République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ÉCOLE NATIONALE POLYTECHNIQUE

DÉPARTEMENT D'ÉLECTRONIQUE

Projet de Fin d'études

En vue de l'obtention du diplôme d'ingénieur d'état en électronique

Thème :

Development of a Rapid Prototyping Platform for IMU applications via MATLAB/Simulink on ZedBoard

Encadré par :

Dr. Rabah SADOUN

Réalisé par :

Smail BECHICHE

Hacene BOUAROUA

Juin 2015

Ce modeste travail est dédié à:

Mes chers parents

Mes chers frères et mes deux chères sœurs

La mémoire de mon grand-père Yahia

Mon grand-père Mohamed

Mon beau-frère Rostom

Toute ma grande famille

Mes amis Salah, Nadir, Hacene, Mohamed, Amine, Hmidou, Brahim, Bilel et Adem

Tous ceux qui me sont chers…

B. Smail

Je dédie ce travail à:

Mes chers parents

Mes deux chers frères Salah et Ahmed et chère sœur Menna

Mes chers oncles Kacem et Salah et leurs familles

Toute la grande famille

Tout mes amis Marouane, Brahim, Farouk, Tarek, Khaled, James et Sadeddine

B.Hacene

Remerciements

En premier lieu, nous tenons à remercier Dieu, de nous avoir aidé et de nous avoir donné la patience et la foi de finaliser ce mémoire.

Nous exprimons toutes nos reconnaissances à Monsieur Rabah SADOUN, notre encadreur; pour sa patience, la qualité de ses conseils et sa serviabilité.

Nous tenons à remercier M. Ahmed BOUZID qui a consacré beaucoup de son temps pour nous aider, M. Salah BOUHOUN pour son support et ces précieux conseils et aussi les camarades avec lesquels nous avons travaillé pour la période du projet, particulièrement Sami BENABIDELLAH et Sid Ali DIOUANI.

Nous adressons nos sincères remerciements à tous les membres du Jury chargé d'examiner la soutenance de notre projet de fin d'études.

Nous voudrons exprimer notre profond respect à tous les Enseignants de l'École Nationale Polytechnique pour la formation qu'ils nous ont donnée.

Finalement, nous remercions tous ceux qui n'ont épargné aucun effort, de près ou de loin, pour nous permettre d'accomplir ce modeste travail.

ملخص:

الهدف من هذا العمل هو تطوير قاعدة نمذجة سريعة لتنفيذ نظم الملاحة العطالية ، وهذا باستعمال طريقة Model-Based Design . استعملنا أدوات تطوير النماذج MATLAB/Simulink لإنشاء و محاكاة و تحميل مشاريع تصميم النظم المضمنة. نستهدف كنظام SoC Zynq -7000 لشركة Xilinx المتواجدة في بطاقة التطوير ZedBoard.

نعطي أيضا كمثال لتطبيقات هذه القاعدة تجريب خوارزمية نظام مرجعي لزوايا الموقف والتوجيه (AHRS): فلتر التوجيه بحساب نزول الميل (GDOF) باستعمال قياسات الوقت الحقيقي من خلال حسّاس وحدة قياس عطالي MPU6050 (IMU) موصول بالبطاقة ZedBoard.

كلمات مفتاحية: (نظم الملاحة العطالية ، حسّاس عطالي ، Model-Based Design، Simulink ، Xilinx ، Zynq، Zedboard ، بناء نظام Linux ، محاكاة في وقت حقيقي ، PIL، SIL ، AHRS ، INS ، IMU، GDOF)

Abstract:

The aim of this work is the development of a rapid protoyping platform for the implementation of inertial navigation systems, using Model-Based Design method. MATLAB/Simulink represent the software development tool where the design projects are created, simulated and deployed on embedded systems. We will be targeting the Xilinx Zynq-7000 SoC equipping the ZedBoard development board.

An illustration of the platform's applications was demonstrated by testing an Attitude and Heading Reference System (AHRS) algorithm: the Gradient Decent Orientation Filter (GDOF) using real time measurements from the MPU6050 IMU (Inertial Measurement Unit) sensor interfaced to the ZedBoard.

Keywords (Inertial systems, inertial sensors, Model-Based Design, Simulink,Xilinx Zynq, Zedboard , Linux compilation, real time simulation, PIL, SIL, AHRS, INS, IMU ,GDOF)

Résumé:

Le but de ce travail est le développement d'une plateforme de prototypage rapide pour l'implémentation des systèmes de navigation inertiels et ce, en utilisant la méthode du Model-Based Design. MATLAB/Simulink représente l'outil de développement du modèle où les projets de conceptions sont créés, simulés et chargés dans des systèmes embarqués. On vise comme système le SoC Zynq -7000 du Xilinx équipant la carte de développement ZedBoard.

Une illustration des applications de la plateforme est démontrée en testant un algorithme pour système référentiel d'attitude et direction (AHRS): le Filtre d'Orientation à Gradient de Descente (GDOF) en utilisant des mesures en temps réel depuis une Unité de Mesure Inertielle (IMU) MPU6050 interfacée à la ZedBoard.

Mots clefs (Systemes inertiels, Capteurs inertiels, Model-Based Design, Simulink, Xilinx Zynq, Zedboard , compilation Linux, simulation en temps réel, PIL, SIL, AHRS, INS, IMU ,GDOF)

Contents

List of Figures

Introduction

In this project we intend to create a prototyping platform for testing and simulating new designs and control algorithms in the domain of inertial navigation. This platform is founded on relatively new approach adopted by many researchers and industrials called Model-Based Design. As well-known, Simulink provided by Mathworks is an excellent simulation tool to apply, visualize and analyze the performance of algorithms as they are applied to mathematical models of physical systems. Although real world factors such as sensor noise or physical perturbations may also be modeled and introduced into the simulated system, they remain mathematical approximations.

With this new method, Simulink models can be deployed and tested simultaneously in evaluation boards and prototyping hardware platforms, which - in our case - is the ZedBoard. We will have the ability to integrate real data signals provided directly form sensors instead of using mathematically modeled ones. We shall also, evaluate the designed algorithm in the target hardware before deciding to produce the final product. Using Model-Based Design, we are going to make a Simulink driver block for an IMU sensor. Once we have readings from the sensor, we exploit them to implement a real time application for an Attitude and Heading Reference System (AHRS);a relatively new decent gradient sensor fusion algorithm.

Chapter one is a general overview of the inertial sensors, systems' technologies and applications, Inertial Measurement Units (IMU) will be the focus of this chapter, since we are going to use them. A brief history and specifications of inertial sensors is given before concluding the chapter.

Chapter two details the Model-Based Design concept adopted by Simulink with explanation of its parts and the tools used to build, simulate and test our model and functional settings for the development hardware utilized in this project and Chapter three introduces the ZedBoard; the target hardware hosting the Zynq-7000 SoC which is used for our application. Also, the IMU sensor and its characteristics were specified in addition the I2C communication protocol used for interconnection between the sensor and the ZedBoard.

In chapter four, the system configuration steps were detailed; showing how we assemble all the necessary elements of the design model, ensuring the communication between the ZedBoard and Simulink in one hand, and between the sensor and the ZedBoard on the other hand. The important part is building a Simulink Block that acts as a driver for the sensor to get its data directly onto the Simulink environment.

Finally, in chapter five we are going to take advantage of this new created platform to apply and simulate an AHRS algorithm using real time data from sensor readings.

1

Inertial systems

In this chapter, general definition and terminologies are discussed in order to introduce inertial sensors and systems, their types and a brief history before concluding with different inertial sensors specifications.

1.1 Inertial Sensors

To determine the dynamic behavior of a moving object, inertial sensors take in consideration inertial forces acting on this object. To do so, acceleration along some axis and the angular rate are the basic dynamic parameters. External forces acting on a body cause an acceleration and/or a change of its orientation (angular position). The angular velocity (angular rate) is the rate of change of the angular position . Inertial sensors are unable to measure a constant velocity of a body that is not exposed to inertial forces, unlike speedometers which are not inertial sensors; if the initial conditions of the body are known, their evolution can be calculated by integrating the dynamic equation on the basis of the measured acceleration and rate signals [1].

Figure 1.1: Example of early gimbaled inertial systems used in missiles [2]

Figure 1.2: LGM-118 Peacekeeper ICBM Inertial Measurement Unit[3].

Accelerations and angular velocities are vectorial signals possessing absolute values and orientations. If only one component of the vector should be measured the sensor is denoted 1D or one-axis. If two or all three components of the acceleration or the rate signal should be captured, the sensor is called a 2D or 3D accelerometer, or a rate sensor.

Inertial forces caused by the input acceleration or rate signal are converted by inertial sensors into some physical changes like the deflection of masses or deviations of stresses; these are captured by transducers and transformed into electrical signals. The electrical signal is subjected to estimation procedures such as linear or nonlinear filtering in order to approach the real input value. The final output represents the calibrated value of the measured acceleration or velocity. Not only electrical output signals are feasible; because, other forms of the output, such as optical signals where optic fibers and lasers are used [1] in exceptional cases as in highly explosive environments.

1.2 Inertial Navigation Systems

Autonomous systems that provide pose information (position, velocity and attitude) based on measurements by inertial sensors and applying the dead reckoning (DR) principle are called inertial navigation systems. DR is the determination of the moving object's current position based on knowledge of its previous position and the sensors measuring accelerations and angular rotations. A first integration of acceleration provides velocity and a second one yields position given specified initial conditions. Once integration is processed, the attitude of the moving body is given by angular rates in terms of rotations and with transformation of navigation parameters from the body frame to the local frame.

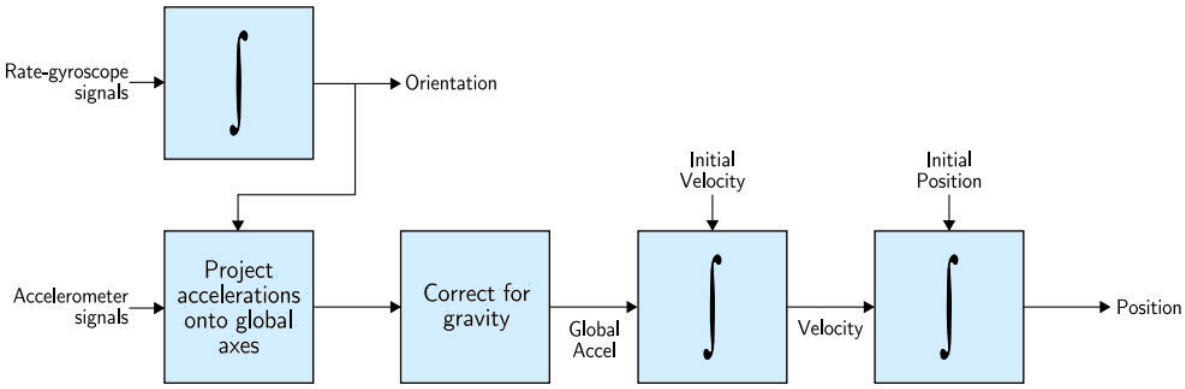

Figure 1.3: A typical inertial navigation algorithm

We can distinguish two main concepts for the determination of orientation: gimbaled systems and strapdown systems. In gimbaled systems, inertial sensors are mounted on a pivoting platform Fig 1.4.a. Rotation of the platform is detected by the gyroscopes to send stabilizing commands to suspension frames of the platform, the platform's orientation is kept inertially stabilized by moving these gimbaled frames.

An alternative principle was used for the Apollo spacecraft [5], spinning inertias can be used to passively keep the platform stable. For example, the orientation of a spacecraft with respect to an initial condition can be determined by measuring the joint angles of the gimbaled frame.

Parts such as gimbals, support bearings, motors, and rotors that need accurate machining and assembly are the building pieces of conventional spinning rotor gyroscopes; these aspects of construction prevent conventional mechanical gyroscopes from becoming

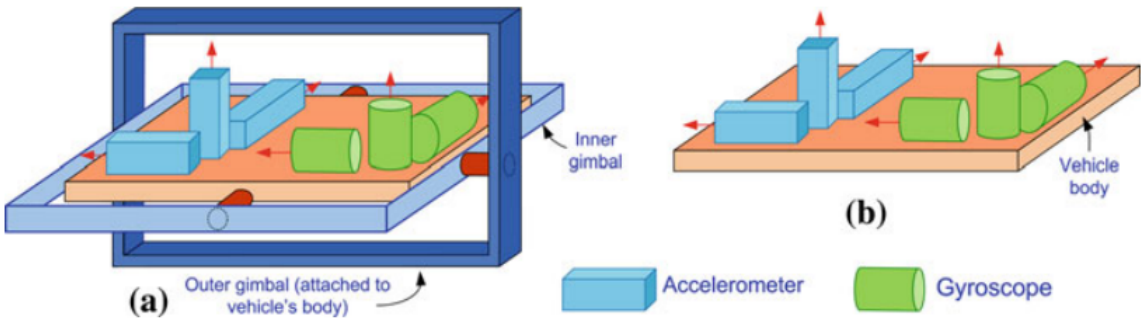

Figure 1.4: [4] Arrangement of the components of (a) gimbled inertial sensor, (b) strapdown inertial sensor.

low-cost devices. Figure 1.2 shows an IMU from the LGM-118 Peacekeeper intercontinental ballistic missile that consists of a platform isolated from vehicle movement by gimbals. Since the platform does not rotate or move with the vehicle, its orientation remains fixed, or inertial, in space. This reference is then used to measure vehicle attitude in flight with high precision. Attitude data is used for turn coordination and steering command guidance by the flight control computers[3].

Gimbaled gyroscopes only meet the performance specifications for a limited number of functioning hours [6] because of wear on the motors and bearings during operation. Therefore strapdown systems are the base of modern navigation, where inertial sensors are body fixed on a vehicle and spinning parts are no longer used(Fig 1.4.b). By knowledge of the current orientation, sensor signals can be converted from the body frame to the navigation frame Fig 1.5. Since the focus goes from hardware to software integration depending on the required accuracy, this principle is suitable for miniaturization of relatively cheaper devices.

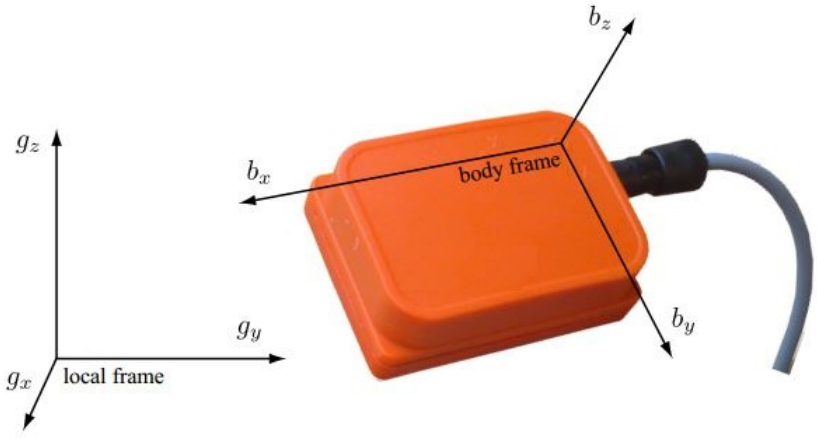

Figure 1.5: Illustration of body and local frames of reference [7]

1.3 Strapdown Inertial sensors

Strapdown systems can be structured with respect to their functional integration, the basic building blocs of a full strapdown navigation system are shown schematically in Fig

1.6. The core is a simple inertial sensor assembly (ISA) with gyroscopes, accelerometers and, if required, magnetometers, with raw and possibly analog output[5].

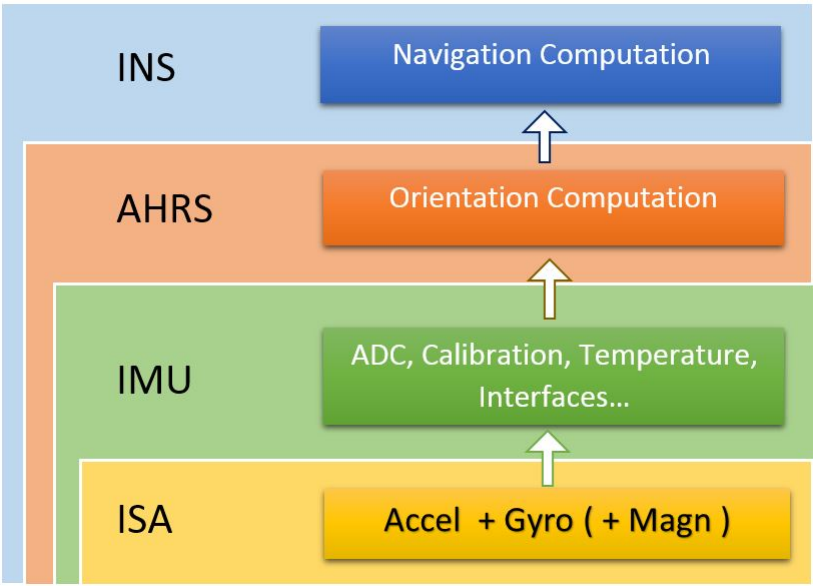

Figure 1.6: The building blocs for an INS

In exchange, in most cases we find a second bloc that includes signal conditionning and Analog to Digital Convertion (ADC) beside other functions as Bus drivers, self-calibration and temperature sensors to form an Inertial Measurement Unit (IMU).

Outputs from the IMU bloc (other sensors output may be fused alongside) can be processed through an Attitude and Heading Reference Systems (AHRS) to provide body orientation or attitude and again through an Inertial Navigation System to provide position with respect to a reference frame.

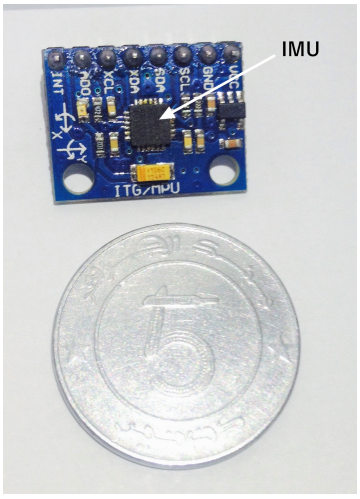

Figure 1.7: MEMS IMU size compared to a coin

Essentially, a strapdown inertial system has the advantage that it is selfcontained; i.e. it does not require external references. An additional advantage is that these sensors have high sampling rates. , strapdown inertial sensors are mainly designed using Micro-Electro

Mechanical Systms (MEMS) because of high miniaturization with relatively, a good precision ,higher integration and lower cost compared to gimbaled systems [4].

1.3.1 Microelectromechanical systems (MEMS)

Microelectromechanical systems (MEMS) are an integration of mechanical elements, sensors, actuators, and electronics on a common infrastructure. MEMS gether silicon-based microelectronics and micromachining technologies on a common platform 1.6. MEMS sensors provide information about the environment by measuring mechanical, thermal, biological, chemical, optical, and magnetic variations. The electronics then process the information derived from the sensors and through a decision-making capability directing the actuators to respond by moving, positioning, regulating, pumping, and filtering, thereby controlling the environment for some desired effect. MEMS are miniaturized, low-cost, low-power, silicon-based sensors. Since they are compact and widely growing products, these sensors are becoming popular in navigation systems of consumer-grade.

Building complete sensor systems including natural sensing elements, transducers, signal-processing blocks, packaging, and testing can nowadays be performed using MEMS technologies. Testing is not apparently related to miniaturization when the interface between the test equipment and the device under test is neglected. Generic micro fabrication technologies, which are promoted to manufacture both integrated circuits and microsystems are anyway, the base of creation for sensor elements and transducers [1].

1.3.2 Inertial Measurement Units

The measurements of the acceleration and the rotation of the vehicle are made by inertial sensors mounted in the inertial measurement unit (IMU). This holds two orthogonal sensor triads, A three-axis triad of accelerometers in addition to an other three-axis triad of gyroscopes.

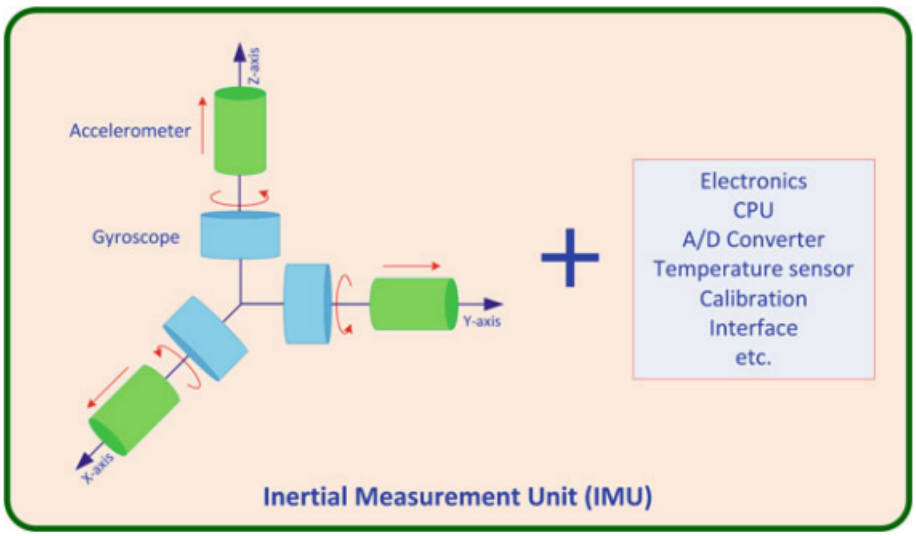

Figure 1.8: The components of a typical IMU [4]

Gyroscopes measure angular motion in three reciprocally orthogonal directions and

accelerometers measure linear motion in three mutually orthogonal directions. The axes of these two triads are parallel, sharing the origin of the accelerometer triad. We mean by body axes or body frame the sensor axes, because they are fixed on the body of the IMU.

The IMU also contains related electronics to perform selfcalibration apart from the inertial sensors, to sample the inertial sensor readings and then to convert them into the appropriate form for the navigation equipment and algorithms. IMUs are also enhanced with analog-digitalconverters (ADC), a processing unit with some prefiltering and suitable interfaces to overlaying systems. Figure 1.8 shows the components of a typical IMU.

1.3.3 MEMS IMUs

Auto industry embraced MEMS inertial sensors in their quest to improve performance, reduce cost, and enhance the reliability of the vehicles (see section 1.4).

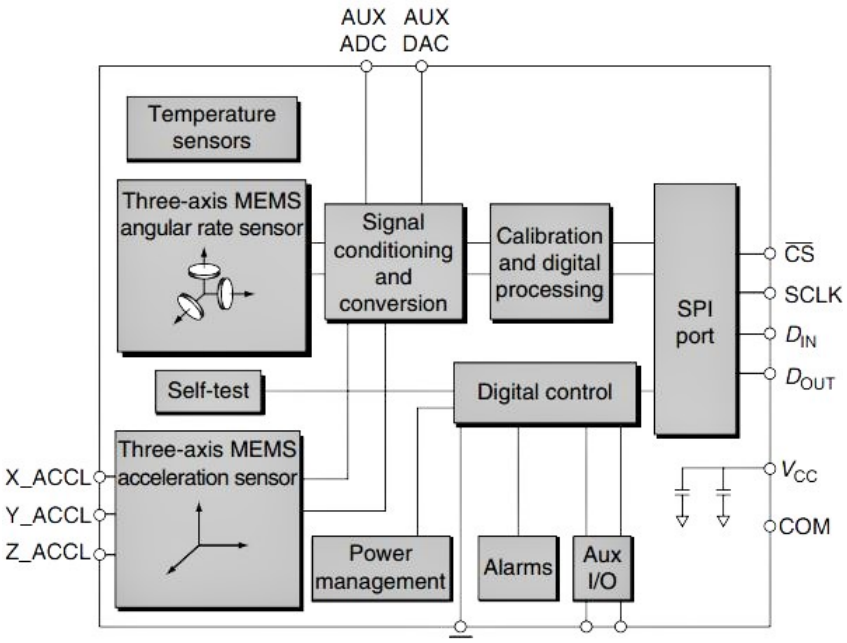

Figure 1.9: Block diagram example of a MEMS-based IMU showing the various interface and functional elements[8].

Nowadays, a MEMS-based accelerometer or gyroscope is considered as a complete product that is packaged, calibrated, and tested, and has to be delivered to the customer, to integrate this component with minimal effort into a higher-level measurement or control system. The meant level of accuracy depends on the application. Also The environmental conditions for the integration of inertial MEMS at the customer site is actually different and usually are divided into classes with respect to the applicable temperature range and the exposure to humidity and aggressive chemicals as well as to shocks and vibration . Whereas low-cost MEMS sensors gives the possibility to new applications, this results a lot of progress has been achieved in the past few years on their robustness. MEMS products are now accepted in high-reliability environments, and are even starting to replace precices Gyros made of Fiber Optic and other technologies in sophisticated applications

[9].

The achieved work in this thesis was based on a MEMS IMU, for the sake of simplicity we'll be referring to it as IMU in short.

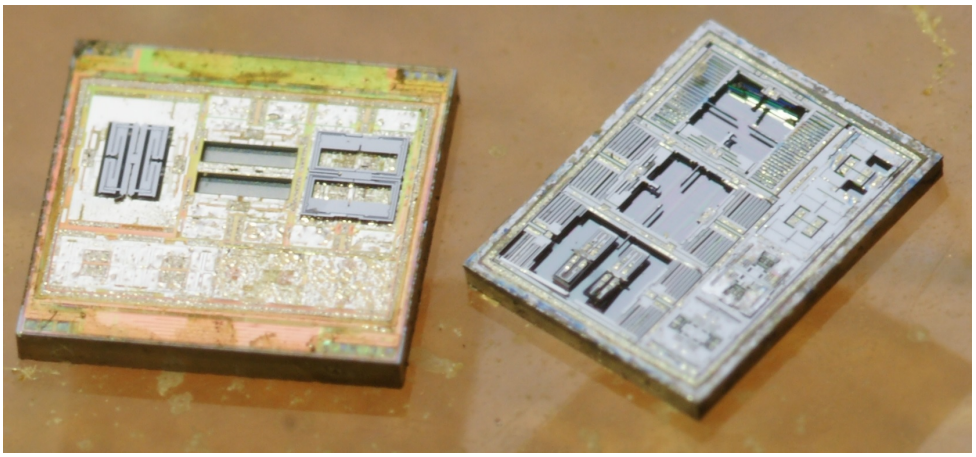

Figure 1.10: Invensense MPU6050 6-axis MEMS IMU teared down [10], the package size is $4 \times 4 \times 0.9$ mm.

1.3.4 Accelerometers

The internal composition of an accelerometer consists of a proof mass, m, connected to a rigid case by a pair of springs as shown in Fig 1.11. inside the case the sensitive axis of the accelerometer is along the spring in the horizontal axis.The application of an accelerated motion to the case will displace the proof mass from its equilibrium position, and the amount of displacement is proportional to the accelerated movement. The displacement from the equilibrium position is sensed by a pickoff and is then scaled to provide an indication of acceleration along this axis.

[4]For example if we have an accelerometer stood on a bench with the sensitive axis along a vertical position, where a presence of the gravitational field. The proof mass will be displaced downward with respect to the case, indicating a positive acceleration. The fact that the gravitational acceleration is downward, and in the same direction as the displacement as shown in Fig 1.12, is sometimes a cause of confusion for the beginners in navigation. The explanation for this lies in the equivalence principle, according to which, in the terrestrial environment it is not possible to separate inertia and navigation by the accelerometer measurement in a single point. Therefore, the output of an accelerometer due to a gravitational field is the negative of the field acceleration. Resulted accelerometer outputs is called the specific force, and is given by:

$$f = a - g \tag{1.1}$$

where
f is the specific force $m.s^{-2}$.
a is the acceleration with respect to the inertial frame.
g is the gravitational acceleration which is about $+9.8\, m.s^{-2}$.

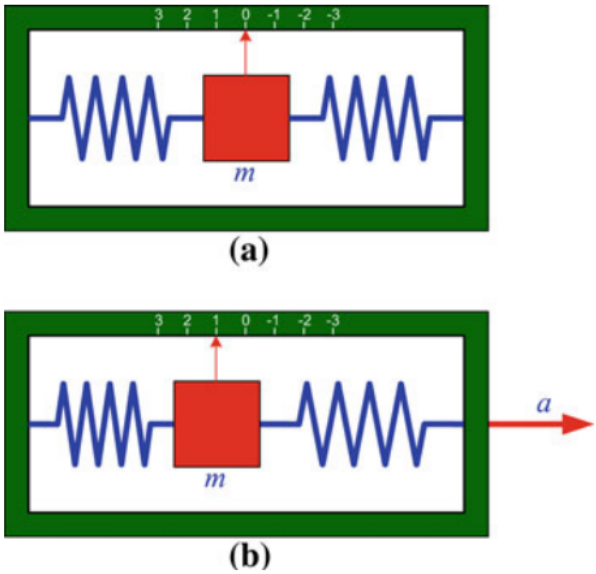

Figure 1.11: a. An accelerometer in the null position with no force acting on it, b. the same accelerometer measuring a linear acceleration of the vehicle in the positive direction (to the right) [4]

[4]This is what causes confusion. The easy way to remember this relation is to think in one of two cases. If the accelerometer is sitting on a bench it is at rest so acceleration a is zero, The force on the accelerometer is the force of reaction of the bench against the case, which is the negative of g along the positive (upward) direction and therefore causes the mass to move downward Fig 1.13. Or imagine dropping the accelerometer in a vacuum. In this case the specific force read by the accelerometer f is zero and the actual acceleration is $a = g$. To navigate with respect to the inertial frame we need a, therefore in the navigation equations we convert the output of an accelerometer from f to a by adding g.

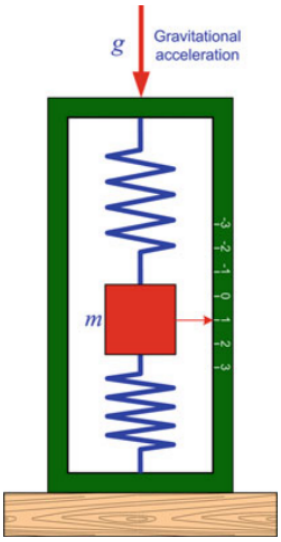

Figure 1.12: An accelerometer resting on a bench with gravitational acceleration acting on it [4]

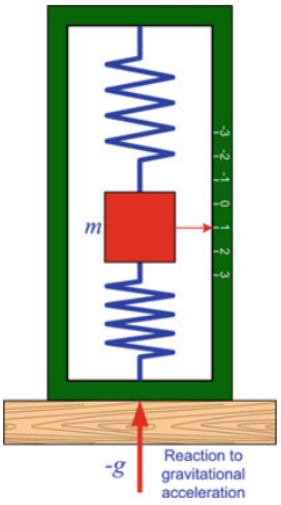

Figure 1.13: An accelerometer resting on a bench where reaction to the gravitational acceleration is acting on it [4]

A common way to classify accelerometers is based on the types of transduction used for converting the mechanical displacement of the proof mass to electrical signal. Some of the common principles are piezoresistive, piezoelectric sensing, optical sensing, and tunneling current sensing, but capacitive sensing accelerometers are the most common ones.

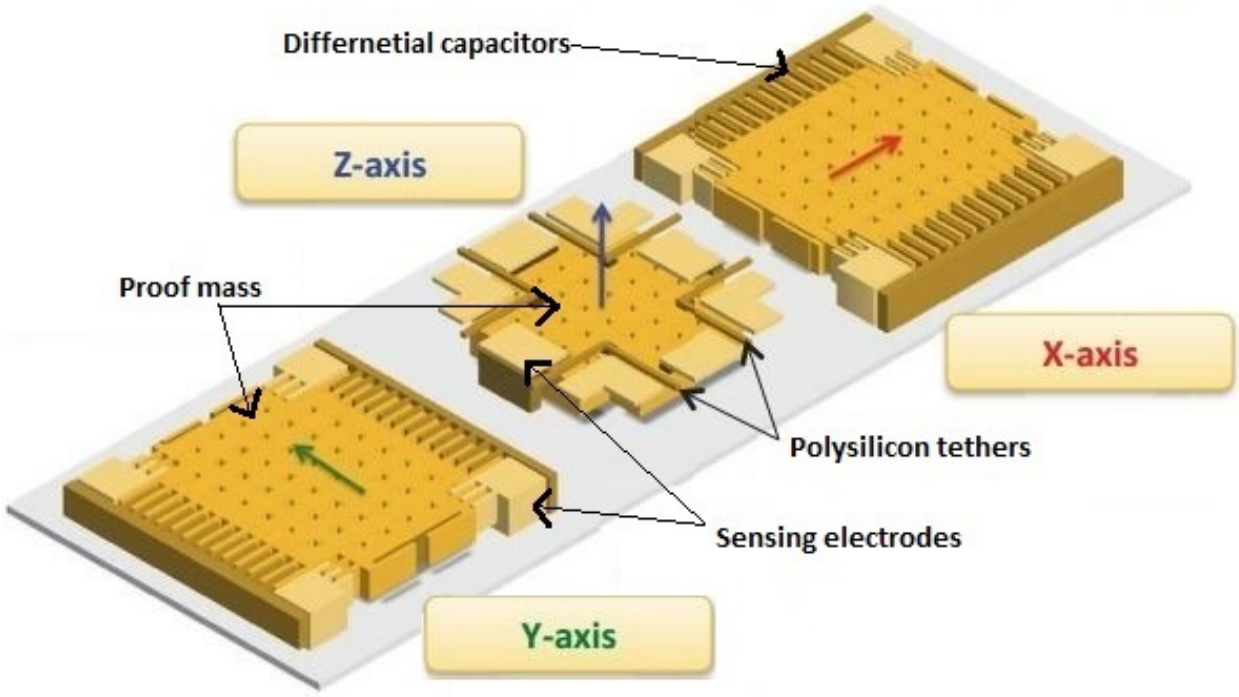

Figure 1.14: Typical structure of a tri-axial capacitive sensing accelerometer [11]

Capacitive accelerometers (Fig 1.14) incorporate an air-damped, opposed-plate capacitor as their sensing element. This mechanism creates a very stable, accurate measurement device, which is inherently insensitive to base strain and transverse acceleration effects. Differential capacitors are formed by using the proof mass as the common contact of the opposite polarity electrodes. On applying external acceleration, the proof mass is displaced, thus changing the capacitance between the proof mass and fixed electrodes. These devices have a high sensitivity, good DC response, low temperature sensitivity, linear output, and low power dissipation along with their simple structure [12].

Key features of MEMS accelerometers include full-range of directional motion, low-g capability, low-current consumption (500 mA), sleep-mode for extended battery use (3 mA), low-voltage operation (2.2–3.6 V), microelectronic interface compatibility, and fast power-up response at 1 ms [8].

1.3.5 Gyroscopes

Rotational motion and translational motion must be measured to fully describe the motion of a body in 3-D space. gyroscopes measure angular rates with respect to an inertial frame of reference. If the angular rates are mathematically integrated this will provide the change in angle with respect to an initial reference angle. Traditionally, these rotational measurements are made using the angular momentum of a spinning rotor. The gyroscopes either output angular rate or attitude depending upon whether they are of the

rate sensing or rate integrating type. It is customary to use the word gyro as a short form of the word gyroscope, so in the ensuing treatment these words are used interchangeably.

