Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Miniature navigation system for UAS application

Grigoris Mantaos

Supervisor: doc. Ing. Jan Roháč, Ph.D.
May 2022

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Mantaos Grigoris Panagiotis**
Personal ID number:	**498100**
Faculty / Institute:	**Faculty of Electrical Engineering**
Department / Institute:	**Department of Measurement**
Study program:	**Aerospace Engineering**
Branch of study:	**Avionics**

II. Master’s thesis details

Master’s thesis title in English:

Miniature navigation system for UAS application

Master’s thesis title in Czech:

Miniaturní navigační systém pro bezpilotní létající prostředky

Guidelines:

Design and develop a light-weighted and small-sized inertial navigation system (miniINS) consisting of inertial measurement unit, GNSS receiver, and pressure measurement unit as key parts. Specify the requirements based on which the miniINS should be composed of and further developed. The choice of components should be made based on a market analysis. Use Altium Designer/KiCAD for the miniINS development. Second part of the assignment is related to the navigation solution incorporated into the miniINS. Extend a given navigation algorithm written in C/C++ language with a data collecting part which should ensure synchronism withing the system and finally provide navigation solution in terms of position, velocity, and attitude as the solution outputs provided via CAN bus. The main part of the algorithm will be provided, nevertheless, based on chosen sensors and aiding, it will require tuning. Perform practical experiments with the miniINS put on a UAS and test and verify the final solution. Specify the reached accuracy.

Bibliography / sources:

[1] S. Grewal / P. Andrews: Kalman Filtering – Theory and Practice using Matlab ® 3rd edition 2008, Wiley.
[2] J. A. Farrell / M. Barth: The global positioning system & inertial navigation 2nd edition 1999, Mc Graw Hill.
[3] S.Grewal et al: Global positioning systems, inertial navigation, and integration, 2nd edition 2007, John Wiley & Sons.
[4] R.M.Rogers: Applied mathematics in integrated navigation systems, 2nd edition, 2003 AIAA

Name and workplace of master’s thesis supervisor:

doc. Ing. Jan Roháč, Ph.D. Department of Measurement FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **07.02.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until:
by the end of summer semester 2022/2023

doc. Ing. Jan Roháč, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.
Date of assignment receipt Student’s signature

Declaration

I declare that the presented work was developed independently, and that I have listed all sources of information used within it, in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 20, 2022

[illegible]

Abstract

This diploma thesis will outline the design and programming of an Inertial Navigation System, dubbed *Mini-INS*. The completed system will, like most INS platforms, provide as output the 9-DOF state of a host vehicle in terms of *Position, Velocity and Attitude* (PVA). Specifically, the implementation will focus on the navigational requirements of *Unmanned Aerial Systems* (UAS). However, no discrimination shall be made on flight mechanics, meaning that the *Mini-INS* should be a thorough *plug-and-play* solution for any aerial or land vehicle, be it a fixed-wing drone, a multi-copter, or a caterpillar tracked robot.

In the context of this implementation, the system is designed and manufactured from top to bottom. This includes the hardware design of a schematic, its realization on a *Printed Circuit Board* (PCB), and its programming so that it may execute the functions of an INS. The primary objective, as the title suggests, is to build a navigation system for use by small unmanned vehicles. As a consequence, the focus of the design will be to minimize the size of the PCB footprint, as well as the board's total power consumption. To achieve this, the board utilizes a combination of MEMS sensors and a low-power ARM microprocessor.

Keywords: UAV, IMU, INS, GNSS, Aided Navigation, Inertial Navigation, Kalman filter

Supervisor: doc. Ing. Jan Roháč, Ph.D.

Contents

Chapter 1 Introduction

This document represents a written essay, which is part of the diploma thesis leading up to the author's graduation from the *Aerospace Engineering* program at the *Czech Technical University* of Prague. The topic of the assignment is the planning, design, and realization of an *Inertial Navigation System*. Such a unit is typically tasked with providing position, velocity, and attitude information to the vehicle it is affixed to. The definite system to be actualized shall employ principles of aided navigation, which involve the complementary fusion of high-rate inertial sensors with a GNSS receiver. The motivation behind the development of this system, is that it may be utilized by vehicles which require the evaluation of these quantities at a high frequency. Examples of such vehicles include autonomous cars, unmanned aerial systems, guided missiles, locomotive robots, and more.

In the interest of fulfilling the objectives outlined by the assignment, certain elements of the implementation were given to the author by his thesis supervisor. These include unequivocal guidance towards the understanding and implementation of navigational principles; instructions on the effectuation of aided navigation; and implementation of a linear Kalman filter model. Specifically, a working version of a Kalman filter implementation in MATLAB was also provided.

The author was tasked with the abstract design of the INS, as well as its physical formulation. The explicit objectives were the operative conception of the system, the design of a circuit schematic that would fulfill these functions on a *Printed Circuit Board*, and the programming of the navigational functionality on an embedded microcontroller. The latter also includes the adaptation and optimization of the existing Kalman filter implementation from MATLAB to the embedded firmware. Additional goals of this endeavor were that the system be designed in such a way that it is physically very small, and operates with a low power consumption.

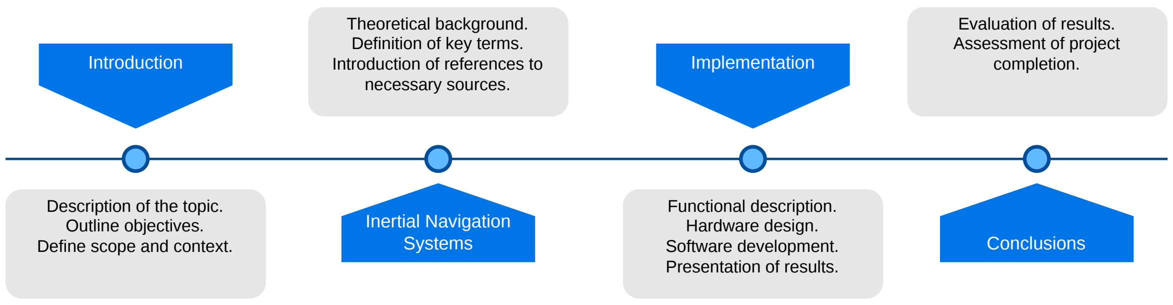


Figure 1.1: Overview of the thesis chapter structure.

This thesis is logically segmented into four distinct chapters, as shown in figure 1.1.

Chapter 2
Inertial navigation systems

This chapter will provide the necessary theoretical foundation for the implementation of the technical part of the diploma thesis. Given that the final product will enact the functionality of an INS, the necessary theory to study spans from navigational basics, all the way to probabilistic state estimations. In this chapter, this overview will be split into two sections. The first one endeavors to familiarize the reader with key terms and definitions used throughout this document, as well as to introduce all references to external sources that were drawn from during the development process. The second section summarizes all information from the sources cited in the first one, that is directly relevant to the realization of the Mini-INS, such as mathematical equations.

2.1 Theoretical background

2.1.1 Frames of reference

Position, and by extension velocity and acceleration, are inherently relative. Evidently, to define any sort of position or orientation, it is necessary to also declare the frame of reference, to which said position and orientation are relative. As with most navigational solutions, the following base frames are needed.

- **Inertial frame i**
  As described in [1]: according to Newtonian dynamics, the inertial reference frame is one where a body not subject to any force, will always be in rectilinear and uniform motion. This is usually important to define in cases such as this, where inertial measurement units are involved, since all inertial sensors produce measurements relative to an inertial frame [2].
- **Sensor frame s**
  This frame generally needs to be defined separately when dealing with *Inertial Measurement Units* (IMU), as later sections will illustrate. This is because an IMU is typically sensitive along a set of physical axes calibrated by the manufacturer, which may not be parallel to the vehicle's axes once installed.
- **Platform frame p**
  This is an intermediate frame, which is useful in case multiple sensors are placed on the same PCB, but with different orientations. In this case, each sensor will need to have its tri-axial measurement vector rotated to the platform frame. This frame typically originates at the center of the PCB, has its z axis perpendicular to the PCB surface pointing out from the bottom side, and its other two axes pointing one length-wise and another width-wise. Then, a fixed rotation can be defined, which will convert measured

forces from the platform frame to the body frame, which will depend on how the PCB is mounted on the vehicle with respect to its primary body axes.

