University of Minho

Master's degree in
Industrial Electronic and Computers Engineering

Embedded Systems

IMU Data Acquisition System for Real-Time Processing

Authors:
Marcelo Ribeiro PG54028
Pedro Pereira PG54155

Professors:
Sofia Paiva
Vítor Silva

June 2024

Agenda

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

With the advancement of technology, modern vehicles are increasingly equipped with sensors and electronic systems that enhance their operation and elevate the level of safety in critical automotive systems. In response to this trend, the European Union (EU) has recently enacted legislation mandating the inclusion of Event Data Recorders (EDRs), or black boxes, in all motor vehicles by 2024. These recorders are designed to capture vital data before, during, and immediately after a collision, serving as a crucial tool for analyzing and understanding the causes of accidents.

This project aims to address the challenges posed by the EU legislation and the need for efficient data acquisition systems by proposing the development of an IMU Data Acquisition System for Real-Time Processing. Through the selection of appropriate automotive-grade IMUs, the creation of testing procedures to assess IMU performance, and the design of a data consumer system, this project seeks to not only ensure compliance with regulatory requirements but also contribute to advancing road safety and accident analysis.

1.2 Problem Statement Analysis

To address this problem statement multiple stages are involved, such as:

- Conduct in-depth research on the state-of-the-art IMUs to select three most suitable automotive-grade IMUs.
- Test the IMUs on three different development boards to obtain a broader range of data acquisition.
- Implement Kalman Filter (both software and FPGA-accelerated) for noise filtering.
- Develop a car crash detection algorithm.
- Create a program to visualize IMU behavior in a 3D view for vehicle movement analysis.
- Implement a replay mode to visualize the full accident exactly as it happened.
- Compare all the obtained results.

Chapter 2

Analysis

2.1 State of Art

In this section will be presented a brief explanation on what is an Inertial Measurement Units and the some of the most interesting IMUs currently available.

2.1.1 IMU

Inertial Measurement Units (IMUs), can encompass up to three sensors gathering information from three different axes:

- **Accelerometer**, which measures the G-Force/acceleration;
- **Gyroscope**, which measures the angular rate in degrees per seconds (DPS);
- **Magnetometer**, that measures the magnetic field, acting like a compass.

This characteristics when fused together, make a system capable of acquiring and processing such high volumes of data in real-time of most importance, particularly when utilizing this data to detect collisions.

IMUs have become essential across industries, since they find applications in aerospace and aviation where they provide crucial data for flight control systems, also involved in sports training applications that need to measure, for example, the precise angle and force of a swing in golf or baseball.

Overall, IMUs have become indispensable tools across various fields, contributing to safety, efficiency, and innovation in a wide range of applications.

Market Research

Although our market research uncovered several notable IMUs, represented in Figure 2.1, it's worth noticing that among these options, we also identified a fourth IMU that serves as a developer kit. This developer kit, while not a final automotive-grade IMU, presents a valuable resource for conducting initial tests and evaluations.

These preliminary assessments will provide essential insights and groundwork before transitioning to the selection and integration of the chosen automotive-grade IMU into our system.

((a)) InvenSense IMU

((b)) BOSH IMU

((c)) STM IMU

((d)) BOSH-BMI088 Dev Kit

Figure 2.1: IMU Research

2.1.2 Chosen IMUs

The three IMUs chosen for this project are detailed below. The BMI323 from Bosch serves as the base IMU and served as a reference for selecting the other two IMUs: the BMI088, also from Bosch, and the ASM330LHB from STMicroelectronics. These IMUs feature high-accuracy 6-axis inertial measurement units with 3-axis digital gyroscopes and accelerometers.

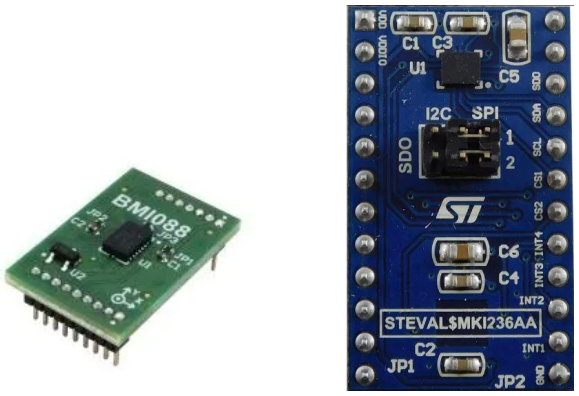
While these IMUs may appear similar, STMicroelectronics' ASM330LHB offers additional functionalities compared to Bosch's offerings. This sensor not only includes an embedded machine learning core but also features a state machine.

Developed to meet ASIL B (Automotive Safety Integrity Level) requirements, this sensor is used in applications such as satellite precise positioning, V2X (Vehicle-to-everything) communication, ADAS (Advanced Driver-Assistance Systems), and more.

((a)) BMI088 IMU ((b)) ASM330LHB IMU ((c)) BMI323 IMU

Figure 2.2: Choosen IMU

In order to facilitate the extraction of data from the sensors as much as possible, we opted for the respective DevKits.



((a)) Bosch Shuttleboard 3.0 ((b)) STEVAL-MKI236A

Figure 2.3: Choosen DevKits

2.2 IMU Study

In this section, we will understand the requirements and specifications of each IMU and define the functionality needed from the driver (e.g., data acquisition, calibration, interrupts).

2.2.1 BMI323 IMU

The BMI323 is a highly integrated, low power inertial measurement unit (IMU) that combines precise acceleration and angular rate (gyroscopic) measurement with intelligent on-chip motion-triggered interrupt features The IMU is RoHS (Restriction of Certain Hazardous Substances) compliant, halogen and lead free.

Pin #	Name	I/O Type	Description	Connect to		
				in SPI 4w	in SPI 3w	in I²C/I3C
1	SDO	Digital I/O	SDO Serial data output in SPI 4W	SDO	DNC	GND for default I²C address
			I²C Address bit-0 select in I²C mode			
2	NC	Digital I/O		do not connect		
3	NC	Digital I/O		do not connect		
4	INT1	Digital I/O	Interrupt pin 1	INT1	INT1	INT1
5	VDDIO	Supply	Digital I/O supply voltage 1.08V ... 3.63V	VDDIO	VDDIO	VDDIO
6	GNDIO	Ground	Ground for I/O	GNDIO	GNDIO	GNDIO
7	GND	Ground	Ground for digital & analog	GND	GND	GND
8	VDD	Supply	Power supply analog & digital domain 1.71V ... 3.63V	VDD	VDD	VDD
9	INT2	Digital I/O	Interrupt pin 2	INT2	INT2	INT2
10	NC			do not connect		
11	NC			do not connect		
12	CSB	Digital in	Chip select for SPI mode	CSB	CSB	VDDIO
13	SCx	Digital in	SCK for SPI serial clock	SCK	SCK	SCL
			SCL for I²C serial clock			
14	SDx	Digital I/O	SDA serial data I/O in I²C/I3C	SDI	SDIO	SDA
			SDI serial data input in SPI 4-wire			
			SDA serial data I/O in SPI 3-wire			

Figure 2.4: BMI323 Pinout and pin connections - Datasheet pag. 198

By default, the device operates in I2C mode. The interface of the device can be configured to operate in a SPI 4-wire configuration as well. It can also be re-configured by software to work in 3-wire mode instead of 4-wire mode. All three possible digital interfaces share partly the same pins. The mapping for the primary interface of device is given in Figure 2.5.

Pin #	Name	I/O Type	Description	in SPI4W	in SPI3W	in I²C/I3C
1	SDO	Digital I/O	SDO Serial data output in SPI 4W; I²C Address bit-0 select in I²C mode	SDO	DNC	GND for default I²C address
4	INT1	Digital I/O	Interrupt pin 1	INT1	INT1	INT1
9	INT2	Digital I/O	Interrupt pin 2	INT2	INT2	INT2
12	CSB	Digital in	Chip select for SPI mode	CSB	CSB	VDDIO
13	SCx	Digital in	SCK for SPI serial clock ; SCL for I²C/ I3C serial clock	SCK	SCK	SCL
14	SDx	Digital I/O	SDA serial data I/O in I²C/I3C; SDI serial data input in SPI 4W; SDA serial data I/O in SPI 3W	SDI	SDA	SDA

Figure 2.5: BMI323 Pin mapping for digital interface - Datasheet pag. 212

2.2.2 BMI088 IMU

The BMI088 is a high-performance 6-axis inertial measurement unit (IMU) designed by Bosch for a variety of applications, including automotive and industrial use. The chip contains a total of 16 pins for power and control, as shown in figure 2.6.

Pin#	Name	I/O Type	Description	SPI mode	I²C mode
1*	INT2	Digital I/O	Interrupt pin 2 (accel int #2)	INT2	INT2
2	NC	--	--	GND	GND
3	VDD	Supply	Power supply analog & digital domain (2.4 – 3.6V)	VDD	VDD
4	GNDA	Ground	Ground for analog domain	GND	GND
5	CSB2	Digital in	SPI Chip select Gyro	CSB2	DNC (float)
6	GNDIO	Ground	Ground for I/O	GND	GND
7	PS	Digital in	Protocol select gyroscope (GND = SPI, VDDIO = I²C)	GND	VDDIO
8	SCL / SCK	Digital in	SPI: serial clock SCK I²C: serial clock SCL	SCK	SCL
9	SDA / SDI	Digital I/O	I²C: SDA serial data I/O SPI 4W: SDI serial data I SPI 3W: SDA serial data I/O	SDI	SDA
10	SDO2	Digital out	SPI Serial data out Gyro Address select in I²C mode see chapter 9.2	SDO2	GND for default addr.
11	VDDIO	Supply	Digital I/O supply voltage (1.2V ... 3.6V)	VDDIO	VDDIO
12*	INT3	Digital I/O	Interrupt pin 3 (gyro int #1)	INT3	INT3
13*	INT4	Digital I/O	Interrupt pin 4 (gyro int #2)	INT4	INT4
14	CSB1	Digital in	SPI Chip select Accel	CSB1	VDDIO or DNC (float)
15	SDO1	Digital out	SPI Serial data out Accel Address select in I²C mode see chapter 9.2	SDO1	GND for default addr.
16*	INT1	Digital I/O	Interrupt pin 1 (accel int #1)	INT1	INT1

Figure 2.6: BMI088 Pinout and pin connections - Datasheet pag. 52

In this project, since the chosen data transfer protocol is I2C, to establish I2C communication with the IMU I2C slave, the 'PS' pin must be set to VDDIO. At this point, the SDA (I2C data) and SCL (I2C clock) pins must be connected, and the I2C is ready to use if the 'VDDI' and 'VDDIO' pins are provided with appropriate

voltage (typically 3.3V), and the ground pins (GND, GNDA, and GNDIO) are connected to the ground of the development board.

The interrupt pins (INT1, INT2, INT3, INT4) can be configured to trigger on specific events, allowing the development board to perform other tasks instead of continuously polling the device.

Since we pulled both SDO1 and SDO2 to VDDIO, the default I2C addresses are 0x19 for the accelerometer and 0x69 for the gyroscope.

BMI088 Register Configuration

To start the IMU data transfer, key registers must be configured. Important registers include `ACC_CONF` (0x40), which configures and powers the accelerometer sensor. The gyroscope is already running after power-up.

After configuration, IMU data can be read from register addresses 0x12 to 0x17 for accelerometer output data, 0x22 to 0x23 for temperature data, and 0x02 to 0x07 for gyroscope data.

BMI088 FIFO

BMI088 offers two integrated FIFO (First In, First Out) buffers for accelerometer and gyroscope sensor signals, helping the user to reduce or even omit time critical read access to the sensor in order to obtain data with a high timing precision.

The FIFO operates in two main modes:

- **FIFO (or stop-at-full)** In this mode, sensor values are sequentially stored in the FIFO buffer until it reaches full capacity.
- **STREAM** Similar to FIFO mode, but in STREAM mode, once the buffer is full, the oldest data is overwritten with the newest sensor data.

The FIFO supports two types of interrupts:

- **Watermark Interrupt** Triggered when the FIFO fill level reaches a user-defined threshold.
- **FIFO-full Interrupt** Triggered when the FIFO buffer is completely full.

The BMI088 accelerometer includes a dedicated 1024-byte data FIFO. It captures data from sensor data registers in frames, with each frame containing a single sensor data sample.

The gyroscope part of BMI088 features an integrated FIFO memory capable of storing up to 100 frames of data in FIFO mode. Each frame consists of three 16-bit rate_x,y,z data words, and 16 bits of interrupt data sampled at the same point in time.

The FIFO input data rate corresponds to the configured Output Data Rate (ODR) of the sensor. Adjust the data sampling rate by setting a down-sampling factor 2^k with k ranging from 0 to 7. Configure this factor by writing to bits 4-6 of register 0x45 (refer to section 5.3.12 of the datasheet).

If INT1 and/or INT2 pins are configured as input pins (using int2_io in register INT2_IO_CTRL and/or int1_io in register INT1_IO_CTRL), signals on these pins can be recorded in the FIFO. Ensure these pins are activated for FIFO recording in FIFO_CONFIG_1 register (refer to section 5.3.15).

2.2.3 ASM330LHB IMU

The ASM330LHB is a high-accuracy 6-axis inertial measurement unit (IMU) for ASIL B automotive applications made by STMicroelectronics. The chip contains a total of 14 pins to power and control the sensor, as can be seen in Figure 2.7.

Pin #	Name	Pin function	Pin status
1	SDO	SPI 4-wire interface serial data output (SDO)	Default: input without pull-up Pull-up is enabled if bit SDO_PU_EN = 1 in register PIN_CTRL (02h).
	SA0	I²C least significant bit of the device address (SA0) MIPI I3C℠ least significant bit of the static address (SA0)	
2	RES	Connect to Vdd_IO or GND	
3	RES	Connect to Vdd_IO or GND	
4	INT1	Programmable interrupt 1. If device is used as MIPI I3C℠ pure slave, this pin must be set to 1.	Default: input with pull-down(1) Pull-down is disabled if bit PD_DIS_INT1 = 1 in register I3C_BUS_AVB (62h).
5	Vdd_IO	Power supply for I/O pins	
6	NC	Connect to Vdd_IO or GND or leave unconnected	
7	GND	0 V supply	
8	Vdd	Power supply	
9	INT2	Programmable interrupt 2 (INT2) / data enabled (DEN)	Default: output forced to ground
10	NC	Leave unconnected	
11	NC	Leave unconnected	
12	CS	I²C and MIPI I3C℠/SPI mode selection (1:SPI idle mode / I²C and MIPI I3C℠ communication enabled; 0: SPI communication mode / I²C and MIPI I3C℠ disabled)	Default: input with pull-up Pull-up is disabled if bit I2C_disable = 1 in register CTRL4_C (13h) and I3C_disable = 1 in register CTRL9_XL (18h).
13	SCL	I²C/MIPI I3C℠ serial clock (SCL) / SPI serial port clock (SPC)	Default: input without pull-up
14	SDA	I²C/MIPI I3C℠ serial data (SDA) / SPI serial data input (SDI) / 3-wire interface serial data output (SDO)	Default: input without pull-up

Figure 2.7: ASM330 Pinout and pin connections - Datasheet pag. 36

In this project, since the chosen data transfer protocol is I2C, as analyzed above, in order to establish I2C communication with the IMU I2C slave, the 'CS' pin must be set to 0. At this point, the SDA (I2C data) and SCL (I2C clock) lines must be connected, and the I2C is ready to use if the 'Vdd' and 'Vdd_IO' pins also have a voltage of 3.3 volts and share a ground with the development board.

The two remaining relevant pins are interrupt pins, which can be configured to be asserted when certain events occur. This is useful because it allows the development board to execute other tasks instead of continuously polling the device to check if an event has happened.

To start the IMU data transfer, two very important registers must be configured: the CTRL1_XL (10h) and CTRL2_G (11h) registers, which configure the ODR (Output Data Rate) and the sensor's range.

After configuring, the IMU data can be read from register addresses 20h to 2Ch, where the temperature, gyroscope, and accelerometer output data registers are located.

ASM330LHB Machine Learning Core

The ASM330LHB includes a dedicated core for machine learning (ML) processing, enhancing system flexibility by allowing some algorithms to be shifted from the application processor to the MEMS sensor, significantly reducing power consumption. This ML core can identify if a data pattern, such as motion, pressure, temperature, or magnetic data, matches user-defined classes, making it ideal for applications like activity detection (e.g., running, walking, driving).

It processes data from the accelerometer and gyroscope sensors, with input data filtered and processed within a user-defined time window using configurable computation blocks. The ML processing is based on logical operations composed of configurable nodes characterized by "if-then-else" conditions, where feature values are evaluated against defined thresholds. However, due to lack of time and since its not the main goal of this project this feature was not explored, although it can be integrated in a future work.

ASM330LHB FIFO

The presence of a FIFO allows consistent power saving for the system since the host processor does not need to continuously poll data from the sensor. Instead, it can wake up only when needed and retrieve significant data from the FIFO.

The ASM330LHB includes a 3 KB FIFO to store the following data:

- Gyroscope
- Accelerometer
- Timestamp
- Temperature

Writing data to the FIFO is triggered by the accelerometer/gyroscope data-ready signal. Applications have maximum flexibility in choosing the rate of batching

for physical sensors with FIFO-dedicated configurations. The batch rates for the accelerometer, gyroscope, and temperature sensor can be selected by the user. Additionally, timestamp batching in the FIFO can be decimated by a factor of 1, 8, or 32.

Reconstruction of a FIFO stream is straightforward due to the FIFO_DATA _OUT_TAG byte, which allows recognizing the meaning of a word in the FIFO. The FIFO allows correct reconstruction of timestamp information for each sensor stored in it. If there is a change in the ODR (Output Data Rate) or BDR (Batch Data Rate) configuration, the application can correctly reconstruct the timestamp and know exactly when the change was applied without disabling FIFO batching. The FIFO stores information on the new configuration and the timestamp when the change was applied.

The programmable FIFO watermark threshold can be set in FIFO_CTRL1 (07h) and FIFO_CTRL2 (08h) using the WTM[8:0] bits. To monitor the FIFO status, dedicated registers FIFO_STATUS1 (3Ah) and FIFO_STATUS2 (3Bh) can be read to detect FIFO overrun events, FIFO full status, FIFO empty status, FIFO watermark status, and the number of unread samples stored in the FIFO. Dedicated interrupts on the INT1 and INT2 pins for these status events can be configured in INT1_CTRL (0Dh) and INT2_CTRL (0Eh).

The FIFO buffer can be configured according to six different modes:

- **Bypass** mode
- **FIFO** mode
- **Continuous** mode
- **Continuous-to-FIFO** mode
- **Bypass-to-continuous** mode
- **Bypass-to-FIFO** mode

Each mode is selected by the FIFO_MODE_[2:0] bits in the FIFO_CTRL4 (0Ah) register

However for the same reasons presented in the previous section this feature will not be used.

2.3 Technology Stack

The technology stack presented in figure 2.8 provides a representation of the different technologies and components that work together to ensure the proper functioning of the system.

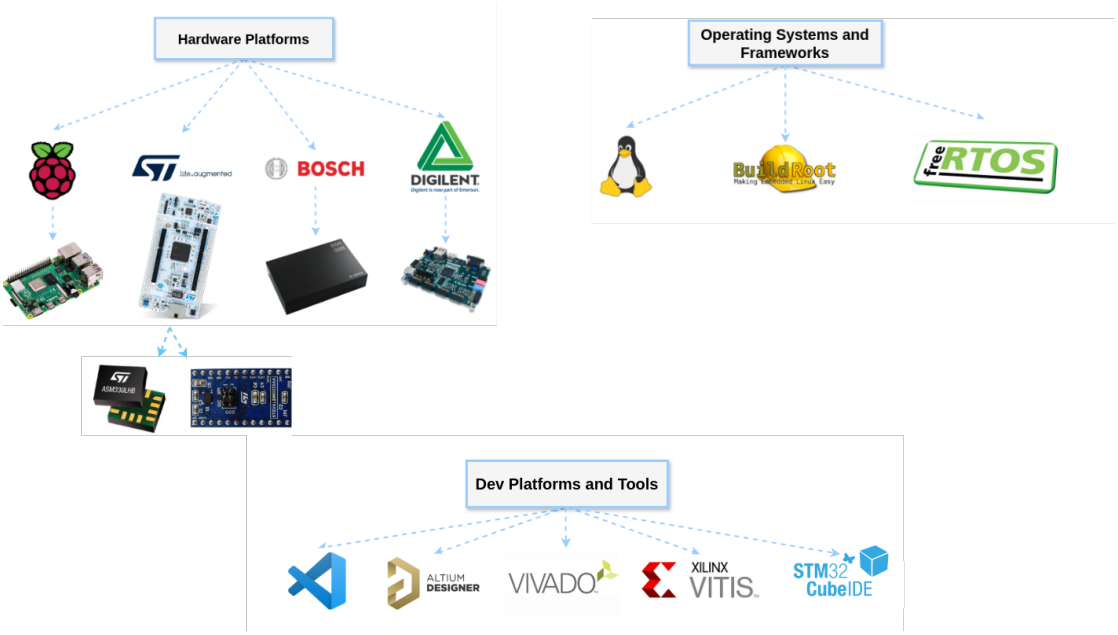


Figure 2.8: Technology Stack

2.3.1 Vitis HLS

For the hardware accelerator kalman filter, we will use Vitis HLS to convert the Kalman filter implemented in C code into RTL (Register Transfer Level) code. This IP will be exported to Vivado to be connected to the FPGA processing system via AXI (Advanced eXtensible Interface), which is a protocol for efficient interconnect in FPGA and SoC designs.