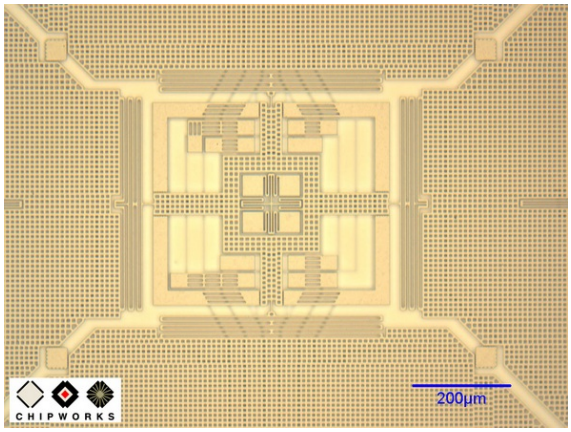

Figure 1.15: the GK10A MEMS die, found in the L3G4200D Gyroscope of STMicroelectronics[13].

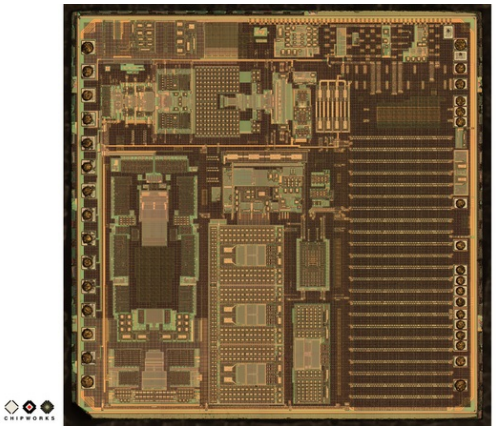

Figure 1.16: The V654A ASIC die converts the capacitive signals from the GK10A MEMS die into a digital signal which is fed into the iPhone 4 [13].

Use of MEMS micromachined technology allows designing of a miniature gyroscope where the rotating disk is replaced with a vibrating element. The design takes advantage of the techniques developed in the electronic industry and is highly suited to high-volume manufacture. Besides, the vibrating gyro is much more robust and can withstand the environments typical of many military and aerospace applications .

[12] gyroscopes use vibrating mechanical elements, to monitor rotational motion. They don't have any rotating part that requires bearings and hence can be miniaturized to reduce the overall cost of the sensor. These vibratory gyroscopes are based on the principle of energy transfer between two vibration modes of a structure caused by Coriolis force as shown in Figure 1.17. In this figure, x^e and y^e represent the x-axis and the y-axis of the Earth's reference frame. Coriolis force is named after a French scientist and engineer G.G. de Coriolis. Coriolis force is an apparent acceleration needed to hold Newton's laws of motion in rotating reference frames. Sometimes this force is called a fictitious force (or pseudo force), because it does not appear when the motion is expressed in an inertial frame of reference. Due to the effect of the Coriolis force, a body moving from point A to B on the surface of the Earth will follow a curved line instead of a straight line as demonstrated in Figure 1.17. At a given rate of rotation of the observer, the magnitude of the Coriolis acceleration is related to the body velocity, the angle between the direction of motion of the body, and the axis of rotation as given by Equation 1.2.

$$\vec{a_c} = -2\vec{\omega} \times \vec{v} \tag{1.2}$$

where $\vec{v}$ is the body velocity in the rotating frame and $\vec{\omega}$ is the angular velocity of the rotating system.

Equation 1.2 implies that the Coriolis acceleration is perpendicular to both the direction of the velocity of the moving mass and the rotation axis. All vibrating gyroscopes

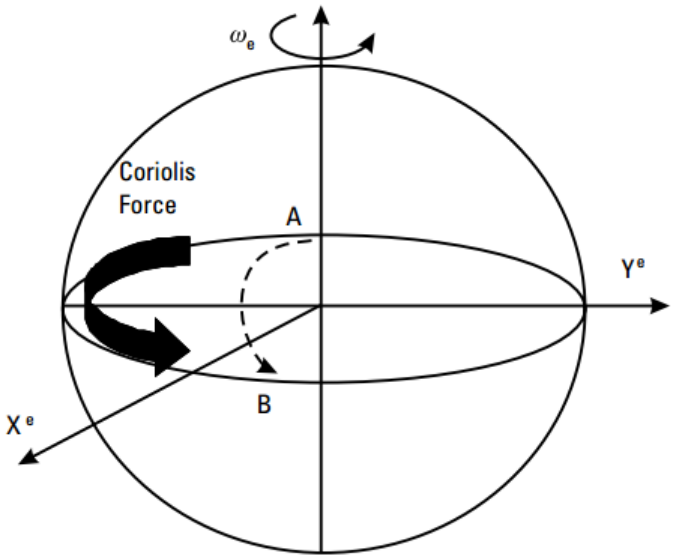

Figure 1.17: Coriolis force [12]

employ the Coriolis effect for measuring the angular rates.

There are several practical ways to build a vibrating gyro, however, all of them can be divided into three principle groups [6]:

1. Simple oscillators (mass on a string, beams)
2. Balanced oscillators (tuning forks)
3. Shell resonators (wine glass, cylinder, ring)

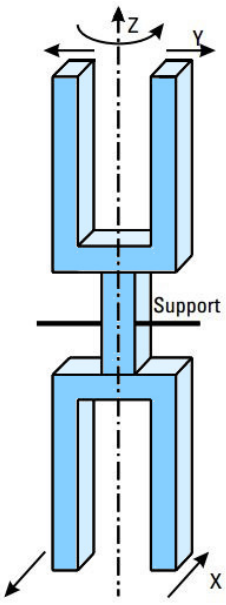

Figure 1.18: Tuning vibratory gyroscope.

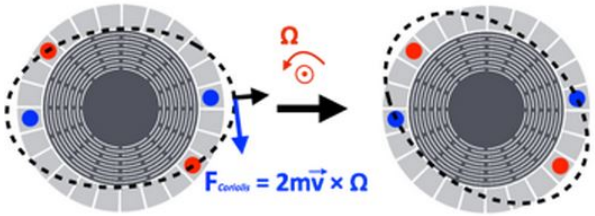

Figure 1.19: Principle of the wine glass gyroscope[14].

The tuning fork consists of two tines, which are connected together at the junction as shown in Figure 1.18. The tines are resonated differentially along two axes by a suspension system. When this system is rotated, Coriolis force is generated, which causes a differen-

tial sinusoidal force to develop on the individual tines orthogonal to the main vibrations. The actuation mechanism used for producing resonance on the tines can be electrostatic, electromagnetic, or piezoelectric. Also capacitive, piezoresistive, or piezoelectric methods are used to sense the Coriolis-induced vibrations in the sense mode[12].

In the wine glass gyroscope, the resonance of a resonant ring is the measured variable, and the positions of the nodal points indicate the rotation angle, they have been attractive candidates for high performance MEMS gyroscopes because of inherent advantages such as thermal stability and electrostatic tuning capability [14].

1.4 History and Applications

This section was mainly reported from [1].

The history of inertial sensors is relatively short. Despite the fundamental role played by inertial sensors in controlling the movement of a body, very little is known about early applications. This is even more remarkable given that most of the ingredients for building acceleration and angular-rate sensors, such as fine mechanics and precise spring technologies, were available from the late Middle Ages on and were used in the construction of, for instance, beautiful precision clocks.

Driven by the requirements for cost reduction and miniaturization, around 1960 new types of gyros like the vibrating-string gyro, the tuning-fork gyro and the vibrating-shell resonator emerged and opened the way to a drastic size and weight reduction, which finally ended with the transfer of these principles into the world of MEMS.

The piezoelectric and piezoresistive transducer principles were among the first to be employed at the beginning of the entry into the world of inertial MEMS and nanometer-scale mechanical elements, sensors,actuators, and electronic circuits on one carrier or even on one chip.

The commercialization of similar devices began around 10 years later and was very soon based on a variety of available transducer principles such as the sensing of capacitance changes between fixed and movable plates, the frequency measurement of resonant devices, the stabilization of a tunneling current by a closed-loop system, the sensing of thermal changes between a heater and a movable heat sink, and the sensing of changes of the thermal distribution within an air bubble. Everybody knows the pioneering role of MEMS-based 50g accelerometers used in airbag ignition devices, which became the first high-volume product in the area of inertial MEMS.

Analog Devices, supported by strong governmental funding, developed a special BiCMOS-MEMS process combining a known microelectronic process with a polysilicon deposition, etching, and release technology. Various inertial sensors were developed on the basis of this process, of which the first was the 50g accelerometer.

Within the MEMS market modern inertial sensors, accelerometers and gyros have gained a considerable share, exceeding 20 ٪ of the expected 12 Bn $ MEMS market in

2011 (∼ 6 Bn $ in 2009). In 2005 nearly 80 % of all applications were related to automotive safety functions such as automatic break systems (ABSs), airbag sensors, roll-over sensors, electronic stabilization systems (ESP), and other anti-skid systems as well as to navigation. Starting with 50g accelerometers in airbag safety systems, the next step – the introduction of electronic stabilization control (ESC) by Bosch and Systron Donner in 1994 – was significantly accelerated by the disastrous elk test of the newly invented Mercedes-Benz A-Class in 1997. ESC had to rescue the reputation of the brand name of one of the leading makers of high-quality cars. In ESC, yaw-rate gyroscopes and low-g sensors are usually the decisive information sources for controlling the finely allotted brake forces on the different wheels to avoid accidents.

Anti-skid control | Antenna stabilisation | Image stabilisation

Figure 1.20: Example applications of IMUs.

Besides the automotive industry, the remainder of the market is dictated by consumer applications Fig 1.20 1.21, which in recent years have shown dramatic growth. Accepted and growing applications are related to the use of inertial sensors in hand-held cameras for picture stabilization, in personal computers for hard-disk protection against mechanical shocks, in pedometers for motion sensing, and in more exotic products such as the two-wheel Human Transporter of Segway (Figure Cegway). Probably one of the most interesting applications is the motion sensing integrated into mobile phones, game controllers (Figure Gaming), toys and other human–machine interfaces. The Nintendo Wii's motion-sensing remote control has attracted the broad interest of the public to inertial MEMS. The consumer and information technology (IT) sector has increased from about 10 % in 2005 to about a 45 % share in 2009/10.

ST Microelectronics introduced in 2009 a three-axis high-performance gyroscope, whereas VTI announced a 3D accelerometer combined with a 1D gyroscope. Sensordynamics introduced in 2008 a combined one-axis accelerometer and one-axis gyroscope and announced a three-axis accelerometer plus one-axis gyro combination. Nowadays, new systems such as very small and cheap tri-axis sensors as well as more highly integrated multidimensional IMUs that include magnetometers and a pressure sensure are now on the market. The race into the world of multi-axis inertial sensors, including completely integrated 10-axis IMUs, is fully under way. Applications are numerous, ranging from medical 3D gesture and motion recognition via human–machine interfaces (HMIs) for game controllers, Virtaul reality and mobile phones to personal navigation systems.

Interactive gaming | Free fall detection | Platform stabilisation

Figure 1.21: Other applications of IMUs.

STMicroelectronics (ST) is still the global leader in the inertial consumer sensor market with more than 30% market share. InvenSense and Bosch struggle to compete with it today. InvenSense has reached more than 10% of the market, going ahead of Bosch Sensortec. These companies, including AKM, are preparing for the future, with InvenSense ahead as it seems to be stronger in the competition on 9-axis IMUs, found in a large numbers of products in development like Google Glass [9].

1.5 IMU's Specifications

The performance characteristics of inertial sensors (either accelerometers or gyroscopes) are usually described in terms of the following principal parameters: range, sensor bias, sensor scale factor, noise and bandwidth.

1.6 Errors[4]

Inertial sensors are prone to various errors which get more complex as the sensor gets cheaper. The errors limit the accuracy to which the observables can be measured. They are classified according to two broad categories of systematic and stochastic (or random) errors.

1.6.1 Systematic Errors

Systematic errors are due to manufacturing defects and can be calibrated out from the data. As we have seen before constructors include a factory calibration bloc in the package to correct few errors each time the sensor is powered on. Some systematic errors are given below:

Bias: This is a bias offset exhibited by all accelerometers and gyros. It is defined as the output of the sensor when there is zero input, and is depicted in Fig 1.22. It is independent of the underlying specific force and angular rate.

Scale Factor Error: This is the deviation of the input–output gradient from unity. The accelerometer output error due to scale factor error is proportional to the true specific force along the sensitive axis, whereas the gyroscope output error due to scale factor error is proportional to the true angular rate about the sensitive axis.Sensors usually are factory calibrated for this type of error. Figure 1.23 illustrates the effect of the scale factor error.

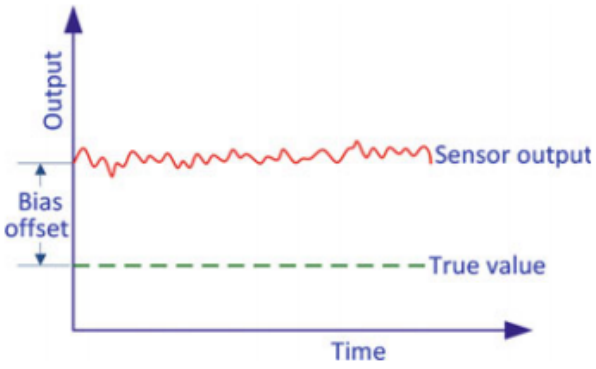

Figure 1.22: Inertial sensor bias error.[4]

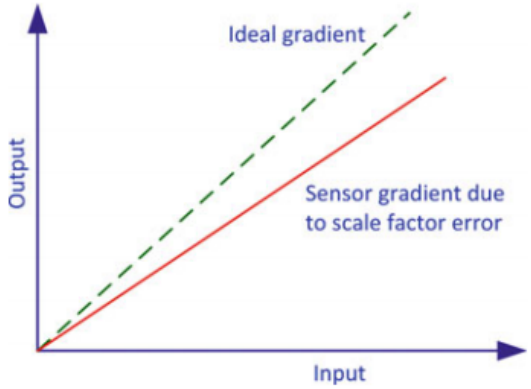

Figure 1.23: Inertial sensor scale factor error.[4]

Non-linearity:
This is non-linearity between the input and the output, as shown in Fig 1.24.

Scale Factor Sign Asymmetry:
This is due to the different scale factors for positive and negative inputs, as show in Fig 1.25.

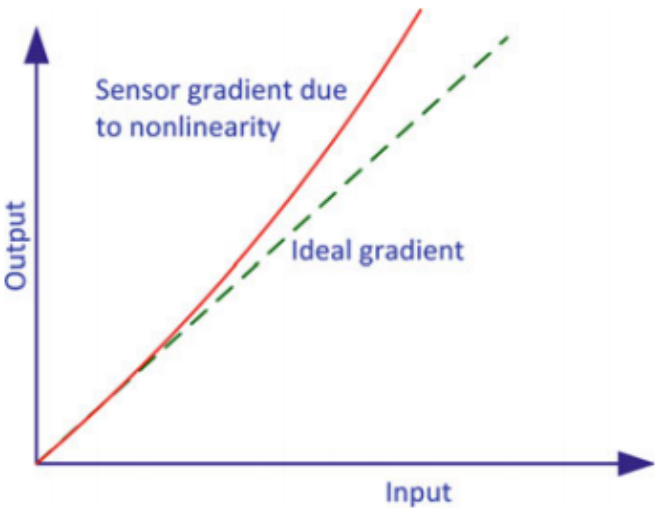

Figure 1.24: Nonlinearity of inertial sensor output.[4]

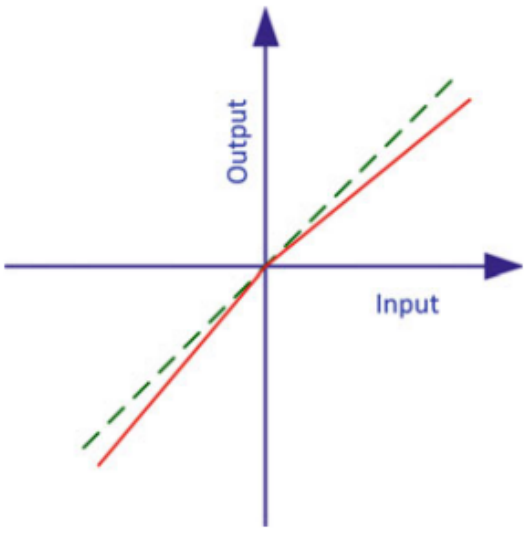

Figure 1.25: Scale factor sign asymmetry.[4]

Dead Zone
This is the range where there is no output despite the presence of an input, and it is

shown in Fig 1.26.

Quantization Error:
This type of error is present in all digital systems which generate their inputs from analog signals, and is illustrated in Fig 1.27.

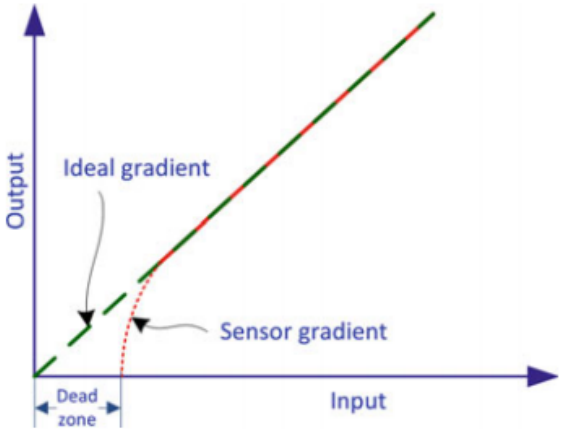

Figure 1.26: Dead zone in the output of an inertial sensor.[4]

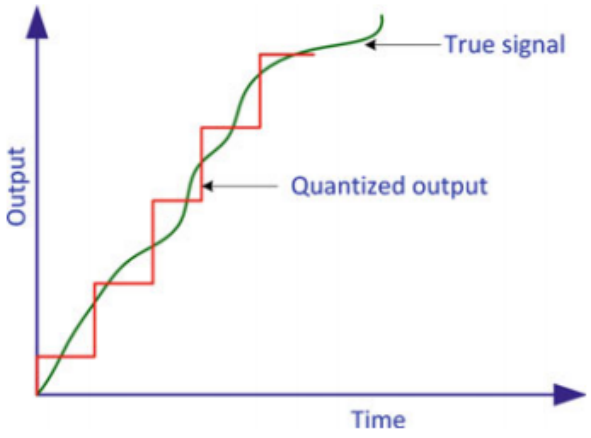

Figure 1.27: The error due to quantization of an analog signal to a digital signal.[4]

on-orthogonality Error:
As the name suggests, non-orthogonality errors occur when any of the axes of the sensor triad depart from mutual orthogonality. This usually happens at the time of manufacturing. Figure 1.28 depicts the case of the z-axis being misaligned by an angular offset of θ_{zx} from xz-plane and θ_{yz} from the yz-plane.

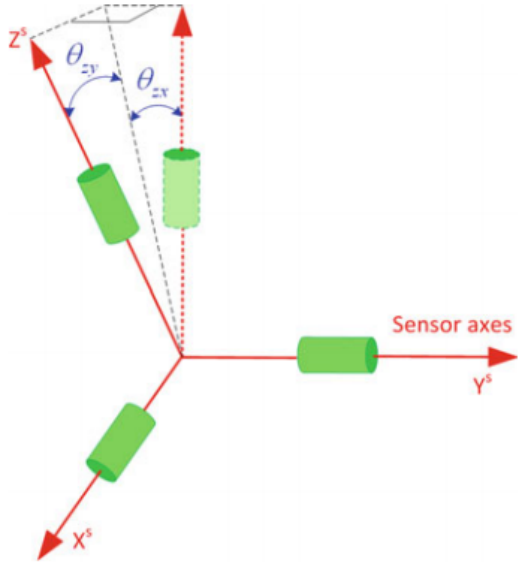

Figure 1.28: Sensor axes nonorthogonality error [4]

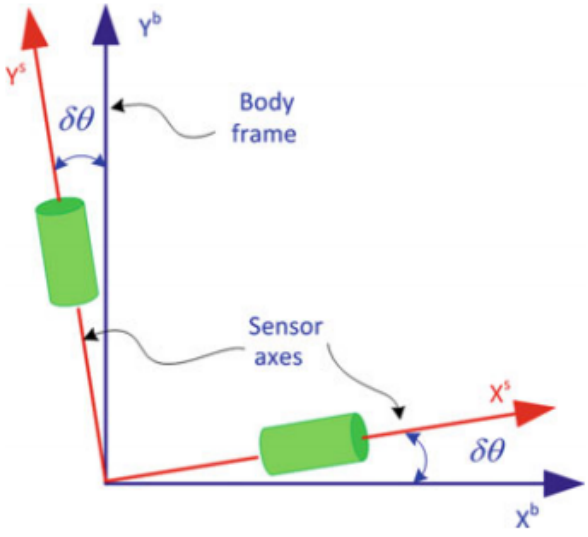

Figure 1.29: Misalignment error between the body frame and the sensor axes [4]

Misalignment Error:
This is the result of misaligning the sensitive axes of the inertial sensors relative to the orthogonal axes of the body frame as a result of mounting imperfections. This is depicted in Fig 1.29 for a sensor frame misalignment (using superscript 's') with respect to the body in a 2D system in which the axes are offset by the small angle $\delta\theta$.

1.6.2 Random Errors

Inertial sensors suffer from a variety of random errors which are usually modeled stochastically in order to mitigate their effects. Most manufacturers express the randomness associated with their inertial sensors by the concept of random walk. The angle random walk (ARW) for gyroscopes is usually specified in terms of $deg/hr/\sqrt{Hz}$ or $deg/\sqrt{Hz}$, and the velocity random walk (VRW) for accelerometers is given in terms of $\mu g/\sqrt{Hz}$ or $m/s/\sqrt{Hz}$. This requires knowledge of the data rate (sampling frequency) at which the sensor measurements are acquired by the data acquisition systems, hence knowig the bandwidth of the sensor.

Run-to-Run Bias Offset:
If the bias offset changes for every run, this falls under the bias repeatability error, and is called the run-to-run bias offset.

Bias Drift:
This is a random change in bias over time during a run. It is the instability in the sensor bias for a single run, and is called bias drift. It is illustrated in Fig 1.30. Bias is deterministic but bias drift is stochastic. One cause of bias drift is a change in temperature.

Scale Factor Instability:
Random changes in scale factor during a single run. This is usually the result of temperature variations. The scale factor can also change from run to run, but stay constant during a particular run. This demonstrates the repeatability of the sensor and is also called the run-to-run scale factor.

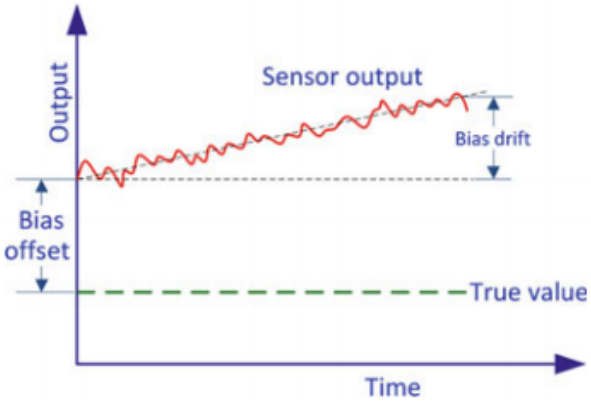

Figure 1.30: Error in sensor output due to bias drift [4]

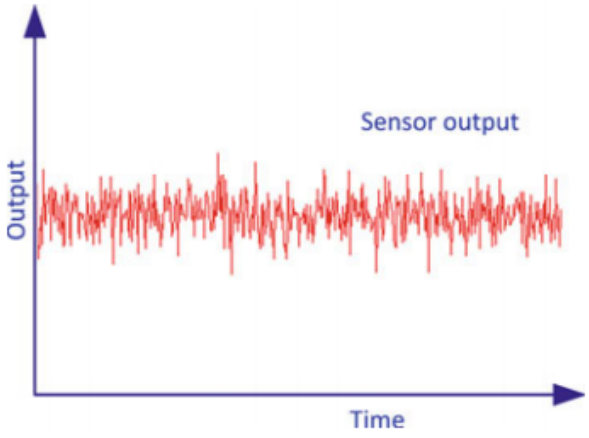

Figure 1.31: A depiction of white noise error. [4]

White Noise:
This is an uncorrelated noise that is evenly distributed in all frequencies. This type of noise can be caused by power sources but can also be intrinsic to semiconductor devices. White noise is illustrated in Fig.1.31.

Conclusion

In this chapter we have seen an initial background on inertial sensors and systems as well and their two main types: gimbaled and strapdown technologies. We have talked about MEMS inertial systems, their internal structure and a brief history with given examples of application.

Finally, we cited different error types which can be categorized into two types: Systematic errors corrected by manufactory calibration and random errors (stochastic) wich will be corrected using filters once implemented within applications.

2

System Design

This chapter introduces a relatively new design concept adopted by engineers and manufacturers for rapid products prototyping with a revolutionary integration in the industry and thatis Model-Based Design.

For the purpose of using this method in our project, we will show how MathWork's Simulink uses this approach for designing, implementing and testing models and algorithms on different hardware platforms.

2.1 Model Based Design (MBD)

Model-Based Design is a methodological concept used by engineers and scientists to model, validate and simulate systems and algorithms by harnessing the power of sophisticated computer simulations in the design process, allowing to generate a suitable algorithm code depending on a targeted platform. Before actually creating any physical prototypes, engineers build a virtual model of the system components, and can use computer simulations to test how the design will perform in the real world. Computer-based engineering simulation early in the development process allows them to refine and validate designs at a stage where the cost of making changes is minimal.

Software design and development encountered earlier many challenges by getting to design complex systems with interacting components of different natures, as the specifications and the physical prototypes may change depending on the system requirements; thus resulting in alterations that could affect every stage in the development process in order to satisfy eventual modifications in thousands of code lines (Fig 2.1).

Over the past decades, computer-based engineering simulation technology has evolved to give designers a unified platform that connects all the physics together. Companies are nowadays studying many possible design variations for a given system before choosing the final design that they actually want to produce.

Model Based Design is now just beginning to be implemented in many disciplines. It starts by constructing a virtual model taking in the requirements and laying out a rough architecture. Various types of, essentially functional block diagrams like Data flow diagrams, Schematic models, event-driven systems (state machines) or textual coding (programming languages) can be used in the same system. Each block has certain inputs and functions that it has to do. One could be a signal acquisition bloc, another could be a signal conditionner, one could be a 3D model controlled by the first ones, etc. For each of those functional attributes there is a more detailed design.

On the websites of tool vendors (like Mathworks [15], dSpace [16] , ASNSYS [17]) many success stories can be found, which report of efficiency gains from up to 50 % in the development, high error reductions and a rapid increase of the developed functions maturity level, just because of Model Based development adoption.

Among the advantages of Model Based Design, we state:

✓ Flexibility in development process.
✓ Early continuous concept verification and validation.
✓ Exploring various bloc design possibilities for higher product quality.
✓ Multi-Domain modeling and algorithm development with enhanced team work support.
✓ Accelerated code development and reviews for rapid prototyping.
✓ Automatic code generation.
✓ Traceability: Requirement ⇔ Model ⇔ Code.
✓ Reduced Time-To-Market .

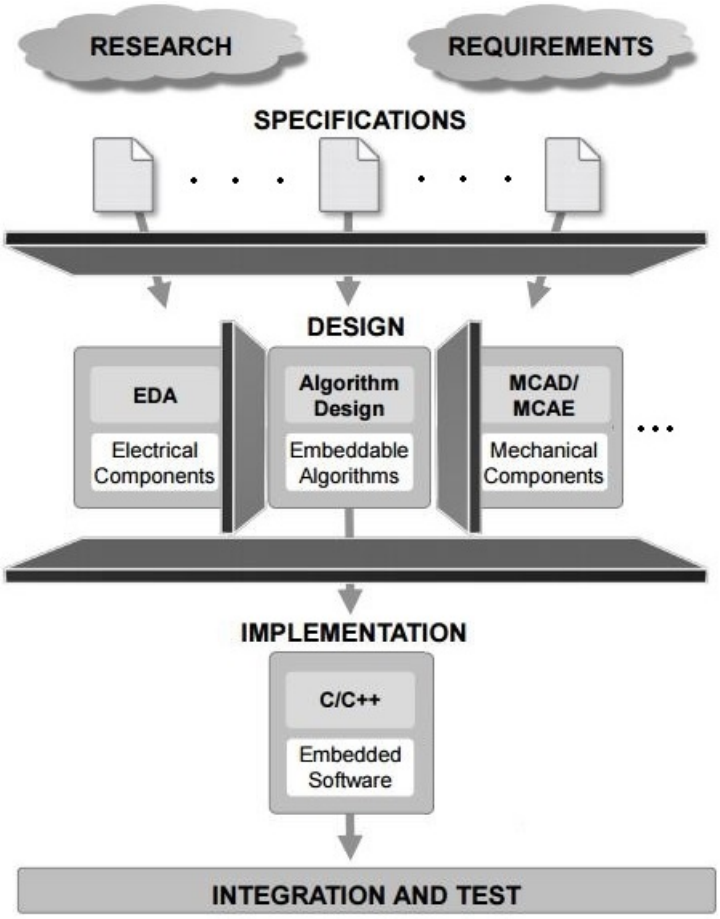

Figure 2.1: A common (classic) design workflow [15]

The V-model, or system-V, is a terminology used in this model based product development approach. On the top left side of the V, the process begins with the rough concepts and requirements. As you go down the left side you make a more detailed design of the product. A functional architectures of the main system, then functional architecture of the sub-components and systems, and at the bottom of the V are the detailed coded models and multiphysics analysis. So essentially it goes from requirements to the detailed

design at the bottom.

On the right side of the V-model is the opposite, where we go from the detailed models to subsequent higher levels of implementation and testing. Starting with testing of components and subsystems, and ultimately testing of the whole system as one. At all levels there is a comparison of the right side to the left side, verifying that the whole system and its performance matches the original requirements that were set on the left side of the V-model.

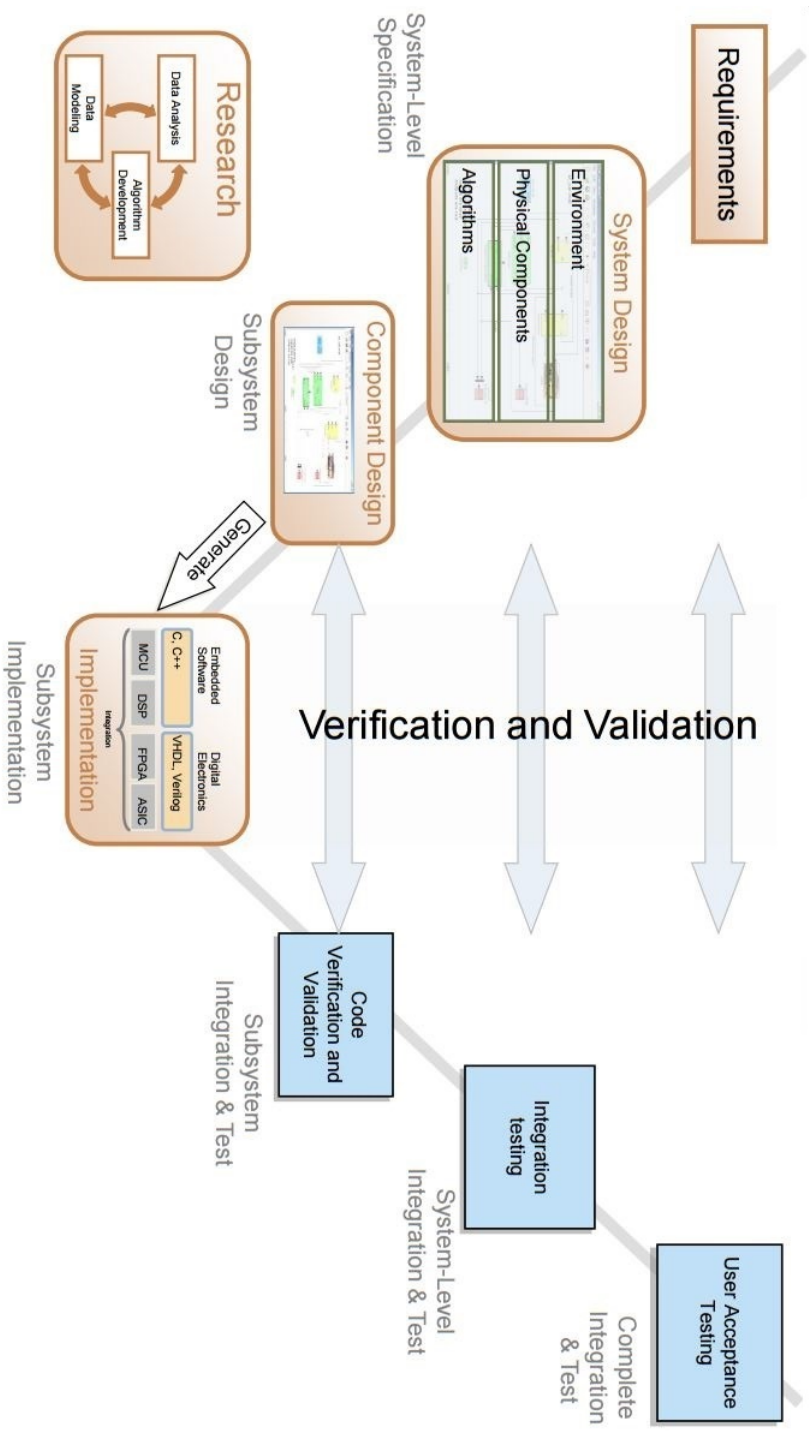

Figure 2.2: Model-Based Design development Process [15]

So, model based design is a way of doing computer models that can take a complete product down from the left side of the V-model and up the right side.

2.2 MBD with Simulink

Since we are going to use this platform for designing our project, we should first give a brief presentation of Simulink and how it is organized for Model-Based Design applications. The figure below gives the general Simulink Model-Based Design workflow.