The motivation for this auxiliary frame is to separate the rotation from the sensor frame to the body frame into two distinct parts. The first part, which is the rotation from s to p, can be hard-coded for each sensor during manufacturing. And the second one, which is the rotation from p, can be the only one that has to be configured by the user of the platform.

- **Body frame b**

  The body frame, also sometimes called the vehicle frame, is a local reference frame that remains rigidly attached to the body of the vehicle [2]. This type of frame typically consists of an origin placed at the vehicle's expected center of gravity, and three orthogonal axes.

- **Navigation frame n**

  The navigation frame is constructed by placing a tangent plane at some point on the geodetic reference ellipse [2]. Its purpose is to help define not only orientation with reference to true north and the ground, but also the local positions and velocities. The relationship between the body frame and the navigation frame is shown in figure 2.1.

  Two of the most common local frames are the *North East Down* (NED) and *East North Up* (ENU), whose three-worded names denote the direction of the n, e and d axes.

- **Geographic frame e**

  Whereas the navigation frame is used to define local attitude and position, the geographic frame is used to define global position relative to the Earth. If to work with Cartesian coordinates, the *Earth-Centered, Earth-Fixed* (ECEF) is used, which originates at the center of the earth and defines its axes according to the North Pole and prime meridian. For this work however, as in most navigational implementations, the geodetic coordinate system is used, which defines a point's position in terms of *latitude* $\hat{\phi}$, *longitude* $\hat{\lambda}$, and *altitude* $\hat{h}$.

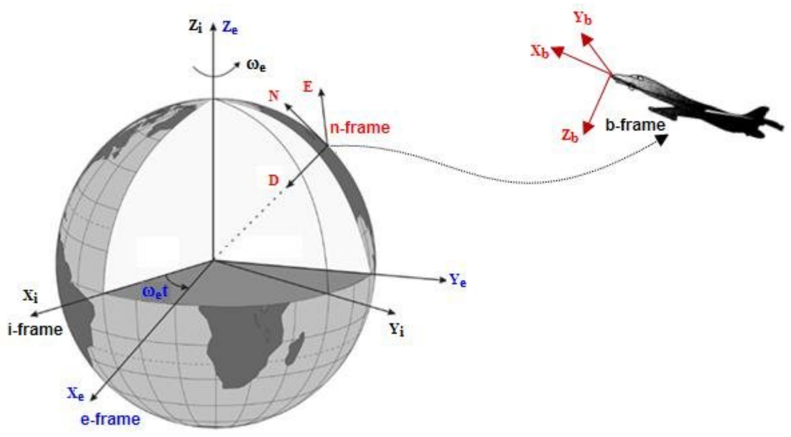


Figure 2.1: Reference coordinates frames in inertial navigation systems. [3]

2.1.2 Inertial Measurement Units

Inertial Measurement Units (IMU) are sets of sensors which typically measure angular rate or specific force relative to an inertial frame. Utilizing such measurements, it becomes possible for the system to calculate its velocity and orientation. Because usually multiple sensors are embedded together to fulfill this purpose, they have come to be treated as a single unit referred to as an IMU. According to [4], there are two categories:

- **IMU with accelerometer and gyroscope**
  The first kind are IMUs consisting of a tri-axial accelerometer, and a tri-axial gyroscope. A combination of the two, brings the total degrees of freedom up to six. These units can be used to determine a system's orientation, in terms of *roll* and *pitch* angles in the navigation frame.
- **IMU with accelerometer, gyroscope, and magnetometer**
  The second type still includes an accelerometer and gyroscope like the first, but is expanded with a tri-axial magnetometer. This makes for a total of nine degrees of freedom in this case. One of the main ways the magnetometer enhances the system, is by providing measurements that make it possible to compute the magnetic heading. The main disadvantage of the inclusion of the magnetometer, is that it is susceptible to external disturbances in the magnetic field, which may for example be common in a metropolitan environment.

Owing to technological innovations, widely-available IMUs can now be small, inexpensive, and accurate [5]. Accelerometers and gyroscopes can be calibrated, and their readings can usually be compensated for biases or linear thermal drift, which makes them quite adequate at providing accurate measurements with a high bandwidth in the short-term. In the long-term however, these sensors tend to suffer from considerable drift and other errors [6], which make them unsuitable for navigation systems in most cases since even small errors are compounded through integration. These inaccuracies are particularly profound when using MEMS sensors, such as in this diploma thesis.

According to [7], IMUs are generally characterized by one of four grades, according to their performance. Specifically, the key performance indicator that differentiates them is the bias instability of their gyroscopes.

- **Navigation grade** refers to IMUs with a gyro bias instability less than $0.1\ \frac{\circ}{h}$. These units are typically meant for military applications, since they can aid in navigation for up to several hours in GNSS-denied environments.
- **Tactical grade** refers to IMUs with a gyro bias instability less than $1\ \frac{\circ}{h}$. They can typically aid in navigation for up to 10 minutes in GNSS-denied environments, and are usually placed on guided munitions.
- **Industrial grade** refers to IMUs with a gyro bias instability less than $10\ \frac{\circ}{h}$. These can only aid in navigation for about a minute without GNSS assistance, and are used for UAVs.

- **Consumer grade** refers to IMUs that do not meet any of the above gyro bias instability thresholds. They are typically unsuitable for navigation, and are placed in consumer electronics.

2.1.3 Aided navigation

Aided navigation according to [2] is the combination of the inaccurate, but high-rate inertial sensors, with a low-rate but accurate sensor that can help the system compensate for the errors introduced by the former. One example of such a complementary combination, which is relevant in the context of this diploma thesis, is the combination of an IMU with a *Global Navigation Satellite System* (GNSS) receiver. Combinations like these are commonly referred to as sensor fusion [8], which is the practice of combining data from multiple sensors to obtain information that each individual sensor separately would never be able to ascertain. Alternatively, sensor fusion is a term still used to describe systems where multiple sensors effectively measure the same quantity, but utilizing different operation principles, and their combined operation offers expanded coverage, improved precision, or redundancy. What makes such a combination beneficial to the overall system, is the fact that the components being incorporated offer complementary characteristics, and measure non-overlapping quantities.

Specifically in the case of aided navigation, the combination of an IMU with a GNSS receiver allows for the system to satisfy its operational demands. It can now rely on the high bandwidth, rate, and short-term precision of the inertial sensors, while ensuring their long-term errors do not compromise the navigation system's reliability.

The GNSS receiver provides the system with position, velocity, and time. With this information, a system utilizing such sensor fusion is able to:

- Correct its position and velocity at a regular interval. This mitigates the compounding of errors from the integration of the acceleration and angular rates measured from the IMU. The GNSS receiver will, of course, introduce its own errors in the process. However, the idea here is that since these two systems determine position and velocity by measuring different quantities, their unified estimation will be a better one, since a source of error that affects one's measured quantities is unlikely to affect the other's as well.
- Estimate the current bias of the inertial sensors. This is possible using the knowledge of the error that was observed between the actual position and velocity, and the position and velocity calculated from the IMU.

Many methods exist for the fusion of complementary sensors in navigation systems. This document though, will focus exclusively on the application of a linear time-varying *Kalman Filter*, as it is described in [2].

2.1.4 Kalman filter

The Kalman filter, also known as *Linear Quadratic Estimator* (LQE), was first introduced by *R. E. Kalman* in 1960 [9]. It is a state estimator, which uses the joint probability distribution from a series of measurements of multiple variables. The motivation for this, depending on the physical characteristics of the system, is twofold: this probabilistic combination may improve the accuracy of measured quantities, and also potentially allow for the estimation of quantities that cannot be measured directly. An overview of its working principle is shown in figure 2.2. This document will not endeavor to delve into the theoretical details of Kalman filtering or its nonlinear models. Instead, the focus will lie entirely on implementing an existing model of the linear Kalman filter, adapting it for inertial navigation as proposed by the author of [2].