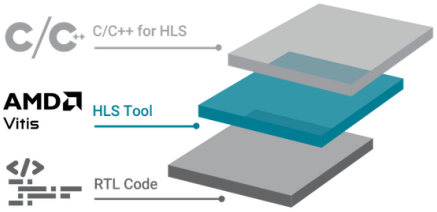


Figure 2.9: Vitis HLS tool

2.4 Systems Overview

2.4.1 STM32 System Overview

The STM32 system is straightforward, consisting solely of the microcontroller and its corresponding IMU. The communication protocol used is I2C. This setup will be used for initial testing with the acquired development kits, focusing on implementing the first version of the device driver and gaining familiarity with the IMUs.

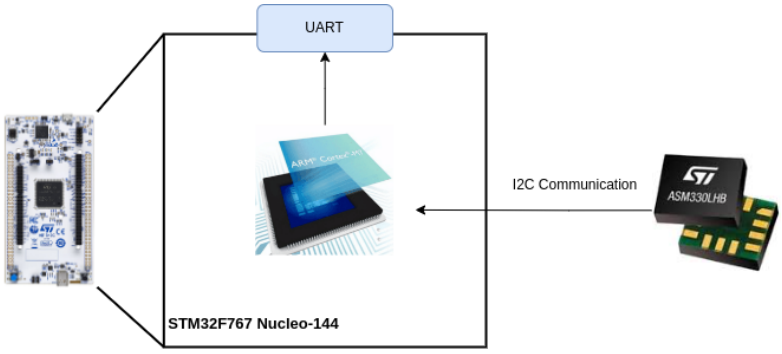


Figure 2.10: STM32 System overview

2.4.2 Raspberry System Overview

The Raspberry Pi's system overview is represented in Figure 2.11. The data communication between the board and the IMU is again established via I2C. After receiving the raw data, the Kalman filter processes it and, depending on the actual system state (FSM explained later in the Design), places the filtered data either on the memory card or in the UART buffer.

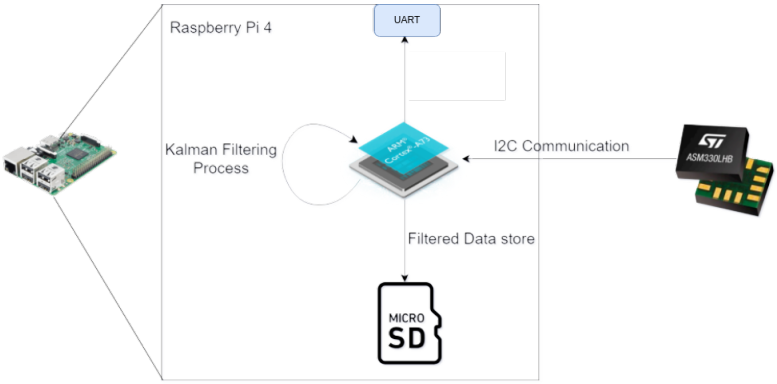


Figure 2.11: Raspberry overview

2.4.3 Zybo System Overview

In the FPGA system, the input data receiving protocol for all three sensors is, again, I2C. By connecting the IMU sensors to the FPGA PMOD, the Zybo processing system (PS) processes the data, and the hardware-accelerated Kalman filter processes it further, sending it back to the PS. Afterwards, the final data is sent to the UART buffer, as can be seen in Figure 2.12.

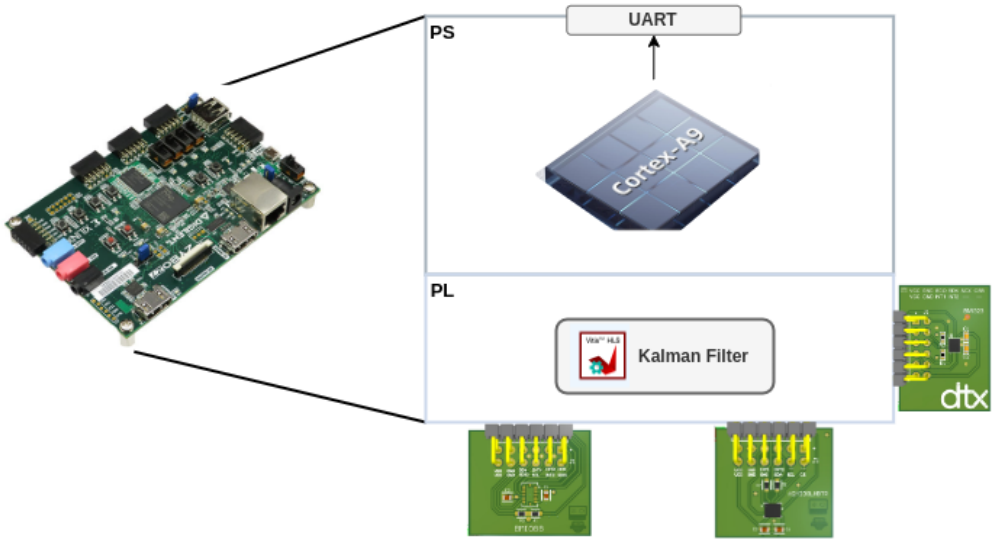


Figure 2.12: FPGA System Overview

2.5 Kalman Filter

When reading data from sensors, such as accelerometers and gyroscopes, various challenges can arise, including inherent sensor noise, environmental interference, and measurement inaccuracies. These factors can lead to noisy and unreliable data over time, making it challenging to accurately estimate the true state of the system being monitored.

To address these challenges, the Kalman filter emerges as a powerful solution. The Kalman filter is an algorithm that combines noisy sensor measurements with a dynamic system model to estimate the true state of the system. By leveraging the system dynamics and the statistical properties of the sensor measurements, the Kalman filter provides a robust estimation that is less affected by noise and measurement errors. This effect can be observed in Figure 2.13, where the raw sensor measurements are depicted in blue, and the results after being processed by the Kalman Filter are shown in red.

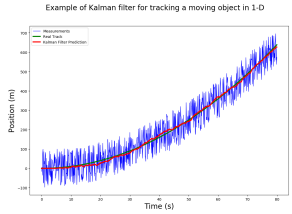


Figure 2.13: Kalman filter actuation

After defining the dynamic system and measurement models, initializing the filter with initial state estimates and covariance matrices, predicting the next state using the system model, and correcting the predicted state using sensor measurements, the Kalman filter iterates through this prediction-correction cycle for each new measurement to dynamically update the state estimates. This iterative process enables the Kalman filter to effectively filter out noise and provide accurate estimates of the true system state. An accompanying image (Fig.2.14) with the formulas of the Kalman filter will further elucidate the implementation process.

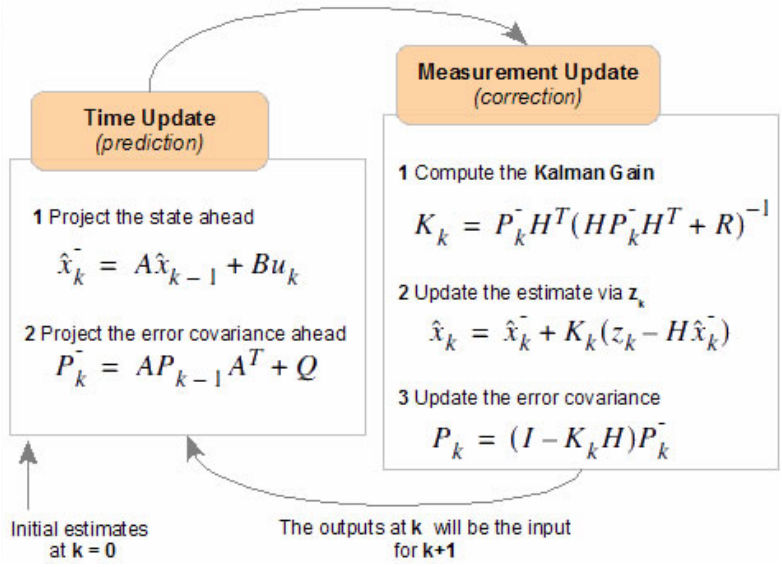


Figure 2.14: Kalman Formulas

In automotive-grade applications, the Kalman filter is indispensable for its ability to refine sensor data, ensuring precision and reliability in critical systems. By effectively reducing noise and correcting errors in sensor measurements, this way we can guarantee that vehicles are able to make accurate decisions in real-time, enhancing safety and performance.

2.6 Sensor Fusion

As mentioned before IMU is a sensor suite that includes an accelerometer and gyroscope, while a "MARG" sensor (Magnetic, Angular Rate, and Gravity), contains an IMU plus a magnetometer. These sensors measure physical properties related to Earth's fields or orientation due to angular momentum. Although, individually, these sensors have limitations that the others can compensate for, for instance:

- When a sensor's axis aligns with Earth's field, trigonometric functions cannot determine orientation because tan(90) is undefined.
- When the gyroscope provides the angular velocity in degrees per second, it is theoretically possible to determine the current orientation of the IMU. However, the only method available (using only the gyroscope) is to integrate the angular velocity over time to obtain the angular displacement in degrees. Unfortunately, this integration process leads to significant data drifts over time, mainly due to accumulation of integration errors.
- When trying to obtain space position by integrating the acceleration twice overtime which would cause the same problem as the gyroscope integration.

Therefore, fusing these sensors is critical for achieving accurate orientation and ensuring a complete transformation from the sensor's inertial frame to the Earth frame.

2.6.1 Madgwick Filter

The Madgwick Filter is a popular method for fusing data from an IMU, and optionally a MARG. It utilizes gradient descent to optimize a quaternion that aligns accelerometer data to a known reference of gravity. This quaternion is weighted and integrated with the gyroscope's quaternion and the previous orientation. The result is then normalized and converted to Euler angles, providing a reliable measure of orientation.

A quaternion is a four-dimensional number, extending the complex plane, comprising one real component and three imaginary components. Unlike Euler angles that rotate about three axes, a quaternion rotates a certain amount of degrees about a vector. The mathematical operations involving quaternions are significantly simpler compared to Euler rotation matrices, making them advantageous for sensor fusion in IMUs and MARGs. This simplicity and efficiency in computation contribute to the accuracy and reliability of sensor fusion in real-time applications.

2.7 OpenGL for a 3D Scenario

OpenGL (Open Graphics Library) is a widely-used, cross-language, cross-platform API for rendering 2D and 3D vector graphics. It provides a powerful set of functions and tools to create complex visualizations by leveraging the capabilities of modern graphics hardware. OpenGL is particularly well-suited for applications requiring high-performance rendering, such as video games, simulations, and scientific visualization.

Figure 2.15: OpenGL

In this project, OpenGL will be utilized to develop a program for visualizing a 3D model of the IMU rotating and moving in space. By using OpenGL, we can render the IMUs orientation and movements in real-time, providing a clear and detailed view of its behavior in three-dimensional space. This visualization will be crucial for analyzing the IMU data, especially in the context of vehicle dynamics and accident analysis. The ability to see the IMUs movements in 3D will enhance our understanding of the data, leading to faster implementation and verification phases of this project.

Chapter 3

Design

3.1 Hardware Platforms

3.1.1 STM32F767 Nucleo-144

This board will be used to perform initial testing and driver implementation for the selected IMUs and the board's development environment to ensure rapid prototyping and debugging.

Figure 3.1: STM32F767 Nucleo-144

3.1.2 Raspberry Pi 4B

The Raspberry Pi is great for developing advanced real-time systems due to its high computational power, which allows it to handle complex algorithms and data processing tasks efficiently. However, due to unreliable connections caused by loose wires (IMU and UART connections), this setup is not ideal for comparing different IMUs. In this project, the Raspberry Pi will primarily be used to emulate a real-world scenario, such as an Event Data Recorder in an actual vehicle.

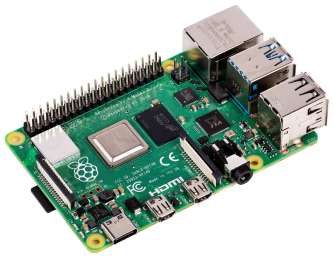


Figure 3.2: Raspberry Pi Model 4 B

BuildRoot

Buildroot will be used to create a custom Linux image for the Raspberry Pi's memory card. some of the most important packages used in the image are presented bellow:

`i2c-tools` package provides a collection of I2C tools for interacting with I2C devices from userspace.

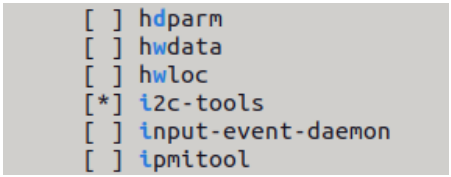


Figure 3.3: i2c-tools package

`dhcpcd` is a DHCP client daemon that manages the automatic IP address configuration of a device when connected to a network. `dropbear` is a lightweight SSH server and client implementation suitable for embedded environments.

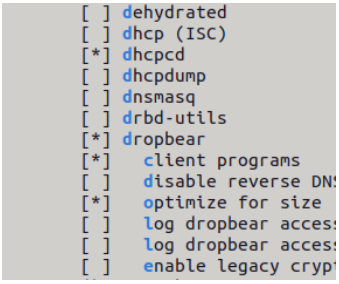


Figure 3.4: dhcpcd and dropbear packages

`openssh` is a more feature-rich and widely used SSH server and client suite compared to dropbear.

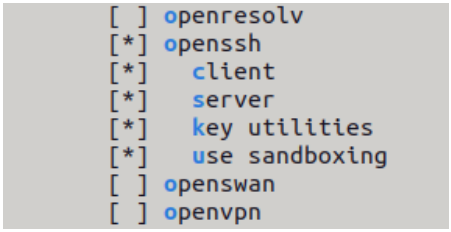


Figure 3.5: openssh package

3.1.3 Xilinx Zybo Z-10

To achieve optimal performance, the IMUs will be connected to a custom PCB directly interfacing with the FPGA PMODs, and UART communication will be established via a Micro USB cable. This setup is ideal for comparing the performance of the three different IMUs. Additionally, the FPGA will be utilized to implement programmable logic for data processing. This will be accomplished by creating a hardware-accelerated Kalman filter IP in Vitis HLS, using C as the high-level programming language.

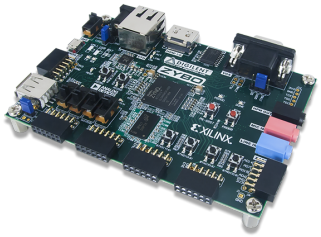


Figure 3.6: Xilinx Zybo Z-10

3.2 Custom PCB Design for IMU Integration

The integration of the Inertial Measurement Units onto a custom-designed PCB is a fundamental aspect of our project's implementation. This section outlines the detailed process involved in designing and fabricating the PCB, employing Altium (Figure 3.7) as our primary design software.

The PCB design process commenced with a comprehensive evaluation of the IMUs technical specifications. This step was crucial in understanding their operational parameters and integration requirements.

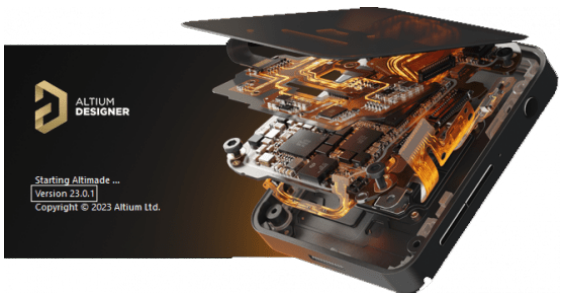


Figure 3.7: Altium

During the design phase, meticulous attention was paid to various aspects such as component placement, routing and traces, power distribution, and mechanical

considerations. Strategic placement of components was essential to minimize signal interference and optimize system performance.

Precision routing of traces was carried out to maintain signal integrity and minimize noise, adhering to industry best practices for high-speed digital and analog signals. Additionally, efficient power distribution was ensured to guarantee stable operation of the IMU and other components, enhancing overall reliability.

Furthermore, mechanical constraints were carefully considered to ensure compatibility with enclosure designs and mounting requirements.

3.2.1 ASM330LHB PCB

To start the PCB design, we first designed the circuit schematics. This initial step involved the meticulous connection of key components, including the IMU, two pull-up resistors essential for ensuring smooth I2C protocol communication, and power supply decoupling capacitors strategically placed in close proximity to the supply pin of the IMU Integrated Circuit.

Additionally, was incorporated one header pin connector with 12 pin to facilitate seamless integration of the PCB with the FPGA PMODs (Zybo Z7-10 FPGA has five headers with 12 female ports each). The following Schematic (fig.3.8) laid the groundwork for the subsequent stages of PCB layout design and fabrication.

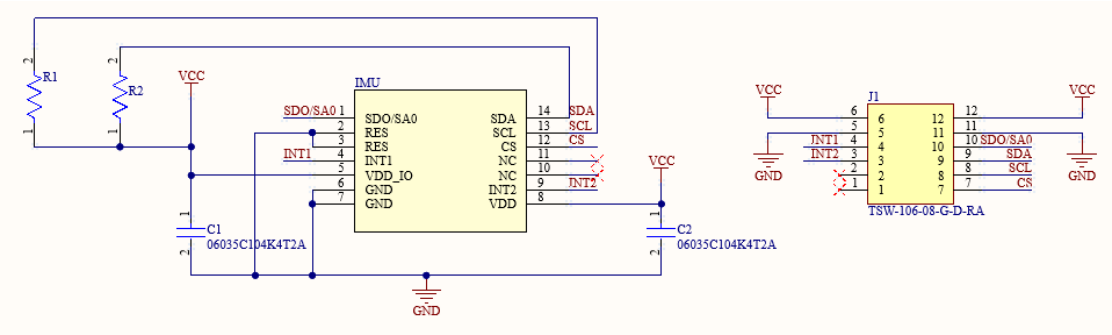


Figure 3.8: PCB Schematic

Following the circuit schematics, our attention turned towards PCB design, with a primary goal of optimizing component placement for minimal PCB size. We aimed for a compact and efficient layout to seamlessly integrate with Zybo Z7 PMods, reducing the risk of cable noise and ensuring optimal data transmission quality.

With a meticulous detail, we carefully assessed the generated component footprints and their spatial requirements. Leveraging our expertise in PCB design and layout, we made informed decisions regarding the positioning of each component

to ensure minimal wasted space and optimal utilization of the available area as can be seen in figure number 3.9 and 3.10.

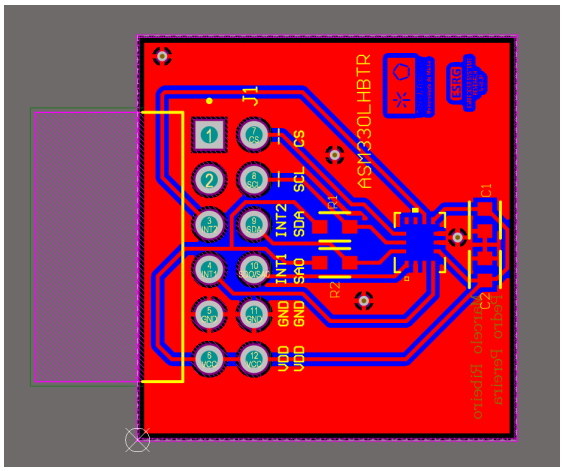


Figure 3.9: PCB Design 2D

The polished and finalized layout can be observed in Figure 3.10, presented in a three-dimensional perspective.

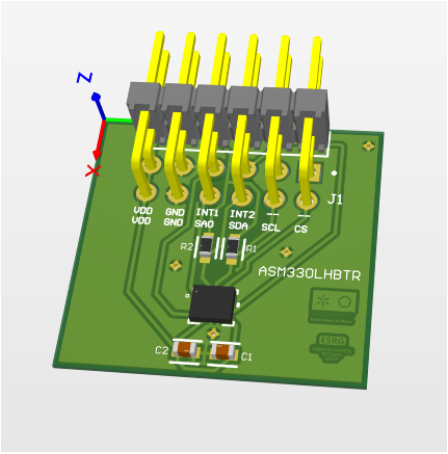


Figure 3.10: PCB Design 3D - ASM330LHBTR

3.2.2 BMI088 PCB

The same process was followed for the BMI088 IMU as follows in figure 3.12.