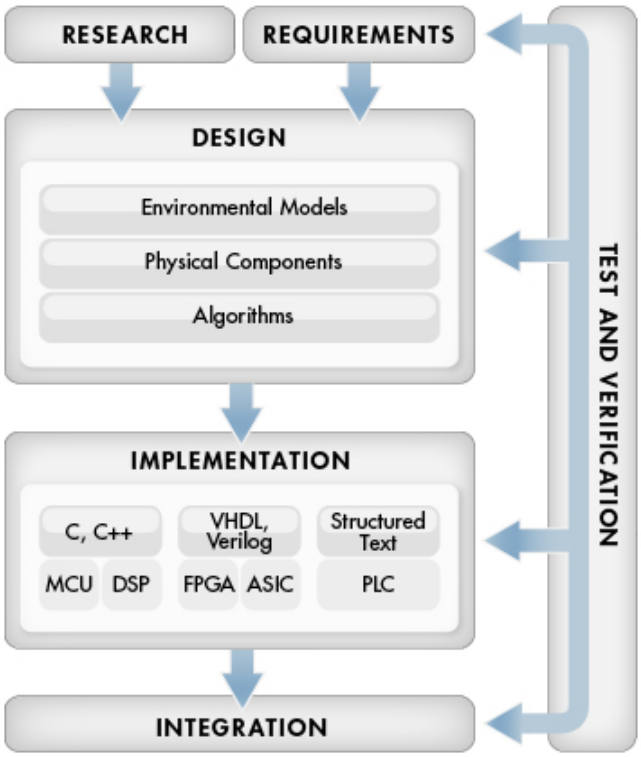

Figure 2.3: Model-Based Design development Process [18]

Engineers look for solutions for technical problems. This solution depends on the complexity of the algorithm and also the hardware used for implementation. These two criterions represent the requirement in Fig 2.3. The label research means the potential new algorithms or solution methods continuously proposed to optimize the problem solution, together we can find the appropriate design for the solution.

The design window represents the model design that includes every affective component to system behavior like algorithms, control logic, physical components, and Inetllectual Properties (IP) developed in MATLAB, C, HDL, or domain-specific modeling tools. At this point we can simulate the model and analyze system performance.

Simulink gives another advantage by providing user-friendly simulations and tests directly on the hardware platform (FPGA, Embedded Systems, PLC ...) and getting the results in real time simulation. Direct design evaluation and test on the hardware avoids spending time over code implementation and hardware tests. So, this also is a major advantage of choosing the hardware that fits our requirements since the model design will

remain the same, we only need to change simulation environment parameters, therefore development time is reduced.

After the model design and simulation, the next step is implementation. Simulink provides automatically generated code for the specified Hardware target.

2.3 Simulink and indusrty standards

MATLAB, Simulink and Embedded Coder support key development activities involving Model-Based Design for DO-178C standard.

DO-178C (Software Considerations in Airborne Systems and Equipment Certification) is the latest update to the DO-178 standard (January 2012) [19]. The previous version, DO-178B, was published in 1992 when most software was coded by hand [20]. Since software development processes have evolved from hand coding to automatic coding with Model-Based Design, DO-178C was developed to keep pace with industry.

MathWorks also provides other products and services that support important verification tasks:

- Simulink Code Inspector for automating source code reviews of generated code.
- Simulink Verification and Validation for automating requirements tracing, modeling standards compliance checking, and test-harness generation.
- Simulink Report Generator for documenting models and test results.
- Polyspace Bug Finder and Polyspace Code Prover for formal verification of hand-written or generated code.
- DO Qualification Kit for qualification of these and other verification products used in Model-Based Design.

The next figure shows Simulink products used in project design to satisfy the Model-Based Design's V-model requirements and shows dedicated applications for verification and validation in conjunction to conformance satisfaction to industry standards.

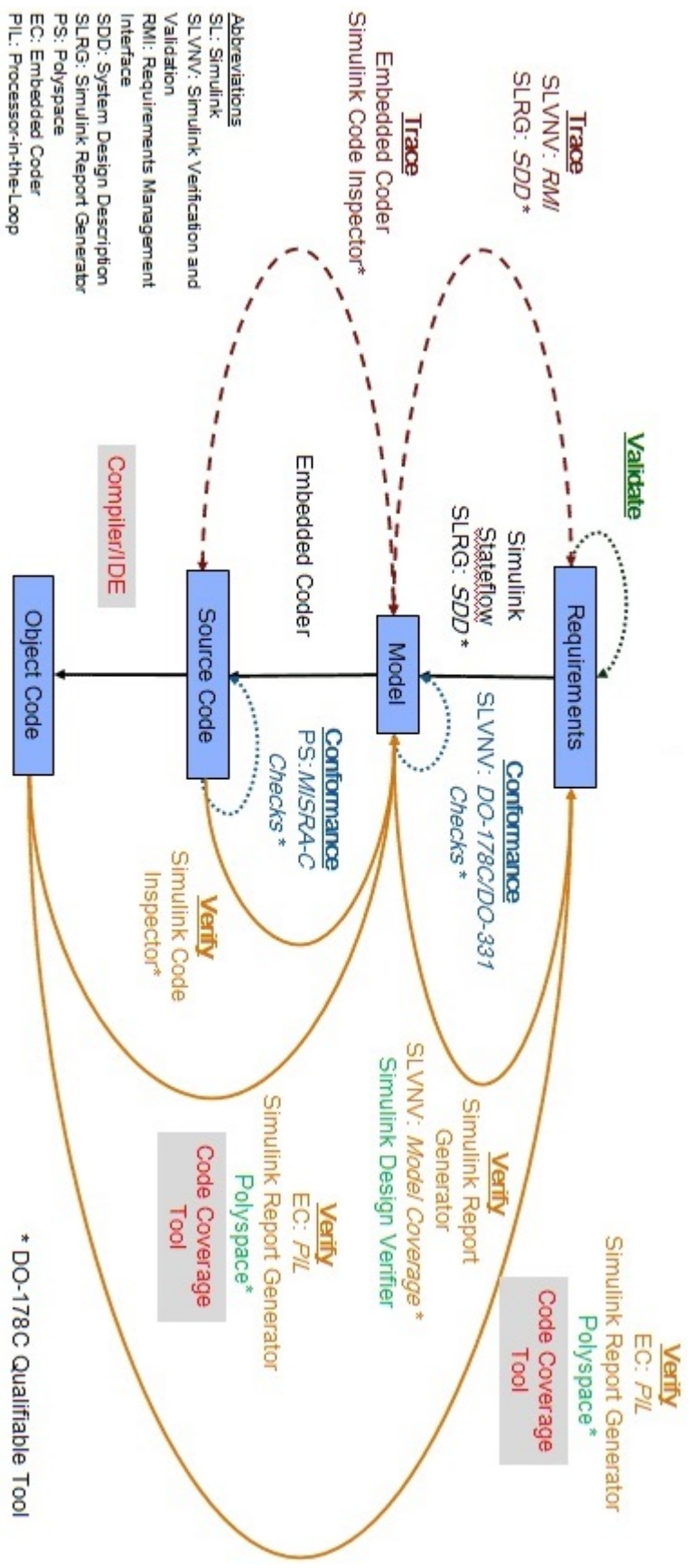

Figure 2.4: Analogy to Model-Based Design development process in Simulink products [15].

2.4 Multiple Hardware Architectures

Simulink applications packages (like Embedded Coder, HDL Coder, etc.) provide automatically generated code for the specified Hardware target.

Embedded Coder generates an optimized C and C++ code for use on a variety of embedded processors (like ARM, Intel, Texas Instruments) and DSP.

HDL coder on the other hand generates portable, synthesizable Verilog and VHDL code from Simulink models. The generated HDL code can be used for FPGA programming or ASIC prototyping and design. New hybrid technology products wich combine different hardware architectures (like Xilinx Zynq and Altera SoC) are completely supported by Simulink, allowing to generate a multi-platform code for each part of the hardware.

Simulink can also generate Structured Text to be implemented in PLC systems. Another useful option is the ability to modify the generated code to be more optimized and combined with hand-written code.

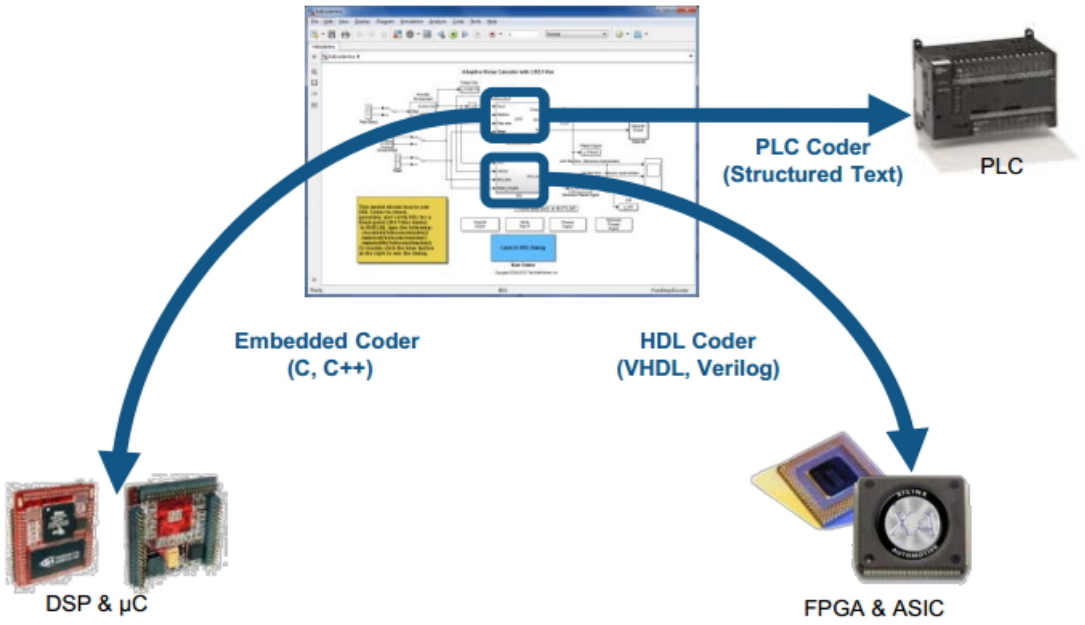

Figure 2.5: Simulink model deployed on different hardware platforms

2.5 Embedded Coder

"Embedded Coder" gives you an advantage for generating readable, compact, and fast C and C++ code for use on embedded processors, on-target rapid prototyping boards, and microprocessors used in mass production. Based on "MATLAB Coder" and "Simulink Coder"; Embedded coder enables additional configuration options and advanced optimizations for fine-grain control of the generated code's functions, files, and data. You can incorporate a third-party development environment into the build process to produce an executable to be deployed on your embedded system and tested in real-time execution [21].

Embedded coder offers a set of choices and option for generating and implementing a

C and C++ code. In this report we are going to show few options that we are going to use in our project; more option can be found in MathWorks website [15].

From the Simulink Model Explorer window, you can:

- Select an Embedded Coder target.
- Configure the target for code generation.
- Create, deploy, and reuse multiple configuration sets.

Selecting Targets:

Embedded Coder uses configuration objects and system target files to translate your MATLAB code and Simulink models into production-quality source code and executables. For a Simulink system target file, you specify the real-time environment on which your generated code will run.

Embedded Coder includes target files for several ready-to-run configurations, and supports third-party and custom targets as well. Built-in targets include:

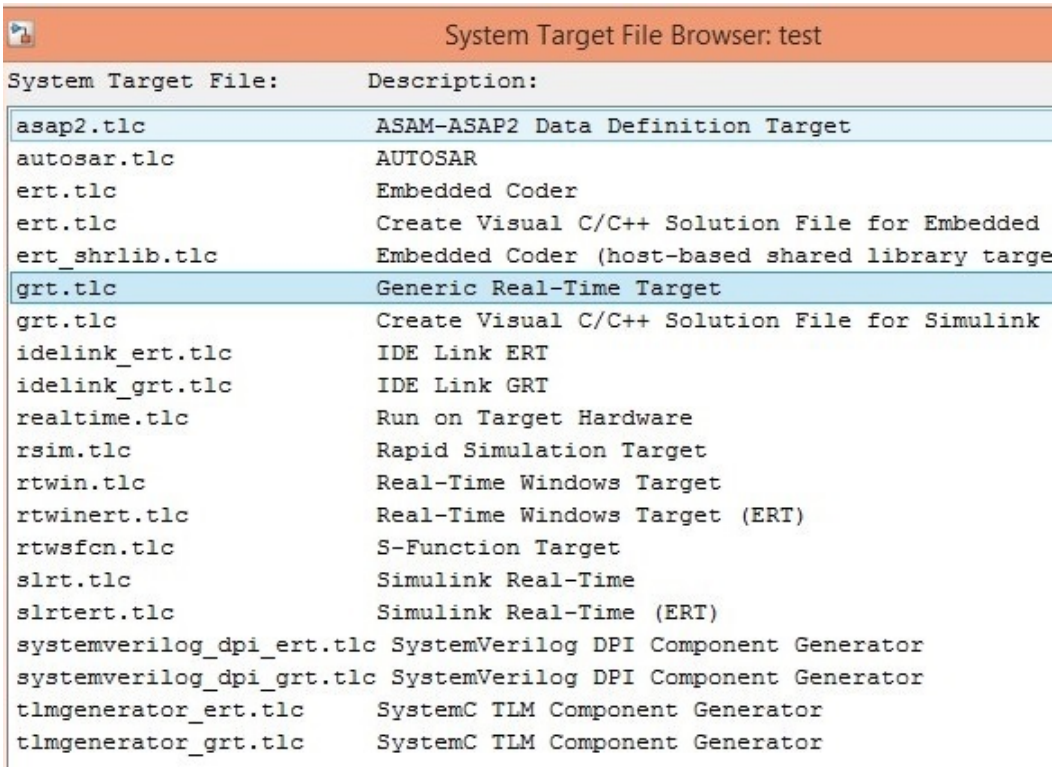

Figure 2.6: Target Language Compiler (TLC) files that could be generated for different targets

- Embedded Real-Time Target : Generates ANSI/ISO C, C++, and encapsulated C++ code with floating-point and fixed-point data for efficient real-time execution on virtually any production processor
- AUTOSAR Target : Generates C code and run-time interfaces that support development of AUTOSAR (AUTomotive Open System ARchitecture) software components

- Shared Library Target : Generates a shared library version of your code for host platform execution, either as a Windows® dynamic link library (.dll) file or a UNIX® shared object (.so) file

- IDE Link Target : Generates code for compilation and deployment using a supported third-party integrated development environment (IDE) such as Texas Instruments' Code Composer Studio.

Defining Embedded Hardware Characteristics:

For MATLAB or Simulink code generation, you select the deployment processor from a predefined list or use generic target settings. You can also extend the predefined list for your custom environment.

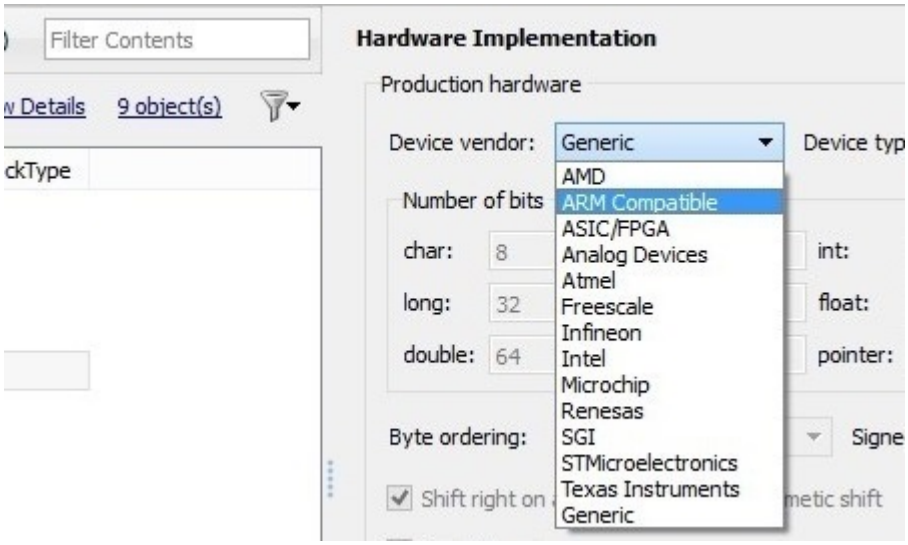

Figure 2.7: Hardware Implementation choices in the Model Explorer

Generating a Main Program:

Embedded Coder generates an extensible main program based on information you provide for deploying the code in your real-time environment. This capability lets you generate code from a model, that is optimized for speed, memory usage, simplicity, and possibly, compliance with industry standards and guidelines..

Executing and Verifying Code:

Embedded Coder enables you to incorporate generated code into your code execution environment. With Simulink, Embedded Coder significantly extends the real-time execution framework provided by "Simulink Coder" (previously "Real Time Workshop"). By default, the code can be executed with or without a real-time operating system (RTOS) and in single-tasking, multitasking, or asynchronous mode. You can also verify the code execution results using software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing.

You execute in SIL and PIL emulations to generated code with your plant model within Simulink to verify successful conversion of the model to code by testing an object code

component with a plant or environment model in an open- or closed-loop simulation to verify successful model-to-code conversion, cross-compilation, and software integration. A SIL simulation involves compiling and running production source code on your host computer to verify the source code. whereas, a PIL simulation involves cross-compiling and running production object code on a target processor or an equivalent instruction set simulator.

You might change the code to emulate target word size behavior and verify numerical results expected when the code runs on an embedded microprocessor. Or, you might use actual target word sizes and just test production code behavior.

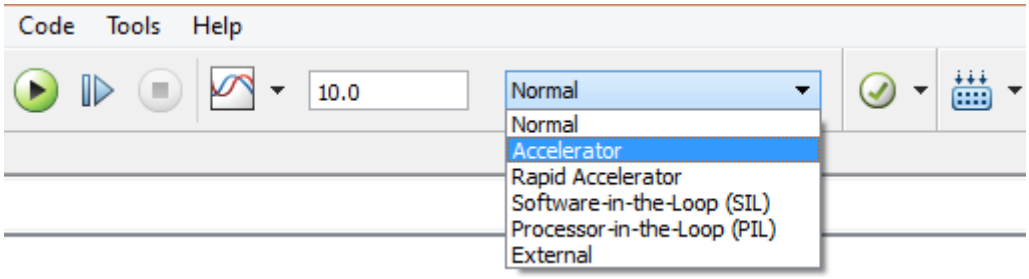

Figure 2.8: Different options for code execution results' verification

2.6 HDL Coder

Due to the order of magnitude performance improvement when running high-performance applications, algorithm designers are increasingly using FPGAs to prototype and validate their innovations instead of using traditional processors. However, many of the algorithms are implemented in MATLAB due to the simple-to-use programming model and rich analysis and visualization capabilities. When targeting FPGAs or ASICs these MATLAB algorithms have to be manually translated to HDL.

For many algorithm developers, mastering the FPGA design workflow is a challenge. Unlike software algorithm development, hardware development requires them to think parallel. Other obstacles include: learning the VHDL or Verilog language, mastering IDEs from FPGA vendors, and understanding esoteric terms like "multi-cycle path" and "delay balancing".

The process of translating MATLAB designs to hardware consists of the following steps:

- Model your algorithm in MATLAB - use MATLAB to simulate, debug, and iteratively test and optimize the design.
- Generate HDL code - automatically create HDL code for FPGA prototyping.
- Verify HDL code - reuse your MATLAB test bench to verify the generated HDL code.
- Create and verify FPGA prototype - implement and verify your design on FPGAs.

There are some challenges in translating MATLAB to hardware. MATLAB code is procedural and can be highly abstract; it can use floating-point data and has no notion

of time. Complex loops can be inferred from matrix operations and toolbox functions.

- Converting floating-point MATLAB code to fixed-point MATLAB code with optimized bit widths suitable for efficient hardware generation.
- Identifying and mapping procedural constructs to concurrent area- and speed-optimized hardware operations.
- Introducing the concept of time by adding clocks and clock rates to schedule the operations in hardware.
- Creating resource-shared architectures to implement expensive operators like multipliers and for-loop bodies.
- Mapping large persistent arrays to block RAM in hardware.

HDL Coder simplifies the above tasks though workflow automation by generating portable, synthesizable Verilog and VHDL code from MATLAB functions, Simulink models, and Stateflow charts. The generated HDL code can be used for FPGA programming or ASIC prototyping and design.

HDL Coder provides a workflow advisor that automates the programming of Xilinx and Altera FPGAs. You can control HDL architecture and implementation, highlight critical paths, and generate hardware resource utilization estimates by providing traceability between your Simulink model and the generated Verilog and VHDL code.

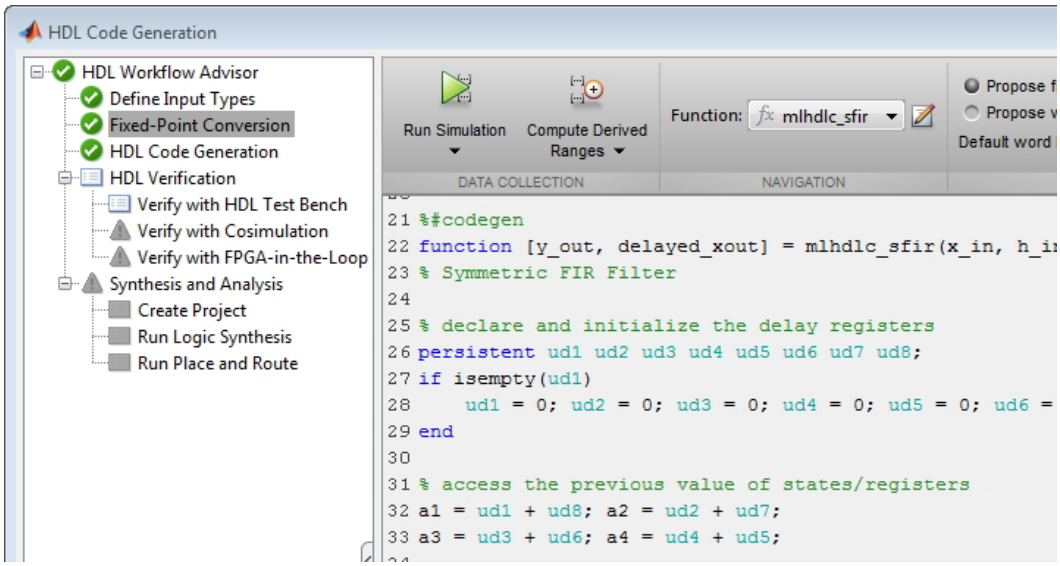

Figure 2.9: HDL coder automated HDL generation steps

The HDL Workflow Advisor as shown above helps automate the steps and provides a guided path from MATLAB to hardware. You can see the following key steps of the workflow in the left pane of the workflow advisor:

- Fixed-Point Conversion
- HDL Code Generation
- HDL Verification
- HDL Synthesis and Analysis

Conclusion

By concluding the present chapter, we have seen the design flow of MBD with its application in Simulink and the variety of options that it provides like automatic code generation and multi-targeted hardware implementation. We can now specify the hardware that we are are going to adapt to the Simulink environment in order to use it for the design of our project.

3

Target Hardware

After we introduced the Model-Based Design with Simulink, we now focus in this chapter on the hardware which is going to be used to develop our prototyping platform.

A general overview of the system will be discussed, before introducing each part of the system in more details; then, we'll have enough information in order to dwell with the design procedures in Chapter 4.

Our design will be implemented using the following hardware configuration as shown in the Figure below:

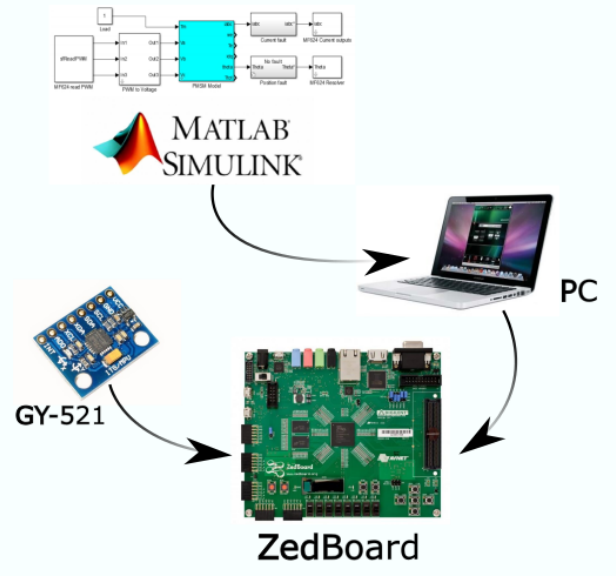

Figure 3.1: Target Hardware

MATLAB and Simulink will be hosted on a PC connected to a development board: ZedBoard Zynq™-7000 Development Board. The design platform is developed around an MPU6050 based IMU sensor to validate it: The GY-521 Evaluation Board, which is also connected and driven by the ZedBoard. We aim to visualize and interact with the design by injecting the sensor measurments in a real-time simulation.

3.1 The Zedboard

In this section we are going to introduce the architecture and features of Zedboard, as well as the Zynq platform architecture and the peripheral interfaces.

To explain the name, 'Zed' stands for Zynq Evaluation and Development. The Board contains a new Xilinx All programmable SoC (System on chip) technology called the Zynq. The Zedboard is one of the low cost development and evaluation boards that uses the Zynq chip. It is a joint venture between Xilinx, Avnet (the distributor), and Digilent (the board manufacturer). It is priced at a suitable level for students and enthusiasts, and it forms the focus of an online community of ZedBoard users [22].

the ZedBoard features a ZC7Z020 Zynq device. This is one of the smaller devices in the Zynq-7000 range, and it is based on the Artix-7 logic fabric, with a capacity of 13,300 logic slices, 220 DSP48E1s, and 140 BlockRAMs. The device also contains an XADC hard IP block, although it does not feature high-speed transceivers or PCIExpress blocks.

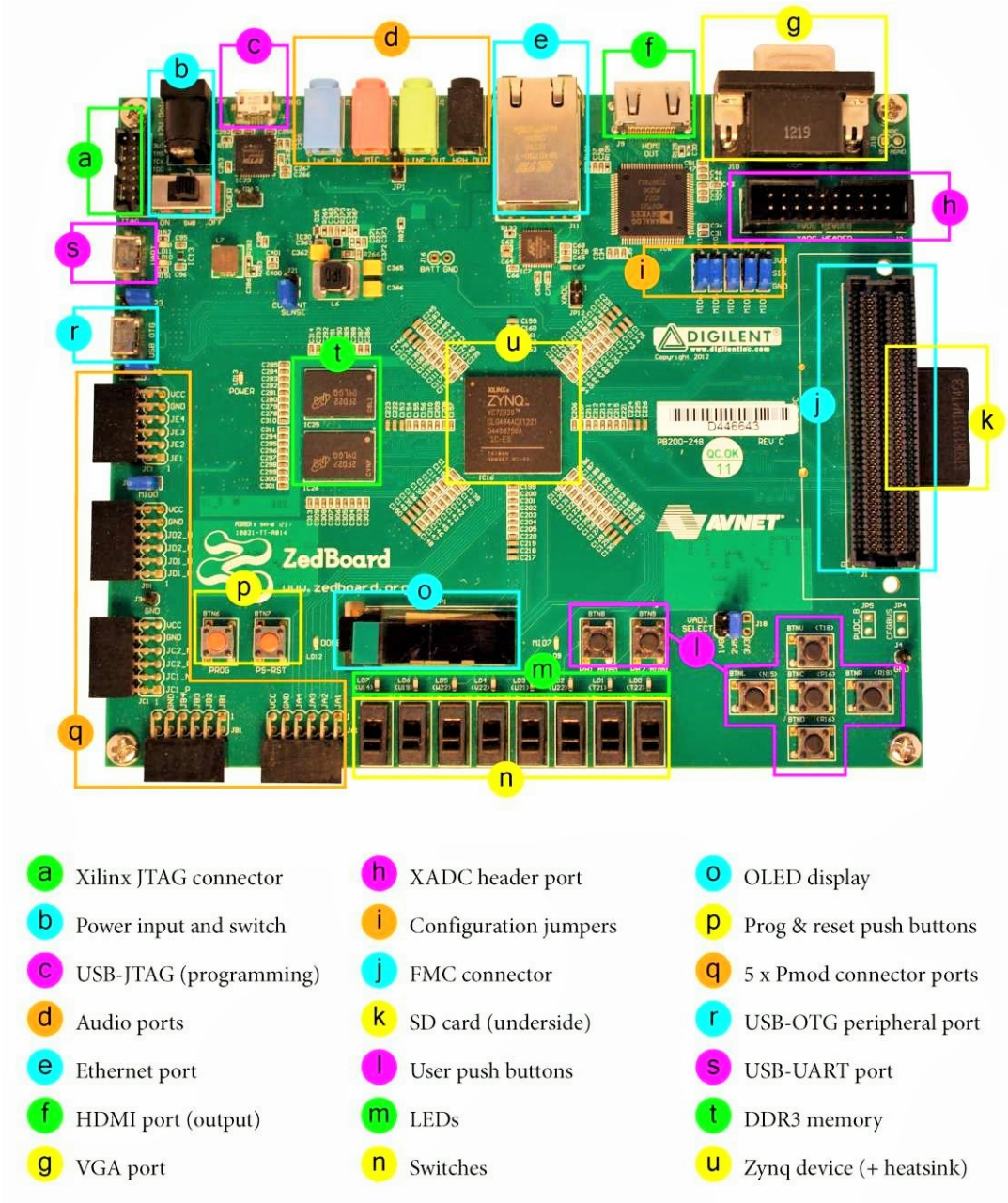

Figure 3.2: Different components of the ZedBoard [23].

There are a number of peripheral interfaces on the ZedBoard:

- GPIO: in total, 9 x LEDs, 8 x switches, 7 x push buttons
- Audio codec (Analog Devices ADAU1761, supporting line in, line out, microphone (in), and headphone (out))
- Video (HDMI)
- Video (VGA)
- Organic Light Emitting Diode (OLED) display
- Pmod interfaces (x 5)
- Ethernet
- USB-OTG (peripherals)
- USB-JTAG (programming)
- USB-UART (communication)
- SD card slot (located on the underside of the board)
- FMC interface
- XADC header
- Xilinx JTAG header

Additionally, the Zynq device interfaces to a 256 Mbit flash memory and 512 MB DDR3 memory, both of which are found on the board. There are two oscillator clock sources, one at 100MHz, and the other at 33.3333 MHz [23].

The features residing on the front of the ZedBoard are highlighted in Figure 3.2.

3.1.1 The Zynq SoC

As we mentioned before, the Zynq is a new Xilinx All programmable SoC called the Zynq-7000 and it is a family of product with difference in hardware characteristics but the same basic architecture. This chip comprises two sections: the Processing System (PS), and the Programmable Logic (PL). These can be used independently or together, and in fact the power circuitry is configured with separate domains for each, enabling either the PS or PL to be powered down if not in use. However, the most compelling use model for Zynq is when both of its constituent parts are used in conjunction. This new co-existence of two different architecture PS-PL in one chip made it one of the powerful platform for designing complex circuits and create additional features for engineers to develop and innovate new embedded systems. Therefor it is important to appreciate the structure of both sections, as well as the interfaces between them [23].

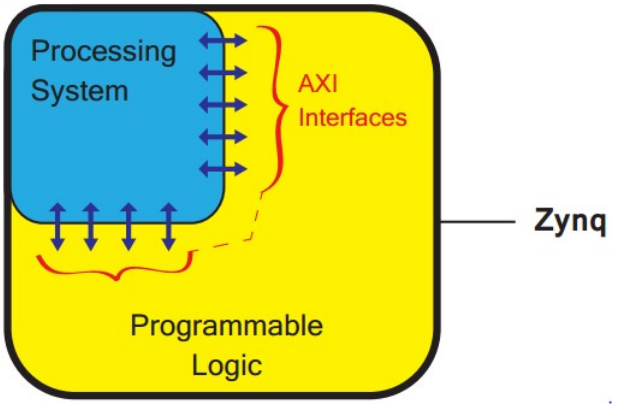

Figure 3.3: A simplified model of the Zynq architecture [23]

3.1.2 Processing System (PS)

All devices in the Zynq family have the same processing system which is a dual-core ARM Cortex-A9 processor. It is a 'hard' processor, i.e. it exists as a dedicated and optimized silicon element on the device.

For comparison purposes, the alternative to a hard processor is a 'soft' processor like the Xilinx MicroBlaze, which is formed by combining elements of the Programmable Logic (PL) fabric. The implementation of a soft processor is therefore the equivalent of any other IP block deployed in the logic fabric of an FPGA. In general, the advantage of soft processors is that the number and precise implementation of processor instances is flexible. On the other hand, hard processors can achieve considerably higher performance, as is true of Zynq's ARM processor.

This co-existence of the PS beside a PL can create a collaboration between an ARM processor (hard processor) and one or more soft processor MicroBlaze formed by combining elements of the programmable logic. And the connection between them is achieved by communication bus named AXI (Advanced eXtensible Interface). This Bus technology is not only used to connect a hard and soft processor, it also gives the capability for one of them to use some of the resources of the other, mainly the external peripherals (I/O).

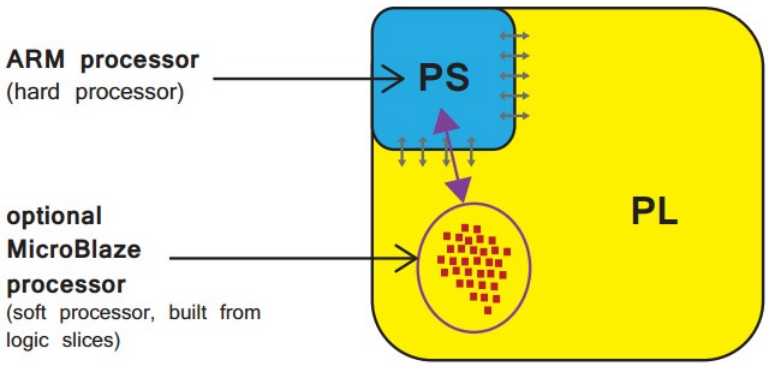

Figure 3.4: Locations of hard (ARM Cortex-A9) and soft (MicroBlaze) processors on a Zynq device [23]

Let's focuses on the internal structure of the Processing system. Fig 3.5 below gives

an overview of all the component of the PS. We see the ARM processor, with a set of associated processing resources forming an Application Processing Unit (APU), and further peripheral interfaces, cache memory, memory interfaces, interconnect, and clock generation circuitry.

