper functionality.

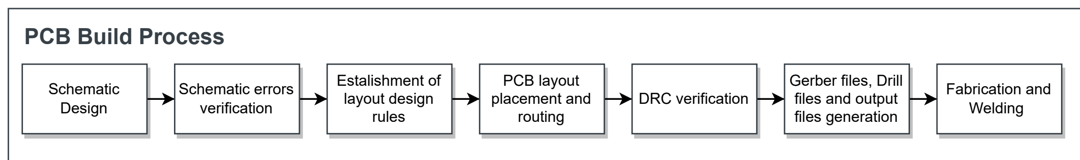


Figure 18: PCB Build Process

Thus, at this stage, it is relevant to design several schematics and connect all the components to get a view of the several blocks from a theoretical perspective. Figure 19 shows the PCB block diagram used, where it is possible to see all the relevant components and connections that will be present in this board. In this section, the first step of the build process is covered, where the operating mode of each device, accompanied by the recommended schematic for its use, will be presented.

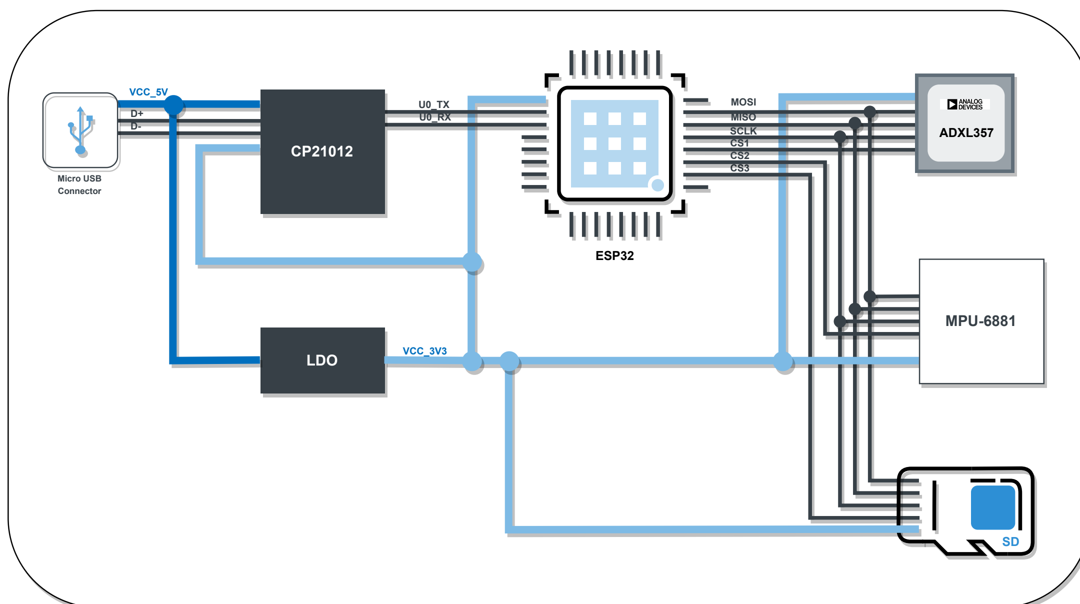


Figure 19: PCB Overview

In figure 19, some components are chosen by the compatibility and usage in the ESP32 development boards. This decision will be also covered in this section. Another important decision is the use of SPI protocol instead of other protocols. The digital interfaces for the majority of components are I2C and SPI, so it was decided to use SPI as a communication vehicle between these components. One of the reasons for using this protocol is that it is faster than I2C. Besides that, it's a full-duplex protocol, is generally easy to understand and it encompasses all the sensors and the microSD, which make it pratical to use.

4.1.1 LDO (Low-Dropout)

The sensors and CPU specified in the previous phase uses 3.3V as supply voltage. Since, the intention of this board is act as prototype, the choice to power-up the board was by using a micro-USB connection.

However this connection uses 5V as power voltage, which makes necessary the use of a circuit to lower the voltage in order to power the other components. Another choice was using an LDO to lower the voltage and provide a stable output to the PCB.

The power schematic, shown in figure 20, shows the circuit responsible for reducing the voltage and ensuring that all of the ICs on the PCB receive a stable power supply. This circuit is composed by an LDO and a Led to indicate if the board is power-up. The LDO used is the LT1117, since it is a popular choice on ESP32 development boards and it is a low dropout positive regulator that can provide a stable 3.3V output voltage at up to 800mA of current. As meationed earlier, this LDO is used to reduce the voltage, and its schematic is taken from the component's datasheet to ensure proper operation [32]. The capacitors has several important functions on the power sheet. The input capacitor filters out noise on the input voltage, and it also prevents the malfunction of the LDO in the event of a drop in the input voltage due to a sudden change in output current. The output capacitor improves the response to changes in output current and provides phase compensation for the feedback loop. In addition to its low dropout voltage and high current output, the LT1117 also has several protective features that make it well-suited for use in this application. These features include overcurrent and thermal protection, short-circuit protection, and overvoltage protection, which can help to prevent damage to the board and its components in the event of a fault.

The power led is used to indicate if the board is powered up, and the resistor is calculated to set a current limit in that LED.

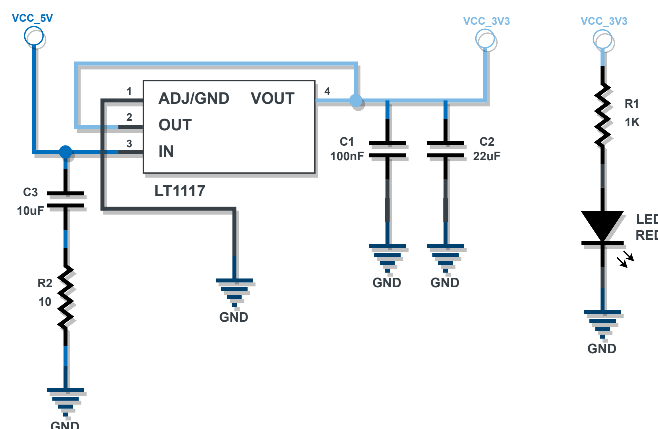


Figure 20: LDO schematic

4.1.2 ADXL357

The ADXL357 schematic, illustrated in Figure 21, shows all the necessary connections and components for operating the device following the recommendations provided in the device's datasheet, according to SPI operation [33]. It is important to note that a pull-up resistor on the chip-select signal is required to ensure the proper functioning of the SPI protocol, allowing this line to always stay in logic "1" outside the

communication time. The DRDY, INT2, and INT1 signals, which can be programmed and may be useful for future modifications, have been directly connected to the ESP32 GPIO pins. Internally, the interrupt pins can map to 1 of the 5 bits of the status register, which allows versatility in the software driver design.

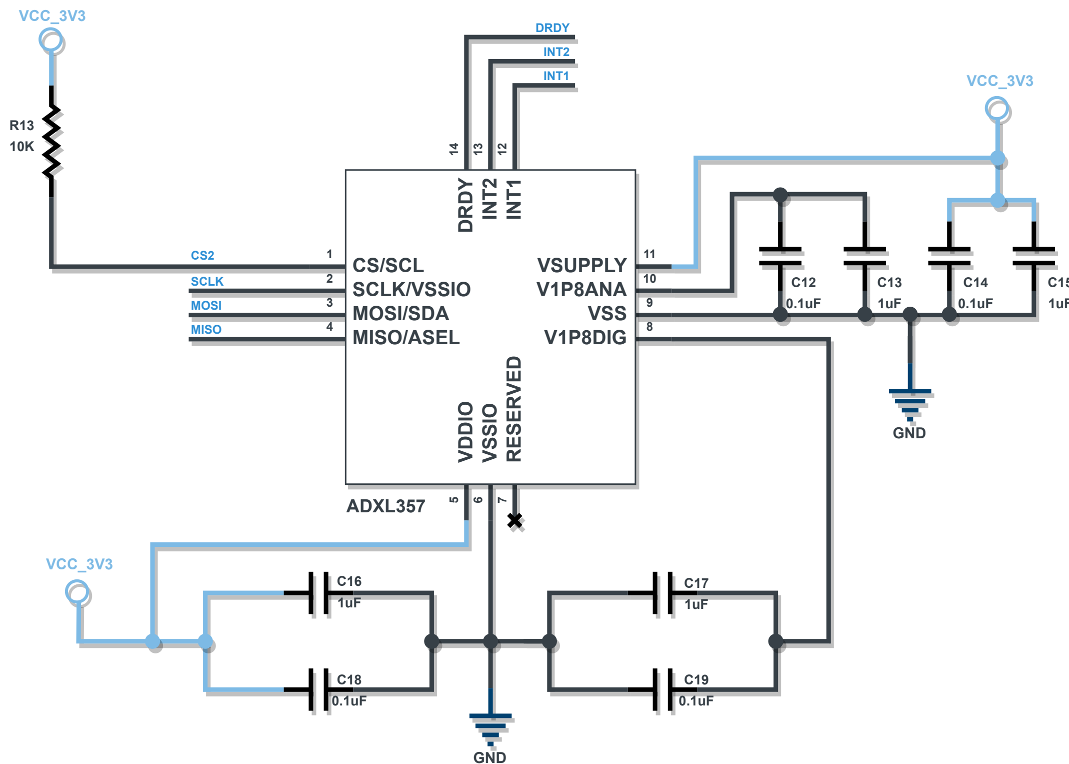


Figure 21: ADXL357 schematic

4.1.3 MicroSD Card

In the previous phase, the micro SD card was the device and method chosen to save the colision data. In order to connect a micro SD card to the PCB, the board must support this type of connection through the use of a micro SD connector. The microSD connector has different types of schematics that can be used. This particulary one, as illustrated in Figure 22, is designed to communicate with the micro SD card via SPI. As it is possible to see, the schematic incorporates several pull-up resistors on most of the SPI signals, which will prevent floating states in these signals. If a pull-up resistor is not used on the chip-select line for example, it may lead to communication errors when multiple devices are connected. The card detection signals in this schematic, which are used to detect the presence of a microSD card, are not necessary to use, which explains their connection to the ground.

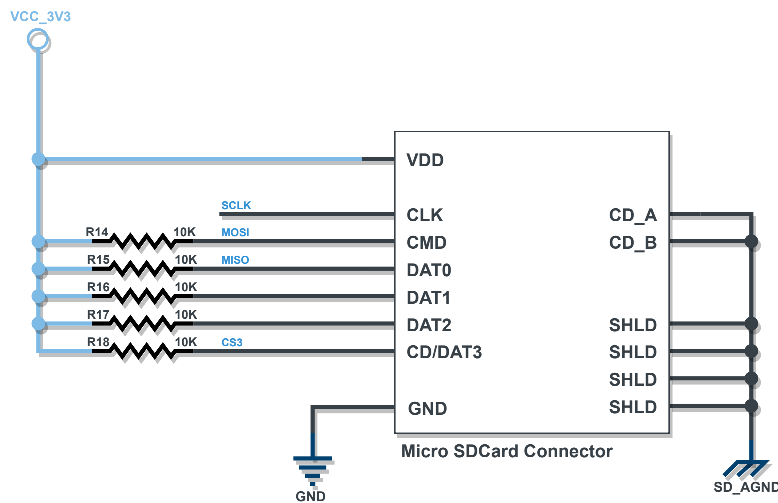


Figure 22: MicroSD Card schematic

In the schematic is not actually possible to understand how the connections to a real microSD Card are linked, so in figure 23 is presented the pin number identification used and in table 5 is represented the respective connection via SPI using the pin number of figure 23.

Figure 23: MicroSD Pin Number Order

Table 5: MicroSD pinout

Pin	SPI
1	-
2	CS
3	Data In
4	VDD
5	SCLK
6	VSS
7	Data Out
8	-

4.1.4 MPU-6881

Figure 24 illustrates the schematic used to operate with the MPU-6881 device, including all necessary connections. This sensor was described and chosen in the system specification phase and, similar to

other schematics, the sensor circuit is based on the manufacturer's recommendations and uses the SPI protocol for communication [34]. A pull-up resistor has been included on the chip-select signal. The other SPI signals are connected to the CPU sheet and this connection is only represented through labels. In addition, the FSYNC and INT0 signals have been connected to the CPU in case they are needed for future logic requirements.

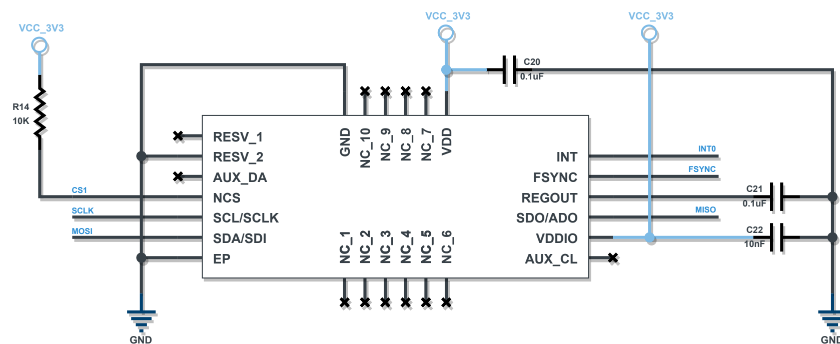


Figure 24: MPU-6881 schematic

4.1.5 CPU

In Figure 25, the schematic for the CPU operation is shown. This schematic follows the ESP32 datasheet recomendations [35]. In this circuit, starting from the pinout, it is possible to observe that the most pins are connected to a bus that will be linked to the connectors sheet. The remain pins are the power pin, reset pin, enable pin and the UART (Universal Asynchronous Receiver/Transmitter) pins.

The power line includes several capacitors to filter out the noise and smooth out voltage fluctuations. The larger capacitor helps to stabilize lower-frequency variations in the supply voltage, while the smaller capacitor effectively helps to filter out the high-frequency noise on the power line.

The reset and boot pins are both directly connected to a button. In the reset signal line, a RC filter is also used to improve the reliability and accuracy of the enabled signal by removing noise and interference. By default, the signal logic is "1", and pressing the button initiates a reset of the CPU.

The boot pin is just a verification pin checked during the reset process, also called strapping pin. If the boot pin is set to logic "1", the CPU will not enter a downloading mode and will simply reset the CPU state. However, if the reset occurs and the boot pin is "0", the CPU will enter a downloading mode and will wait for the binary software to be loaded via UART pins. This boot pin does not need to be connected to the power line because the ESP32 has a weak pull-up on this pin. The button associated to this pin is used to perform a manual downloading mode, however its possible through some logic to automaticaly put ESP32 in downloading mode. This is done by changing the RTS and DTR signal from USB protocol and is usually done by software. In this case the button is used in case of some failure or to debug errors during the firmware installition and configuration process.

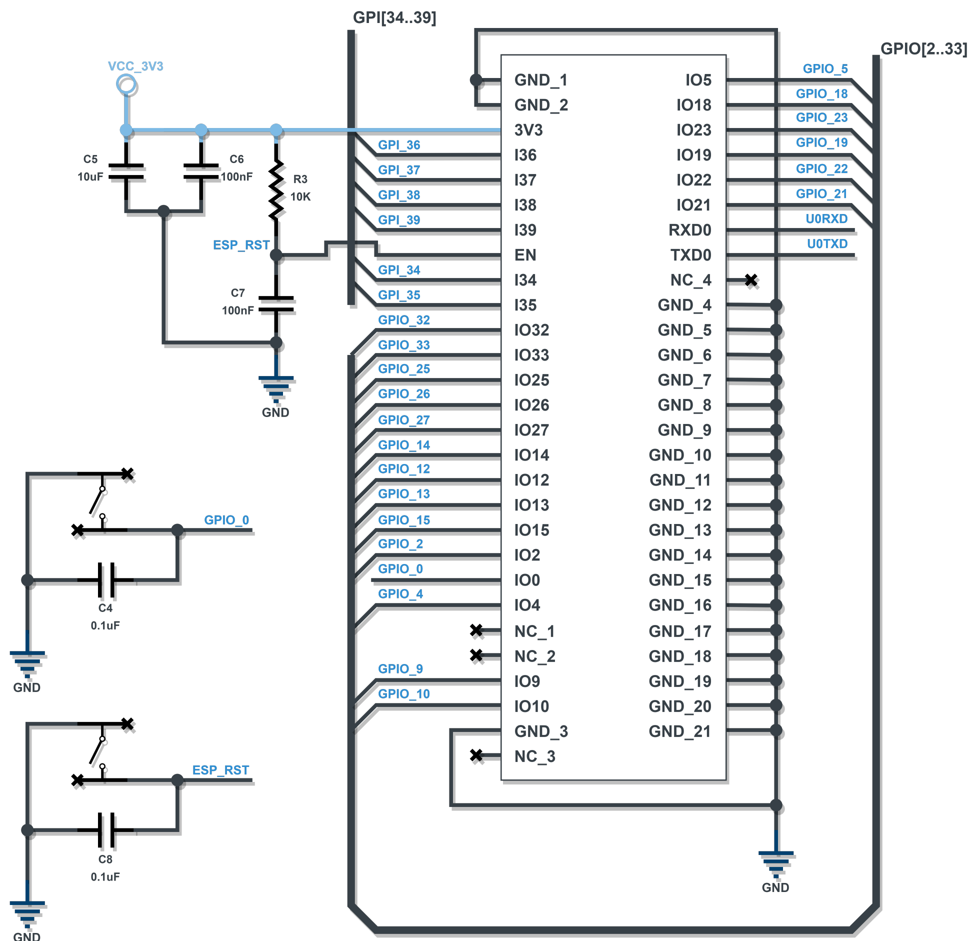


Figure 25: CPU schematic

4.1.6 Micro USB Connector

In chapter 3, the micro USB was chosen to program the microcontroller. Also in the LDO section, for pratical reasons, was chosen using the micro USB to power up the entire board. In this case, to support this connection, figure 26 shows the schematic for the micro USB connector that accomplish the desired purposes. The majority of the elements shown in the schematic are protection components, which are used to protect the system from several types of damage. These components include a fuse to protect against excess current, a Schottky diode to prevent voltage spikes from damaging the system, a ferrite to filter out power line noise, and TVS diodes in all lines to protect against electrical transients. All of these components are necessary for ensuring the reliability and longevity of the system.

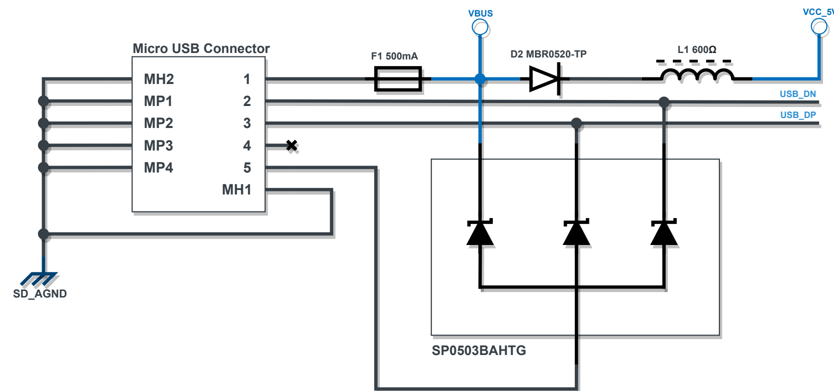


Figure 26: MicroUSB connector schematic

4.1.7 CP2102N

Another important decision was to use a “UART bridge” to facilitate communication between the microprocessor and the PC. This device, also known as a UART converter, acts as an intermediary between the two devices, allowing them to communicate using different interfaces. In this case, the PC sends data via USB, and the UART bridge converts it to the UART protocol, allowing it to be received by the microprocessor. The chip commonly used for this purpose on ESP32 development boards is the CP2102N. Although it would have been possible to use a device with a built-in debugger, it was decided to use an external debugger if necessary. The schematic for this component, shown in Figure 27, follows the recommendations provided in the component datasheet, ensuring that is properly configured and works as intended [36]. The UART lines, Tx and Rx, are then connected directly to the UART0 port on the ESP32 CPU.

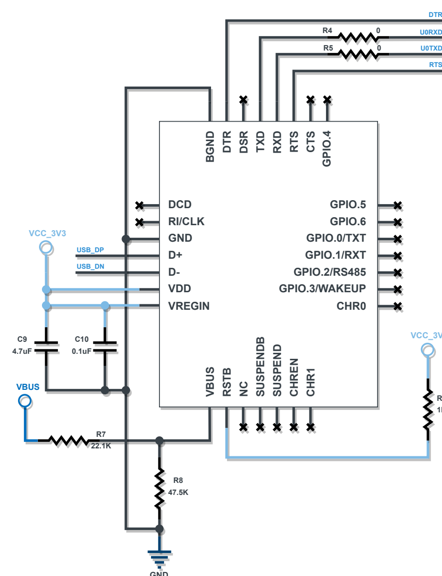


Figure 27: CP2102N schematic

In the CPU section, was refered that the automaticly downloading mode is done by changing the RTS and DTR signal from USB protocol. Figure 28 shows the logic used to automaticly put the CPU in the downloading mode. The RTS and DTR signals are used to synchronize the communication between the devices and to put the ESP32 in a programming state where it is ready to receive and execute instructions.

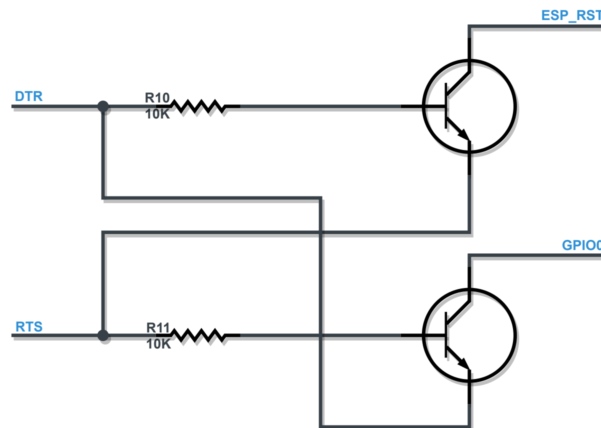


Figure 28: Programming logic schematic

4.2 Software Considerations

The realization of an embedded system is only complete with the construction of the logical part of the program that will run on the developed hardware. For the development of this software part, flowcharts and diagrams will be used to organize the ideas. In this section, the tasks overview will be presented, as well as a detailed explanation of the logic of each tasks and drivers.

4.2.1 Task Overview

The use of tasks in the software design provides several benefits, such as improved organization, resource utilization, responsiveness, and fault isolation. By breaking down the data operations into separate tasks, the code can be more modular and easier to maintain and scale. The ability to schedule tasks to run at different times can improve the responsiveness of the system and allow it to react quickly to changes. While the prototype creation did not require multitasking usage due to its sequential approach, the decision was made to implement a multitasking system to improve the code's scalability. This will make it easier to add additional functionalities or acquisition tasks in the future.

