TECHNISCHE UNIVERSITÄT
ILMENAU

Faculty of Computer Science and Automation
Computer Architecture and Embedded Systems

Group Studies Report

Supervisor: Dr.-Ing. Bernd Däne

Submitted by: B.Sc. Atieh Nazariasl, B.Sc. Tuna Temiz

TECHNISCHE UNIVERSITÄT
ILMENAU

Faculty of Computer Science and Automation
Computer Architecture and Embedded Systems

Sensor and Actuator Data Processing with FPGA

Start Date:	01.05.2021
End Date:	27.09.2021
Supervisor:	Dr.-Ing. Bernd Däne
Submitted by:	B.Sc. Atieh Nazariasl, B.Sc. Tuna Temiz

Abstract

One of the important tasks of a self-balancing vehicle is the determination of the device's position and orientation. The need for a constant line-of-sight connection to a GPS satellite, causes a number of problems with using the Global Positioning System (GPS) in GPS-denied environments. Furthermore, GPS positions cannot fulfill the real-time requirements of a self-balancing vehicle. Therefore, we will focus on developing an Inertial Navigation System (INS) in this work, which is a more reliable and real-time solution for determining an object's position. The Inertial Navigation System (INS) is a sensor-based, self-contained, dead-reckoning navigation system in which measurements at a high sample rate, provided by an accelerometer and a gyroscope. We will integrate the outputs of the gyroscope to determine the orientation of the device at each given time. The traveled distance of the device will be determined by integrating twice over the outputs of an accelerometer. Finally, we will pinpoint the location of the device compared to a starting-point by combining these two results. However, the needed integrations and the sensors errors lead to unbounded accumulation of errors. Therefore, it is imperative to control for the error factors, and compensante them as much as possible, in order to attain reliable measures. Additionally, as the system in which our implementation of the INS is to be employed there will be DC motors involved, we have also developed an open loop-control mechanism which allows for the operation of up to two motors by the user through the universal asynchronous receiver-transmitter (UART) interface. This paper presents the implementation of an INS and a DC motor driver on a fabric processor based Field Programmable Gate Array (FPGA). Furthermore, we will investigate the error rate of the used sensors, along with factors which will influence the measurement accuracy. We will then suggest a few methods to to reduce drift rate and possible future directions for the improvement in the functionality of the developed systems.

Contents

1 Introduction **1**

2 Background Research **3**

2.1 Inertial Navigation Systems . . . 3

2.1.1 The Principles of Inertial Navigation Systems . . . 3

2.1.2 The Evolution of Inertial Navigation Systems . . . 6

2.1.3 Limitations of Inertial Navigation Systems . . . 8

2.1.4 Suggested Methods for Error Compensation in Inertial Navigation . 10

2.2 Motor Controllers . . . 10

2.2.1 Pulse-width Modulation . . . 11

2.2.2 Universal Asynchronous Receiver-transmitter . . . 14

2.2.3 Open-loop Control . . . 15

3 Hardware Design and Software Implementation **17**

3.1 Hardware Design . . . 17

3.2 Software Configuration . . . 19

3.3 INS Implementation . . . 21

3.3.1 Design Overview . . . 22

3.3.2 Implementation Process . . . 22

3.4 PWM Implementation . . . 30

4 Results **34**

4.1 INS . . . 34

4.1.1 Experiments with Gyroscope . . . 34

4.1.2 Experiments with Accelerometer . . . 38

4.2 PWM . . . 39

5 Conclusions **43**

List of Abreviations **I**

List of Tables II

List of Figures III

Bibliography IV

1 Introduction

Determination of an object's position and orientation is of great importance to many applications. Auto-balancing vehicles especially need to access this data constantly and in real-time, in order to fulfill their purpose.

Although the Global Positioning System (GPS) provides sufficiently accurate positioning, in the order of meters to centimeters, especially when augmented with other satellite-based or ground-based augmentation systems, is unable to fulfill the requirements of continuity and reliability in our application. As GPS requires a line-of-sight (LOS) between the receiver's antenna and the satellites, in GPS-denied environments, such as tunnels, the LOS requirement cannot be always met, resulting in GPS outages caused by GPS signal blockage, interference, jamming and multi-path effects.

Due to the mentioned reasons, GPS cannot provide a continuous and reliable solution when used as a navigation system. Using INS (Inertial Navigation System), however, we will not have to worry about the continuity and real-time requirements of our application, as we integrate the measurements derived from an embedded IMU (Inertial Measurement Unit), consisting of an accelerometer and gyroscope. These measurements are used by a navigation processor to compute the position, velocity and attitude of the moving object relative to a known starting position, velocity and attitude. Measurements are integrated twice for accelerometers and once for gyroscopes to yield position and attitude. Therefore, inertial sensors alone are unsuitable for accurate positioning over an extended period of time.

Although the data provided by the IMU fulfills our real-time requirements, small errors in the measurement of acceleration and angular velocity are integrated into progressively larger errors in velocity, which are compounded into still greater errors in position.[Bri71] [ST15] Since the new position is calculated from the previous calculated position and the measured acceleration and angular velocity, these errors accumulate almost in proportion to to the time since the initial position was input. Even the best accelerometers would accumulate a highly inaccurate error in a matter of minutes. Therefore, the errors must be compensated correctly to achieve reliable measurements.

Of course, an INS is one of the many components in a fully functioning system, and in the case of a self-balancing vehicle, as it is the eventual intention for developing our system, usage of electrical motors will be necessary. Traditionally such motors, assuming they use direct current electrical energy, function through forces generated by fields of magnetism, and operated through a device or devices in a way that is previously defined by the operator or the system itself. Such a controller should have a reliable method of starting or stopping the motor, choosing the direction in which the motor is going to operate, select and regulate it's movement speed and it should have a method of preventing electrical faults such as short circuits.

In such early stages of a project where the current hardware and software setup only allows for testing and development and not actual usage of the proposed vehicle, having an open-loop controller for the motors is beneficial, as the controller is not affected by any output from any processes, and the user is free to run any test case they might wish. Such systems are rather common in many different systems, and while they cannot employ self-correction, there is a reduction in the overall complexity which provides an ideal opportunity for testing and development. [BD99]

It is possible to access the motors used in our system through the use of a peripheral module (PMOD) interface, which is an open standard that was developed by Digilent Inc. While the manufacturer provides operational instructions and libraries for the basic functionality of such PMODs, they are insufficient to be employed for our purposes in their current state, and the aforementioned systems were developed based on the initial functionality that was provided by the manufacturer, which makes their essentials necessary to explain, in addition to the adjustments and improvements that can be suggested to the current state of the systems.

The rest of this report is organized as follows: in section 2, an overview of important concepts and the previous related research is presented. Section 3 provides an overview of the used architecture and the proposed solution. Then section 4 summarizes the numerical experiments based on the implemented algorithms, and finally, a summary of our work and the future research directions are discussed in section 5.

2 Background Research

In this chapter, we will present the basic understanding necessary to investigate the research problem at hand. In our project, “Inertial Navigation Systems” and “Motor Controllers” stand out as the two important points of focus. As such, we will provide the reader with the existing literature about inertial navigation systems and using pulse width modulators.

2.1 Inertial Navigation Systems

An inertial navigation system (INS) is a navigation device that uses at least a motion detector (in most cases an accelerometer), and rotation sensors (gyroscopes) to calculate the position and the orientation of a moving object. These systems depend upon knowing the beginning position of the object, which puts them in the category of Dead Reckoning navigation systems.

In the following, we will explore the principles and the evolution of inertial navigation systems. Furthermore, we will point out the problems that may arise by relying only on INS in navigation systems, and suggest a few solutions to address them.

2.1.1 The Principles of Inertial Navigation Systems

Accelerometers and Calculation of the Position

As mentioned above, every INS consists of an acceleration sensor. This usually contains one accelerometer for the x axis, one for the y axis, and one accelerometer that returns the acceleration along the z axis. An accelerometer consists of two surface micromachined capacitive semiconductive sensing cells (g-cell) and a signal conditioning Application-specific IC (ASIC) contained in a single package. The g-cell can be considered as a set of beams attached to a movable central mass that move between fixed beams[LPSp08]. The movable

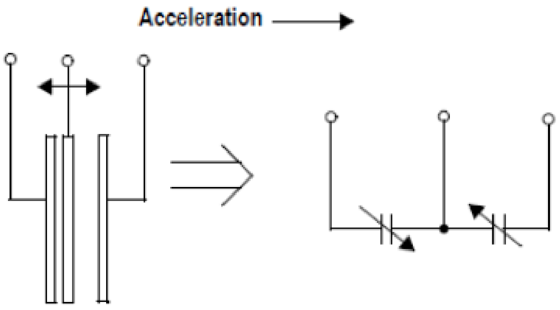


Figure 2.1 – A capacitative accelerometer

beams can move from their rest position by movements in the system, as seen in Figure 2.1.

The distance between the beams and the fixed beams on one side will increase by the same decrease in distance to the fixed beams on the other side. The change in distance is a measure of acceleration, as shown in Equation 2.1.

$$a(t) = \frac{A\epsilon}{D} \tag{2.1}$$

An accelerometer measures the acceleration and gravity it experiences. Acceleration is defined as the rate of change in velocity, and velocity is the rate of change in the position, thus:

$$v(t) = \int_0^t a(\theta)d\theta + V_0 \tag{2.2}$$

and

$$x(t) = \int_0^t (a(\theta) + v_0(\theta))d\theta + X_0 \tag{2.3}$$

where *a(t)*, *v(t) x(t)* represent the acceleration, velocity, and the position of the moving object at time *t*, respectively.

As in the INS use case, at *t = 0* the object is stationary and at position 0, we will arrive at the following:

$$v(t) = \int_0^t a(\theta)d\theta \ \ \& \ \ x(t) = \int_0^t v(\theta)d\theta \tag{2.4}$$

Using the definition of integral as the area under a curve, we can estimate the integration as the sum of small areas whose width is near zero, and their height is the magnitude of the integrated function. Thus, the integral of a function between two arbitrary points, *a and b*, can be expressed as:

$$\int_a^b f(x)dx = \lim_{n \to +\infty} \sum_{i=1}^{n} f(x_i)\Delta x \tag{2.5}$$

where $\Delta x = \frac{b-a}{n}$

An accelerometer gets us instant values of the magnitude of the object's acceleration. Therefore, the sampling interval between two measured accelerations will represent the base of this area, and the sampled acceleration represents the height. By combining 2.5 and 2.4, and assuming small enough sampling times, Δt we get:

$$v(t) = \lim_{n \to +\infty} \sum_{\theta=1}^{n} a(t)\Delta t \tag{2.6}$$

Gyroscope Measurements

A typical inertial navigation system uses a combination of gyroscopes, to compensate for the registered accelerations along x, y and z axis due to gravity. Gyroscopes measure the angular velocity, i.e. the rate of change of the system's angle with respect to time:

$$\omega = \frac{d\phi}{dt} \tag{2.7}$$

By considering the original orientation of the system as 0 and integrating over the angular velocity, the system's current orientation can be computed as:

$$\phi(t) = \int_0^t \omega(\theta)d\theta \tag{2.8}$$

As with the case of accelerometer, we can estimate this integral using a sum of small areas as well, arriving at:

$$\phi(t) = \lim_{n \to +\infty} \sum_{\theta=1}^{n} \omega(t)\Delta t \tag{2.9}$$

where Δt will be the sampling time, and $\omega(t)$ is the measured rotational speed around a specific angle.