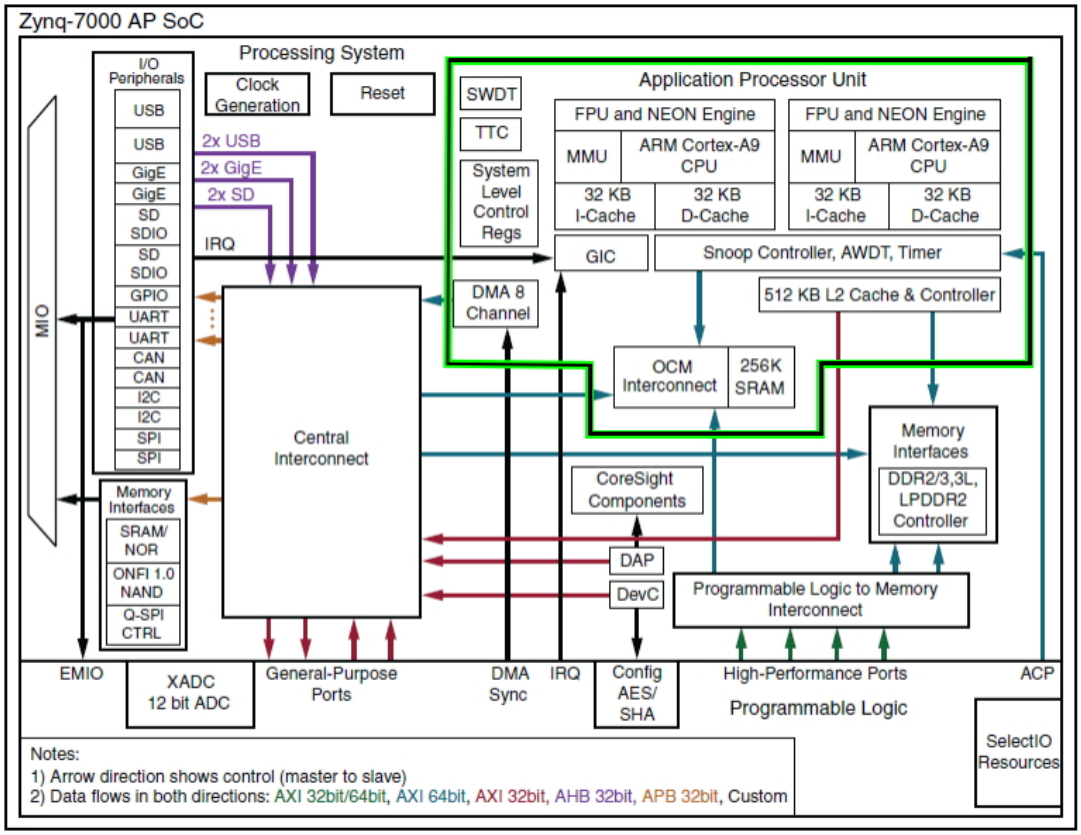

Figure 3.5: The internal componetns of the PS and its connection to the PL [23]

3.1.3 Processing System External Interfaces

Communication between the PS and external interfaces is achieved primarily via the Multiplexed Input/Output (MIO) bank, which provides 54 pins of flexible connectivity (Fig 3.6), meaning that the mapping between peripherals and pins can be defined as required (as we'll see in the next chapter). Certain connections can also be made via the Extended MIO (EMIO), which is used as a direct path to provide the PS with external connections of the the PL's I/O resources. These are both shown at the left side of Fig 3.5. The EMIO can be used when extension beyond 54 pins is required, or as a method of interfacing the PS with an IP block implemented in the PL.

Table 3.1: Main available I/O interfaces and their description

I/O Interface	Description
SPI (x2)	Serial Peripheral Interface. Standard for serial communications based on a 4-pin interface. Master or slave mode.
I2C (x2)	I2C bus Compliant with the I2C bus specification, version 2. Supports master and slave modes.
CAN (x2)	Controller Area Network Bus interface controller
UART (x2)	Universal Asynchronous Receiver Transmitter Low rate data modem interface for serial communication. Often used for Terminal connections to a host PC.
GPIO	General Purpose Input/Output There are 4 banks GPIO, each of 32 bits.
SD (x2)	For interfacing with SD card memory.
USB (x2)	Universal Serial Bus Compliant with USB 2.0,
GigE (x2)	Ethernet MAC peripheral, supporting 10Mbps, 100Mbps and 1Gbps modes.

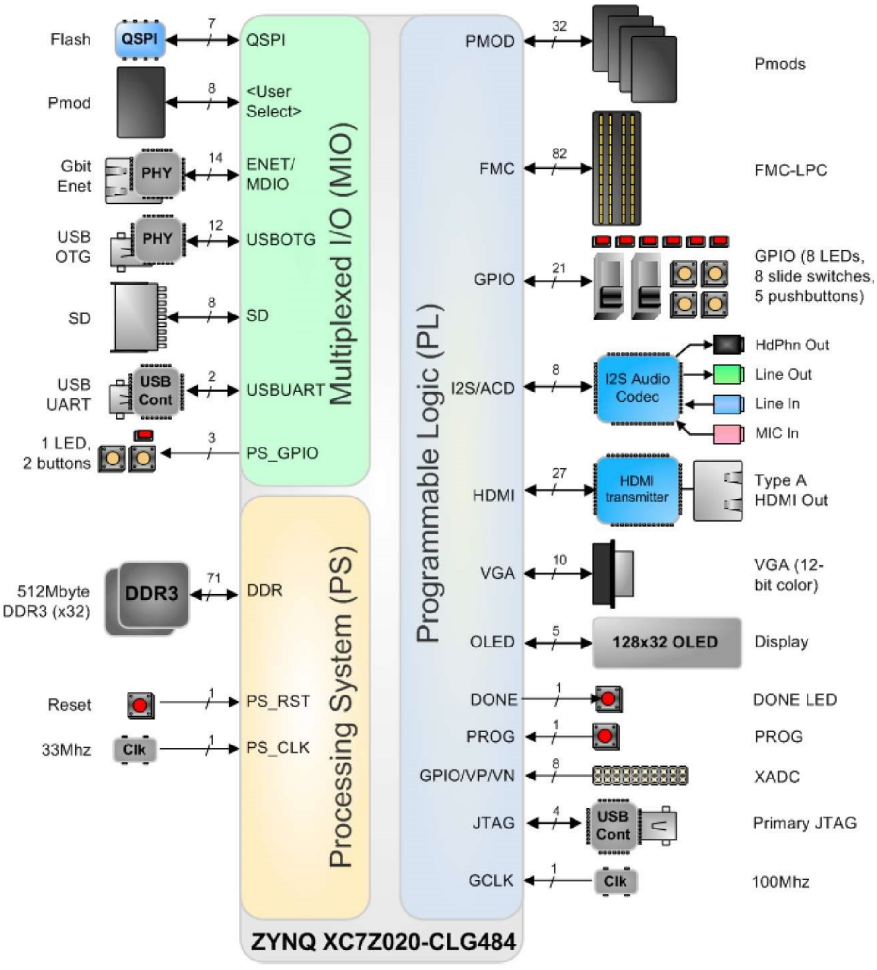

Figure 3.6: ZedBoard Block Diagram[23]

3.2 Programmable Logic (PL)

The second principal part of the Zynq architecture is the programmable logic. This is based on the Artix®-7 and Kintex®-7 FPGA fabric, The PL is predominantly composed of general purpose FPGA logic fabric, which is composed of slices and Configurable Logic Blocks (CLBs), and there are also Input/Output Blocks (IOBs) for interfacing. (Note that these are all Xilinx-specific terms.)

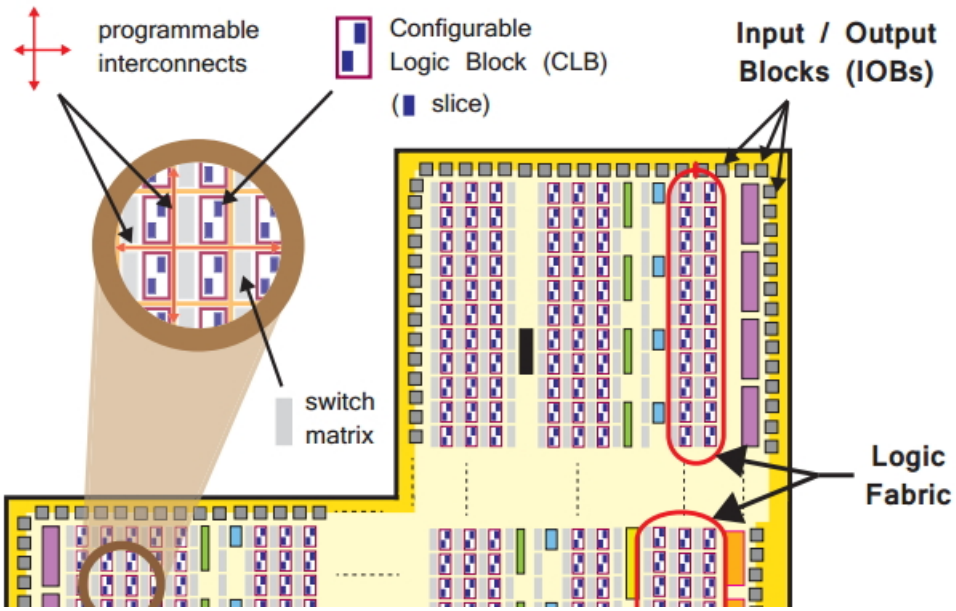

Figure 3.7: Internal architecture of the Zynq's PL part[23].

The CLBs contains in general the basic components that can be used to build complex logic circuits for example the Lookup Table (LUT) to implement (i) a logic function of up to six inputs; (ii) a small Read Only Memory (ROM); (iii) a small Random Access Memory (RAM); or (iv) a shift register. Another known component that the CLBs have is Flip-flop (FF), and it's widely used in creating basic logic circuits.

Input / Output Blocks (IOBs)— IOBs are resources that provide interfacing between the PL logic resources, and the physical device 'pads' used to connect to external circuitry. Each IOB can handle a 1-bit input or output signal. IOBs are usually located around the perimeter of the device.

Extended information on the PL resources can be found in the Zynq Book [23], like the DSP48E1s and Block RAMs. Also the reader will find the logic External Interfaces, the General purpose Input/Output and more.

3.3 Sensor specifications: InvenSense MPU6050

We have discussed in the previous chapter MEMS IMUs and their structures in addition to applications where this sensor type is used, we now start the description of the

used IMU in our project and its specifications.

Invensense claims that the MPU6050 is the world's first integrated 6-axis Motion-Tracking device. The sensor combines a 3-axis gyroscope, 3-axis accelerometer, and a digital Motion Processor (DMP) all in a small 4x4x0.9mm package fig7. It accepts also a 3-axis compass to be added and integrated using an I2C communication protocol to achieve a complete 9-axis Motions-Fusion output.

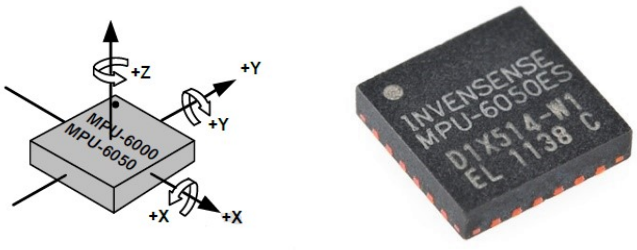

Figure 3.8: The InvenSence MPU6050 package[24]

The MPU-6050 features three 16-bit analog-to-digital converters (ADCs) for digitizing the gyroscope outputs and three 16-bit ADCs for digitizing the accelerometer outputs. For precision tracking of both fast and slow motions, the parts feature a user-programmable gyroscope full-scale range of ±250, ±500, ±1000, and ±2000°/sec (dps) and a user-programmable accelerometer full-scale range of ±2g, ±4g, ±8g, and ±16g.

Communication with the sensor registers is performed using I2C communication protocol at 400 kHz. Also the MPU6050 includes an embedded temperature sensor and an on-chip oscillator with ±1% variation over the operating temperature range.

The evaluation board that contents the Mpu6050 and the one we used is the GY-521 (Fig). This sensor board has a voltage regulator. When using 3.3V to the VCC the resulting voltage (after the onboard voltage regulator) might be too low for a good working I2C bus. It is preferred to apply 5V to the VCC pin of the sensor board. The board has pull-up resistors on the I2C-bus. We can find a numerous evaluation board designers that contains the MPU6050 in the market for example Sparkfun SEN-11028, Drotek MPU-6050 Invensense, Flyduino MPU6050 Break Out onboard and others.

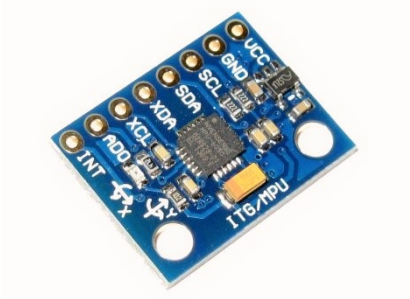

Figure 3.9: The GY-521 evaluation Board with MPU6050 at the middle

3.3.1 MPU6050 Gyroscope features

The triple-axis MEMS gyroscope in the MPU-60X0 includes a wide range of features:

- Digital-output X-, Y-, and Z-Axis angular rate sensors (gyroscopes) with a user programmable full-scale range of ±250, ±500, ±1000, and ±2000°/sec
- External sync signal connected to the FSYNC pin supports image, video and GPS synchronization
- Integrated 16-bit ADCs enable simultaneous sampling of gyros
- Enhanced bias and sensitivity temperature stability reduces the need for user calibration
- Digitally-programmable low-pass filter
- Gyroscope operating current: 3.6mA
- Standby current: 5µA
- Factory calibrated sensitivity scale factor
- User self-test

Further specifications can be found at Appendix B.

3.3.2 MPU6050 Accelerometer features

The triple-axis MEMS accelerometer in MPU-60X0 includes a wide range of features:

- Digital-output triple-axis accelerometer with a programmable full scale range of ±2g, ±4g, ±8g and ±16g
- Integrated 16-bit ADCs enable simultaneous sampling of accelerometers while requiring no external multiplexer
- Accelerometer normal operating current: 500µA
- Low power accelerometer mode current: 10µA at 1.25Hz, 20µA at 5Hz, 60µA at 20Hz, 110µA at 40Hz
- Orientation detection and signaling
- Tap detection
- User-programmable interrupts
- High-G interrupt
- User self-test

Other specifications can be found at Appendix B.

3.4 Inter-Integrated Circuit (I2C) communication protocol

The MPU6050 uses the I2C communication protocol as a medium to interact with processing units, we introduce the I2C communication protocol used for accessing the MPU6050 internal registers to properly configure and extract sensing data from them.

I2C is a multi-master, multi-slave, single-ended, serial computer bus invented by Philips Semiconductor (now NXP Semiconductors)[25]. It is typically used for attaching lower-speed peripheral ICs to processors and microcontrollers. I2C is a two-wire interface that comprises of two signals, serial data (SDA) and serial clock (SCL). In general, the lines are open-drain and bi-directional (Half Duplex). In a generalized I2C interface implementation, attached devices can be a master or a slave. The master device broadcasts the slave's address on the bus, and the slave device with the matching address acknowledges the master of his presence, allowing the master to begin the transmission, the bus then is occupied until the end of transaction session.

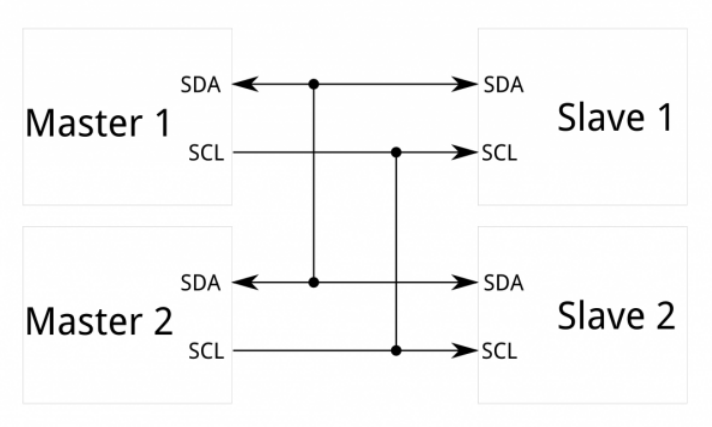

Figure 3.10: I2C bus structure with two masters and two slaves

ICs using I2C bus can operate at different speeds denoted as modes. We can find in the industry for a bidirectional bus:

- Standard-mode (Sm), with a bit rate up to 100 kbit/s
- Fast-mode (Fm), with a bit rate up to 400 kbit/s
- Fast-mode Plus (Fm+), with a bit rate up to 1 Mbit/s
- High-speed mode (Hs-mode), with a bit rate up to 3.4 Mbit/s.

The MPU-6050 always operates as a slave device when communicating to the system processor, which thus acts as the master. SDA and SCL lines typically need pull-up resistors to VDD. The maximum bus speed is 400 kHz. The data line is bidirectional and hence is capable of sending and receiving data to and from multiple slave devices sent over the SDA bus with each slave device identified via its own unique 7-bit address. The data transmitted over the SDA line is divided into 8-bit (1 byte) packets.

The default slave address of the MPU-6050 is b110100X or 0x68 which is 7 bits long. The LSB bit of the 7 bit address is determined by the logic level on pin AD0. This allows two MPU-6050s to be connected to the same I2C bus. When used in this configuration, the address of the one of the devices should be b1101000 (pin AD0 is logic low) and the address of the other should be b1101001 (pin AD0 is logic high).

The details about the protocol's communication signaling will not be detailed, because in fact we shall adopt a Higher abstraction layer by implementation of user mode I2C

client drivers library named i2c-dev (see the end subsection 4.4.2).

Conclusion

By the end of the present chapter, we have setup a primary environment of the underlaying hardware where the model will be implemented on. The ZedBoard grants an open set of hardware peripherals, leading to a variety of design options. Specifications of the used sensor in our case were detailed and a brief description of the main communication protocol was seen.

4

System Configuration

In this chapter we are going through the design steps of our project.

The design choices will be justified upon discussing the available possibilities, then the Simulink Environment configuration steps will be detailed in conjunction to Simulink connection to the ZedBorad.

Once our environment is prepared, we begin the actual design details, until the obtention of the desired data from our sensor to the Simulink development platform.

4.1 Hardware System Development on Zynq

Due to the nature of the Zynq SoC, many configuration might be adopted; so we ought to choose between the different possible design configurations made available by it. We can cite some basic ones:

- Using the PS part for a Standalone (bare-metal) implementation.
- Using an Operating System running on the PS with the desried application executed on top of it.
- Using the PL part.
- Using a co-design with both the PS and the PL parts interconnected with AXI bus.

Since Simulink has everything needed to use the stated options of realizations, and that the project is constrained by a short time schedule; the second option appears to be a good choice to start with, as the Operating Sytem offers many advantages like the availability of interface drivers (mainly for I2C controllers, Ethernet, SPI, UART,...etc.) and the fact that the mainstream use of the type of applications in our case is within Embedded Operating Systems, where they are loaded in conjunction to many other user applications; take for example a cellphone that uses IMU data to stabilize the camera's snapshots, visualize 360^o scenes (Google CardBoard app [26]) or a quadcopter that uses the data to make decisions and issue commands to its motors for hovering stabilization.

A kernel in this case simplifies communication between different loaded processes (user applications) running at the same time and dependent on each other and provides them with access to peripherals. Moreover, the existence of the OS kernel gives the possibility for extension in a recursive way and flexibility to integrate updated functionalities with the rest of the system as well as portability to different embedded OS flavors (like the

Raspberry-Pi and the Beaglebone).

Design Chart:

The software system can be thought of as a stack, or set of layers, which is built upon the **Hardware Base System** of the ZedBoard as shown in the figure below:

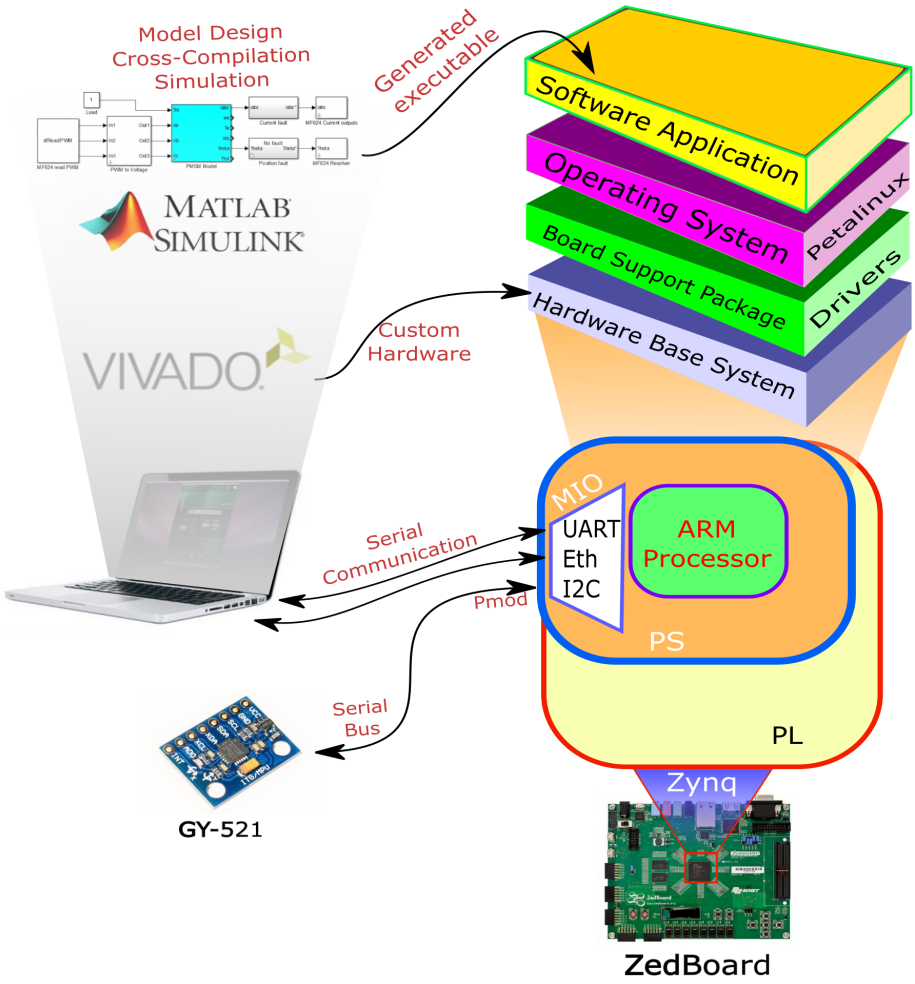

Figure 4.1: Overview of hardware connections

The 'hardware base system' or 'hardware platform' can be exported from Vivado and it represents the customised hardware upon which the software is based. The layer directly above the Hardware Base System is the **Board Support Package (BSP)**, the set of low-level drivers and functions that are used by the next layer up: the **Operating System** (Petalinux) to communicate with the hardware.

Software Applications run on top of the Operating System and represent the uppermost layer in the software stack and the highest level of abstraction from hardware. Simulink will be used to design our model, cross-compile it and charge it to run in the Software Application layer; as well as real time simulations that involve interaction be-

tween Simulink and the OS.

Before we give details about the used OS and begin preparing it, we should first setup Simulink to build our model and load it on the ZedBoard.

4.2 Setting up the Simulink environment for the ZedBoard

In order for Simulink to deploy compiled binaries on the Zedboard we need to prepare the environment by installing the Embedded Coder packages, applying the right modification for simulation settings and preparing the MATLAB workspace to communicate with the board.

Before beginning any configuration, the user should install the Xilinx development suite available on its website, including Xilinx ISE and Xilinx Vivado. The first one will provide us with the Xilinx Software Development Kit(SDK) which contains Xilinx ARM tool-chains as they are necessary for cross-compilation, we mean by cross-compilation compiling code intended to be executed on a different instruction-set architecture than the host machine where we perform the compilation (in our case we have an x86 host and cross-compile for an ARM architecture).

Vivado on the other hand is used to create a hardware description file that contains the configuration of the Hardware Base System layer as shown in Fig 4.1. We will supply this file for configuring the Operating System before its building process.

4.2.1 Installation and Setup of ZedBoard support packages for MATLAB/Simulink

MathWorks provides a support package called Xilinx Zynq Design Package. This package makes it easier to program on Zynq platforms,it provides a framework for integrated hardware/software design, simulation and verification. This makes it more practical to integrate Model-Based Design into your workflow, since it enables fast design iteration cycles and helps to detect and correct design and specification errors early.

The support package for the Zedboard may be downloaded free of charge from the MathWorks website or installed via the MATLAB command line via the command “targetinstaller”. Select “Xilinx Zynq-7000” from the Support Package Installer window. Note that a MathWorks account ID is required for this method of installation. The support package installer is shown below.

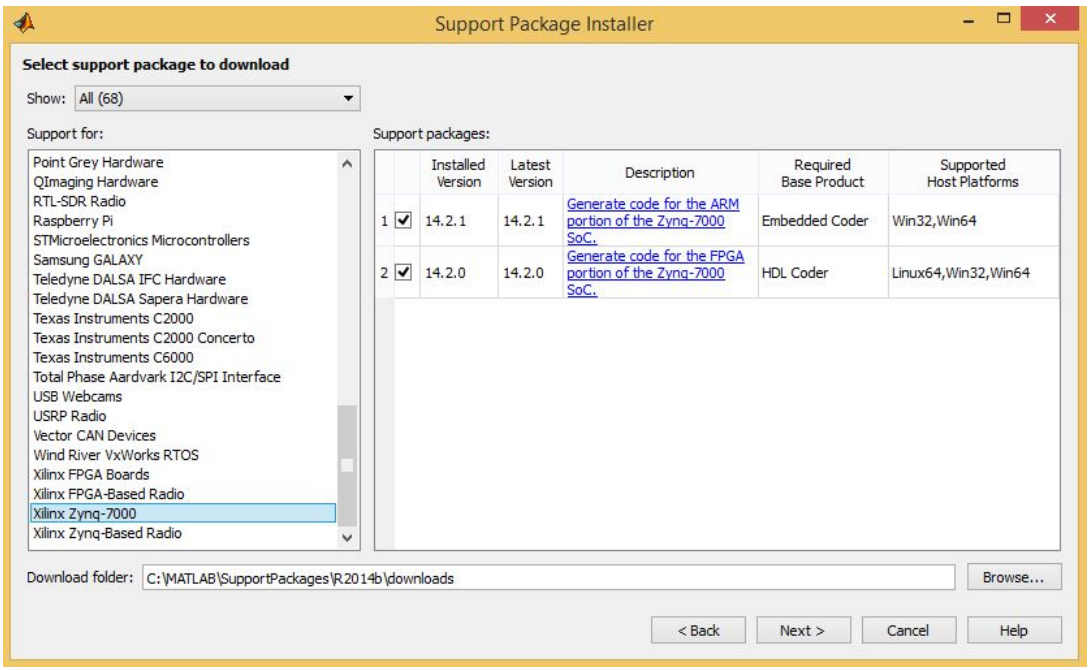

Figure 4.2: The Support Package Installer Window

Next, the toolchain used by Simulink must be set-up. This is carried out via the configuration of a "makefile" which essentially specifies the used software within the toolchain allowing for an automated build process. Typing "xmakefilesetup" in the MATLAB command window brings up the XMakefile User Configuration window. For MATLAB versions 2014b configure the "Template" and "Configuration" options as shown below:

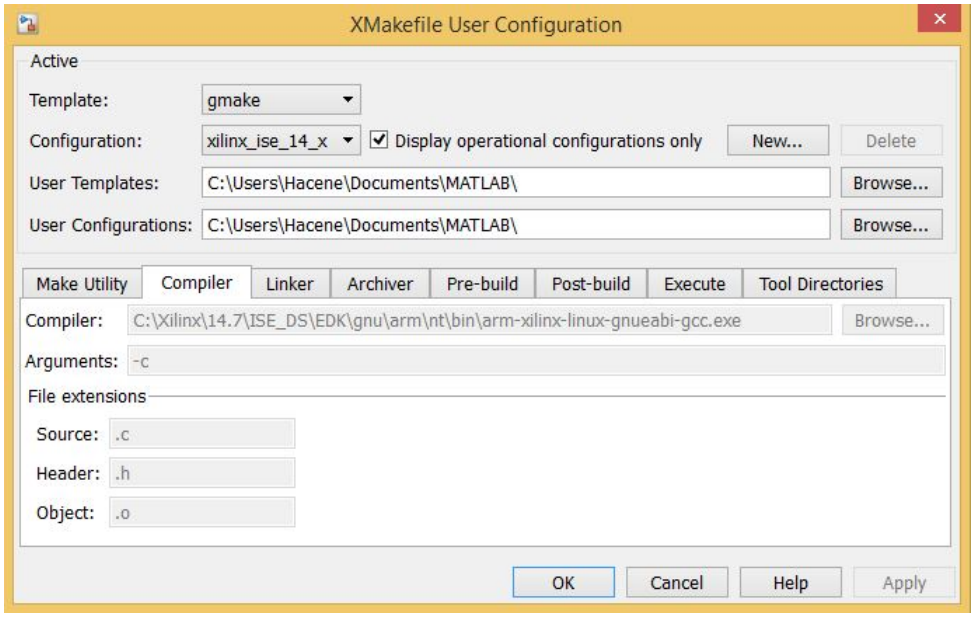

Figure 4.3: The XMakefile User Configuration window

It is important to check that Simulink uses the right tool-chains for our target, a good advice is to use Xilinx tool-chains for compatibility issues.

Checking C Compiler availability and compatibilty:

To make sure that a proper compiler is installed in our system and that MATLAB is using it for code compilation, we enter the following command in MATLAB Command Window:

```
Command Window
>> mex -setup
MEX configured to use 'Microsoft Windows SDK 7.1 (C)' for C language compilat
Warning: The MATLAB C and Fortran API has changed to support MATLAB
     variables with more than 2^32-1 elements. In the near future
     you will be required to update your code to utilize the
     new API. You can find more information about this at:
     http://www.mathworks.com/help/matlab/matlab_external/upgrading-mex-files

To choose a different language, select one from the following:
 mex -setup C++
 mex -setup FORTRAN
fx >>
```

If no C compiler is found on the host PC or a non-compatible one is installed, the user will have to fetch a compatible one as indicated in the Mathworks website [27]. We should note that every version of MATLAB has different compatible versions of compilers.

4.3 Zedboard interface configuration

In this section we highlight and map the essential interface configurations needed to establish the communication between the sensor and the board on one hand, and between the board and the PC on the other hand (Fig 4.1).

Hardware configuration will be performed using Xilinx Vivado in order to create a Harware Description File (.hdf); this last one will be provided later to build the Linux image.

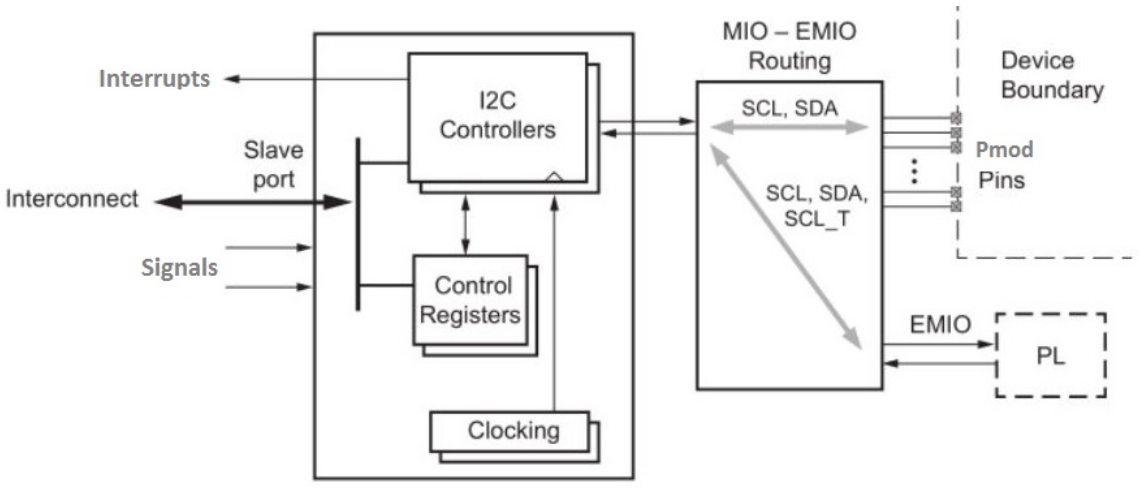

Figure 4.4: I2C connection routed from the PS through the Pmod to an external device

For I2C connected modules, the Pmod digital signal characteristics conform to the I2C specification. Digilent system boards (including the ZedBoard) operate at 3.3V, and the

modules are primarily intended for operation at 3.3V. Pmod modules support other communication standards like UART, SPI, GPIO [28]. Figure 4.4 shows the I2C connections from the PS through the Pmod to an external device.

The top most Pmod connctor (JE1) shown in Fig 3.2 is connected to the PS part. Pmod connectors have twelve-pins and provide eight digital I/O signal pins, two 3.3V power pins and two ground pins. Figure 4.5 shows the JE1 Pmod pins schematic.

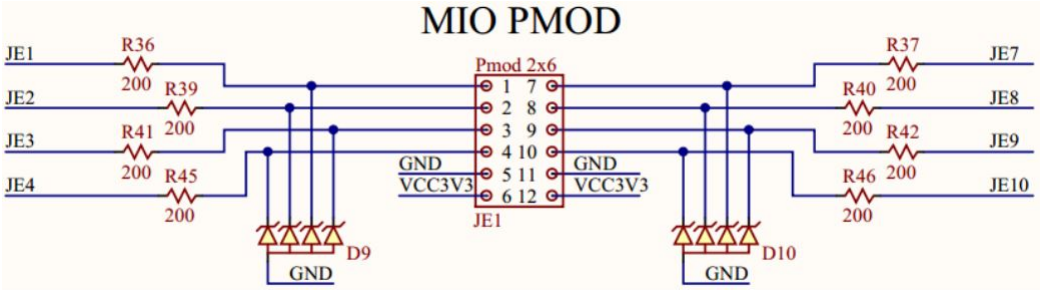

Figure 4.5: JE1 Pmod pins schematic

The Multiplexer Input Output (MIO) block is divided into many bancs, JE1 Pmod connector is plugged to the Banc 500 which provides connection to the two I2C controllers (Fig 4.4). Fig 4.6 shows as well the connections mapping.

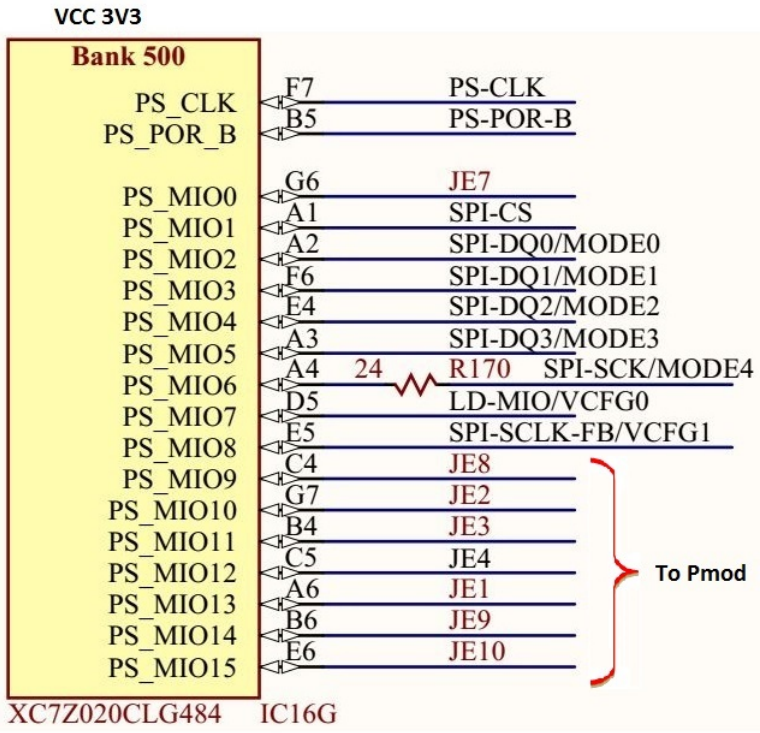

Figure 4.6: JE1 Pmod pins schematic

Based upon the previous interfacing descriptions, we have now the necessary knowledge to begin the generation of our Harware Description File on Vivado.