Figure 29 presents an overview of the software tasks used and the logic between them. First, three threads are used to perform the data operations, where SPI transactions are established to execute reading and writing operations. Three queues that act as FIFO (First In, First Out) buffers are used to coordinate the tasks in the system. These buffers are connected to the main task, which is responsible for performing

collision detection. The FIFO buffers ensure that the threads are executed in a sequential order, allowing the collision detection logic to function correctly.

The detection task presents itself as the control task of the whole system. This task will receive the data from the two accelerometers, and this data will be combined and filtered. Then, the Kalman filter and the detection collision algorithm are applied. After this task detects a crash, the SD card task is signalised to write the data on the microSD card. Before this operation, this thread will get the time from RTC and save this data too. Some logic is included in the collision task to save data after and before the crash.

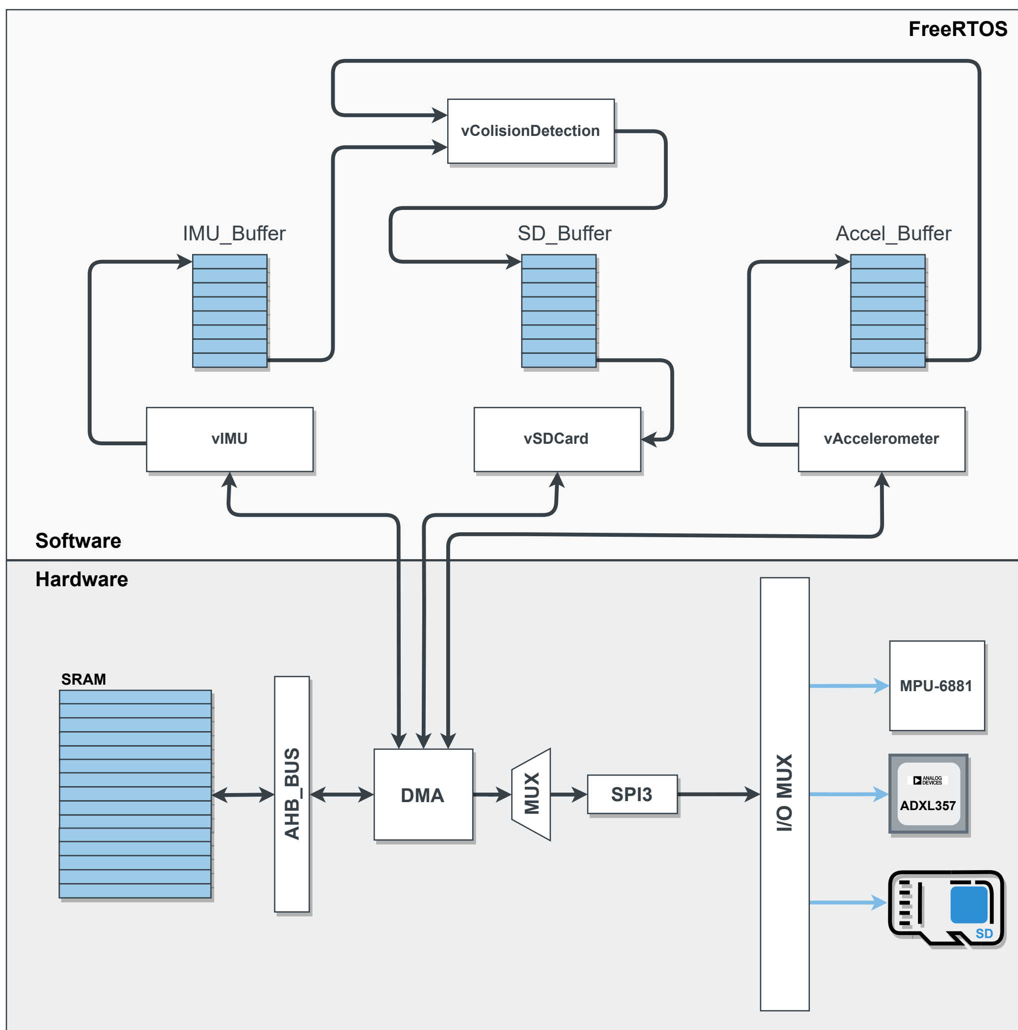


Figure 29: Tasks Overview

This task overview section is only complete with the presentation of how the data will be synchronize, as well as the program flow (Figure 30). First the vaccelerometer task will be notified by a interrupt timer that will be schedule with a sampling frequency of 1KHz. After the vaccelerometer task aquire one sample, it will notify the IMU task that will do exactly the same but with the other sensor. This sequential order, guarantee that the control task will have a sample of each sensor to combine. The acquisition of the two sensors could be done in parallel, but as both sensors use the same SPI resource, one task always ends

up waiting for the other. This way, the option taken was to approach the problem in a simpler and faster way.

The vIMU task is responsible to notify the control task in the end of the sample aquisition mode. As meantioned earlier, if the control task doesn't detect a colision, the sensors tasks will again wait for a notification to sample more data. However, if a colision occurs, this task should notify the vSD_Card task to start the save data program.

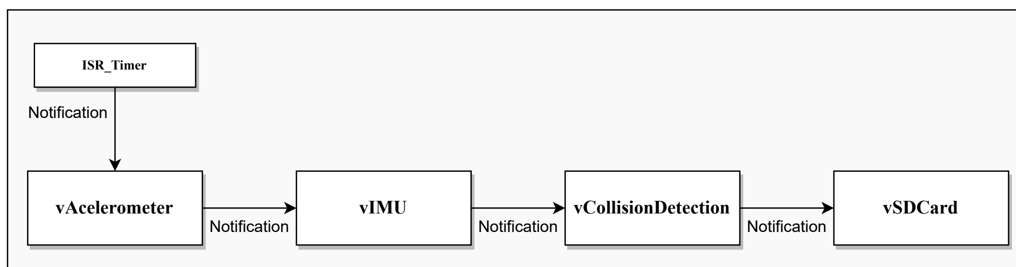


Figure 30: Tasks Synchronization

4.2.2 Drivers

In system specification, all the software architecture necessary to solve the application requirements was determined. The use of three devices connected by SPI requires the creation of a driver module to manage read-and-write transactions with the board sensors. The ESP-IDF (Espressif IoT Development Framework) essentially contains an API (Application Programming Interface) with drivers that communicate with the SPI peripherals on the ESP32. These drivers allow the bus to be initialised and SPI transactions to be performed according to several parameters, such as address bits, command bits, and buffers for receiving or sending data. The SPI module creation should implement functions considering the ESP32 API driver. These functions will allow devices to be inserted and removed in the bus, reading and writing from these devices.

ADXL357

The driver that will handle the communications with the ADXL357 accelerometer, have to consider two factors in his design. The first one is the SPI data format that the device needs, specially, the necessary address format and which bit from that address indicates a read or write operation. The other factor is the conversion from the raw read data to the g's unit. When a read operation is preformed from the acceleration registers, that data should be converted to g's.

To accomplish the first requirement of this driver, first is necessary to identify the SPI format of ADXL357 from the datasheet. Figure 31 presents the SPI communication diagram of the ADXL357 accelerometer For reading or writing to the registers of this accelerometer, the processor must send seven bits with the intended address and the last bit that will specify if the master wants to read or write. The master, which in

this case is the processor, must send this address byte through the MOSI line, and the accelerometer, as a slave, sends the information of this register through the MISO line. The processor is responsible for the entire frame creation, which means that the chip selection, clock, and MOSI frames creation are entirely responsability of the master. This way, if the processor wants to read the same register several times, it is enough to extend the SCLK for one more byte, and the slave will continue to send the data.

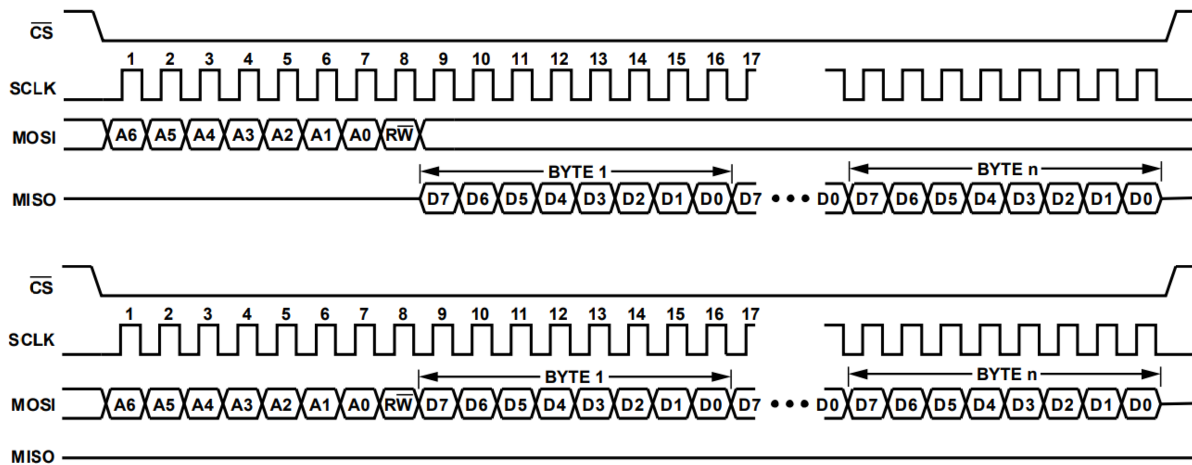


Figure 31: SPI Format ADXL357

The conversion from raw data to g's is the second factor to fulfil and to perform this conversion is necessary to get the sensitivity factor from the datasheet. This value will allow to convert the raw data according with the selected range. Using the values shown in figure 32, the conversion is done by dividing the raw data by correct range sensitivity.

SENSITIVITY[1]	Each axis				
X-Axis, Y-Axis, and Z-Axis Sensitivity	±10 *g*	47,104	51,200	55,296	LSB/*g*
	±20 *g*	23,552	25,600	27,648	LSB/*g*
	±40 *g*	11,776	12,800	13,824	LSB/*g*

Figure 32: ADXL357 Sensitivity

MPU-6881

The MPU-6881 SPI format, shown in figure 33, follows a diffrent address scheme, because the bit that indicates a read or write operation is now the MSB(Most significant bit). Depending on the master intentions, the data bytes are received in the MISO line or sent by the MOSI line to this device. Remember that this "chip select" is a different signal from the one used for the ADXL357 accelerometer, so the processor must handle the additions of these devices on the bus, indicating the corresponding pin.

SPI Address format

MSB							**LSB**
R/W	A6	A5	A4	A3	A2	A1	A0

SPI Data format

MSB							**LSB**
D7	D6	D5	D4	D3	D2	D1	D0

Figure 33: MPU-6881 SPI Format

The convertion from the raw data to G's is again performed using the sensitivity factor from the device's datasheet. Then the acceleration value is obtaining by dividing the raw data by the sensitivity, according with the operating range. The device sensitivity is presented in the figure 34.

Sensitivity Scale Factor	AFS_SEL=0		16,384		LSB/*g*
	AFS_SEL=1		8,192		LSB/*g*
	AFS_SEL=2		4,096		LSB/*g*
	AFS_SEL=3		2,048		LSB/*g*

Figure 34: MPU-6881 sensitivity

4.2.3 Kalman

In order to effectively implement the Kalman filter in this project, it is crucial to establish the model in which the filter will operate. The prediction phase uses two key equations: the state extrapolation equation and the covariance extrapolation equation. These equations are used in conjunction to estimate the position, velocity, and acceleration of the system at any given time. As outlined in the design phase, the initial step in this process is to compute the state extrapolation equation. Additionally, as this project does not involve any control input from car acceleration, matrix U is not required.

The state extrapolation equation, as presented, is given by:

$$X_{kp} = AX_{k-1} + \cancelto{0}{BU_k} + W_k$$

To predict the future state of the system, it is imperative to define the state transition matrix, which describes the evolution of the system state over time. The state transition matrix can be calculated using kinematics equations, which allow for the calculation of the position, velocity, and acceleration of the car at any given time. These kinematics equations are given by:

$$\begin{cases} X_k = X_{k-1} + \dot{X}_{k-1}\Delta t + \frac{1}{2}\ddot{X}_{k-1}\Delta t^2 \\ \dot{X}_k = \dot{X}_{k-1} + \ddot{X}_{k-1}\Delta t \\ \ddot{X}_k = \ddot{X}_{k-1} \\ Y_k = Y_{k-1} + \dot{Y}_{k-1}\Delta t + \frac{1}{2}\ddot{Y}_{k-1}\Delta t^2 \\ \dot{Y}_k = \dot{Y}_{k-1} + \ddot{Y}_{k-1}\Delta t \\ \ddot{Y}_k = \ddot{Y}_{k-1} \\ Y_k = Z_{k-1} + \dot{Z}_{k-1}\Delta t + \frac{1}{2}\ddot{Z}_{k-1}\Delta t^2 \\ \dot{Z}_k = \dot{Z}_{k-1} + \ddot{Z}_{k-1}\Delta t \\ \ddot{Z}_k = \ddot{Z}_{k-1} \end{cases}$$

The state transition matrix A and the state extrapolation equation, as presented in the equations below, are crucial components in the adaptation of the Kalman filter for use in this dissertation. The matrix A, obtained through the use of kinematics equations, represents the evolution of the system state over time and is used to update the current state of the system by multiplying it with the previous state of the system. The state extrapolation equation combines the state transition matrix A and the previous state of the system to estimate the position, velocity, and acceleration of the system at any given time in the project.

$$A = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \dot{x}_k \\ \ddot{x}_k \\ \dddot{x}_k \\ \dot{y}_k \\ \ddot{y}_k \\ \dddot{y}_k \\ \dot{z}_k \\ \ddot{z}_k \\ \dddot{z}_k \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x}_{k-1} \\ \ddot{x}_{k-1} \\ \dddot{x}_{k-1} \\ \dot{y}_{k-1} \\ \ddot{y}_{k-1} \\ \dddot{y}_{k-1} \\ \dot{z}_{k-1} \\ \ddot{z}_{k-1} \\ \dddot{z}_{k-1} \end{bmatrix}$$

To conclude the prediction phase, the covariance extrapolation equation is utilized to estimate the uncertainty in the state estimate at time k, based on the uncertainty at time k-1. The equation for this calculation is represented as:

$$P_{\text{kp}} = AP_{\text{k-1}}A^T + Q_{\text{k}}$$

Where P_{kp} represents the uncertainty (covariance) matrix of the current state estimation, calculated by combining the previous state's uncertainty matrix, P_{k-1}, with the state transition matrix A and the process noise matrix Q_k. This calculation plays a crucial role in determining the accuracy of the state estimate and is calculated by:

$$P_{\text{kp}} = \begin{bmatrix} P_x & P_{x\dot{x}} & P_{x\ddot{x}} & P_{xy} & P_{x\dot{y}} & P_{x\ddot{y}} & P_{xz} & P_{x\dot{z}} & P_{x\ddot{z}} \\ P_{\dot{x}x} & P_{\dot{x}} & P_{\dot{x}\ddot{x}} & P_{\dot{x}y} & P_{\dot{x}\dot{y}} & P_{\dot{x}\ddot{y}} & P_{\dot{x}z} & P_{\dot{x}\dot{z}} & P_{\dot{x}\ddot{z}} \\ P_{\ddot{x}x} & P_{\ddot{x}\dot{x}} & P_{\ddot{x}} & P_{\ddot{x}y} & P_{\ddot{x}\dot{y}} & P_{\ddot{x}\ddot{y}} & P_{\ddot{x}z} & P_{\ddot{x}\dot{z}} & P_{\ddot{x}\ddot{z}} \\ P_{yx} & P_{y\dot{x}} & P_{y\ddot{x}} & P_y & P_{y\dot{y}} & P_{y\ddot{y}} & P_{yz} & P_{y\dot{z}} & P_{y\ddot{z}} \\ P_{\dot{y}x} & P_{\dot{y}\dot{x}} & P_{\dot{y}\ddot{x}} & P_{\dot{y}y} & P_{\dot{y}} & P_{\dot{y}\ddot{y}} & P_{\dot{y}z} & P_{\dot{y}\dot{z}} & P_{\dot{y}\ddot{z}} \\ P_{\ddot{y}x} & P_{\ddot{y}\dot{x}} & P_{\ddot{y}\ddot{x}} & P_{\ddot{y}y} & P_{\ddot{y}\dot{y}} & P_{\ddot{y}} & P_{\ddot{y}z} & P_{\ddot{y}\dot{z}} & P_{\ddot{y}\ddot{z}} \\ P_{zx} & P_{z\dot{x}} & P_{z\ddot{x}} & P_{zy} & P_{z\dot{y}} & P_{z\ddot{y}} & P_z & P_{z\dot{z}} & P_{z\ddot{z}} \\ P_{\dot{z}x} & P_{\dot{z}\dot{x}} & P_{\dot{z}\ddot{x}} & P_{\dot{z}y} & P_{\dot{z}\dot{y}} & P_{\dot{z}\ddot{y}} & P_{\dot{z}z} & P_{\dot{z}} & P_{\dot{z}\ddot{z}} \\ P_{\ddot{z}x} & P_{\ddot{z}\dot{x}} & P_{\ddot{z}\ddot{x}} & P_{\ddot{z}y} & P_{\ddot{z}\dot{y}} & P_{\ddot{z}\ddot{y}} & P_{\ddot{z}z} & P_{\ddot{z}\dot{z}} & P_{\ddot{z}} \end{bmatrix}$$

The main diagonal elements of a matrix represent the variances of an estimation. In this case, the matrix represents the variances of the X, Y, and Z coordinate positions, velocities, and accelerations. Specifically, $P_x, P_{\dot{x}}, P_{\ddot{x}}$ represents the variance of the X coordinate position, velocity, and acceleration estimation respectively. Similarly, $P_y, P_{\dot{y}}, P_{\ddot{y}}$ and $P_z, P_{\dot{z}}, P_{\ddot{z}}$ represent the variances of the Y and Z coordinate estimations respectively.

It's important to note that the off-diagonal elements of the matrix are covariances, which measure the relationship between the different variables. In this case, it is assumed that the estimation errors in the

X, Y, and Z axes are not correlated, so the mutual terms can be set to zero. This means that the estimate uncertainty in matrix form is given by:

$$P_{\mathrm{kp}} = \begin{bmatrix} P_x^2 & P_{x\dot{x}}^2 & P_{x\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\dot{x}x}^2 & P_{\dot{x}}^2 & P_{\dot{x}\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\ddot{x}x}^2 & P_{\ddot{x}\dot{x}}^2 & P_{\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_y^2 & P_{y\dot{y}}^2 & P_{y\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\dot{y}y}^2 & P_{\dot{y}}^2 & P_{\dot{y}\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\ddot{y}y}^2 & P_{\ddot{y}\dot{y}}^2 & P_{\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_z^2 & P_{z\dot{z}}^2 & P_{z\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\dot{z}z}^2 & P_{\dot{z}}^2 & P_{\dot{z}\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\ddot{z}z}^2 & P_{\ddot{z}\dot{z}}^2 & P_{\ddot{z}}^2 \end{bmatrix}$$

The next step in the derivation process is to calculate the process noise matrix. The variance of the process noise plays a crucial role in determining the performance of the Kalman filter. A low value for Q can lead to lag errors, while a high value can result in noisy estimations that closely follow the measurements. There are also two distinct models for environmental process noise to consider:

- Discrete noise model
- Continuous noise model

In this example, the discrete noise model is assumed, where the noise is different at each time sample but is constant between time samples. The process noise matrix to this model is represented by:

$$Q_{\mathrm{k}} = \begin{bmatrix} \sigma_x^2 & \sigma_{x\dot{x}}^2 & \sigma_{x\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ \sigma_{\dot{x}x}^2 & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ \sigma_{\ddot{x}x}^2 & \sigma_{\ddot{x}\dot{x}}^2 & \sigma_{\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_y^2 & \sigma_{y\dot{y}}^2 & \sigma_{y\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{\dot{y}y}^2 & \sigma_{\dot{y}}^2 & \sigma_{\dot{y}\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_{\ddot{y}y}^2 & \sigma_{\ddot{y}\dot{y}}^2 & \sigma_{\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \sigma_z^2 & \sigma_{z\dot{z}}^2 & \sigma_{z\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & \sigma_{\dot{z}z}^2 & \sigma_{\dot{z}}^2 & \sigma_{\dot{z}\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & \sigma_{\ddot{z}z}^2 & \sigma_{\ddot{z}\dot{z}}^2 & \sigma_{\ddot{z}}^2 \end{bmatrix}$$

Assuming that X, Y and Z are not correlated, the process noise matrix becomes:

$$Q_{\mathrm{k}} = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^2}{2} & \Delta t & 1 \end{bmatrix} \sigma_a^2$$

where Δt is the time between sucessive measurements and σ_a^2 is the random variance in acceleration. With the transition matrix and process noise matrix derived, it is now possible to derive the covariance extrapolation equation:

$$P_{\mathrm{kp}} = AP_{\mathrm{k\text{-}1}}A^T + Q_{\mathrm{k}} \Leftrightarrow$$

$$\Leftrightarrow \begin{bmatrix} P_x^2 & P_{x\dot{x}}^2 & P_{x\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\dot{x}x}^2 & P_{\dot{x}}^2 & P_{\dot{x}\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\ddot{x}x}^2 & P_{\ddot{x}\dot{x}}^2 & P_{\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_y^2 & P_{y\dot{y}}^2 & P_{y\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\dot{y}y}^2 & P_{\dot{y}}^2 & P_{\dot{y}\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\ddot{y}y}^2 & P_{\ddot{y}\dot{y}}^2 & P_{\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_z^2 & P_{z\dot{z}}^2 & P_{z\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\dot{z}z}^2 & P_{\dot{z}}^2 & P_{\dot{z}\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\ddot{z}z}^2 & P_{\ddot{z}\dot{z}}^2 & P_{\ddot{z}}^2 \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_{x_{k-1}}^2 & P_{x\dot{x}_{k-1}}^2 & P_{x\ddot{x}_{k-1}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\dot{x}x_{k-1}}^2 & P_{\dot{x}_{k-1}}^2 & P_{\dot{x}\ddot{x}_{k-1}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\ddot{x}x_{k-1}}^2 & P_{\ddot{x}\dot{x}_{k-1}}^2 & P_{\ddot{x}_{k-1}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{y_{k-1}}^2 & P_{y\dot{y}_{k-1}}^2 & P_{y\ddot{y}_{k-1}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\dot{y}y_{k-1}}^2 & P_{\dot{y}_{k-1}}^2 & P_{\dot{y}\ddot{y}_{k-1}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\ddot{y}y_{k-1}}^2 & P_{\ddot{y}\dot{y}_{k-1}}^2 & P_{\ddot{y}_{k-1}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{z_{k-1}}^2 & P_{z\dot{z}_{k-1}}^2 & P_{z\ddot{z}_{k-1}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\dot{z}z_{k-1}}^2 & P_{\dot{z}_{k-1}}^2 & P_{\dot{z}\ddot{z}_{k-1}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\ddot{z}z_{k-1}}^2 & P_{\ddot{z}\dot{z}_{k-1}}^2 & P_{\ddot{z}_{k-1}}^2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \Delta t & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2}\Delta t^2 & \Delta t & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta t & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}\Delta t^2 & \Delta t & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \Delta t & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2}\Delta t^2 & 0 & 1 \end{bmatrix} + \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{\Delta t^2}{2} & \Delta t & 1 \end{bmatrix} \sigma_a^2$$

The parameter σ_a^2 represents the random variance on acceleration, which will be further explained in the next section.