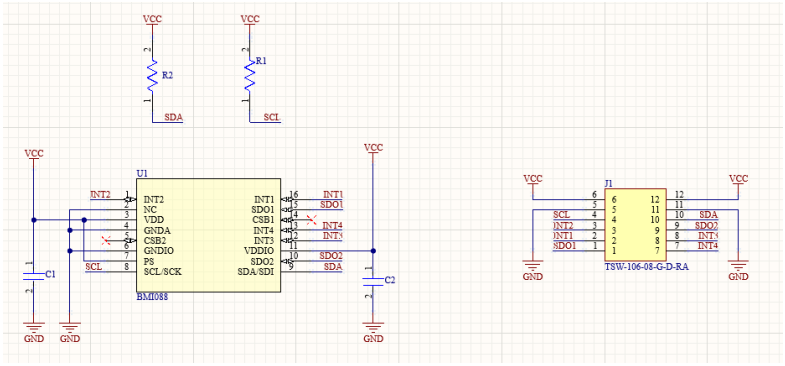
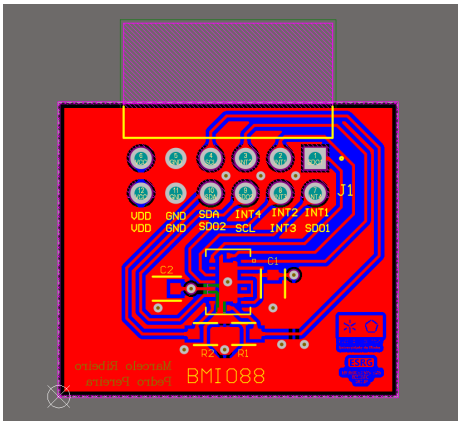
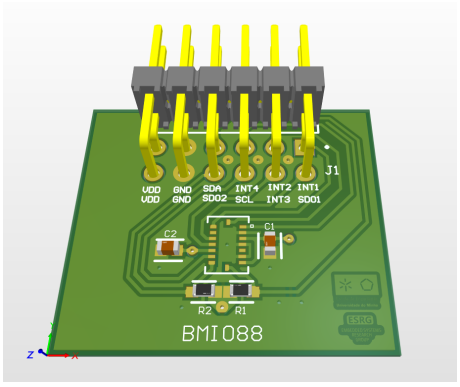


Figure 3.11: PCB Schematic - BMI088



((a)) PCB Design 2D - BMI088



((b)) PCB Design 3D - BMI088

Figure 3.12: BMI088 IMU PCB

3.2.3 BMI323 PCB

The PCB for the BMI323 was designed by Mário Mesquita, a Senior PCB Designer working for DTx, who walked us through the whole process of the PCBs design. Mário designed the BMI323 IMU PCB(that followed as a model for the others) as follows in figure 3.14.

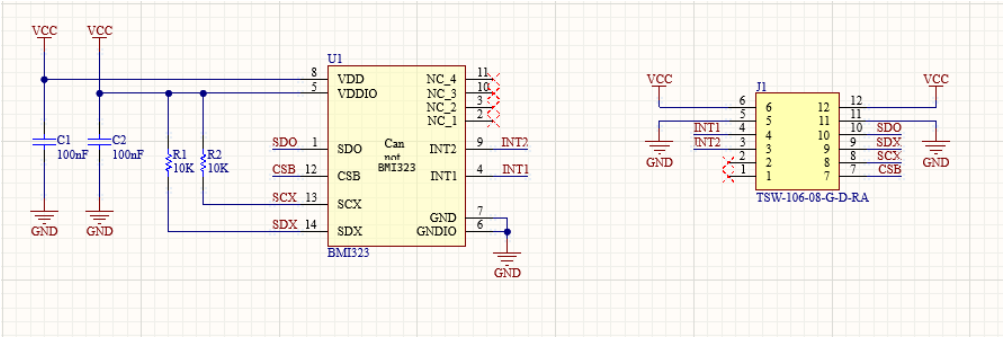
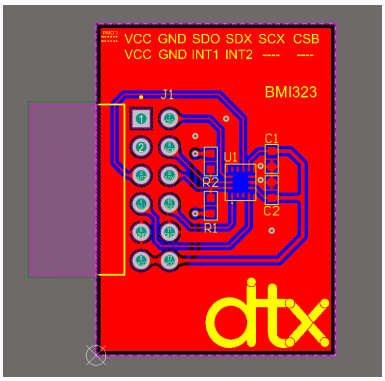
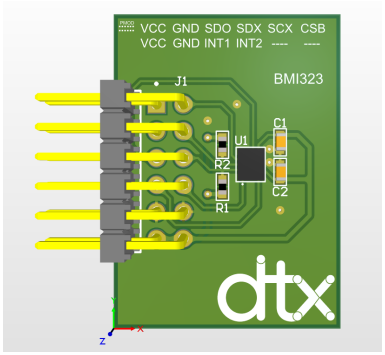


Figure 3.13: PCB Schematic - BMI323



((a)) PCB Design 2D - BMI323



((b)) PCB Design 3D - BMI323

Figure 3.14: BMI323 IMU PCB

3.3 Pinout planning: Pins Usage and Functions

3.3.1 STM32

In the STM board the IMU dev-kit was connected directly in the GPIO to avoid wire noise as follows in figure 3.15

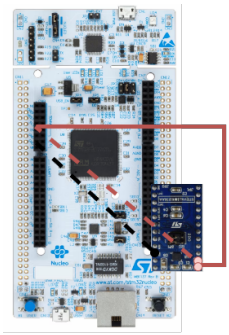


Figure 3.15: STM32 System Connections

3.3.2 Raspberry

The Raspberry system connections can be visualized in Figure 3.16.

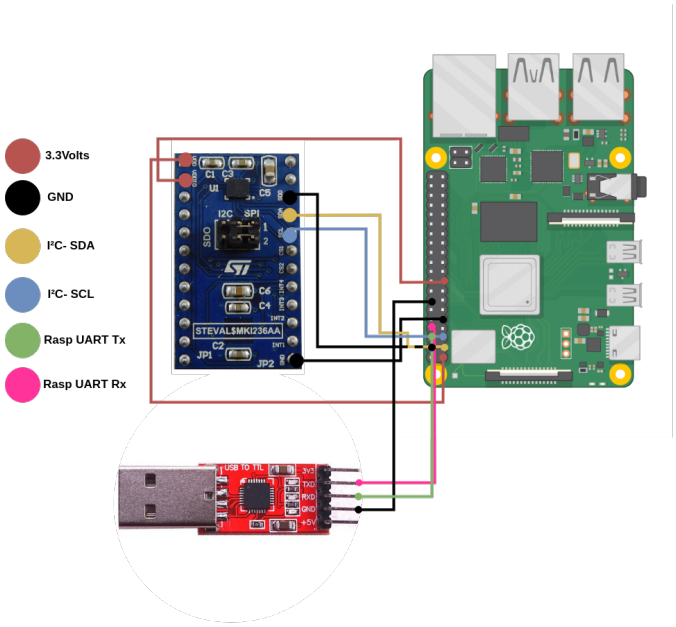


Figure 3.16: Raspberry System Connections

3.3.3 Zybo Z7-10

As mentioned before, the FPGA system won't have any volatile wires since the IMU PCBs were made, and the UART connection is established via the micro USB cable, which also powers the FPGA. This system can be visualized in Figure 3.16.

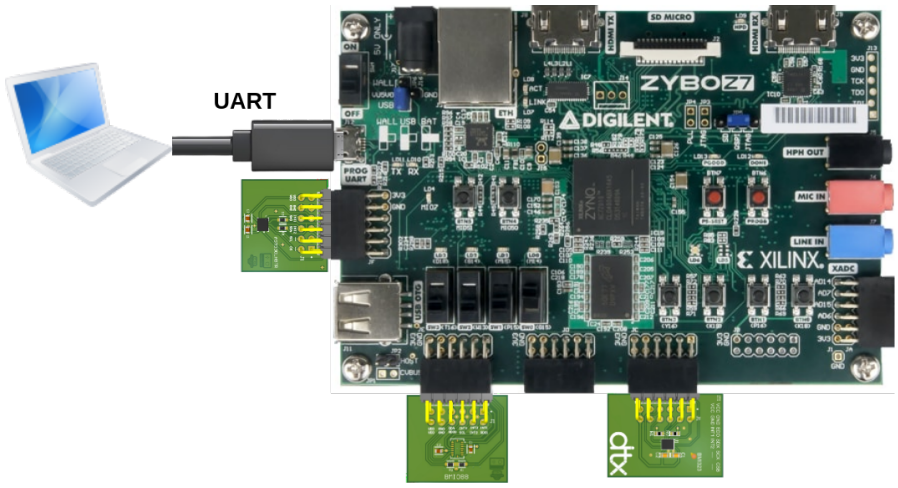


Figure 3.17: FPGA System Connections

3.4 Software Design

3.4.1 Raspberry System

The Raspberry system will be running a Linux image, generated with Buildroot. It will execute a C program with threads dealing with data processing, memory management, crash detection and UART transmission. A set of flowcharts describing the system's behaviour can be seen below. The general system flowchart is presented in Figure 3.18. This system is responsible for simulating a real use case scenario by reading and processing data from the IMU, where this data is continuously stored and updated. Additionally, the system includes an algorithm for detection of car crash patterns, triggering a mode that replays the car's behavior before and shortly after a crash.

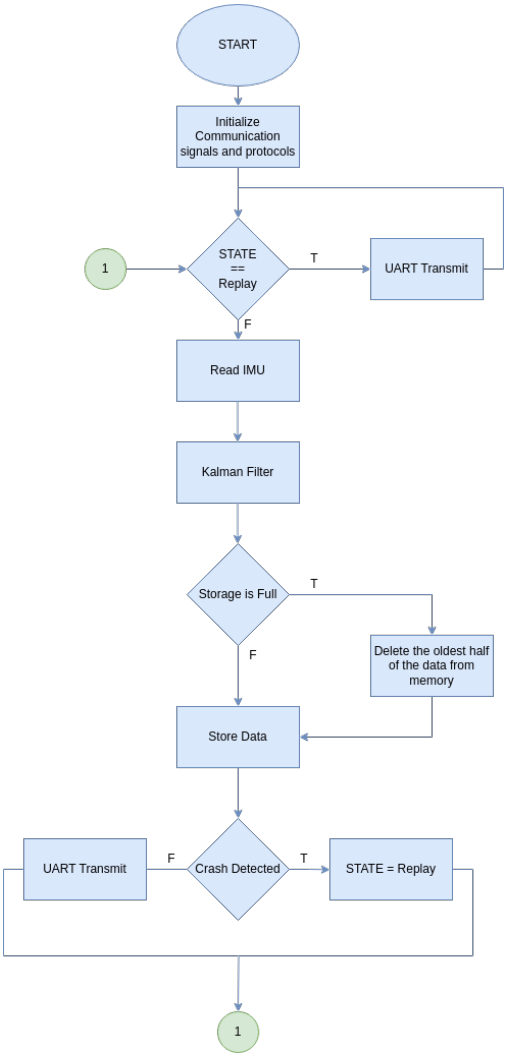


Figure 3.18: Raspberry System flowchart

3.4.2 Raspberry System Threads

IMU Data Processing Thread

This thread starts the IMU by configuring all the necessary registers following the users requirements and then enters in the infinite while loop where the IMU will be processed, only if the current operating mode is in `REAL_TIME`, i.e, not in `REPLAY` mode. This happens inside a critical zone defined by a Mutex to ensure that zone is not interrupted by others with the same Mutex token.

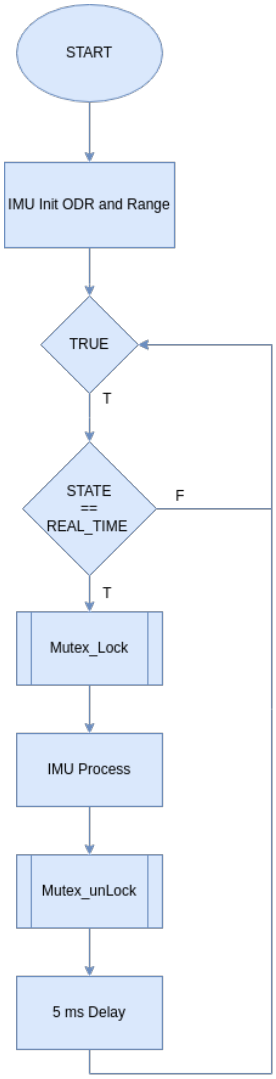


Figure 3.19: Data Processing Thread Flowchart

UART Transmission Thread

This thread is the one responsible for transmitting the needed data to the UART buffer. The first thing to be verified is the Mode.

If the mode is `REAL_TIME` the current data being read and processed from the sensor will be transmitted by UART in real time.

If the operating mode is `REPLAY` it means that a crash as happened before so the data to be transmitted will be the one stored on the raspberry Pi memory card. This replay will be repeated until the raspberry is restarted, although a

trigger could be implemented, in future work, to unlock this feature in real time processing.

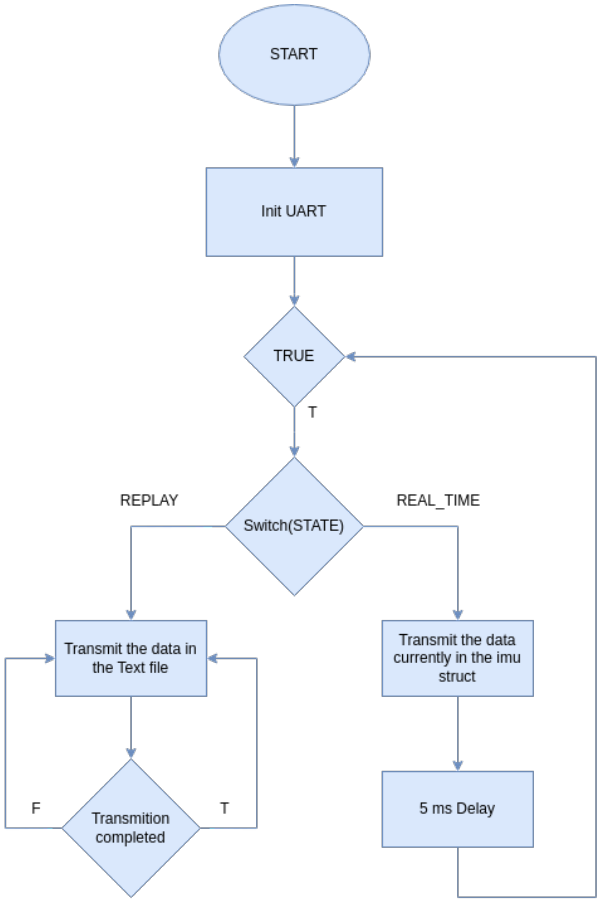


Figure 3.20: Memory Management Thread Flowchart

Memory management Thread

This thread, as the name implies, manages the storage of the Raspberry Pi microSD card. A folder named "IMU" was created with a file called "BlackBox.txt" inside it. Before entering the loop, the thread resets the text file because initializing the thread indicates that the Raspberry Pi has been reset, and the current data is not up-to-date.

Since it doesnt make sense to remove or write data to the memory while in`REPLAY` mode, this verification is done at the beginning of the while loop.

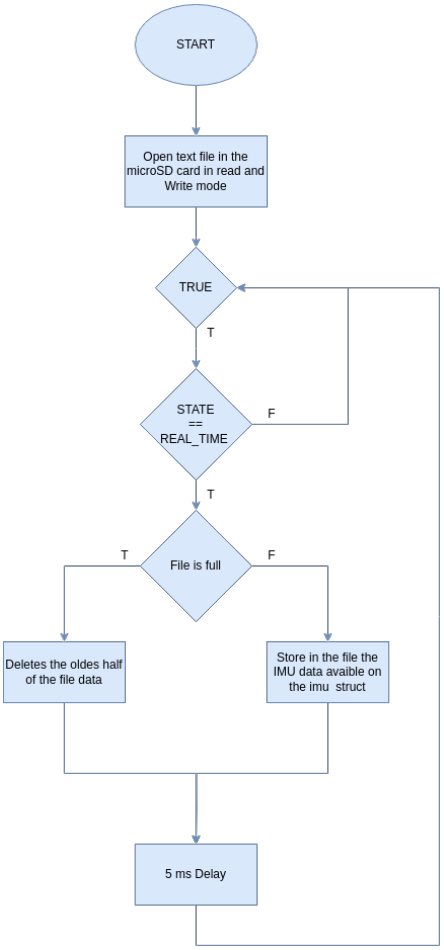


Figure 3.21: Memory Management Thread Flowchart

If the current mode is `REAL_TIME` it checks if the file is full to either continue writing to the file or clean half of the file and place the most recent data at the beginning of it.

$$15\text{min\_in\_Bytes} = \frac{\text{Random\_Num\_Bytes} \times (15 \times 60)}{\text{Seconds\_to\_get\_Random\_Num\_Bytes}} \tag{3.1}$$

In this thread it is necessary to define the maximum time of IMU data the text file will contain. The value chosen is 15 minutes of updated information. After testing how many seconds takes to reach a certain amount of data (in bytes) it is possible to find the linear equation to calculate the memory size that corresponds to 15 minutes of IMU data. The following formula exemplifies the calculations:

Crash detection Thread

The following thread contains a threshold-based car crash detection, which only makes sense to be active if the system is fetching data in real time. The chosen **safe-zone** thresholds, suitable for automotive-grade applications, are:

- Pitch: [-80, 80]degrees
- Roll : [-60, 60]degrees
- Accel: [-15, 15]m/s^2

The pitch angle range of -80 to +80 degrees allows the system to detect significant forward or backward tilts of the vehicle. This is crucial for identifying instances where the vehicle may be tilted abnormally due to a collision or deviation from the intended path. The roll angle range of -60 to +60 degrees enables detection of abnormal lateral tilts of the vehicle. This can indicate a side collision or a sudden change in vehicle direction.

The acceleration range of -15 to +15 m/sš is critical for detecting sudden changes in vehicle speed, such as those caused by collisions. Limiting the acceleration range helps distinguish between normal vehicle movements and events requiring immediate attention, such as abrupt stops or collisions.

If any of these condition are violated the `REAL_TIME` mode changes to `REPLAY` after 30 seconds, in order to visualize the Post-crash car movements, after that the replay time is calculated to inform the user.

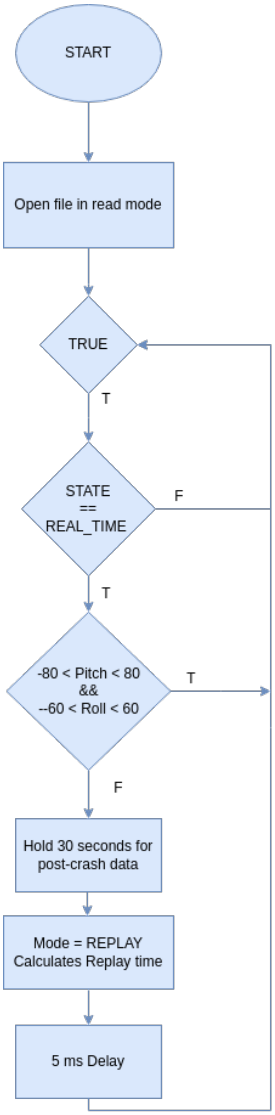


Figure 3.22: Crash detection Thread's Flowchart

3.4.3 STM32 and Zybo System

The STM32 system runs as a standalone system with only one IMU process happening, on the other hand, in Zybo system each IMU has its own dedicated thread within a FreeRTOS system. The software design that represents the stm system or one thread of the Zybo system is represented in figure **??**.

Initially, it initializes communication and the sensors of each IMU (accelerometer and gyroscope), and subsequently processes their data. In the `IMU_Process()` function, accelerometer and gyroscope data are read, followed by the application of Madgwick and Kalman filters. Finally, the processed data is transmitted via the UART terminal.

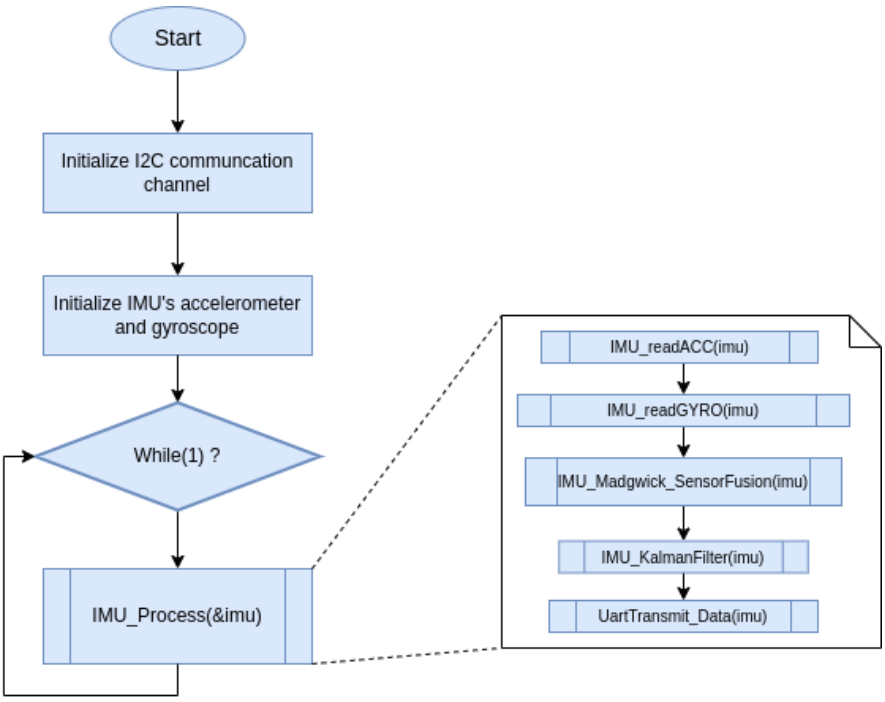


Figure 3.23: IMU processing

3.5 IMU Device Drivers

The device drivers are designed to provide a comprehensive interface for interacting with the sensor, since the IMUs are all very similar they can be designed the same way. The drivers ease the initialization, communication with the sensor through I2C, data acquisition and data Filtering. The following sections outline the critical components and functionality in the driver.

Initialization Functions

Initialization functions are going to be implemented to configure the I2C interface and ensure the sensor is ready for operation.

Data Acquisition Functions

Functions for reading data from the gyroscope, accelerometer and temperature sensor are also going to be implemented. These functions perform I2C read operations to retrieve raw sensor data and then process it into meaningful units, such as degrees per second for gyroscope data, meters per second squared for accelerometer data and celcius for temperature data.