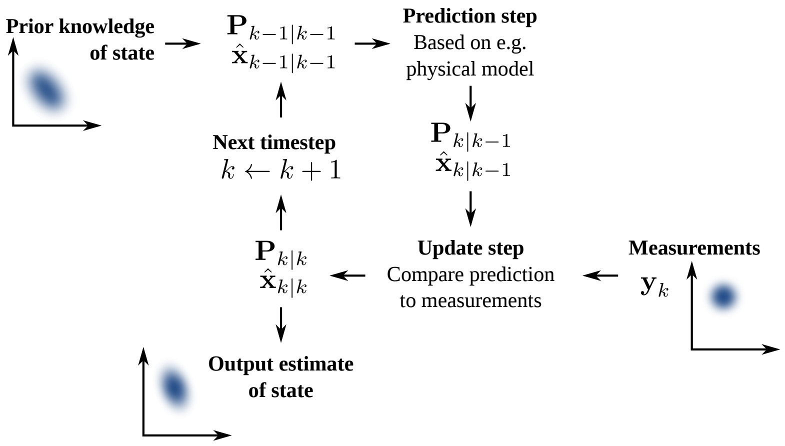


Figure 2.2: Basic concept of Kalman filtering. [10]

2.2 Navigation equations and algorithms

This section will summarize the works that are referenced in this document, by listing all mathematical equations and algorithms that are used in the implementation of this diploma thesis. They will be split into two subsections, separated logically according to each part's inputs and outputs. In the first one, the mechanization of the navigation equations will be described, which includes the computation of the system's *Position, Velocity and Attitude* (PVA) from the measurements of the IMU. The algorithm outlined in this first subsection, will need to be computed every time new measurements have been accumulated from the IMU, which will be required in order to provide an output of the state of the system at a high rate. The second subsection will detail the implementation of the linear Kalman filter, which will require GNSS position and velocity measurements as inputs, and estimate the current state error of the model. This algorithm will represent a complete Kalman filter update cycle, which will be executed at a lower rate only when new GNSS data are available.

Note that any rotation shall be defined via a *Direction Cosine Matrix* (DCM). Conventionally, these matrices will be symbolized as C_a^b to denote a rotation from frame a to frame b.

2.2.1 Mechanization of navigation equations

The inputs of this stage are the data from the IMU, which include measurements from an accelerometer, a gyroscope, and a magnetometer. Firstly, the tri-axial measurement vector from each sensor needs to be converted to the body frame, which, as described before, requires two rotations: one to the platform frame and one to the body frame.

$$f^b = C_p^b C_{as}^p \begin{bmatrix} f_x & f_y & f_z \end{bmatrix}^\mathsf{T} - b_a^b \tag{1}$$

$$\omega^b = C_p^b C_{gs}^p \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^\mathsf{T} - b_g^b \tag{2}$$

$$\eta^b = C_p^b C_{ms}^p \begin{bmatrix} \eta_x & \eta_y & \eta_z \end{bmatrix}^\mathsf{T} \tag{3}$$

$f_{\{x,y,z\}}$	— specific force measured along the accelerometer's x, y, or z axis
$\omega_{\{x,y,z\}}$	— angular rate measured around the gyroscope's x, y, or z axis
$\eta_{\{x,y,z\}}$	— magnetic field intensity measured along the magnetometer's x, y, or z axis
b_a^b	— 3x1 accelerometer bias in the body frame
b_g^b	— 3x1 gyroscope bias in the body frame
C_{as}^p	— DCM rotation from the accelerometer's sensor frame to the platform frame
C_{gs}^p	— DCM rotation from the gyroscope's sensor frame to the platform frame
C_{ms}^p	— DCM rotation from the magnetometer's sensor frame to the platform frame
C_p^b	— DCM rotation from the platform frame to the body frame

Note that, in equations (1) and (2), the biases are also subtracted. In the context of this section, these biases are initialized to zero, and updated using the Kalman filter which will be described later.

In the initialization phase of the system, the first series of IMU measurements will be used to calculate the initial attitude of the system in terms of Euler angles. The *roll* and *pitch* can be computed from the accelerometer [11], if to assume that the system is initialized in a standstill state where the only force being applied to it is the force of gravity. Then, knowing that, the *yaw* can be computed from the magnetometer using tilt compensation [12].

$$\phi = \arctan\left(\frac{-f_y^b}{\sqrt{(f_x^b)^2 + (f_z^b)^2}}\right) \tag{4}$$

$$\theta = \arctan\left(\frac{f_x^b}{-f_z^b}\right) \tag{5}$$

$$\psi = -\arctan\left(\frac{\eta_y^b \cos\phi + \eta_z^b \sin\phi}{\eta_x^b \cos\theta + \eta_y^b \sin\phi \sin\theta + \eta_z^b \cos\phi \sin\theta}\right) \tag{6}$$

For computational purposes, the attitude shall be kept as a DCM, which, as described in the previous section, shall define a rotation from the body frame to the navigation frame.

$$C_b^n = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix} \tag{7}$$

After initialization, the specific force from the accelerometer will no longer be used to determine attitude. In this state, the angular rates from the gyroscope will be the new source of attitude information, which will be computed through integration from the initial attitude. Following that, the specific force f from the accelerometer will be rotated to the navigation frame, then transformed to acceleration by compensating for gravity, and integrated to compute velocity.

The goal, as described previously, is to integrate the angular rates to obtain the system's orientation. However, since the attitude is expressed as a DCM, the desired integration can be expressed as a rotation of the DCM, which can be approximated by discrete additive updates over time to the C_b^n matrix. These updates are in the form of a skew-symmetric matrix rotated to the navigation frame.

$$\delta C_b^n = C_b^n \begin{bmatrix} 0 & -\omega_z^b & \omega_y^b \\ \omega_z^b & 0 & -\omega_x^b \\ -\omega_y^b & \omega_x^b & 0 \end{bmatrix} \tag{8}$$

δC_b^n — discrete additive update to the attitude DCM

Then, the centripetal acceleration can be computed in the body frame.

$$a_{centripetal}^b = \begin{bmatrix} 0 & -\omega_z^b & \omega_y^b \\ \omega_z^b & 0 & -\omega_x^b \\ -\omega_y^b & \omega_x^b & 0 \end{bmatrix} C_n^b v^n \tag{9}$$

v^n — velocity in the navigation frame

The centripetal acceleration that was computed in (9) should be subtracted from the specific force in the body frame. Once in the navigation frame, the gravity vector should now be added. This is because the accelerometer measures specific force, which is effectively acceleration minus gravity.

$$a^n = C_b^n \left(f^b - a_{centripetal}^b\right) + \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{10}$$

g — local gravity at the current latitude and altitude

The acceleration a is now in the navigation frame, which constitutes the mechanization update to the velocity vector. Note that these δ-updates are instantaneous, and will be multiplied by the time elapsed during the discrete integration to update their respective vectors.

$$\delta v^n = a^n \tag{11}$$

δv^n — mechanization update to velocity

Going further, to update the system's position, it is necessary to convert the velocity from the navigation frame to the global frame. This velocity in the global frame can then be used to update the global position.

$$\delta p^e = v^e = \begin{bmatrix} \frac{1}{R_M+\hat{h}} v_n^n \\ \\ \frac{1}{(R_N+\hat{h})\cos\hat{\phi}} v_e^n \\ \\ -v_d^n \end{bmatrix} \tag{12}$$

δp^e — mechanization update to the position vector
R_N — radius of curvature in the prime vertical
R_M — meridian radius of curvature
$\hat{\phi}$ — computed latitude
$\hat{h}$ — computed altitude

Finally, to complete a single update to the mechanization, the discrete updates computed previously need to be added to the PVA vectors. Since the integration occurs through time, these updates need to be multiplied by the time Δt, which is the time that elapsed since the previous update.

$$\begin{aligned} C_b^n &:= C_b^n & +\Delta t \cdot \delta C_b^n \\ v^n &:= v^n & +\Delta t \cdot \delta v^n \\ p^e &:= p^e & +\Delta t \cdot \delta p^e \end{aligned} \tag{13}$$

Δt — sample time
v^n — velocity in the NED frame, $\begin{bmatrix} v_n^n & v_e^n & v_d^n \end{bmatrix}^\intercal$
p^e — position in the LLA frame, $\begin{bmatrix} \hat{\phi} & \hat{\lambda} & \hat{h} \end{bmatrix}^\intercal$

2.2.2 Error and bias estimation

For this stage of computations, it is assumed that there already is some computed PVA state from the IMU measurements. This refers to the p^e, v^n, and C_b^n values that were computed in the previous subsection, for position, velocity and attitude respectively. The new inputs here are the position and velocity from a GNSS receiver. With these measurements, the system's state vector shall be estimated using a Kalman filter. This state, which represents estimated errors in position, velocity, attitude, and sensor biases, will in the end be used to correct those quantities in the navigation model.