Pre-processing step

Before advancing to the update phase, let's define how the data from the two accelerometers will be combined and how this will affect some other parameters, like the variance of acceleration. To fuse the sensors data, it will combine measurements from multiple sensors based on the variances of each sensor. The method uses a weighted average of the sensor measurements, where the weighting factors are the inverse of the variances of the sensors. So the combined aceleration in each axis will be calculated by:

$$\bar{a} = \frac{\frac{1}{\sigma_{a_1}^2} a_1 + \frac{1}{\sigma_{a_2}^2} a_2}{\frac{1}{\sigma_{a_1}^2} + \frac{1}{\sigma_{a_2}^2}}$$

This weighted formula makes use of accelerometers variance which is calculated by using the noise spectral density from the devices datasheets and applying the formula for converting noise spectral density to the variance that is given by:

$$\sigma_a^2 = \int_{f_1}^{f_2} S_n(f)df$$

Where:

σ^2 is the variance of the noise

$S_n(f)$ is the noise spectral density

f_1 is the lower frequency limit

f_2 is the upper frequency limit

This formula is based on the concept that the variance is the integral of the power spectral density over the frequency range of interest. Starting from the ADXL357 datasheet, the spectral density in all axis for the 40G range is 110 g/$\sqrt{Hz}$.

Deriving using the formula:

$$\sigma_a^2 = \int_{f_1}^{f_2} S_n(f)df \Leftrightarrow$$

$$\Leftrightarrow \sigma_a^2 = \int_{10}^{1000} \frac{110 \times 10^{-6}}{\sqrt{(f)}} df = 110 \times 10^{-6} \int_{10}^{1000} f^{-\frac{1}{2}} df \Leftrightarrow$$

$$\Leftrightarrow \sigma_a^2 = \left[110 \times 10^{-6} \times \frac{1}{1/2} f^{1/2}\right]_{10}^{1000} = 110 \times 10^{-6} \times 2 \times \left[\sqrt{1000} - \sqrt{10}\right] \Leftrightarrow$$

$$\Leftrightarrow \sigma_a^2 = 0.006261$$

The MPU accelerometer has a noise spectral density of 400 g/$\sqrt{Hz}$.

Applying the formula:

$$\sigma_a^2 = \int_{f_1}^{f_2} S_n(f)df \Leftrightarrow$$

$$\Leftrightarrow \sigma_a^2 = \int_{10}^{1000} \frac{400 \times 10^{-6}}{\sqrt{(f)}} df = 400 \times 10^{-6} \int_{10}^{1000} f^{-\frac{1}{2}} df \Leftrightarrow$$

$$\Leftrightarrow \sigma_a^2 = \left[400 \times 10^{-6} \times \frac{1}{1/2} f^{1/2}\right]_{10}^{1000} = 400 \times 10^{-6} \times 2 \times \left[\sqrt{1000} - \sqrt{10}\right] \Leftrightarrow$$

$$\Leftrightarrow \sigma_a^2 = 0.0228$$

Now, the fusion variance of acceleration, assuming the sensors are not correlated, is given by:

$$\sigma_{combined}^2 = \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}} = \frac{1}{\frac{1}{\sigma_{ADXL}^2} + \frac{1}{\sigma_{MPU}^2}} = \frac{1}{\frac{1}{0.006261} + \frac{1}{0.0228}} \Leftrightarrow$$

$$\Leftrightarrow \sigma_{combined}^2 = 0.004912109012$$

Now the combined acceleration in each axis will be calculated by:

$$\bar{a} = \frac{\frac{1}{0.006261} a_1 + \frac{1}{0.00228} a_2}{\frac{1}{0.006261} + \frac{1}{0.00228}}$$

The measurement equation

In the measurement equation, the Kalman filter uses the observed values from the system and compares them to the predicted values. The observation matrix, C, is used to extract the relevant information from the measured values. In this project, matrix C is set up to extract the x, y, and z accelerations from the measured sensor values.

The generalized measurement equation in the matrix form is given by:

$$Y_k = CY_{k_{measured}} + Z_k$$

where Y_k is the output matrix with only acceleration measured values, $Y_{kmeasured}$ is a hidden system state matrix that contains all the observed variable values that can be provided from sensors or not. C is the observation matrix that will select which values on the $Y_{kmeasured}$ are provided from sensors, which in this case is the acceleration from x,y and z directions. So, in this case, the observation matrix C is given by:

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and the $Y_{kmeasured}$ is given by:

$$Y_{k_{measured}} = \begin{bmatrix} 0 \\ 0 \\ x_{measured} \\ 0 \\ 0 \\ y_{measured} \\ 0 \\ 0 \\ z_{measured} \end{bmatrix}$$

This allow to derive the measurement equation that will be given by:

$$Y_k = CY_{k_{measured}} + Z_k \Leftrightarrow$$

$$\Leftrightarrow \begin{bmatrix} x_{measured} \\ y_{measured} \\ z_{measured} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ x_{measured} \\ 0 \\ 0 \\ y_{measured} \\ 0 \\ 0 \\ z_{measured} \end{bmatrix}$$

The measurement uncertainty

In real-life applications, the measurement uncertainty can differ between measurements. In many systems, the measurement uncertainty depends on the measurement SNR (signal-to-noise ratio), the angle between the sensor (or sensors) and target, signal frequency, and many other parameters. The measurement covariance matrix is defined as:

$$R = \begin{bmatrix} \sigma_{x_m}^2 & \sigma_{y_m}\sigma_{x_m} & \sigma_{z_m}\sigma_{x_m} \\ \sigma_{x_m}\sigma_{x_m} & \sigma_{y_m}^2 & \sigma_{z_m}\sigma_{y_m} \\ \sigma_{x_m}\sigma_{z_m} & \sigma_{y_m}\sigma_{z_m} & \sigma_{z_m}^2 \end{bmatrix}$$

This matrix represents the uncertainty associated of the measured values. If x,y and z measurements are not correlated, the matrix assumes a diagonal form. Since the measured values are from MEMS acceleroemters, and the axis are uncorrelated, this matrix is given by:

$$R = \begin{bmatrix} \sigma_{x_m}^2 & 0 & 0 \\ 0 & \sigma_{y_m}^2 & 0 \\ 0 & 0 & \sigma_{z_m}^2 \end{bmatrix}$$

The Kalman gain

After defining the measurement uncertainty matrix, all the necessary matrix are already defined and the kalman gain can be derived by:

$$K = \frac{P_{kp}C^T}{CP_{kp}C^T + R} \Leftrightarrow$$

$$\Leftrightarrow K = \begin{bmatrix} P_x^2 & P_{x\dot{x}}^2 & P_{x\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\dot{x}x}^2 & P_{\dot{x}}^2 & P_{\dot{x}\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\ddot{x}x}^2 & P_{\ddot{x}\dot{x}}^2 & P_{\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_y^2 & P_{y\dot{y}}^2 & P_{y\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\dot{y}y}^2 & P_{\dot{y}}^2 & P_{\dot{y}\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\ddot{y}y}^2 & P_{\ddot{y}\dot{y}}^2 & P_{\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_z^2 & P_{z\dot{z}}^2 & P_{z\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\dot{z}z}^2 & P_{\dot{z}}^2 & P_{\dot{z}\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\ddot{z}z}^2 & P_{\ddot{z}\dot{z}}^2 & P_{\ddot{z}}^2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x^2 & P_{x\dot{x}}^2 & P_{x\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\dot{x}x}^2 & P_{\dot{x}}^2 & P_{\dot{x}\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ P_{\ddot{x}x}^2 & P_{\ddot{x}\dot{x}}^2 & P_{\ddot{x}}^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_y^2 & P_{y\dot{y}}^2 & P_{y\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\dot{y}y}^2 & P_{\dot{y}}^2 & P_{\dot{y}\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & P_{\ddot{y}y}^2 & P_{\ddot{y}\dot{y}}^2 & P_{\ddot{y}}^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_z^2 & P_{z\dot{z}}^2 & P_{z\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\dot{z}z}^2 & P_{\dot{z}}^2 & P_{\dot{z}\ddot{z}}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & P_{\ddot{z}z}^2 & P_{\ddot{z}\dot{z}}^2 & P_{\ddot{z}}^2 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} \sigma_{x_m}^2 & 0 & 0 \\ 0 & \sigma_{y_m}^2 & 0 \\ 0 & 0 & \sigma_{z_m}^2 \end{bmatrix} \right)^{-1}$$

The state update equation and covariance update equation

After obtaining the kalman matrix, all the necessary matrix's has already been defined. Then, the state update equation is calculated by:

$$X_{\mathrm{k}} = X_{\mathrm{kp}} + K(Y_{\mathrm{k}} - HX_{\mathrm{kp}})$$

and the covariance update equation is calculated by:

$$P_k = (I - KH)P_{kp}(I - KH)^T + KRK^T$$

After completed all this stages is time to prepare the next iteration, where the current state matrix and the covariance matrix becomes the previous values.

$$X_{\mathrm{k}} \rightarrow X_{\mathrm{k\text{-}1}}$$

$$P_{\mathrm{k}} \rightarrow P_{\mathrm{k\text{-}1}}$$

4.2.4 FreeRTOS Tasks

By developing modules with functions for each device, the software structure becomes more user-friendly and easier to understand. This modularization enables a more intuitive task structure for the operating system and facilitates redesigning. The designer can structure which tasks are required for data acquisition, processing, and connection elements between threads. As a result, five tasks were created, with three of them handling SPI device data, the control task and the main task that will create all the other threads.

vAccelerometer

The main task goal is to perform the data acquisition of the ADXL357 accelerometer and provide that data in a queue that acts as a FIFO. Figure 35 presents the task flowchart. This task is responsible for initialising the variables used and starting the ADXL357 accelerometer. The accelerometer initialisation is done by using a driver created for that purpose, which prepares the accelerometer and turns it into the desired configuration to start readings. After the start-up process, it's time to run the data acquisition loop. Inside this loop, it should wait for a timer notification to start reading data. After the alert, the thread uses a function to reserve the mutex associated with the SPI resource, making no other tasks use the peripheral until this thread releases it. It then performs a data sample to an array previously created. After the data acquisition, the next step is to pass these data to a buffer. This buffer is the Accel_Buffer that serves to pass the data to the collision detection task.

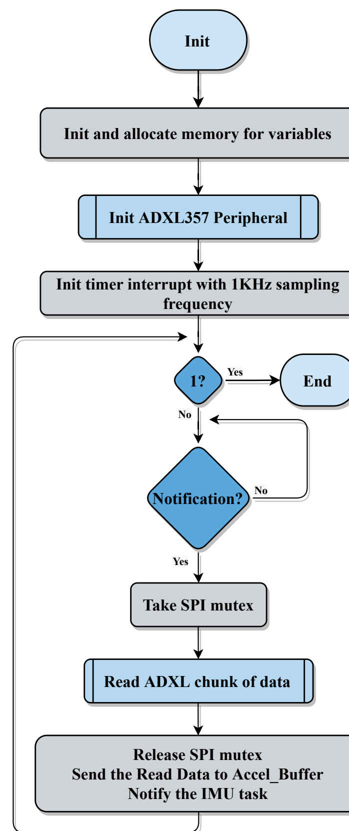


Figure 35: vAccelerometer task

vIMU

The task flowchart for data acquisition from the MPU-6881 sensor is shown in Figure 36. Similar to the accelerometer task, this task is responsible for setting up and using the MPU-6881 sensor to acquire data. After initialising memory for variables and starting the MPU-6881 peripheral, this task waits for a notification from the accelerometer task. This notification is used as a synchronization mechanism and allows the collision detection task to be organized. The next step is to read data from IMU, send that data to the IMU_Buffer and notify the collision detection task. The SPI read transaction is accompanied by a mutex that protects other tasks from accessing this resource. Usually, this mutex is not needed because the use of the notification mechanism prevents the accelerometer and IMU task to read at the same time. But, when detected a crash the situation is different. The collision detection task should notify the SDCard task for writing in SD Card and to get time from RTC, which makes a race condition to the SPI resource. To prevent that, a mutex is used before the read transaction and released after it.

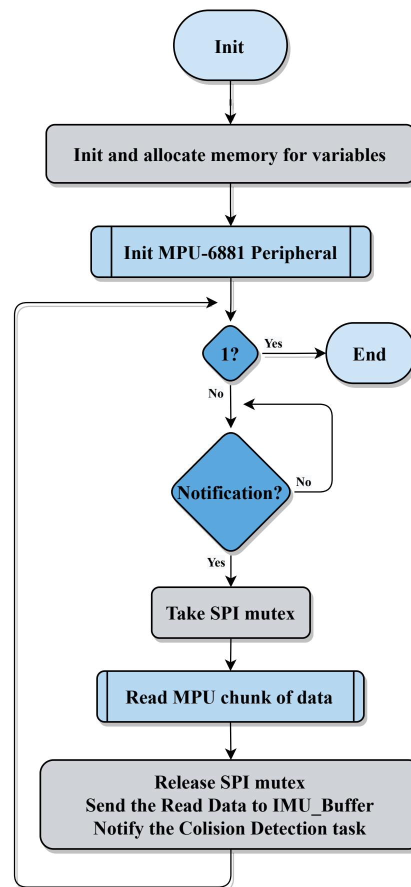


Figure 36: vIMU task

vCollisionDetection

The collision detection task, shown in Figure 37, is responsible for continuously reading data from the acquisition buffers, applying a Kalman filter, and performing a collision detection algorithm. It writes the resulting data to the SD_Buffer. The task begins by initializing the necessary memory and variables and initializing the Kalman filter module. It then enters a loop and waits for a notification from the IMU task, indicating that data is available in the IMU_buffer and Accel_buffer. Upon receiving this notification, the task reads the data and starts the Kalman prediction phase. It then performs the Kalman update phase using the newly acquired data and executes a collision detection algorithm to determine if a crash has occurred. In a colision, the data before and after the collision are also important, so in this project was decided to reserve half of the buffer to the data before colision and the other half for after the collision. In this case, the vSDCard task is notified only if a collision occurs and the SD_Buffer is full. In a normal operation, half of the buffer retains the old data by removing the oldest element from the buffer each time the half buffer is full.

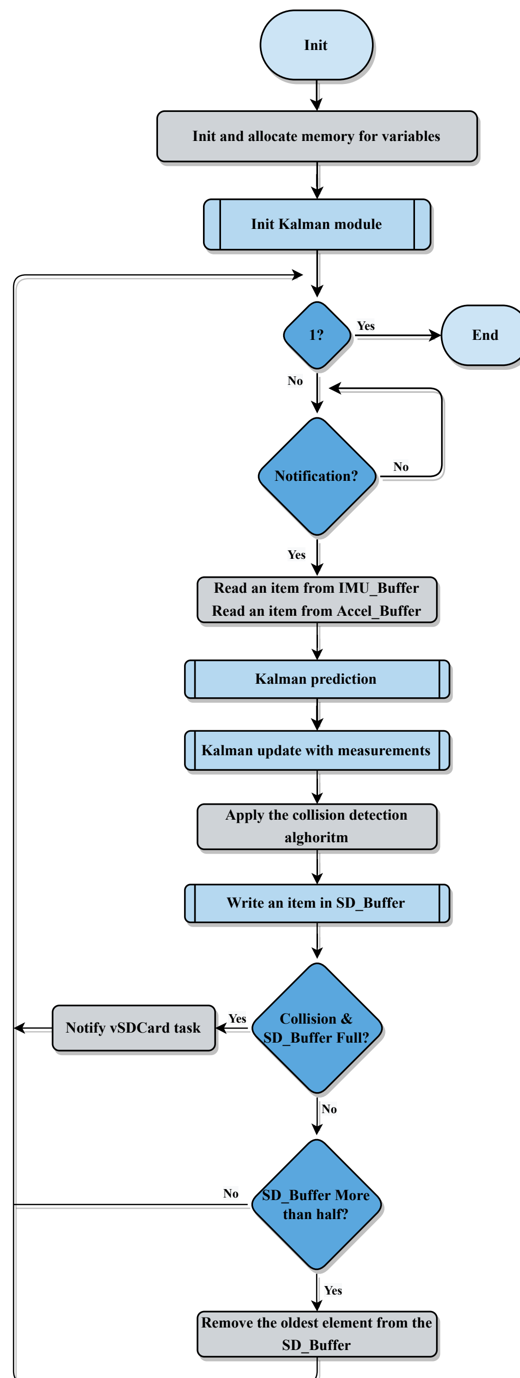


Figure 37: vCollisionDetection task

vSDCard

The SD Card task, present in figure 38 is responsible for saving data to an SD card and activating the mesh in response to a car crash. First, the task initialize variables and allocate the necessary memory in order to proceed to the micro SD card peripheral inicialization, that includes mounting the Fat filesystem and adding the microSD device to the SPI bus. The task then waits for a notification from the collision

detection task before proceeding. Once notified, it reads data from the SD_Buffer and writes it to the SD card. The thread then checks the number of items in the FIFO buffer, entering a loop until all data has been stored. After all the data has been written to the SD card, the task sends a signal to activate a GPIO and closes the microSD card, unmounting it and removing it from the SPI bus.

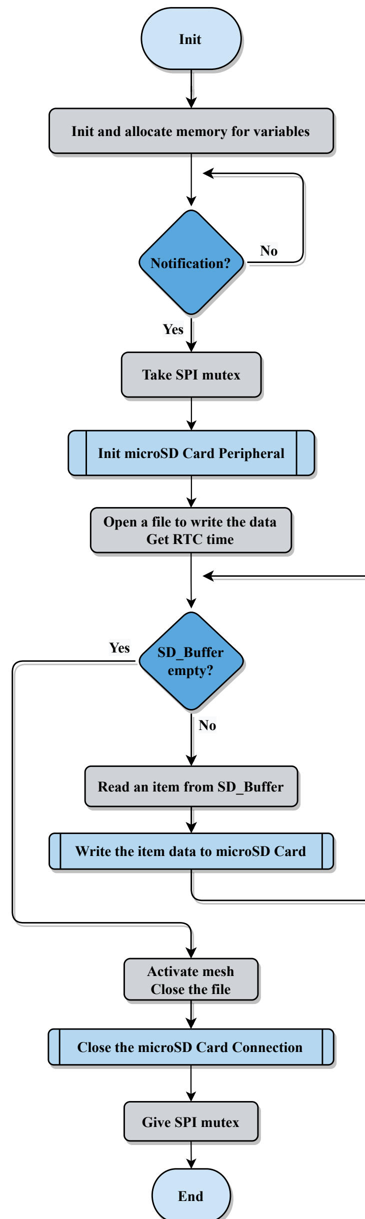


Figure 38: vSDCard task

vMain

The main task is the default task in the system and is used to create other FreeRTOS tasks and initialize peripherals. As shown in figure 32, the main task starts by initializing the GPIOs and setting default values

for them. It then initializes the SPI peripheral using the ESP32 SPI driver and initializes the RTC. Finally, the main task creates the other threads and completes their execution.

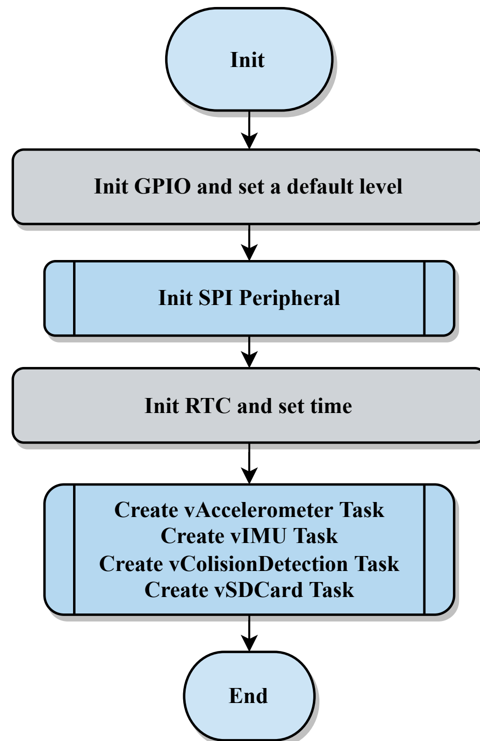


Figure 39: vMain

Chapter 5

Implementation

In this chapter, the implementation details of the proposed system are presented. The implementation phase involves taking the plan developed during the design phase and bringing it to realization using the necessary hardware and software components. A detailed description of the schematics implementation, and the techniques used, are provided. The challenges encountered during the implementation of the embedded system and the solutions adopted are also discussed. Finally, the results of the implemented system are presented.

5.1 Hardware Schematic

This section describes the hardware components and their connections in the implemented system. The hardware follows a hierarchical architecture, with the main sheet containing several blocks representing individual sheets with their respective circuits. All the schematics covered in the design phase were implemented using Altium software.

5.1.1 Main sheet

The primary schematic, illustrated in Figure 40, comprises the key elements utilized on the PCB. The micro-USB sheet supplies 5V to the board and converts the USB protocol to UART for communication with the CPU sheet. The power sheet lowers the voltage to 3.3V to power all ICs. Additionally, the micro-USB sheet enables the CPU to be programmed utilizing the reset and enable signals which initiate the programming mode and allows for software to be loaded via UART.