Data Processing

The driver will include a function to process the acquired sensor data. This involves applying sensor fusion algorithms and Kalman filtering to provide more accurate and reliable measurements of roll, pitch, and yaw.

3.5.1 Kalman Filter

The solution proposed models a system with a set of n^{th}-order differential equations, converts them into an equivalent set of first-order differential equations, and puts them into the matrix form $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$. Once in this form, several techniques are used to convert these linear differential equations into the recursive equation $\mathbf{x}_t = \mathbf{F}\mathbf{x}_{t-1}$.

The Kalman filter has two steps:

The **prediction step** estimates the next state, and its covariance, at time t of the system given the previous state at time
t - 1.

The **correction step** rectifies the estimation with a set of measurements z at time t.

$$\mathbf{v}_t = \mathbf{z}_t - \mathbf{H}\hat{\mathbf{x}}_t \tag{3.2}$$

$$\mathbf{S}_t = \mathbf{H}\hat{\mathbf{P}}_t\mathbf{H}^T + \mathbf{R} \tag{3.3}$$

$$\mathbf{K}_t = \hat{\mathbf{P}}_t\mathbf{H}^T(\mathbf{S}_t)^{-1} \tag{3.4}$$

$$\mathbf{x}_t = \hat{\mathbf{x}}_t + \mathbf{K}_t\mathbf{v}_t \tag{3.5}$$

$$\mathbf{P}_t = \hat{\mathbf{P}}_t - \mathbf{K}_t\mathbf{S}_t\mathbf{K}_t^T \tag{3.6}$$

where

- $\hat{\mathbf{P}}_t \in \mathbb{R}^{n \times n}$ is the **Predicted Covariance** of the state before seeing the measurements $\mathbf{z}_t$.
- $\mathbf{P}_t \in \mathbb{R}^{n \times n}$ is the **Estimated Covariance** of the state after seeing the measurements $\mathbf{z}_t$.
- $\mathbf{u}_t \in \mathbb{R}^k$ is a **Control input vector** defining the expected behaviour of the system.
- $\mathbf{B} \in \mathbb{R}^{n \times k}$ is the **Control input model**.
- $\mathbf{H} \in \mathbb{R}^{m \times n}$ is the **Observation model** linking the predicted state and the measurements.
- $\mathbf{v}_t \in \mathbb{R}^m$ is the **Innovation** or **Measurement residual**.
- $\mathbf{S}_t \in \mathbb{R}^{m \times m}$ is the **Measurement Prediction Covariance**.
- $\mathbf{K}_t \in \mathbb{R}^{n \times m}$ is the filter gain, a.k.a. the **Kalman Gain**, telling how much the predictions should be corrected.

The predicted state $\hat{\mathbf{x}}_t$ is estimated in the first step based on the previously computed state $\mathbf{x}_{t-1}$, and later is corrected during the second step to obtain the final estimation $\mathbf{x}_t$. A similar computation happens with its covariance $\mathbf{P}$.

The loop starts with prior mean $\mathbf{x}_0$ and prior covariance $\mathbf{P}_0$, which are defined by the system model.

3.5.2 Madgwick Filter

Madgwick Filter, an orientation filter is applicable to IMUs consisting of tri-axial gyroscopes and accelerometers, and MARG arrays, which also include tri-axial magnetometers, proposed by Sebastian Madgwick.

The orientation of the Earth frame relative to the sensor frame at time $\mathbf{q}_{\omega,t} = \begin{bmatrix} q_w & q_x & q_y & q_z \end{bmatrix}$ can be computed by numerically integrating the quaternion derivative $\dot{\mathbf{q}}_t = \frac{1}{2}\mathbf{q}_{t-1}\omega_t$ as:

$$\begin{aligned} \mathbf{q}_{\omega,t} &= \mathbf{q}_{t-1} + \dot{\mathbf{q}}_{\omega,t}\Delta t \\ &= \mathbf{q}_{t-1} + \frac{1}{2}\left(\mathbf{q}_{t-1}{}^{\mathbf{S}}\omega_t\right)\Delta t \end{aligned} \tag{3.7}$$

where Δt is the sampling period and ${}^{S}\omega = \begin{bmatrix} 0 & \omega_x & \omega_y & \omega_z \end{bmatrix}$ is the tri-axial angular rate, in rad/s, measured in the sensor frame and represented as a pure quaternion.

Orientation as solution of Gradient Descent

A quaternion representation requires a complete solution to be found. This may be achieved through the formulation of an optimization problem where an orientation of the sensor, $\mathbf{q}$, is that which aligns any predefined reference in the Earth frame, ${}^{E}\mathbf{d} = \begin{bmatrix} 0 & d_x & d_y & d_z \end{bmatrix}$, with its corresponding measured direction in the sensor frame, ${}^{S}\mathbf{s} = \begin{bmatrix} 0 & s_x & s_y & s_z \end{bmatrix}$.

So, the objective function is:

$$\begin{aligned} f(\mathbf{q},\, {}^{E}\mathbf{d},\, {}^{S}\mathbf{s}) &= \mathbf{q}^{*}\, {}^{E}\mathbf{d}\, \mathbf{q} - {}^{S}\mathbf{s} \\ &= \begin{bmatrix} 2d_x\left(\frac{1}{2} - q_y^2 - q_z^2\right) + 2d_y(q_w q_z + q_x q_y) + 2d_z(q_x q_z - q_w q_y) - s_x \\ 2d_x(q_x q_y - q_w q_z) + 2d_y\left(\frac{1}{2} - q_x^2 - q_z^2\right) + 2d_z(q_w q_x + q_y q_z) - s_y \\ 2d_x(q_w q_y + q_x q_z) + 2d_y(q_y q_z - q_w q_x) + 2d_z\left(\frac{1}{2} - q_x^2 - q_y^2\right) - s_z \end{bmatrix} \end{aligned} \tag{3.8}$$

where $\mathbf{q}^{*}$ is the conjugate of $\mathbf{q}$.

From an initial guess $\mathbf{q}_0$ and a step-size μ , the GDA (Gradient Descent Algorithm) for n iterations, which estimates $n+1$ orientations, is described as:

$$\mathbf{q}_{k+1} = \mathbf{q}_k - \mu \frac{\nabla f(\mathbf{q}_k,\, {}^{E}\mathbf{d},\, {}^{S}\mathbf{s})}{\|\nabla f(\mathbf{q}_k,\, {}^{E}\mathbf{d},\, {}^{S}\mathbf{s})\|} \tag{3.9}$$

where $k = 0, 1, 2 \ldots n$, and the gradient of the solution is defined by the objective function and its Jacobian:

$$\nabla f(\mathbf{q}_k,\, {}^{E}\mathbf{d},\, {}^{S}\mathbf{s}) = J(\mathbf{q}_k,\, {}^{E}\mathbf{d})^{T} f(\mathbf{q}_k,\, {}^{E}\mathbf{d},\, {}^{S}\mathbf{s}) \tag{3.10}$$

The **Jacobian** of the objective function is:

$$\begin{aligned} J(\mathbf{q}_k,\, {}^{E}\mathbf{d}) &= \begin{bmatrix} \dfrac{\partial f(\mathbf{q}, {}^{E}\mathbf{d}, {}^{S}\mathbf{s})}{\partial q_w} & \dfrac{\partial f(\mathbf{q}, {}^{E}\mathbf{d}, {}^{S}\mathbf{s})}{\partial q_x} & \dfrac{\partial f(\mathbf{q}, {}^{E}\mathbf{d}, {}^{S}\mathbf{s})}{\partial q_y} & \dfrac{\partial f(\mathbf{q}, {}^{E}\mathbf{d}, {}^{S}\mathbf{s})}{\partial q_z} \end{bmatrix} \\ &= \begin{bmatrix} 2d_y q_z - 2d_z q_y & 2d_y q_y + 2d_z q_z & -4d_x q_y + 2d_y q_x - 2d_z q_w & -4d_x q_z + 2d_y q_w + 2d_z q_x \\ -2d_x q_z + 2d_z q_x & 2d_x q_y - 4d_y q_x + 2d_z q_w & 2d_x q_x + 2d_z q_z & -2d_x q_w - 4d_y q_z + 2d_z q_y \\ 2d_x q_y - 2d_y q_x & 2d_x q_z - 2d_y q_w - 4d_z q_x & 2d_x q_w + 2d_y q_z - 4d_z q_y & 2d_x q_x + 2d_y q_y \end{bmatrix} \end{aligned} \tag{3.11}$$

This general form of the algorithm can be applied to a field predefined in any direction, as it will be shown for the IMU.

3.5.3 Yaw Angle Drift

In the design phase, it's crucial to consider the limitations of using an IMU without a magnetometer. These IMUs struggle with maintaining accurate yaw (rotation around the Z-axis) over time. This is because gyroscopes, the primary sensors for yaw rate, have inherent limitations. They can have small built-in errors, pick up electrical noise, and become slightly more or less sensitive depending on temperature. These factors all contribute to "yaw drift," where the estimated yaw angle deviates from the true value.

The severity of yaw drift can impact this project's performance, this way, the yaw angle will not be used in order to have a reliable system. Although tests will be done in the verification phase demonstrating how yaw behaves when integrating gyroscope z-axis angular rate over time.

3.5.4 OpenGL 3D Scenario

For the IMU real-time or replay 3D visualization, an OpenGL scenario will be built using C++. This program will use the "GLUT" interface to utilize the OpenGL API and the <thread> API to create a thread-based program that renders the scenario and receives data from the UART serial port concurrently. The IMU or, in this scenario, the cube orientation and the cube space tracking will be implemented in a three-dimensional simulated space, as represented in Figure 3.24.

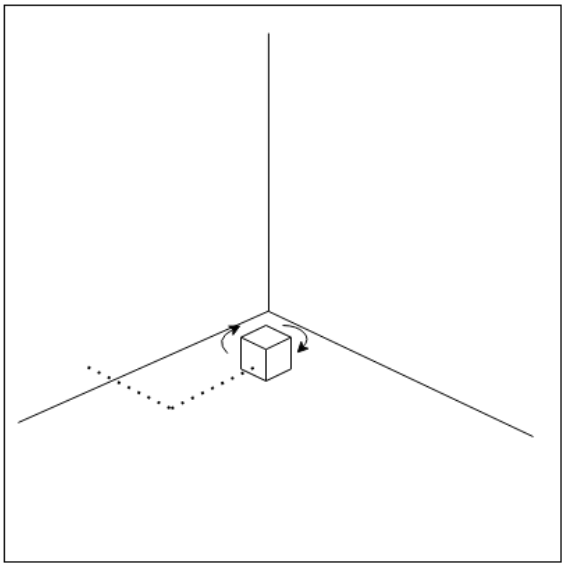


Figure 3.24: OpenGL Scenario Design

Chapter 4

Implementation

4.1 Kalman Filter

Three Kalman filter functions (Kalman_Roll, Kalman_Pitch, and Kalman_Yaw) are implemented to estimate the orientation angles (roll, pitch, and yaw) using sensor data from an IMU. Each function follows a two-step process: prediction and correction.

Initially, predictions are made for the state estimate (Xapriori) and error covariance (Papriori) based on predefined parameters (A, B, u, Q, R, H, I) and previous state values (Xant_, Pant_). Subsequently, corrections are applied using measurement updates (imu->roll, imu->pitch, imu->yaw) to refine the estimates (X) and update the error covariance (P).

The KalmanFilter function sequentially calls these processes for all three angles. Its goal is to provide smoothed and accurate orientation outputs by minimizing sensor noise and errors over time.

```
1  static const float A = 1;
2  static const float B = 0;
3  static const float C = 0;
4  static const float D = 0;
5  static const float u = 0;
6  static const float H = 1;
7  static const float Q = 0.005;
8  static const float R = 0.1;
9  static const float I = 1;
10
11
12 static void Kalman_Roll(IMU *imu){
13     //Prediction
14     float Xapriori = A * imu->Xant_Roll + B * u;
15     float Papriori = A * imu->Pant_Roll * A + Q;
16     //Correction
17     float K = Papriori * H / (H * Papriori * H + R);
18     float X = Xapriori + K * (imu->roll - H * Xapriori);
19     float P = (I - K * H) * Papriori;
20     imu->Pant_Roll = P;
21     imu->Xant_Roll = X;
22
23     imu->roll = imu->Xant_Roll;
24 }
25
26 static void Kalman_Pitch(IMU *imu){
27     //Prediction
```

```
28      float Xapriori = A * imu->Xant_Pitch + B * u;
29      float Papriori = A * imu->Pant_Pitch * A + Q;
30      //Correction
31      float K = Papriori * H / (H * Papriori * H + R);
32      float X = Xapriori + K * (imu->pitch - H * Xapriori);
33      float P = (I - K * H) * Papriori;
34      imu->Pant_Pitch = P;
35      imu->Xant_Pitch = X;
36
37      imu->pitch = imu->Xant_Pitch;
38  }
39
40  static void Kalman_Yaw(IMU *imu){
41      //Prediction
42      float Xapriori = A * imu->Xant_Yaw + B * u;
43      float Papriori = A * imu->Pant_Yaw * A + Q;
44      //Correction
45      float K = Papriori * H / (H * Papriori * H + R);
46      float X = Xapriori + K * (imu->yaw - H * Xapriori);
47      float P = (I - K * H) * Papriori;
48      imu->Pant_Yaw = P;
49      imu->Xant_Yaw = X;
50
51      imu->yaw = imu->Xant_Yaw;
52  }
53
54  void KalmanFilter(IMU_STRUCTURE *imu){
55      Kalman_Pitch(imu);
56      Kalman_Roll(imu);
57      Kalman_Yaw(imu);
58  }
59  }
```

Listing 1: Implementation of Kalman filter functions for estimating roll, pitch, and yaw angles.

4.2 Madgwick Filter

The Madgwick filter was implemented following the formulas presented in the design phase. First, it reads accelerometer and gyroscope data from their respective sensors, normalizes the accelerometer data, computes the corrective step for the gradient descent algorithm, and integrates the rate of change of the quaternion over time.
Subsequently, it calculates the Roll and Pitch, and optionally Yaw by integrating the gyroscopes z-axis data (yaw can experience significant drift over time without

a magnetometer).
It's important to note that since the Kalman filter will be applied to Euler angles, the Madgwick filter does not inherently provide continuous values. For example, starting with a base roll value of 180 degrees on a flat surface, rotating left reduces it from -180 to 0 degrees, while rotating right reduces it from 180 to 0 degrees. This inconsistency makes it unreliable to use with the Kalman filter. Therefore, it's necessary to unwrap the roll value, setting 0 degrees as the base roll value and allowing it to decrease to -180 or increase to 180 degrees when rotating left or right, respectively.

```
1  void Madgwick_SensorFusion(IMU *imu){
2      // Updates the variables with the IMU data
3      float ax = imu->acc_ms2[0], ay = imu->acc_ms2[1], az = imu->acc_ms2[2], gx
   ↪   = imu->gyro_dps[0] *PI/180.0f, gy= imu->gyro_dps[1]*PI/180.0f,gz =
   ↪   imu->gyro_dps[2]*PI/180.0f;
4      static float q[4];
5      float q1 = q[0], q2 = q[1], q3 = q[2], q4 = q[3];   // short name local
   ↪   variable for readability
6      float norm;
7      float s1, s2, s3, s4;
8      float qDot1, qDot2, qDot3, qDot4;
9
10     float GyroMeasError = PI * (40.0f / 180.0f);        // gyroscope
   ↪   measurement error in rads/s (start at 40 deg/s)
11     float beta = sqrt(3.0f / 4.0f) * GyroMeasError;     // compute beta
12
13     // Auxiliary variables to avoid repeated arithmetic
14     float _2q1 = 2.0f * q1;
15     float _2q2 = 2.0f * q2;
16     float _2q3 = 2.0f * q3;
17     float _2q4 = 2.0f * q4;
18     float _4q1 = 4.0f * q1;
19     float _4q2 = 4.0f * q2;
20     float _4q3 = 4.0f * q3;
21     float _8q2 = 8.0f * q2;
22     float _8q3 = 8.0f * q3;
23     float q1q1 = q1 * q1;
24     float q2q2 = q2 * q2;
25     float q3q3 = q3 * q3;
26     float q4q4 = q4 * q4;
27
28     // Normalise accelerometer measurement
29     norm = sqrt(ax * ax + ay * ay + az * az);
30     if (norm == 0.0f) return;                                    // handle NaN
31     norm = 1.0f / norm;
32     ax *= norm;
33     ay *= norm;
```

```
34      az *= norm;
35
36      // Gradient decent algorithm corrective step
37      s1 = _4q1 * q3q3 + _2q3 * ax + _4q1 * q2q2 - _2q2 * ay;
38      s2 = _4q2 * q4q4 - _2q4 * ax + 4.0f * q1q1 * q2 - _2q1 * ay - _4q2 + _8q2
    ↪   * q2q2 + _8q2 * q3q3 + 4.0f * az;
39      s3 = 4.0f * q1q1 * q3 + _2q1 * ax + 4.0f * q3 * q4q4 - _2q4 * ay - _4q3 +
    ↪   _8q3 * q2q2 + _8q3 * q3q3;
40      s4 = 4.0f * q2q2 * q4 - _2q2 * ax + 4.0f * q3q3 * q4 - _2q3 * ay;
41      norm = sqrt(s1 * s1 + s2 * s2 + s3 * s3 + s4 * s4);         // normalise
    ↪   step magnitude
42      norm = 1.0f / norm;
43      s1 *= norm;
44      s2 *= norm;
45      s3 *= norm;
46      s4 *= norm;
47
48      // Compute rate of change of quaternion
49      qDot1 = 0.5f * (-q2 * gx - q3 * gy - q4 * gz) - beta * s1;
50      qDot2 = 0.5f * (q1 * gx + q3 * gz - q4 * gy) - beta * s2;
51      qDot3 = 0.5f * (q1 * gy - q2 * gz + q4 * gx) - beta * s3;
52      qDot4 = 0.5f * (q1 * gz + q2 * gy - q3 * gx) - beta * s4;
53
54      float dt = 0.05;
55
56      q1 += qDot1 * dt;
57      q2 += qDot2 * dt;
58      q3 += qDot3 * dt;
59      q4 += qDot4 * dt;
60      norm = sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4);         // normalise
    ↪   quaternion
61      norm = 1.0f / norm;
62      q[0] = q1 * norm;
63      q[1] = q2 * norm;
64      q[2] = q3 * norm;
65      q[3] = q4 * norm;
66
67      static float yaw = 0;
68      //yaw = ( yaw + (imu->gyro_dps[2] * dt*2.40)); //Integrate over time (%240
    ↪   time offset due to drift)
69      imu->yaw = yaw;
70      imu->pitch  = -asin(2.0f * (q[1] * q[3] - q[0] * q[2])) * RAD_TO_DEG;
71      float roll_raw  = atan2(2.0f * (q[0] * q[1] + q[2] * q[3]), q[0] * q[0] -
    ↪   q[1] * q[1] - q[2] * q[2] + q[3] * q[3]) * RAD_TO_DEG;
72
73
74      // Unwrapping Roll Value (default: [-180, 180] Unwrapped: []
75      float delta_roll = roll_raw - previous_roll;
```

```
76      if (delta_roll > 180.0f) {
77          roll_raw -= 360.0f;
78      } else if (delta_roll < -180.0f) {
79          roll_raw += 360.0f;
80      }
81
82      imu->roll = roll_raw -180;
83      previous_roll = roll_raw;
84
85      return;
86  }
```

Listing 2: Implementation of the Madgwick sensor fusion algorithm for estimating orientation angles (roll, pitch, and yaw) using accelerometer and gyroscope data

4.3 IMU Device Drivers

Three device drivers were developed to compare them with the three. Since the FPGA system had this purpose, the generic device driver implemented with the Zynq drivers' APIs will be demonstrated.
However, the complete implementations of all three IMUs, along with the rest of the project content, will be annexed
The implemented `IMU` data structure is used to temporarily store and manage data and configurations for the IMU sensor. Each field serves a specific purpose, whether it's for holding raw data, processed data, or configuration settings.
Note: The "Status" variable was created to handle errors in run time.

```
1   typedef struct imuStruct
2   {
3       XIicPs_Config *Config;
4
5       float gyroRange;
6       float accRange;
7
8       float temp_celsius;
9       volatile float gyro_dps[3];
10      volatile float acc_ms2[3];
11
12      /* KALMAN FILTER */
13      volatile float Xant_Roll, Xant_Pitch, Xant_Yaw;
14      volatile float Pant_Roll, Pant_Pitch, Pant_Yaw;
15
16      volatile float roll,pitch,yaw;
17
18  }IMU;
```

Listing 3: IMU Structure

Initialization Functions

This function initializes I2C communication for the IMU sensor. It configures the I2C interface, conducts a self-test, and sets the I2C clock rate.

```
int IMU_Init_default(IMU *imu) {
    int Status;

    printf("Initializing I2C...\n");
    imu->Config = XIicPs_LookupConfig(I2C_DEVICE_ID);
    if (imu->Config == NULL) {
        printf("Erro finding I2C config !\n");
        return XST_FAILURE;
    }

    Status = XIicPs_CfgInitialize(&Iic_Instance, imu->Config,
    ↪ I2C_01_BASE_ADDR);
    if (Status != XST_SUCCESS) {
        printf("Error initializing I2C !\n");
        return XST_FAILURE;
    }

    Status = XIicPs_SelfTest(&Iic_Instance);
    if (Status != XST_SUCCESS) {
        printf("Error in I2C self test !\n");
        return XST_FAILURE;
    }

    Status = XIicPs_SetSClk(&Iic_Instance, I2C_CLK_RATE);  //I2C CLOCK
    if (Status != XST_SUCCESS) {
        printf("Error defining I2C clock !\n");
        return XST_FAILURE;
    }

    usleep(30000);
    printf("I2C initialized with success !\n");

    return XST_SUCCESS;
}
```