2.1.2 The Evolution of Inertial Navigation Systems

During the past 40 years, Inertial navigation has undergone many developments, with their ascension from an expensive, almost impossible technology for guiding strategic missiles, to showing up in almost every controlled closed-loop moving object. Though the improvements through the years have had remarkable effects, inertial navigation still continues to provide many challenges in engineering.[KS98]

Gimballed Inertial Platform

Gimbals are essentially hinges that allow freedom of rotation about one axis. Each axis consists of a symmetric body (disk or ring) spinning about its axis, isolated from external torques. Ideally, the symmetric body's isolation from the external torques must be done in a frictionless manner.[Kel94] Sensors in the bearings provide measurements of gimbal angles. Three gimbals allow freedom of rotation of a vehicle about three axes while a central platform remains stationary with respect to inertial space.[WP09]

Strapdown Systems

Gimballed inertial navigation systems can be quite reliable. However, the gimbal arrangement is mechanically very complex, and mechanical resonances are unavoidable. They can also be expensive to maintain - if a gyro or accelerometer needs to be replaced, the gimbal set has to be completely dismantled and calibrated from the start. Moreover, testing inertial platforms, in order to figure out the Schuler period is time-consuming. As a result, since the early 1970s, the I.N. industry started moving towards the 'strap down' approach, in which the gimbals are altogether discarded, and the gyros and the accelerometers are strapped down onto the mounting frame. In this approach, the gyros can also measure rotations in space, in order to always knows which direction the accelerometer axis set is pointing in at any instant. In effect, we have a 'mathematical gimbal set', replacing the mechanical gimbals.[KIN98]

Gradually, mechanical gyroscopes were almost completely replaced by optical gyroscopes. Optical gyros are solid state devices, which makes them more reliable, and able to withstand high g fields beyond the capabilities of mechanical devices. Furthermore, they can be designed for harsh environments, and exhibit large dynamic range.[Kel94]

Despite years of development, mechanical and optical gyroscopes remain expensive, as they still have high part counts and require parts with high-precision tolerances and intricate assembly techniques. In contrast microelectromechanical sensors (MEMS) built using silicon micro-machining techniques have low part counts (a MEMS gyroscope can consist of as few as three parts) and are relatively cheap to manufacture.[Woo07]

A Blink into the Future of Inertial Navigation Systems

As we will review the limitations of standalone INS navigation, it has been observed that the accuracy of inertial navigation systems degrades rapidly with time due to the accumulations of nonlinear errors and noises from accelerometers and gyros, as shown in Figure 2.2. Although Low-cost, INS-based navigation can experience large position and attitude errors in a short period of time, in comparison with high-grade systems. These errors can be compensated by measurements from other sensors (e.g. magnetometer or GNSS). One popular trend in navigation system development is the integration of an INS with a GPS sensor, which through many studies, has been proven to be the ideal technique for seamless vehicular navigation [PLC12]. Integration of INS with GPS can also improve performance of GPS, by keeping track of the device's location in areas with bad or no reception of GPS signals, for example in urban tunnels [Joh09].

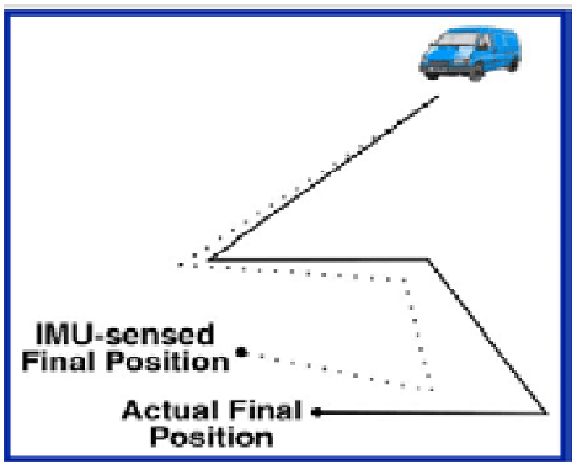


Figure 2.2 – The limitations of INS-based navigation systems

In the following section we will examine the sources of error that can arise in and INS-based navigation system, and suggest a few methods to compensate for these errors.

2.1.3 Limitations of Inertial Navigation Systems

The fact that INS-based navigation does not rely on any external environment or hardware, together with its low cost and little complexity make inertial navigation a favorable navigation mechanism. However this method is prone to unbounded error emerging from various error sources. Due to the nature of double integration of the accelerometer data, even slightly offsets can accumulate to big errors. This makes information solely based on INS particularly inaccurate over a longer period of time.[FH18]

Error Type	Description	Effects on Gyroscope Measurements	Effects on Accelerometer Measurements
Bias	constant bias ϵ	linear growing error in angle calculation	quadratically growing error in traveled distance calculation
White Noise	White noise with some standard deviation σ	A random walk with a the growth of standard deviation with the square root of time	White noise error which grows proportionally to time to the power of 1.5. [Nic16]
Temperature Effects	Temperature dependent residual bias	error with linear growth in time proportional to thermal bias	error with quadratic growth in time proportional to thermal bias
Calibration	Deterministic errors in scale factors, alignments and linearities	Angle calculation drift with linear growth proportional to the rate and duration of motion	Angle calculation drift with quadratic growth in time and proportional to the squared rate and duration of acceleration
Bias Instability	Bias fluctuations (usually modeled as a bias random walk)	A second-order random walk	A third-order random walk

Table 2.1 – Error Types and their consequences in inertial navigation systems

2.1.4 Suggested Methods for Error Compensation in Inertial Navigation

As suggested in "A Blink into the Future of Inertial Navigation Systems", We can improve the accuracy of an INS by incorporating its results with that of other sensors. The current technology usually combines the standard inertial navigation sensors with magnetometers, GPS, or "Ego Motion" sensors.

Using advanced algorithms to predict the errors is another way of improving our results. Most researches have suggested a variety of a correction algorithm through Kalman filter (KF).

The Kalman Filter filters noisy measurement data, estimating a state in each step, which makes it very suitable for real-time applications [MD17], [FH18]. The filtering consists of two steps feeding each other:

- The prediction step: using the dynamic model differential equations, the current state and covariance matrix for the next timestep is predicted.
- The update step: updates the estimate based on the measured real output of the system and a comparison between predicted and the measured output. The difference between both outputs is weighted by the Kalman Gain K_t

As the model equations for the normal Kalman Filter need to be linear, for more complex systems, we have to use the Extended Kalman Filter (EKF). This method approximates the error propagation with the Jacobian matrices to keep the Gaussian property.

Combination of Kalman Filter with "Deep Learning" algorithms in the recent years has proved very effective. [XF08] estimates the state based on time evaluation of the system's state transition matrix. This study utilizes neural network ensembles to deal with the Kalman filter.

2.2 Motor Controllers

In every system, a component that acts as an intermediary between the microcontroller, motor and power source is needed. The exact specifications and complexity of the hardware and software employed in the controller can vary widely based on the system that is being developed, however a traditional motor controller for DC motors controls the speed and direction of the motor, or in some cases multiple motors. It should be noted that a single power source generally provides the necessary power for all the motors involved in the system. Simplest way to control a DC Motor is probably an H-bridge circuit as employed in our project, which operate with four switches that are controlled in pairs. Pairing different switches together will allow different operational functionality to be observed on the motor.

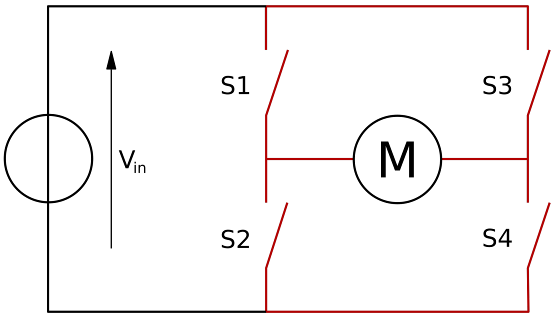


Figure 2.3 – Structure of an H-bridge circuit

Speed control using such circuitry is also possible, and based on the direction of the speed and torque, four possible states can be achieved. Torque controls whether there is braking or motion, and speed controls whether it is forward or reverse in nature. [Her12]

It is also useful to have more in depth information about PWM, which is a simple and efficient method of circuitry control, methods interfacing with the circuitry and control loops that are open-loop in nature as those were used in our project. Therefore the remaining chapter will explore these concepts and point out current research that is done related to them.

2.2.1 Pulse-width Modulation

What is called Pulse-width Modulation (PWM) is an efficient method of controlling the circuitry which is analog through the digital outputs of a given microprocessor. It is used widely in many different applications for many different purposes, which includes but not limited to testing, measuring, communicating as well as conversion and control of power.

Naturally, it is important to understand the nature of analog circuitry to make better sense of such a control method. A signal that is called analog possesses a value that changes

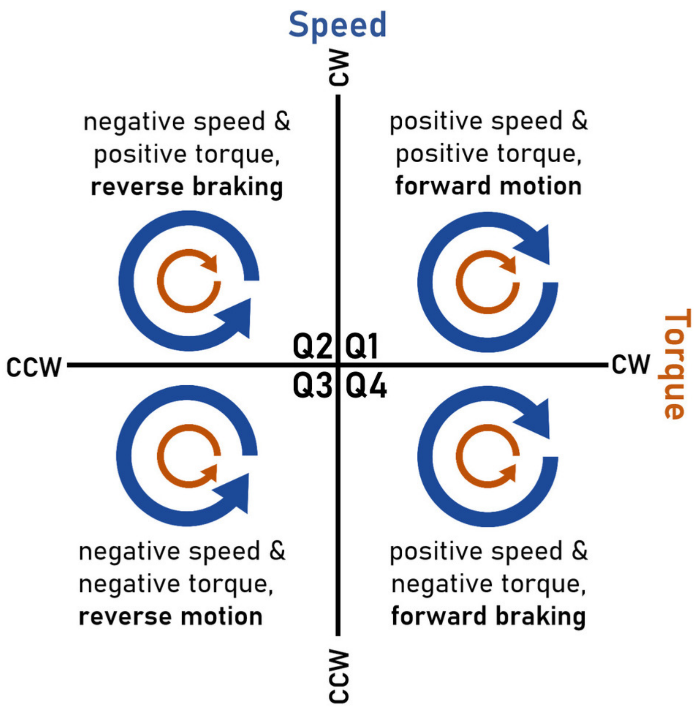

Speed
CW
negative speed & positive torque, **reverse braking**
positive speed & positive torque, **forward motion**
Q2 Q1
Q3 Q4
CCW
CW
Torque
negative speed & negative torque, **reverse motion**
positive speed & negative torque, **forward braking**
CCW

Figure 2.4 – Representation of speed and torque based on changes in the DC Motor polarity

without pause, and both magnitude and time properties have a resolution that is infinite. One such example could be a regular battery operating at nine volts, where the voltage output does not exactly equal to 9V, as it fluctuates over time, and therefore can possibly be any value that is a real number. Likewise, there is no limit to the possibilities in terms of the amount of current that could be drawn from it. This marks the primary difference between what is called digital and analog signals, as the former will always require a predetermined set of values between two numbers.

Direct control of objects is therefore possible through the use of analog signals, such as the volume on a radio, which can be achieved by possibly turning a dial. This affects the resistance in the system, which in turn affects the flowing current and therefore the volume goes up or down. While this is a rather simple method of control, analog circuits might be out of tune, get hot, and be sensitive to noise. Digital control is necessary to make the most efficient use out of such circuitry, and PWM basically does the encoding necessary to create the required digital control. As at any given time the current supply is either on or off, the PWM signal is called a digital one. [Bar01]

This is where the concept of duty cycle appears, as for example in a %10 duty cycle, the signal is only on for ten percent of the time and off for the rest. This directly controls how much of the full strength of the power source is being employed; using a %50 duty cycle for a 9V power supply would result in a analog signal that is equal to 4.5 volts.

A rectangular pulse wave is employed in the modulation, and the width of the said wave is controlled in order to alter the average value of the waveform. To describe the principle in a more mathematical manner, assuming the pulse waveform is *f(t)*, with a period value of T, low and high values of y_{min} and y_{max} respectively as well as the previously mentioned duty cycle D, we get the waveform through:

$$\overline{y} = \frac{1}{T}\int_0^T f(t)dt \tag{2.10}$$

Since *f(t)* is a pulse wave, the value of it is y_{max} for $0 < t < D.T$ and y_{min} for $D.T < t < T$ and therefore turns into:

$$\overline{y} = \frac{1}{T}(\int_0^{DT} y_{max}dt + \int_{DT}^{T} y_{min}dt) \tag{2.11}$$

$$= \frac{1}{T}(D \cdot T \cdot y_{max} + T(1-D)y_{min}) \tag{2.12}$$

$$= D \cdot y_{max} + (1-D)y_{min} \tag{2.13}$$

Last expression can be simplified in various different situations if

$$y_{min} = 0$$

because $\overline{y} = D{\cdot}y_{max}$. If this is the case, the average of the signal is defined by the duty cycle D.