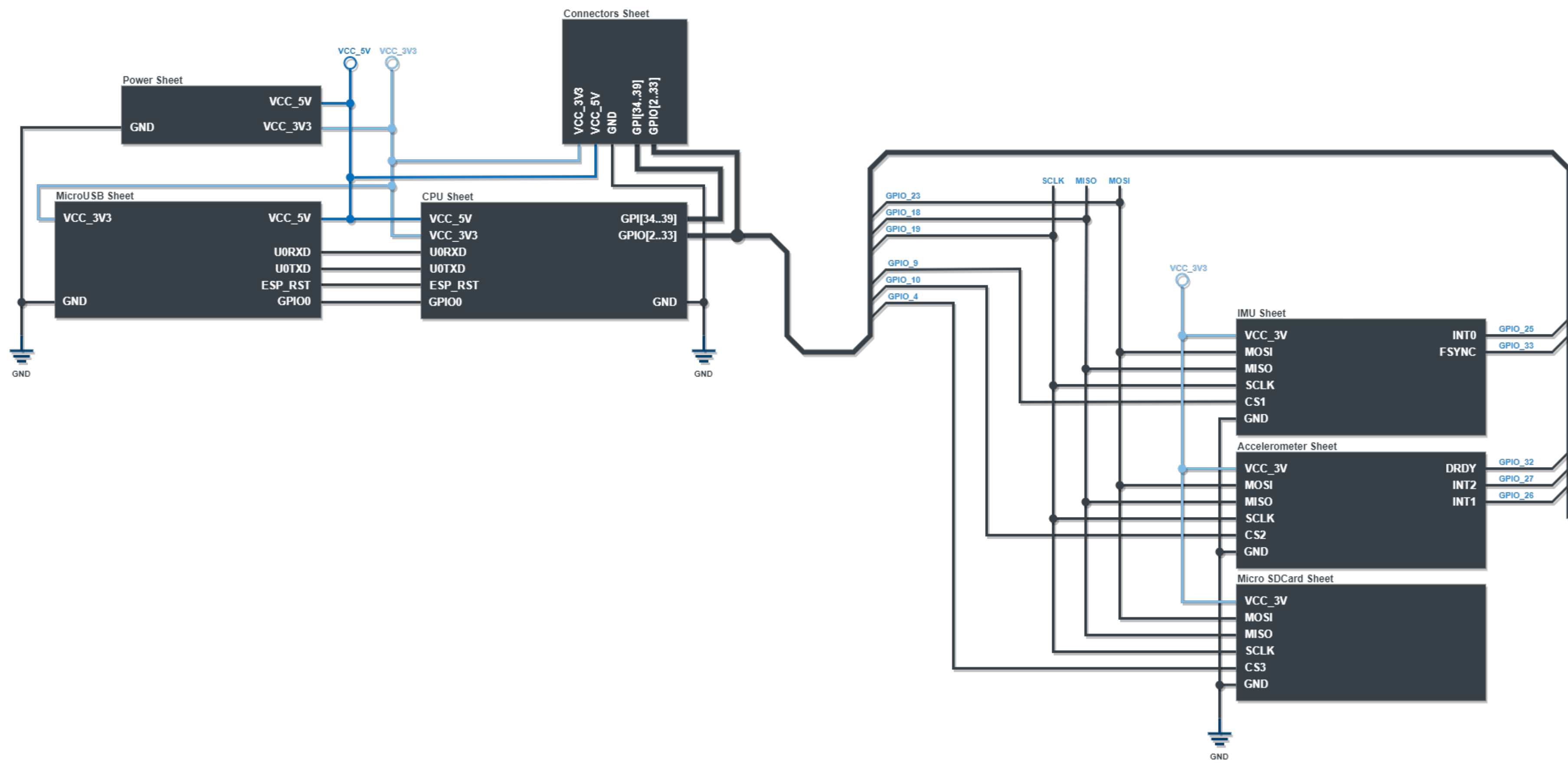


Figure 40: Main schematic

On the other side of the main sheet, three devices are connected to the microprocessor via SPI. The MISO, MOSI, and SCLK signals are all connected, and each SPI device has its chip-select signal connected to a different GPIO. This allows for communication and control between the devices and the microprocessor. Moreover, the connector sheet is also linked to the CPU sheet, allowing external connections to the system. As mentioned, the block diagrams represented are connected to individual sheets containing several circuits. This main sheet acts as an abstraction sheet and provides an overview of the system's architecture, making it easy to understand the organization of the hardware components. Figure 41 also helps to understand how all systems elements work together to accomplish the system's purpose.

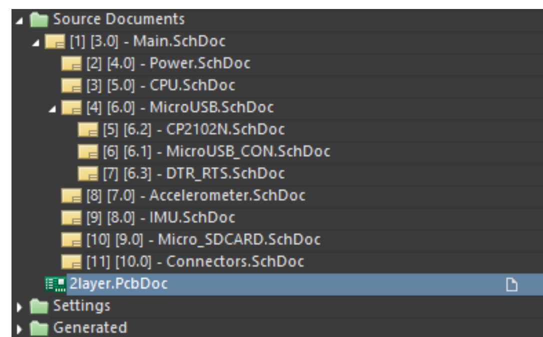


Figure 41: Altium Project Organization

5.1.2 Micro USB

The micro USB sheet, as shown in figure 42, serves as another abstraction sheet, providing a visual representation of several signals transmitted between the micro USB connector and the CPU. The schematic of the micro USB connector is included in the micro USB sheet and is utilized to supply the system with a 5V power source. The data received by the micro USB connector is then passed on to the UART-Bridge sheet for translation into the UART protocol. The resulting UART signals, TX and RX, are then directed to the ESP32 CPU. This connection allows for communication and data transfer between the system and external devices connected via the micro USB connector. The RTS and DTR signals from the UART bridge sheet are connected to a logic sheet, which can be used to initiate the CPU's programming mode when needed. This allows for the system to be reprogrammed or updated as necessary. The micro USB sheet provides a comprehensive view of the interactions between the micro USB connector, UART-Bridge, and the ESP32 CPU, making it easy to understand the communication flow and troubleshoot any issues that may arise.

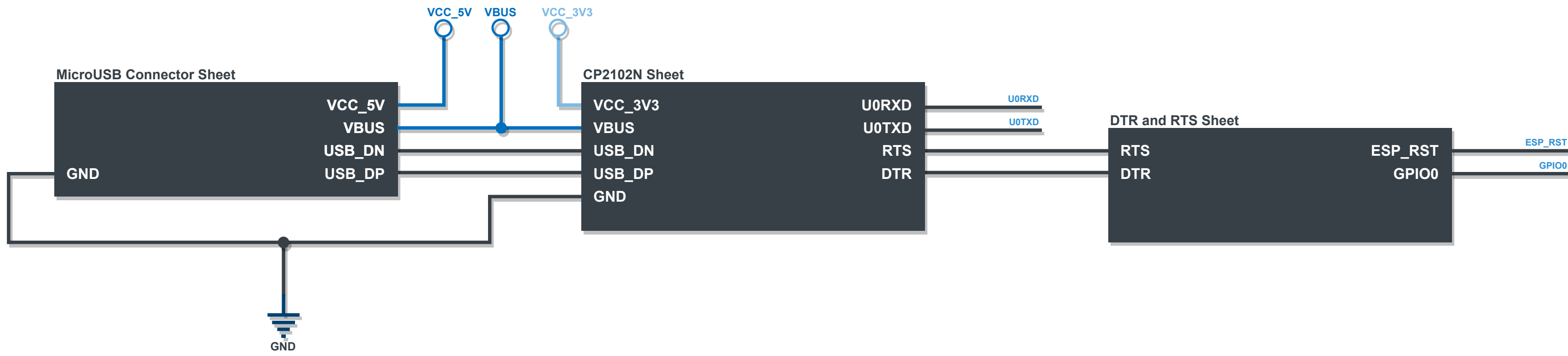


Figure 42: MicroUSB main schematic

5.2 Hardware Layout

The development of a PCB layout is a crucial step in the design process as it can affect the final product performance if not executed properly. Before starting the layout phase, it is relevant to ensure that the footprints of all components is align with the recommendations outlined in their datasheets. This verification may also involve creating custom footprints for parts that are not available. Once all the footprints have been created and imported, the layout process can commence. In this case, the board layout was designed to use two layers, as it is a cost-effective option, and the complexity of the board does not necessitate the use of a four-layer board. The layer stack used is presented in figure 43, which illustrates the two layers utilized in the design.

#	Name	Material	Type	Weight	Thickness
	Top Overlay		Overlay		
	Top Solder	Solder Resist	Solder Mask		0.4mil
1	TOP	CF-004	Signal	1oz	1.378mil
	Dielectric 1	Core-039	Core		59.055mil
2	BOTTOM	CF-004	Signal	1oz	1.378mil
	Bottom Solder	Solder Resist	Solder Mask		0.4mil
	Bottom Overlay		Overlay		

Figure 43: Layer Stack

5.2.1 Layout Process

The layout process of a PCB starts with the power part of the circuit, beginning with the placement of the micro USB connector. This connector is a key component, and a correct positioning is relevant to ensure proper operation and accessibility. Following the schematics, all parts are placed in sequential order, as shown in figure 44. The micro USB connector has a different ground plane that connects to the chassis.

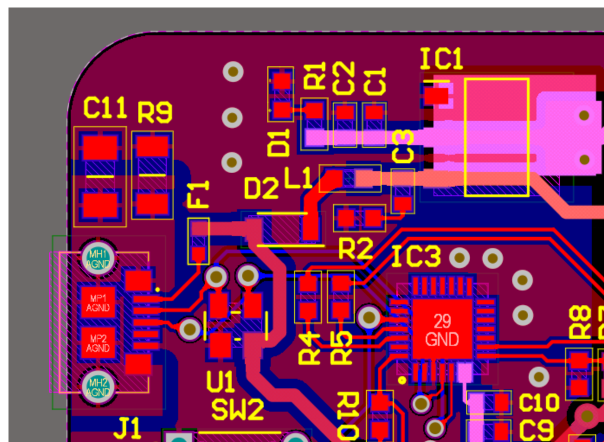


Figure 44: PCB layout LDO

After the fuse, Schottky diode, and ferrite, the power line goes to the LDO. These components are placed very close together in a sequential order to minimize trace lengths and improve performance. The TVS diodes are placed in the different USB signal lines and the LDO capacitors (C1 and C3) are placed close as possible to help the LDO to minimize trace lengths and reduce noise. A relief rectangle is used in the LDO as recommended in the datasheet for proper operation and prevent any issues during manufacturing. The UART bridge is placed close to where the data signals from the USB connector are the inputs. This placement ensures that the data signals are properly routed and that the board is reliable and performs as intended.

The layout process involves careful consideration of all the components and their placement, following the schematics. For example, the ESP32 microcontroller and its associated bypass capacitors(C5 and C6) are placed as close as possible to the ESP32. This step is relevant to ensure a stable power supply and reduce noise. Also, the positioning of the ESP32 needs some attention, since a bad layout may compromise the use of its antenna. Usually, the ESP32 is positioned in one edge of the board to avoid interference of signals from the board with the antenna. However, the antenna can be used with the ESP32 placed in a central position, as long as any existing signals or polygons are removed from under the antenna. At this prototype stage, the ESP32's antenna is not used, but this PCB is prepared for its use in the event of a future need. Figure 45 shows the ESP32 layout and the connections to the connectors. This connectors are labeled with the according GPIO's name, except for debug pins that are used to facilate the connections with an external debugger.

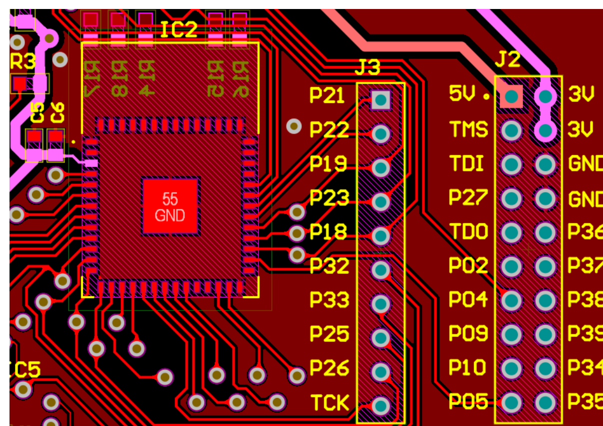
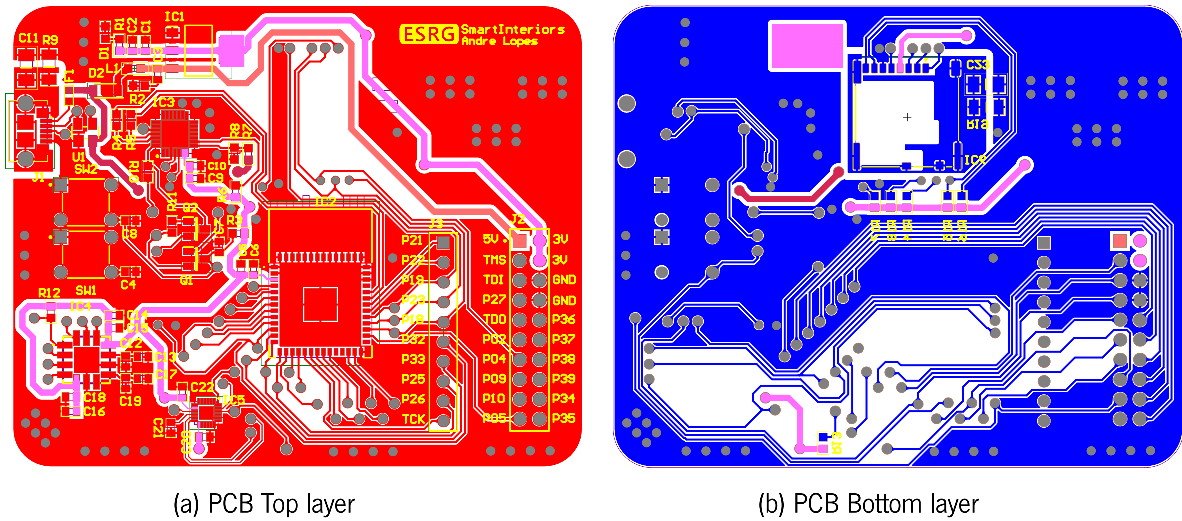


Figure 45: PCB layout ESP32

All other layout circuits are placed follwing the same rules with all the components like capacitors placed properly in order to the components take the desired effect. After all the connections are made and the DRC tests validate the board layout, the board is functionality ready to fabrique. Figure 46 present the two final PCB layout layers.

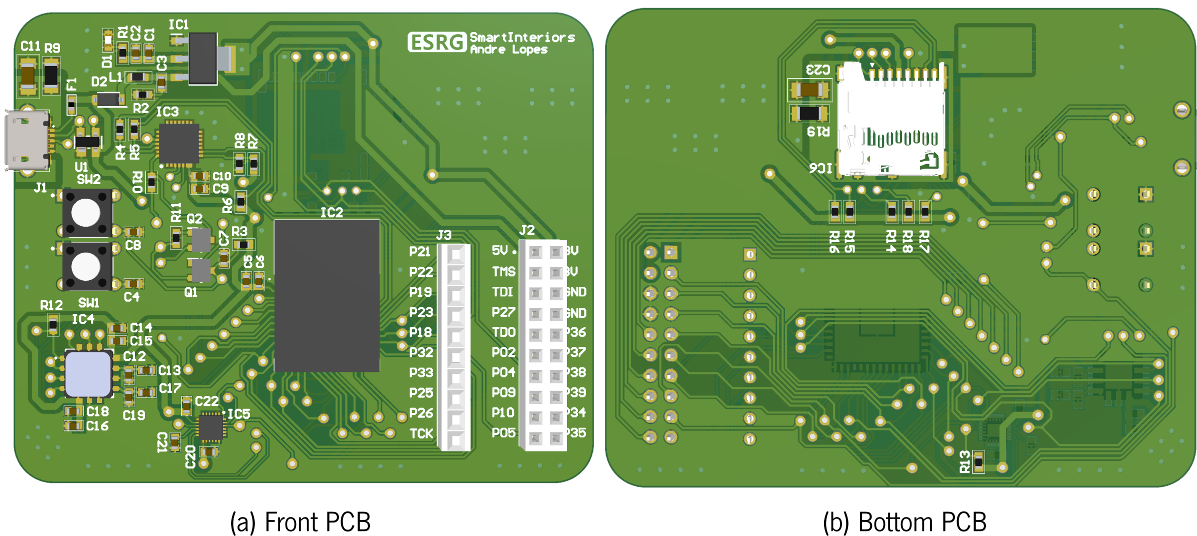


(a) PCB Top layer

(b) PCB Bottom layer

Figure 46: PCB layout layers

Figure 47 shows a 3D image of the two layers of the PCB.



(a) Front PCB

(b) Bottom PCB

Figure 47: 3D PCB

5.3 Software

The hardware part describe, in detail, the whole PCB implementation in terms of its schematics and layout. The software design presented a diagram with all the main tasks to be implemented, but for these tasks to be implemented in practice, the board's firmware will have to be prepared, and the drivers that handle all the sensor read/write transactions will have to be developed. In this section, all the software topics related to the preparation and implementation will be described. This description will be presented in a sequential order to better explain the whole workflow.

5.3.1 Drivers

ESP-IDF is the official development framework for the ESP32 and is the first step to start the software development. After the installation, a new default project is created specific to the processor in question. Then, a communication with the hardware is necessary to be established in order to encounter a few errors either from the hardware itself, related to the soldering of the components or from the firmware itself and its configurations that may not be adequate. After the communication with the board is established, the conditions are met to develop the drivers related to the chosen components. Sometimes, manufacturers have drivers already implemented that can be added and modified. In this case, neither the accelerometer ADXL357 nor the MPU-6881 have drivers already implemented, so the option was to create these drivers from scratch.

SPI Driver

The driver that handles the API provided by the ESP-IDF includes, as main functions, inicialization of the SPI driver, the establishment of read and write transactions and the option to add/remove devices from the bus. In listing 15 present in appendix A.1, it's possible to see the header file of this driver, which provides all the necessary information for its handling. This driver contains the pins to which the used SPI signals are connected and also the clock speed defined in the transactions. This speed is determined by considering the specifications of the severals connected devices and pratical tested in order to find a balance value for clock speed.

Both the receive function and the transmit function are quite similar since the arguments passed are the same. What makes these functions different is the allocation of buffers used by the API. This difference is presented in listing 16 of appendix A.1, where each transaction uses at least two buffers that can be connected depending on the desired transaction. In this listing is also possible to see the init function where the pin number of the SPI signals are used and a DMA (Direct Memory Access) indication is used to perform the SPI transactions via DMA.

ADXL Driver

With the latest driver established, it is then possible to create transactions and communicate with the various SPI sensors/devices. The creation of the SPI driver begins the initialisation process, where settings, such as its operating range, the filters to be used and the start of the acquisition, is necessary before performing the data acquisition. Listing 1 shows the three functions responsible by either SPI communication and the sensor inicialization. The first two functions serve to write and read sensor registers. In these functions, the particularity of the address and the bits indicating the read or write presented in the design phase are taken into account. Once the address register has been set, the corresponding function of the SPI driver is called. The other function is the initialisation function, that as mentioned, is responsible in a first phase for adding the device to the SPI bus, configuring the sensor with the range allowed and activating the sensor in the acquisition mode. The parameters of the filters used are also configured.

```
bool adxl_spi_read(uint8_t reg_addr, uint8_t length, uint8_t *data) {
    reg_addr = (reg_addr << 0x01) | 0x01;
    int retVal = 0;
    spi_receive_transaction(reg_addr, length, data, adxl_handle);
    return retVal != -1;
}
bool adxl_spi_write(uint8_t reg_addr, uint8_t length, uint8_t *data) {
    int retVal = 0;
    reg_addr = (reg_addr << 0x01);
    spi_send_transaction(reg_addr, length, data, adxl_handle);
    return retVal != -1;
}
void adxl_init(void) {
    // Configuration
    uint8_t command;
    uint8_t *data;
    data = &command;

    spi_add_device(10, &dev_config, &adxl_handle, host);  // Add ADXL357 accelerometer to SPI Bus
    /*-----------------------------------------------------------------------------------------*/
    /*  Set ADXL range to +-40 g   */
    /*-----------------------------------------------------------------------------------------*/
    command = 0b00000011;  //+-40 g
    adxl_spi_write(ADXL_RANGE, 1, data);
    vTaskDelay(10 / portTICK_PERIOD_MS);
    /*-----------------------------------------------------------------------------------------*/
    /*  DRDY_OFF output to 0, Temperature processing off and Measurement mode   */
    /*-----------------------------------------------------------------------------------------*/
    command = 0b00000000;
    adxl_spi_write(ADXL_POWER_CTL, 1, data);
    vTaskDelay(10 / portTICK_PERIOD_MS);

    command = 0b00010000;  // 10hz High Pass; 1Khz Low Pass
    adxl_spi_write(ADXL_FILTER, 1, data);
    vTaskDelay(10 / portTICK_PERIOD_MS);
}
```

Listing 1: ADXL Driver main functions

Once the sensor is initialized, the read accelerometer function, presented in listing 2, is implemented in order to obtain the correct acelerometer values. First, a sample containing the acelerometer values from all the three axis are obtaining by reading nine bytes from the first register since all the bytes are sequential.

Figure 5 presents the ADXL357 aceleration registers and their organization. As is possible to see, the nine registers are sequential and the third byte of each axis has only four important bits. With this organization in mind, read function concatenates all this values in variables. Also, the read values are in a two's complement format. The conversion from raw data to G's unit is realized by using the ADXL sensitivity for 40G range.