4.3.1 Generating the hardware description file using Vivado

First, we begin by opening the Vivado development tool and creating a new project, We name the project, choose an RTL project type and keep default parameters for rest of the steps. For the board type choice, we choose the "Zedboard Zynq Evaluation and Development Kit" (Fig 4.7).

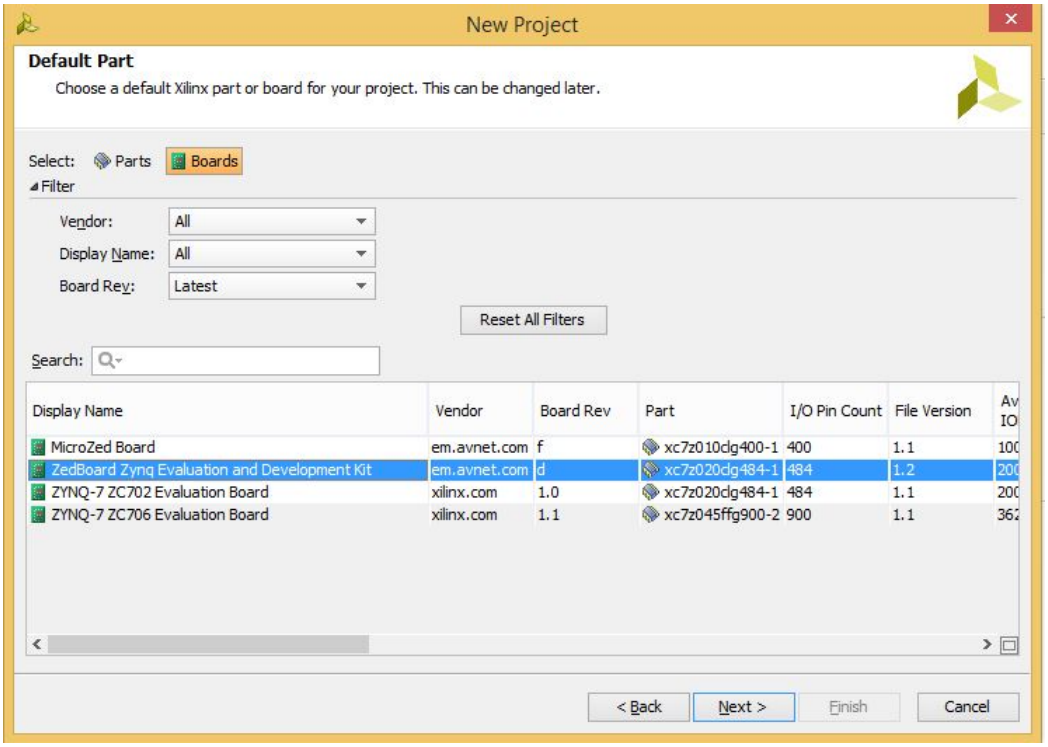

Figure 4.7: The board type choice

Once the main project window is open, we click on "Create Bloc Design" at the "Flow Navigator" frame and we validate by "OK".

A new blanc "Diagram" window appears allowing us to begin setting in our IP blocks.

We right-click with the mouse and choose "Add IP..." from the menu, we type then on the popped small window "ZYNQ7 Processing System" Fig (4.8) and we validate. A new block appears as shown on Fig 4.9

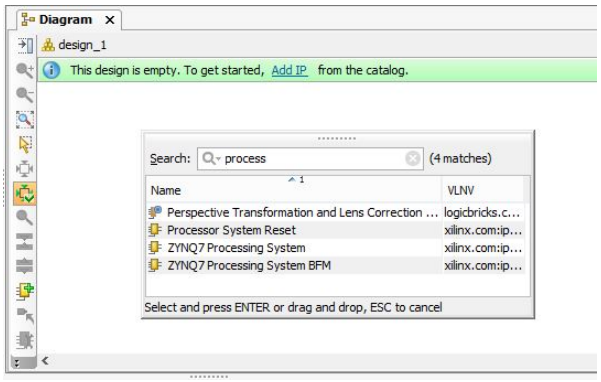

Figure 4.8: We choose "ZYNQ7 Processing System"

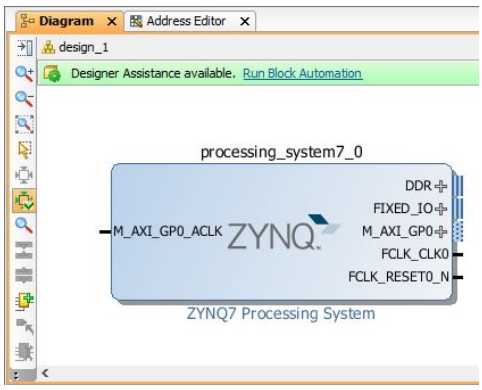

Figure 4.9: The ZYNQ7 Processing System block

To speed things up we click on "Run Bloc Automation" in order to auto-generate a minimal configuration; for example default routing of Ethernet, UART, connections to DDR Memory, etc. We then open the block and add the proper configuration of the I2C controllers interfacing according to the previous described connections at the beginning of subsection 4.3 as shown in Fig 4.10.

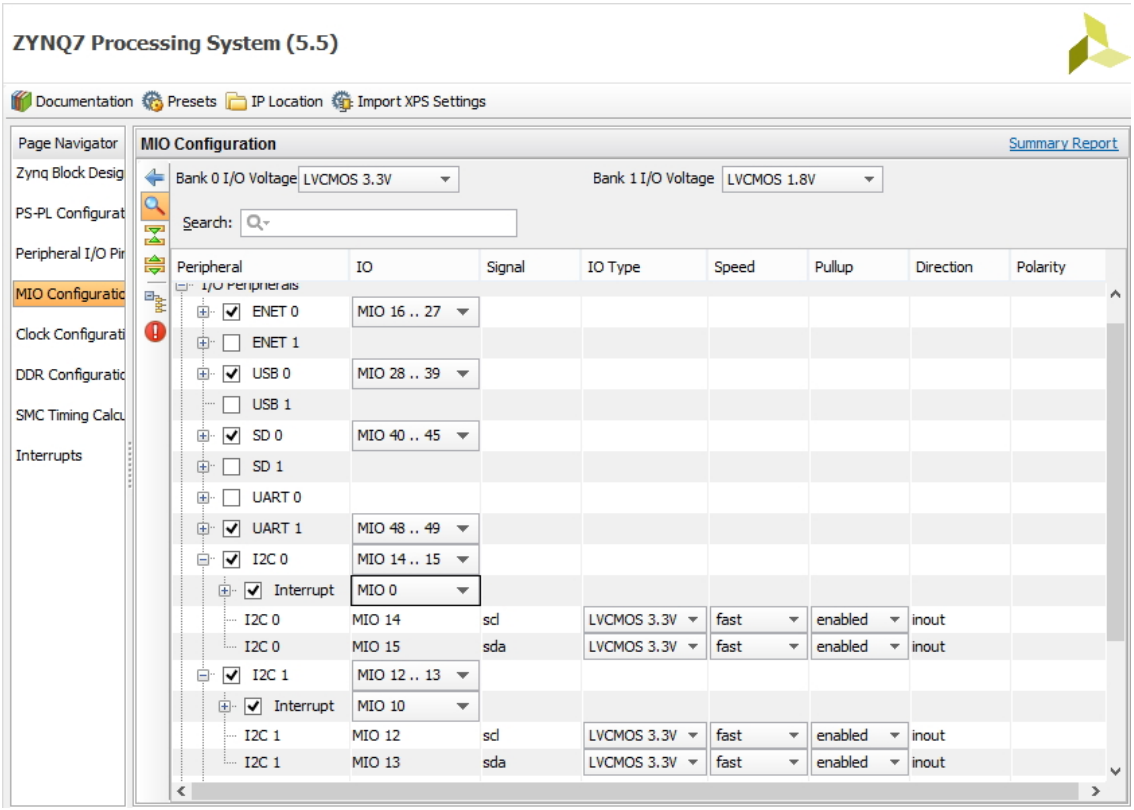

Figure 4.10: configuration of the I2C controllers interfacing

As we are not going to use any AXI inerfaces ,we uncheck the box of "M AXI GP0 Interface" (the input clock for the single configured AXI bus on the PS) in the "PS-PL Configuration" tab (Fig 4.11).

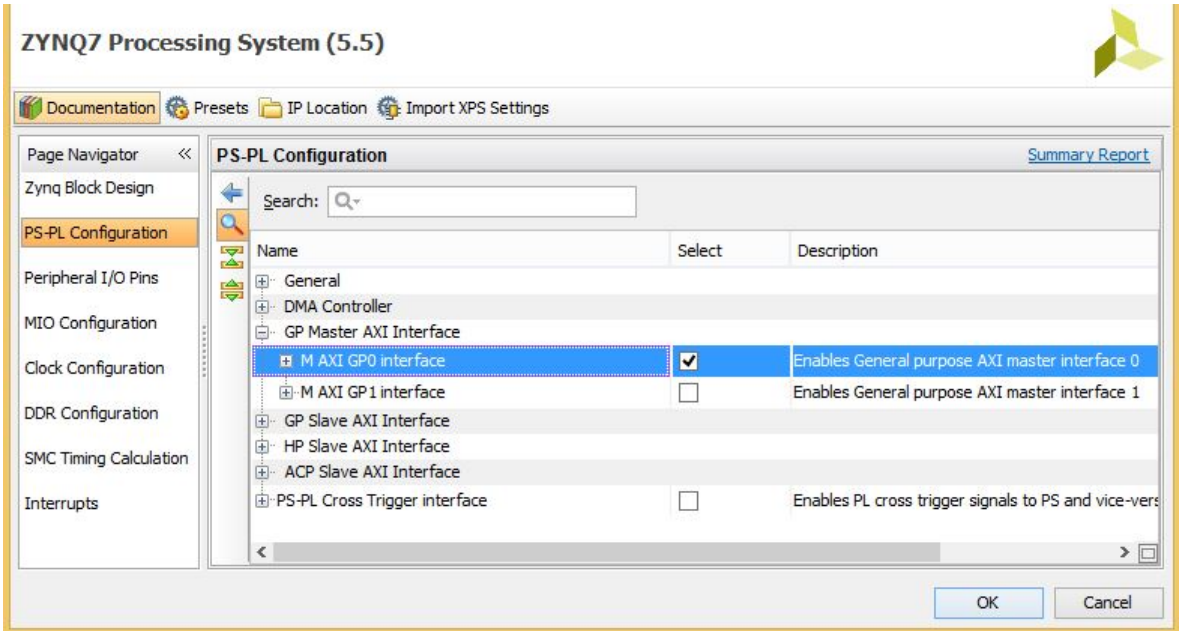

Figure 4.11: We uncheck the box "M AXI GP0 Interface"

An overview of the applied modifications can be seen in Fig 4.12 and the final resulting block configuration is shown in Fig 4.13.

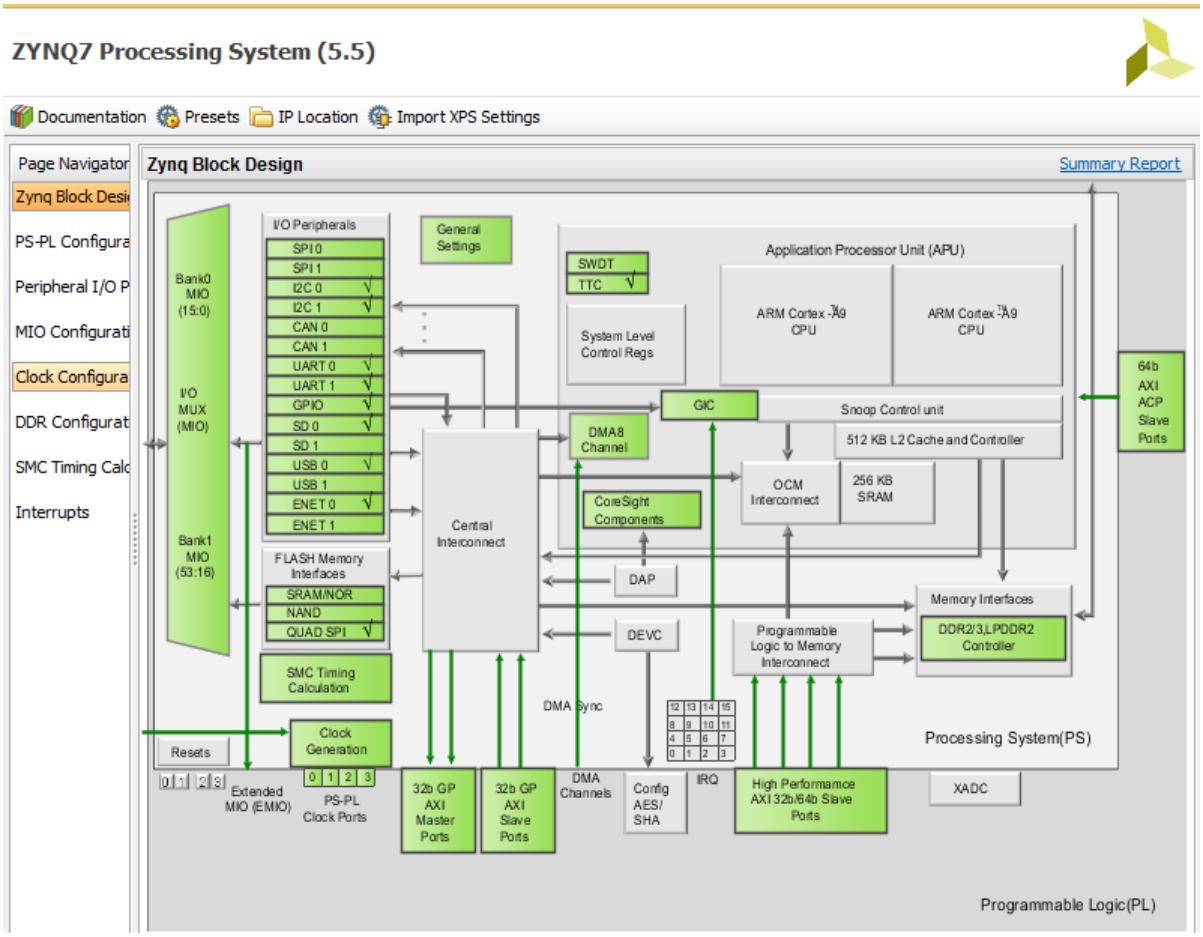

Figure 4.12: Overview of the applied modifications

Now as we have completed the task, we can export our configuration as an hdf file by clicking on the top "File" menu, then choose: "Export" $\Longrightarrow$ "Export Hardware..." . The resulting file has as a name "$< project\_name > .hdf$"

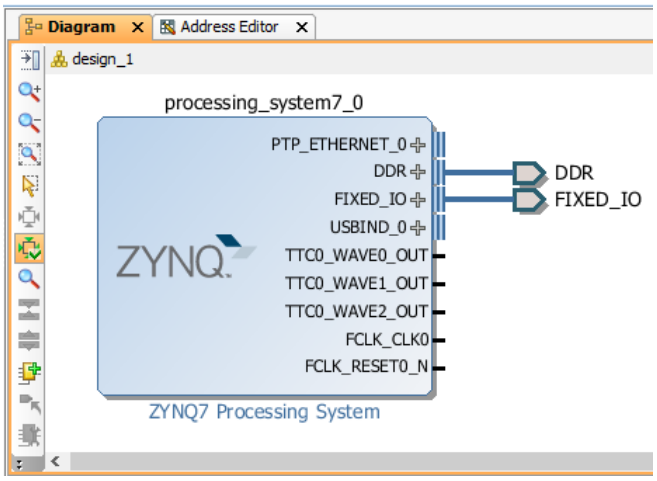

Figure 4.13: The final block configuration

4.4 Embedded Linux Setup

The ZedBoard support software was built so as to allow for deployment of Simulink models to the target hardware running a Petalinux embedded OS. As such, a Petalinux image must be prepared to boot from the external SD card on the Zedboard.

We have first to emphasis on the fact that the prebuilt image of Petalinux OS present on Xilinx website lacks many drivers as in our case the I2C controllers definitions, thus the OS wouldn't detect any I2C controller. So, as a first step we managed to build a new Petalinux image including the lacking interface definitions.

Xilinx provides Petalinux Tools which offers everything necessary to customize, build and deploy Embedded Linux on Xilinx processing systems. We download and install Petalinux Tools on a Virtual Machine (VM) hosting a Linux Ubuntu 14.04 x64 destrinbution with Xilinx development tools installed on it.

A summary of the necessary steps to have a completely configured and ready-to-use Petalinux image is described at the chart below:

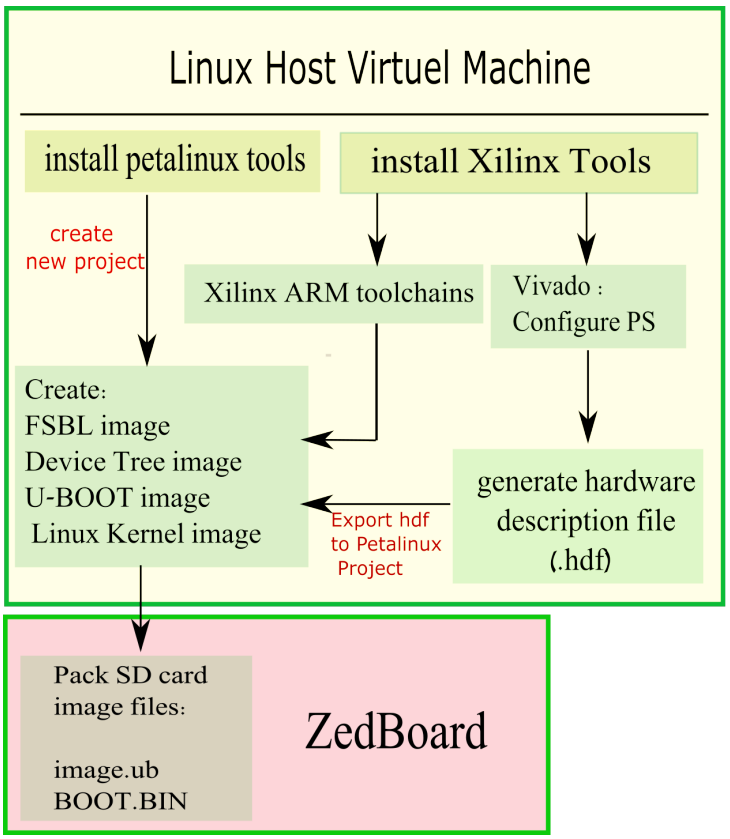

Figure 4.14: Summary of Petalinux building procedure

4.4.1 Petalinux building steps

Petalinux release package is downloaded for Petalinux building :

petalinux-v2014.4-final-installer.run

We install this package as shown below, an installation path may be specified, as example */opt/pkg*, root privileges must be acquired :

```
$ mkdir /opt/pkg
$ ./petalinux-v2014.4-final-installer.run /opt/pkg
```

After the installation, the remainder of the setup is completed automatically by sourcing the provided "settings" scripts, in order to update the environment variables of the host system:

```
$ source <path-to-installed-PetaLinux>/settings.sh
```

We indicate the path to the Xilinx installation as the Xilinx ARM toolchains are going to be used to cross-compile the linux kernel:

```
$ source <path-to-installed-Xilinx-Vivado>/settings64.sh
```

Now we create a new Petalinux Tools project in a separate folder

```
$ mkdir MyProjDir
$ cd MyProjDir
$ petalinux-create –type project –template zynq –name Petlinux_image
$ cd Petlinux_image
```

Here, the tool creates a new project folder named *Petlinux_image* that includes many source and configuration files; we put our privious *hdf* file in the root folder of our project as it is going to be used to configure the kernel device tree by adding "- -get-hw-description" option to the next command:

```
$ petalinux-config - -get-hw-description
```

A graphical window pops-up presenting various additional configuration options, we keep the default settings by exiting.

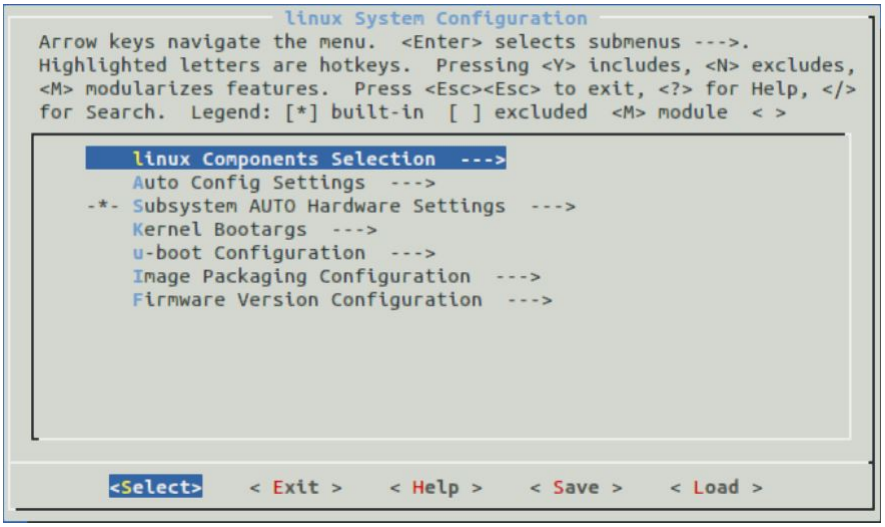

Figure 4.15: Linux System Configuration window

We are going to add packages in order to verify our I2C interfaces configuration:

```
$ petalinux-config -c rootfs
```

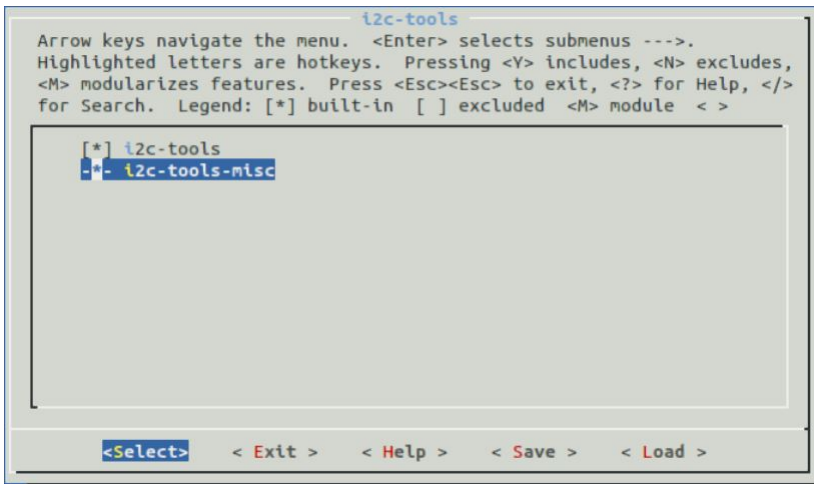

Figure 4.16: Adding program packages in *linux/rootfs* Configuration window

We navigate to $FilesystemPackages \Rightarrow Base \Rightarrow i2c-tools$, and we select "i2c-tools", we save our settings and validate.

Finally, we can now build our Petalinux image by executing the following command:

```
$ petalinux-build
```

The building process will take a while depending on the host hardware ressources, in our case it took around 15 min. Once the system image is built we pack the necessary files to create an SD card bootable image. We should indicate the relative placement of the generated first stage bootloader (fsbl) image from the project folder:

```
$ petalinux-package –boot –fsbl images/linux/zynq_fsbl.elf –u-boot
```

We now only need to copy system files to the SD card:

- BOOT.BIN: Contains a FSBL image, a device tree configuration, a root filesystem image and the U-Boot bootloader which is going to load the kernel system image on the board memory.
- image.ub: The U-Boot kernel system image.

4.4.2 Booting Petalinux image on ZedBoard and configuration checking

Before powering the Zedboard on, we should set its jumpers configuration as shown in the picture (Fig 4.17) in order to set the SD card as a boot device. Fig 4.18 shows the IMU connected to the JE1 Pmod. the connections are as follows:

MPU6050	ZedBoard's JE1 Pmod connector
VCC	VCC
GND	GND
SDA	JE1
SCL	JE4
INT	JE2

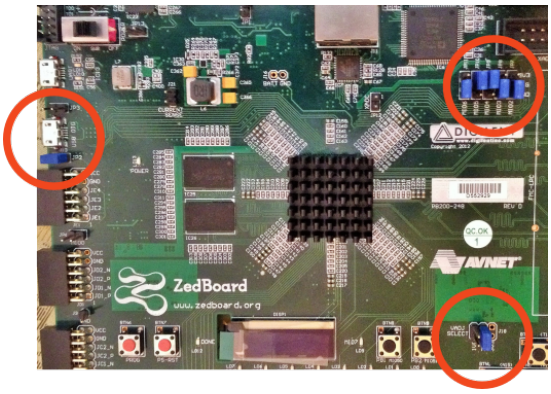

Figure 4.17: the ZedBoard jumpers set to boot from the SD card on startup

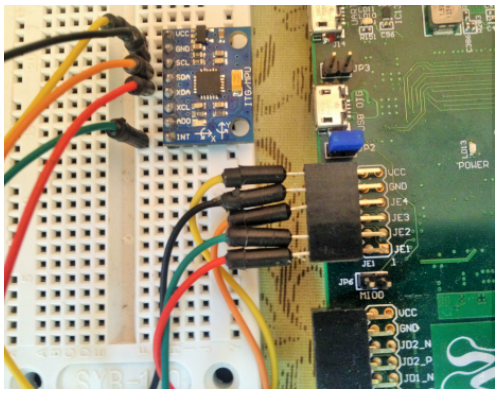

Figure 4.18: The MPU6050 plugged into the JE1 Pmod

At the host computer we prepare a serial communication software (using Putty for example), we power on the board and begin a communication session. We observe the boot sequence of Petalinux before being prompted to enter the user and password login.

We can see that the kernel has reconized the I2C controllers and created node files for each one in /dev directory, as observed in Fig 4.19.

COM4 - PuTTY

```
root@good_kernel:~# ls /dev
console             ram10               tty20               tty49
cpu_dma_latency     ram11               tty21               tty5
flash               ram12               tty22               tty50
full                ram13               tty23               tty51
i2c-0               ram14               tty24               tty52
i2c-1               ram15               tty25               tty53
iio:device0         ram2                tty26               tty54
initctl             ram3                tty27               tty55
input               ram4                tty28               tty56
kmsg                ram5                tty29               tty57
loop-control        ram6                tty3                tty58
loop0               ram7                tty30               tty59
loop1               ram8                tty31               tty6
loop2               ram9                tty32               tty60
loop3               random              tty33               tty61
```

Figure 4.19: *ls /dev* command result showing reconized peripherals

i2c-tools added packages allow us to perform a quick communication check on command line terminal. These packages are based on the i2c-dev standard libraries which are used as kernel drivers for I2C protocol, thus simplifying the communication by providing a higher abstraction of protocol signaling since it is built from C/C++ functions. The figure below 4.20 illustrates the existing relations between different layers for a typical

I2C communication .

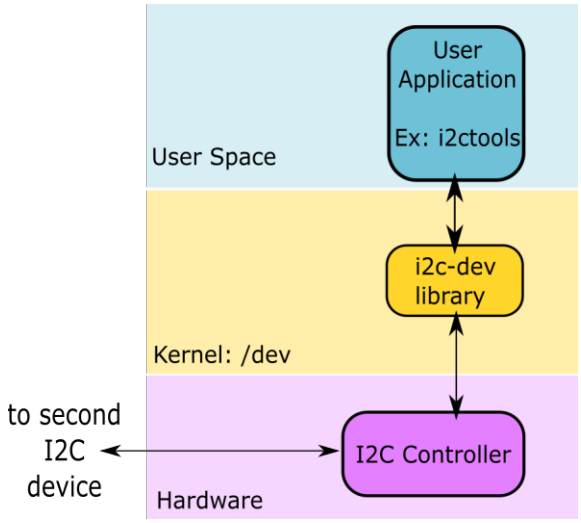

Figure 4.20: I2C Communication presented in different layers.

$i2cdetect - l$ lists the available I2C controllers and gives their peripheric adresses. On the other hand, when $i2cdetect - r < ctrlr\_nbr >$ is called it scans for the presence of devices on the given bus number as shown in Fig 4.21. The sensor has 0x68 as an address (as mentioned before).

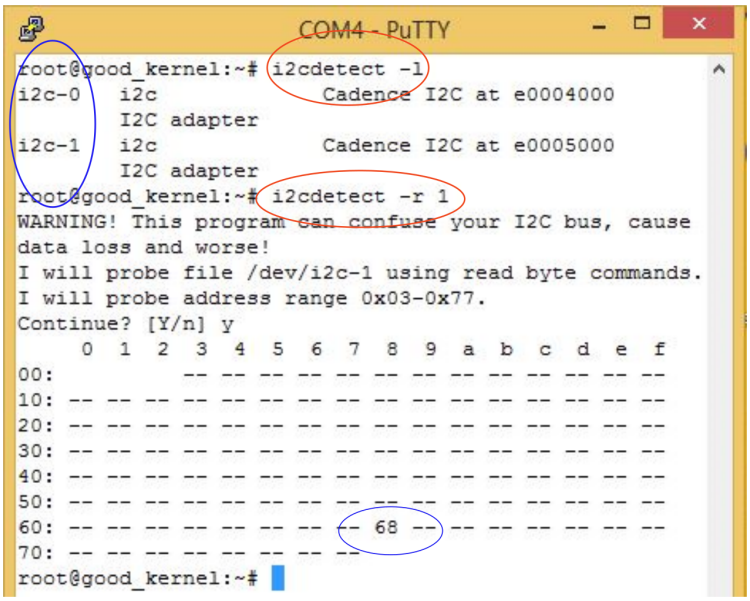

Figure 4.21: The use of using *i2cdetect*

We have concluded our hardware configuration setup as the ZedBoard is communicating properly with the sensor. We can focus on getting Simulink environment ready to deploy our design on the board.

4.5 Setting up Simulink Code Generation and Solver parameters

After we set the Zynq platform to use petalinux, we set the proper configuration in Simulink to generate equivalent C code of the Simulink model, download it to ZedBoard, compile, build and run it on the target hardware. The process also allows Simulink to continually read and write data variables while the model is executing on the target board.

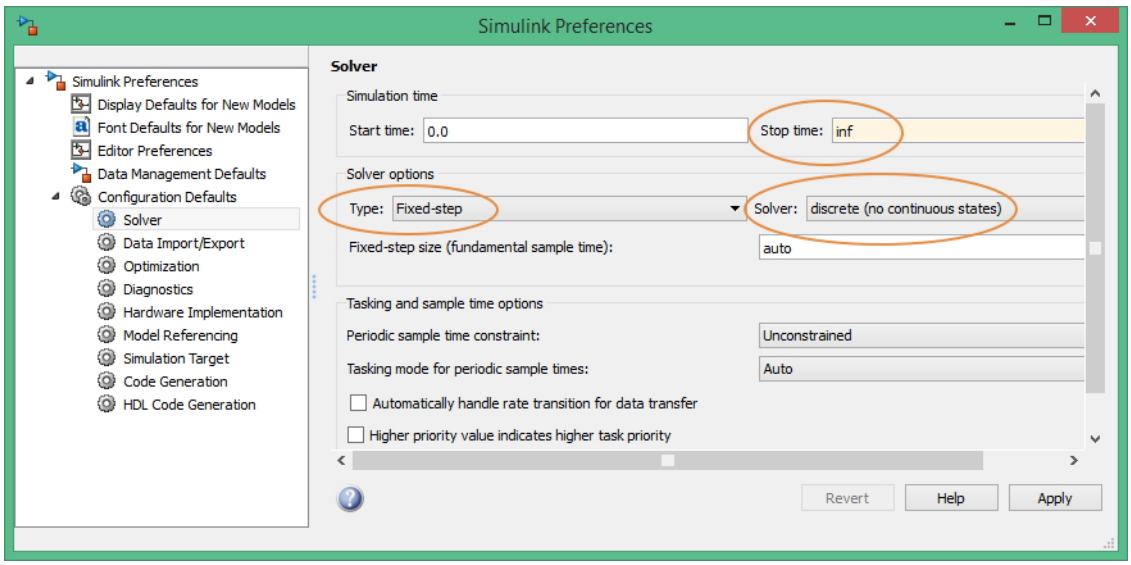

Figure 4.22: Solver configuration

Steps :

We open MATLAB Preferences and choose Simulink ⇒ Launch Simulink Preferences.

First, we configure the Solver under Configuration Defaults tab as follows:
Next, to generate and deploy C code for our board we must make these changes to the Code Generation configuration tab (Fig 4.23).

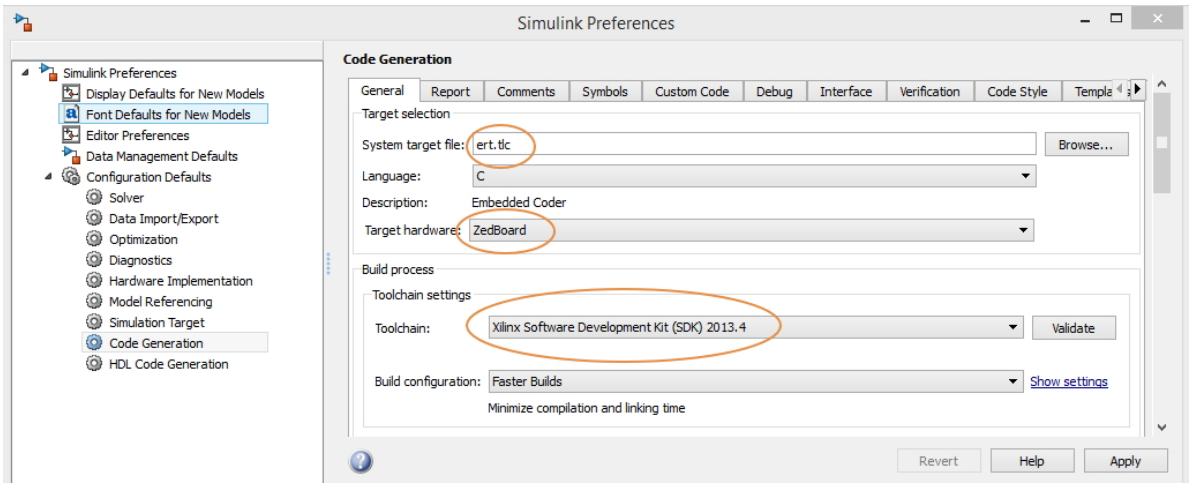

Figure 4.23: Solver configuration

4.5.1 Establishing the connection ZedBoard ⇔ MATLAB

We connect the ZedBoard to the host PC using an Ethernet cable and make sure that the two devices are on the same network address.

Typing the next command on the MATLAB Command Window is the latest thing to do in order to establish a connection between the ZedBoard and MATLAB:

```
Command Window
>> zynq

ans =

  LinuxServices with properties:

     IPAddress: '192.168.1.8'
      Username: 'root'
    SerialPort: 'COM4'
```

We can see information of the established connection, like the Ethernet adapter's IP address, the session username and the chosen serial port.