Listing 4: Initialization function for IMU with I2C configuration and self-test

In order to access the IMU internal registers, two helper functions were implemented to facilitate communication via I2C. These functions are designed as follows:

```
int IMU_readReg(IMU *imu, u16 Address, u8 Reg, u8 *Buffer, s32 ByteCount) {
    int Status;

    Status = XIicPs_MasterSendPolled(&Iic_Instance, &Reg, 1, Address);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    Status = XIicPs_MasterRecvPolled(&Iic_Instance, Buffer, ByteCount,
    ↪ Address);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}

int IMU_writeReg(IMU *imu, u16 Address, u8 Reg, u8 *Data, s32 ByteCount) {
    int Status;
    u8 SendBuffer[64];

    SendBuffer[0] = Reg;
    for (int i = 0; i < ByteCount; i++) {
        SendBuffer[i + 1] = Data[i];
    }

    Status = XIicPs_MasterSendPolled(&Iic_Instance, SendBuffer, ByteCount + 1,
    ↪ Address);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}
```

Listing 5: Initialization function for IMU with I2C configuration and self-test

After implementing the previous helper functions, the read function was used to access and read the output registers of the temperature sensor, accelerometer, and gyroscope in a single operation.
The raw ADC data from gyroscopes and accelerometers is then transformed into meaningful units for angular velocity (gyro_dps) and acceleration (acc_ms2). In the code below, transformation matrices (tX, tY, tZ) are used to convert sensor-specific readings into standard coordinate axes. These matrices adjust for the physical orientation of the sensors.

```
static const int16_t tX[3] = {1, 0, 0};
static const int16_t tY[3] = {0, -1, 0};
static const int16_t tZ[3] = {0, 0, -1};
```

Listing 6: Convertion from sensor frame to right hand coordinate system

Each component of the raw data is scaled by a factor that considers the sensor's resolution (LSB) and the desired measurement range (`gyroRange` for gyroscope and `accRange` for accelerometer), ensuring accurate representation in degrees per second for gyroscope data and meters per second squared for accelerometer data.

```
static u8 ASM330LHB_readData(ASM330LHB *imu){
    u8 regData[14];
    u8 Status = ASM330LHB_readReg(imu, ASM330LHB_SLAVE_ADDR, TEMP_REG,
    ↪ regData,14);
    if(Status == XST_FAILURE)
        return ASM330LHB_error("Error Reading I2C");

    /*
    * The value is expressed as a 16-bit word in twos complement so a signed
    ↪ array in needed
    */
    int16_t tempRaw = ((int16_t)regData[1] << 8 | (int16_t)regData[0]);
    int16_t gyroRaw[3];
    gyroRaw[0] =((int16_t) regData[3] << 8 | (int16_t) regData[2]); /*X-AXIS*/
    gyroRaw[1] =((int16_t) regData[5] << 8 | (int16_t) regData[4]); /*Y-AXIS*/
    gyroRaw[2] =((int16_t) regData[7] << 8 | (int16_t) regData[6]); /*Z-AXIS*/
    int16_t accRaw[3];
    accRaw[0] =((int16_t) regData[9] << 8  | (int16_t) regData[8]); /*X-AXIS*/
    accRaw[1] =((int16_t) regData[11] << 8 | (int16_t) regData[10]); /*Y-AXIS*/
    accRaw[2] =((int16_t) regData[13] << 8 | (int16_t) regData[12]); /*Z-AXIS*/

    /*
     * CONVERT RAW TO Celcius
     * 1st We take of the offset from the raw data - 0 LSB -> 25 Celcius
     * 2nd We multiply by the sensitivity which is 1/ (LSB/Celcius) (page 12
    ↪ Datasheet)
     */
    imu->temp_celsius = 0.00390625f *( (float) tempRaw - 0) + 25.0f;

    /*
     * RAW TO dps(angle / s) CONVERTION
     * Configuration is at 500dps and we have 16-1 bits of data(since we lose
    ↪ 1 bit to the signal).
     * Sensitivity calculus(dps / LSB)
     * Sense = (500/ 2^15) = 0,015... -> 1/0,015 = 65.53(65.5360000107)
     */

```

```
34      imu->gyro_dps[0] = (float) (gyroRaw[0] * tX[0] + gyroRaw[1] * tX[1] +
    ↪  gyroRaw[2] * tX[2])  / LSB * (float)imu->gyroRange;
35      imu->gyro_dps[1] = (float) (gyroRaw[0] * tY[0] + gyroRaw[1] * tY[1] +
    ↪  gyroRaw[2] * tY[2])  / LSB * (float)imu->gyroRange;
36      imu->gyro_dps[2] = (float) (gyroRaw[0] * tZ[0] + gyroRaw[1] * tZ[1] +
    ↪  gyroRaw[2] * tZ[2])  / LSB * (float)imu->gyroRange;
37
38      /**
39       * RAW TO m/ss CONVERTION
40       * Configuration is at +/-4.096 g and we have 16-1 bits of data(since we
    ↪  lose 1 bit to the signal).
41       * Sensitivity calculus(m/ss / LSB) is the m/ss we have per unit
42       * Sense = (4.096*9.81 / 2^15)
43       */
44      imu->acc_ms2[0] = (float) (accRaw[0] * tX[0] + accRaw[1] * tX[1] +
    ↪  accRaw[2] * tX[2]) * (imu->accRange / LSB) * G_ACCELERATION;
45      imu->acc_ms2[1] = (float) (accRaw[0] * tY[0] + accRaw[1] * tY[1] +
    ↪  accRaw[2] * tY[2]) * (imu->accRange / LSB) * G_ACCELERATION;
46      imu->acc_ms2[2] = (float) (accRaw[0] * tZ[0] + accRaw[1] * tZ[1] +
    ↪  accRaw[2] * tZ[2]) * (imu->accRange / LSB) * G_ACCELERATION;
47
48      usleep(10000);
49          return Status;
50  }
```

Listing 7: Function to read and process sensor data from the IMU, including temperature, gyroscope, and accelerometer measurements

To resume all the init functions was created one init function that receives the IMU's configurations as arguments, storing the configuration in the ASM330LHB struct members as follows:

```
1   u8 ASM330LHB_Init_config(IMU *imu, u8 GYRO_ODR, u8 ACC_ODR, u8 GYRO_RANGE, u8
    ↪  ACC_RANGE){
2       if(ASM330LHB_Init_default(imu) == XST_FAILURE)
3           return ASM330LHB_error("Error Initializing I2C");
4
5       /* ACCELEROMETER CONFIGURATION */
6       switch (ACC_RANGE){
7           case ACC_RANGE_2_G:          imu->accRange = 2.048f;  break;
8           case ACC_RANGE_4_G:          imu->accRange = 4.096f;  break;
9           case ACC_RANGE_8_G:          imu->accRange = 8.192f;  break;
10          case ACC_RANGE_16_G:         imu->accRange = 16.384f; break;
11      }
12      u8 data = (ACC_ODR<<4 | ACC_RANGE<<2);
13      if(IMU_writeReg(imu, IMU_SLAVE_ADDR, CTRL1_XL, &data, 1) == XST_FAILURE)
```

```
14          return ASM330LHB_error("Error in IMU Accelerometer configuration.");
15
16      /* GYROSCOPE CONFIGURATION */
17      switch (GYRO_RANGE){
18          case GYRO_RANGE_125_DPS:        imu->gyroRange = 125;           data
    ↪   = (GYRO_ODR «4 | 0b00«2              | 0b1 « 1 | 0);        //activates
    ↪   125 dps special pin
19          break;
20          case GYRO_RANGE_250_DPS:        imu->gyroRange = 250;           data
    ↪   = (GYRO_ODR «4 | GYRO_RANGE«2 | 0b00 « 2);
21          break;
22          case GYRO_RANGE_500_DPS:        imu->gyroRange = 500;           data
    ↪   = (GYRO_ODR «4 | GYRO_RANGE«2 | 0b00 « 2);
23          break;
24          case GYRO_RANGE_1000_DPS:        imu->gyroRange = 1000;
    ↪   data = (GYRO_ODR «4 | GYRO_RANGE«2 | 0b00 « 2);
25          break;
26          case GYRO_RANGE_2000_DPS:         imu->gyroRange = 2000;
    ↪   data = (GYRO_ODR «4 | GYRO_RANGE«2 | 0b00 « 2);
27          break;
28          case GYRO_RANGE_4000_DPS:        imu->gyroRange = 4000;
    ↪   data = (GYRO_ODR «4 | 0b00«2              | 0b0 « 1 |
    ↪   1);        //activates 4000 dps special pin
29          break;
30      }
31      if(IMU_writeReg(imu, IMU_SLAVE_ADDR, CTRL2_G, &data, 1) == XST_FAILURE)
32          return IMU_error("Error in IMU Gyroscope configuration.");
33
34      return 0;
35  }
```

Listing 8: Function to initialize and configure the IMU with specified settings for gyroscope and accelerometer output data rates (ODR) and ranges

Finally, the function that runs in the loop to perform all the steps mentioned before was implemented. It starts by reading from the IMU internal registers and converting that data into meaningful values. Then, it fuses the accelerometer and gyroscope data using the Madgwick filter, applies the Kalman filter to the Euler angles (Roll, Pitch, Yaw), and finally transmits the final orientation through UART.

```
1   u8 IMU_Process(IMU *imu){
2       if (IMU_readData(imu) == XST_FAILURE)
3           return IMU_error("Error in data reading.");
4       IMU_Madgwick_SensorFusion(imu);
5
```

```
6       IMU_KalmanFilter(imu);
7
8       UartTransmit_Data(imu);
9
10      return XST_SUCCESS;
11  }
```

Listing 9: Function to process the IMU data by reading sensor data, performing sensor fusion using the Madgwick algorithm, applying Kalman filtering, and transmitting the processed data via UART

4.4 STM32 System

As mentioned before, the STM32 was used to do the initial drivers and testing, this way the read is made by pooling, not needing a RTOS for this effect.

```
1   while (1)
2     {
3       /* USER CODE END WHILE */
4
5       /* USER CODE BEGIN 3 */
6             if(Status == HAL_OK){
7                     IMU_dataProcess(&imu);
8             }
9
10    }
11    /* USER CODE END 3 */
12  }
```

:

To establish input and output communication (I2C and UART), the following pins were configured as follows:

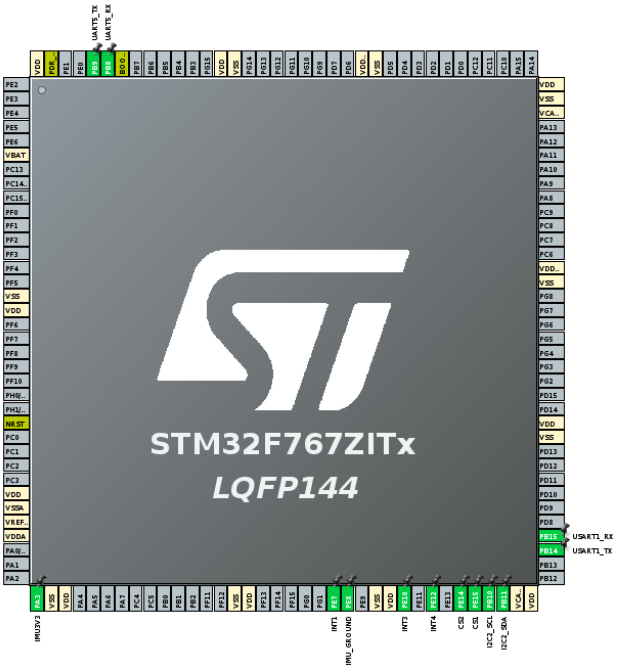


Figure 4.1: STM32 configuration

To avoid more wires, the sensor was connected directly to the board reducing the noise and increasing reliability as can be seen in Figure 4.2.

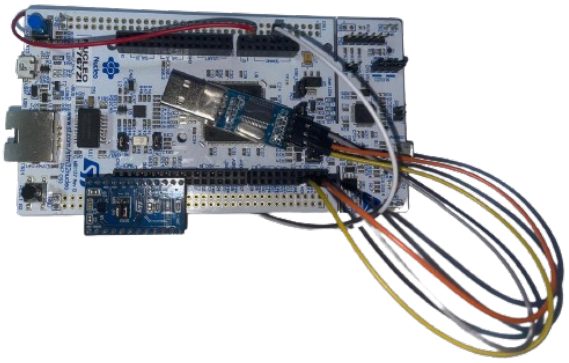


Figure 4.2: STM32 Board Connections

4.5 Raspberry Real-Time System

As related before, the Raspberry Pi will be used to emulate a real use case scenario such as an Event Data Record in a real vehicle. This way, it is crucial on the raspberry run a Real-Time Operating System with multiple threads running concurrently. For this effect was used the <pthread.h> API and four threads were implemented and will be shown below.
First of all was created a main header in order to organize the code as follows:

```
#pragma once

#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include "IMU/IMU.h"

#define STORAGE_PATH    "/IMU/BlackBox.txt"
#define SAMPLE_TIME     0.00518f     //This time is the real time mesured between samples in seconds (5.18ms = 66bytes tranfered)
#define SIZE_TEST       127413       //Bytes ->Corresponds to 10 seconds of data (Replace by size_test and verify the file clean at 10 seconds)
#define MAX_SIZE        16000000     //Bytes -> 16MB of space corresponding to 21min of IMU data
#define NUM_THREADS     4
#define MIN_REPLAY_TIME 5            //Replays must have at least <MIN_REPLAY_TIME> seconds

#define MODE_INIT REAL_TIME  //REPLAY   OR   REAL_TIME

static float threshold_positive = 15.0;  // Threshold de aceleração positiva em m/s^2

typedef enum{
    REAL_TIME,
    REPLAY
}operatingMode;

typedef struct{
    operatingMode mode;
    IMU imu;
    pthread_mutex_t mutex;
}IMUData;
```

Listing 10: Header file defining constants, includes, and structures for IMU data handling and multi-threaded operations.

4.5.1 readIMU()

In this section, we describe the implementation of a thread responsible for reading and processing data from the IMU. The following code snippet demonstrates the initialization and real-time processing of IMU data. The function readIMU runs in a separate thread, continuously reading IMU data when the system is in real-time mode. The IMU is initialized with specific configurations for the gyroscope and accelerometer, and data is processed in a thread-safe manner using mutex locks.

```
static void *readIMU(void *arg){
    IMUData *RT = (IMUData *)arg;

    IMU_Init(&(RT->imu), GYRO_ODR_208_HZ, ACC_ODR_208_HZ, GYRO_RANGE_500_DPS,
    ACC_RANGE_2_G);

    while(1){
       if(RT->mode == REAL_TIME){   //In replay mode there is no need to
    process imu data
            pthread_mutex_lock(&(RT->mutex));
            IMU_Process(&(RT->imu));
            pthread_mutex_unlock(&(RT->mutex));

            usleep(5000);
       }
    }
    return NULL;                        //Never gets here
}
```

Listing 11: Function implementing a thread to read and process IMU data in real-time, using a mutex for thread-safe operations

4.5.2 manageStorage()

As described previously this thread needs to have a maximum number of bytes in order to know the limit. The maximum number of Bytes the file can have(which represent 15minutes of Replay time) was calculated following the formula 3.1 and corresponds to approximately 16MegaBytes

```
#define MAX_SIZE        16000000     //main.h
```

```
static void *manageStorage(void *arg){
    IMUData *RT = (IMUData *)arg;
    ASM330LHB *imu = &RT->imu;
```

```
4
5       FILE *file = fopen(STORAGE_PATH, "w");  //  When rasp is initialized the
    ↪   file is reset since we ware dealing with diferent car crashes
6       if (file == NULL)
7           ASM330LHB_error("Error opening file.\n",0);
8       fclose(file);
9
10      char temp[MAX_SIZE / 2];                //  Temporary Buffer to save the
    ↪   7.5 most recent min of data
11      file = fopen(STORAGE_PATH, "a+");       //  Opens in Read and Write mode
12      if (file == NULL)
13          ASM330LHB_error("Error opening file.\n",0);
14
15      while(1)
16      {
17          if(RT->mode == REAL_TIME){            //  In replay mode there is no
    ↪   need to manage storage
18              fseek(file, 0, SEEK_END);           //  Places the file pointer in
    ↪   the end to check the file size
19              long int fileSize = ftell(file);    //  Returns the actual file
    ↪   size
20
21              if(fileSize < MAX_SIZE)             //  Keeps storing data in the
    ↪   text file
22                  fprintf(file,"IMU_Roll-> %.2f degrees\n\rIMU_Pitch-> %.2f
    ↪   degrees\n\rASM330_Yaw-> %.2f degrees\n\r", imu->roll, imu->pitch,
    ↪   imu->yaw);
23              else{
24                  printf("THE FILE IS FULL! CLEANING 7.5 MINUTES OF DATA...\n");
25                  fseek(file, MAX_SIZE / 2, SEEK_SET);            //  Places the
    ↪   pointer in the middle of the file
26                  fread(temp, sizeof(char), MAX_SIZE / 2, file);  //  Reads the
    ↪   twelve most recent hours of data into "temp"
27                  fclose(file);
28
29                  file = fopen(STORAGE_PATH, "w");                //  Re-opens
    ↪   the file in write mode cleaning its content
30                  if (file == NULL)
31                      IMU_error("Error opening file.\n",0);
32                  fwrite(temp, sizeof(char), MAX_SIZE / 2, file); //  Writes in
    ↪   the empty file the stored recent data
33              }
34              usleep(5000);
35          }
36      }
37      //Never gets here
38      fclose(file);
39      return NULL;
```

```
40 }
```

Listing 12: Thread function to manage IMU data storage in real-time, ensuring the file size stays within limits by periodically clearing old data

4.5.3 detectCrash()

In this section, we describe the implementation of a thread responsible for detecting crashes based on IMU data. The following code snippet demonstrates how the system monitors IMU readings to detect potential crashes and transitions to a replay mode to fetch post-crash data.

```
1  static void *detectCrash(void *arg){
2      IMUData *RTOS = (IMUData *)arg;
3      ASM330LHB *imu = &RTOS->imu;
4
5      FILE* file = fopen(STORAGE_PATH, "r");
6      if(file == NULL)
7          ASM330LHB_error("Error opening file.",0);
8
9      while(1){
10        if(RTOS->mode == REAL_TIME){     //In replay mode there is no need to
   ↪  process imu data
11             //Threshold-based collison detection
12         if ((imu->pitch > 80 || imu->pitch < -80) ||        //Car rotation
   ↪  check
13             (imu->roll > 60 || imu->roll < -60)   ||        //Car rotation
   ↪  check
14             (imu->acc_ms2[0]  < -threshold_positive || imu->acc_ms2[0] >
   ↪  threshold_positive ||  //Car X Velocity change check
15              imu->acc_ms2[1]  < -threshold_positive || imu->acc_ms2[1] >
   ↪  threshold_positive))   //Car Y Velocity change check
16             {
17                 printf("A crash as been detected! Fetching Post-Crash data(30
   ↪  seconds)\n");
18                 sleep(30);
19                 RTOS->mode = REPLAY;            //  Changes Transmition Mode
20
21                 //CALCULATE RELPLAY TIME
22                 fseek(file, 0, SEEK_END);                              //  Places
   ↪  the file pointer in the end to check the file size
23                 long int fileSize = ftell(file);                       //
   ↪  Returns the actual file size
24                 int replayTime = (fileSize * 0.00518) / 66;         //  5.18ms
   ↪  -> 66 bytes, So filesize bytes -> replayTime(seconds)REPLAY
25                 if(replayTime < MIN_REPLAY_TIME)
```

```
26                      IMRTU_error("Replay Time is to short.",0);
27                  else
28                      printf("[MODE]: REPLAY -> Replay Time: %dmin and %dsec\n",
    ↪ (replayTime/60), (replayTime%60));
29                  rewind(file);
30              }
31              usleep(5000);
32          }
33      }
34      return NULL;                        //Never gets here
35  }
36
```

Listing 13: Thread function for detecting collisions based on IMU data, triggering a mode switch to replay mode and calculating replay time when a collision is detected

4.5.4 transmitData()

In this section, is described the implementation of a thread responsible for transmitting IMU data either in real-time or from stored data during replay mode. The following code snippet demonstrates how the system handles data transmission over UART. The transmitData function runs continuously, monitoring the system mode and transmitting data accordingly

```
1   static void *transmitData(void *arg){
2       IMUData *RT = (IMUData *)arg;
3       ASM330LHB *imu = &RT->imu;
4
5       char uartMessage[87] = {0};
6       char buffer[87] = {0};
7       char line[21] = {0};
8       int lineCounter = 0;
9
10      ASM330LHB_uartInit(imu);
11
12      FILE *file = fopen(STORAGE_PATH, "r");
13      if(file == NULL)
14          ASM330LHB_error("Error opening file.",0);
15
16      while (1)
17      {
18          switch (RT->mode)
19          {
20              case REAL_TIME:     //Transmits the current IMU data in real-time
21                  sprintf(uartMessage, "ASM330_Roll-> %.2f degrees\nIMU_Pitch->
    ↪ %.2f degrees\nIMU_Yaw-> %.2f degrees\n", imu->roll, imu->pitch, imu->yaw);
```

```
22
23                  if(IMU_uartTransmit(imu, uartMessage) != 0)
24                      IMU_error("Error UART Transmit", imu->uart_fd);
25              break;
26              case REPLAY:        //Transmits the data stored in the raspberry
27                  while (fgets(line, sizeof(line), file) != NULL)
28                  {
29                      //Concatenate roll pitch and yaw from the file and trasmit
    ↪  them to UART
30                      lineCounter++;
31                      strcat(buffer, line);
32                      if (lineCounter == 3) {
33                          IMU_uartTransmit(imu, buffer);
34                          memset(buffer, 0, sizeof(buffer));
35                          lineCounter = 0;
36                      }
37                  }
38                  if (feof(file)) {
39                      printf("TRANSMITION COMPLETED - RESTARTING REPLAY...\n");
40                      rewind(file);       //  Points file to the begining
41
42                  }
43              break;
44          }
45
46          usleep(10000);
47      }
48      //Never gets here
49      fclose(file);
50      return NULL;
51  }
```

Listing 14: Thread function for transmitting IMU data via UART. In real-time mode, it sends the current IMU data; in replay mode, it transmits stored data from a file

Hardware

The final Raspberry system connections can be seen in Figure 4.3.