0x08	XDATA3	XDATA, Bits[19:12]		0x00	R
0x09	XDATA2	XDATA, Bits[11:4]		0x00	R
0x0A	XDATA1	XDATA, Bits[3:0]	Reserved	0x00	R
0x0B	YDATA3	YDATA, Bits[19:12]		0x00	R
0x0C	YDATA2	YDATA, Bits[11:4]		0x00	R
0x0D	YDATA1	YDATA, Bits[3:0]	Reserved	0x00	R
0x0E	ZDATA3	ZDATA, Bits[19:12]		0x00	R
0x0F	ZDATA2	ZDATA, Bits[11:4]		0x00	R
0x10	ZDATA1	ZDATA, Bits[3:0]	Reserved	0x00	R

Figure 48: ADXL357 acceleration registers organization

```
1  void adxl_read_accel(adxl_accel_t *p_accel, uint8_t *data) {
2      adxl_spi_read(ADXL_XDATA3, 9, data);
3      buffer3 = data;  // buffer3 points to the begining of DMA buffer
4      for (uint8_t i = 0; i < 9; i++, buffer3++) {
5          RX_buffer[i] = *buffer3;
6      }
7      ACCEL_Z = (RX_buffer[6] << 12) | (RX_buffer[7] << 4) | (RX_buffer[8] >> 4);
8      ACCEL_Y = (RX_buffer[3] << 12) | (RX_buffer[4] << 4) | (RX_buffer[5] >> 4);
9      ACCEL_X = (RX_buffer[0] << 12) | (RX_buffer[1] << 4) | (RX_buffer[2] >> 4);
10
11     if (ACCEL_X & 0x80000)
12         ACCEL_X = (ACCEL_X & 0x7FFFF) - 0x80000;
13     if (ACCEL_Y & 0x80000)
14         ACCEL_Y = (ACCEL_Y & 0x7FFFF) - 0x80000;
15     if (ACCEL_Z & 0x80000)
16         ACCEL_Z = (ACCEL_Z & 0x7FFFF) - 0x80000;
17
18     p_accel->ACCEL_X = (float)ACCEL_X / 51200;
19     p_accel->ACCEL_Y = (float)ACCEL_Y / 51200;
20     p_accel->ACCEL_Z = (float)ACCEL_Z / 51200;
21 }
```

Listing 2: ADXL Driver read function

MPU Driver

The driver responsible to communicate with the IMU device presents similar functions to those described in the previous driver. Listing 3 shows the code used to read/write registers, and these again use the SPI driver. As mentioned in the design phase, the first byte of the SPI temporal diagram presents the address number and the write/read bit. In this case, only the read function has to modify the desired address by setting the most significant bit. After the read/write transaction functions are implemented, the sensor can start sampling. After the device is added to the bus, the next step is configure the peripheral registers in order to change the accelerometer range, activate filters and start the sensor acquisition.

```
1 void mpu_spi_read(uint8_t reg_addr, const uint8_t length, uint8_t *data) {
2     reg_addr |= 0x80;
3     spi_receive_transaction(reg_addr, length, data, mpu_handle);
4 }
5 void mpu_spi_write(uint8_t reg_addr, const uint8_t length, uint8_t *data) {
6     spi_send_transaction(reg_addr, length, data, mpu_handle);
7 }
```

```
8  void mpu_init() {
9      // Configuration
10     uint8_t command;
11     uint8_t *data;
12     data = &command;
13 
14     spi_add_device(9, &dev_config, &mpu_handle, host);  // Add MPU acelerometer to SPI Bus
15     /*-----------------------------------------------------------------------------------*/
16     /* Send Power Management Command: Reset registers and 20Mhz clock select */
17     /*-----------------------------------------------------------------------------------*/
18     command = 0b10001000;  // 20Mhz CLKSEL and Hard Reset all registers
19     mpu_spi_write(MPU_PWR_MGMT_1, 1, data);
20     vTaskDelay(10 / portTICK_PERIOD_MS);
21     /*-----------------------------------------------------------------------------------*/
22     /* Send Accel configuration Command: Set Accel full scale to +/-16g  */
23     /*-----------------------------------------------------------------------------------*/
24     command = 0b00000000;
25     mpu_spi_write(MPU_CONFIG, 1, data);
26     vTaskDelay(10 / portTICK_PERIOD_MS);
27 
28     command = 0b00000000;
29     mpu_spi_write(MPU_ACCEL_CONFIG, 1, data);
30     vTaskDelay(10 / portTICK_PERIOD_MS);
31 
32     command = 0b00001000;
33     mpu_spi_write(MPU_ACCEL_CONFIG2, 1, data);
34     vTaskDelay(10 / portTICK_PERIOD_MS);
35 }
```

Listing 3: MPU Driver main functions

The read MPU function, shown in listing 4, is also similar to the ADXL357 driver because it follows the same principles. The necessary bytes are read from the sensor to complete a sample with all the data from the accelerometer axis. This that is then concatenated in variables divided by the three axis. In this case the data is not in two's complement format, so is just necessary to convert this data to G's.

```
1  void mpu_read_accel(mpu_accel_t *p_accel, uint32_t length, uint8_t *data) {
2      mpu_spi_read(MPU_ACCEL_XOUT_H, 6, data);
3      buffer3 = data;
4      for (uint8_t i = 0; i < 6; i++, buffer3++)
5          RX_buffer[i] = *buffer3;
6      ACCEL_Z = (RX_buffer[4] << 8) | RX_buffer[5];
7      ACCEL_Y = (RX_buffer[2] << 8) | RX_buffer[3];
8      ACCEL_X = (RX_buffer[0] << 8) | RX_buffer[1];
9      p_accel[0].ACCEL_X = (float)ACCEL_X / 16384;
10     p_accel[0].ACCEL_Y = (float)ACCEL_Y / 16384;
11     p_accel[0].ACCEL_Z = (float)ACCEL_Z / 16384;
12 }
```

Listing 4: MPU Driver read function

SDCard Driver

Listing 17 in appendix A.2 presents the driver responsible for handling the microSD Card. This driver starts by mounting a FAT filesystem on the micro SDCard. This Fat filesystem is implemented in the ESP-IDF. This driver implemented by the framework, allows C library functions like fopen or fprintf to work with the filesystem devices. In this listing, the "esp_vfs_fat_sdspi_mount" function is used to initializes an SPI Master device based on the SPI Master driver with configuration in "slot_config", and attach it to an initialized SPI bus. This configuration takes in account the pin used for the chip select signal, which is used to connect the CPU and the microSD card in the PCB implementation. After mounting the filesystem with a given path, the communication with this device is ready.

The mount function is then used to initialize the micro SD Card. The base path used and all the code for the init and close function is shown in listing 18 of appendix A.2.

Matrix module

The Kalman filter utilizes matrix operations to efficiently handle all necessary calculations and, whenever possible, enhance the performance of these calculations. All the functions that have been implemented in the matrix module's implementation, are presented in listing 5 that shows the header file of the module.

```
1  typedef float matrix_data_t;
2  typedef struct matrix_s {
3      uint8_t rows;
4      uint8_t cols;
5      matrix_data_t *data;
6  } matrix_t;
7
8  void matrix_get_column_copy(const matrix_t *const mat, const uint8_t column, matrix_data_t *const row_data);
9  void matrix_init( int M, int N, double **mat);
10 void matrix_clean( int M, int N, double **mat);
11 void matrix_mul(matrix_t *A, matrix_t *B, matrix_t *C);
12 void matrix_mul_transb(const matrix_t *const a, const matrix_t *const b, const matrix_t *c);
13 void matrix_add(matrix_t *A, matrix_t *B, matrix_t *C);
14 void matrix_sub(const matrix_t *const a, matrix_t *const b, const matrix_t *c);
15 void matrix_invert_lower(const matrix_t *const lower, matrix_t *inverse);
16
17 void set_matrix_template(matrix_t *A, float *template, const uint8_t rows, const uint8_t columns);
18 void set_matrix(matrix_t *A, const uint8_t rows, const uint8_t columns);
19 void set_matrix_position(matrix_t *mat, const uint_fast8_t row, const uint_fast8_t column, const matrix_data_t value);
20 int cholesky_decompose_lower(const matrix_t *const mat);
```

Listing 5: Matrix Module main functions

Kalman module

The implementation of the matrix module is closely connected to the overall Kalman filter development. Therefore, the two modules were developed in parallel. The initialization begins by utilizing predefined templates and placing them in a buffer for proper interpretation by the matrix module. Additionally, the matrix module includes a function that fills all other matrices with zeros. The init function ultimately generates the parameters for the specific high-pass filter that will be applied to the IMU data.

```
1  void kalman_init(void) {
2      set_matrix_template(&A, &template_A, 9, 9);
3      set_matrix_template(&Previous_P, &template_Previous_P, 9, 9);
4      set_matrix_template(&Q, &template_Q, 9, 9);
5      set_matrix_template(&C, &template_C, 3, 9);
6      set_matrix_template(&R, &template_R, 3, 3);
7      set_matrix_template(&I, &template_I, 9, 9);
8
9      set_matrix(&baux, 9, 1);
10     set_matrix(&Previous_X, 9, 1);
11     set_matrix(&X_p, 9, 1);
12     set_matrix(&aux_pre, 9, 9);
13     set_matrix(&aux_pre1, 9, 9);
14     set_matrix(&Pre, 9, 9);
15     set_matrix(&Y, 3, 1);
16     set_matrix(&S, 3, 3);
17     set_matrix(&S_inv, 3, 3);
18     set_matrix(&Y_meas, 9, 1);
19     set_matrix(&K, 9, 3);
20     set_matrix(&aux_K, 9, 3);
21     set_matrix(&aux_K2, 3, 3);
22     set_matrix(&aux_K1, 3, 9);
23     set_matrix(&aux_X1, 3, 1);
24     set_matrix(&aux_X2, 3, 1);
25     set_matrix(&aux_X3, 9, 1);
26     set_matrix(&X, 9, 1);
27     set_matrix(&P, 9, 9);
28     set_matrix(&aux_P1, 9, 9);
29     set_matrix(&aux_P2, 9, 9);
30
31     filtered = (kalman_accel_t *) malloc(sizeof(kalman_accel_t));
32
33     filter_x = create_bw_high_pass_filter(8,1000,10);
34     filter_y = create_bw_high_pass_filter(8,1000,10);
35     filter_z = create_bw_high_pass_filter(8,1000,10);
36 }
```

Listing 6: Kalman module init function

Listing 7 contains the predict function used to estimate the state of the system and estimate the covariance. The functions implemented in the matrix module that handle different operations between the matrices are now used to perform the two predict kalman equations presented earlier.

```
1  void kalman_predition(void) {
2      //------------------------------------------------------------------------------
3      // The Predicted State --> X_p = A * Previous_X
4      //------------------------------------------------------------------------------
5      matrix_mul(&A, &Previous_X, &X_p);
6
7      //------------------------------------------------------------------------------
8      // The Predicted Process Covariance Matrix --> Pre = A*Previous_P*transpose(A)+Q
9      //------------------------------------------------------------------------------
10     matrix_mul(&A, &Previous_P, &aux_pre);
```

```
11     matrix_mul_transb(&aux_pre, &A, &aux_pre1);
12     matrix_add(&aux_pre1, &Q, &Pre);
13 }
```

Listing 7: Kalman module predict function

Before the filter update phase, the function allows applying the high-pass filter on the IMU data and preparing the combination of the acceleration values of the two sensors. This function is present in listing 8, where one can observe precisely this preparation that is accompanied by the variance values previously calculated.

```
1  void kalman_update(kalman_accel_t *setup, kalman_accel_t *filtered) {
2     //----------------------------------------------------------------------------------------
3     // The New Observation --> Y = C*Y_meas
4     //----------------------------------------------------------------------------------------
5     set_matrix_position(&Y_meas, 2, 0, setup->ACCEL_X);
6     set_matrix_position(&Y_meas, 5, 0, setup->ACCEL_Y);
7     set_matrix_position(&Y_meas, 8, 0, setup->ACCEL_Z);
8     matrix_mul(&C, &Y_meas, &Y);  // Y = C*Y_meas
9     //----------------------------------------------------------------------------------------
10    // Calculating the Kalman Gain --> K = Pre*transpose(C) / (C*Pre*transpose(C) + R)
11    //----------------------------------------------------------------------------------------
12    matrix_mul_transb(&Pre, &C, &aux_K);        // aux_K = Pre*transpose(C)
13    matrix_mul(&C, &Pre, &aux_K1);              // aux_K1 = C*Pre
14    matrix_mul_transb(&aux_K1, &C, &aux_K2);    // aux_K2 = aux_K1*transpose(C)
15    matrix_add(&aux_K2, &R, &S);                // S = C*Pre*transpose(C) + R
16    cholesky_decompose_lower(&S);               // Decompose Matrix using cholesky method
17    matrix_invert_lower(&S, &S_inv);            // S_inv = S^-1
18    matrix_mul(&aux_K, &S_inv, &K);             // aux_K1 = C*Pre
19
20    //----------------------------------------------------------------------------------------
21    // Calculating the current State --> X = X_p + K * ( Y - C * X_p)
22    //----------------------------------------------------------------------------------------
23    matrix_mul(&C, &X_p, &aux_X1);              // aux_X1 = C*X_p
24    matrix_sub(&Y, &aux_X1, &aux_X2);           // aux_X2 = Y - aux_X1
25    matrix_mul(&K, &aux_X2, &aux_X3);           // aux_X3 = K * aux_X2
26    matrix_add(&aux_X3, &X_p, &X);              // X = X_p + aux_X3
27
28
29    //----------------------------------------------------------------------------------------
30    // Updating the process Covariance Matrix --> P=(I-K*C)*Pre
31    //----------------------------------------------------------------------------------------
32    matrix_mul(&K, &C, &aux_P1);                // aux_P1 = K*C
33    matrix_sub(&I, &aux_P1, &aux_P2);           // aux_P2 = I - aux_P1
34    matrix_mul(&aux_P2, &Pre, &P);              // P = aux_P2 * Pre
35
36    //----------------------------------------------------------------------------------------
37    // Current becomes previous
38    //----------------------------------------------------------------------------------------
39    Previous_X = X;
40    Previous_P = P;
41
42    filtered->ACCEL_X = Previous_X.data[2];
```

```
43     filtered->ACCEL_Y =  Previous_X.data[5];
44     filtered->ACCEL_Z =  Previous_X.data[8];
45 }
```

Listing 8: Kalman module setup function

After that, the Kalman filter update function described in listing 9 will go into action, starting by executing the measurement equation where the acceleration values will be used. After that, the Kalman gain is then calculated due to the shape of the equation forcing the use of a matrix division. For reasons of optimization, the Cholesky decomposition is used, and all necessary matrix operations are performed to obtain the Kalman gain matrix. Then, using the same procedures, the rest of the equations will use the Kalman gain to update the system state and the covariance matrix. This function ends, preparing the next iteration of the filter, where the current values of the state matrix and the covariance matrix are considered as previous values.

```
1  void kalman_setup(kalman_accel_t *mpu_data, kalman_accel_t *adxl_data, kalman_accel_t* result){
2      mpu_data->ACCEL_X = bw_high_pass(filter_x, mpu_data->ACCEL_X);
3      mpu_data->ACCEL_Y = bw_high_pass(filter_y, mpu_data->ACCEL_Y);
4      mpu_data->ACCEL_Z = bw_high_pass(filter_z, mpu_data->ACCEL_Z);
5
6      result->ACCEL_X = ((VARIANCE_ADXL_10G)*adxl_data->ACCEL_X + (VARIANCE_MPU)*mpu_data->ACCEL_X)
7      result->ACCEL_X /= (VARIANCE_MPU + VARIANCE_ADXL_10G);
8
9      result->ACCEL_Y = ((VARIANCE_ADXL_10G)*adxl_data->ACCEL_Y + (VARIANCE_MPU)*mpu_data->ACCEL_Y)
10     result->ACCEL_Y /= (VARIANCE_MPU + VARIANCE_ADXL_10G);
11
12     result->ACCEL_Z = ((VARIANCE_ADXL_10G)*adxl_data->ACCEL_Z + (VARIANCE_MPU)*mpu_data->ACCEL_Z)
13     result->ACCEL_Z /= (VARIANCE_MPU + VARIANCE_ADXL_10G);
14 }
```

Listing 9: Kalman module update function

5.3.2 Tasks

The implementation of the necessary drivers and modules allows an easier implementation of the FreeRTOS tasks. This way, the logic of the implemented code is easier to understand, which makes the whole debugging process and error identification uncomplicated. Thus, this implementation phase starts by developing the tasks responsible for all the data processing related to the sensors and then goes on to the development of the control task, where this data is filtered and combined to process the kalman filter. This control task ends with the detection algorithm implementation and deciding the next steps. Finally, the project ends by implementing the saving data from the collision task.

vMain

The main task code, presented in listing 10, is the default task that FreeRTOS launch and is responsible to prepare the other tasks and launch them. As is possible to see, first the GPIOs are set and the RTC is

prepared and configured to the correct timezone. Then, the current time for the RTC is setted and all the FreeRTOS tasks are created.

```
void app_main() {
  ESP_LOGI(Main_tag, "Hello Main Task");

  //Init GPIO and set a default level
  gpio_set_direction(5, GPIO_MODE_OUTPUT);
  gpio_set_direction(26, GPIO_MODE_OUTPUT);
  gpio_set_direction(33, GPIO_MODE_OUTPUT);
  gpio_set_direction(21, GPIO_MODE_OUTPUT);
  gpio_set_level(5, 0);
  gpio_set_level(26, 0);
  gpio_set_level(33, 0);
  gpio_set_level(21, 0);

  spi_master_init(host); //Init SPI pheripheral

  //Init RTC
  time_t now;
  char strftime_buf[64];
  struct tm timeinfo;

  // Set timezone to Lisbon Standard Time
  setenv("TZ", "WET0WEST,M3.5.0/1,M10.5.0", 1);
  tzset();

  // set current day/time
  struct timeval tv;
  tv.tv_sec =   1660141086;  //enter UTC UNIX time (get it from https://www.unixtimestamp.com )
  settimeofday(&tv, NULL);
  time(&now);
  localtime_r(&now, &timeinfo);
  strftime(strftime_buf, sizeof(strftime_buf), "%c", &timeinfo);
  ESP_LOGI(Main_tag, "The current date/time in Portugal is: %s", strftime_buf);

  //Create all the tasks
  xTaskCreate(&vAccelerometer, "Accel_Task", 2048, NULL, 5, &x_accel);
  vTaskDelay(30 / portTICK_PERIOD_MS);
  xTaskCreate(&vIMU, "IMU_Task", 5 * 1024, NULL, 5, &x_imu);
  xTaskCreate(&vColisionDetection, "Colision_Detection_Task", 2*1024, NULL, 5, &x_colision);
  xTaskCreate(&vSDCard, "SD_Card_Task", 3*1024, NULL, 5, &x_sdcard);
}
```

Listing 10: vMain Task

vAccelerometer

The task responsible for processing the ADXL accelerometer data, whose code is present in listing 11, was designed in the previous phase and starts by allocating the necessary memory for the variables to be used. The data buffer, which is 9 bytes allocated, is used to store a sensor sample. This sample takes into account the three axes and the bits needed for each axis. After the initialization phase, the synchronization mechanisms like mutex and FIFO buffer are created, and the initialization function uses

the driver previously implemented. After that, before starting the acquisition loop, a timer is created, which will be used to notify this task at a frequency of 1KHz. In the acquisition loop, after the notification of the timer, the accelerometer values are read, and the samples are placed in the buffer previously created. The read acceleration function called is protected by a mutex to guarantee that other tasks don't use the same SPI driver resource. Before ending the loop, this thread notifies the IMU task for the acquisition process to become sequential.

```
void vAccelerometer(void *pvParameters) {
  ESP_LOGI(vAccel_tag, "Hello from Accelerometer Task");

  // Init and allocate memory for variables
  spi_device_handle_t adxl_handle;
  spi_device_interface_config_t dev_config;
  uint32_t ulInterruptStatus;
  uint8_t *data = (uint8_t *)heap_caps_malloc(9, MALLOC_CAP_8BIT);
  p_accel = (adxl_accel_t *)heap_caps_malloc(1 * sizeof(adxl_accel_t), MALLOC_CAP_8BIT);
  vSemaphoreCreateBinary(xSemaphore); // Create Mutex for SPI resource
  Accel_Buffer = xQueueCreate(512, sizeof(adxl_accel_t)); // Create a FIFO Queue with 512 positions
  if (Accel_Buffer == NULL) {
    printf("Failed to create Queue\r\n");
    return;
  }

  adxl_init(); //Init ADXL357 accelerometer

  example_tg_timer_init(TIMER_GROUP_0, TIMER_0, true, 0.001); //Sampling frequency ->1000Hz

  while(1){
    xTaskNotifyWaitIndexed(0, 0x00, ULONG_MAX, &ulInterruptStatus, portMAX_DELAY);

    xSemaphoreTake(xSemaphore, portMAX_DELAY); // Take SPI mutex

    adxl_read_accel(p_accel, data); // Read ADXL chunk of data

    xSemaphoreGive(xSemaphore); // Release SPI mutex
    xQueueSendToBack(Accel_Buffer, (void *)&p_accel, portMAX_DELAY); // Send the data to Accel_Buffer
    xTaskNotify( x_imu, 0, eNoAction ); // Notify IMU task
  }
}
```

Listing 11: vAccelerometer Task

vIMU

As already mentioned, the MPU-6881 warm-up task, shown in listing 12, follows the same model as the previous task because it has the same structure. The task starts by initializing the variables to be used, where the memory allocated to a data sample is 6 bytes. This allocation size is due to the fact that the ADC bit resolution is smaller, which makes it need only 2 bytes per axis to store a data sample. Next, the task initializes the sensor through its driver and creates the FIFO buffer necessary for communication between the control task and it. Once everything is initialized, the thread enters a loop, where it waits for

the notification coming from the vAccelerometer task. After this event, it samples the data, accompanied by the respective mutex, and sends them to the buffer, notifying the control task that it has the data from the sensors ready to be processed.

```
void vIMU(void *pvParameters) {
  ESP_LOGI(vIMU_tag, "Hello from IMU Task");

  // Init and allocate memory for variables
  data = malloc(6);  // Allocate memory
  p_imu = (mpu_accel_t *)heap_caps_malloc(sizeof(mpu_accel_t), MALLOC_CAP_8BIT);
  IMU_Buffer = xQueueCreate(512, sizeof(mpu_accel_t)); // Create a Queue with 512 positions
  if (IMU_Buffer == NULL) {
    printf("Failed to create Queue\r\n");
    return;
  }

  mpu_init(); // Init MPU-6881 Peripheral

  while (1) {
    xTaskNotifyWait(0,0x00, 0,portMAX_DELAY); // Wait for vAccelerometer notification

    xSemaphoreTake(xSemaphore, portMAX_DELAY); // Take the SPI mutex

    mpu_read_accel(p_imu, 1, data); // Read MPU chunk of data

    xSemaphoreGive(xSemaphore); // Release the SPI mutex
    xQueueSendToBack(IMU_Buffer, (void *)&p_imu, portMAX_DELAY); // Send the sample to the buffer
    xTaskNotify( x_colision, 0, eNoAction ); // Notify the vCollisionDetection task
  }

}
```