PWM controllers are included with most microcontrollers, in which the operation starts setting the timer that is included on the chip which is responsible for controlling the wave, setting the direction of the modulation output, initiating the timer and enabling the controller. While implementation details vary for different modulation controllers, the core idea and functionality is mostly identical. [Ped16]

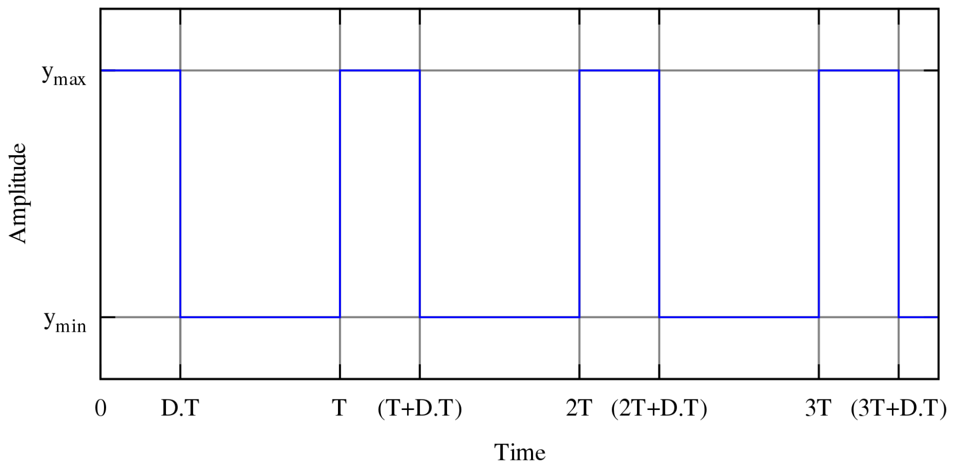


Figure 2.5 – Depiction of a pulse wave with the values y_{min}, y_{max} and D

All the way starting from the processor to the system that is being controlled, the signal stays as digital and the possible affection from noise is reduced, which makes it one of the advantages of using such a modulation system. PWM is widely implemented, it's understanding being a vital part for the control of direct current systems and circuitry. Current research directions involving PWM includes the modulation of various inverters [LVL+04], comparison of different schemes in time domain [KKC03], different switching strategies for PWM power converters [COKK07] and the efficiency of applications that employ such a modulation in their systems.

2.2.2 Universal Asynchronous Receiver-transmitter

The integrated circuits called universal asynchronous receiver-transmitters allow the control of the device at hand by interfacing with it through the serial devices it has connected to. For the purposes of outbound transmission, this circuitry allows the conversion of bytes it receives from the device into one bit stream, using parallel circuits in the process. In case the transmission type is inbound, it can convert the bit stream into bytes for the said system. In case of a transmission that is outbound, the addition of a parity bit is necessary, and it allows for the management of interrupts that might be issued by attached serial devices, for example the mouse and keyboard.

The design itself has three parts that matter, controllers for transmitting and receiving, as well as the baud rate generator. The baud rate corresponds to the travel speed of the information in the given channel. An internal clock signal is responsible for controlling all operations of the UART hardware, and at each pulse of the clock the point where the starting bit begins is tested by the receiver. Sampling of the line state is done in fixed intervals, and the results are stored in a shift register. The receiving system is made avare of that is stored in the shift register after the predetermined required time has elapsed, and a new flag is set by the UART to indicate the existence of new data. [PM08] [NP16]

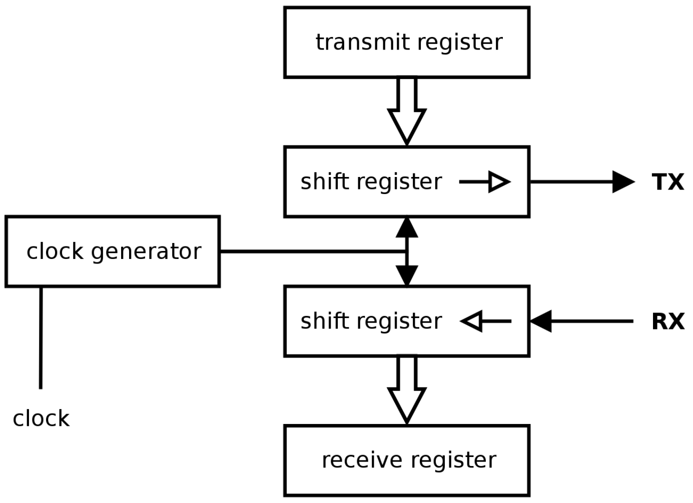


Figure 2.6 – Block diagram for a UART

In terms of transmission the state of the line has no bearing on the timing, so the process can be considered simpler. The moment the shift register has a character inserted into it, a start bit it created by the UART and after that the parity and stop bits are created and sent, and a flag is maintained to indicate whether the current state is busy or not, as transmission speed is dependent on the CPU and may vary widely from device to device.

Current research directions related to UART include the development of fast and effective transmitters and new communication modules [GRKR20], various implementations on different chips using different techniques and coming up with robust architectures. [PTNG07]

2.2.3 Open-loop Control

A system is called open-loop if the output of the system is dependent on the input that was put into it, but the controlling mechanism of the system operates outside the influence of the said output. As there is no feedback of any kind these sort of systems are also called non-feedback systems. Therefore, the output is not processed or used any further for the future functioning of the system.

As control systems are used to carry out the intended functionality of every system that is around us, proper controlling for optimal operation is necessary. When the system is non-feedback, to get a certain kind of output from the system, a specific reference input is fed into it, and that is where the usage of the output stops. One thing to point out here would be that since there is no way to change the input once the system starts operation, errors might occur if the output is different than what is desired, making the system prone to errors. However if the system is rather simple, or if the designer of the said system considers the possibility of the output deviating from the expected, then an open-loop control can be employed rather effectively. [Her12]

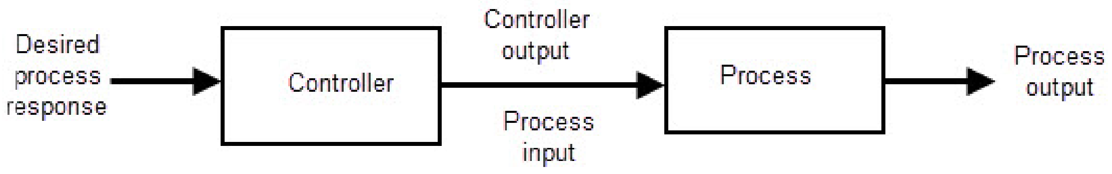


Figure 2.7 – Block diagram of an open-loop control system

Therefore some of the main advantages can be ease of maintenance and the simplicity of design, being economic in nature due to the overall smaller number of inputs to be fed, relative stability and easy predictability in terms of output, simple and convenient manner of operation, which proves to be valuable for testing purposes as we do here in our project.

Of course since certain errors and deviations from the normal occurs frequently in such systems, regular monitoring and recalibrating can be a necessity, and since outside disturbances have great effect on such systems, they must be considered in depth as well.

Effective usage examples of such systems can be traffic lights, automatic doors, heaters, washing machines and remote control mechanisms.

3 Hardware Design and Software Implementation

In this section, we will first introduce the used hardware and the used architecture as well as the software tools and the challenges we have faced during their configuration. We will then focus on the software implementation of an Inertial Navigation system, and finally close this section by introducing the Pulse Width Modulation system.

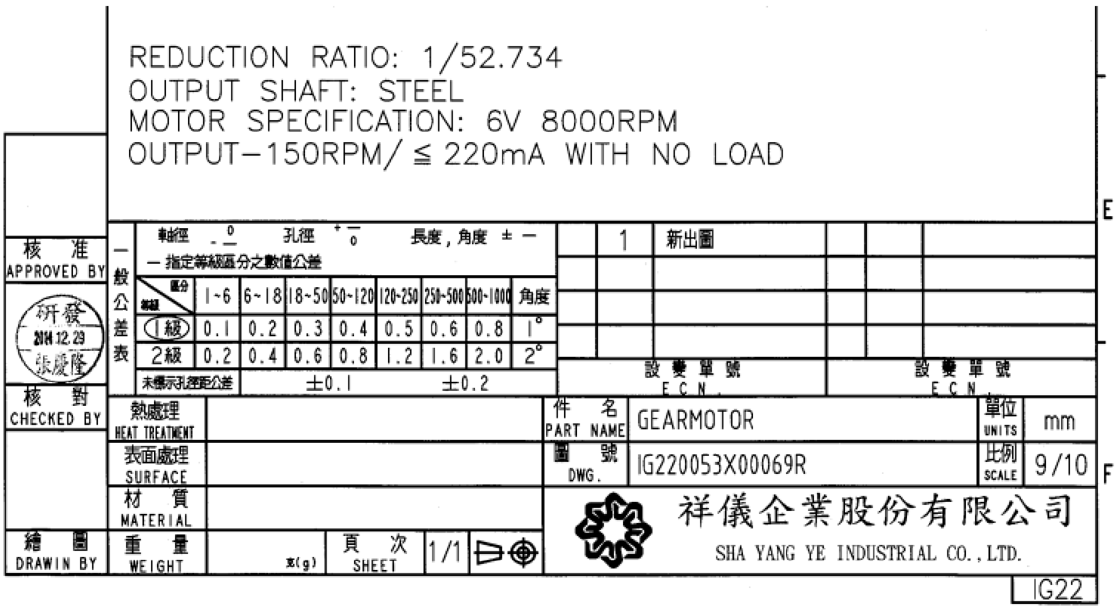
3.1 Hardware Design

The hardware components we have used in our design can be listed as a Xilinx development board, three peripheral module interfaces (PMOD), and a 6V DC motor designed for Digilent robot kits.

The model of the development board is Zybo Z7-20, which is one that is built around the Xilinx Zynq-7000 family. It possesses a dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA). It can be powered either by USB or any 5V external power source, and we used the latter for our hardware setup. It possesses six PMOD ports, three of which were used for our purposes. Further detailed information about the board can be found in the reference manual which can be accessed through the Digilent Inc. website.

The PMOD we have used in our PWM implementation is PMOD DHB1, which is a dual H-Bridge motor driver that is capable of driving two DC motors, a bipolar stepper motor, and other devices with inductive loads. The communication with the board is achieved through the general purpose input/output (GPIO) protocol, which allows for the host to send any type of signal to the PMOD and get an almost immediate response. It has over-current protection an a logic input voltage range of 2.5 V to 5 V. In our hardware setup PMOD DHB1 was attached to the JB high speed PMOD port on the board with a single DC motor connected to one of it’s two JST 6-pin ports for testing purposes.

The motor we have used for our implementation is a 6V DC motor/gearbox with rugged, heavy duty construction, and includes an encoder for sensing rotation and speed. The gear ratio of the motor is 1:53, and it is flexible enough to be used in custom applications as well as with Digilent Inc. kits. It's detailed specifications and the instructions for it's assembly, maintenance and operation can be found in it's specification manual written by the manufacturer Shayang Ye Industiral Co., Ltd.



REDUCTION RATIO: 1/52.734
OUTPUT SHAFT: STEEL
MOTOR SPECIFICATION: 6V 8000RPM
OUTPUT−150RPM/ ≦ 220mA WITH NO LOAD

一般公差表

軸徑 $^{0}_{-}$ 孔徑 $^{+}_{0}$ 長度，角度 ± 一

一 指定等級區分之數值公差

區分 / 等級	1~6	6~18	18~50	50~120	120~250	250~500	500~1000	角度
1級	0.1	0.2	0.3	0.4	0.5	0.6	0.8	1°
2級	0.2	0.4	0.6	0.8	1.2	1.6	2.0	2°
未標示孔徑距公差	±0.1			±0.2				

	1	新出圖			
設 變 單 號 E C N .			設 變 單 號 E C N .		

核 准 APPROVED BY
研發 2014.12.29 張慶隆
核 對 CHECKED BY
繪 圖 DRAWIN BY

熱處理 HEAT TREATMENT			
表面處理 SURFACE			
材 質 MATERIAL			
重 量 WEIGHT	克(g)	頁 次 SHEET	1/1

件 名 PART NAME	GEARMOTOR	單位 UNITS	mm
圖 號 DWG.	IG220053X00069R	比例 SCALE	9/10

祥儀企業股份有限公司
SHA YANG YE INDUSTRIAL CO., LTD.