Figure 4.3: Raspberry Board

4.6 Hardware Accelerated Kalman Filter

To Implement the accelerator as mentioned before, was used Vitis HLS with the C programming language. The implementation of the filter in Vitis HLS is very similar with the one used in the raspberry since the language used is the same. The implementation can be seen bellow:

```
#include "example.h"

// Kalman Filter variables
const float A = 1;
const float B = 0;
const float C = 0;
const float u = 0;
const float H = 1;
```

```
9  const float Q = 0.005;
10 const float R = 0.1;
11 const float I = 1;
12
13 float Kalman_Filter(float Data) {
14
15     static float Xant = 0; // Static variables to maintain state
16     static float Pant = 1; // Static variables to maintain state
17
18     // Prediction
19     float Xapriori = A * Xant;
20
21     // Split the Papriori calculation into two steps to simplify the critical
   ↪  path
22     float Papriori_part1 = A * Pant;
23     float Papriori = Papriori_part1 * A + Q;
24
25     // Correction
26     float K = Papriori * H / (H * Papriori * H + R);
27     float X = Xapriori + K * (Data - H * Xapriori); // Perform operation on
   ↪  individual data value
28     float P = (I - K * H) * Papriori;
29
30     Pant = P; // Update state variables
31     Xant = X; // Update state variables
32
33     return X;
34 }
35
36 float top(float in) {
37 #pragma HLS DATAFLOW
38
39     return Kalman_Filter(in); // Call Kalman filter function
40 }
```

Listing 15: Implementation of a Kalman Filter in a hardware description language context, where the ‘top‘ function applies the filter to input data ‘in‘

In the top function was used the pragma `HLS DATAFLOW` which is a directive used to enable dataflow optimization. This directive allows the concurrent execution of multiple functions or loops, improving throughput by overlapping computation and communication.

When using pragma `HLS DATAFLOW`, the tool attempts to break down the operations into smaller, pipelined stages that can run simultaneously. An example can be seen in 4.4.

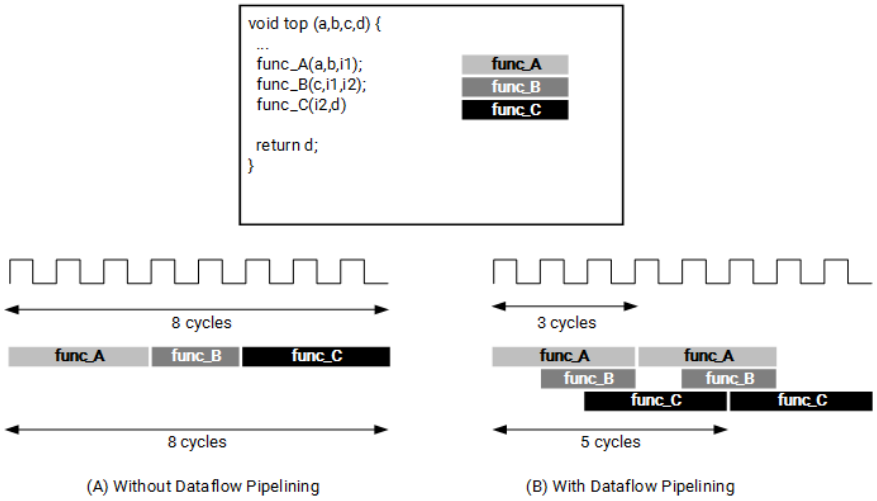


Figure 4.4: DATAFLOW pragma example

After implementing the code , the IP was exported to Vivado IP Repository (Figure 4.5) in order to link it to our FPGA block design.

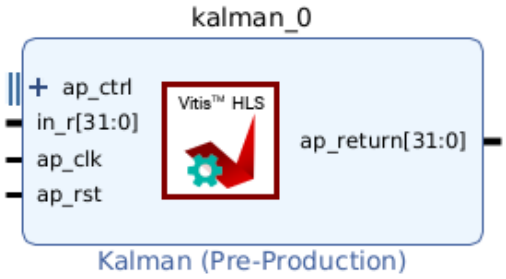


Figure 4.5: Kalman IP

To connect this IP to the Processing System, the only method available is via AXI. Therefore, an AXI wrapper for the Kalman filter IP was created, connecting the inputs and outputs of the IP to the slave registers of the AXI protocol as follows:

```
400  (....)
401    wire ap_done_0;
```

```
402     wire ap_idle_0;
403     wire ap_ready_0;
404     wire ap_start_0;
405     wire [31:0]data_IN;
406     wire [31:0]data_Out;
407     wire reset_rtl;
408     wire sys_clock;
409
410     assign sys_clock = S_AXI_ACLK;
411     assign reset_rtl = ~S_AXI_ARESETN;
412
413     // Control signals
414     assign ap_start_0 = slv_reg0[0]; // Bit 0 of slv_reg0 is start signal
415
416           // Add user logic here
417     design_1 kalman_design_i
418          (.ap_done_0(ap_done_0),
419           .ap_idle_0(ap_idle_0),
420           .ap_ready_0(ap_ready_0),
421           .ap_start_0(ap_start_0),
422           .data_In(slv_reg1),
423           .data_Out(slv_reg2),
424           .reset_rtl(reset_rtl),
425           .sys_clock(sys_clock));
426           // User logic ends
```

Listing 16: Verilog module instantiation ‘kalman_design_i‘ connecting Kalman filter IP inputs and outputs to AXI slave registers for integration with the Processing System

4.7 Zybo FreeRTOS

As mentioned before, the FPGA system, which has the least noise (IMUs connect to PMODs via PCB male connection and UART via micro USB cable), features only the IMU performance comparison and testing.
To implement the FPGA system, we used the Vitis unified IDE. First, we needed to create a hardware platform (e.g., bitsream.xsa) on Vivado and export it to Vitis. The hardware platform, shown in Figure 4.6, includes the FPGA Processing System, configured to read/write from two different I2C channels (for future simultaneous IMU comparison) and an AXI interconnect to connect the PS to the KalmanIP AXI Wrapper.

The I2C Controller (IIC) were mapped to the PMODs using the extended multiplexed I/O interface (EMIO).

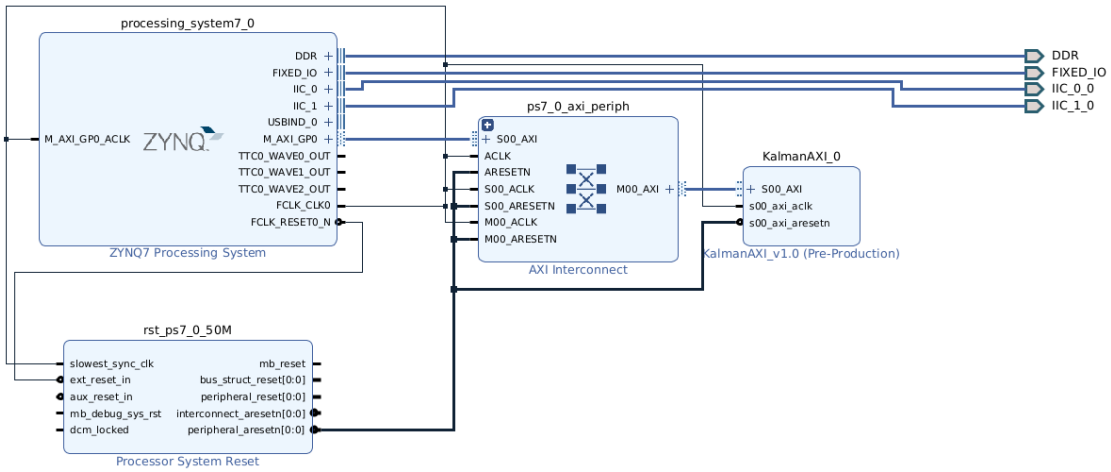


Figure 4.6: Vivado Design Diagram Block

When the platform is exported to Vitis, the software was built on top of the FreeRTOS OS to process different IMUs simultaneously.
Due to UART glitches caused by the overload of three different IMUs sending data, we only created two threads to test two IMUs at a time, despite having three IMUs available.

Vitis Integrated Design Environment

After creating the hardware platform along with the generated bitstream, we need to create a platform component in our Vitis workspace. To do so, the following figures represent the steps for that
After exporting the hardware along with the generated bitstream, it is necessary to create the platform in Vitis. The figures below demonstrate the process. First, the XSA (Xilinx Support Archive) file is loaded.

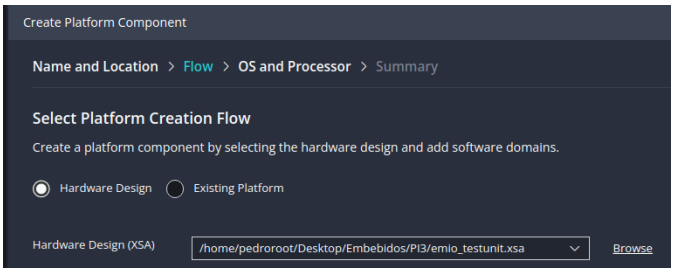


Figure 4.7: Vitis IDE Config 1

It is then possible to see that the respective processor (Cortex-A9) is detected, and the operating system, which in this case will be FreeRTOS, is selected.

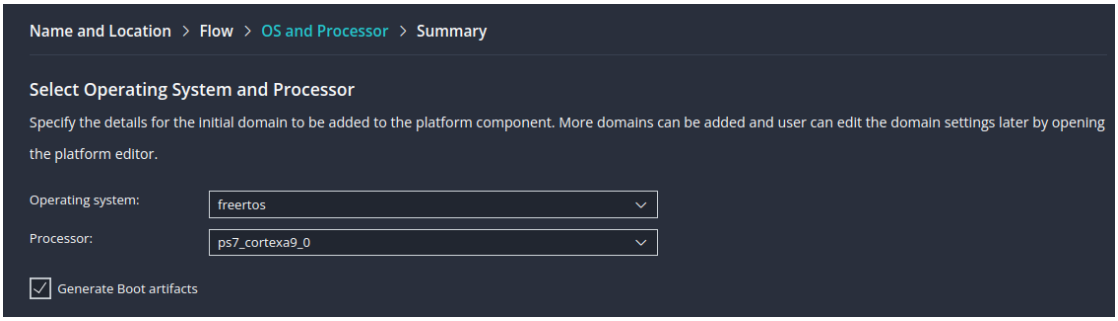


Figure 4.8: Vitis IDE Config 2

After the platform is created, it is only necessary to create an application and associate it with the newly created platform.

This code was implemented for the BMI088 and ASM330LHB IMU sensors.

```
57  static BMI088 imu_BMI;
58  static ASM330LHB imu_ASM;
59
60  int main( void )
61  {
62      BMI088_Init(&imu_BMI);
63      ASM330LHB_Init_config(&imu_ASM, GYRO_ODR_208_HZ, ACC_ODR_208_HZ,
    ↪   GYPRO_RANGE_SET_500DPS, ACC_RANGE_2_G);
64
65      // Create the mutex
66      xMutex = xSemaphoreCreateMutex();
67      configASSERT( xMutex ); // Ensure mutex was created
68
69  #if ( configSUPPORT_STATIC_ALLOCATION == 0 ) /* Normal or standard use case */
70
71          xTaskCreate( prvTask_BMI088,
72                                      ( const char * ) "BMI088",
73                                      configMINIMAL_STACK_SIZE,
74                                      NULL,
75                                      tskIDLE_PRIORITY + 1,
76                                      &Task_BMI088 );
77
78          xTaskCreate( prvTask_ASM330LHB,
79                                      ( const char * ) "ASM330LHB",
80                                      configMINIMAL_STACK_SIZE,
81                                      NULL,
```

```
82                                      tskIDLE_PRIORITY + 1,
83                                      &Task_ASM330LHB );
```

Listing 17: Implementation of FreeRTOS Tasks

Both IMU process functions share the same semaphore to avoid data corruption. This ensures that each task executes to completion without interruption from the other, as shown below.

```
115 /*-----------------------------------------------------------*/
116 static void prvTask_BMI088( void *pvParameters )
117 {
118         for( ;; )
119         {
120
121                 if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
122             //BMI088_Process(&imu_BMI);
123                         xSemaphoreGive(xMutex);
124                 }
125                 vTaskDelay(5);
126         }
127 }
128
129 /*-----------------------------------------------------------*/
130 static void prvTask_ASM330LHB( void *pvParameters )
131 {
132         for( ;; )
133         {
134
135                 if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
136                         //xil_printf("ASM330LHB processed\r\n");
137             ASM330LHB_Process(&imu_ASM);
138                         xSemaphoreGive(xMutex);
139                 }
140                 vTaskDelay(5);
141         }
142 }
```

Listing 18: FreeRTOS Tasks for BMI088 and ASM330LHB Sensor Processing

Hardware

After soldering all the components to the PCB in Figure 4.9 can be seen the final results.

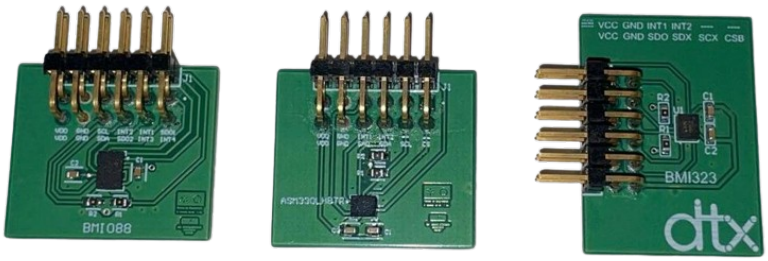


Figure 4.9: PCBs

After connecting the IMUs in the respective PMODs the structure was finalized as can be seen in the next figure.

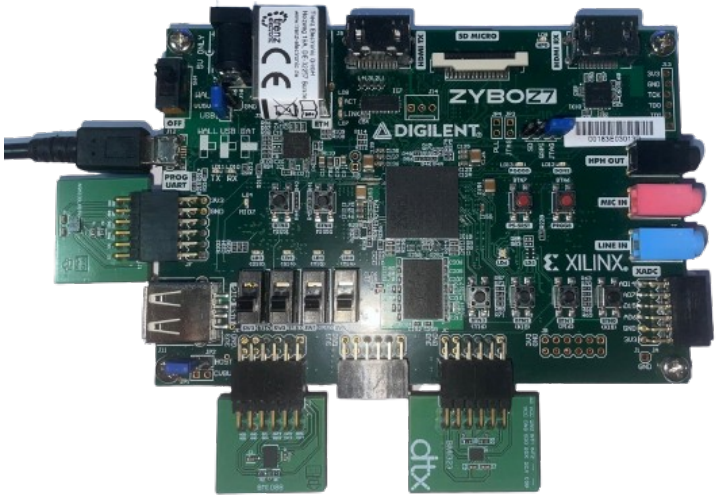


Figure 4.10: FPGA board

4.8 OpenGL 3D Scenario

To implement the 3D scenario with OpenGL, as mentioned in the Design phase, was used C++ as the programming language and <thread> API to have one

thread rendering the scenario and other to receive and process data from UART serial port.
First were created two data structures, one for the IMU Euler Angles and another for the Points that together will form the line of the IMU tracking. Since the code is extensive only the most important functions will be presented in the report.

```
19 struct point
20 {
21         float x, y, z;
22 
23         point(float x, float y, float z) : x(x), y(y), z(z){};
24 };
25 
26 typedef struct imuData{
27     float pitch;
28     float roll;
29     float yaw;
30 }IMU_LHB;
31 
32 float x = 0.0f, y = 0.0f, z = 0.0f;
33 vector<point> line;
34 static float angleX = 0;
35 static float angleY = 0;
36 static float angleZ = 0;
37 
38 IMU_LHB imu;
```

Listing 19: Definition of IMU Data Structure, point structure and variables

The `serialThread()` thread first configures the UART communication and after simply tokenizes the received data placing the angles in the respective variables.

```
92 void serialCOM(){
93         // Set up serial port parameters
94     boost::asio::io_service io;
95     boost::asio::serial_port port(io);
96     port.open("/dev/ttyUSB0");  // Specify your serial port device here
97     port.set_option(boost::asio::serial_port_base::baud_rate(115200));
98     port.set_option(boost::asio::serial_port_base::character_size(8));
99 
100    port.set_option(boost::asio::serial_port_base::stop_bits(
101    boost::asio::serial_port_base::stop_bits::one));
102 
103    port.set_option(boost::asio::serial_port_base::parity(
104    boost::asio::serial_port_base::parity::none));
105 
```

```
106     port.set_option(boost::asio::serial_port_base::flow_control(
107     boost::asio::serial_port_base::flow_control::none));
108 
109     // Read data from serial port
110     try {
111         while (true) {
112             boost::asio::streambuf buf;
113             boost::asio::read_until(port, buf, '\n');  // Read until newline
114             istream is(&buf);
115             string line;
116             getline(is, line);
117                         extractFloat(line);
118 
119                         // Print the extracted roll and pitch angles
120                         cout « "IMU_Roll : " « imu.roll « " degrees" « endl;
121                         cout « "IMU_Pitch: " « imu.pitch  « " degrees" « endl;
122             cout « "IMU_Yaw  : " « imu.yaw  « " degrees" « endl;
123         }
124     } catch (exception& e) {
125         cerr « "Exception: " « e.what() « endl;
126     }
127 }
128 void serialThread() {
129     /* Thread to get IMU data parallelly to the scene rendering*/
130         while (true) {
131         serialCOM();
132     }
133 }
```

Listing 20: Serial Communication and IMU Data Parsing

The Thread `renderScene()` is automatically ran when called the `glutMainLoop()` function in the `main()`. This thread is responsible for rendering the scene constantly by updating the Cube orientation based on the received angles and position in space by pushing back the last received point.
Since this project doesn't include a GPS or an LVDT the space position tracking cannot be really used in this project, however, this feature is implemented in the scenario and can be used in future work if any linear position sensor is added.

```
43 void renderScene(void)
44 {
45         initScene();
46 
47         rotateCube(imu.roll,imu.pitch,imu.yaw);
48 
49         /* Line Movement */
```

```
50          line.push_back(point(x, y, z));
51
52      //Here the cube will walk 0.1 units trough the chosen axis
53          //x += 0.1f;
54      //y += 0.1f;
55          //z += 0.1f;
56
57          // End of frame
58          glutSwapBuffers();
59  }
```

Listing 21: Rendering Scene Function

And finally the `main()` function that initializes everything and starts the threads.

```
292 int main(int argc, char **argv)
293 {
294         //GLUT init
295         glutInit(&argc, argv);
296         glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
297         glutInitWindowPosition(100, 100);
298         glutInitWindowSize(1000, 1000);
299         glutCreateWindow("IMU Scenario");
300
301         //Callback Registry
302         glutDisplayFunc(renderScene);
303         glutReshapeFunc(changeSize);
304         glutIdleFunc(renderScene);
305
306         //OpenGL settings
307         glEnable(GL_DEPTH_TEST);
308         glEnable(GL_CULL_FACE);
309         glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
310
311         line.push_back(point(0.0f, 0.0f, 0.0f));
312
313         thread serial(serialThread);
314
315         //GLUTs main cycle
316         glutMainLoop();
317
318         serial.join();
319
320         return 1;
321 }
```

Listing 22: Main Function Initialization and Thread Start

Chapter 5

Verification

5.1 OpenGL 3D Scenario

To test the scenario on the IMU, space position values were manually set for demonstration purposes, and orientation angles were read from the IMU. As shown in Figure 5.1, the cube rotates, and a line traces its trajectory.