Listing 12: vIMU Task

vColisionDetection

After acquiring data from the sensors, the detection task, presented in listing 13 uses this data to apply the necessary filtering and proceed to collision detection. As defined in the design phase, the thread starts by initializing the variables, creating the FIFO buffer and initializing the drivers responsible for filtering, such as Kalman. After that, the task enters the loop where it waits for the notification that indicates that the two accelerometers have already injected data in the buffer created. After the notification arrives, the data is taken from the FIFO and the execution of the first phase of the Kalman filter proceeds. Also, the setup function of the Kalman module is executed to perform the data combination in a weighted way according to the variance of the sensors. Next, it proceeds to the update phase of this filter, thus concluding the iteration of the Kalman filter. After the filtering phase comes the collision detection algorithm performed according to a threshold and considering the modulus of accelerations. Finally, some logic is applied in case of a collision, which causes the five previous second's data to be saved along with the five subsequent seconds. In case of a crash, these data are inserted in a buffer that is shared with the vSDCard task.

```
void vColisionDetection(void *pvParameters) {
  ESP_LOGI(vCollisionDetection_tag, "Hello from Colision Detection Task");

  // Init and allocate memory for variables
  adxl_accel_t *p_ADXL_accel = (adxl_accel_t *)malloc(sizeof(adxl_accel_t));
  mpu_accel_t *p_MPU_accel = (mpu_accel_t *)malloc(sizeof(mpu_accel_t));
  sd_accel_t *SD_accel = (sd_accel_t *)malloc(sizeof(sd_accel_t));
  sd_accel_t *SD_accel1 = (sd_accel_t *)malloc(sizeof(sd_accel_t));
  kalman_accel_t *filtered = (kalman_accel_t *)malloc(sizeof(kalman_accel_t));
  kalman_accel_t *setup = (kalman_accel_t *)malloc(sizeof(kalman_accel_t));
  double acceleration_x = 0, acceleration_y = 0, acceleration_z = 0;
  static uint8_t police = 0;
  static int count = 0;

  SD_Buffer = xQueueCreateStatic(QUEUE_LENGTH, ITEM_SIZE,&( ucQueueStorage[ 0 ] ), &xQueueBuffer);
  if (SD_Buffer == NULL) {
    printf("Failed to create the queue\r\n");
    return;
  }

  kalman_init(); //Init the kalman module

  while (1) {
    xTaskNotifyWait(0,0x00, 0,portMAX_DELAY);

    // Read the data from the both sensors
    xQueueReceive(Accel_Buffer, (void *)&p_ADXL_accel, portMAX_DELAY);
    xQueueReceive(IMU_Buffer, (void *)&p_MPU_accel, portMAX_DELAY);

    // Kalman filter
    kalman_predition();
    kalman_setup(p_MPU_accel, p_ADXL_accel, setup);
    kalman_update(setup, filtered);
    SD_accel->kalman = *filtered;

    xQueueSendToBack(SD_Buffer, (void *)SD_accel, portMAX_DELAY); // Write the result of kalman data

    //Apply detection alghoritm and notify vSDCard task if a collision occurs
    count++;
    police = (sqrt(abs(p_ADXL_accel->ACCEL_X)*abs(p_ADXL_accel->ACCEL_X) + abs(p_ADXL_accel->ACCEL_Y)*abs(p_ADXL_accel->ACCEL_Y) +
      abs(p_ADXL_accel->ACCEL_Z)*abs(p_ADXL_accel->ACCEL_Z)) > 5 || police);
    (count > 9999 && police) ? xTaskNotify( x_sdcard, 0, eNoAction) : ((count > 5000 && !police) ? xQueueReceive(SD_Buffer, (void *)SD_accel1,
      portMAX_DELAY), count = 5000 : NULL);
  }
}
```

Listing 13: vColisionDetection Task

vSDCard

Software development ends with the implementation of the task responsible for sending the collision data to the micro SD card. Listing 14 shows the code that was implemented. The task starts, as usual, by initializing variables that are needed during its process. Then the task enters a loop, where it waits for

a notification from the collision detection task, suspends all other threads and initializes the sd card by mounting its filesystem. After that, it opens a file in append mode, obtains the final collision time and sends the combination of all data to the micro SD Card. This task ends with closing this file and unmounting the filesystem.

```
1  void vSDCard(void *pvParameters) {
2    ESP_LOGI(vSDCard_tag, "Hello from SD Task");
3
4    // Init and allocate memory for variables
5    char *file = "/fat/Colision.txt";
6    static uint32_t count = 0;
7    sd_accel_t *SD_accel = (sd_accel_t *)heap_caps_malloc(sizeof(sd_accel_t), MALLOC_CAP_8BIT);
8    uint16_t sample_rate = 1000; // hz
9    uint16_t time = 10; // Hours * Minutes * seconds
10   uint32_t Samples = time * sample_rate;
11
12   xTaskNotifyWait(0,0x00, 0,portMAX_DELAY); // Wait for vCollisionDetection task
13
14   xSemaphoreTake(xSemaphore, portMAX_DELAY); // Take SPI mutex
15   sdcard_init(); // Init SD Card and mount the filesystem
16
17   f = fopen(file, "a+"); // Open the file
18   if (f == NULL) {
19     ESP_LOGE(vSDCard, "Failed to open file for writing");
20     return;
21   }
22
23   //Get RTC time
24   struct timeval tv_now;
25   gettimeofday(&tv_now, NULL);
26   int64_t time_us = (int64_t)tv_now.tv_sec * 1000000L + (int64_t)tv_now.tv_usec;
27
28   //Include RTC Time with the samples and save it on micro SD Card
29   for(int i=0; i<Samples-1; i++){
30     if(xQueueReceive(xQueue3, (void *)SD_accel, portMAX_DELAY) == pdTRUE){
31       custom_string(SD_accel,tv_now);
32       tv_now.tv_usec -= 1000; // 1/1000Hz = 0,001 = 1000*10^-6
33     }
34   }
35
36   mesh_on(); // activate mesh
37   fclose(f); //Close file
38   printf("File Closed\r\n");
39   sdcard_close(); // Unmount the filesystem
40   xSemaphoreGive(xSemaphore); // Give SPI mutex
41
42   while(1);
43 }
```

Listing 14: vSDCard Task

Chapter 6

Tests and Results

The prototype developed in the previous phase must undergo rigorous testing to validate its effectiveness and purpose. This chapter outlines the several tests conducted to assess the performance of the acquisition system and the collision detection algorithm.

To evaluate the efficiency of the filters and the acquisition system, a shaker test with a closed-loop system was performed. This allowed us to understand the embedded system's performance following the shaker profile. For the collision detection test, a drop test was executed by dropping the system from a certain height and observing the outcome. This test was designed to determine the accuracy of the collision detection algorithm and the correctness of the collision data.

6.1 Shaker Tests

Shaker tests are essential in evaluating the performance and durability of mechanical devices and structures. The shaker test process utilizes three crucial elements, as illustrated in Figure 49. The primary component of the process, the shaker, plays a crucial role in generating motion. It does so by producing motion in a single direction, which is usually vertical.

The device under test is placed on a plate at the top of the shaker and an accelerometer is used to monitor the motion and provide feedback to the system. The accelerometer is a critical component as it allows for real-time measurement of the vibration, allowing for precise control and adjustment of the test. The amplifier and controller work in conjunction with each other to ensure that the shaker operates in a closed-loop manner, following a pre-defined profile. This ensures that the test results are accurate and consistent, and that the device under test is subjected to the appropriate level of stress. To ensure accurate results, shaker tests are performed in all three axes, and it is necessary to secure the device under test

using coupling structures to prevent it from moving or becoming dislodged during the testing process.

(a) Shaker Controller (b) Shaker Amplifier (c) Shaker

Figure 49: Shaker Components

Figure 50 shows the 3D holder that was designed to hold the board properly to the shaker. After the board is attached to this 3D model, it is then screwed to the board on top of the shaker. This module will not only ensure that the board is coupled to the shaker, but it will also allow the board to experience the same acceleration since it is attached.

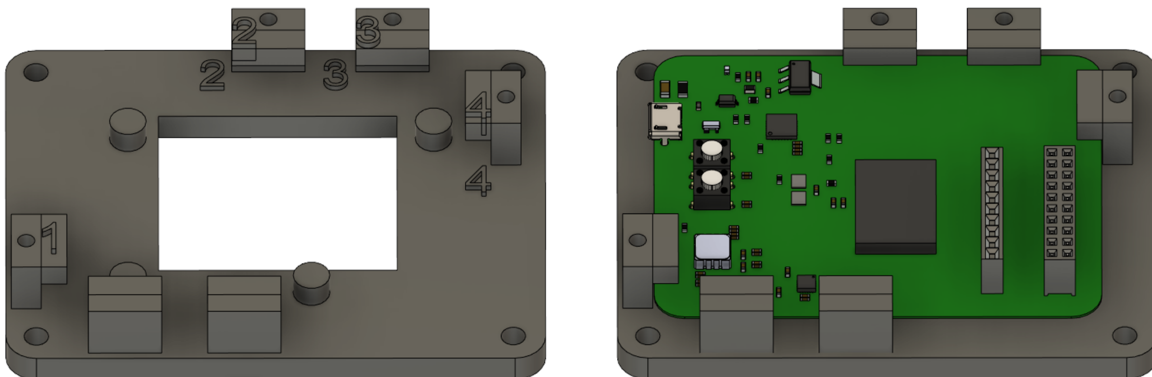
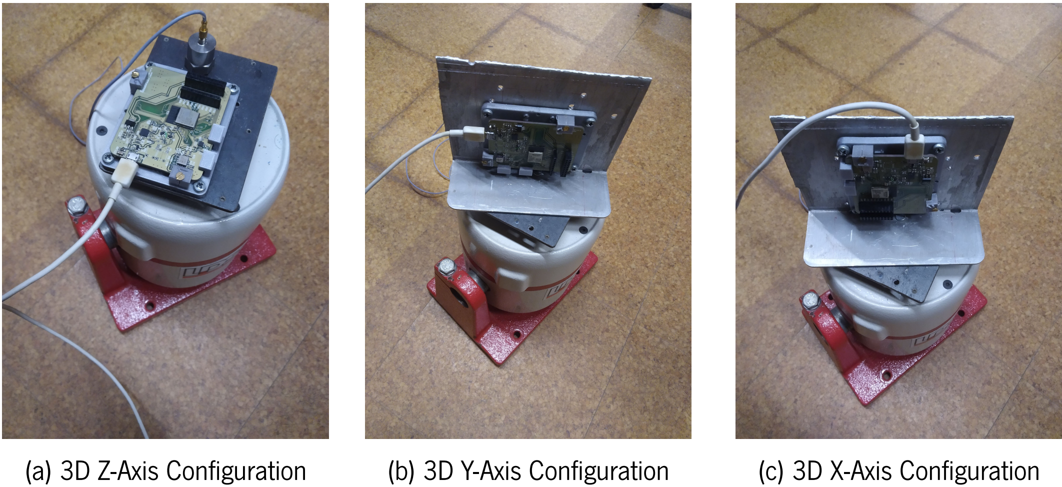


Figure 50: 3D holder model

However, there is still an obstacle to these tests that have to do with the fact that it is not possible to test all the axes of the prototype without some adaptation. The use of the holder only allows the PCB to be tested in the Z axis. To test the remaining axes, a metal plate was used screwed to the shaker. This plate allows coupling the 3D holder in the reamining two positions, as can be seen in figure 51, thus being able to test all the axes of the prototype.

In terms of software, as is normal, there is some adaptation, since the acquisition of the sensor data have to be permanently stored on the microSD card. This is due to some memory limitations on the part of the ESP32 since if there was enough memory, the ESP32 could simply save the data in its memory and at the end of the test send them to the microSD card.



(a) 3D Z-Axis Configuration (b) 3D Y-Axis Configuration (c) 3D X-Axis Configuration

Figure 51: PCB Mounting Configuration

Due to these memory limitations and since the amount of data is significant given the roughly 10 minute test time, the sampling frequency ends up being affected. Since writing to the microSD card delays the whole acquisition process. Therefore, the software used presents a sampling frequency of 1KHz, which means that the projection of the shaker test only goes up to 500Hz in order to not incur in the aliasing phenomenon. During this first process, the Kalman filter was disabled to increase the sampling frequency. To test the performance of this filter, at an early stage, the Kalman filter was built in Matlab, which allowed us to understand all the phases of this filter as well as helped in the development of the Kalman filter of the prototype. This Kalman filter in Matlab is present in appendix A.3 and is used to understand and help in the design of the results. In order to set up the shaker, one must establish a profile for the shaker to use and set the frequency range that the shaker will run. The profile used by the shaker and the established frequency range is shown in figures 53 and 52, respectively.

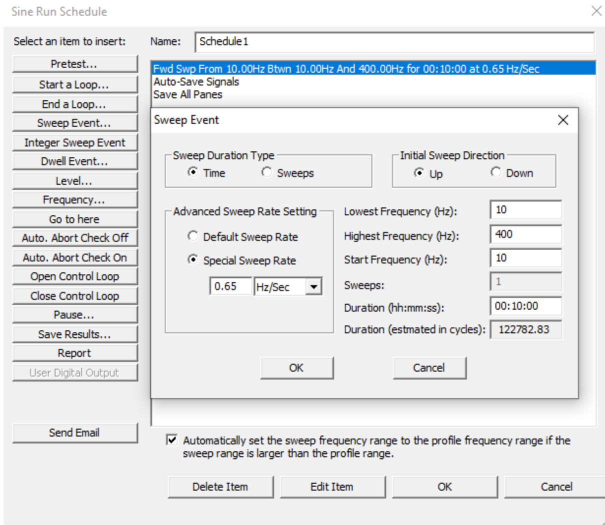


Figure 52: Shaker profile frequency ranges

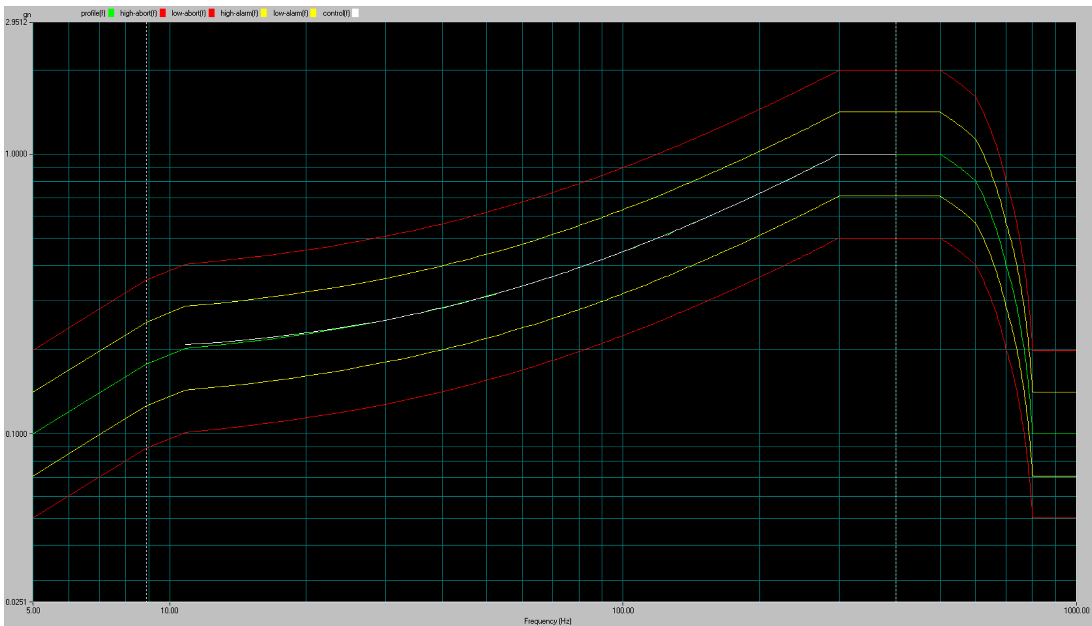


Figure 53: Shaker profile

The figures 54, 55 show the shaker test performed in the X-Axis and Y-Axis. These tests align with the established shaker profile, but deviations in the results are noticeable at higher frequencies. The Kalman filter gives priority to the High-G accelerometer in light of the variance observed in the IMU accelerometer. The existence of vibrations and resonances in the metallic adaptation component also influences the assessment of test performance.

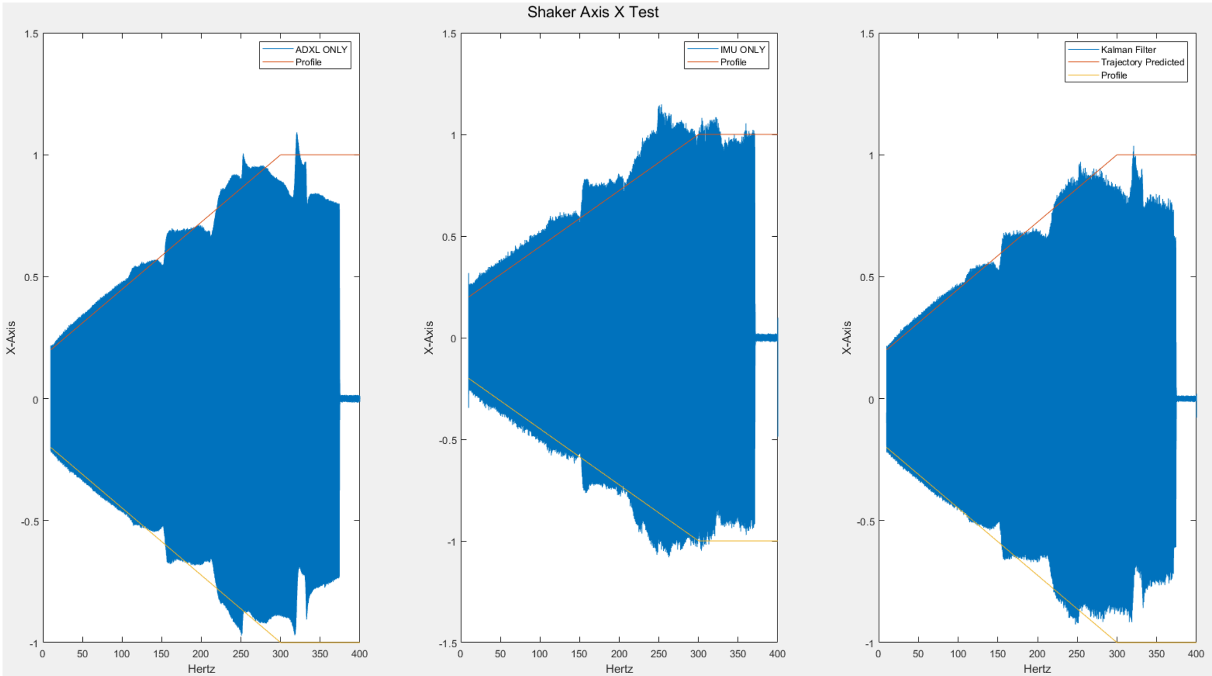


Figure 54: Kalman Shaker X

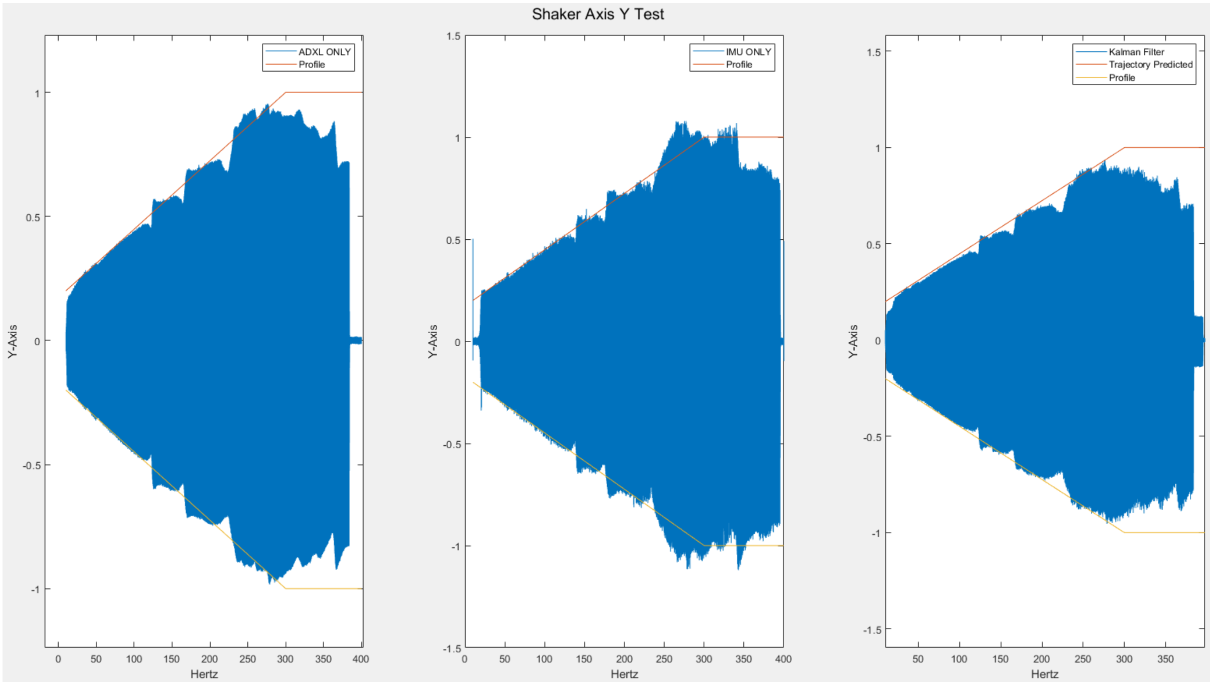


Figure 55: Kalman Shaker Y

Figure 56 presents the results of the shaker test along the Z-axis. The absence of the metal adaptation part leads to improved performance in these tests. The Kalman filter effectively combines the data from the sensors, giving a higher weight to the accelerometer ADXL357.

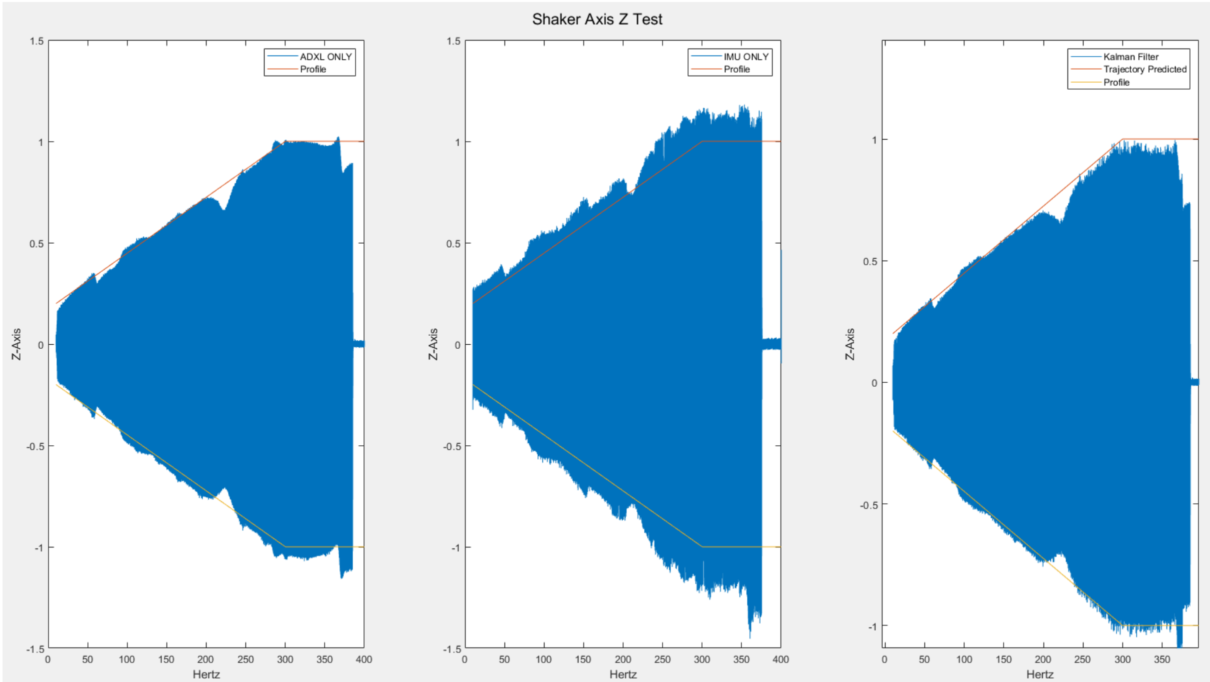


Figure 56: Kalman Shaker Z

However, the Kalman filter of the PCB also needs to be evaluated, so it proceeded again to the shaker

tests, where only the final results are stored after the filter, guaranteeing a sampling frequency of 1KHz, the same as the previous test. This time just the final filter values were shown, as can be seen in figures 57, 58 and 59.