The algorithm, in the context of this project, will have to run on an embedded microprocessor. Therefore, it is deemed preferable to implement the Kalman filter updates using the UDU factorization methods, which were proposed by Catherine Thornton in

[13] and Gerald Bierman in [14]. In this section, only the relevant equations will be written for these proposed measurement update and time update steps. For further details and evaluations of these methods, the reader is referred to the original publications.

As described in the previous section, the Kalman filter that was provided as a linearized model which is valid around the zero state. Therefore the state is described in terms of errors from the computed state to the actual state. This way, errors in the state will get corrected at regular intervals, allowing the model to operate around the zero state indefinitely. Specifically, the state vector in this implementation will have 15 values.

$$x \equiv \begin{bmatrix} \delta p & \delta v & \delta \rho & \delta b_a^b & \delta b_g^b \end{bmatrix}^\intercal \tag{14}$$

δp — 3x1 position error in geodetic frame

δv — 3x1 velocity error in navigation frame

$\delta \rho$ — 3x1 attitude error in Euler angles

δb_a^b — 3x1 error in accelerometer per-axis bias

δb_g^b — 3x1 error in gyroscope per-axis bias

Upon receiving the GNSS data, the first step is to compute the Kalman filter's measurement vector, which in this case is the error between the position and velocity from the GNSS, and the computed position and velocity from the IMU.

$$e = \begin{bmatrix} p_{GNSS} - p^e \\ v_{GNSS} - v^e \end{bmatrix} \tag{15}$$

e — measured residuals

p_{GNSS} — 3x1 position in the geodetic LLA frame from the GNSS

v_{GNSS} — 3x1 velocity in the navigation frame from the GNSS

p^e — 3x1 position in the global frame from the mechanization

v^n — 3x1 velocity in the navigation frame from the mechanization

Next is the Kalman filter measurement update step, where the model prediction is compared to the measurements to obtain an estimation of the state error. This step consists of the following set of equations [2]. As it was mentioned previously, the matrix inversion in the first equation, makes this computationally exorbitant for a standalone ARM microprocessor without a dedicated matrix multiplication unit. So, in practice it will not be used, and it is simply mentioned here for completeness.

$$\begin{aligned} K &:= P_k H^\intercal \left(H P_k H^\intercal + R\right)^{-1} \\ x_{k+1} &:= x_k + Ke \\ P_{k+1} &:= (I - KH) P_k (I - KH)^\intercal + KRK^\intercal \end{aligned} \tag{16}$$

$$\begin{aligned} P_k &- \text{prior covariance matrix} \\ P_{k+1} &- \text{posterior covariance matrix} \\ x_k &- \text{prior state vector} \\ x_{k+1} &- \text{posterior state vector} \end{aligned}$$

Instead of the infeasible equations in (16), Bierman's algorithm [14] will be used, which relies on the UDU factorization of the covariance matrix P. The entire algorithm is given below.

$$\begin{aligned} P_k &\to U_k D_k U_k^\intercal \\ U_{k+1} &:= U_k \\ D_{k+1} &:= D_k \\ a_0 &:= R \\ f^\intercal &:= A U_k \\ g &:= D_k f \\ \text{for } j := 1, ..., n &\begin{cases} a_j & := a_{j-1} + f_j g_j \\ D_{k+1,j} & := \begin{cases} D_{k,j} & \text{if } a_j = 0 \\ \frac{a_{j-1}}{a_j} D_{k,j} & \text{otherwise} \end{cases} \\ v_j & := g_j \\ \lambda & := \begin{cases} 0 & \text{if } a_{j-1} = 0 \\ -\frac{f_j}{a_{j-1}} & \text{otherwise} \end{cases} \\ \text{for } i := 1, ..., j-1 & \begin{cases} U_{k+1,ij} & := U_{k,ij} + v_i \lambda \\ v_i & := v_i + U_{k,ij} v_j \end{cases} \end{cases} \end{aligned} \tag{17}$$

At the end of algorithm (17), v is the normalized Kalman gain, which, when multiplied by g and the residual from (15), can approximate the K in (16). This allows for the computation of the posterior state and posterior covariance matrix, which concludes the measurement update step.

$$\begin{aligned} x_{k+1}^j &:= x_k^j + v_j g_j \sum_i e_i \\ P_{k+1} &:= U_{k+1} D_{k+1} U_{k+1}^\intercal \end{aligned} \tag{18}$$

The posterior state x_{k+1} that is computed in (18) is sufficient for the compensation of errors in the system's state, since it contains the estimation of the current state deltas. Furthermore, in the final implementation, the posterior error state vector x_{k+1} shall always be immediately used to correct any offsets in the system's position and velocity. This process is shown below in (19). In doing so, the state of the system immediately after the correction will theoretically be error-free. This means that x_{k+1} must also be reset to zero.

$$\begin{aligned}
x_{k+1} &\equiv \begin{bmatrix}\delta p & \delta v & \delta \rho & \delta b_a^b & \delta b_g^b\end{bmatrix}^\intercal \\
p^e &:= p^e + \delta p \\
v^e &:= v^e + \delta v \\
C_b^n &:= \left(I + \begin{bmatrix} 0 & -\delta\rho_z & \delta\rho_y \\ \delta\rho_z & 0 & -\delta\rho_x \\ -\delta\rho_y & \delta\rho_x & 0 \end{bmatrix} \right) C_b^n \\
b_a &:= b_a + \delta b_a^b \\
b_g &:= b_g + \delta b_g^b \\
x_{k+1} &:= \begin{bmatrix}0 & 0 & 0 & 0 & 0\end{bmatrix}
\end{aligned} \tag{19}$$

The next necessary step as illustrated in figure 2.2, is the time propagation step, which transitions the state space model one time step forward. In the context of the discrete model described here, the time step is the time that elapses between two measurements. Again from [2], this step consists of the set of equations shown in (20). Same as with the measurement update step, the details of these equations will not be considered here, rather referring the reader to [2]. Instead, the implementation proposed in [13] will be implemented directly, as will be shown in equations (24) and (25).

$$\begin{aligned}
x_{k+1} &:= \Phi x_{k+1} + G u_k \\
P_{k+1} &:= \Phi P_{k+1} \Phi^\intercal + Q_d \\
k &:= k + 1
\end{aligned} \tag{20}$$

Note that x_{k+1} has been reset to zero in (19). Having compensated the errors, it is feasible to assume that the control input vector u_k will also be zero. Therefore, the step of updating x_{k+1} shown in (20) may be neglected entirely.

The next necessary step is to define the transition matrix Φ for the model. The following transition matrix was given to the author by his diploma thesis supervisor, and will not be analyzed further in this document.

$$\Phi = \begin{bmatrix} I & F_{pv} & F_{p\rho} & F_{pba} & 0 \\ 0 & I & F_{v\rho} & F_{vba} & F_{vbg} \\ 0 & 0 & I & 0 & F_{\rho bg} \\ 0 & 0 & 0 & F_{aa} & 0 \\ 0 & 0 & 0 & 0 & F_{bb} \end{bmatrix} \tag{21}$$

$$
\begin{aligned}
F_{aa} &= I\left[1+\frac{1}{2}\left(\frac{1}{\tau_a}\Delta t\right)^2+\frac{1}{\tau_a}\Delta t\right]\\
F_{gg} &= I\left[1+\frac{1}{2}\left(\frac{1}{\tau_g}\Delta t\right)^2+\frac{1}{\tau_g}\Delta t\right]\\
F_{pv} &= \begin{bmatrix} \frac{1}{R_M+\hat{h}} & 0 & 0\\ 0 & \frac{1}{(R_N+\hat{h})\cos\hat{\phi}} & 0\\ 0 & 0 & -1 \end{bmatrix}\Delta t\\
F_{v\rho} &= -\begin{bmatrix} 0 & -f_z^n & f_y^n\\ f_z^n & 0 & -f_x^n\\ -f_y^n & f_x^n & 0 \end{bmatrix}\Delta t\\
F_{p\rho} &= \frac{1}{2}F_{pv}F_{v\rho}\\
F_{pba} &= \frac{1}{2}F_{pv}F_{vba}\\
F_{vba} &= -\frac{1}{2}F_{aa}(C_b^n\begin{bmatrix}0 & 0 & g_0\end{bmatrix}^\intercal\Delta t) - C_b^n\begin{bmatrix}0 & 0 & g_0\end{bmatrix}^\intercal\Delta t\\
F_{vbg} &= \frac{1}{2}C_b^n\Delta t F_{v\rho}\\
F_{\rho bg} &= \frac{1}{2}F_{gg}C_b^n\Delta t + C_b^n\Delta t
\end{aligned}
\tag{22}
$$