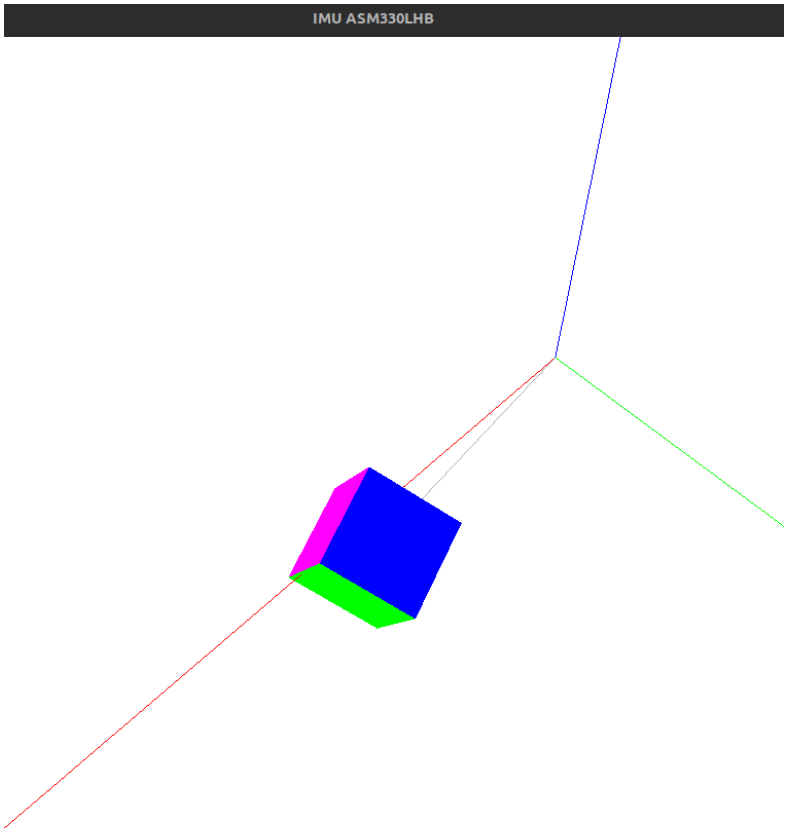


Figure 5.1: Final OpenGL scenario

5.2 Kalman Filter Unitary Test

In order to test the Kalman Filter algorithm, an isolated unitary test was performed using the C++ programming language for the filter implementation and

Python for data visualization. The following code snippet was implemented in this validation to plot the data.

```
import matplotlib.pyplot as plt
import csv

t_roll = []
Xestimado_r = []
MedicoesRoll = []
t_pitch = []
Xestimado_P = []
MedicoesPitch = []

with open('Roll.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        t_roll.append(float(row[0]))
        Xestimado_r.append(float(row[1]))
        MedicoesRoll.append(float(row[2]))

with open('Pitch.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        t_pitch.append(float(row[0]))
        Xestimado_P.append(float(row[1]))
        MedicoesPitch.append(float(row[2]))

plt.style.use('bmh')

# Plot ROLL data
plt.plot(t_roll, Xestimado_r, label='Prediction (ROLL)', color='orange')
plt.plot(t_roll, MedicoesRoll, label='Raw (ROLL)', color='red')

# Plot Pitch data
plt.plot(t_pitch, Xestimado_P, label='Prediction (PITCH)', color='green')
plt.plot(t_pitch, MedicoesPitch, label='Raw (PITCH)', color='blue')

plt.xlabel('Samples')
plt.ylabel('Degree')
plt.title('Prediction vs. Raw data (ROLL & PITCH)')
plt.legend()
plt.grid(True)
plt.show()
```

Listing 23: Python script to plot the data

As can be seen the kalman filter corrects the noisy data correctly.

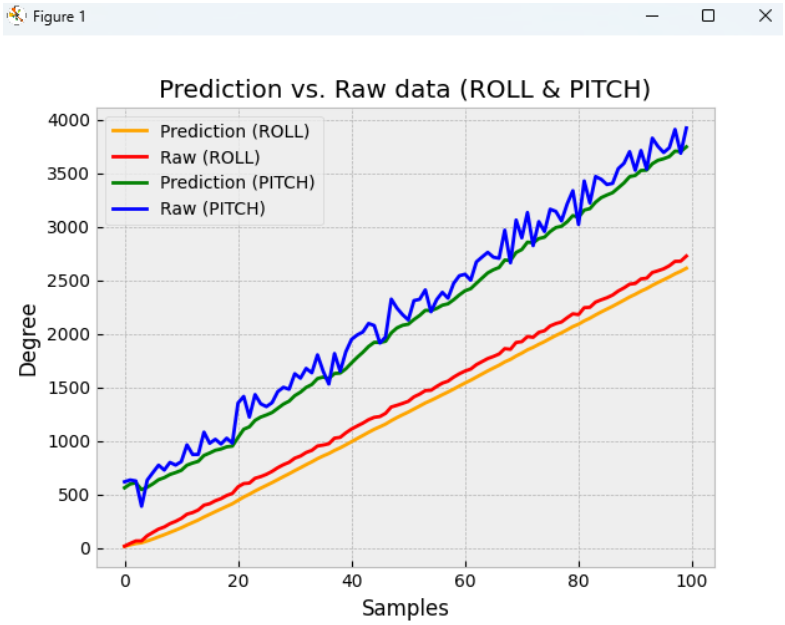


Figure 5.2: Kalman Filter - Unitary Test

5.3 Hardware-accelerated Kalman Filter

Before exporting the IP to Vivado, a testbench was created to verify the accelerator's correct behavior in software. This implementation involves creating an input stream with the values '200' and '0' to check if the filter transforms the output into a value close to '100'.

```
1  float example(float in);
2
3  int main()
4  {
5    float in = 0;
6    float out = 0;
7
8    int N = 20;
9    for (int i = 0; i < N; i++){
10           if((i % 2) == 0)
11                   in = 200;
12           else
13                   in = 0;
14
15           out = example(in);
```

```
16              printf("[%f] ",out);
17      }
18      return 0;
19  }
```

Listing 24

As shown in Figure 5.3, this test was successfully validated, as the output values are very close to the expected value of '100'.

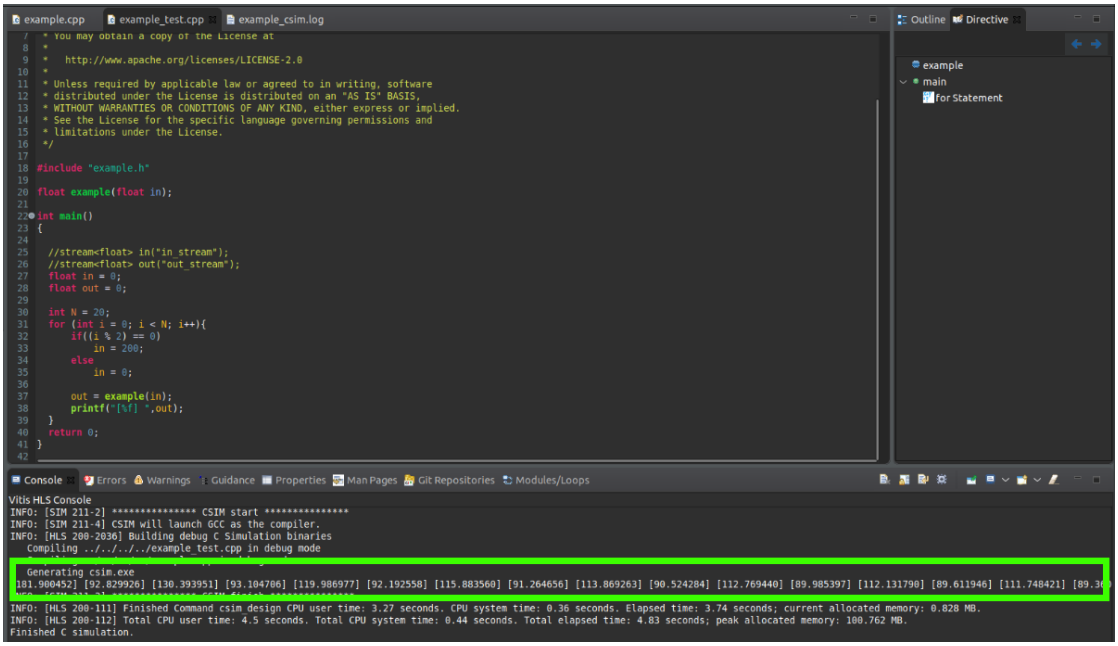


Figure 5.3: Kalman Filter Accelerator - Software Test

After the C simulation, the program was synthesized without any errors or warnings, as visible in Figure 5.4, which shows the Synthesis Summary.

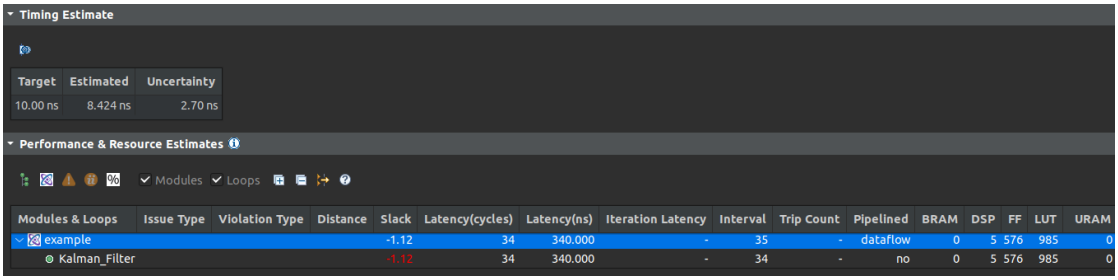


Figure 5.4: Synthesis Summary

Now, to test the accelerator hardware, after exporting the IP to Vivado, a testbench was created to emulate inputs changing between "200" and "0", with an expected output of approximately "100". The following testbench setup was used:

```
`timescale 1 ns / 1 ps

module design_1_wrapper_tb;

  reg [31:0] INK;
  wire [31:0] OUTK;
  wire ap_done_0;
  wire ap_idle_0;
  wire ap_ready_0;
  reg ap_start_0;
  reg reset_rtl;
  reg sys_clock;

  design_1_wrapper uut (
    .INK(INK),
    .OUTK(OUTK),
    .ap_done_0(ap_done_0),
    .ap_idle_0(ap_idle_0),
    .ap_ready_0(ap_ready_0),
    .ap_start_0(ap_start_0),
    .reset_rtl(reset_rtl),
    .sys_clock(sys_clock)
  );

  initial begin
    sys_clock = 0;
    forever #5 sys_clock = ~sys_clock;
  end

initial begin
    INK = 32'h00000000;
    ap_start_0 = 0;

    #20;
    reset_rtl = 0;
    ap_start_0=1;
    #20;
    INK = 32'd200;
    #100;
    INK = 32'd1;
    #100;
    INK = 32'd200;
    #100;
    INK = 32'd1;
```

```
45        #100;
46        INK = 32'd200;
47        #100;
48        INK = 32'd1;
49        #100;
50      end
51
52  endmodule
```

Listing 25: Testbench for the Kalman Filter Accelerator

Unfortunately, the accelerator malfunctioned, resulting in unexpected output values. Due to the short project time constraints, a solution could not be identified, rendering the hardware test of the hardware-accelerated Kalman filter inoperable (as shown in Figure 5.5).

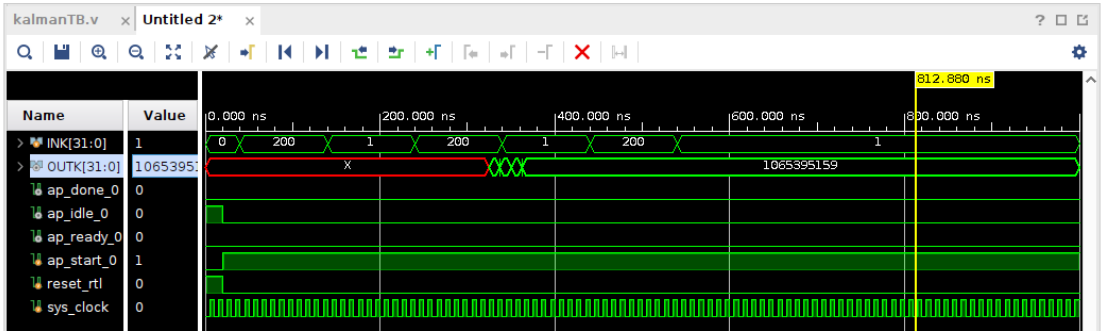


Figure 5.5: Hardware simulation of the Kalman Filter

5.4 STM32 System

To verify the STM32 system, Serial Wire Viewer (SWV) was enabled during debugging. The configurations are shown below:

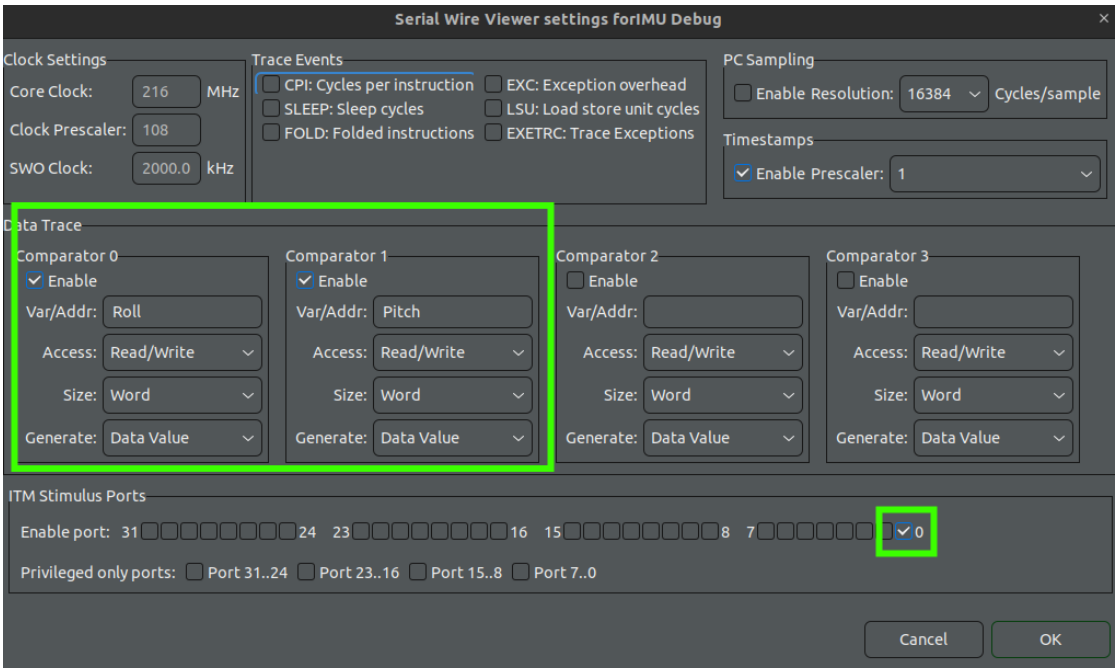


Figure 5.6: SWV Configuration

After this step, the program was started and now the correct values from the IMU can be seen successfully

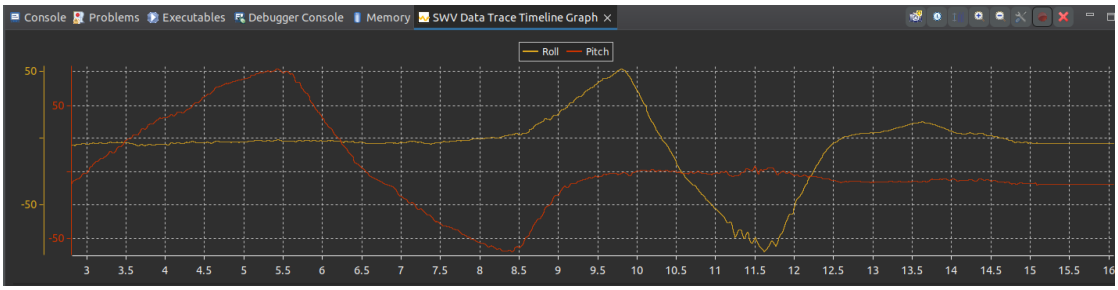


Figure 5.7: STM Verification - Angle values

5.5 Raspberry Pi System

5.5.1 IMU Processing

To verify the IMU processing with the Madgwick Filter, the Roll and Pitch values (for verification purposes only) were printed on the Raspberry Pi system. Figure 5.8 shows the IMU tilted 90 degrees to the right and 0 degrees to the front/back, displaying the corresponding angles below.

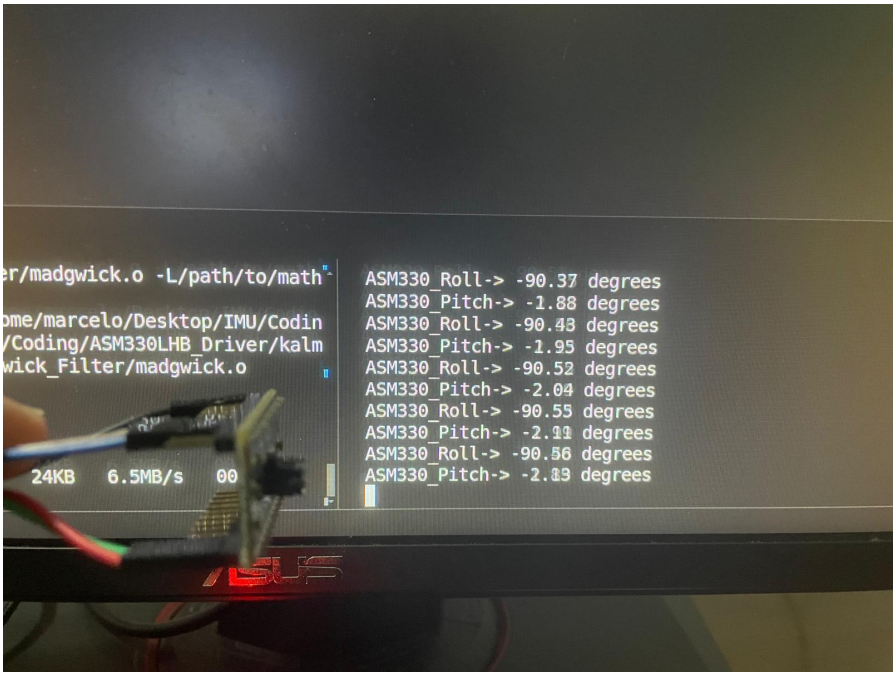


Figure 5.8: Processing Thread Verification

5.5.2 UART Transmit

To verify UART transmission, Figure 5.9 displays a GTK Terminal receiving UART information via FTDI module.

During this verification, the IMU orientation was tested, with the yaw value set to '0' due to the absence of a magnetometer in this IMU.

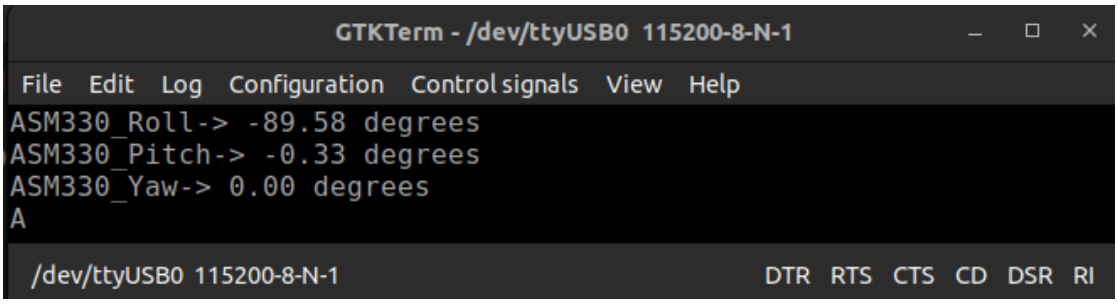


Figure 5.9: UART Thread Verification

5.5.3 Car crash detection

This subsection focuses on verifying the car crash detection algorithm by analyzing the angles at which the IMU detects a crash. As illustrated in Figure 5.10, the IMU exceeding a pitch of 80 degrees triggers replay mode with a 5-second delay. The replay then repeats upon completion.

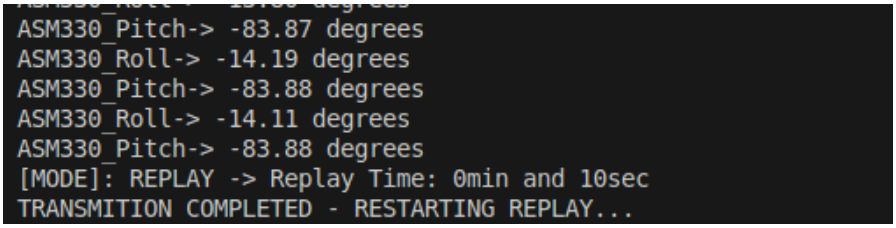


Figure 5.10: Car Crash Verification

5.6 Zybo System

5.6.1 PCBs Verification

The functionality of the PCBs with the IMUs was verified using previously implemented drivers. FPGA PMODs were configured to match the PCB outputs (Figure 5.11).
In this verification, the PCB was tilted 90 degrees to the left, with the angle displayed via UART.