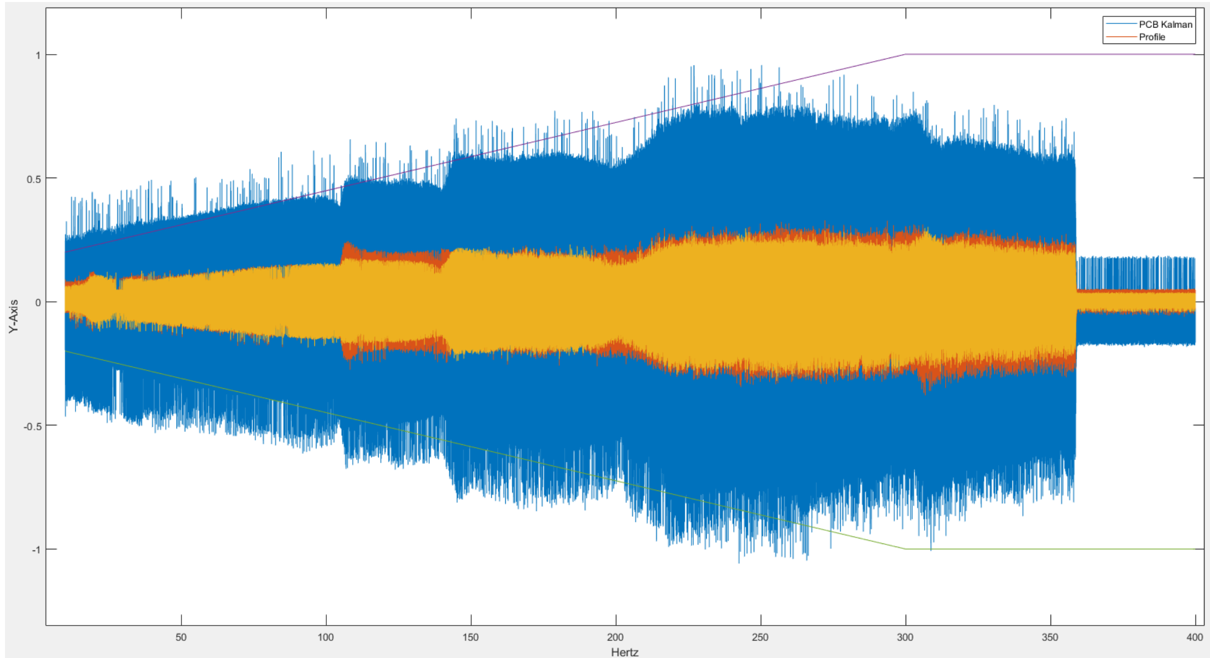


Figure 57: PCB Kalman Shaker Y

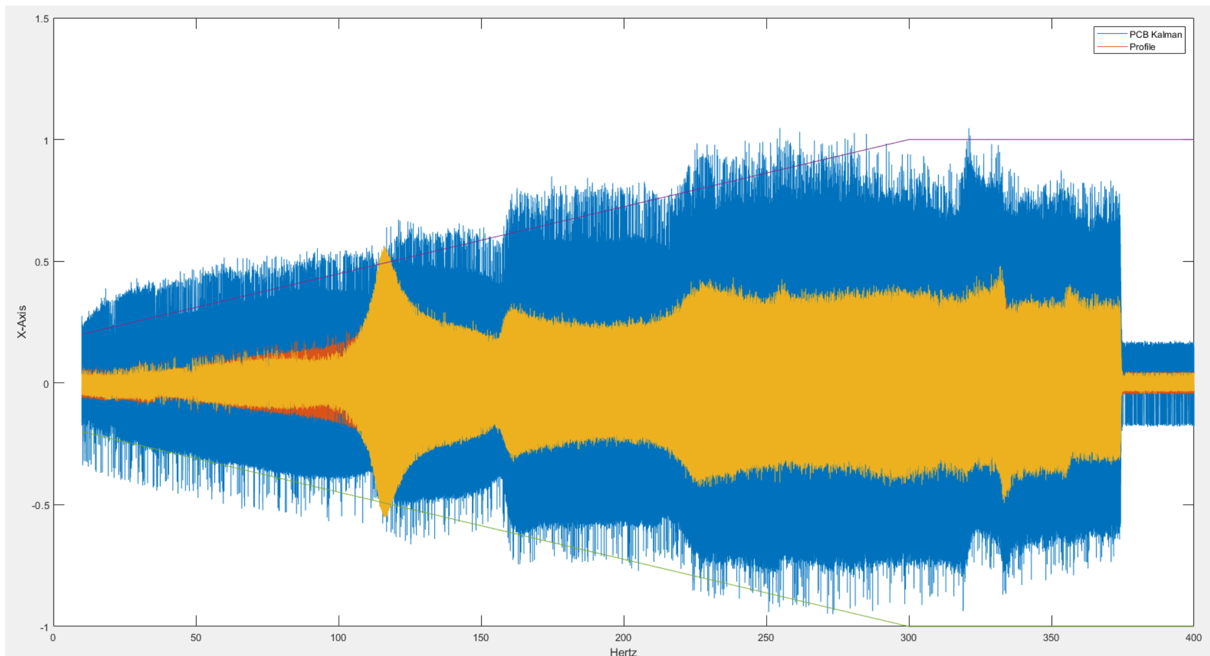


Figure 58: PCB Kalman Shaker X

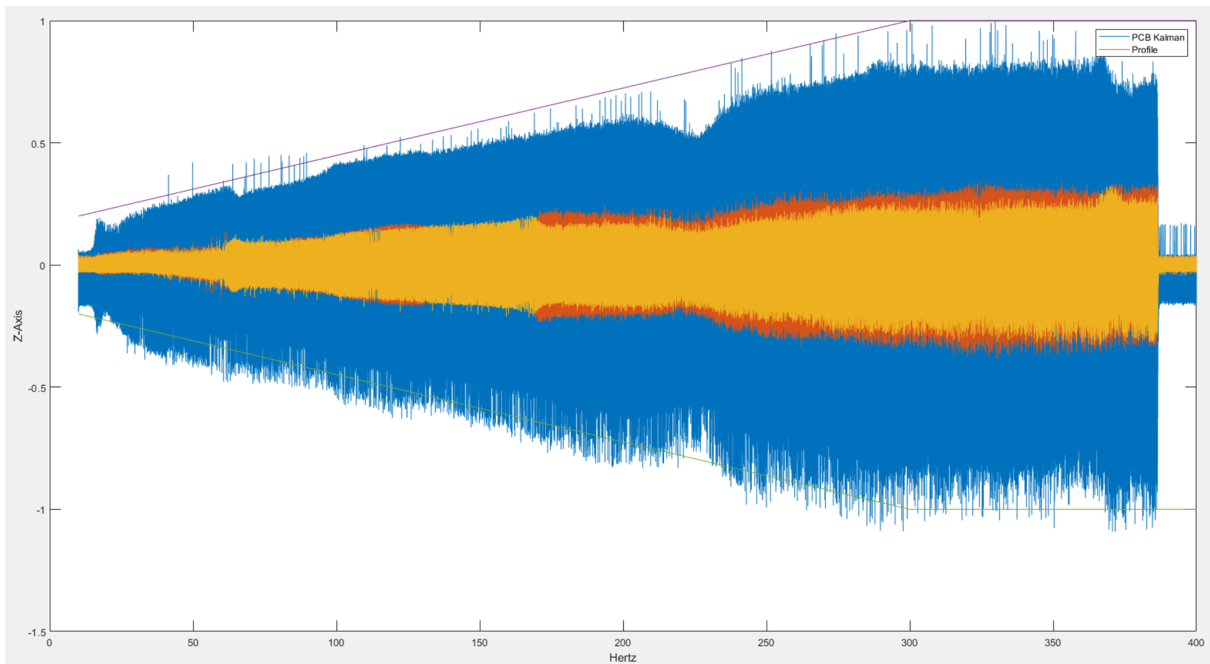


Figure 59: PCB Kalman Shaker Z

The results of our tests indicate that the Kalman filter and the acquisition system effectively adhere to the established profile for the shaker. However, the results obtained are not optimal due to several factors including assembly issues and vibrations in the metal parts. Additionally, the limited accuracy of the two accelerometers used in the tests also impact the results. Despite these limitations, the results are acceptable given the constraints of the test.

To improve the performance of the system, the use of an extended Kalman filter is recommended, as this would provide more accurate results. Furthermore, incorporating additional accelerometers into the system would allow for a more comprehensive approximation of the real values experienced by the board.

In conclusion, despite the limitations, the purpose of the test was successfully achieved as the acquisition system was found to follow the designated profile, which validates its suitability for further use.

6.2 Collision Test

The prototype crash tests were performed by dropping the prototype from a specified height due to limitations in testing resources. The objective of these tests is to evaluate the efficacy of the collision detection algorithm and the accuracy of the data stored on the microSD card. Although not entirely representative of a real-life car crash, the tests are still valid as they simulate similar acceleration patterns. In actual collisions, the acceleration experienced by the car seat is typically lower due to the presence of components that dampen the impact. Conversely, the acceleration experienced by a plate hitting the ground during these tests can sometimes be higher than what would be recorded in most real-world accidents.

Conducting these tests with the plate fully exposed would result in potential damage to the prototype. To mitigate this risk, a protective 3D-modeled box, presented in figure 60 was designed to secure the board inside, ensuring that the acceleration experienced by the box is consistent with that of the prototype. This box, as illustrated in figure 61, features a cover to completely enclose the frame.

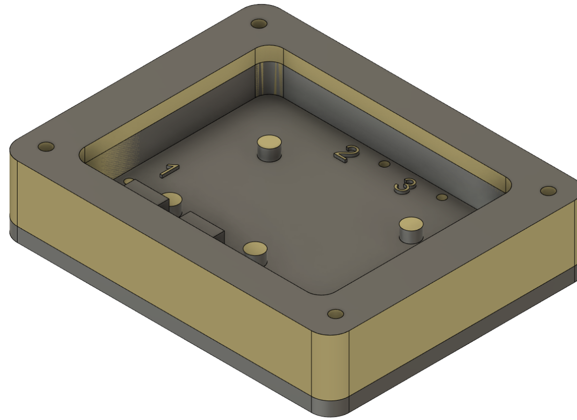
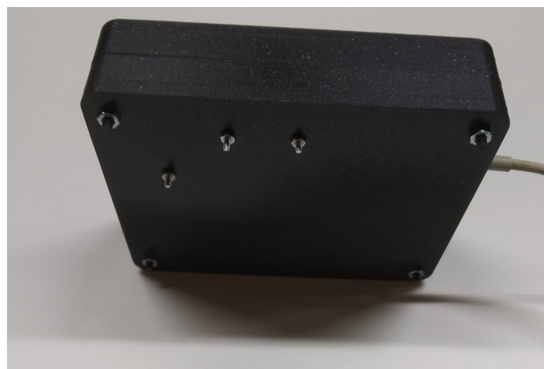


Figure 60: Box enclusure

(a) 3D Box model open



(b) 3D Box model closed

Figure 61: 3D Box model

In accordance with the previous chapters, the board is equipped with the capability to store collision data on a microSD card. To visualize this data, a python script, present in appendix A.4, was developed to display the collision data graphically. Figure 62 presents an example of one such collision, where it can be observed that the data of the collision is accurately recorded, including a five-second window before and after the event, as well as the time of the collision.

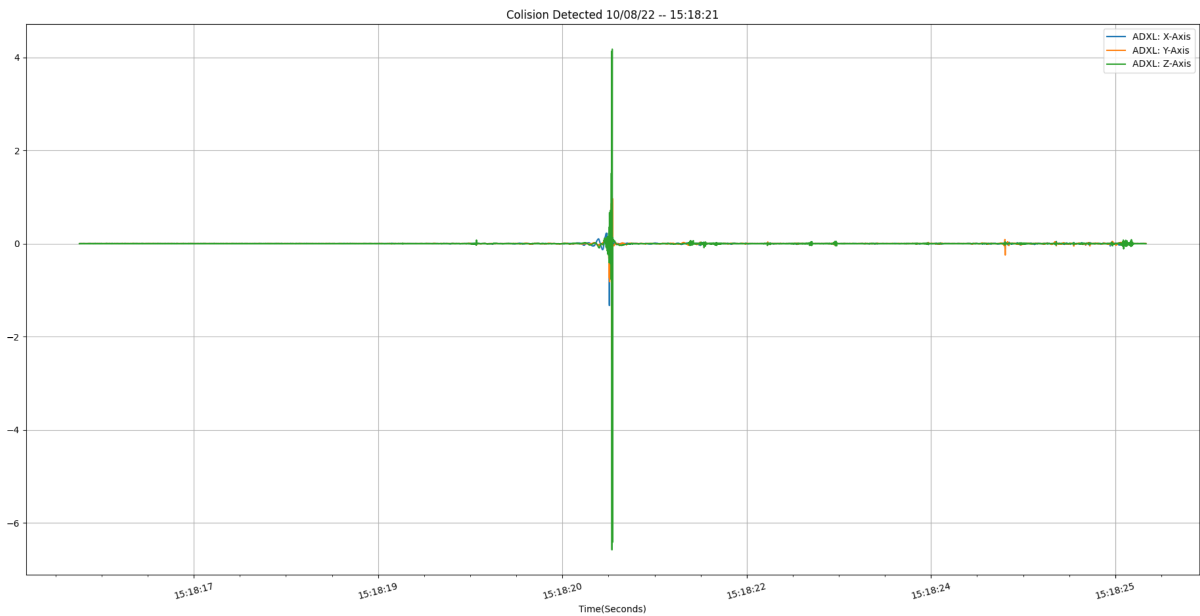


Figure 62: Collision Test

This data suggests that the board effectively detects the collision based on the specified threshold. It is important to note that this test represents an extreme scenario, as a car seat would not typically experience such abrupt accelerations. If such an occurrence were to occur in reality, it would likely result in a serious accident.

To summarize, this test was valuable in demonstrating the effectiveness of the system and verifying its compliance with established requirements. As noted, while this test was not a full-scale car crash test, it successfully demonstrated the system's ability to detect collisions through a defined threshold. The primary objective of this dissertation is to develop an embedded system for collision detection and data acquisition. This objective has been accomplished through the test, although there is room for improvement in terms of refining the algorithm and conducting real-world car collision tests.

Chapter 7

Conclusions

In summary, the goal to construct a device capable of detecting automobile collisions in a car seat has been fulfilled. First, the state-of-the-art showed the study of similar devices and relevant studies to gain a deeper understanding. Then, the project requirements were established, and a solution was presented to meet these requirements. Key components were chosen, and the system was designed using graphs, diagrams, and flowcharts. The hardware and software were then implemented and tested, proving the usefulness of the system and ensuring that all requirements were met.

The collision tests and the corresponding algorithm proved sufficient in detecting most collisions, although the algorithm is considered rudimentary. On the other hand, the shaker tests and the combination of sensor data were performed with some success. The Kalman filter, in particular, demonstrated a good performance in the shaker tests. Despite the limitations posed by the resonance of the metal plate and the limitations of the shaker itself, the results of these tests are promising. The combination of the shaker tests and the Kalman filter's ability to effectively combine and filter sensor data confirms the system's viability for real-world applications. The collision test confirmed that the detection alghoritm can detect a collision by a predefined threshold and the embedded system can save the collision data in the microSD Card for future data reconstruction.

The project achieved its goals and demonstrated that the implemented prototype and the tests performed are able to accurately detect and store collision data. The results of the tests provide a solid foundation for future improvements and developments in the field.

7.1 Future Work

Based on the results and limitations of the current prototype, future work should focus on optimize the Kalman filter and incorporating more sensors, such as IMU or accelerometers, to enhance data fusion. Also, the extended Kalman filter should be used to get a more realistic predictive model. This will lead to a more sophisticated algorithm detection and improved robustness of the system. Furthermore, further tests with real collisions should be performed to validate the system's capability under real-world conditions.

In this project, the mesh actuation integration was not consider a requirement. As future work, connect this device with a car ECU and activate an eletric mesh will provide a full system that can be used to inform medical teams. An integration with other car devices, such as airbags can generate a second opinion about the collision and provide to the airbag control unit more information about the car aceleration.

In summary, the future work should focus on enhance the current system's performance and capabilities, providing a more advanced and reliable solution for detecting and storing collision data. The aim is to continuously improve the system and bring it closer to real-world applications, ensuring its viability and impact in the field.

References

[1] F. E. Vaca, C. L. Anderson, H. Herrera, C. Patel, E. F. Silman, R. DeGuzman, S. Lahham, and V. Kohl. “Crash Injury Prediction and Vehicle Damage Reporting by Paramedics.” In: *Western Journal of Emergency Medicine* 10 (2 May 2009), p. 62. issn: 1936-900X. url: /pmc/articles/PMC2691511//pmc/articles/PMC2691511/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC2691511/.

[2] W. Wach. “Reconstruction of vehicle kinematics by transformations of raw measurement data.” In: *11th International Science and Technical Conference Automotive Safety, AUTOMOTIVE SAFETY 2018* (June 2018), pp. 1–5. doi: 10.1109/AUTOSAFE.2018.8373324.

[3] D. S. Dima and D Covaciu. “Solutions for acceleration measurement in vehicle crash tests.” In: *IOP Conference Series: Materials Science and Engineering* 252 (2017), p. 012007. doi: 10.1088/1757-899x/252/1/012007.

[4] M. Barr. *Programming embedded systems: with C and GNU development tools*. 2006.

[5] *Kernel (operating system) - Wikipedia*. url: https://en.wikipedia.org/wiki/Kernel_(operating_system).

[6] L. Kumar. *Difference between RTOS and GPOS - CPQ Linux*. url: https://www.cpqlinux.com/difference-between-rtos-and-gpos.

[7] *FreeRTOS task states and state transitions described*. url: https://www.freertos.org/RTOS-task-states.html.

[8] *Choosing the Most Suitable MEMS Accelerometer for Your Application—Part 1 | Analog Devices*. url: https://www.analog.com/en/analog-dialogue/articles/choosing-the-most-suitable-mems-accelerometer-for-your-application-part-1.html.

[9] *What is an Inertial Measurement Unit? · VectorNav*. url: https://www.vectornav.com/resources/inertial-navigation-articles/what-is-an-inertial-measurement-unit-imu.

[10] R. W. Diller, M. A. Ross, both of Calif, B. K. Blackburn, and J. F. Mazur. "Method and apparatus for sensing a vehicle crash using energy and velocity as measures of crash violence." In: 875 (358 May 1990), p. 763.

[11] R. W. Diller. "Electronic Sensing of Automobile Crashes for Airbag Deployment." In: *SAE Technical Papers* (Feb. 1991). issn: 0148-7191. doi: 10.4271/910276. url: https://www.sae.org/publications/technical-papers/content/910276/.

[12] J. L. Allen. "Power-Rate Crash Sensing Method for Safety Device Actuation." In: *SAE Technical Papers* (Feb. 1992). issn: 0148-7191. doi: 10.4271/920478. url: https://www.sae.org/publications/technical-papers/content/920478/.

[13] T. Gioutsos. "A Predictive Based Algorithm for Actuation of an Airbag." In: *SAE Technical Papers* (Feb. 1992). issn: 0148-7191. doi: 10.4271/920479. url: https://www.sae.org/publications/technical-papers/content/920479/.

[14] W. Nitschke, W. Kihn, W. Drobny, H. Weller, P. Taufer, E. Jeenicke, and K. Reischle. "Method for controlling the release of passenger restraint systems." In: (Dec. 1989).

[15] R. W. Diller. "Apparatus and method employing multiple crash evaluation algorithms and evaluation expertise for actuating a restraint system in a passenger vehicle." In: (Oct. 1990).

[16] J. Eigler and R. Weber. "Control unit for a passenger restraint system and/or passenger protection system for vehicles." In: (Sept. 1991).

[17] S. Tohbaru. "Collision Determining Circuit Having a Starting Signal Gen- erating Circuit." In: (1993).

[18] R. Cashler and J. Kelley. "SIR Deployment Method Based on Occu- pant Displacement and Crash Severity." In: (1995).

[19] G. McIver. "Method and Apparatus for Sensing a Vehicle Crash Condition Using Velocity Enhanced Acceleration Crash Metrics." In: (1996).

[20] W. K. Kosiak. "Method and apparatus for crash sensing using anticipatory sensor inputs." In: (Feb. 1997).

[21] T. Nguyen and S. Wooters. "FPGA-Based Development for Sophisticated Automotive Embedded Safety Critical System." In: *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 7 (1 2014), pp. 125–132. issn: 19464622. doi: 10.4271/2014-01-0240.

[22] A. Leschke. *Algorithm Concept for Crash Detection in Passenger Cars*. Springer Fachmedien Wiesbaden, 2020, pp. 23–61. isbn: 978-3-658-29391-8. doi: 10.1007/978-3-658-29392-5.

[23] *The Physics of a Car Collision*. url: https://www.thoughtco.com/what-is-the-physics-of-a-car-collision-2698920.

[24] *Acceleration - Wikipedia*. url: https://en.wikipedia.org/wiki/Acceleration.

[25] *g-force - Wikipedia*. url: https://en.wikipedia.org/wiki/G-force.

[26] D. F. Shanahan. *Human Tolerance and Crash Survivability*.

[27] *Peripheral acceleration sensor*. url: https://www.bosch-mobility-solutions.com/en/solutions/sensors/peripheral-acceleration-sensor/.

[28] S. MS, G. ND, and M. AG. *Reliable Automotive Crash Detection using Multi Sensor Decision Fusion Analysis of SDLC Models for Embedded Systems View project Reliable Automotive Crash Detection using Multi Sensor Decision Fusion*. url: https://www.researchgate.net/publication/321759621.

[29] M. S. A. Rupok, H. K. Patnaik, X. Ding, and S. Ganesan. "MEMS accelerometer based low-cost collision impact analyzer." In: vol. 2016-August. IEEE Computer Society, Aug. 2016, pp. 393–396. isbn: 9781467399852. doi: 10.1109/EIT.2016.7535272.