I — 3x3 identity matrix

τ_a — parameterized correlation time for accelerometer biases

τ_g — parameterized correlation time for gyroscope biases

$\hat{h}$ — altitude

$\hat{\phi}$ — latitude

f^n — specific force in the navigation frame

The next step that is needed for Thornton's time update algorithm, is to compute the Q_d matrix from the parameterized Q_0 covariance of the model. This computation can be found in [2].

$$
\begin{aligned}
G &:= \begin{bmatrix} I & 0 & 0 & 0 & 0\\ 0 & -C_b^n & 0 & 0 & 0\\ 0 & 0 & C_b^n & 0 & 0\\ 0 & 0 & 0 & I & 0\\ 0 & 0 & 0 & 0 & I \end{bmatrix}\\
Q_d &:= GQ_0G^\intercal
\end{aligned}
\tag{23}
$$

The above allows then for the computation of the Q_k covariance matrix for the current time step. And by decomposing it, the algorithm is then computed according to [2] as shown below. In the interest of brevity, the reader is referred to [13] for details regarding the meaning and construction of the matrix w.

$$Q_k := \frac{1}{2}\Delta t(\Phi Q_d + Q_d \Phi^\intercal) \tag{24}$$

$$\begin{aligned}
Q_k &\rightarrow U_k D_k U_k^\intercal \\
\text{for } j := 1, ..., n &\begin{cases} D_{k+1,j} & := w_j^{n-j} D_k (w_j^{n-j})^\intercal \\ \text{for } i := 1, ..., j-1 & \begin{cases} U_{k,ij} & := \frac{w_i^{n-j} D_k (w_j^{n-j})^\intercal}{D_{k,j}} \\ w_i^{n-j+1} & := w_i^{n-j} - U_{k,ij} w_j^{n-j} \end{cases} \end{cases} \\
D_{k,1} &:= w_1^{n-1} D_k (w^{n-1})^\intercal \\
P_{k+1} &:= U_k D_k U_k^\intercal
\end{aligned} \tag{25}$$

And with the P_k covariance matrix updated, and the state x_k reset to zero for the new time instance k, one cycle of Kalman filter updates is concluded.

Chapter 3
Implementation

This chapter will recount the entire development of the Mini-INS project. It will be split into four sections, each detailing a distinct part of the work that was done. These sections are also arranged in the appropriate order, such that the logical flow from one to the next matches the order in which these parts were planned out and implemented during development. This arrangement is illustrated in figure 3.1.

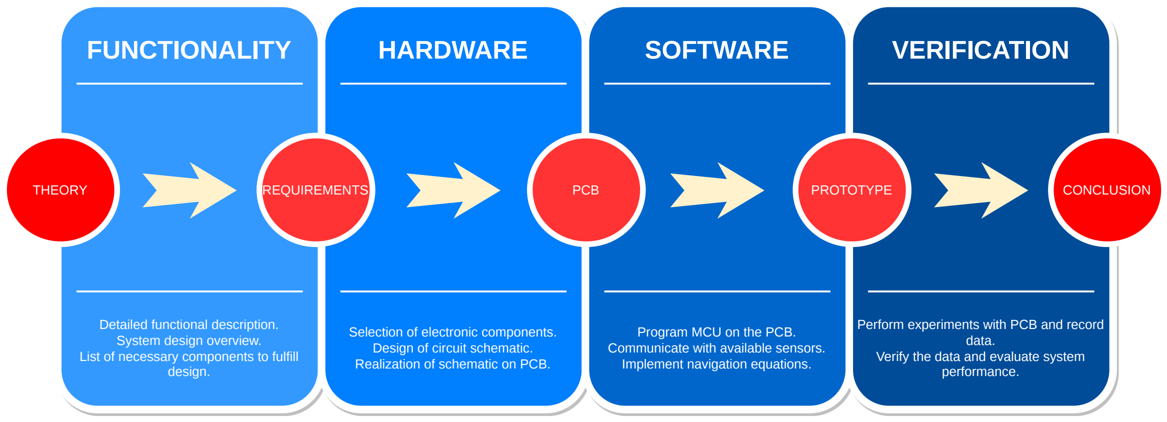

FUNCTIONALITY
HARDWARE
SOFTWARE
VERIFICATION
THEORY
REQUIREMENTS
PCB
PROTOTYPE
CONCLUSION
Detailed functional description.
System design overview.
List of necessary components to fulfill design.
Selection of electronic components.
Design of circuit schematic.
Realization of schematic on PCB.
Program MCU on the PCB.
Communicate with available sensors.
Implement navigation equations.
Perform experiments with PCB and record data.
Verify the data and evaluate system performance.

Figure 3.1: Overview of sections in the Implementation chapter.

3.1 System design

As the previous chapter may have hinted at, this diploma thesis involves the design and realization of an INS on a *Printed Circuit Board* (PCB). The desired function of the completed product can be summarized by the following three key points.

1. Provide *position*, *velocity*, and *attitude* (PVA) information at the output.
2. Maintain an output rate of around $100\ Hz$.
3. Fulfill this functionality with the minimum footprint size and power consumption.

The board incorporates the necessary sensors for obtaining an accurate PVA estimation, as was described in the previous chapter and illustrated in figure 3.2. The first sensor is an *Inertial Measurement Unit* (IMU), featuring an accelerometer and a gyroscope. It will be a *Microelectromechanical System* (MEMS) sensor which fits the requirements for a small size and low power consumption. Complementary to the IMU, a magnetometer will also be included to assist in determining magnetic heading. In addition, a GNSS receiver is placed on the PCB, which will enhance the navigation computations from the IMU by performing regular error and bias corrections through a Kalman filter. Finally, the schematic design and layout will also plan for the existence of a static pressure sensor which could be fused together with the rest to further improve the accuracy of the system for *Unmanned Aerial*

Vehicles (UAV). These aforementioned sensors will provide raw readings of various quantities. However, the centerpiece of the board is a programmable ARM *Microcontroller Unit* (MCU). The firmware of the processor is tasked with collecting the data from all sensors and implementing the navigation equations to compute the PVA state of the system. Furthermore, the same MCU is also responsible for handling the system's digital interface, and communicate the outputs to other nodes that are connected to the PCB.

Figure 3.2 provides a high-level overview of the system design, in the form of a block diagram. It shows the main components that are needed to fullfil the goals of the project, as well as a hint at how they shall communicate with each other. Note that the completed system contains additional secondary components, like power regulators and electrical connectors. However, this first diagram only showcases components that are of operational interest to the system design. This includes processing units, interfaces, and sensors.

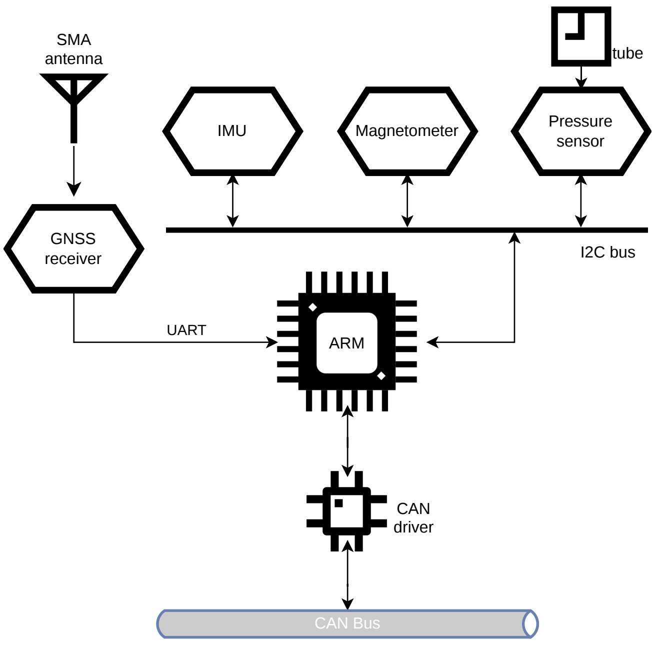


Figure 3.2: High-level system design block diagram.