4.6 Development of S-Function Driver Block for MPU6050

Development of the MPU6050 specific driver block for Simulink allows for live interaction between the target hardware and a Simulink model. This method can be used as a template for gaining the ability to interact with the physical world via the ZedBoard's peripherals. An extension of this is to control the behaviour of a system based on sensed inputs from the surrounding environment.

The driver block harnesses the ZedBoard's ability to read digital or analog data from sensors as well as control physical objects through analog and digital outputs. Therefore sensing (input) drivers and actuating (output) drivers may be combined with a well-designed control algorithm to rapidly prototype autonomous control of a user defined system.

To do so, we begin designing a news Simulink model and rely on a fundamental block which is the S-Function Builder block that we can find in the Simulink Library Browser under Libraries ⇒ User-Defined Functions ⇒ S-Function Builder. Essentially, the S-Function builder allows the user to create a MATLAB executable file (MEX) from source code written in C. Typically, this means that a user can create blocks to be run in simulation mode using pre-existing C functions and libraries which are readily available and somewhat more prevalent than existing MATLAB solutions.

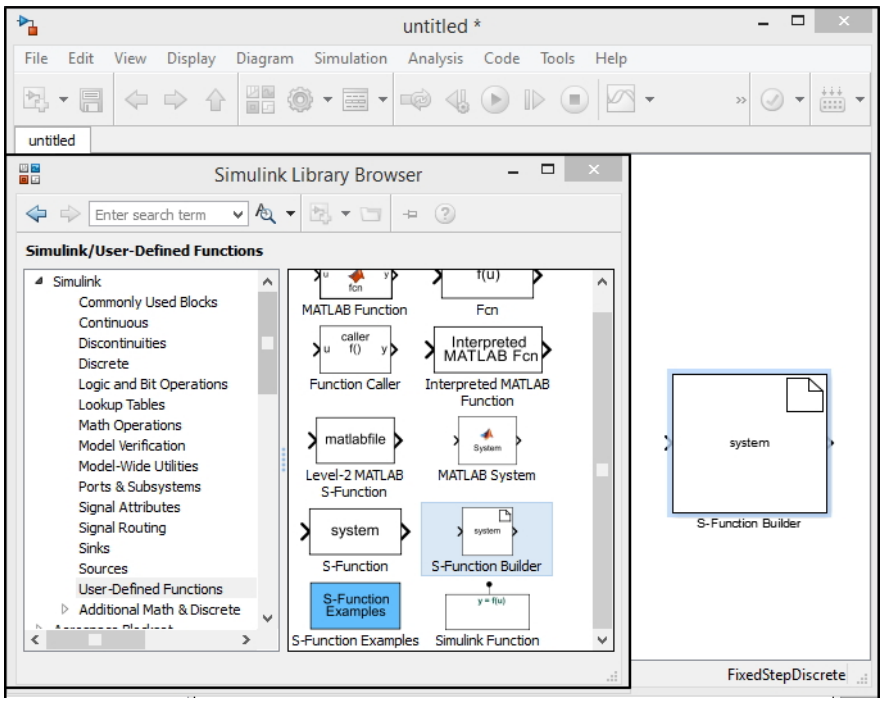

Figure 4.24: S-Function Builder Bloc found under User Defined Functions Category

In this application, the C-code wrapped by the S-Function Builder is used to interact with the Zynq's Petalinux OS via its virtual file system, commonly referred to as "sysfs". This file system exposes the various drivers defined on the OS allowing for read/write access to peripherals.

It should be noted, that code developed for the driver blocks do not execute on the host computer in simulation mode. They are, in fact, executed only on the specified target hardware when Simulink is placed into "External Mode".

S-Function Builder Configuration

One must simply double click thee S-Function Builder block inserted into a new Simulink model to open its configuration box. It is from this configuration dialog box that all parameters relating to the block must be specified and configured.

S-Function Parameters Panel

The parameters panel features the general configuration of the S-Function block. This includes the name to be given to the S-Function and will serve as the filename root for all files generated during the build process.

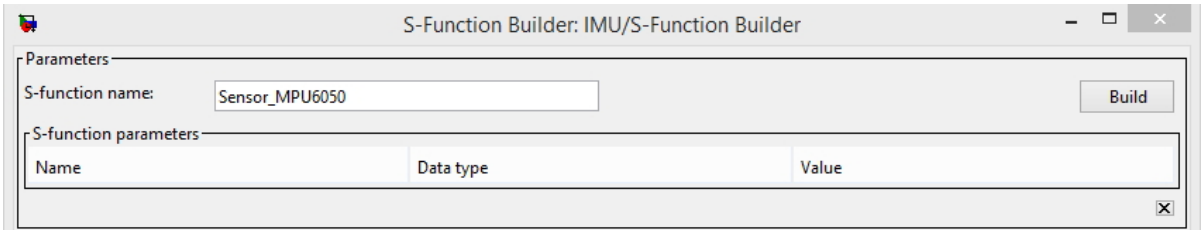

Figure 4.25: S-Function Builder Bloc: Parameter Panel.

S-Function Initialization Tab:

The Initialization pane establishes the block sample time and its number of continuous and discrete states. Given that the code is running on external hardware, the driver block executes in discrete time. Therefore no continuous states are to be configured..

This implementation of a driver block requires that we set at least a single discretetime state, which must be initialized to 0. One could add more states if needed, but the first element of the discrete state vector (Discrete State Initial Condition is xD[0]) must be initialized to 0 in order for the code in the initialization part (which we'll see in Descrtie Update tab) to work.

It is important to know that the MPU6050 readings' sampling rate is by default 1 kHz for its Accelerometer and 8 kHz for its Gyroscope, so choosing sample time values less than 1 μs will result in redundant readings of the Accelerometer's measurments.

Designers would want to configure the sensor's registers in order to have an optimal sensor readings' sampling rate for power consumption considerations; viewing that the sensor sinks too little current when it is put in standby mode during concurrent readings, ($3.9mA$ in normal mode versus around $10\mu A$ in standby mode for the MPU6050) and power consumption depends directly to the chosen sampling frequency.

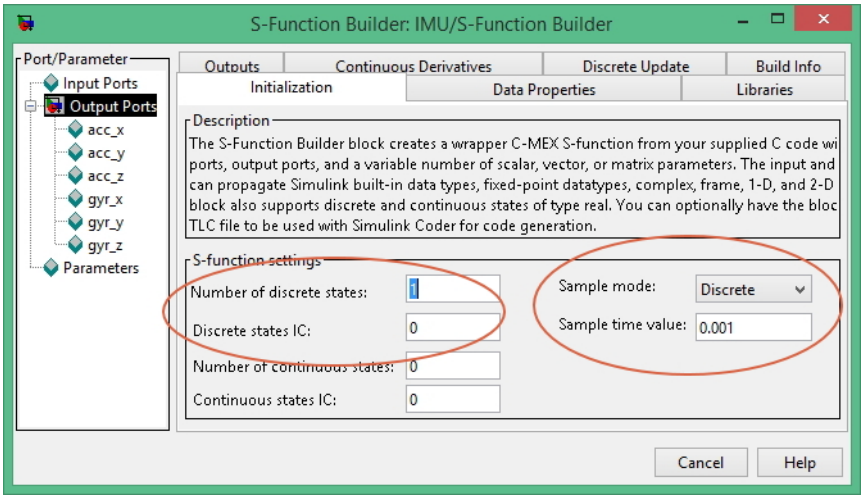

Figure 4.26: S-Function Initialization tab.

S-Function Data Properties Tab:

In this Tab we define our output signals knowing that we want to extract signal data from the sensor so the block will appear as an input block. Thus it will not include any inputs. Six digital signals are driven from the driver block representing the 3-axis Accelerometer and a 3-axis Gyro outputs. The figure below shows the given names and type for each signal.

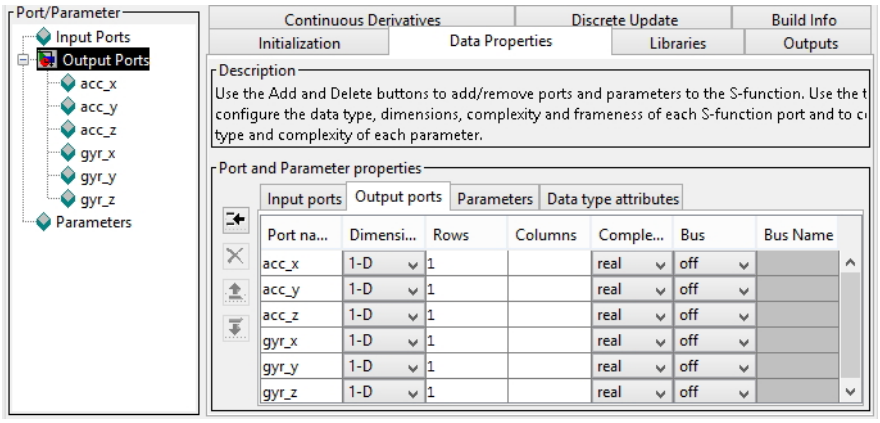

Figure 4.27: S-Function Builder Bloc: Data properites Tab.

In the **Data type attribute** tab we define the data type for each output.

S-Function Libraries Tab:

For our driver block we need to specify the necessary libraries and external source files located outside the MATLAB or project directory as well as the declaration of external functions and include files. Global variables may also be declared here.

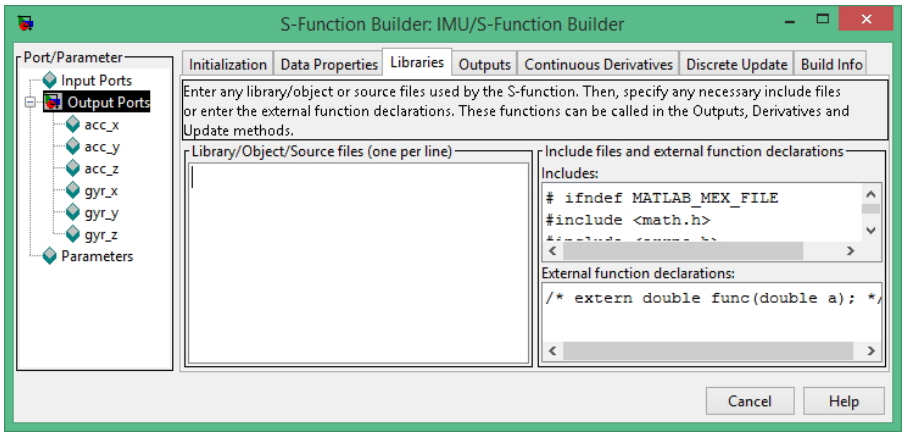

Figure 4.28: S-Function Builder Bloc: Libraries Panel.

In addition to the necessary standard libraries, two external libraries had to be included in the project folder. The first library "lsquaredc.c" and its header "lsquaredc.h" define the i2c-dev based functions that provide a higher abstraction level to interact with I2C buses . The second library "mpu6050.c" uses the previous library's functions to perform reading and writing operations on the sensor's registers which are defined in its header

file "mpu6050.h".

The libraries' source files were originally obtained from an existing project on GitHub website [29] with full rights for use and modification. The second library "mpu6050.c" with its header file "mpu6050.h" were modified because they didn't contain any code concerning the gyroscope's registers definitions, sensitivity computation and reading function calls. The necessary information for the coded variables and functions were obtained from the MPU6050's datasheet [30].

S-Function Outputs Tab:

The **Code Description** pane of the Outputs tab contains the main source C-code to be executed by S-Function block at every iteration in simulation. C-Code is typed directly into this pane by the block's designer.

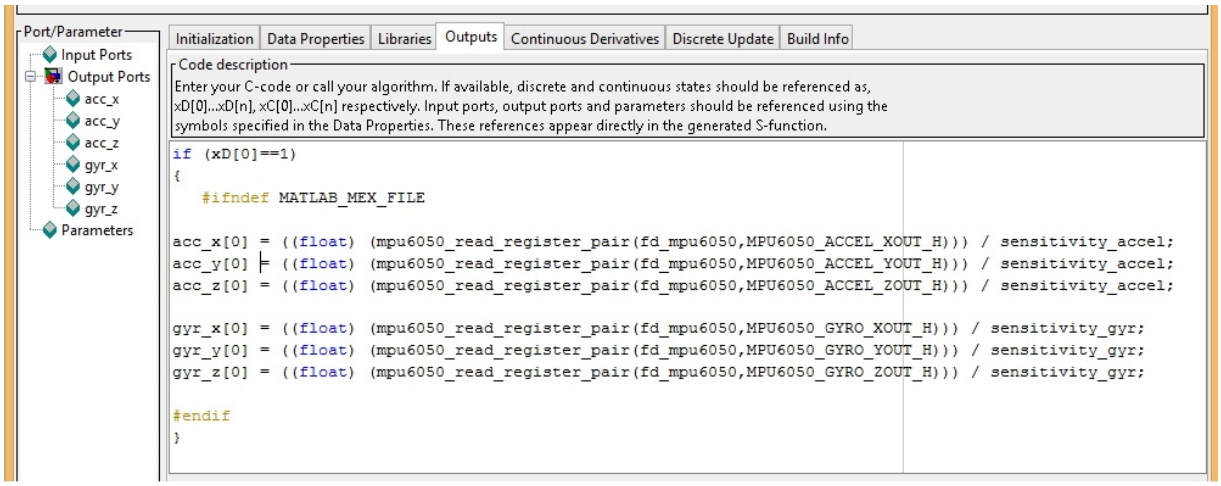

Figure 4.29: S-Function Builder Bloc: Outputs Tab.

The source code entered here has read/write access to the output variables configured via the Output Ports pane in the Data Properties tab.

All driver blocks developed for the purpose of external execution on the ZedBoard contain the following lines of code:

```
    /* Code only executed on The ZedBoard  */
#ifndef MATLAB_MEX_FILE
    /* Output Code */
#else
#endif
```

As previously mentioned, during the build process, a MATLAB executable (MEX) file is generated in order to execute the Simulink model. The above code ensures that code found within in the "ifndef" statement is only executed on the target hardware and not on the host system running Simulink. This conditional structure may also be used in the "Libraries" and "Discrete Update" tabs.

S-Function Discrete Update Tab:

The Discrete Update pane defines, in general, the evolution laws for the discrete state vector. Therefore data variables stored in the state vector propagate from one time-step to the next. This provides a fairly useful functionality for applications such as counters or state machines.

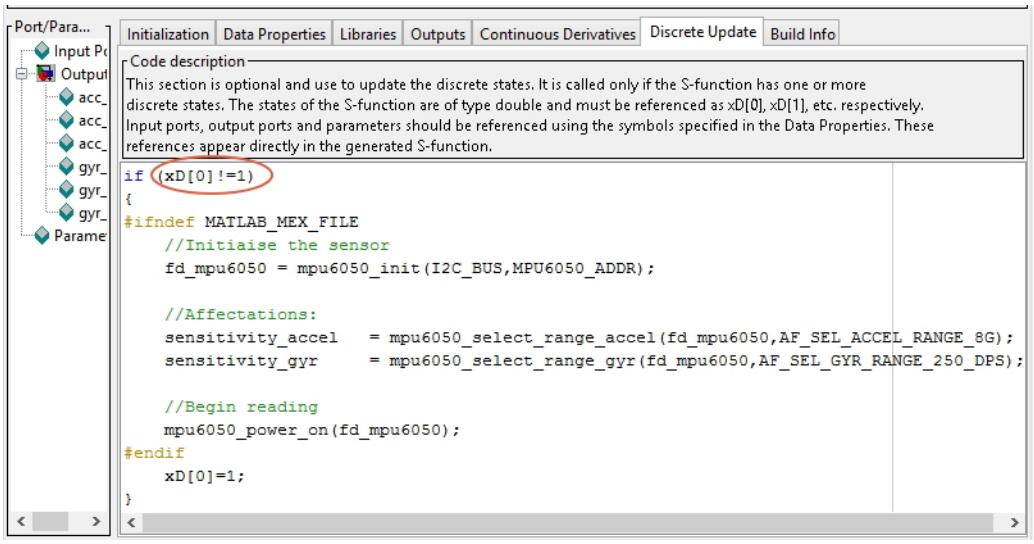

Figure 4.30: S-Function Discrete Update Tab.

As shown in the figure above, it is used here to run some initialization code, which we type directly in the edit field.

S–Function Build Info Tab:

Once the S-Function is fully configured it must be complied and placed into a wrapper file. This task, among others, is carried out in the build process and may be monitored from the Build Info tab's Compilation Diagnostics pane. The overall state of the compilation as well as any syntax or reference errors detected during compilation will be indicated here.

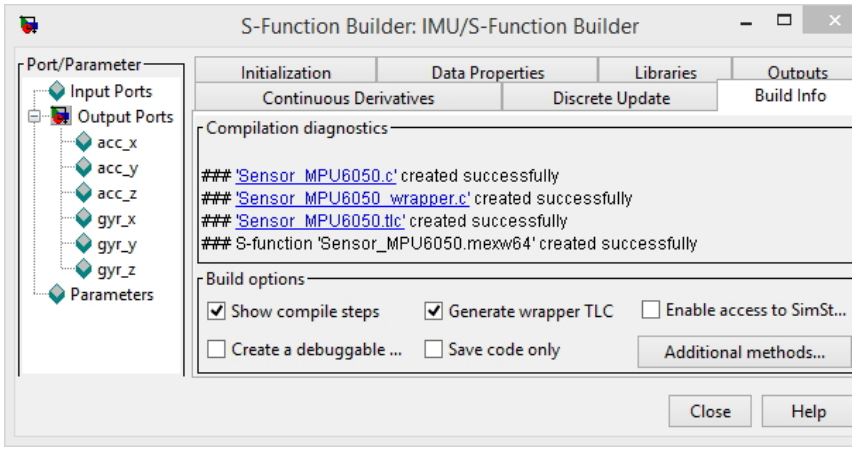

Figure 4.31: S–Function Build Info Tab.

The “Generate wrapper TLC” checkbox must be checked. This will generate the TLC-file which will then be used to build the executable that will run on the target. On the other hand, the “Enable access to SimStruct” check should be left unchecked (unless you really need it) because it might prevent the block from working on targets that do not support non-inlined S-functions.

4.7 Model Building and Testing

We have finalized the "Sensor_MPU6050" S-Function Builder driver block configuration and we can now press the Build button on the top of the S-Function Builder dialog box. **Six** files hence, are going to be generated:

- **Sensor_MPU6050.c:** Contains C source code for external declarations of the wrapped Outputs and Update functions found in the **Sensor_MPU6050_wrapper.c** file. Standard S-Function methods are included to validate and initialize the size and value of the outputs and discrete data variables.
- **Sensor_MPU6050.tlc:** Target Language Complier (TLC) file allows MATLAB’s Simulink Coder to include the S-Function block code in the generated C-Code for the Simulink Coder which is the primary engine for generating and executing C and C++ code from Simulink models. Essentially, the TLC file acts as an interface allowing the Simulink Coder to grab the S-Function block code and include it in the overall C code generated for the entire Simulink model. This overall C code will then be compiled to run on the target hardware.
- **Sensor_MPU6050.mex64:** MATLAB Executable file generated from the C code entered into the S-Function builder dialog box for simulation purposes.
- **Sensor_MPU6050_wrapper.c:** The code written into the Discrete Update, Outputs panes are wrapped into **individual functions** and placed - in addition to the libraries and global declared variables - into this wrapper file. These functions are referenced by the TLC file during simulation.
- **rtwmakecfg.m** and **SFB_Sensor_MPU6050_SFB.mat:** Two configuration files, the first inculdes Simulink Coder (RTW) configuration which is used for simulation in the target hardware (The ZedBoard), the second one contains absolute directory paths to design files.

The file content of **Sensor_MPU6050_wrapper.c** can be found at Appendix **??**

4.7.1 Model deployment

In addition to the S-Function Builder driver block, a Time Scope provides real time visualization of the sensor individual output signals. An overview of the design is shown at Fig 4.32

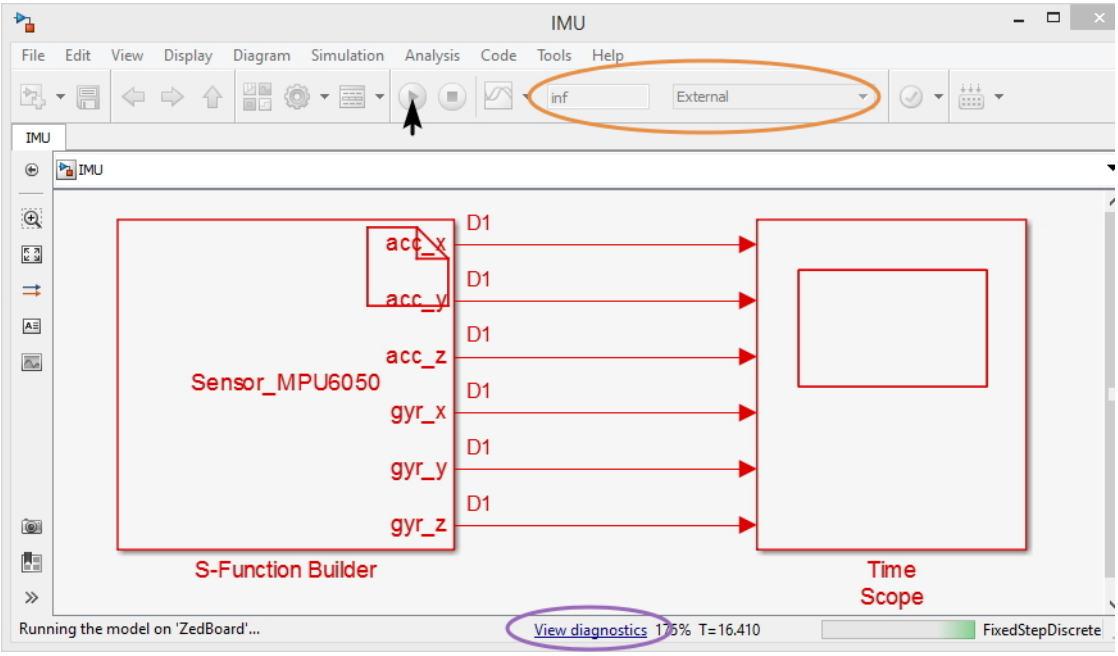

Figure 4.32: Deployed model and runnig simulation

Before starting deployment and simulation, we make sure that the simulation time is set to "inf" and the simulation mode is "External". Once the simulation started, we can find a diagnostics report accessible at the bottom of the window (Fig 4.35) and containing detailed information about code compilation, execution and simulation parameters. A second window (textual) opens as shown in Fig 4.33 with inforamtion that we put in the code for debugging purposes. With the given IP address in the report, we can establish an serial communication session with ZedBoard's Petalinux and verify the running process which has the same name as our Simulink model "IMU" and an PID given by the OS kernel of 944 (Fig 4.34).

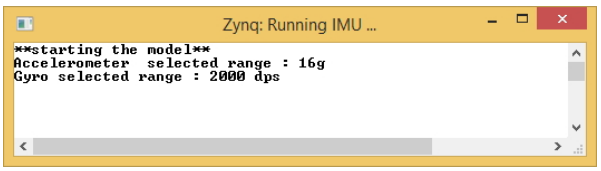

Figure 4.33: Textual printed window showing debugging information

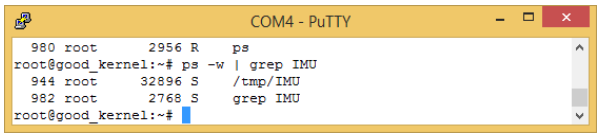

Figure 4.34: Information about deployed and running model on the ZedBorad's OS

We open the Time Scope block to observe the 6-axis output signals. Real time variations are seen while interacting with the IMU by applying accelerations and tilts around the three axis. A snapshot of the resulted simulation is shown in Fig 4.36.

We can check from the snapshot that the variations of the readings span the acctual selected ranges, $16g$ for the Accelerometer and $2000deg/s$ for the Gyroscope (Fig 4.33) the variation of the acceleration of the z axis everage at $1g$ for a horizontal position of the IMU as expected, as it gives as measurement the earth's gravitational field.

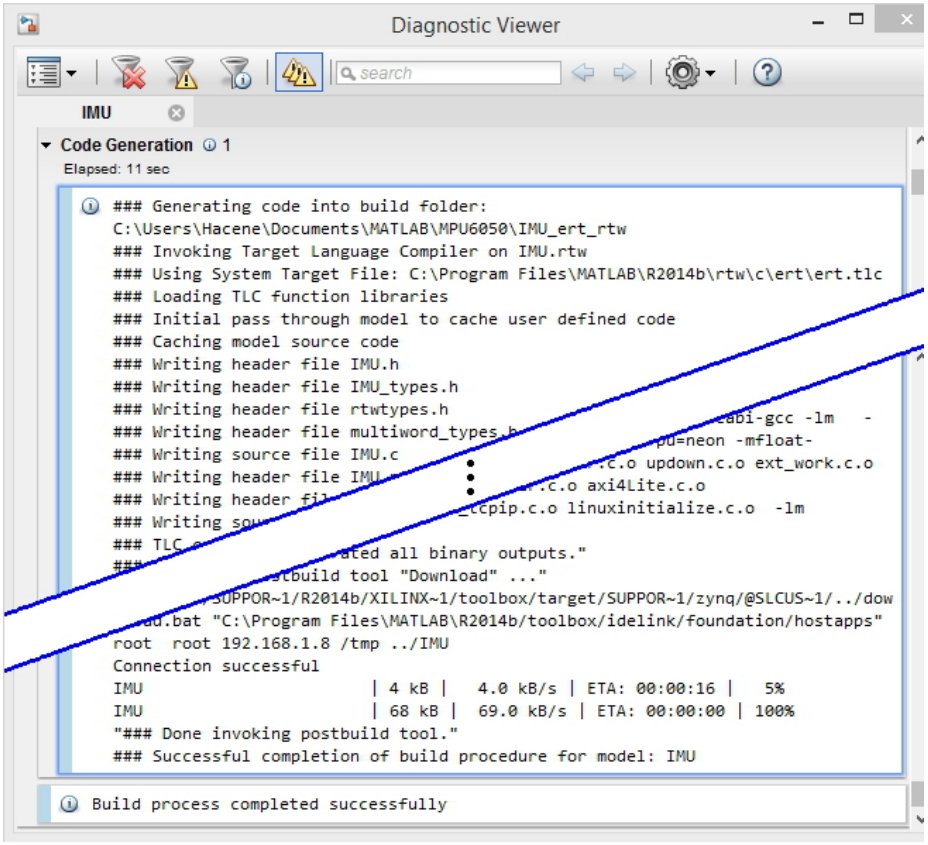

Figure 4.35: Building process report showing a successful compilation

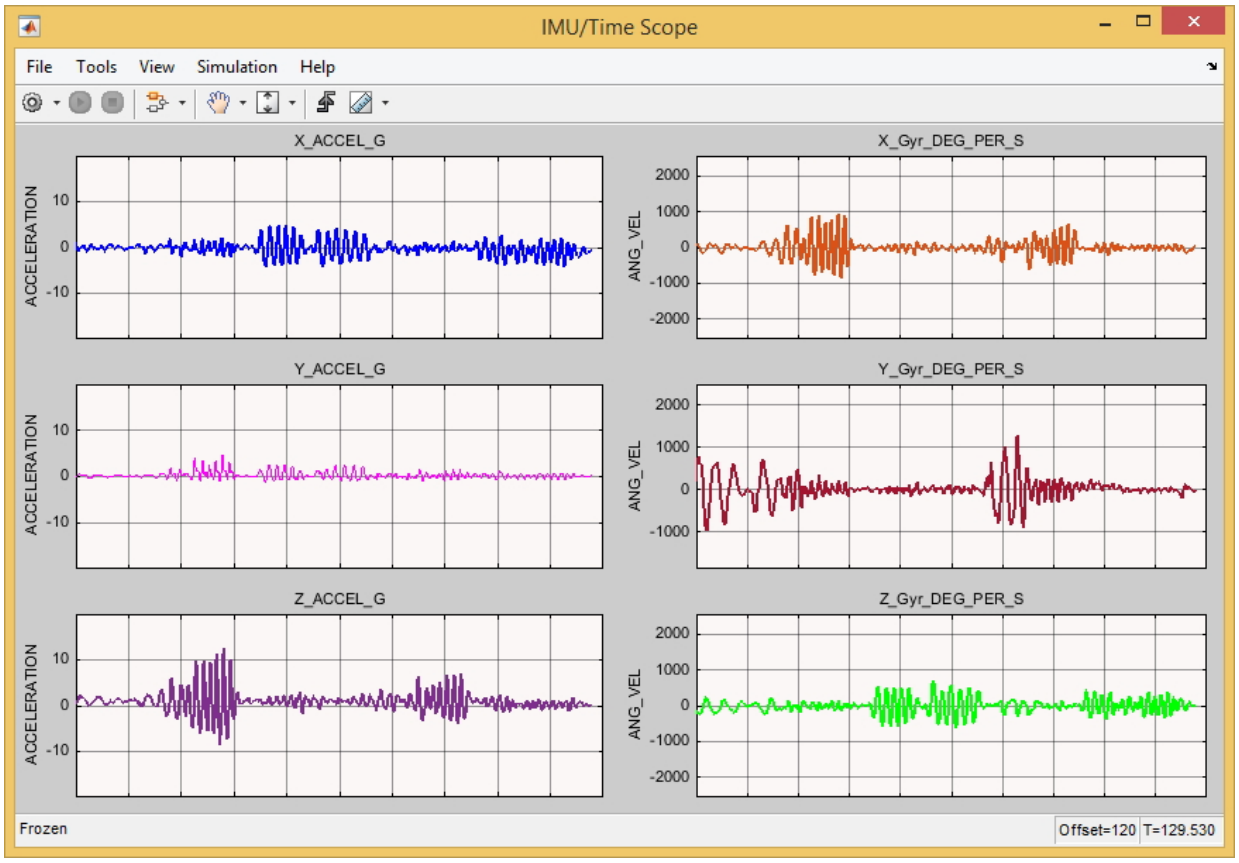

Figure 4.36: Signal Variations in the Time Scope window

Random errors from sensor readings can be observed while the sonsor is in rest by zooming on the amplitudes axis. To verify the presence of any bias, we collect measurements for a given time and compute the mean value for our readings.

For the gyroscope, the noise RMS given by the constructer in the datasheet is about $0.05^o/s - rms$, our measured noise rms is comparable to this range, thus we may assume that our readings from the gyroscope are reliable.

The accelerometer's errors on the other hand present a difficulty for bias evaluation, as it must be handled perfectly aligned while taking our measurements in order for the gravitational field vector to have only one component along the z-axis, if else, the other two axis will read also components of the gravitational filed misleading us to take them as bias.

Conclusion

In this chapter we have concluded the design and deployment of our application platform.

First, possible hardware system development choices were discussed before justifying the considered chosen method.

The Simulink environment configuration was established with the proper needed packages in order to take in consideration the Zedboard as an embedded hardware tareget.

In our situation, an Embedded Linux Operating System was prepared as it was needed for our design.

The main Simulink block is the S-Function Builder which was used to create the sensor driver block whose aim is to communicate with the IMU and acquire the axis data signals directly to the Simulink platform.

5

Example Application: Attitude Estimation Using an Orientation Filter

Attitude estimation involves a two-part process [31]:

1) estimation of a vehicle's orientation from body measurements and known reference observations.

2) filtering of noisy measurements.

The second part is achieved by combining the measurements with models, which in itself can be done a number of different ways.

The first step, where an attitude estimate is obtained from body measurements to feed a filter (or an observer), ends up in one of many known representations, e.g., Euler angles, quaternions, Euler angle-axis representation, rotation matrix, etc. For the filtering process there is also a very large number of alternatives, depending on the models and representations of the attitude. Kinematic models, which resort basically to three-axis rate gyros, are exact.

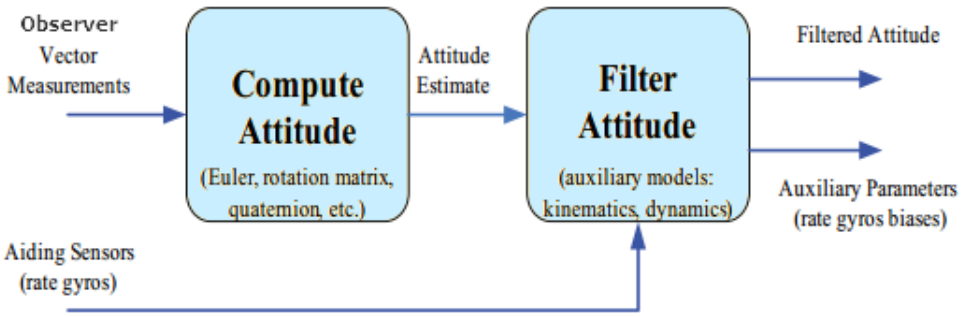

Figure 5.1: Classic attitude estimation

Fig 5.1 depicts a traditional attitude estimation solution. As it is possible to observe, vector measurements such as the gravitational and magnetic fields are first used to compute a representation of the attitude of the vehicle. Afterwards, the attitude filter evolves according to the its representation and resorting to kinematic or dynamic attitude models.

The signal output of low-cost IMU systems is characterized by low resolution signals subject to high noise levels as well as general time-varying bias terms [32]. Therefore, raw signals must be processed to reconstruct smoothed attitude estimates and bias-corrected angular velocity measurements through suitable sensor fusion algorithms. In fact, suitable exploitation of acceleration measurements can avoid drift caused by numerical integration of gyroscopic measurements.

However, the use of only these two source of information cannot correct the drift of the estimated heading, thus an additional sensor is needed, i.e., a tri-axial magnetometer, which allows to obtain a correct heading estimation.

consider a vehicle equipped with an Inertial Measurement Unit, which contains three triads of orthogonally mounted rate gyros, accelerometers, and magnetometers. The magnetometers provide the magnetic field in bodyfixed coordinates. This quantity is locally constant in inertial coordinates and it is therefore a feasible vector observation for attitude estimation.

On the other hand, for sufficiently low frequency bandwidths, the gravitational field also dominates the accelerometer measurements. This provides a second vector observation, which is, in general, not parallel to the first. Therefore, it is possible to determine the accurate attitude of the vehicle with an 9-axis IMU.

5.1 Gradient Descent Orientation Filter (GDOF)

Attitude estimation for the IMU and AHRS systems developed during the course of this project employs the use of a relatively new gradient descent orientation filter (GDOF) algorithm for sensor fusion. Initial research revealed a small but growing interest in this algorithm within the open source UAV autopilot community over the well know heavyweights such as the Kalman and Extended Kalman filter approaches. Reasons for this increased interest may be distilled into three main factors[33].

The first, and most prevalent, appears to be the low computational load required in the implementation of this orientation filter [32]. This fact proves especially attractive for embedded systems where microcontroller and microprocessor power, although increasingly enhanced, are still behind what may be found in a conventional computer. Lower required computational power translates into lighter packages and smaller footprints opening the integration possibilities to small UAVs and even wearable technology.