IG22

E
F

Figure 3.1 – Data sheet of the motor

For the implementation of the INS, we have employed two additional PMODs, PMOD GYRO and PMOD ACL as our gyroscope and accelerometer respectively. PMOD GYRO allows access to motion sensing data on each of the three Cartesian axes and allows for the full configuration of the measured data by the user. It possesses two customizable interrupt pins, user configurable signal filtering, power-down and sleep modes. The accelerometer can provide us with high resolution acceleration changes, including inclination changes of less than 1.0 degrees. Both PMODs communicate with the board through the serial peripheral interface (SPI) protocol, which employs four communication pins along with a power and ground pin. The PMODs were attached to the JD high speed PMOD port and JE standard PMOD port on our board respectively. As with the board and motor, further information on the PMODs can be found in their reference manual.

Following the connection of the DC motor to the hardware setup, the board, alongside all the aforementioned components were installed on a metal chassis for safety, stability and convenience purposes.

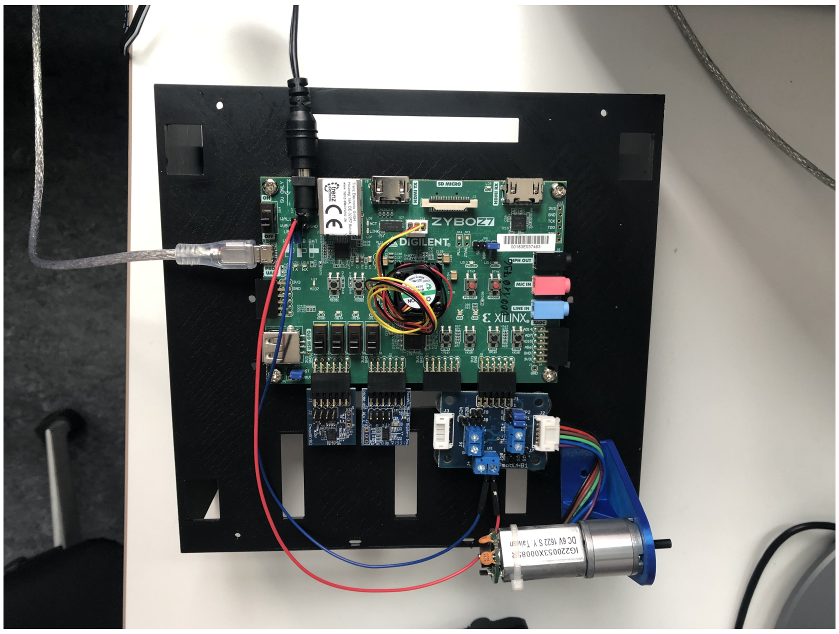


Figure 3.2 – Picture of the hardware setup with all the components attached

3.2 Software Configuration

The software configuration started alongside the hardware design, since further components were added to the board as the project progressed. In it's initial state, only the board was used for the beginning tests.

The software suite used for the development of the project was Xilinx Vivado Design Suite, and Xilinx Software Development Kit (XSDK) in particular, which is included in the Vivado Design Suite. The version of the design suite used was 2018.3. The entire project was worked on a single computer running on Linux.

The first step in the software configuration was checking whether the board was connected properly, and whether the design suite was functioning as intended. Therefore the first program that was ran on the board was a demo written in VHDL for controlling the LED lights on the board. Various modifications to the demo code was made by us to test different cases and get better acquainted with both the general architecture of the board and the user interface of the design suite. When these goals were met, the second stage of configuring and familiarizing ourselves with the XSDK started. The first step in using the software development kit was the creation of a block diagram on the design suit and importing it to the XSDK. Tutorials on the Digilent website proved helpful in coming up with our initial block diagram, but it is important to note that the instructions on the tutorial was not made specificially for our board, and initial attempts at running a “Hello World” program was delayed until we made the necessary changes in the block diagram specifications. For these changes we employed various other tutorials that used our board, and multiple changes were made at every attempt to realize a working block diagramEmbedded C language was used for development using the XSDK.

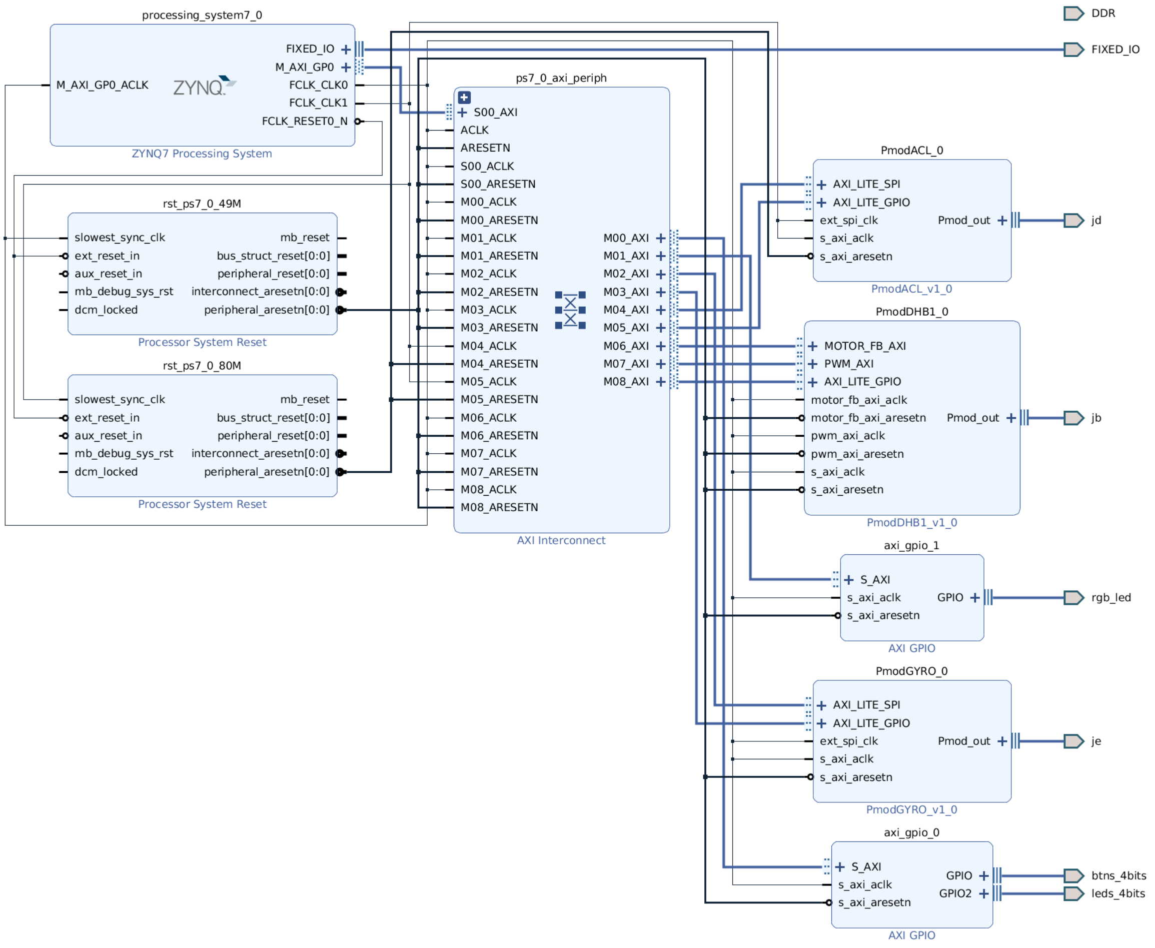


Figure 3.3 – Image of the final version of the block design, with all the used PMODs included

Another challenge worth noting was on how to interface with the board due to out unfamiliarity with the development kit, however this was eventually achieved by using PuTTY, which is a free terminal emulator and serial console application. Later on during the implementation for the open-loop control system for the motor, the built-in SDK terminal was also used, which will be elaborated on further in the next chapter.

The next step was to get the output for the demo driver code that is provided by the manufacturer for all of the individual PMODs, and the first one we integrated into the hardware design was PMOD GYRO, for which the tutorials on the Digilent website proved helpful. The correctness of the block design terminal configuration was tested by running the “Hello World” code at every step. Of course the installation of the correct libraries were necessary to have the PMODs function, and their correct working parameters had to be looked up from the reference manual. After the correct output for the first PMOD was received, the block design was rapidly extended to include all three PMODs to be used at the same time, and tests were conducted to see if they function correctly and simultaneously. It is at this stage where the development of the INS was feasible. The motor driver PMOD was tested right after being attached to the board through the use of an oscillator, but the actual motor was not attached to it until slightly later.

Two major challenges were met while configuring the software and our project files. The first is an issue where multiple projects using the same library files for the PMODs cause both projects to not recognize the library files and therefore making it impossible to operate the hardware. This problem was circumvented by having only one project at a time on the XSDK. Our research suggests that this is a commonly occuring issue on our version of the software, however whether the issue was fixed in later versions is not clear. The second issue is where the board is shown as detected and successfully connected to on the design suit, but upon the launch of the XSDK, the board is not detected when the code is built and the FPGA is being reprogrammed, where there was an error message citing a “JTAG port open error”. The only fix we had was closing the applications and starting them again. It is recommended to close the XSDK first before closing the design suit, as our experience shows that doing so corrupts the project files, making it impossible to launch the XSDK through the design suit and forcing the user to scrap the current project and go through the entire development from the start.

3.3 INS Implementation

In this section we provide an overview of the software and hardware configuration used for implementing the INS. We will then examine the implementation process and the overcome challenges in more detail. We will close this section by explaining the decisions made during the implementation process about the configurations of the code.

3.3.1 Design Overview

As mentioned in section 3.1, the Digilent PMOD GYRO, a 3-axis digital gyroscope expansion module, alongside PMOD ACL, a 3-axis digital accelerometer module, were our gyroscope and accelerometer of choice. As seen in Figure 3.4, the primary goal of our implementation was to calculate the angular position of the FPGA board after being rotated or moved. We received the output of the program through a UART configuration on the serial terminal “PuTTY”.

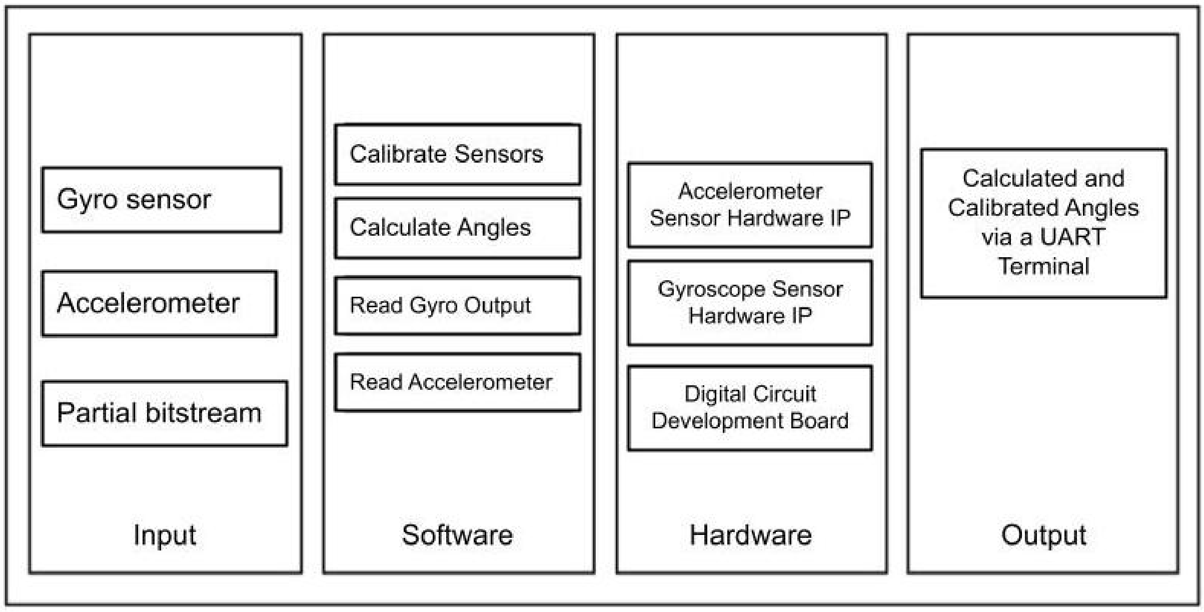

Gyro sensor
Accelerometer
Partial bitstream
Input
Calibrate Sensors
Calculate Angles
Read Gyro Output
Read Accelerometer
Software
Accelerometer Sensor Hardware IP
Gyroscope Sensor Hardware IP
Digital Circuit Development Board
Hardware
Calculated and Calibrated Angles via a UART Terminal
Output