[30] A. Cismas, I. Matei, V. Ciobanu, and G. Casu. "Crash Detection Using IMU Sensors." In: Institute of Electrical and Electronics Engineers Inc., July 2017, pp. 672–676. isbn: 9781538618394. doi: 10.1109/CSCS.2017.103.

[31] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems." In: *Journal of Basic Engineering* 82 (1 Mar. 1960), pp. 35–45. issn: 0021-9223. doi: 10.1115/1.3662552.

[32] L. T. Corporation. *LT1117/LT1117-2.85 LT1117-3.3/LT1117-5 - 800mA Low Dropout Positive Regulators Adjustable and Fixed 2.85V,3.3V, 5V*. url: http://www.linear.com/leadfree/.

[33] A. Devices. *ADXL356/ADXL357 (Rev. A)*. url: www.analog.com.

[34] *MPU-6881 Product Specification Revision 1.0*. 2014. url: www.invensense.com.

[35] *ESP32MINI1 ESP32MINI1U Datasheet 2.4 GHz WiFi + Bluetooth ® + Bluetooth LE module Built around ESP32 series of SoC, Xtensa ® dualcore 32bit LX6 microprocessor 4 MB flash 28 GPIOs, rich set of peripherals Onboard PCB antenna or external antenna connector ESP32MINI1 ESP32MINI1U*. 2021. url: https://espressif.com/sites/default/files/documentation/esp32-mini-1_datasheet_en.pdf.

[36] S. Labs. *USBXpress™ Family CP2102N Data Sheet*.

Appendix 1

A.1 SPI Driver

```
#ifndef __SPI_BUS_H
#define __SPI_BUS_H

#include "driver/spi_common.h"
#include "driver/spi_master.h"

#define SPI_MODE 0
#define MISO_PIN 18
#define MOSI_PIN 23
#define SCLK_PIN 19
#define CS_PIN 10
#define CS1_PIN 9
#define CS3_PIN 4
#define SPI_CLOCK 1000000 // 1 MHz

void spi_master_init(spi_host_device_t host);
esp_err_t spi_receive_transaction(const uint8_t reg_addr, const uint8_t length, uint8_t *data, const spi_device_handle_t handle);
esp_err_t spi_send_transaction(const uint8_t reg_addr, const uint8_t length, uint8_t *data, const spi_device_handle_t handle);
void spi_add_device(const uint8_t CS_PI, spi_device_interface_config_t *dev_config, const spi_device_handle_t *handle, const spi_host_device_t
    ↪ host);

#endif
```

Listing 15: SPI Driver main header file functions

```
void spi_master_init(spi_host_device_t host) {
    spi_bus_config_t config;
    memset(&config, 0, sizeof(spi_bus_config_t));
    config.mosi_io_num = MOSI_PIN;
    config.miso_io_num = MISO_PIN;
```

```
    config.sclk_io_num = SCLK_PIN;
    config.quadwp_io_num = -1; // -1 not used
    config.quadhd_io_num = -1; // -1 not used
    config.max_transfer_sz = SPI_MAX_DMA_LEN;
    spi_bus_initialize(host, &config, 1); // DMA used
}
esp_err_t spi_send_transaction(const uint8_t reg_addr, const uint8_t length, uint8_t *data, const spi_device_handle_t handle) {
    // Create a SPI transaction
    spi_transaction_t spi_trans;
    spi_trans.flags = 0;
    spi_trans.cmd = 0;
    spi_trans.addr = reg_addr;
    spi_trans.length = (uint8_t)length * 8;
    spi_trans.rxlength = (uint8_t)length * 8;
    spi_trans.user = NULL;
    spi_trans.tx_buffer = data; // Pointer to the transmission buffer
    spi_trans.rx_buffer = NULL; // Not used

    return spi_device_transmit(handle, &spi_trans);  // Queue the transaction as a interrupt
}

esp_err_t spi_receive_transaction(const uint8_t reg_addr, const uint8_t length, uint8_t *data, const spi_device_handle_t handle) {
    // Create a SPI transaction
    spi_transaction_t spi_trans;
    spi_trans.flags = 0;
    spi_trans.cmd = 0;
    spi_trans.addr = reg_addr;
    spi_trans.length = (uint8_t)length * 8;
    spi_trans.rxlength = (uint8_t)length * 8;
    spi_trans.user = NULL;
    spi_trans.tx_buffer = NULL; // Not used
    spi_trans.rx_buffer = data; // Pointer to the reveived buffer

    return spi_device_transmit(handle, &spi_trans);  // Queue the transaction as a interrupt
}
```

Listing 16: SPI Driver main functions

A.2 SDCard Driver

```
void mount_sdcard(const char *base_path) {
    sdspi_device_config_t device;
    sdspi_dev_handle_t out_handle;
    sdmmc_host_t host1 = SDSPI_HOST_DEFAULT();
    sdspi_device_config_t slot_config = SDSPI_DEVICE_CONFIG_DEFAULT();
    slot_config.gpio_cs = CS3_PIN;
    slot_config.host_id = host1.slot;


    ESP_LOGI(sd_card_tag, "Initializing SD card, using SPI peripheral");
    // Configure the mount config and mount
    esp_vfs_fat_mount_config_t mount_config = {
```

```
13        .format_if_mount_failed = true,
14        .max_files = 5,
15        .allocation_unit_size = 32*1024};
16    esp_err_t ret = esp_vfs_fat_sdspi_mount(base_path, &host1, &slot_config, &mount_config, &card);
17
18    if (ret != ESP_OK) {
19        if (ret == ESP_FAIL) {
20            ESP_LOGE(sd_card_tag,
21                    "Failed to mount filesystem. "
22                    "If you want the card to be formatted, set format_if_mount_failed = true.");
23        } else {
24            ESP_LOGE(sd_card_tag,
25                    "Failed to initialize the card (%d). "
26                    "Make sure SD card lines have pull-up resistors in place.",
27                    ret);
28        }
29        return;
30    }
31 }
```

Listing 17: SD Card Driver mount function

```
1  const char *base_path = "/fat";
2  /*----------------------------------------------------------------------------------------------*/
3  void sdcard_init() {
4      mount_sdcard(base_path);
5      ESP_LOGI(sd_card_tag, "Mounted in %s", base_path);
6  }
7
8  void sdcard_close() {
9      esp_vfs_fat_sdcard_unmount(base_path, &card);
10     ESP_LOGI(sd_card_tag, "Unmounted");
11 }
```

Listing 18: SD Card Driver init and close functions

A.3 Matlab Kalman

```
1  % Kalman Filter Example
2  % Using Acelerometer data
3
4  %-------------------------------------------
5  % Initialization
6  %-------------------------------------------
7  Data_length = 599996;
8  M = readmatrix('shaker_results_MPU_y_1000Hz.csv',"Range",3);
9  M1 = transpose(M);
10
11 N = readmatrix('shaker_results_ADXL_y_1000Hz.csv',"Range",3);
12 N1 = transpose(N);
13
```

```
14 K = readmatrix('shaker_results_Kalman_y_1000Hz.csv',"Range",3);
15 K1 = transpose(K);
16
17 % Initialization values
18 declive = 0;
19 start_freq = 10;
20 stop_freq = 400;
21 start_profile_freq = 10;
22 stop_profile_freq = 300;
23 profile_value_start = 0.2;
24 profile_value_stop = 1;
25 profile_declive = (profile_value_stop−profile_value_start)/(stop_profile_freq−start_profile_freq);
26 profile_b = 1 − profile_declive*stop_profile_freq;
27 declive = (stop_freq − start_freq)/(Data_length−0); % y = mx + b
28 b = stop_freq − declive*Data_length; % y = mx + b
29 fs = 1000;
30 delta_t = 1/(fs);
31 x_0 = 0;
32 y_0 = 0;
33 z_0 = 0;
34 vx_0 = 0;
35 vy_0 = 0;
36 vz_0 = 0;
37 ax_0 = 0;
38 ay_0 = 0;
39 az_0 = 0;
40 x(1,Data_length) = zeros;
41 y(1,Data_length) = zeros;
42 z(1,Data_length) = zeros;
43 xK_Array(1,Data_length) = zeros;
44 yK_Array(1,Data_length) = zeros;
45 zK_Array(1,Data_length) = zeros;
46 xP_Array(1,Data_length) = zeros;
47 yP_Array(1,Data_length) = zeros;
48 zP_Array(1,Data_length) = zeros;
49
50 for i=1:Data_length
51     Time(1,i) = i;
52     ADXL_x(1,i) = N1(1,i);
53     ADXL_y(1,i) = N1(2,i);
54     ADXL_z(1,i) = N1(3,i);
55     IMU_x(1,i) = M1(1,i);
56     IMU_y(1,i) = M1(2,i);
57     IMU_z(1,i) = M1(3,i);
58     PCB_KALMAN_x(1,i) = K1(1,i);
59     PCB_KALMAN_y(1,i) = K1(2,i);
60     PCB_KALMAN_z(1,i) = K1(3,i);
61     hz(1,i) = (declive*i+b);
62     if hz(1,i) < 300
63         profile(1,i)= (profile_declive*hz(1,i)+profile_b);
64     else
65         profile(1,i) = 1;
66     end
67     profile_neg(1,i)=(profile(1,i)*−1);
```

```
68  end
69  IMU_x = highpass(IMU_x,10,fs);
70  IMU_y = highpass(IMU_y,10,fs);
71  IMU_z = highpass(IMU_z,10,fs);
72
73  VARIANCE_ADXL_10G = 234.2;
74  VARIANCE_ADXL_40G = 159.71;
75  VARIANCE_MPU = 43.92;
76  DELTA_XM = 0.0701;
77  DELTA_YM = 0.0701;
78  DELTA_ZM = 0.0701;
79
80  VARIANCE_A = 0.0049;
81
82  % IMU_x = lowpass(IMU_x,1000,700);
83  % IMU_y = lowpass(IMU_y,1000,700);
84  % IMU_z = lowpass(IMU_z,1000,700);
85
86  x = (ADXL_x*VARIANCE_ADXL_10G + IMU_x*VARIANCE_MPU)/(VARIANCE_MPU+VARIANCE_ADXL_10G);
87  y = (ADXL_y*VARIANCE_ADXL_10G + IMU_y*VARIANCE_MPU)/(VARIANCE_MPU+VARIANCE_ADXL_10G);
88  z = (ADXL_z*VARIANCE_ADXL_10G + IMU_z*VARIANCE_MPU)/(VARIANCE_MPU+VARIANCE_ADXL_10G);
89  %------------------------------------------
90  % Initial Conditions
91  %------------------------------------------
92
93  %Observation
94  delta_xm = sqrt(1/(VARIANCE_MPU+VARIANCE_ADXL_10G)); %m
95  delta_ym = sqrt(1/(VARIANCE_MPU+VARIANCE_ADXL_10G)); %m
96  delta_zm = sqrt(1/(VARIANCE_MPU+VARIANCE_ADXL_10G)); %m
97
98  A = [  1     delta_t    (1/2)*(delta_t^2)  0      0      0                  0      0      0;
99         0     1          delta_t            0      0      0                  0      0      0;
100        0     0          1                  0      0      0                  0      0      0;
101        0     0          0                  1      delta_t (1/2)*(delta_t^2) 0      0      0;
102        0     0          0                  0      1      delta_t            0      0      0;
103        0     0          0                  0      0      1                  0      0      0;
104        0     0          0                  0      0      0                  1      delta_t (1/2)*(delta_t^2);
105        0     0          0                  0      0      0                  0      1      delta_t;
106        0     0          0                  0      0      0                  0      0      1];
107
108 Previous_X = [ x_0;
109             vx_0;
110             ax_0;
111             y_0;
112             vy_0;
113             ay_0
114             z_0;
115             vz_0;
116             az_0 ];
117 %------------------------------------------
118 % The Initial Process Covariance Matrix
119 %------------------------------------------
120 Previous_P =[   500  0   0   0   0   0    0   0   0;
121             0    500 0   0   0   0    0   0   0;
```

```
122             0   0   500 0   0   0    0   0   0;
123             0   0   0   500 0   0    0   0   0;
124             0   0   0   0   500 0    0   0   0;
125             0   0   0   0   0   500  0   0   0;
126             0   0   0   0   0   0    500 0   0;
127             0   0   0   0   0   0    0   500 0;
128             0   0   0   0   0   0    0   0   500];
129
130 Q=[(delta_t^4)/4          (delta_t^3)/2          (delta_t^2)/2          0          0          0          0          0          0;
131   (delta_t^3)/2          delta_t^2              delta_t                0          0          0          0          0          0;
132   (delta_t^2)/2          delta_t                1                      0          0          0          0          0          0;
133    0                      0                      0                      (delta_t^4)/4   (delta_t^3)/2   (delta_t^2)/2   0          0          0;
134    0                      0                      0                      (delta_t^3)/2   delta_t^2       delta_t         0          0          0;
135    0                      0                      0                      (delta_t^2)/2   delta_t         1               0          0          0;
136    0                      0                      0                      0          0          0          (delta_t^4)/4   (delta_t^3)/2   (delta_t^2)/2;
137    0                      0                      0                      0          0          0          (delta_t^3)/2   delta_t^2       delta_t;
138    0                      0                      0                      0          0          0          (delta_t^2)/2   delta_t         1] * sqrt(1/(VARIANCE_MPU
       ↪ +VARIANCE_ADXL_40G));
139
140 % disp("Initial Process Covariance Matrix: ");
141
142 for i=1:Data_length
143
144   %---------------------------------------------
145   % The Predicted State
146   %---------------------------------------------
147     X_p = A * Previous_X;
148
149     % Save data to plot
150     xP_Array(1,i) = X_p(1,1);
151     vxP_Array(1,i) = X_p(2,1);
152     axP_Array(1,i) = X_p(3,1);
153     yP_Array(1,i) = X_p(4,1);
154     vyP_Array(1,i) = X_p(5,1);
155     ayP_Array(1,i) = X_p(6,1);
156     zP_Array(1,i) = X_p(7,1);
157     vzP_Array(1,i) = X_p(8,1);
158     azP_Array(1,i) = X_p(9,1);
159   %---------------------------------------------
160   % The Predicted Process Covariance Matrix
161   %---------------------------------------------
162     Pre = A*Previous_P*transpose(A)+Q;
163
164   %---------------------------------------------
165   % The New Observation
166   %---------------------------------------------
167     C = [0 0 1 0 0 0 0 0 0;
168          0 0 0 0 0 1 0 0 0;
169          0 0 0 0 0 0 0 0 1];
170
171     Y_meas = [0;
172              0;
173              x(1,i);
174              0;
```

```
175                 0;
176                 y(1,i);
177                 0;
178                 0;
179                 z(1,i)];
180
181         Y = C*Y_meas;
182     %-------------------------------------------
183     % Calculating the Kalman Gain
184     %-------------------------------------------;
185         R = [delta_xm^2        delta_ym*delta_xm  delta_zm*delta_xm;
186             delta_xm*delta_ym   delta_ym^2        delta_zm*delta_ym;
187             delta_xm*delta_zm   delta_ym*delta_zm  delta_zm^2;
188             ];
189         K = Pre*transpose(C) / (C*Pre*transpose(C) + R);
190     %-------------------------------------------
191     % Calculating the current State
192     %-------------------------------------------
193         X = X_p + K * ( Y - C * X_p);
194
195         % Save data to plot
196         xK_Array(1,i)   = X(1,1);
197         vxK_Array(1,i)  = X(2,1);
198         axK_Array(1,i)  = X(3,1);
199         yK_Array(1,i)   = X(4,1);
200         vyK_Array(1,i)  = X(5,1);
201         ayK_Array(1,i)  = X(6,1);
202         zK_Array(1,i)   = X(7,1);
203         vzK_Array(1,i)  = X(8,1);
204         azK_Array(1,i)  = X(9,1);
205     %-------------------------------------------
206     % Updating the process Covariance Matrix
207     %-------------------------------------------
208         I=[1 0 0 0 0 0 0 0 0;
209            0 1 0 0 0 0 0 0 0;
210            0 0 1 0 0 0 0 0 0;
211            0 0 0 1 0 0 0 0 0;
212            0 0 0 0 1 0 0 0 0;
213            0 0 0 0 0 1 0 0 0;
214            0 0 0 0 0 0 1 0 0;
215            0 0 0 0 0 0 0 1 0;
216            0 0 0 0 0 0 0 0 1];
217 %     P=(I-K*C)*Pre*transpose(I-K*C) + K*R*transpose(K);
218     P=(I-K*C)*Pre;
219     %-------------------------------------------
220     % Current becomes previous
221     %-------------------------------------------
222         Previous_X = X;
223         Previous_P = P;
224 end
225
226 %-------------------------------------------
227 % Plot Area
228 %-------------------------------------------
```

```
229
230 % Plot aceleration in x,y,z(Kalman + Predicted)
231     figure();
232
233     subplot(1,3,1);
234     plot(hz, ADXL_x, hz, profile, hz, profile_neg);
235     legend('ADXL ONLY', 'Profile');
236     xlabel('Hertz');
237     ylabel('Y−Axis');
238
239     subplot(1,3,2);
240     plot(hz, IMU_x, hz, profile, hz, profile_neg);
241     legend('IMU ONLY', 'Profile');
242     xlabel('Hertz');
243     ylabel('Y−Axis');
244
245     subplot(1,3,3);
246     plot(hz, axK_Array,hz, profile, hz, profile_neg);
247     legend('Kalman Filter', 'Trajectory Predicted', 'Profile');
248     xlabel('Hertz');
249     ylabel('Y−Axis');
250
251     sgtitle('Shaker Axis Z Test')
252
253     figure();
254     plot(hz, ayK_Array,hz, azK_Array,hz, axK_Array, hz, profile, hz, profile_neg);
255     legend('Matlab Kalman', 'Profile');
256     xlabel('Hertz');
257     ylabel('Y−Axis');
258
259     figure();
260     plot(hz, PCB_KALMAN_y,hz, PCB_KALMAN_z,hz, PCB_KALMAN_x, hz, profile, hz, profile_neg);
261     legend('PCB Kalman', 'Profile');
262     xlabel('Hertz');
263     ylabel('Y−Axis');
264
265
266 %     saveas(gcf,'Shaker_x_test.png');
```

Listing 19: Matlab Kalman Implementation

A.4 Python Code to present the results

```
1 ##################################################################################################
2 # Imports
3 ##################################################################################################
4 import matplotlib.pyplot as plt
5 import matplotlib.pyplot as plt1
6 import pandas as pd
7 import seaborn as sns
8 import numpy as np
9 import math
```

```
10 from math import sin, pi
11 import warnings
12 import csv
13 from qbstyles import mpl_style
14
15 ##############################################################################################
16 # Plot Style
17 ##############################################################################################
18 # plt.style.use('C://Users//drede//OneDrive//Ambiente de Trabalho//ShakerData//matplotlib//dracula.mplstyle')
19
20 ##############################################################################################
21 # Initialization
22 ##############################################################################################
23 Average,x,hz,profile_neg,profile,y0,y1,y2,y3,y4,y5,y6,y7,y8 = ([] for i in range(14))
24 row_count = 0
25 declive = 0
26 start_freq = 10
27 stop_freq = 400
28 start_profile_freq = 10
29 stop_profile_freq = 300
30 profile_value_start = 0.2
31 profile_value_stop = 1
32 profile_declive = (profile_value_stop−profile_value_start)/(stop_profile_freq−start_profile_freq)
33 profile_b = 1 − profile_declive∗stop_profile_freq
34 ##############################################################################################
35 # Read data from csv
36 ##############################################################################################
37 # with open('D://Esp32//Thesis//ShakerData//PCB_KALMAN_Y.csv','r') as csvfile:
38 with open('D://Esp32//Thesis//Matlab//shaker_results_Kalman_x_1000Hz.csv','r') as csvfile:
39     plots = csv.reader(csvfile, delimiter = ';')
40     for row in plots:
41         row_count += 1
42         y0.append(float(row[0]))
43         y1.append(float(row[1]))
44         y2.append(float(row[2]))
45         Average.append(math.sqrt(abs(float(row[0])∗float(row[0])) + abs(float(row[1])∗float(row[1])) + abs(float(row[2])∗float(row[2]))))
46 declive = (stop_freq − start_freq)/(row_count−0) # y = mx + b
47 b = stop_freq − declive∗row_count # y = mx + b
48 for i in np.arange(0,row_count,1):
49         hz.append(declive∗i+b)
50         if (hz[i] < 300):
51             profile.append(profile_declive∗hz[i]+profile_b)
52         else:
53             profile.append(1)
54
55         profile_neg.append(profile[i]∗−1)
56 ##############################################################################################
57 # Plot Settings
58 ##############################################################################################
59 left = 0.05
60 right = 0.98
61 bottom = 0.088
62 top = 0.938
63 wspace = 0.2
```

```
64  hspace = 0.2
65  plt.figure("PCB Kalman Shaker Y Test")
66  plt.subplots_adjust(left,bottom,right,top,wspace,hspace)
67  plt.plot(hz,Average, label='Sum of all axis absolute values')
68  plt.plot(hz,y0, label='Kalman - X-Axis')
69  plt.plot(hz,y1, label='Kalman - Y-Axis')
70  plt.plot(hz,y2, label='Kalman - Z-Axis')
71  plt.plot(hz,profile,label='Profile')
72  plt.plot(hz,profile_neg)
73  plt.xlabel('Hz')
74  plt.ylabel('Aceleration')
75  plt.title('10 minutes Samples(W/Kalman)')
76  plt.legend()
77  plt.grid()
78  # plt.show()
79
80  plt.figure("PCB Kalman Shaker X Test")
81  plt.subplots_adjust(left,bottom,right,top,wspace,hspace)
82  plt.plot(hz,Average, label='Sum of all axis absolute values')
83  plt.plot(hz,y0, label='Kalman - X-Axis')
84  plt.plot(hz,y1, label='Kalman - Y-Axis')
85  plt.plot(hz,y2, label='Kalman - Z-Axis')
86  plt.plot(hz,profile,label='Profile')
87  plt.plot(hz,profile_neg)
88  plt.xlabel('Hz')
89  plt.ylabel('Aceleration')
90  plt.title('10 minutes Samples(W/Kalman)')
91  plt.legend()
92  plt.grid()
93  plt.show()
94
95  ##############################################################################################################
96  # Reusable Code
97  ##############################################################################################################
98  # plt.plot(hz,y0, label='Adxl - X-Axis')
99  # plt.plot(hz,y1, label='Adxl - Y-Axis')
100 # plt.plot(hz,y2, label='Adxl - Z-Axis')
101 # plt.plot(hz,y3, label='Mpu - X-Axis')
102 # plt.plot(hz,y4, label='Mpu - Y-Axis')
103 # plt.plot(hz,y5, label='Mpu - Z-Axis')
104 # with open('D://Esp32//Thesis//ShakerData//Kalman_shaker.csv','r') as csvfile:
```

Listing 20: Python script to visualize results