In line with the first attractive factor, the second stems from the ability to obtain higher estimation accuracy at lower sampling rates [34]. This is due to the fact an actual analytical solution has been derived for the descent gradient algorithm as opposed to the linear regression iterations required by the Kalman filter approach.

The final pillar supporting the popularity of the approach is the employment of quaternions in order to avoid “Gimbal Lock” type singularities that are prevalent with Euler angle attitude representation.

5.2 Quaternion Representation

The quaternion representation of attitude orientation in three dimensional space may be given by a four dimensional complex number. The conceptual motivation to this approach is to represent any arbitrary rotation of a frame relative to frame as a rotation of angle around a specific axis defined in the frame. A graphical representation is presented in Fig 5.2

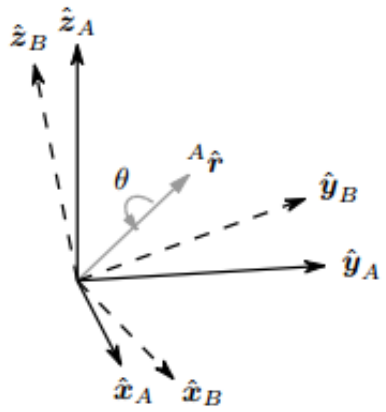

Figure 5.2: The orientation of frame B is achieved by a rotation, from alignment with frame A , of angle θ around the axis ${}^A\hat{r}$ [33]

The quaternion representation of the above frame rotation may be given as:

$$ {}^A_B\hat{q} = [q_1 \quad q_2 \quad q_3 \quad q_4] = [\cos\tfrac{\theta}{2} \quad -r_x\cos\tfrac{\theta}{2} \quad -r_y\cos\tfrac{\theta}{2} \quad -r_z\cos\tfrac{\theta}{2}] \tag{5.1} $$

The quaternion conjugate representing the inverse relationship, that is to say, the orientation of frame with respect to frame, may be given as:

$$ {}^A_B\hat{q}^* = \quad {}^B_A\hat{q} = [q_1 \quad -q_2 \quad -q_3 \quad -q_4] \tag{5.2} $$

For compound orientations representing two independent rotations such as ${}^A_B\hat{q}$ and ${}^B_C\hat{q}$ the rotation ${}^A_C\hat{q}$ may be calculated via the quaternion product denoted by $\otimes$ and represented as:

$$ {}^A_C\hat{q} = \quad {}^B_C\hat{q} \otimes {}^A_B\hat{q} \tag{5.3} $$

The mechanics of the quaternion gives:

$$ a \otimes b \neq b \otimes a \tag{5.4} $$

Rotation of a three dimensional vector -, via a quaternion may be calculated as:

$$ {}^B v = \quad {}^A_B\hat{q} \otimes {}^A v \otimes {}^A_B\hat{q}^* \tag{5.5} $$

Here, ${}^A v$ and ${}^B v$ are representations of the same vector as described from frames and respectively. The vector representation may be cast into a four element row vector as shown below:

$$v = [0 \quad v_x \quad v_y \quad v_z] \tag{5.6}$$

A quaternion rotation such as ${}^{A}_{B}\hat{q}$ may be cast as a rotational matrix ${}^{A}_{B}R$ where:

$${}^{A}_{B}R = \begin{bmatrix} q_1^2 - 1 + 2q_2^2 & 2(q_2q_3 + q_1q_4) & 2(q_2q_4 - q_1q_3) \\ 2(q_2q_3 - q_1q_4) & 2q_1^2 - 1 + 2q_1^2 & 2(q_3q_4 - q_1q_2) \\ 2(q_2q_4 + q_1q_3) & 2(q_3q_4 - q_1q_2) & 2q_1^2 - 1 + 2q_4^2 \end{bmatrix} \tag{5.7}$$

The Euler angle casting of the quaternion rotation ${}^{A}_{B}\hat{q}$ may be given by the following equations:

$$\psi = atan2(2q_2q_3 - 2q_1q_4 \ , \ 2q_1^2 + 2q_2^2 - 1) \tag{5.8}$$

$$\theta = -\sin^{-1}(2q_2q_4 + 2q_1q_3) \tag{5.9}$$

$$\phi = atan2(2q_3q_4 - 2q_1q_2 \ , \ 2q_1^2 + 2q_4^2 - 1) \tag{5.10}$$

5.3 GDOF derivation

Figure 5.3 shows a block diagram representation of the complete orientation filter implementation for an IMU.

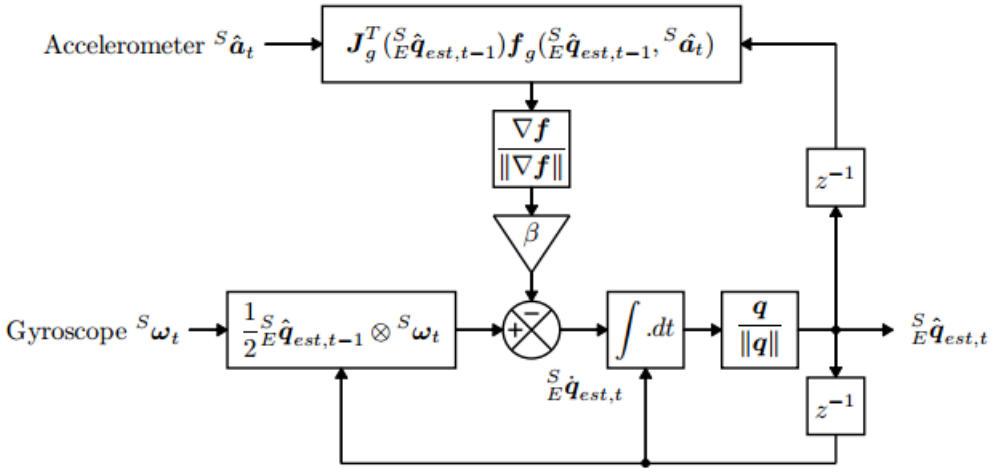

Figure 5.3: Block diagram representation of the complete orientation filter for an IMU implementation [34].

5.3.1 Extraction of Orientation from Gyroscope Angular Rates

The angular velocity provided by the gyroscope sensor may be represented in the following vector format:

$${}^{s}\omega = \begin{bmatrix} 0 & \omega_x & \omega_y & \omega_z \end{bmatrix} \tag{5.11}$$

From the above representation of the angular rates, the quaternion representation of the rate of change of the earth frame relative to the sensor frame may be calculated as:

$$ {}^{S}_{E}\dot{q} = \frac{1}{2} {}^{S}_{E}\hat{q} \otimes {}^{S}\omega \tag{5.12} $$

Hence the orientation of the earth frame relative to the sensor frame at a given point in time may be computed via the integration of equation 5.12 in discrete time. The discrete time representation of equation 5.12 may be expressed as:

$$ {}^{S}_{E}\dot{q}_{w,t} = \frac{1}{2} {}^{S}_{E}\hat{q}_{est,t-1} \otimes {}^{S}\omega_{t} \tag{5.13} $$

The numerical integration of the above equation yields:

$$ {}^{S}_{E}q_{w,t} = \quad {}^{S}_{E}\hat{q}_{est,t-1} + {}^{S}_{E}\dot{q}_{w,t}\,\Delta t \tag{5.14} $$

Equation 5.14 now represents, in quaternions, the orientation of the sensor frame with respect to the earth frame as deduced by integration of the gyroscopic angular rates.

5.3.2 Extraction of Orientation from Field Vector Observations

Before developing explicit solutions for orientation based on accelerometer or magnetometer readings, the general case of determining the rotation of the sensor frame with respect the earth frame given the sensor's field measurements is explored. Conceptually, it may be stated that if the direction of the earth field in the earth frame is known (e.g. gravity, magnetic north), measurement of the field's direction with respect to the sensor frame can be manipulated to determine the orientation of sensor frame with respect to the earth frame. Unfortunately, this approach will not generate a unique solution as infinite orientations are possible when the sensor's axis is perfectly parallel with the measured earth field. In such a case, the sensor's exact orientation round the axis in-line with the earth field cannot be determined. To rectify this, the orientation problem is cast as an optimization problem. The goal is then to determine the orientation of the sensor frame relative to the earth frame, ${}^{S}_{E}\hat{q}$ such that it aligns a predefined reference direction in the earth field, ${}^{E}\hat{d}$ with the measured field components in the sensor frame ${}^{S}\hat{s}$.

Therefore, the optimization problem to derive ${}^{S}_{E}\hat{q}$ may be represented as:

$$ \min_{{}^{S}_{E}\hat{q}\in\Re^4} f\left({}^{S}_{E}\hat{q},\quad {}^{E}\hat{d},\quad {}^{S}\hat{s}\right) \tag{5.15} $$

Where the objective function is defined as:

$$ f\left({}^{S}_{E}\hat{q},\quad {}^{E}\hat{d},\quad {}^{S}\hat{s}\right) = {}^{S}_{E}\hat{q}^{*} \otimes {}^{E}\hat{d} \otimes {}^{S}_{E}\hat{q} - {}^{S}\hat{s} \tag{5.16} $$

The vector representation of the sensor frame orientation, reference direction and sensor measurements are defined respectively as:

$$ {}^{S}_{E}\hat{q} = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix} \tag{5.17} $$

$$ {}^{E}\hat{d} = \begin{bmatrix} 0 & d_x & d_y & d_z \end{bmatrix} \tag{5.18} $$

$$ {}^{S}\hat{s} = \begin{bmatrix} 0 & s_x & s_y & s_z \end{bmatrix} \tag{5.19} $$

As the name of the proposed filter would suggest, the gradient descent algorithm is chosen as the optimization algorithm of choice given its simplicity, ease of implementation and low computational cost. Therefore, the estimation of the sensor orientation with respect to the earth frame after R iterations with step-size μ and an initial orientation guess of ${}^{S}_{E}\hat{q}_0$ is calculated as follows:

$$ {}^{S}_{E}\hat{q}_{k+1} = {}^{S}_{E}\hat{q}_k - \mu \frac{\nabla f\left({}^{S}_{E}\hat{q},\ {}^{E}\hat{d},\ {}^{S}\hat{s}\right)}{\|\nabla f\left({}^{S}_{E}\hat{q},\ {}^{E}\hat{d},\ {}^{S}\hat{s}\right)\|}, \quad k = 0, 1, 2...n \tag{5.20} $$

The gradient of the solution surface may be computed as:

$$ \nabla f\left({}^{S}_{E}\hat{q},\ {}^{E}\hat{d},\ {}^{S}\hat{s}\right) = J^T\left({}^{S}_{E}\hat{q}_k,\ {}^{E}\hat{d}\right) f\left({}^{S}_{E}\hat{q}_k,\ {}^{E}\hat{d},\ {}^{S}\hat{s}\right) \tag{5.21} $$

The vector representation of the objective function and its corresponding Jacobian may be expressed respectively as:

$$ f\left({}^{S}_{E}\hat{q}_k,\ {}^{E}\hat{d},\ {}^{S}\hat{s}\right) = $$

$$ \begin{bmatrix} 2d_x(\frac{1}{2} - q_3^2 - q_4^2) + 2d_y(q_1q_4 + q_2q_3) + 2d_z(q_2q_4 - q_1q_3) - s_x \\ 2d_x(q_2q_3 - q_1q_4) + 2d_y(\frac{1}{2} - q_2^2 - q_4^2) + 2d_z(q_1q_2 - q_3q_4) - s_y \\ 2d_x(q_1q_3 - q_2q_4) + 2d_y(q_3q_4 - q_1q_2) + 2d_z(\frac{1}{2} - q_2^2 - q_3^2) - s_z \end{bmatrix} \tag{5.22} $$

$$ J^T\left({}^{S}_{E}\hat{q}_k,\ {}^{E}\hat{d}\right) = $$

$$ \begin{bmatrix} 2d_yq_4 - 2d_zq_3 & 2d_yq_3 + 2d_zq_4 & -4d_xq_3 + 2d_yq_2 - 2d_zq_1 & -4d_xq_4 + 2d_yq_1 + 2d_zq_2 \\ -2d_xq_4 + 2d_zq_2 & 2d_xq_3 - 4d_yq_2 + 2d_zq_1 & 2d_xq_2 - 2d_zq_4 & -2d_xq_1 - d_yq_4 + 2d_zq_3 \\ 2d_xq_3 - 2d_yq_2 & 2d_xq_4 - 4d_yq_1 + 2d_zq_2 & 2d_xq_1 + 2d_yq_4 - 4d_zq_3 & 2d_xq_2 - 2d_yq_3 \end{bmatrix} \tag{5.23} $$

Hence, equations (5.20 – 5.23) define the general solution for determining the earth orientation with respect to the sensor frame given a predefined known reference direction in the earth field.

5.3.3 Extraction of Orientation from Accelerometer Gravity Vector Observations

The general equations (5.20 – 5.23) for determining the sensor frame orientation relative to the earth frame may now be applied to the case of the gravity field sensed by the accelerometer. Assuming (quite confidently) that the orientation of the gravity vector relative to the earth frame is known and defines the earth frame z-axis, then the reference direction for the normalized earth filed is given as:

$$ {}^{E}\hat{d} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.24} $$

The normalized accelerometer measurements are represented as:

$$ {}^{S}\hat{a} = \begin{bmatrix} 0 & a_x & a_y & a_z \end{bmatrix} \tag{5.25} $$

Substituting the above two equations in equations (5.18 & 5.19) yields the following versions of the objective function and its Jacobian yield:

$$f_g({}^S_E\hat{q},\ {}^S\hat{a}) = \begin{bmatrix} 2d_z(q_2q_4 - q_1q_3) - a_x \\ 2d_z(q_1q_2 - 4q_3q_4) - a_y \\ 2d_z(\frac{1}{2} - q_2^2 - q_3^2) - a_z \end{bmatrix} \tag{5.26}$$

$$J_g({}^S_E\hat{q}) = \begin{bmatrix} -2q_3 & 2q_4 & -2q_1 & 2q_2 \\ 2q_2 & 2q_1 & 2q_4 & 2q_3 \\ 0 & -4q_2 & -4q_3 & 0 \end{bmatrix} \tag{5.27}$$

5.3.4 Extraction of Orientation Solution

In order to determine the solution point of the objective function, the expression for the iterative calculation of the estimated orientation based on the previous time-step estimate, is given as:

$${}^S_E\hat{q}_{\nabla,t} = {}^S_E\hat{q}_{est,t-1} - \mu_t \frac{\nabla f}{\|\nabla f\|} \tag{5.28}$$

Where:

$$\nabla f = J_g^T({}^S_E\hat{q}_{est,t-1}) f_g({}^S_E\hat{q}_{est,t-1},\ {}^S\hat{a}) \tag{5.29}$$

Note that the step-size for the gradient decent iteration may be given by:

$$\mu_t = \alpha \|{}^S_E\dot{q}_{\omega,t}\| \Delta t, \quad \alpha > 1 \tag{5.30}$$

Where Δt is the sampling period, ${}^S_E\dot{q}_{\omega,t}$ is the orientation rate sensed by the gyroscope and α is a constant aimed at counteracting the effects of noise on the accelerometer.

5.3.5 Filter Fusion Algorithm

The sensory data extracted from the gyroscope and accelerometer must be combined to obtain a single orientation estimate of the sensor frame relative to the earth frame. This is carried out by the fusion of the orientation estimated by the numerical integration of the gyroscope's angular rate readings and the orientation estimated by the gradient descent solution of the objective function stated in section 5.3.2 and 5.3.4 respectively. This calculation is carried out by the weighted linear combination of the two estimates mentioned above and may be expressed as:

$${}^S_E q_{est,t} = \gamma_t\, {}^S_E q_{\nabla,t} + (1 - \gamma_t) {}^S_E q_{\omega,t} \quad , 0 \leq \gamma_t \leq 1 \tag{5.31}$$

For optimal fusion, the weighting factor may be chosen as:

$$\gamma_t = \frac{\beta}{\frac{\mu}{\Delta t} + \beta} \tag{5.32}$$

Where $\frac{\mu}{\Delta t}$ is a representation of the rate of convergence of ${}^S_E\hat{q}_\nabla$ and β represents the divergence rate of ${}^S_E q_\omega$.

The objective of the fusion algorithm is to improve the orientation estimate by using the angular rate integral $({}^S_E q_\omega)$ provided by the gyroscope to filter out high frequency errors present in the gradient decent $({}^S_E\hat{q}_{\Delta,t})$ estimate. The gradient descent estimate is used to correct the accumulated error from the integral and to provide convergence from initial conditions.

An approximation has been given when a large value of μ_t used in equation 5.28, it means that ${}^{S}_{E}\hat{q}_{est,t-1}$ becomes negligible and the equation can be re-written as :

$$ {}^{S}_{E}q_{\nabla,t} \approx -\mu_t \frac{\nabla f}{\|\nabla f\|} \tag{5.33} $$

The definition of in equation 5.31 simplifies if the β term in the denominator becomes negligible and the equation can be rewritten as below. It is possible to also assume that $\gamma_t \approx 0$.

$$ \gamma_t \approx \frac{\beta \Delta t}{\mu_t} \tag{5.34} $$

Further simplifications yield:

$$ {}^{S}_{E}\hat{q}_{est,t} = {}^{S}_{E}\hat{q}_{est,t-1} + {}^{S}_{E}\dot{q}_{est,t}\,\Delta t \tag{5.35} $$

$$ {}^{S}_{E}\dot{q}_{est,t} = {}^{S}_{E}\dot{q}_{\omega,t} - \beta \frac{\nabla f}{\|\nabla f\|} \tag{5.36} $$

It can be seen from the last two equations that the algorithm calculates the orientation ${}^{S}_{E}q_{est}$ by numerically integrating the estimated rate of change of orientation ${}^{S}_{E}\dot{q}_{est}$. The algorithm computes ${}^{S}_{E}\dot{q}_{est}$ as the rate of change of orientation measured by the gyroscopes, ${}^{S}_{E}\dot{q}_{\omega}$, with the magnitude of the gyroscope measurement error, β, removed in a direction based on accelerometer measurements.

5.3.6 Algorithm adjustable parameter

The orientation estimation algorithm requires one adjustable parameter, β, representing the gyroscope measurement error expressed as the magnitude of a quaternion derivative.

β is defined by the equation below where $\hat{q}$ is any unit quaternion.

$$ \beta = \|\frac{1}{2}\hat{q} \otimes \begin{bmatrix}\tilde{\omega}_{max} & \tilde{\omega}_{max} & \tilde{\omega}_{max}\end{bmatrix}\| = \sqrt{\frac{3}{4}}\tilde{\omega}_{max} \tag{5.37} $$

5.4 Filter performance

The previously discussed steps of GDOF derivation were conducted for the case where we had an 6D IMU (3-axis accelerometer and 3 -axis gyro). The author of the filter has also issued a version for 9D MARG (including a magnetometer). The MARG implementation incorporates magnetic distortion and gyroscope bias drift compensation.

The author in his report has also conducted few tests to evaluate the performance of the filter, this gives us an idea of its limits and offshoots of its application.

Static and dynamic performance tests showed better results than the Kalman filter for two axis of rotation (ϕ,θ). Values of RMS[ψ] were not computed for the IMU implementation of the proposed filter as the filter cannot, nor is intended to, compensate for an

accumulating error in this parameter.

The Filter adjustable parameter β was quantified as the mean of the corresponding RMS values of ϕ and θ. An optimal value of $\beta = 0.034$ for the IMU filter implementation was calculated from given RMSE of the gyro ($0.05^o/s$).

The results of the author's investigation into the effect of sampling rate on filter performance showed that the filter performs at roughly the same manner starting from around 50 Hz for a dynamic error less than 7^o, stating that this level of accuracy could be sufficient for applications such as human motion capture, compared to 512 Hz with Kalman filter.

5.5 Filter Implementation on Simulink

In this section we implement Madgwick's Gradient Decent Orientation Filter (GDOF) algorithm. The code was obtained from the author's published paper named "Estimation of IMU and MARG orientation using gradient descent algorithm" in 2011 and is written in C language.

Our goal is to add Simulink blocks to our previous model which wrap the algorithm code and feed it with real time signal data provided from the "MPU6050" driver block and sink the filtered signals to a 3D model Simulink block.

Following again the same steps explained in Section 4.6, we set the algorithm to be functional by using the S-Function Builder (Madgwick_IMU Block in Fig 5.4). After analyzing the C code's execution flow sequences, we can divide it into two separate states:

The first state is the initialization parameters like the initial state quaternion vector, this part is put in the S-Function's Discrete Update tab as it'll be executed once at startup.

The second part is a loop computation of quaternion outputs, it is put in the S-Function's Output tab. With this manner we manage our S-Function Builder block code insertion.

The sample mode will be set to "Inherited", thus the sampling rate for the Madgwick_IMU block will be dictated from the Driving block and will be updated at each arrival of a new Accelerometer and Gyro signal data. In the S-Function "Libraries" tab we fill the "includes" field with the <math.h> library needed for arithmetic operations and the global variable used in the algorithm.

The algorithm uses Gyro inputs in radian rather than degrees, so a conversion block was added for this purpose. It should be noticed that the filter block outputs signals in quaternion forms.

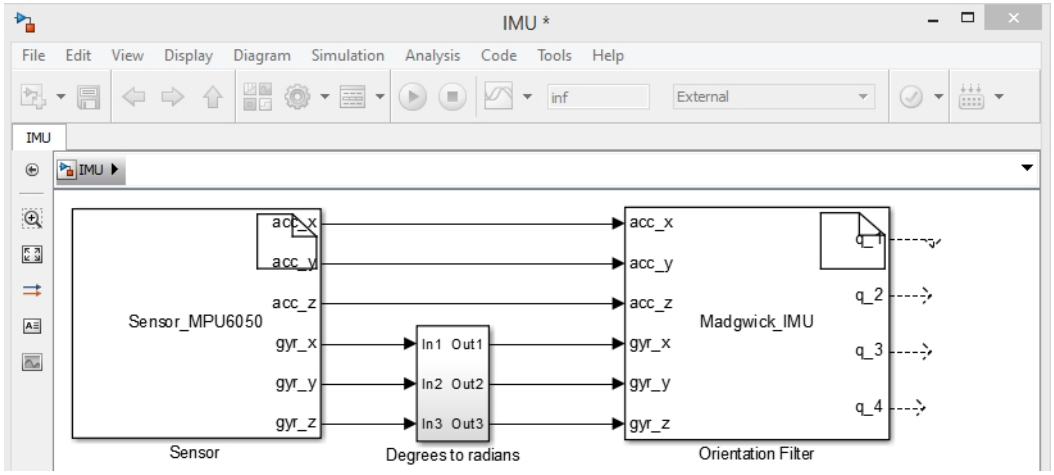

Figure 5.4: implementation of Madgwick Orientation Filter using S-function Builder.

The next step is the visualization of the output of the filter and real time simulation to evaluate the behavior of the filter. The VR Sink block in Simulink is a powerful example of 3D model simulation. It can be found under "Simulink 3D Animation" category in the Simulink Library Browser.

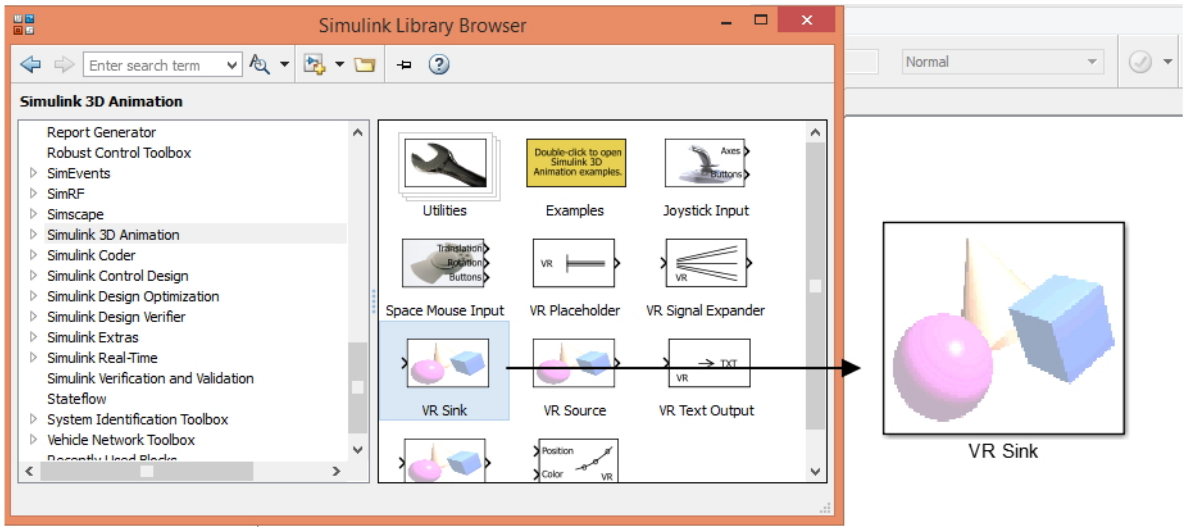

Figure 5.5: VR Sink Simulink bloc

By double clicking on the VR Sink block and again on "New", the 3D World Editor will launch allowing us to design our model. We mainly have to add a "Transform" object as its orientation will be controlled from the input signals. Note that the transform object must be given a name, or else its attributes can't be controlled (in our case the orientation). An example of a box textured with a picture of the MPU6050 sensor is given below:

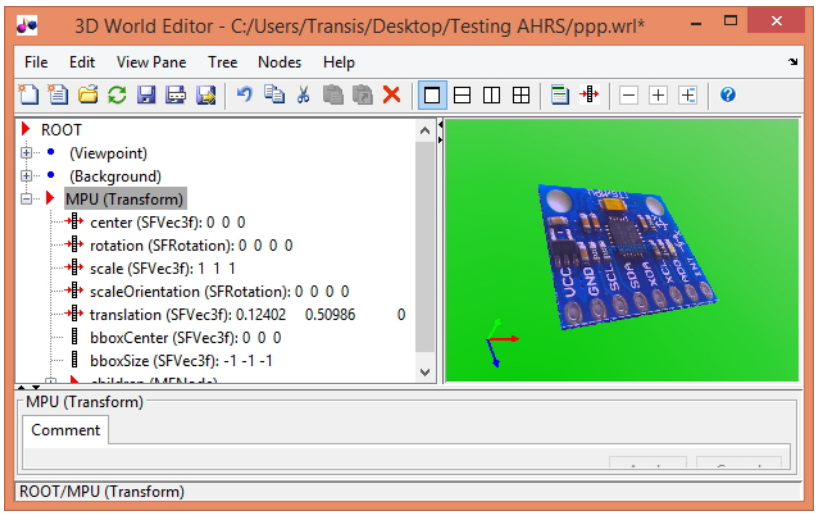

Figure 5.6: Creation of Visualization Bloc VR Sink

After we finished the design of our model we save it under a WRL file format. We open the VR Sink block again and choose the WRL to be our model, we check the "rotation" box to be controlled by input signals and validate.

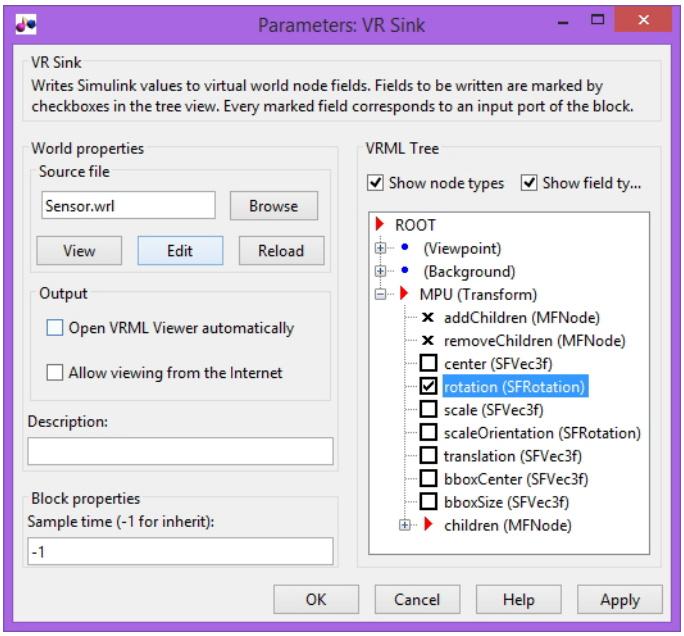

Figure 5.7: The VR Sink settings

Since the VR Sink recognizes only rotation vectors in the Axis-Angle form, we must add a transformation block of the quaternion representation to the Axis-Angle representation. The coded transformation function will be simply added to the Output tab of the S-Function block.

The entire model includes the driver block, the Orientation filter block, the Quaternion Management (transformation) block and 3D simulation block as shown below. The file content of **Madgwick_IMU_wrapper.c** can be found at Appendix **??**

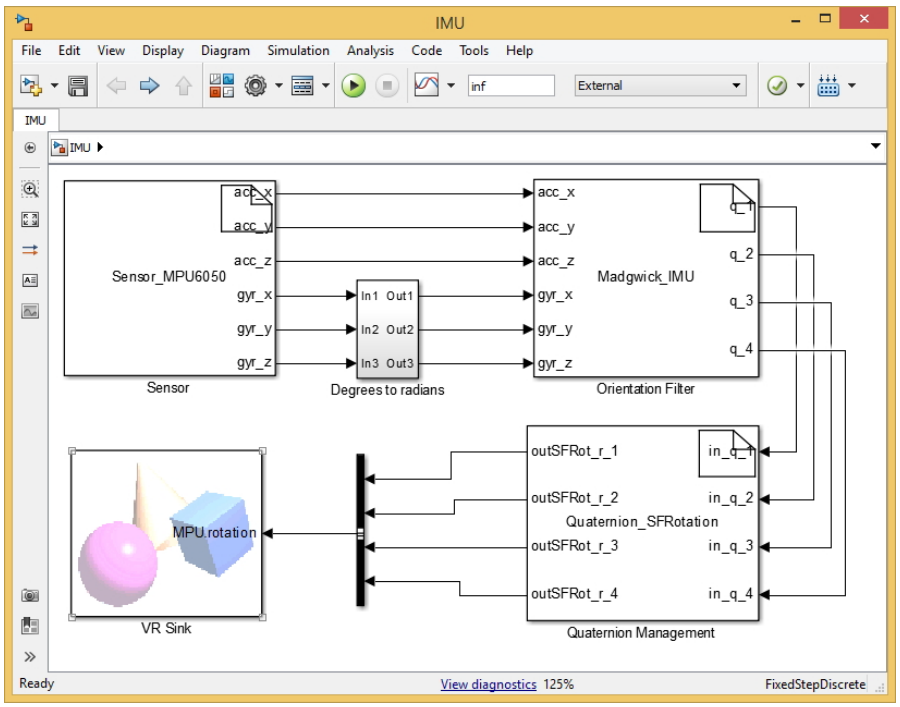

Figure 5.8: The final configuration of the model

The picture below shows real time execution of the model deployed on our embedded system and real time animations in the background while ineracting with the sensor:

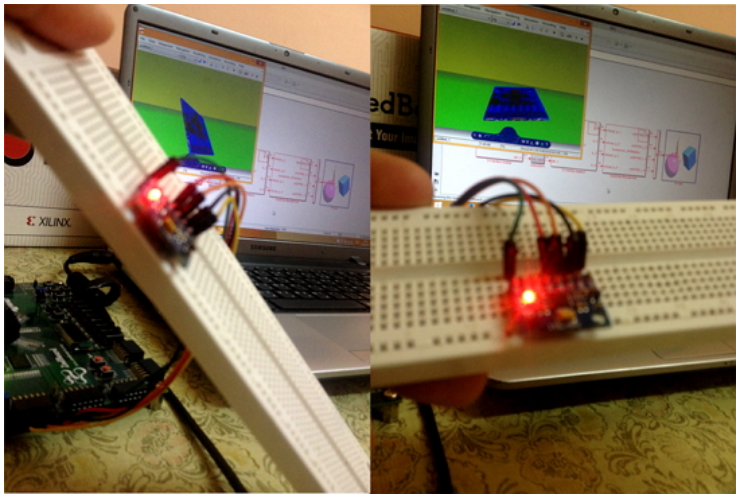

Figure 5.9: Real time animation while ineracting with the sensor

We have noticed the existence of yaw drift when the sensor is held still, it is an expected problem because the filter can not correct drift in the orientation.

Flexibility of the platform

Mahony et al [35] are the authors of the Complementary Filter (CF) with its offshoot: Extended CF, Non Linear CF..etc, the platform can help us evaluate the filter performance by simply adding the code to a new S-Function block and replacing the first one or evaluating them at the same time

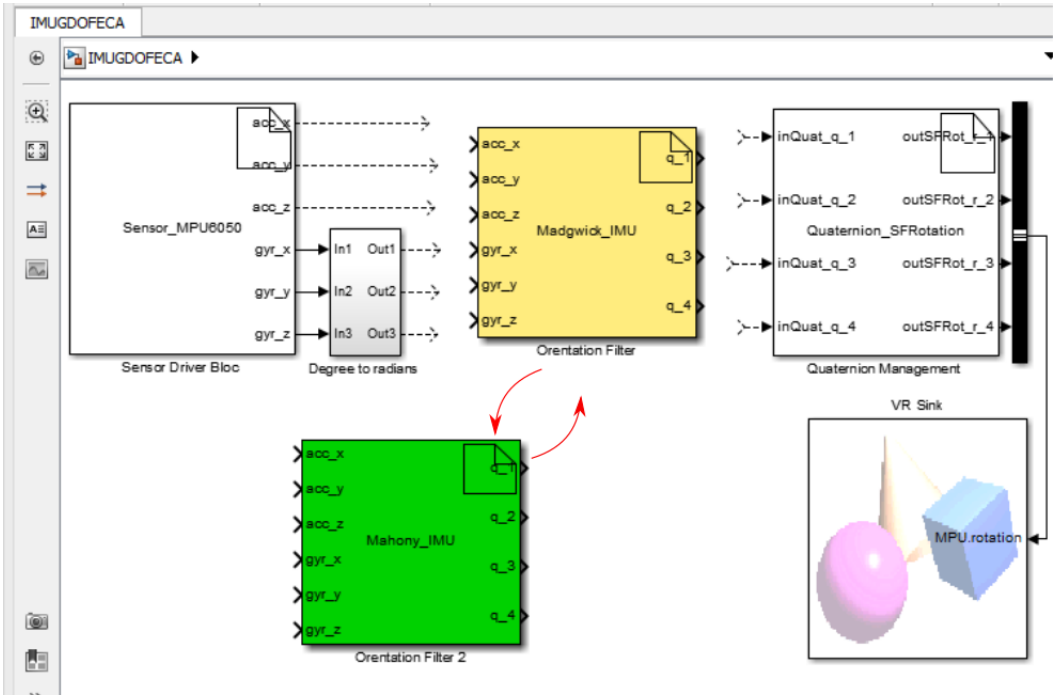

Figure 5.10: Verious possible applications of the platfform

5.6 Code execution Profiling

As well-known profiling is the computation of the code executed complexity and the frequency and duration of function calls; in general the behavior of the program. In addition to simulation and code deployment, Simulink gives another option wish is the code execution profiling.In our case we will perform a PIL (Processor In the Loop) profiling to verify the performance of the orientation filter (Madgwik_IMU) and the quaternion management Simulink bloc within.