3.1.1 System requirements

This subsection lists all the components which are necessary to realize the system design. Additionally, it lists components which are not necessarily present in figure 3.2, but which are necessary for the hardware design from an electrical standpoint.

- **Microcontroller**

  A single programmable microcontroller is at the center of the PCB. Its firmware will configure and control all connected sensors, collect their measurements, and compute the navigation algorithms that will convert the sensor data into position, velocity and attitude. Additionally, it will implement the main interface for the PCB.

  The microcontroller family selected should be one that is promoted by the manufacturer as a low-power solution. Additionally, it should also be available at a small-package variant.

- **IMU**
  A MEMS inertial measurement unit shall be used, with, at minimum, a tri-axial accelerometer and tri-axial gyroscope. It has to provide a digital interface, and feature some method of ensuring that sampled data are not lost during brief pauses in communication with the microcontroller. For the sensor, the following is also expected:
  - At least $1\ kHz$ inner sampling rate.

    For the accelerometer:
  - At least $6\ g$ specific force range.

    For the gyroscope:
  - Tactical-grade bias instability up to $5\ ^\circ/h$.
  - At least $200\ ^\circ/s$ angular rate range.
- **Magnetometer**
  A tri-axial magnetometer will most likely be included as a sensor separate from the IMU. It should ideally feature some way of applying hard-iron and soft-iron compensation at the sampling level.
- **GNSS receiver**
  A multi-constellation GNSS receiver. It should be able to power an active GNSS antenna, and provide digital interface communication. Furthermore, it should provide a *timepulse* for the synchronization of measurements, and feature the necessary NMEA messages for determining time, global position, and tri-axial velocity in the navigation frame.
- **Static pressure sensor**
  The static pressure sensor should be able to operate at a pressure range that covers the altitude range from ground level up to a few thousand meters of altitude. This means a range of at least $[0, 15]\ psi$.
- **CAN driver**
  A CAN driver should translate between the low-voltage *RX* and *TX* CAN lines from the microcontroller, and the higher-voltage *CANH* and *CANL* differential lines of the external bus.
- **Power regulator**
  A power regulator IC shall be included, as per standard practice. It will be used to isolate the sensitive components from noise coming from the power supply, as well as ensure that the operating voltage throughout the PCB does not fluctuate.

Following is a list of operational requirements that are expected out of all electronic components, that are to be incorporated into the hardware design.

- **Extended temperature range**
  All components shall be expected to be operable in the *Extended Commercial* temperature range, between $-40\ ^{\circ}C$ and $+85\ ^{\circ}C$.
- **Low power consumption**
  For miniature UAS applications, battery power is one of the main factors that limit the system design. As such, additional care should be exhibited when selecting components, to limit the board's power consumption as much as possible.
- **Small footprint**
  Components should impact the size and layout of the designed PCB as little as possible. This refers not only to the size of the component's package, but also to its overall impact on the board. For example, all components must be available in SMD packages that can be soldered on a single side of the PCB without through-holes.
- **Harsh environment tolerance**
  Components should operate under harsh mechanical conditions, such as intense vibrations and high-g impacts.
- **Vibration-resistant connectors**
  All connectors that are surface-mounted to the PCB shall be vibration resistant. Auxiliary connectors, such as for programming or debugging the microcontroller, are exempt from this requirement.
- **Compatible digital interfaces**
  To avoid enlarging the PCB footprint with unnecessary circuitry, all selected components should implement interfaces compatible with the other components that they may communicate with. For example, if the IMU provides data output on I2C bus, then the selected microcontroller must have an I2C interface.

3.2 Hardware design

3.2.1 Electronic component selection

This section recounts the process of selecting the necessary electronic components for the circuit. The selection was not performed exhaustively, since the focus of this diploma thesis is not to perform market research. Instead, for each component that was needed, the search was limited to a specific established manufacturer which is known to produce high-quality, reliable parts. This process also offers a benefit: if a specific part were to be switched for another during development or production, there is a good chance that the new part would be found from the same manufacturer, and that it would have the same pinout and footprint. With this in mind, the search for each component was done in the following way:

1. Identify suitable manufacturer, either through prior experience or recommendation by the thesis supervisor.
2. In the appropriate category, list all available components by said manufacturer on common vendor websites, like Mouser, DigiKey, Farnell, and so on.
3. Filter for desired digital interfaces.
4. Filter for needed operating temperature range.
5. Filter out components that are either too large, or not available in an SMD package, or not suitable for low-power operations.
6. If multiple parts remain, pick one in cooperation with thesis supervisor.

Manufacturers usually offer multiple options in a family of products that feature identical footprints and interfaces. Because of this, most components are usually interchangeable with no modifications necessary to the schematic, and only minor alterations needed to the firmware. With this in mind, the final choices for components were not necessarily made optimally. Due to the market and supply situation at the time of writing, as well as the inherently limited time in which a diploma thesis has to be realized, most components were chosen simply because they were suitable and available. The final selections are given in table 3.1.

Function	Manufacturer	Component
Power regulator	Texas Instruments	TPS62007DGSR
CAN driver	Texas Instruments	SN65HVD257DR
Microcontroller Unit (MCU)	STMicroelectronics	STM32G431KBT6
Inertial Measurement Unit (IMU)	STMicroelectronics	ISM330DHCXTR
Magnetometer	STMicroelectronics	LIS2MDLTR
Static pressure sensor	Honeywell	SSCMRND015PGSA3
GNSS receiver	u-blox	ZED-F9T-00B

Table 3.1: List of final selected components.

3.2.2 Work done on prototype

In the beginning of the diploma thesis, development was started using a breadboard for prototyping. The *STM32G431* microcontroller to be used, was soldered onto a breadboard-compatible breakout board, along with the necessary capacitors. Additionally, the *X-NUCLEO-IKS02A1* expansion board was purchased from STMicroelectronics, which contains an *ISM330DHCXTR* IMU, and an *II2SMDC* magnetometer. Note that this magnetometer is not the same that was selected for the PCB, but their footprints and functional specifications are identical. Finally, a board with a GNSS receiver was also obtained. To create a portable platform for experimentation, all these components were glued onto a generic power bank.

To interface with the INS, a *Raspberry Pi Zero* device was used. It is powered by the power bank, and uses one of its output $3.3V$ pins to power the INS. Additionally, a single UART line is connected from the microcontroller to the *Raspberry Pi*, on which the microcontroller continuously transmits the system's PVA state, as well as raw measurements for future evaluation. Furthermore, a hat board from *Waveshare* was connected to the *Raspberry Pi*, which features an $1.44\ in$ LCD screen, along with a few buttons. Using this, a crude user interface was developed, with which the user can choose to display on the screen either an attitude indicator, or a heading and velocity indicator, which are based on the data read-out from the microcontroller on the serial interface. The completed prototype is shown in figure 3.3. A screenshot is also included in figure 3.4, showing how both raw measurements and output data from the prototype could be evaluated in *Simulink*.

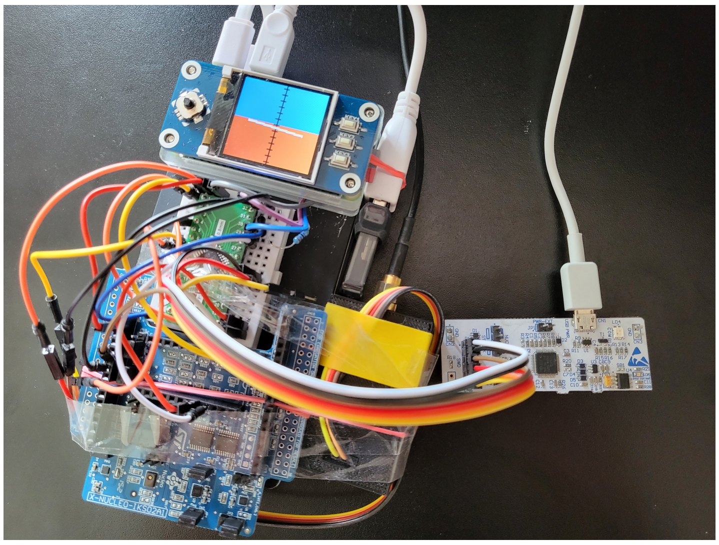


Figure 3.3: Portable prototype of INS.