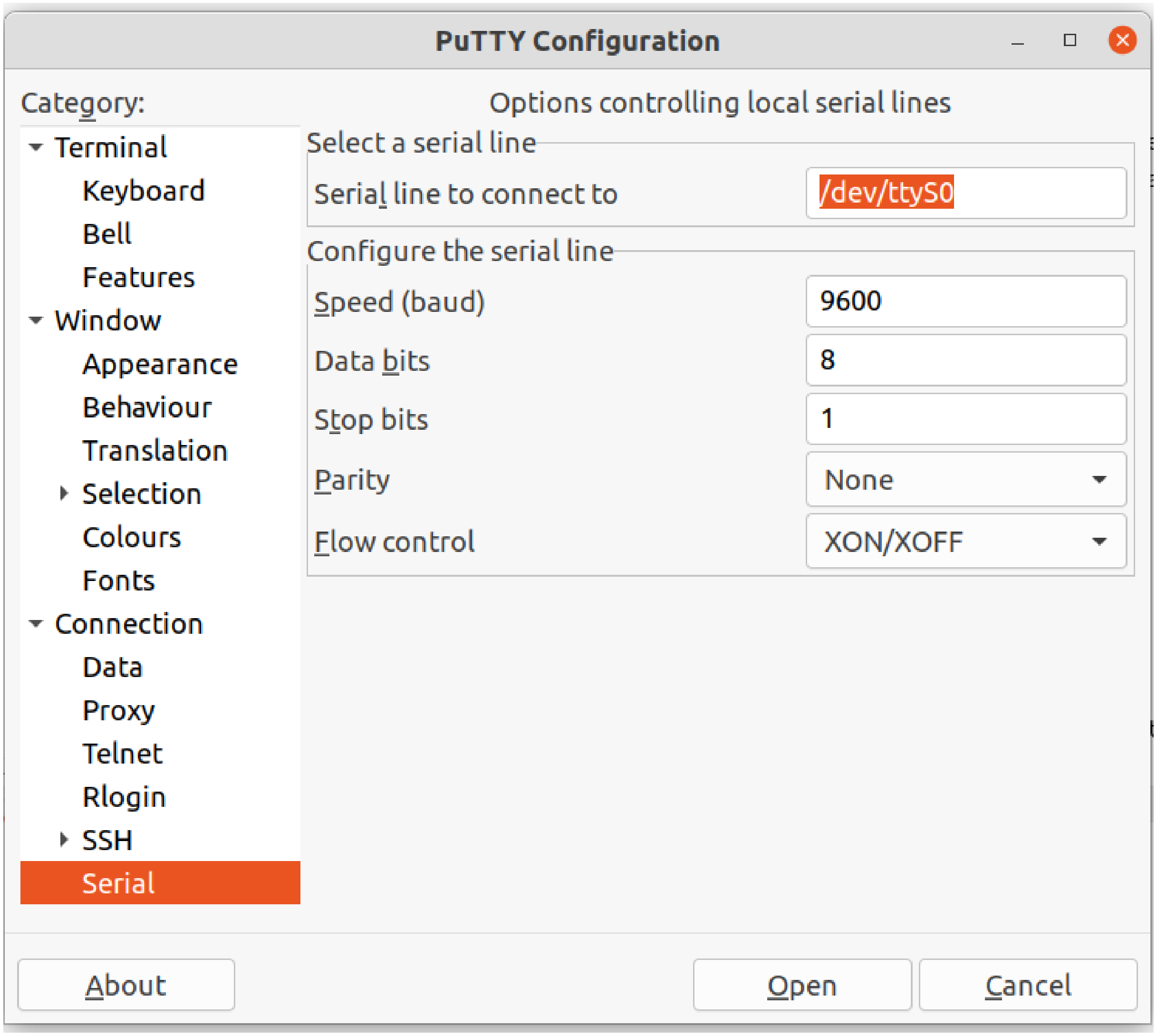
Figure 3.4 – Design Overview of the INS implementation

3.3.2 Implementation Process

Before connecting the accelerometer and the gyroscope, we followed the tutorial at A Hello World tutorial, as a first step to experiment with “Xilinx Software Development Kit”, and getting used to its environment. By running this tutorial, the serial terminal would simply print “Hello World!”

We used “PuTTY serial terminal emulator”, in order to receive data from the board. Figure 3.5 presents the serial port configurations, that allowed us receive data from the board.

As we moved on to Getting Started with the Vivado IP Integrator, we wanted to print the value of any key that would be pressed on the Zybo Board on our serial terminal. Running



Options controlling local serial lines

Select a serial line

Serial line to connect to: /dev/ttyS0

Configure the serial line

Speed (baud): 9600

Data bits: 8

Stop bits: 1

Parity: None

Flow control: XON/XOFF

Figure 3.5 – PuTTY serial configurations

this tutorial presented our first challenge, as no value upon pressing a key on the board was printed on the serial terminal. After more experimenting, the issue was solved by doing the following:

- We enabled "UART1" peripheral IO pin in the configurations of the Zynq processor
- Ignoring the tutorial on the creation of the project on Xilinx SDK, instead of creating an empty project, proceeded by creating a "Hello World" project[1].

[1]This issue was observed by a few Xilinx community members, but no solution was not provided. We assume that the "Hello World" project has a built-in configuration that facilitated the connection between terminal and the board, and this configuration was missing on an empty project.

Our next step was to add the accelerometer and the gyroscope to the block design using the tutorial at: Getting Started with Digilent Pmod IPs. At the end of this tutorial, we were supposed to observe the gyroscope or accelerometer readings in the PuTTY terminal. It is worth mentioning, that we adjusted the tutorial in this step as well, in order to get the program running.

Our first modification to the tutorial was to leave the default frequency of the clocks unchanged, as changing any default frequency interfered with the UART communication, and resulted in observing strange characters on the PuTTY terminal. We also did not connect the PMOD plug-ins using a second clock to the Zynq processor, but used the same clock for connecting both the gyro and the accelerometer. Finally, in contrast to the tutorial, we drew a completely new block design using the PMOD Gyro and PMODACL modules, and later on in the Xilinx SDK, we created a new "Hello World" project, as we observed that adding the gyro and accelerometer modules to the complete design of the last tutorial caused errors in building the project.

Next, we implemented the following functionalities:

- Gyro calibration
- Gyro sampling
- Error compensation for Gyro
- Rotation angle calculation
- Accelerometer sampling

Gyro Calibration

This was the first step towards calculation of the rotation angle. After logging the first outputs of the gyroscope, we observed that even though the board was in the stationary position, the gyroscope showed some amount of angular velocity on every axis. This issue is known as gyro drift, and even though the angular velocities read by the gyroscope were very small, in time, they added up to huge errors. Consequently, we had to calibrate the gyroscope in the first seconds of its use, in order to minimize the drift. For this purpose, we first converted the readings of the gyroscope from a twos complement format into integer values, as shown below:

```
while(1){
     if (GYRO_Int2Status(&myDevice) != 0) {
          xAxis = GYRO_getX(&myDevice);
          yAxis = GYRO_getY(&myDevice);
          zAxis = GYRO_getZ(&myDevice);

          xAxis = convert_twos_complement(xAxis);
          yAxis = convert_twos_complement(yAxis);
          zAxis = convert_twos_complement(zAxis);


     }
     if (read_counter < CALIBRATION_SIZE) {
          calibrate = CALIBRATING;
          xAvg += xAxis;
          yAvg += yAxis;
          zAvg += zAxis;
     } else if (read_counter == CALIBRATION_SIZE) {
          calibrate = CALIBRATION_ENDED;

           XTime_GetTime(&tEnd);

           print_gyro_calibration();
     }
}
```

In the above code, we average over a certain number of the values from the readings, in order to subtract this value from every future reading.

Gyro Sampling

For sampling the gyroscope readings, we logged its outputs in certain time intervals. We experimented with the accuracy of the output with respect to the duration of this interval, in order to determine the optimum interval between two consecutive readings. You can see our code snippet below:

```

while (1) {
          for (sleep_time = 500; sleep_time < 2000; sleep_counter += 100) {
               for (read_counter = 0; read_counter < 20; read_counter++) {
                usleep(sleep_time);
                print_gyro_data();
            }
        }
}
```

In this code, we start by sampling the gyroscope every 500 microseconds, and increase the sampling interval, shown in the code by sleep_time, by 100 microseconds, until the sleep_time reaches 2000 microseconds (2 milliseconds).

Rotation Angle Calculation

After calibrating the gyroscope, we could calculate the rotational angle of the device. For this, we integrated over the readings of the gyroscope over time, as mentioned in Equation 2.8.

Before integrating the values however, we had to multiply the readings by 8.75, as indicated by the data-sheet of the device [STM10]. Then, in order to register the calibrated amount, We logged the subtraction of the average gyro readings from the actual readings from the gyroscope. Furthermore, as the readings are delivered to us in Millie degrees, we had to divide the readings by 1000 to be able to calculate the rotational angle in degrees. We further divided each integrated value by 1000000, as the sleep time is in microseconds, and the readings from the gyroscope are in degrees per second. Below, we print the calculated rotational Angle for an indefinite time in 1,3 seconds intervals.

```
while (1) {
        for (read_counter = 0; read_counter < 1000; read_counter++) {
            usleep(sleep_time);
            sample(SAMPLE_GYRO);
            x = (xAxis - 104) ;
            y = (yAxis - 54)  ;
            z = (zAxis - 73) ;

            float float_x = x * 8.75 /1000;
            float float_y = y * 8.75 /1000;
            float float_z = z * 8.75 /1000;

            total_x += float_x * sleep_time/1000000;
            total_y += float_y * sleep_time/1000000;
            total_z += float_z * sleep_time/1000000;


        }
        printf("calculated angle: %f, %f, %f\n\r", total_x ,
                total_y , total_z );
}
```

Accelerometer sampling

As with gyroscope readings, the accelerometer was also registering values while the device did not move. But unlike the gyroscope driver code, the accelerometer already provided us with a calibration function. We sampled the accelerometer over 100 readings, each 1000 milliseconds apart, with the default calibration function of the driver code. The initialization and readings from the device are shown in the code snippet below:

```
void accelerometer_initialize() {

//printf("\n ACL Initializing\n");
    ACL_begin(&acl, XPAR_PMODACL_0_AXI_LITE_GPIO_BASEADDR,
            XPAR_PMODACL_0_AXI_LITE_SPI_BASEADDR);
    ACL_SetMeasure(&acl, 0);
    ACL_SetGRange(&acl, ACL_PAR_GRANGE_PM4G);
    ACL_SetMeasure(&acl, 1);
    ACL_CalibrateOneAxisGravitational(&acl, ACL_PAR_AXIS_ZP);
    sleep(1); // After calibration, some delay is required for the new
     settings
              // to take effect.
}
```

```
void run_accelerometer() {

    ACL_ReadAccelG(&acl, &x_acc, &y_acc, &z_acc);
    printf("X=%f\tY=%f\tZ=%f\n\r", x_acc, y_acc, z_acc);

}
```

Decisions made for the compromises

We now present the setting of the parameters that we have defined, along with the settings of the default parameters of the accelerometer and gyroscope. We enable the gyroscope with a threshold of 200, and an accuracy of 250 dps.

```
void gyro_initialize() {

        GYRO_begin(&myDevice, XPAR_PMODGYRO_0_AXI_LITE_SPI_BASEADDR,
        XPAR_PMODGYRO_0_AXI_LITE_GPIO_BASEADDR);

        // Set Threshold Registers
        GYRO_setThsXH(&myDevice, 0x0F);
        GYRO_setThsYH(&myDevice, 0x0F);
        GYRO_setThsZH(&myDevice, 0x0F);

        GYRO_enableInt1(&myDevice, GYRO_INT1_XHIE);     // Threshold
         interrupt
        GYRO_enableInt2(&myDevice, GYRO_REG3_I2_DRDY); // Data Rdy/FIFO
         interrupt
}
```

As reading frequency for the gyroscope is 8 Hz, we chose the sleep time to be 1300, as changing the sleep time in the code did not appear to have any effect on the accuracy of the readings. The buffering capacity for the calibration was chosen to be 100, as the results of the calibration did not vary with the increase in the buffer size, which can be seen in Figure 3.6.