To do so, we should go first to "Model Configuration Parameters" on the model window toolbar, select "Code Generation" and then "Verification". We check the "Measure task execution time" checkbox, this option enables to profile execution time for each sample iteration of the model. The "executionProfile" workspace variable holds the profiling data after the PIL simulation ends.

Once the above configuration applied, executing the following code in MATLAB allow us to obtain the profiling report after the model is just ran.

```
»report( executionProfile, 'Units', 'Seconds', 'ScaleFactor', '1e-03', 'NumericFormat', '%0.3f' )
```

Analysis of the report gives data about the different turnaround and execution times for each task. The resulted report is shown below:

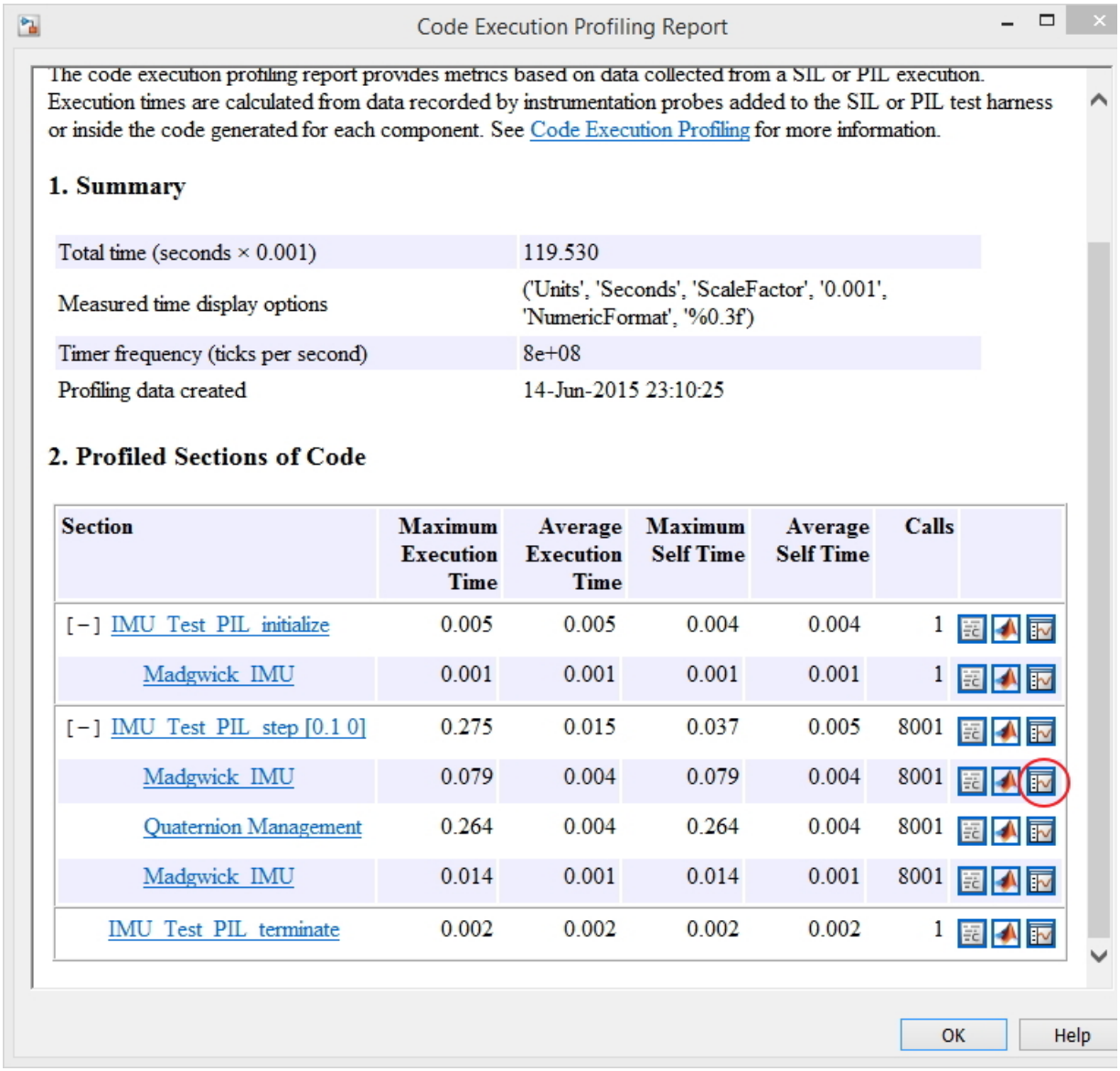

Figure 5.11: The generated PIL profiling report.

To visualize the execution of the code instruction we click in the Simulation data inspector icon (shown in the right of Fig 5.11).

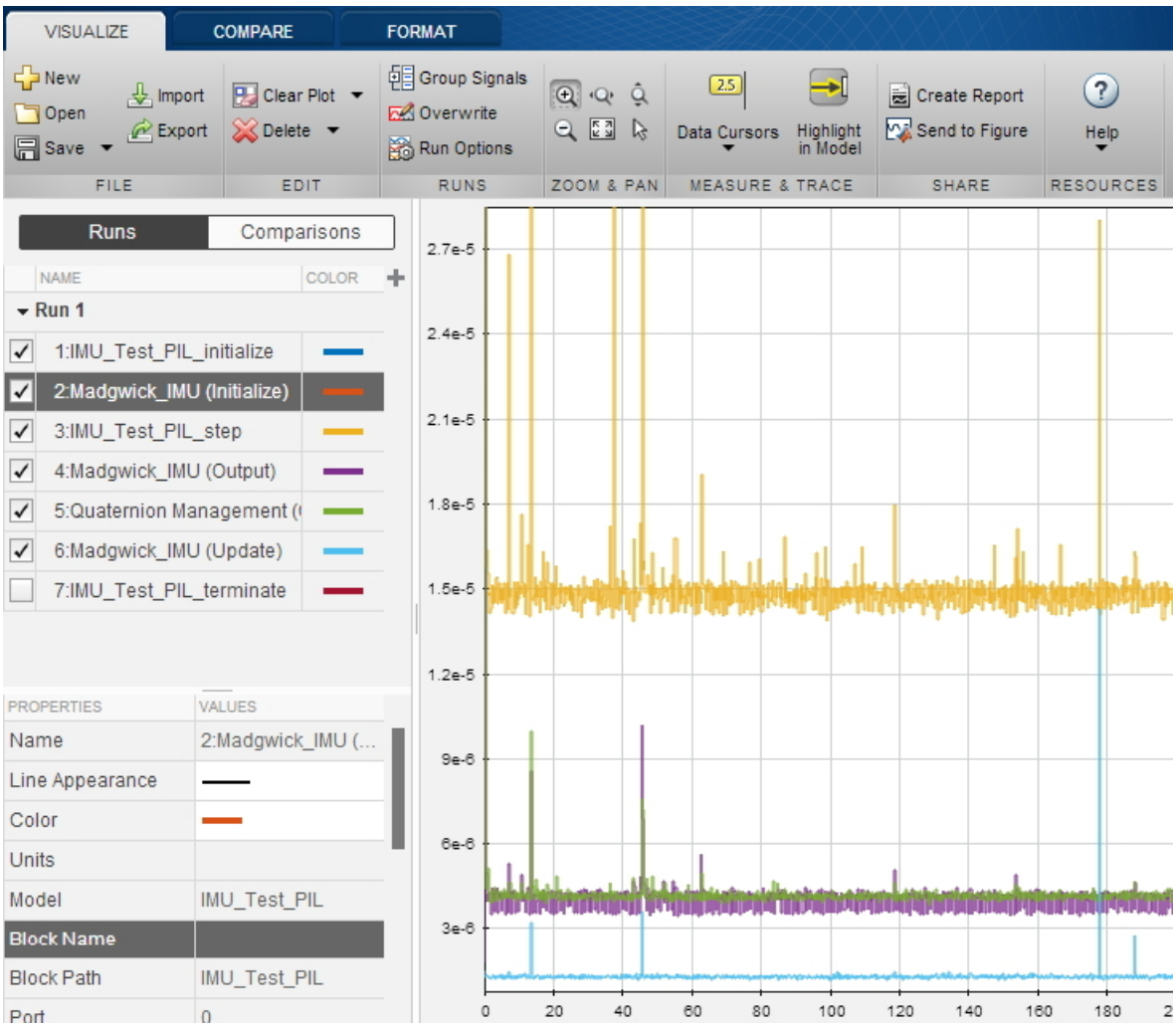

Figure 5.12: The generated PIL profiling histogram.

A window will open as shown in Fig 5.12. In this histogram we have on the horizontal axis the total execution time and in the vertical axis the time that each function of the algorithm Simulink blocs takes, like parameters initialization, algorithms update and output computation at each iteration..etc. note that the time is in ms.

Conclusion

In this final chapter we have tested an AHRS orientation filter algorithm on our new developed platform.

We have briefly given at first, the mathematical derivation of the filter and discussed about its performance, then we added its corresponding coded blocks to the platform.

After successfully running the model, we showed how to perform PIL profiling of the model in order to evaluate the execution time.

General Conclusion

Through the presented work, we have seen and experienced two main fields of applications: Model-Based Design for embedded hardware and inertial navigation systems.

We discovered in Simulink a set of powerful tools that offered dynamic and flexible functionalities which allowed us to evaluate designing through building models and real time simulations.

Despite the use of few types of Simulink blocks in our project, we have testified for the power of Simulink verification, automatic code generation and deployment on a specific hardware as well as real time simulation.

The open nature of the platform and the universality of its use makes it a good approach for similar systems of different sensor types. For signal acquisition, the user can simply replace the libraries which describe the sensor registers ("sensor.h" and "sensor.c") in order to have immediately a ready-to-use I2C communicating sensor and extract the wanted data from it, after making the proper modifications to the S-Function Block. The user can also add multiple sensors in a single project, or use it for an actuation control like motors and manipulators.

Systems drivers available in Linux kernels offer a simple and reliable way to interface external components of various communication bus types like (UART, SPI, I2C, CAN, ..etc) and the availability of their controllers on low cost development boards give an rich platform for university research and engineering.

Zynq SoC provides in addition to the PS part (ARM) many other implementation possibilities, whose manipulation is to be discovered and used in later projects to take advantage of their strength and diversification.

The example project of AHRS filter algorithm can be extended to other inertial-based projects, like navigation, fall or shock detection, tracking and many more.

Results from our orientation filter simulation have shown a considerable amount of drift over time (about 270^o in 5 minutes), depending on sampling rate and filter paramter β. Thus, viewing this amount of drift it would be impractical to use it for actual products. To overcome this problem, addition of a magnetometer is considerable to provide a correction parameter for yaw drift compensation, as it adds a second relative reference to the earth gravitational field by measuring the earths magnetic field. Thus, the MARG version of the GDOF filter can issue a single solution, as confirmed in the author's report.

Bibliography

[1] V. Kempe, *Inertial MEMS: Principles and Practice*. Cambridge ; New York: Cambridge University Press, Feb. 17, 2011, 492 pp., ISBN: 9780521766586.

[2] EdisonTechCenter, *How early inertial guidance worked*, in collab. with YouTube Help, Apr. 17, 2015. [Online]. Available: `https://www.youtube.com/watch?v=la4-F93ADw8` (visited on 05/29/2015).

[3] O. W. S. of Hill Air Force Base. (Jan. 29, 2007). Factsheets : ICBM inertial measurement unit, [Online]. Available: `http://www.hill.af.mil/library/factsheets/factsheet.asp?id=5763` (visited on 05/31/2015).

[4] A. Noureldin, T. B. Karamat, and J. Georgy, *Fundamentals of inertial navigation, satellite-based positioning and their integration*. Springer Science & Business Media, 2012.

[5] W. Gnthner, "Enhancing cognitive assistance systems with inertial measurement units (studies in computational intelligence)", 2008.

[6] J. Fraden, *Handbook of Modern Sensors: Physics, Designs, and Applications*. Springer Science & Business Media, Sep. 22, 2010, 671 pp., ISBN: 9781441964663.

[7] O. Woodman and R. Harle, "Pedestrian localisation for indoor environments", in *Proceedings of the 10th international conference on Ubiquitous computing*, ACM, 2008, pp. 114–123.

[8] A. R. Jha, *MEMS and nanotechnology-based sensors and devices for communications, medical and aerospace applications*. CRC Press, 2008. (visited on 05/31/2015).

[9] C. et al. (Nov. 2014). High-end gyroscopes, accelerometers and IMUs for defense, aerospace & industrial, I-Micronews, [Online]. Available: `http://www.i-micronews.com/high-end-gyroscopes-accelerometers-and-imus-for-defense-aerospace-industrial` (visited on 06/01/2015).

[10] ZeptoBars. (). Invensense MPU6050 6-axis MEMS IMU, [Online]. Available: `http://zeptobars.ru/en/read/Invensense-MPU6050-6d-MEMS-IMU-gyroscope-accelerometer` (visited on 05/31/2015).

[11] D. E. Serrano. (). Design and analysis of MEMS accelerometers, [Online]. Available: `http://ieee-sensors2013.org/sites/ieee-sensors2013.org/files/Serrano_Accels.pdf` (visited on 05/31/2015).

[12] P. Aggarwal, Z. Syed, and N. El-Sheimy, *MEMS-based integrated navigation*. Artech House, 2010.

[13] iFixi. (). iPhone 4 gyroscope teardown, [Online]. Available: `https://www.ifixit.com/Teardown/iPhone+4+Gyroscope+Teardown/3156` (visited on 05/04/2015).

[14] U. Stanford. (). Disk resonator gyroscope (DRG), [Online]. Available: `http://stanford.edu/~ahn1229/DRG/DRG.html` (visited on 05/30/2015).

[15] (). Model-based design, [Online]. Available: http://www.mathworks.com/model-based-design/ (visited on 06/03/2015).

[16] (). dSPACE - model based control design, [Online]. Available: https://www.dspace.com/en/pub/home/products/systems/controldesign.cfm (visited on 06/03/2015).

[17] (). Model-based systems engineering and avionics control systems, [Online]. Available: http://www.ansys.com/Resource+Library/Webinars/Model-based+Systems+Engineering+and+Avionics+Control+Systems (visited on 06/03/2015).

[18] (). Model-based design, [Online]. Available: http://www.mathworks.com/model-based-design/?refresh=true (visited on 06/23/2015).

[19] (). DO-178c and related standards - MATLAB and simulink, [Online]. Available: http://www.mathworks.com/solutions/aerospace-defense/standards/do-178c.html?s_iid=ovp_custom1_-95160_pm (visited on 06/04/2015).

[20] (). DO-178b - MATLAB and simulink, [Online]. Available: http://www.mathworks.com/solutions/aerospace-defense/standards/do-178b.html (visited on 06/04/2015).

[21] (). Simulink embedded coder getting started, [Online]. Available: http://www.mathworks.com/products/datasheets/pdf/embedded-coder.pdf (visited on 06/04/2015).

[22] (). Zedboard, [Online]. Available: http://zedboard.org/ (visited on 06/10/2015).

[23] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685817 (visited on 06/10/2015).

[24] (). MPU-6050 6-axis accelerometer/gyroscope | i2c device library, [Online]. Available: http://www.i2cdevlib.com/devices/mpu6050#links (visited on 06/23/2015).

[25] (). I2c - what's that? - i2c bus, [Online]. Available: http://www.i2c-bus.org/ (visited on 06/10/2015).

[26] (). Get cardboard – google, [Online]. Available: https://www.google.com/get/cardboard/get-cardboard/ (visited on 06/14/2015).

[27] (). Mathworks support - supported and compatible compilers - release 2014b, [Online]. Available: http://www.mathworks.com/support/compilers/R2014b/index.html?sec=win64 (visited on 06/08/2015).

[28] (). Digilent pmod interface specification, [Online]. Available: https://www.digilentinc.com/Pmods/Digilent-Pmod_%20Interface_Specification.pdf (visited on 06/04/2015).

[29] (). Available mpu6050 c projects at github, [Online]. Available: https://github.com/search?l=C&q=MPU6050&type=Repositories&utf8=%C3%A2%C2%9C%C2%93 (visited on 06/08/2015).

[30] (). MPU-6050 6-axis accelerometer/gyroscope | i2c device library, [Online]. Available: http://www.i2cdevlib.com/devices/mpu6050#links (visited on 06/08/2015).

[31] P. Batista, C. Silvestre, P. Oliveira, and B. Cardeira, "Low-cost attitude and heading reference system: filter design and experimental evaluation", in *2010 IEEE International Conference on Robotics and Automation (ICRA)*, May 2010, pp. 2624–2629. DOI: 10.1109/ROBOT.2010.5509537.

[32] A. Cavallo, A. Cirillo, P. Cirillo, G. De Maria, P. Falco, C. Natale, and S. Pirozzi, "Experimental comparison of sensor fusion algorithms for attitude estimation", in *19th World Congress of the International Federation of Automatic Control, Cape Town, South Africa*, 2014. [Online]. Available: http://www.nt.ntnu.no/users/skoge/prost/proceedings/ifac2014/media/files/1173.pdf (visited on 06/14/2015).

[33] S. O. Madgwick, A. J. Harrison, and R. Vaidyanathan, "Estimation of IMU and MARG orientation using a gradient descent algorithm", in *Rehabilitation Robotics (ICORR), 2011 IEEE International Conference on*, IEEE, 2011, pp. 1–7. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5975346 (visited on 06/09/2015).

[34] S. O. Madgwick, "An efficient orientation filter for inertial and inertial/magnetic sensor arrays", *Report x-io and University of Bristol (UK)*, 2010. [Online]. Available: http://sharenet-wii-motion-trac.googlecode.com/files/An_efficient_orientation_filter_for_inertial_and_inertialmagnetic_sensor_arrays.pdf (visited on 06/14/2015).

[35] R. Mahony, T. Hamel, and J.-M. Pflimlin, "Nonlinear complementary filters on the special orthogonal group", *Automatic Control, IEEE Transactions on*, vol. 53, no. 5, pp. 1203–1218, 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4608934 (visited on 06/18/2015).

A

Appendix

Sensor_MPU6050_wrapper.c

```
/*
 * Include Files
 *
 */
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
# ifndef MATLAB_MEX_FILE
#include <math.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include "mpu6050.h"
#include "lsquaredc.h"
#include "mpu6050.c"
#include "lsquaredc.c"
#define MPU6050_ADDR 0x68
#define I2C_BUS 1

int mpu6050_addr = MPU6050_ADDR_A0_L;
int fd_mpu6050;
struct MPU6050 *sensor;

 int sensitivity_accel;
 int sensitivity_gyr;


#endif
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width
#define y_width 1
/*
 * Create external references here.
 *
 */
/* %%%-SFUNWIZ_wrapper_externs_Changes_BEGIN --- EDIT HERE TO _END */
/* extern double func(double a); */
/* %%%-SFUNWIZ_wrapper_externs_Changes_END --- EDIT HERE TO _BEGIN */

/*
 * Output functions
 *
 */
void Sensor_MPU6050_Outputs_wrapper(real_T *acc_x,
            real_T *acc_y,
```

```
            real_T *acc_z,
            real_T *gyr_x,
            real_T *gyr_y,
            real_T *gyr_z,
            const real_T *xD)
{
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
if (xD[0]==1)
{
   #ifndef MATLAB_MEX_FILE


acc_x[0] = ((float)
(mpu6050_read_register_pair(fd_mpu6050,MPU6050_ACCEL_XOUT_H))) /
sensitivity_accel;

acc_y[0]= ((float)
(mpu6050_read_register_pair(fd_mpu6050,MPU6050_ACCEL_YOUT_H))) /
sensitivity_accel;

acc_z[0] = ((float)
(mpu6050_read_register_pair(fd_mpu6050,MPU6050_ACCEL_ZOUT_H))) /
sensitivity_accel;

gyr_x[0] = ((float)
(mpu6050_read_register_pair(fd_mpu6050,MPU6050_GYRO_XOUT_H))) /
sensitivity_gyr;

gyr_y[0] = ((float)
(mpu6050_read_register_pair(fd_mpu6050,MPU6050_GYRO_YOUT_H))) /
sensitivity_gyr;

gyr_z[0] = ((float)
(mpu6050_read_register_pair(fd_mpu6050,MPU6050_GYRO_ZOUT_H))) /
sensitivity_gyr;

#endif

}
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_END --- EDIT HERE TO _BEGIN */
}

/*
  * Updates function
  *
  */
void Sensor_MPU6050_Update_wrapper(const real_T *acc_x,
            const real_T *acc_y,
            const real_T *acc_z,
            const real_T *gyr_x,
            const real_T *gyr_y,
            const real_T *gyr_z,
            real_T *xD)
{
  /* %%%-SFUNWIZ_wrapper_Update_Changes_BEGIN --- EDIT HERE TO _END */
if (xD[0]!=1)
{
   #ifndef MATLAB_MEX_FILE
    fd_mpu6050 = mpu6050_init(I2C_BUS,MPU6050_ADDR);
```

```
    // select sensivity_range
    sensitivity_accel =
mpu6050_select_range_accel(fd_mpu6050,AF_SEL_ACCEL_RANGE_16G);

    sensitivity_gyr   =
mpu6050_select_range_gyr(fd_mpu6050,AF_SEL_GYR_RANGE_2000_DPS);

    // bring it online :)
    mpu6050_power_on(fd_mpu6050);
    // check if we are connected to proper chip - should return it's I2C
address

  #endif
    xD[0]=1;
}
/* %%%-SFUNWIZ_wrapper_Update_Changes_END --- EDIT HERE TO _BEGIN */
}
```

Madgwick_IMU_wrapper.c

```
/*
 * Include Files
 *
 */
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
# ifndef MATLAB_MEX_FILE
#include <math.h>


#define deltat 0.005
#define gyroMeasError 3.14 * (2.0 / 180.0)
#define beta 0.3

double  SEq_1 , SEq_2 , SEq_3 , SEq_4 ;

double a_x, a_y, a_z;
double  w_x, w_y, w_z;







double  norm;
double  SEqDot_omega_1, SEqDot_omega_2, SEqDot_omega_3, SEqDot_omega_4;
double  f_1, f_2, f_3;
double  J_11or24, J_12or23, J_13or22, J_14or21, J_32, J_33;
double  SEqHatDot_1, SEqHatDot_2, SEqHatDot_3, SEqHatDot_4;

 #endif
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width 1
#define y_width 1
/*
 * Create external references here.
 *
 */
/* %%%-SFUNWIZ_wrapper_externs_Changes_BEGIN --- EDIT HERE TO _END */
/* extern double func(double a); */
/* %%%-SFUNWIZ_wrapper_externs_Changes_END --- EDIT HERE TO _BEGIN */

/*
 * Output functions
 *
 */
void Madgwick_IMU_Outputs_wrapper(const real_T *acc_x,
            const real_T *acc_y,
            const real_T *acc_z,
            const real_T *gyr_x,
            const real_T *gyr_y,
            const real_T *gyr_z,
```

```
            real_T *q_1,
            real_T *q_2,
            real_T *q_3,
            real_T *q_4,
            const real_T *xD)
{
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
if (xD[0]==1)
{

# ifndef MATLAB_MEX_FILE

a_x =acc_x[0];
a_y =acc_y[0];
a_z =acc_z[0];


w_x = gyr_x[0];
w_y = gyr_y[0];
w_z = gyr_z[0];


// Axulirary variables to avoid reapeated calcualtions
float halfSEq_1 = 0.5 * SEq_1;
float halfSEq_2 = 0.5 * SEq_2;
float halfSEq_3 = 0.5 * SEq_3;
float halfSEq_4 = 0.5 * SEq_4;
float twoSEq_1 = 2.0 * SEq_1;
float twoSEq_2 = 2.0 * SEq_2;
float twoSEq_3 = 2.0 * SEq_3;



// Normalise the accelerometer measurement
norm = sqrt(a_x * a_x + a_y * a_y + a_z * a_z);
a_x /= norm;
a_y /= norm;
a_z /= norm;

// Compute the objective function and Jacobian
f_1 = twoSEq_2 * SEq_4 - twoSEq_1 * SEq_3 - a_x;
f_2 = twoSEq_1 * SEq_2 + twoSEq_3 * SEq_4 - a_y;
f_3 = 1.0 - twoSEq_2 * SEq_2 - twoSEq_3 * SEq_3 - a_z;

J_11or24 = twoSEq_3; // J_11 negated in matrix multiplication
J_12or23 = 2.0 * SEq_4;
J_13or22 = twoSEq_1; // J_12 negated in matrix multiplication
J_14or21 = twoSEq_2;
J_32 = 2.0 * J_14or21; // negated in matrix multiplication
J_33 = 2.0 * J_11or24; // negated in matrix multiplication


// Compute the gradient (matrix multiplication)
SEqHatDot_1 = J_14or21 * f_2 - J_11or24 * f_1;
SEqHatDot_2 = J_12or23 * f_1 + J_13or22 * f_2 - J_32 * f_3;
SEqHatDot_3 = J_12or23 * f_2 - J_33 * f_3 - J_13or22 * f_1;
SEqHatDot_4 = J_14or21 * f_1 + J_11or24 * f_2;


// Normalise the gradient
```

```
norm = sqrt(SEqHatDot_1 * SEqHatDot_1 + SEqHatDot_2 * SEqHatDot_2 +
SEqHatDot_3 * SEqHatDot_3 + SEqHatDot_4 * SEqHatDot_4);
SEqHatDot_1 /= norm;
SEqHatDot_2 /= norm;
SEqHatDot_3 /= norm;
SEqHatDot_4 /= norm;

// Compute the quaternion derrivative measured by gyroscopes
SEqDot_omega_1 = -halfSEq_2 * w_x - halfSEq_3 * w_y - halfSEq_4 * w_z;
SEqDot_omega_2 = halfSEq_1 * w_x + halfSEq_3 * w_z - halfSEq_4 * w_y;
SEqDot_omega_3 = halfSEq_1 * w_y - halfSEq_2 * w_z + halfSEq_4 * w_x;
SEqDot_omega_4 = halfSEq_1 * w_z + halfSEq_2 * w_y - halfSEq_3 * w_x;

// Compute then integrate the estimated quaternion derrivative
SEq_1 += (SEqDot_omega_1 - (beta * SEqHatDot_1)) * deltat;
SEq_2 += (SEqDot_omega_2 - (beta * SEqHatDot_2)) * deltat;
SEq_3 += (SEqDot_omega_3 - (beta * SEqHatDot_3)) * deltat;
SEq_4 += (SEqDot_omega_4 - (beta * SEqHatDot_4)) * deltat;

// Normalise quaternion
norm = sqrt(SEq_1 * SEq_1 + SEq_2 * SEq_2 + SEq_3 * SEq_3 + SEq_4 * SEq_4);
q_1[0] = SEq_1/ norm;
q_2[0] = SEq_2/ norm;
q_3[0] = SEq_3/ norm;
q_4[0] = SEq_4/ norm;


SEq_1=q_1[0];
SEq_2=q_2[0];
SEq_3=q_3[0];
SEq_4=q_4[0];

 #endif

}
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_END --- EDIT HERE TO _BEGIN */
}

/*
  * Updates function
  *
  */
void Madgwick_IMU_Update_wrapper(const real_T *acc_x,
            const real_T *acc_y,
            const real_T *acc_z,
            const real_T *gyr_x,
            const real_T *gyr_y,
            const real_T *gyr_z,
            const real_T *q_1,
            const real_T *q_2,
            const real_T *q_3,
            const real_T *q_4,
            real_T *xD)
{
  /* %%%-SFUNWIZ_wrapper_Update_Changes_BEGIN --- EDIT HERE TO _END */
if (xD[0]!=1)
{


    // estimated orientation quaternion elements with initial conditions
```

```
# ifndef MATLAB_MEX_FILE

SEq_1=1;
SEq_2=0 ;
SEq_3=0;
SEq_4=0 ;



 #endif

    xD[0]=1;
}
/* %%%-SFUNWIZ_wrapper_Update_Changes_END --- EDIT HERE TO _BEGIN */
}
```

Quaternion_SFRotation_wrapper.c

```
/*
 * Include Files
 *
 */
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
#include <math.h>
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width 1
#define y_width 1
/*
 * Create external references here.
 *
 */
/* %%%-SFUNWIZ_wrapper_externs_Changes_BEGIN --- EDIT HERE TO _END */
/* extern double func(double a); */
/* %%%-SFUNWIZ_wrapper_externs_Changes_END --- EDIT HERE TO _BEGIN */

/*
 * Output functions
 *
 */
void Quaternion_SFRotation_Outputs_wrapper(const real_T *inQuat_q_1,
            const real_T *inQuat_q_2,
            const real_T *inQuat_q_3,
            const real_T *inQuat_q_4,
            real_T *outSFRot_r_1,
            real_T *outSFRot_r_2,
            real_T *outSFRot_r_3,
            real_T *outSFRot_r_4)
{
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */
// Declare Temporary Holder Variables
double dSFRot_r_1=0;
double dSFRot_r_2=0;
double dSFRot_r_3=0;
double dSFRot_r_4=0;
double dSFRot_Angle=0;


// Calculate Output Data

// Rotation Angle
dSFRot_Angle=2*acos(inQuat_q_1[0]);

if (dSFRot_Angle!=0)
{
  dSFRot_r_1=inQuat_q_2[0]/sin(dSFRot_Angle/2);
  dSFRot_r_2=inQuat_q_3[0]/sin(dSFRot_Angle/2);
  dSFRot_r_3=inQuat_q_4[0]/sin(dSFRot_Angle/2);
}
else
{
```

```
  dSFRot_r_1=0;
  dSFRot_r_2=0;
  dSFRot_r_3=0;
}

// Set Output Data
//
// outSFRot_r_1[0]=dSFRot_r_1;
// outSFRot_r_2[0]=dSFRot_r_2;
// outSFRot_r_3[0]=dSFRot_r_3;
// outSFRot_r_4[0]=dSFRot_Angle;

//
// Correct for -Y at Aircaft Head!
outSFRot_r_1[0]=dSFRot_r_2;
outSFRot_r_2[0]=-dSFRot_r_3;
outSFRot_r_3[0]=dSFRot_r_1;
outSFRot_r_4[0]=dSFRot_Angle;


// Correct for +Y at Aircaft Head!
// outSFRot_r_1[0]=-dSFRot_r_1;
// outSFRot_r_2[0]=-dSFRot_r_3;
// outSFRot_r_3[0]=dSFRot_r_2;
// outSFRot_r_4[0]=dSFRot_Angle;
/* %%%-SFUNWIZ_wrapper_Outputs_Changes_END --- EDIT HERE TO _BEGIN */
}
```

B

Appendix

Gyroscope Specifications:

VDD = 2.375V-3.46V, VLOGIC = 1.8V±5% or VDD, T_A = 25°C

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
GYROSCOPE SENSITIVITY						
Full-Scale Range	FS_SEL=0		±250		°/s	
	FS_SEL=1		±500		°/s	
	FS_SEL=2		±1000		°/s	
	FS_SEL=3		±2000		°/s	
Gyroscope ADC Word Length			16		bits	
Sensitivity Scale Factor	FS_SEL=0		131		LSB/(°/s)	
	FS_SEL=1		65.5		LSB/(°/s)	
	FS_SEL=2		32.8		LSB/(°/s)	
	FS_SEL=3		16.4		LSB/(°/s)	
Sensitivity Scale Factor Tolerance	25°C	-3		+3	%	
Sensitivity Scale Factor Variation Over Temperature			±2		%	
Nonlinearity	Best fit straight line; 25°C		0.2		%	
Cross-Axis Sensitivity			±2		%	
GYROSCOPE ZERO-RATE OUTPUT (ZRO)						
Initial ZRO Tolerance	25°C		±20		°/s	
ZRO Variation Over Temperature	-40°C to +85°C		±20		°/s	
Power-Supply Sensitivity (1-10Hz)	Sine wave, 100mVpp; VDD=2.5V		0.2		°/s	
Power-Supply Sensitivity (10 - 250Hz)	Sine wave, 100mVpp; VDD=2.5V		0.2		°/s	
Power-Supply Sensitivity (250Hz - 100kHz)	Sine wave, 100mVpp; VDD=2.5V		4		°/s	
Linear Acceleration Sensitivity	Static		0.1		°/s/*g*	
SELF-TEST RESPONSE						
Relative	Change from factory trim	-14		14	%	1
GYROSCOPE NOISE PERFORMANCE	**FS_SEL=0**					
Total RMS Noise	DLPFCFG=2 (100Hz)		0.05		°/s-rms	
Low-frequency RMS noise	Bandwidth 1Hz to10Hz		0.033		°/s-rms	
Rate Noise Spectral Density	At 10Hz		0.005		°/s/ √ Hz	
GYROSCOPE MECHANICAL FREQUENCIES						
X-Axis		30	33	36	kHz	
Y-Axis		27	30	33	kHz	
Z-Axis		24	27	30	kHz	
LOW PASS FILTER RESPONSE						
	Programmable Range	5		256	Hz	
OUTPUT DATA RATE						
	Programmable	4		8,000	Hz	
GYROSCOPE START-UP TIME	**DLPFCFG=0**					
ZRO Settling (from power-on)	to ±1°/s of Final		30		ms	

Accelerometer Specifications:

VDD = 2.375V-3.46V, VLOGIC = 1.8V±5% or VDD, T_A = 25°C

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
ACCELEROMETER SENSITIVITY						
Full-Scale Range	AFS_SEL=0		±2		*g*	
	AFS_SEL=1		±4		*g*	
	AFS_SEL=2		±8		*g*	
	AFS_SEL=3		±16		*g*	
ADC Word Length	Output in two's complement format		16		bits	
Sensitivity Scale Factor	AFS_SEL=0		16,384		LSB/*g*	
	AFS_SEL=1		8,192		LSB/*g*	
	AFS_SEL=2		4,096		LSB/*g*	
	AFS_SEL=3		2,048		LSB/*g*	
Initial Calibration Tolerance			±3		%	
Sensitivity Change vs. Temperature	AFS_SEL=0, -40°C to +85°C		±0.02		%/°C	
Nonlinearity	Best Fit Straight Line		0.5		%	
Cross-Axis Sensitivity			±2		%	
ZERO-G OUTPUT						
Initial Calibration Tolerance	X and Y axes		±50		*mg*	1
	Z axis		±80		*mg*	
Zero-G Level Change vs. Temperature	X and Y axes, 0°C to +70°C		±35			
	Z axis, 0°C to +70°C		±60		*mg*	
SELF TEST RESPONSE						
Relative	Change from factory trim	-14		14	%	2
NOISE PERFORMANCE						
Power Spectral Density	@10Hz, AFS_SEL=0 & ODR=1kHz		400		µg/ √Hz	
LOW PASS FILTER RESPONSE						
	Programmable Range	5		260	Hz	
OUTPUT DATA RATE						
	Programmable Range	4		1,000	Hz	
INTELLIGENCE FUNCTION INCREMENT			32		*mg*/LSB	