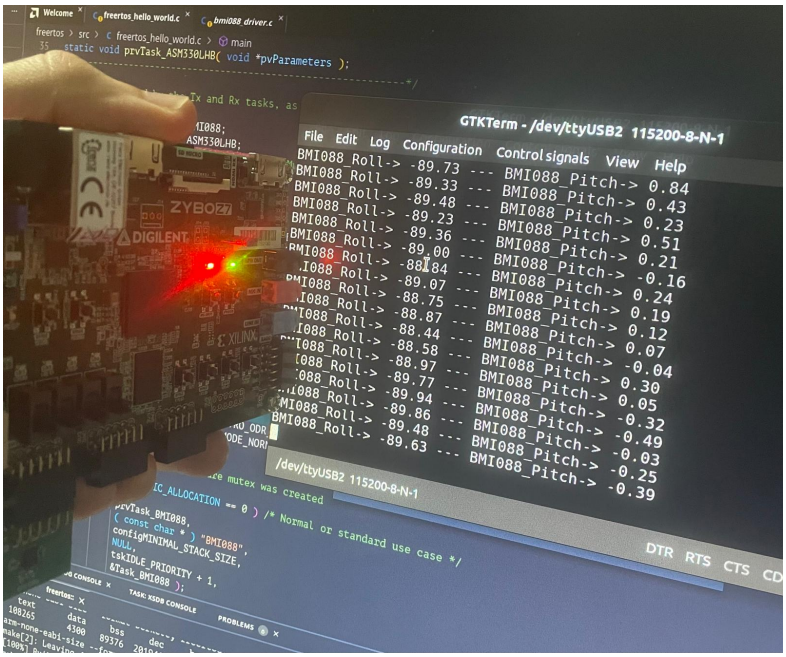


Figure 5.11: BMI088 PCB verification

Consistent with the BMI088 PCB verification (Figure 5.11), Figure 5.12 shows the verification process for the ASM330LHB PCB. Here, the PCB is also tilted 90 degrees to the left, and the angle is again visible in UART.

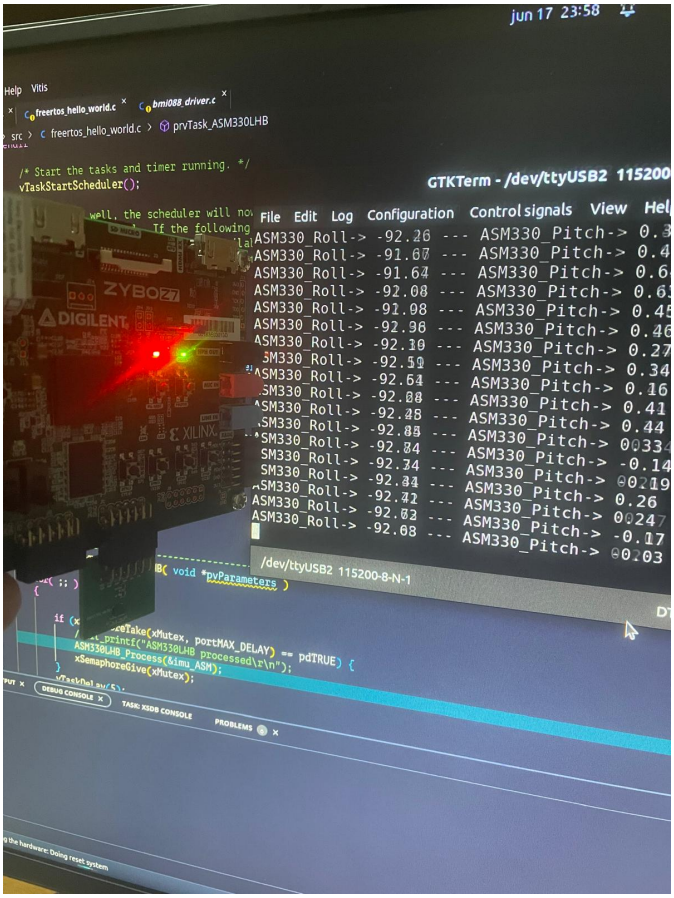
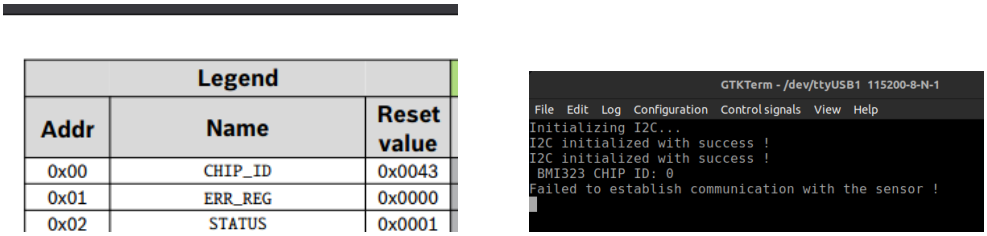


Figure 5.12: ASM330LHB PCB verification

Unfortunately, after testing the BMI323 IMU, were found communication issues with the board. This suggests a potential hardware problem since the software used follows the same steps successfully implemented with other IMUs.

After connecting the IMU pin 'SDO' to Ground, in order to select the default I2C address, the SDX and SCX (I2C data and clock) and after connecting the 'CSB' pin to VDD to activate I2C couldn't be read the IMU Chip ID (in 0x00 address) register's content correctly since the expected value was 0x43 as can be seen in figure below.



Legend		
Addr	Name	Reset value
0x00	CHIP_ID	0x0043
0x01	ERR_REG	0x0000
0x02	STATUS	0x0001

((a)) Chip ID ((b)) Chip ID read

Figure 5.13: Failed BMI323

5.6.2 Unfiltered vs Kalman Filter

To compare the impact of using a software Kalman filter, a Matlab script was created. This script receives data from UART, parses the information, and plots the BMI088 IMU data in real-time, allowing us to visualize the difference between unfiltered and Kalman-filtered data.

```
clear
clc
if ~isempty(instrfind)
    fclose(instrfind);
    delete(instrfind);
end
% Configuração da porta serial
s = serialport("/dev/ttyUSB1", 115200);
configureTerminator(s, "CR/LF");
flush(s);
% Variáveis para armazenar dados
maxSamples = 1000; % Número máximo de amostras a serem mantidas
roll = NaN(maxSamples, 1);
pitch = NaN(maxSamples, 1);
time = NaN(maxSamples, 1);
% Configuração inicial do gráfico
figure;
hRoll = plot(time, roll, 'r');
hold on;
hPitch = plot(time, pitch, 'b');
xlabel('Time (s)');
    ylabel('Angle (Degrees)');
```

```
23      legend('Roll', 'Pitch');
24  grid on;
25  % Contadores e intervalo de atualização
26  sampleCount = 0;
27  updateInterval = 10; % Atualizar a cada 10 novas amostras
28  startTime = datetime('now');
29  while true
30      % Read a line of data from the serial port
31      data = readline(s);
32
33      % Check if data is empty or not a string
34      if isempty(data) || ~isstring(data)
35          disp('Received unexpected data type or empty data.');
36          continue;  % Skip processing and continue with the next iteration
37      end
38
39      % Display received data for debugging
40      disp(['Dados recebidos: ', data]);
41
42      % Parse the line to extract roll and pitch values
43      tokens = regexp(data, 'BMI088_Roll-> ([\d\.\-]+) --- BMI088_Pitch->
    ↪  ([\d\.\-]+)', 'tokens');
44      if ~isempty(tokens)
45          rollValue = str2double(tokens{1}{1});
46          pitchValue = str2double(tokens{1}{2});
47
48          % Update arrays
49          sampleCount = sampleCount + 1;
50          roll(sampleCount) = rollValue;
51          pitch(sampleCount) = pitchValue;
52          time(sampleCount) = seconds(datetime('now') - startTime);
53
54          % Update the plot periodically
55          if mod(sampleCount, updateInterval) == 0
56              set(hRoll, 'XData', time(1:sampleCount), 'YData',
    ↪  roll(1:sampleCount));
57              set(hPitch, 'XData', time(1:sampleCount), 'YData',
    ↪  pitch(1:sampleCount));
58              drawnow;
59          end
60      end
61
62      % Check if maximum samples reached
63      if sampleCount >= maxSamples
64          break;
65      end
66  end
```

Figure 5.14 shows the results of processing the same sensor data with and without a Kalman filter. The clear difference highlights the filter's effectiveness in this system.

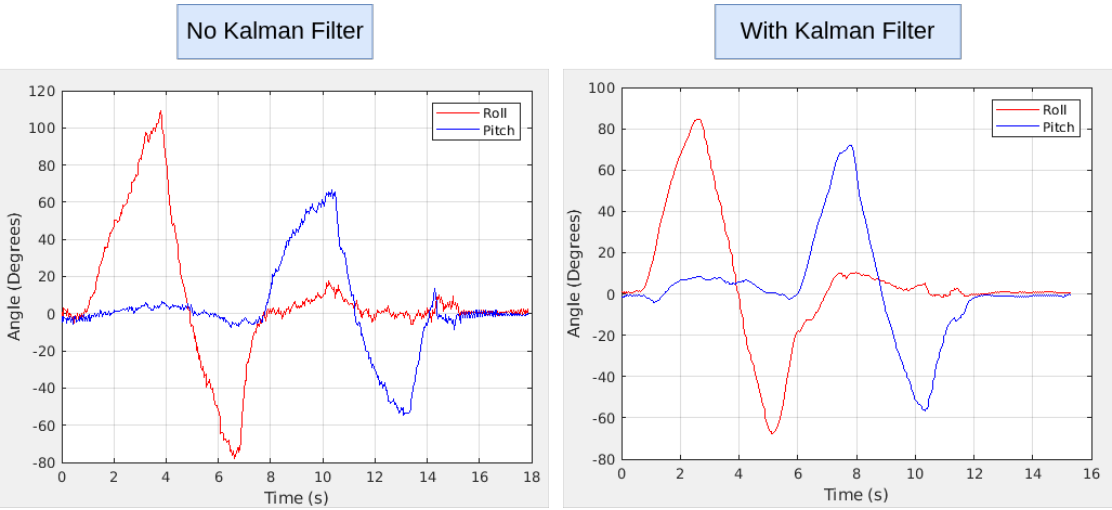


Figure 5.14: Unfiltered vs Kalman Filter

5.6.3 Yaw Drift

As previously mentioned in the design, the yaw drift will be verified to justify the absence of the yaw angle in the measurements and to emphasize the importance of using a magnetometer when dealing with Euler angles in the automotive-grade. This verification was made firstly by rotating the IMU in the Z-axis to prove that it is able to measure the rotation(0-6seconds).

After that the IMU is left static for the rest of the verification and it's possible to see a significant change in the yaw angle due to integration errors accumulated over time.

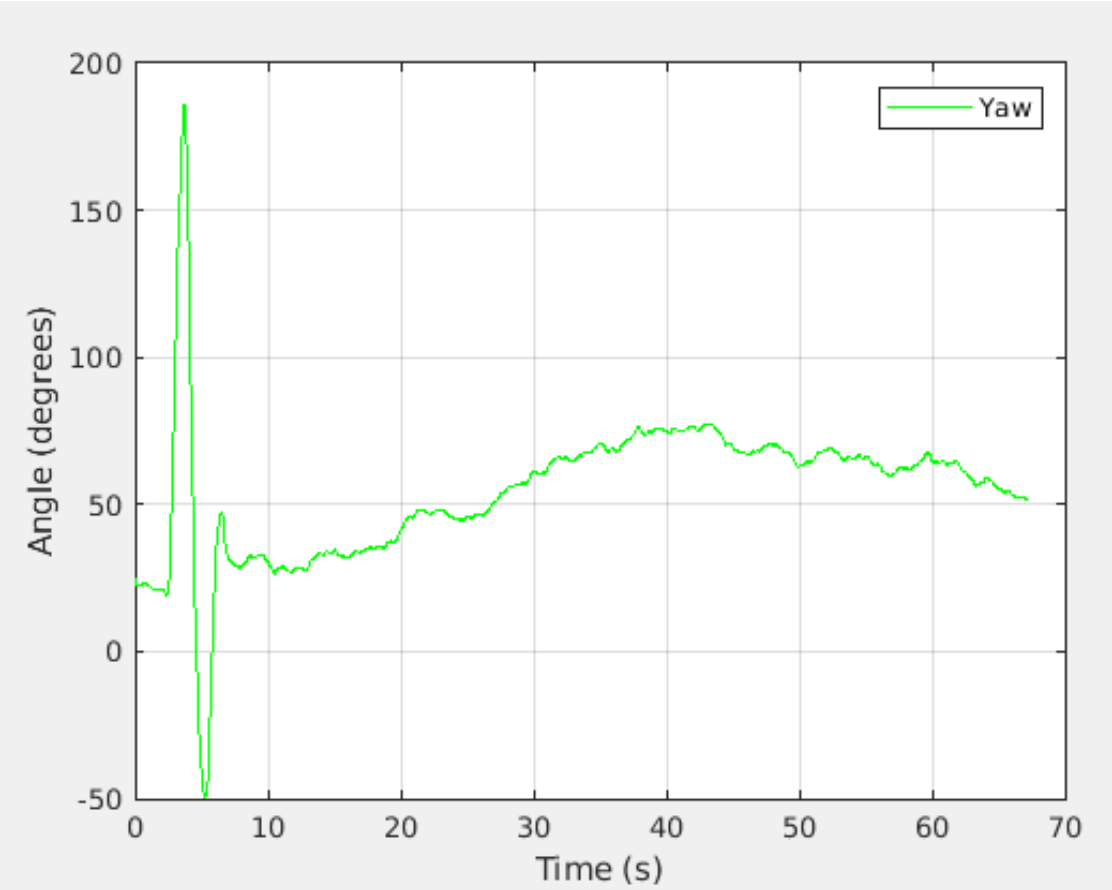


Figure 5.15: Yaw integration drift

5.6.4 ASM330LHB vs BMI088

Finally, the performances of the ASM330LHB and BMI088 IMUs were compared. Both were processed using the same FPGA connected by PCBs and a software Kalman filter.

The results showed that even with abrupt movements, both sensors maintained excellent performance and reliability, as evidenced by their approximately similar measurements.

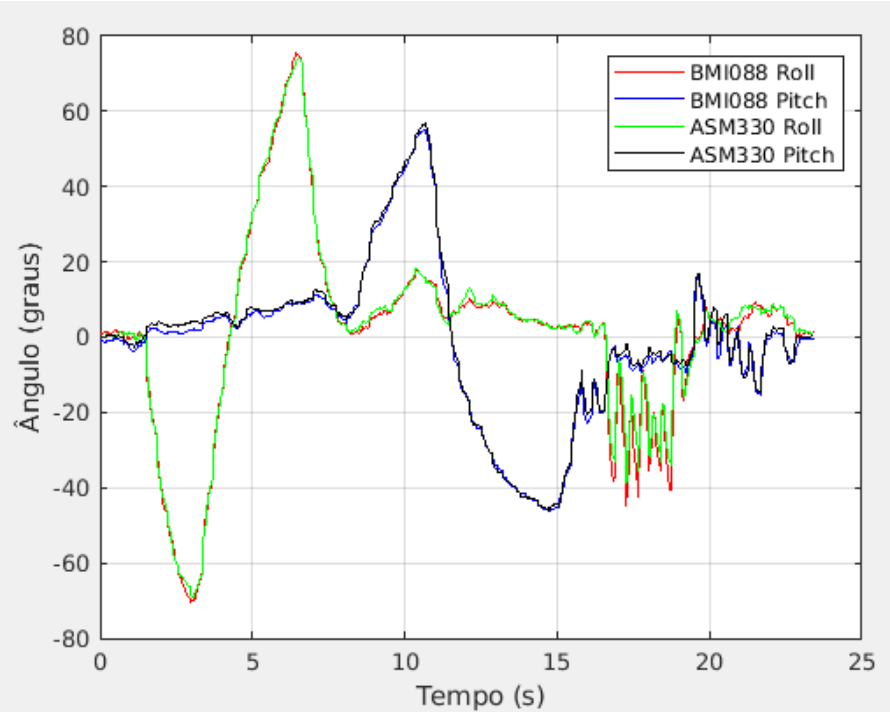
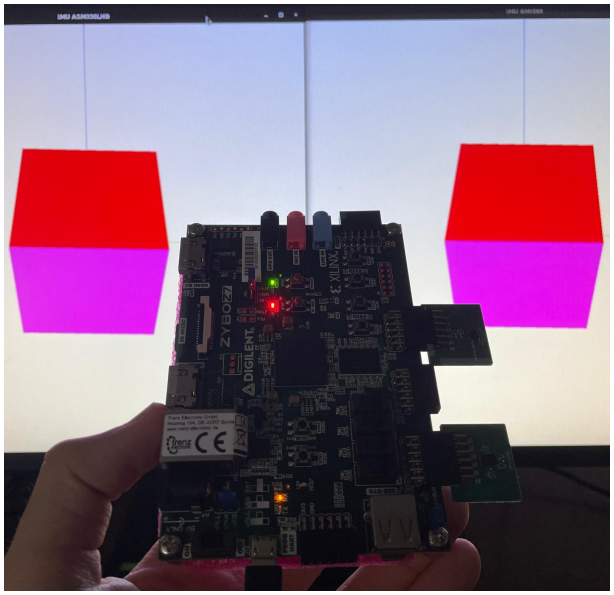
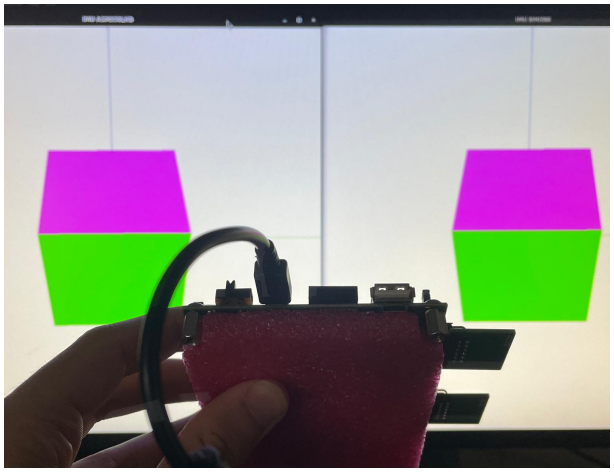


Figure 5.16: ASM330LHB vs BMI088

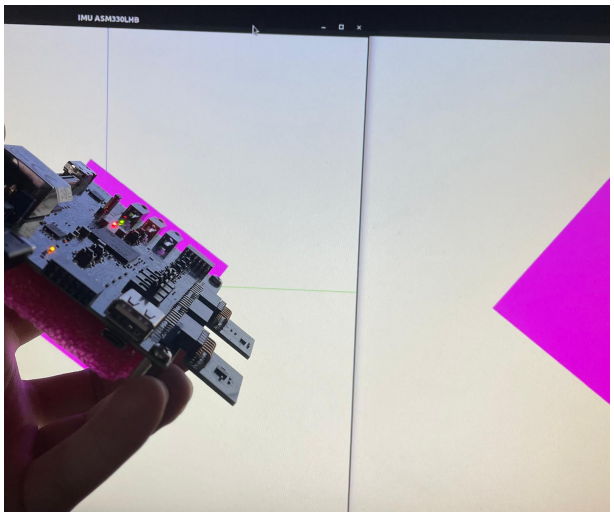
The following image shows the FPGA simultaneously processing data from both the BMI088 and ASM330LHB IMUs while verifying their correct orientation within the OpenGL scenario. It's important to note that transmitting all information through a single UART channel reduces the frames per second (FPS) of this verification process. However, as can be seen in the following page, the final orientation was successfully verified.



((a))



((b))



((c))

Figure 5.17: Final Verification

Chapter 6

Conclusion

To conclude the project, the initial proposal to develop an "IMU Data Acquisition System for real-time processing" was successfully achieved. This involved acquiring three different IMUs, designing three PCBs, and implementing drivers for each one. In this project three developemtn boards were used STM microcontroller, an FPGA, and a Raspberry Pi, each serving specific purposes as outlined earlier.

Additional features were successfully implemented, including an accident detection system and a replay mode for reviewing accidents. An OpenGL application visualizing a cube to replicate the IMU's orientation was also developed. Furthermore, a Kalman filter was implemented in software to effectively filter noise from the IMU data. However, attempts to accelerate the filter using hardware, particularly the FPGA, were unsuccessful.

Overall, the project was a success and achieved its primary objective of developing a real-time data acquisition system for an IMU.

6.1 Future work

For a future work, enhancing accuracy for practical applications, especially those related to vehicles, would be crucial. This could involve integrating an IMU with a magnetometer to establish a stable reference point for absolute orientation. Additionally, integrating GPS would enable precise spatial tracking, providing an exact replay of car crashes as they occurred.

Finally, exploring hardware acceleration for the Kalman filter using the FPGA board can significantly boost measurement accuracy, enabling real-time operation with high reliability in data readings.

Bibliography

[1] Kalman and Madgwick Filters theoretical foundations - https://ahrs.readthedocs.io/en/latest/filters/ekf.html.

[2] Amd vitis hls. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html#methodology.

[3] Bosch imu bmi088 datasheet. https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmi088-ds001.pdf.

[4] Bosch imu bmi088 datasheet. https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmi323-ds000.pdf.

[5] Market research. https://x-io.co.uk/x-imu3/.

[6] Stmicroelectronics imu asm330lhb datasheet. https://www.st.com/en/mems-and-sensors/asm330lhb.html.

[7] Vitis unified software platform documentation landing page. https://docs.amd.com/v/u/en-US/ug1416-vitis-documentation.

[8] Zybo reference manual. https://digilent.com/reference/programmable-logic/zybo/reference-manual.

[9] Zhou Zhang Changrui Bai and Xiaoying Han. A design and realization of fpga-based imu data acquisition system.

[10] A. Di Nisio A.M.L. Lanzolla G. Andria, F. Attivissimo and A. Pellegrino. Development of an automotive data acquisition platform for analysis of driving behavior.

[11] Chunpeng Kang and Zhong Su. Design of data acquisition and processing for imu.