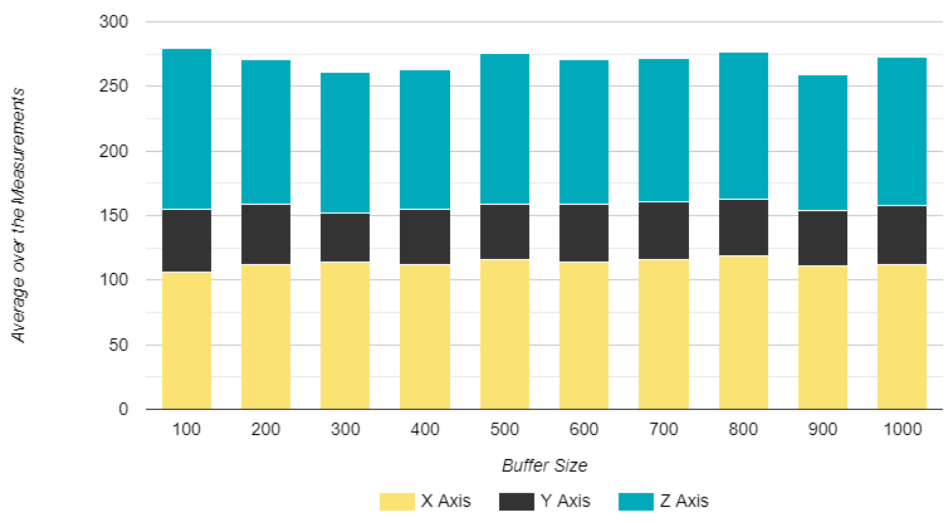


Figure 3.6 – The Average Readings from the Gyroscope with Respect to Buffer Size

As the device should not move during the calibration, we chose the buffer size as 100, in order to minimize the required time for the user to keep the device motionless. The relation between stationary time and the buffer size can be seen in Figure 3.7.

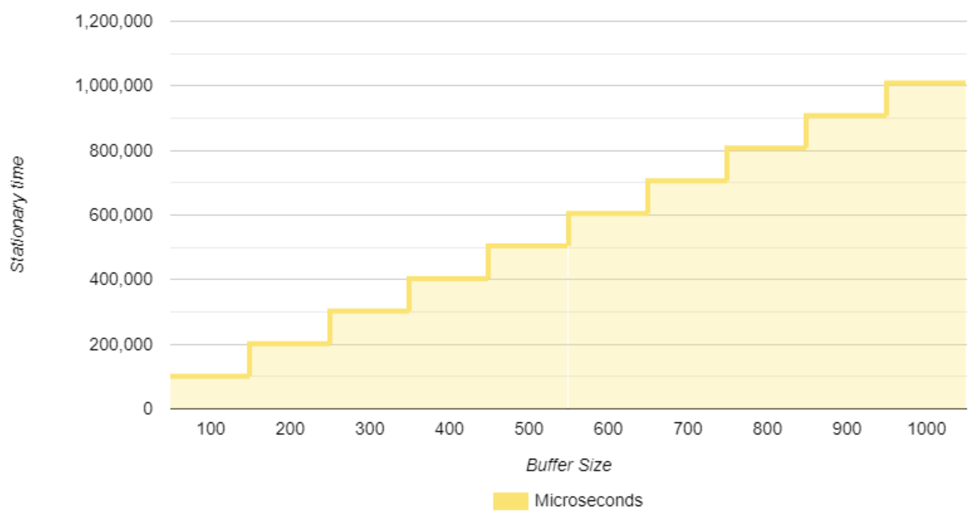


Figure 3.7 – Calibration Time with respect to Buffer Size

3.4 PWM Implementation

The main purpose of this part of the implementation was to have a functioning PWM, therefore to be able to control the speed, direction and the distance to be travelled by the motor and store the total distance travelled, to be able to control the two motors that can be attached to the motor driver circuitry seperately and to be able to operate all these functionalities with user input in the manner of an open-loop control system, which could be interfaced through UART. Therefore two main stages of implementation were made obvious; analysis of the provided driver functionalities to be repurposed, or failing that, to implement the needed functionalities for the desired operation of the motor, and to be able to interface, communicate and feed information to them through the computer console, in this case the built-in Vivado SDK terminal.

The necessary include files for the program consists of the header files of PMOD DHB1 provided by Digilent Inc. that were installed during the software configuration, and other headers necessary for the employment of UART, which are provided by Xilinx and therefore is already included with the Vivado Design Suite software.

Certain constant definitions were necessary, which map to the XPAR parameters that are defined in the included headers. Reason of their definition in the user program is to provide convenience to the user should they need to change the parameters. The length of the buffers that will be sent and received also need to be controlled, so further constants are defined which must be of 32 bytes in length or less as otherwise the complete buffer does not fit into the TX and RX FIFOs of the device. Additionally, a character is needed to terminate the program, and in this case the “ESC” character is used. Further macro definitions are made, and they are taken directly from the PMOD’s library files provided by the manufacturer.

```
/***************** Macros (Inline Functions) Definitions *****************/

#define GPIO_BASEADDR       XPAR_PMODDHB1_0_AXI_LITE_GPIO_BASEADDR
#define PWM_BASEADDR        XPAR_PMODDHB1_0_PWM_AXI_BASEADDR
#define MOTOR_FB_BASEADDR XPAR_PMODDHB1_0_MOTOR_FB_AXI_BASEADDR

#ifdef __MICROBLAZE__
#define CLK_FREQ XPAR_CPU_M_AXI_DP_FREQ_HZ
#else
#define CLK_FREQ 100000000 // FCLK0 frequency not found in xparameters.h
#endif
```

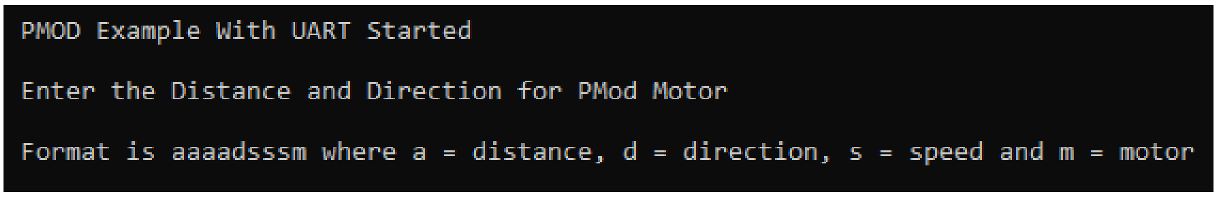
```

#define PWM_PER             2 // This denotes the period of DHB1 Enable
  signal in milliseconds (ms)
#define SENSOR_EDGES_PER_REV 4 // Since the encoder disk has 8 magnets, one
  revolution produces four positive edges
#define GEARBOX_RATIO        53 // For every wheel revolution, the disk
  revolves 53 times (substitute for 48 in case of unexpected behaviour)
```

This is followed by the definition of the function prototypes, first of which is derived from a Xilinx UART example code for echoing written input back on the terminal. While the functionality of the prototype is altered, it's name is not changed in the implementation, and is therefore referred to as "UartPsEchoExample". This is followed by the definition of prototypes for the operation of the PMOD, namely to run, initialize and clean up the demo, to drive the motor and to enable and disable the caches. Again, most of these are altered in functionality compared to the original code provided by the manufacturer, however their names are not altered. Variables are defined, including a send buffer to transmit, send and receive data as well those for the functionality of the PMOD itself.

This is followed by the main function, where first the demo is initialized. This is done by first enabling the caches and using the "DHB1_begin" function. This function, derived from the library, takes five parameters; a pointer to a PmodDHB1 object to initialize, the base addresses of PmodDHB1 GPIO and PWM registers, the clock frequency of the DHB1 IP in Hertz and the period of DHB1 enable signal in milliseconds. Following this, the "MotorFeedback_init" function is called, which initializes the MotorFeedback object with motor and encoder parameters. The parameters of this function are; a pointer to a MotorFeedback object to initialize, the base address of MotorFeedback registers, the frequency of MotorFeedback's clock in Hz, the number of rising edges detected by Hall effect sensor in 1 revolution of the encoder disk and the wheel gear to encoder gear ratio (number of encoder revolutions for every wheel revolution). Following this the motor is disabled and the initialization is complete.

Then begins the main body of the program, named "UartPsEchoExample" like the one that was used from the Xilinx examples, however it's functionality is altered. Originally this function did a minimal test on the UART device using the hardware interface. All the functionalities of the open-loop control mechanism is implemented inside this function. First, the enabling of RX and TX for the device is needed. This is done through the "XUartPs_WriteReg" function, which allows to write a UART register. Parameters for the function include the base address of the device, the offset from it and the value to be written to the register.

It is beneficial to explain the operating principle of the open-loop control. The user operates the motors by running the program and interfacing with it through the terminal. Sending a 9-digit long string of characters is necessary, which is then partitioned and fed into the relevant functions later in the main function. The first four digits correspond to the distance to be travelled by the motor in terms of positive sensor edges, next digit is to set the direction of the motors, the three digits after that are usedto set the PWM duty cycle and therefore the operation speed of the motor, and the last digit is used to choose which motor to control. As an example; the string “050010500” would result in a distance of 500 positive sensor edges to be travelled in forward direction with a duty cycle of %50 by the second motor.



```
PMOD Example With UART Started

Enter the Distance and Direction for PMod Motor

Format is aaaadsssm where a = distance, d = direction, s = speed and m = motor
```

Figure 3.8 – Screenshot of the program prompting the user for input

Therefore, an array for the storage of these characters is defined, as well as integer values for the number of sensor edges (incorrectly named as angle in the code), speed (duty cycle percentage) and to store the total distance travelled to be printed upon the termination of the program. The user is briefly instructed on the input format and prompted to type their command. Here, the program waits until data is receive in the receiver or the FIFO. This is made possible through the “XUartPs_IsReceiveData” function, which takes the base address of the device as a parameter and returns true if data is received. Then, a UART register is read, this time using the function “XUartPs_ReadReg”, which again takes the base address of the device and the offset from it. Afterwards, the 9 characters that were put in by the user are stored in the data array. Following this, the values entered by the user are printed on the screen for each attribute to be set. The array is read through and the values derived are fed into the functions relevant to the operation of the motor.

The speed is set through the use of “DHB1_setMotorSpeeds” function. It sets the duty cycle for one or both motors’ enable signals. A value of 0 turns off the motors while the maximum speed is achieved at the value 100. Therefore, it takes a pointer to a PmodDHB1 object to set motor speeds for and two 8-bit numbers between 0 and 100 to set the speed of individual motors. Afterwards the cumulative position counters for both motors are cleared. Following this the motors are disabled and the program sleeps for six microseconds. The disabling of PWM before changing directions is necessary, otherwise the motor driver can short circuit. This is touched more in detail in the results chapter.

```
/**************** Setting the Direction for the Motors *****************/

DHB1_motorDisable(&pmodDHB1); // Disable PWM before changing direction,
usleep(6);                    // short circuit possible otherwise

if(data[4]=='1')
{
    xil_printf("Forward\r\n");
    DHB1_setDir(mAddr,1); // Set the direction forward for relevant motors
}
else if (data[4]=='0')
{
    xil_printf("Backwards\r\n");
    DHB1_setDir(mAddr,0); // Set the direction backwards for relevant motors
}
```

Following the sleep, the direction pins of the motors are set, as in the code above. The LSB of dir1 and dir2 determine the value of the DIR1 and DIR2 pins, which are the arguments this function takes. Since now all the attributes of the motor are set, the “drive” function is called and the motor starts it’s operation. This is done through first enabling the motor, then calling the “MotorFeedback_getDistanceTraveled” function, which returns the distance travelled by the motors in terms of positive sensor edges. Once the sensed edges equal to the value that was written to the terminal by the user, the position counter is cleared and the motor is disabled. Multiple functions related to the motor feedbacks have been implemented by the manufacturer, which can be examined more in detail by going through the source code. This process can be repeated multiple times and the distance value put in by the user is summed up to have the total distance travelled, which can be acccessed by terminating the program by pressing “ESC”, in which case the value will be printed on the terminal.