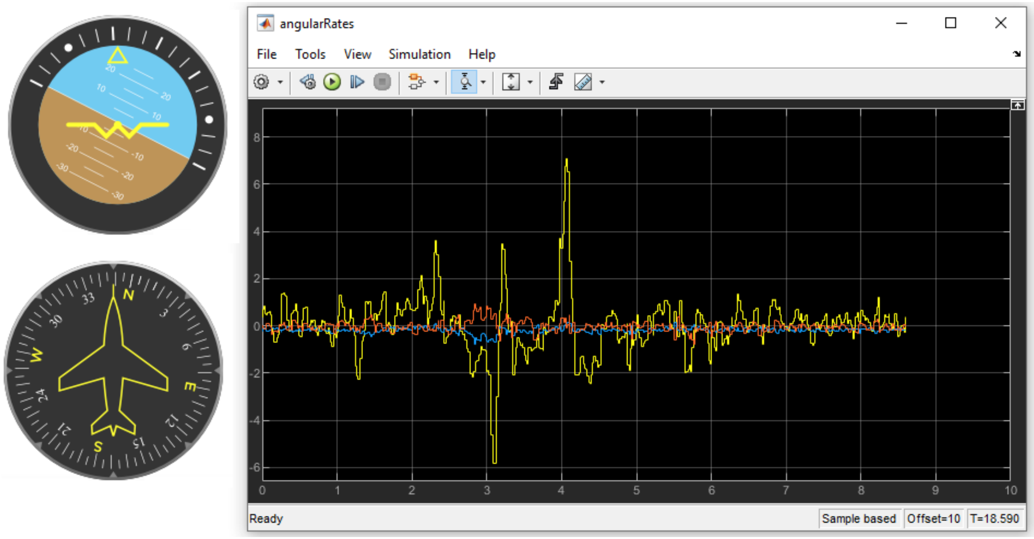


Figure 3.4: Screenshot of data evaluation in Simulink.

3.2.3 Realization of schematic

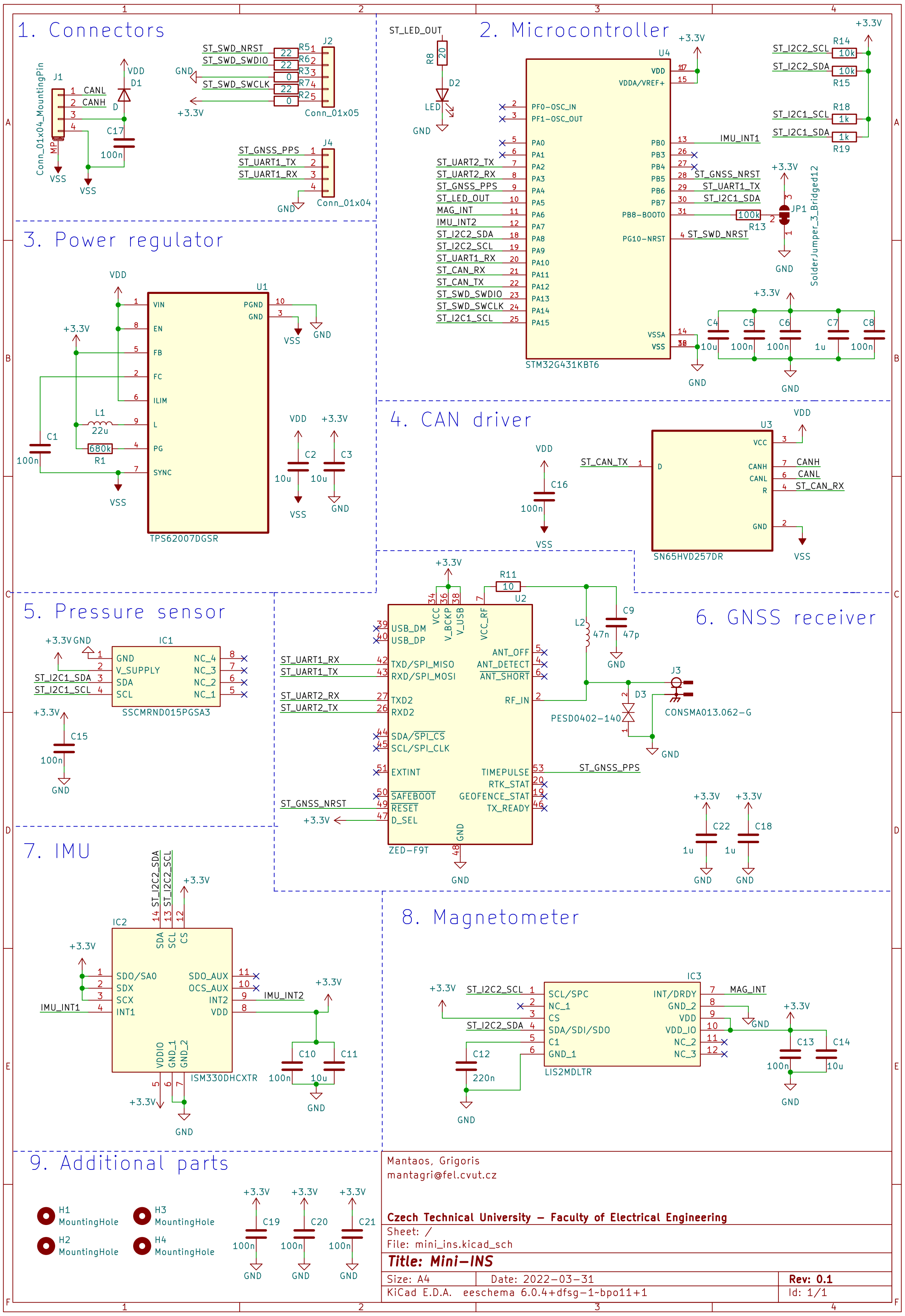


Figure 3.5: Circuit schematic in KiCad.

Following the selection of components, the next step is to realize the schematic shown in 3.2. The schematic was drawn in KiCad, which is *Free and Open Source Software*

(FOSS). The completed version is shown in figure 3.5. It is segmented into logical sections, each containing one of the active components. Descriptions of each section, along with explanations for the passive components used, are given below.

1. **Connectors**

   This schematic section contains the connectors through which the board will interface.

   - **J1:** This is the main power and I/O connector. It is placed on the PCB as a *Molex PicoBlade*, on which the user can connect both the VDD power, and the differential CAN bus lines. It is complemented with a reverse polarity protection diode, and a noise-filtering capacitor which should be placed as close to the connector as possible.
   - **J2:** This connector is used for programming and debugging the microcontroller. It connects to the necessary pins for *Serial Wire Debug* (SWD), so that the microcontroller on the board can be programmed directly using any ST-Link adapter. It will be placed on the board as a standard $2.54\ mm$ pin header.
   - **J4:** This is an auxiliary connector and currently serves no purpose in normal operation. It connects via UART both to the microcontroller, and the GNSS receiver. It is placed in order to future-proof the design, in case a serial I/O would ever be needed. For example, to configure the GNSS receiver externally using *u-blox* software, or to read out raw data from the microcontroller. It will be placed on the board as a non-standard $1.00\ mm$ pin header.

2. **Microcontroller**

   This schematic section contains the microcontroller, and all of the required passive components for its operation.

   - Capacitors are included as per the manufacturer's recommendations, which is one big filtering capacitor for the digital power supply, one big filtering capacitor for the analog power supply, and one small bypass capacitor for each VDD input on the package. The LQFP-32 variant contains three VDD pins, so that's a total of five capacitors.
   - Input interrupt pins ST_GNNS_PPS, MAG_INT, IMU_INT1, and IMU_INT2 are all connected to GPIO with a different number. This is because on this STM32 controller, external interrupts cannot be configured to trigger multiple ports on the same number.
   - A solder bridge JP1 is included, which pulls the BOOT0 pin either to high or low. This pin is used to select the memory area that the firmware will boot from. Leaving it externally controlled through the solder bridge was decided to be preferable, for flexibility.
   - An auxiliary *Light-Emitting Diode* (LED) is connected to an arbitrary GPIO pin on the microcontroller. This will be used to visually communicate the state of the board.
   - The microcontroller is connected to two different I2C buses. That is because the IMU and Magnetometer support a maximum clock rate of $1\ MHz$, while the

pressure sensor only supports up to $400\ kHz$. So they were placed on separate buses, to avoid the small delays that would be associated with resetting and reconfiguring the I2C peripheral between transmissions.