A more detailed description of the capabilities, design challenges and possible future improvements to be made related to the implementation is given in the next chapter.

4 Results

In this chapter, we will present our findings through this project. We will first introduce the numerical results attained from experimenting with the INS implementation. We will then present the testing results from our pulse width modulator.

4.1 INS

The performed tests on the gyroscope and the accelerometer were carried out in both stationary and moving states. The presented results here are all based on the parameter settings explained in subsection "Decisions made for the compromises". We will present the results we achieved by compensating for the gyro drift in the stationary state and during device movements, and compare them to measurements without calibration and error compensation.

We will then test the error accumulation of the accelerometer using its built-in calibration, and without using the calibration. We will close this section by a number of observations and notes for the future developers.

4.1.1 Experiments with Gyroscope

We will now demonstrate the improvements in the accuracy of the angle calculation with and without moving the device.

Gyroscope in Stationary State

We first present the results of the angle calculation without performing any calibration or error compensation. As can be seen from Figure 4.1, the drift problem can cause up to 28 degrees miscalculated angle in the span of only 26 seconds.

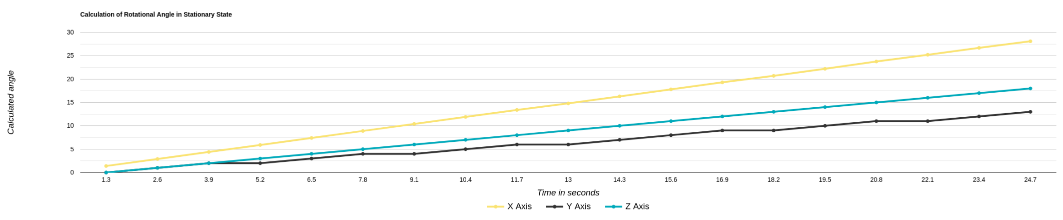


Figure 4.1 – Calculated rotational angle without calibration in the stationary state

On the other hand, as it can be seen in Table 4.1, using our method to calibrate the readings from the gyroscope proves to almost fully nullify the effects of the gyro drift problem, amounting to only 6.8 wrongly calculated rotational angle in the X axis, while not registering any significant rotation in the y axis.

Time	X-Axis	Y-Axis	Z-Axis
1.3	0.306772	0.081331	0.146385
2.6	0.612680	0.134737	0.300550
3.9	0.918550	0.185822	0.434058
5.2	1.276507	0.249659	0.529894
6.5	1.655786	0.294623	0.644666
7.8	1.983703	0.345298	0.808304
9.1	2.330330	0.397884	0.888575
10.4	2.678812	0.472572	1.046163
11.7	3.009014	0.531569	1.187286
13	3.373641	0.588263	1.351040
14.3	3.688546	0.648505	1.532278
15.6	4.077023	0.662518	1.657290
16.9	4.450572	0.729644	1.785556
18.2	4.819099	0.768670	1.985815
19.5	5.156553	0.792694	2.129026
20.8	5.480494	0.839537	2.237736
22.1	5.813023	0.903610	2.354544
23.4	6.150849	0.956164	2.508614
24.7	6.476818	1.030433	2.660892
26	6.812810	1.103925	2.833557

Table 4.1 – Calculated rotational angle with calibration in the stationary state

Moving the Gyroscope

We now present the angle calculation of the device while moving the device, as our primary goal for this project was to estimate the rotational angle of the device at all times. The standard method of testing the angle calculation in a gyroscope is using a rate table. In this way, the system's response to known speeds as well as slight rotations can be tested [RAL15].

As we did not possess a rate table during this project, the function of our system was tested using rotating the device by hand for 90 degrees around the x,y, and z axes. We performed each rotation in intervals of 5.8 seconds, and logged the response of the system to each rotation during 30 seconds, in order to test the accumulated error in angle estimation. Our results are summarized in Table 4.2.

	Calculated Rotation			**Real Rotation**		
Time	**X**	**Y**	**Z**	**X**	**Y**	**Z**
5.8	5.151950	86.235	1.385212	0	90	0
11.6	8.861076	-85.111	-1.856076	0	-90	0
17.4	3.891506	0.635750	93.729378	0	0	90
23.2	11.608211	0.819621	-86.414750	0	0	-90
29	103.071487	0.302261	14.613131	90	0	0
34.8	-89.143015	2.594546	6.430927	-90	0	0
40.6	19.428915	91.986954	11.206043	0	90	0
46.4	23.527397	-88.078202	6.959701	0	-90	0
52.2	18.674654	3.658420	102.760483	0	0	90
58	26.873615	3.804205	-93.087812	0	0	-90
63.8	118.211166	0.344664	16.912405	90	0	0
69.6	-88.357311	4.528293	7.917015	-90	0	0
75.4	30.819271	89.194939	12.855237	0	90	0
81.2	36.261730	-85.898158	9.773219	0	-90	0
87	31.197594	5.477262	105.640175	0	0	90
92.8	39.265388	5.753466	-90.224640	0	0	-90
98.6	131.873795	4.923436	21.561419	90	0	0
114.4	-89.836742	7.234188	14.754968	-90	0	0

Table 4.2 – Calculated rotational angle with calibration while moving the device

One important observation that can be made from our experiments with both calibrated and non-calibrated measurements, is that the calculated rotational angle in the clockwise rotation along all the axes provided more accurate results. This is due to adding negative numbers to the previously calculated angle, which would cancel out some of the accumulated errors in the made calculations.

As expected, the error accumulation using a custom calibration is far less than the error accumulation with no calibration done. This can be seen in Figure , where the calculated rotational angles for calibrated and non-calibrated measurements after three consecutive rotations in 90 degrees along the x axis are shown.

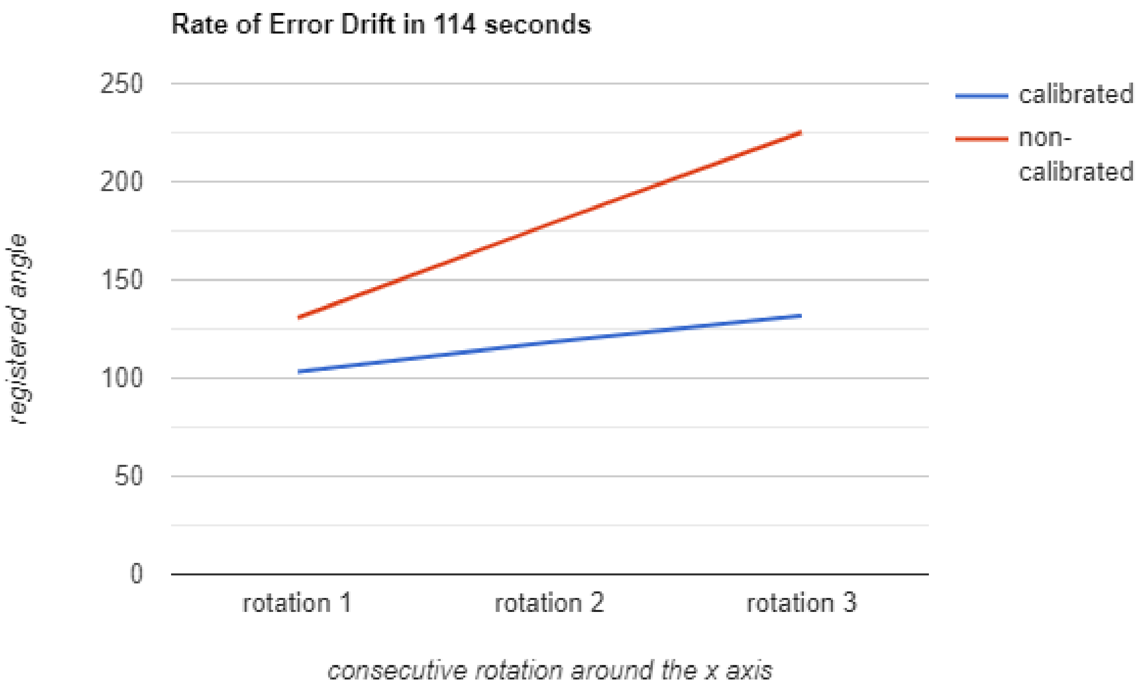


Figure 4.2 – Calculated rotational angle with and without calibration after three consecutive rotations around the x axis

4.1.2 Experiments with Accelerometer

As outlined in section 3.3.2, we also performed experiments using the accelerometer. The aim of these experiments was to measure the measurement errors while relying on the built-in calibration of the PMOD ACL1 device. We logged the erroneous traveled distance calculated without moving the device using calibrated and non-calibrated readings in the span of nine minutes. Figure 4.3 compares the logged traveled distance along the X axis for calibrated and non-calibrated measured accelerations.

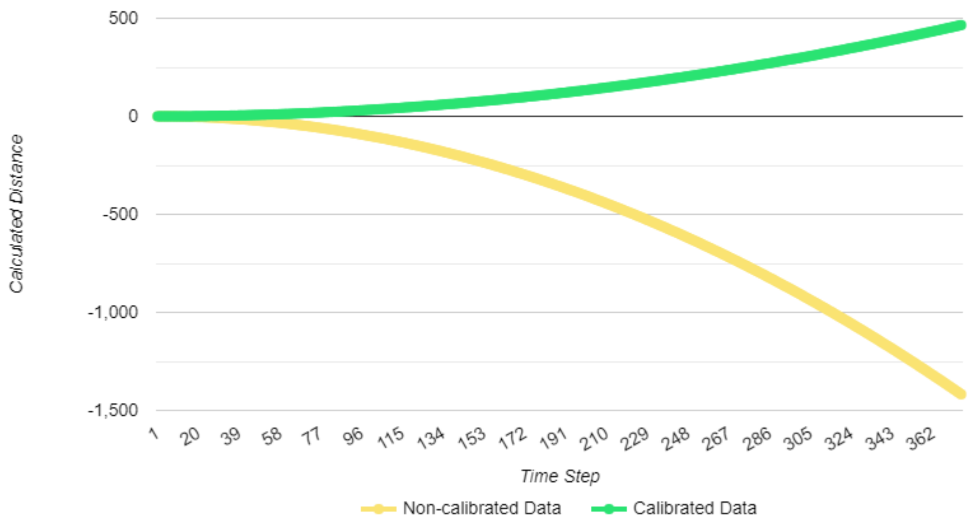


Figure 4.3 – Calculated distance with and without calibration along the X axis

It is evident that the built-in calibration function of the accelerometer can lower the error drift problem, but a more advanced, custom-made calibration mechanism will prove more effective.

Our experiments with the accelerometer and the gyroscope proved the importance of calibrating the data before integrating it, in order to attain the desired measurements. Nevertheless, as pointed out in section 2.1.4, a number of more advanced algorithms and additional devices are necessary, in order to receive reliable data from our inertial measurement unit.

4.2 PWM

In its current state, the resulting PWM and open-loop control implementation is capable of setting the speed, direction and distance to be traveled of the motors as defined by the user. The user can also set the directions for the motors separately. Of course, through the development process certain tests had to be made to make sure of accurate functionality. While the implementation details are already talked about in chapter 3, this section describes the tests made with the implementation and their results, certain challenges that were faced, some things that did not work and other aspects that can be improved.

The testing of the functionalities were actually started while the implementation was still ongoing, and the first of such tests were to enable and disable the pins on the PMOD DHB1 in order to see if the desired functionality is observed. External power applied to the PMOD was within operational parameters, and all the observations were made at room temperature. The results were written in table 4.3, which is identical to the results that should be expected, showing that the enabling of the pins is done correctly and the motor can be started and stopped, or moved in either direction, depending on the input from the user.

DIR1	EN1	Result		DIR2	EN2	Result
0	0	Stop		0	0	Stop
0	1/PWM	Forward		0	1/PWM	Forward
1	0	Stop		1	0	Stop
1	1/PWM	Reverse		1	1/PWM	Reverse

Table 4.3 – Testing of the pins for accurate results

Certain testing was then done related to the PWM, in which various duty cycles between 0 and 100 were inputted through the interface. When the duty cycle is set to 0, the motor naturally does not move. The duty cycle was gradually increased from this value and its effects on the motor were observed. Results show that with our hardware configurations the motor does not move if the duty cycle is set to any value lower than 20%, and only moves at that value if there is an external push to the motor (in this case the push of a finger). This is generally expected behavior, since in a DC motor the average current that passes through the motor is proportional to the torque, which was described in chapter 2. A decrease in the duty cycle also results in a drop in the current. As the duty cycle is decreased, this eventually brings the system to a point where the friction cannot be overcome by the torque and therefore no movement can be observed. As there is no back electromotive force (EMF), the current is slightly higher when the motor is not rotating, though the static friction is also higher. This allows for the motor to operate in a certain range if there is an additional push. For any duty cycle value above 20%, the motor operated correctly and maximum speed is achieved at 100%. It should be noted that before any change in motor specifications are to be made, the motor has to be disabled in order the prevent the occurrence of short circuits.

While various different sleeping times were tested to see how fast the change in movement can be made, it was never beyond the minimum amount that was defined by the manu-

Duty Cycle	Result
<20%	No Movement
20%	Movement with additional push
>20%	Movement as expected

Table 4.4 – Motor behaviour based on duty cycle percentage

facturer, as there was only one such PMOD available for testing and the destruction of it would negatively impact any further steps. The disabling, sleep and enabling of the motor for movement with different speficiations took approximately 0.4 seconds.

There were also certain challenges faced while implementing the user interface. The built-in SDK terminal issues an extra character when the "Send" button is pushed in order to send our input string, which caused unexpected behavior. This problem should not occur again if the code is modified in the future. However, if other terminal emulators are used, a different unexpected behavior may occur and attention should be paid to this. Below is a snippet of the code that reads from the user input and compensates for this issue:

```
if(RecvChar==13) // Condition to control terminal carriage character
{
    count=0;
}
else if(RecvChar !='\n' && count <9 ) // Prevent extra \n character issue
 and store 9 characters
{
    data[count]=RecvChar; // Store each recieved character in a data array
    count++;
}
```

The code allows for the individual setting of directions for either motor, but due to the way the driver functionality is implemented, it is not possible to set speed and distance values separately for either motor. It is possibly implemented in this way by the developers as the two motors attached to the driver are meant to operate at the same speed and go the same distance. The implementation can be adapted to any hardware configuration provided it uses the same PMOD. However, the macros defined in this implementation are for the motor used in our project, and using the wrong specifications may cause minor differences in behavior.

Assuming the input is entered in the expected format, the motor operates correctly. However, exception handling and error messages were not included in this implementation, therefore inputting in the wrong format will be ignored by the program. Additionally, while there is no theoretical limit to the distance that can be traveled by the motor, it was deemed enough to have it up to 4-digits long in the implementation, which can be easily modified in the code should the user want to test behavior for longer distances. Another idea can be the addition of an interrupt function, as right now when a command is issued by the user and the motor starts execution, there is no way to interrupt the process except to wait for the sensor to produce the required amount of edges, after which the motor is disabled. While the interface does possess the basic necessary functionalities, further improvements are possible to make it more intuitive and user friendly, which would allow for more efficient operation of the motors.

The overall process and lessons learned through the implementation can therefore be summed up as: doing the relevant background research and familiarizing oneself with the necessary concepts of embedded systems such as motor controllers, PWM, UART and control loop mechanisms, getting familiar with and configuring embedded hadware and software (Vivado Design Suite), learning the functionalities of manufacturer written code for specific hardware and the creation of new software using them. Additionally testing and documentation skills were considerably employed and improved.

5 Conclusions

In this research, we examined the need for inertial navigation systems, and discussed their limitations and explained the sources of errors in these systems, along with suggesting a few methods to compensate for these errors.

We performed experiments on a gyroscope and an accelerometer module, in order to estimate the error drift in these devices. Furthermore, we implemented a calibration method, in order to reduce the bias-based errors in these devices. Finally, we designed and implemented a PWM program, which controlled the speed of the rotation of the vehicle's wheels, based on the desired input by a user.

We gathered valuable experience about working with the "Vivado Design Suit", the "Vivado Software Development Kit", and the programming procedure for an FPGA, specifically the Zybo z7 board.

List of Abreviations

EMF	Electromotive Force
FIFO	First In First out
FPGA	Field Programmable Gate Array
GPIO	General-purpose Input/Output
INS	Inertial Navigation System
LSB	Least Significant Bit
PMOD	Peripheral Module
PWM	Pulse-width Modulation
SPI	Serial Peripheral Interface
TX & RX	Transmit and Receive
UART	Universal Asynchronous Receiver-transmitter

List of Tables

2.1 Error Types and their consequences in inertial navigation systems 9

4.1 Calculated rotational angle with calibration in the stationary state 35
4.2 Calculated rotational angle with calibration while moving the device 37
4.3 Testing of the pins for accurate results 40
4.4 Motor behaviour based on duty cycle percentage 41

List of Figures

2.1 A capacitative accelerometer 4
2.2 The limitations of INS-based navigation systems 7
2.3 Structure of an H-bridge circuit 11
2.4 Representation of speed and torque based on changes in the DC Motor polarity 12
2.5 Depiction of a pulse wave with the values y_{min}, y_{max} and D 14
2.6 Block diagram for a UART 15
2.7 Block diagram of an open-loop control system 16

3.1 Data sheet of the motor 18
3.2 Picture of the hardware setup with all the components attached 19
3.3 Image of the final version of the block design, with all the used PMODs included 20
3.4 Design Overview of the INS implementation 22
3.5 PuTTY serial configurations 23
3.6 The Average Readings from the Gyroscope with Respect to Buffer Size . . 29
3.7 Calibration Time with respect to Buffer Size 29
3.8 Screenshot of the program prompting the user for input 32

4.1 Calculated rotational angle without calibration in the stationary state . . . 35
4.2 Calculated rotational angle with and without calibration after three consecutive rotations around the x axis 38
4.3 Calculated distance with and without calibration along the X axis 39

Bibliography

[Bar01] Barr, Michael: Pulse width modulation. In: *Embedded Systems Programming* 14 (2001), Nr. 10, S. 103–104

[BD99] Bishop, Robert H. ; Dorf, Richard C.: Teaching modern control system design. In: *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No. 99CH36304)* Bd. 1 IEEE, 1999, S. 364–369

[Bri71] Britting, Kenneth R.: *Inertial navigation systems analysis.* Wiley-Interscience, 1971

[COKK07] Cho, Kyu M. ; Oh, Won S. ; Kim, Young T. ; Kim, Hee J.: A new switching strategy for pulse width modulation (PWM) power converters. In: *IEEE Transactions on Industrial Electronics* 54 (2007), Nr. 1, S. 330–337

[FH18] Fabian Hanke, B. S.: *Ego and Object Motion Estimation.* Munich, Bavaria, Germany, Technical University of Munich, Diplomarbeit, June 2018

[GRKR20] Gupta, Ashok K. ; Raman, Ashish ; Kumar, Naveen ; Ranjan, Ravi: Design and implementation of high-speed universal asynchronous receiver and transmitter (UART). In: *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)* IEEE, 2020, S. 295–300

[Her12] Herman, Stephen L.: *Understanding Motor Controls.* Cengage Learning, 2012

[Joh09] Johns, Kenneth A.: *Inertial Navigation Systems.* University Lecture, 2009

[Kel94] Kelly, Alonzo: *Modern Inertial and Satellite Navigation Systems.* Carnegie Mellon University, 1994

[KIN98] KING, A. D.: Inertial Navigation â€“ Forty Years of Evolution. In: *GEC REVIEW* 13 (1998), Nr. 3, 140–149. http://www.aerostudents.com/courses/avionics/InertialNavigationSystems.pdf

[KKC03] Kwasinski, Alexis ; Krein, Philip T. ; Chapman, Patrick L.: Time domain comparison of pulse-width modulation schemes. In: *IEEE Power Electronics Letters* 1 (2003), Nr. 3, S. 64–68

[KS98] King, A. D. ; Sc, B.: Inertial Navigation â€“ Forty Years of Evolution. In: *GEC Review* (1998), S. 140–149

[LPSp08] Landry, Chris ; Papasideris, Kosta ; Sutter, Brad ; pwls, Archie W.: Inertial Navigation Systems: The Physics behind Personnel Tracking and the ExacTrak System, 2008, S. 300–303

[LVL+04] Loh, Poh C. ; Vilathgamuwa, D M. ; Lai, Yue S. ; Chua, Geok T. ; Li, Yunwei: Pulse-width modulation of Z-source inverters. In: *Conference Record of the 2004 IEEE Industry Applications Conference, 2004. 39th IAS Annual Meeting.* Bd. 1 IEEE, 2004

[MD17] Marchthaler, Reiner ; Dingler, Sebastian: *Kalman Filter.* Springer-Vieweg, 2017

[Nic16] Nichols, Timothy A.: *Propagation of Sensor Noise in Navigation Equations and High Accuracy Dynamic Calibration of Sensors Accuracy Dynamic Calibration of Sensors.* Rochester, Newyork, United States of America, June 2016

[NP16] Nanda, Umakanta ; Pattnaik, Sushant K.: Universal asynchronous receiver and transmitter (uart). In: *2016 3rd international conference on advanced computing and communication systems (ICACCS)* Bd. 1 IEEE, 2016, S. 1–5

[Ped16] Peddapelli, Satish K.: *Pulse Width Modulation.* De Gruyter Oldenbourg, 2016

[PLC12] Peng, K.-Y ; Lin, Cheng-An ; Chiang, Kai-Wei: The performance analysis of an AKF based tightly-coupled INS/GPS integrated positioning and orientation scheme with odometer and non-holonomic constraints. In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 39 (2012), 08, S. 481–486. http://dx.doi.org/10.5194/isprsarchives-XXXIX-B7-481-2012. – DOI 10.5194/isprsarchives–XXXIX–B7–481–2012

[PM08] Priyadarshan, Pradosh ; Mundari, Biswa R.: *Serial communication using uart*, Diss., 2008

[PTNG07] PATEL, Himanshu ; TRIVEDI, Sanjay ; NEELKANTHAN, R ; GUJRATY, VR: A robust UART architecture based on recursive running sum filter for better noise performance. In: *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)* IEEE, 2007, S. 819–823

[RAL15] RAO, MANDADI ; ANANDA, C.M. ; L.R.MANOHAR: Design of Single - Axis Rate Table for calibration of Gyro sensor used in Micro Aerial vehicles (MAV). In: *International Journal of Emerging Technology in Computer Science 'I&' Electronics (IJETCSE)* 14 (2015), 04, S. 765–769. – ISSN 0976–1353

[ST15] SHUKLA, Sandeep K. ; TALPIN, Jean: *Synthesis of Embedded Software: Frameworks and Methodologies for Correctness by Construction.* Springer Science & Business Media, 2015

[STM10] STMICROELECTRONICS: *l3g4200d MEMS motion sensor:ultra-stable three-axis digital output gyroscope.* https://digilent.com/reference/_media/reference/pmod/pmodgyro/stmicroelectronics-l3g4200d-datasheet.pdf, 2010. – l3g4200d gyroscope datasheet

[Woo07] WOODMAN, O.: Technical Report: An introduction to inertial navigation / Computer Laboratory, University of Cambridge. Version: August 2007. http://www.cl.cam.ac.uk/techreports. 15 JJ Thomson Avenue, Cambridge CB3 0FD United Kingdom, August 2007 (696). – Forschungsbericht. – 37 S.

[WP09] WIDNALL, S. ; PERAIRE, J.: *Inertial Instruments and Inertial Navigation.* University Lecture, 2009

[XF08] XU, Fan ; FANG, Jiancheng: Velocity and position error compensation using strapdown inertial navigation system/celestial navigation system integration based on ensemble neural network. In: *Aerospace Science and Technology* 12 (2008), Nr. 4, 302-307. http://dx.doi.org/https://doi.org/10.1016/j.ast.2007.08.005. – DOI https://doi.org/10.1016/j.ast.2007.08.005. – ISSN 1270–9638