3. **Power regulator**
   The step-down DC-DC converter that was selected, should always output a constant $3.3V$ supply voltage. It is placed in such a way, so as to isolate the components on the board from the external power supply and ground. Bypass capacitors C2 and C3 are placed as per standard practice, one for each potential side. The rest of the passive elements are placed according to the manufacturer's recommendations.
4. **CAN driver**
   This IC should translate between the internal voltage ST_CAN_TX and ST_CAN_RX lines, and the external differential bus CANH and CANL lines. A single bypass capacitor C16 is placed next to this component as per the manufacturer's recommendations.
5. **Pressure sensor**
   The pressure sensor is connected to the I2C1 bus, which according to the part's datasheet is connected to $1\ k\Omega$ pull-up resistors. Also, a bypass capacitor is placed next to its power pin as usual.
6. **GNSS receiver**
   The GNSS receiver is connected to the microcontroller with two different pairs of UART RX and TX lines, one of which is also led out through J4 for future purposes. The nRESET pin is also led to a GPIO on the microcontroller, allowing for the firmware to reset the GNSS receiver if needed.

   The antenna is connected through an SMA connector, with passive elements, according to the manufacturer's recommendations. Additionally, a protection fuse D3 is included, which is meant to protect against *Electrostatic Discharge* (ESD).
7. **IMU**
   The IMU is connected to the I2C2 bus, and its two interrupt output pins are lead to the microcontroller in case they are needed for data handling. Additionally, as per the manufacturer's recommendation, one large filtering capacitor and one small bypass capacitor are included, which should be placed next to the voltage supply pins.
8. **Magnetometer**
   The Magnetometer is also connected to the I2C2 bus, and its interrupt output pin is lead to the microcontroller in case it is needed for data handling. Then, as per the manufacturer's recommendation, one large filtering capacitor and one small bypass capacitor are included, which should be placed next to the voltage supply pins. Finally, again following the datasheet, a small capacitor is also placed between pins 5 and 6.
9. **Additional parts**
   This schematic section is left for additional parts that are useful for the PCB.

- First are the mounting holes, which are simple through-holes meant for mounting screws.
- And second are the three additional noise-filtering capacitors, which should be placed at various positions throughout the PCB. These should ideally be spread out, and placed in areas that lack a nearby capacitor.

3.2.4 Design of Printed Circuit Board layout

This subsection summarily describes the design process of the PCB layout. Like the schematic, the PCB was also drawn in KiCad. In the interest of brevity, not every component placement, track, and via, is rigorously analyzed in this document. Instead, this subsection focuses only on providing a high-level overview of the PCB layout, along with the thought process that lead to the board's shape and size. The first decision is the size and outline of the PCB, and it is described below.

1. To reduce the board's size, both sides of the PCB will be used to place components.
2. All connectors, or components that have connectors, should be placed on the front side. This includes the Molex connector J1, the SMA antenna connector, and the pressure sensor since it bears a barbed port for a tube.
3. The GNSS receiver and the SMA antenna connector must be on the same plane, because the trace from the antenna to RF_IN should not be routed through a via. Therefore, the GNSS receiver should also be placed on the front side.
4. These four aforementioned components, together with the four mounting holes in the corners, impose the minimum width and height for the board. After laying them out, the final outline was obtained.

It was decided that the PCB shall consist of four layers. As is common practice in this case, the two inner layers are to be filled with copper to create power areas for short routing to either supply voltage or the ground. Specifically, the back-inner filled layer is connected to the supply voltage and referred to as the *power plane*, while the front-inner filled layer is connected to the ground and referred to as the *ground plane*. The next step in the design process is to draw the layout of copper-filled zones, while planning for the components that are to be placed on them. The resulting segmentation is shown in figure 3.6 and described below.

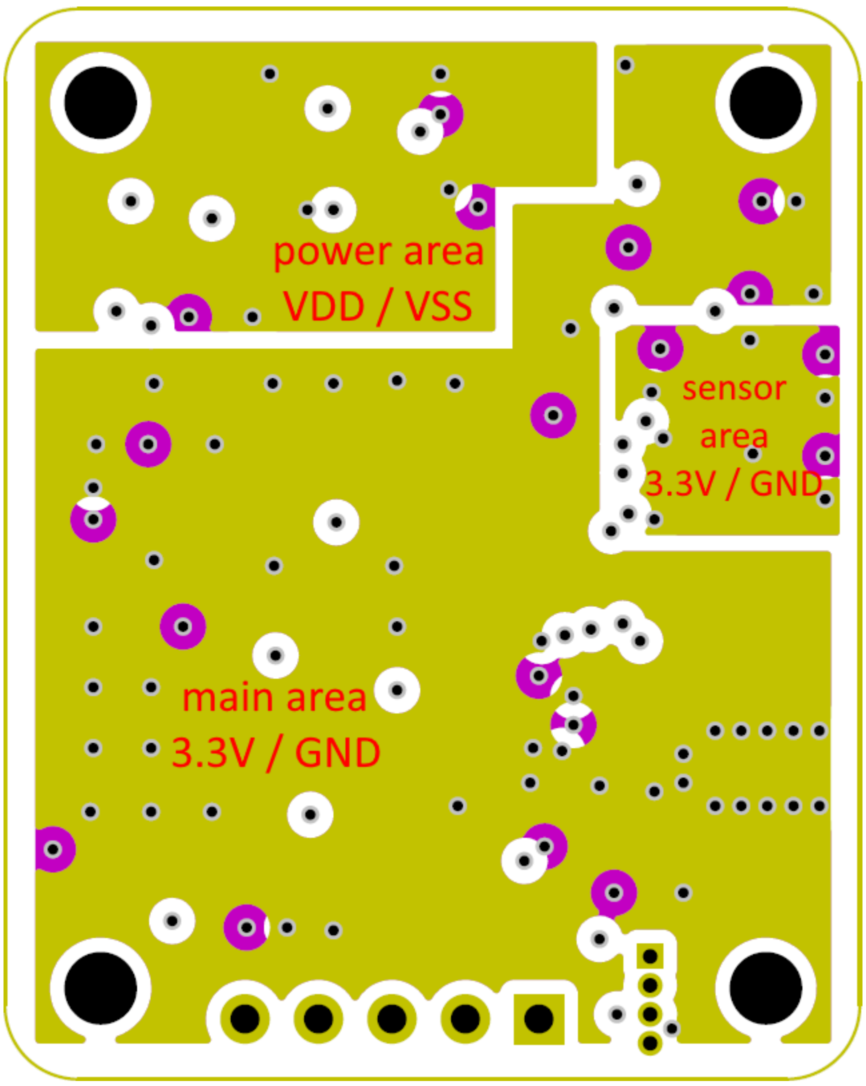


Figure 3.6: Copper-filled area segmentation on the PCB.

- **Power area**
  The first area that is needed, is the one that will encompass the J1 connector as well as the power regulator. This area is filled with the external power supply and ground. Conveniently, the CAN driver can also be placed on this area on the bottom side, since it also needs to be supplied from VDD and be close to the external CAN bus.

- **Main area**
  This area is filled on the *power plane* with the $3.3V$ operating voltage of all other components on the board, as well as the internal GND on the *ground plane*.

- **Sensor area**
  This area is also filled with $3.3V$ and GND but is separated from the main area. The motivation for this is that two of the critical microelectronic sensors, the IMU and the magnetometer, should be separated from noise as much as possible. Therefore, it is considered beneficial to place them on their own isolated area, where they are protected from noise that may be emitted from the microcontroller or other high-frequency signal lines.

And with the aforementioned broad guidelines in mind, all components can be placed and connected on the PCB. The first component which must be incorporated into the design is the SMA antenna connector, which has to be placed as close as possible to the GNSS receiver's RF_IN pin. Importantly, this connection must be made as directly as possible, with curved tracks if a slight turn is required. The path of this track should be, to the extend possible, surrounded by vias to the ground plane in order to protect it from noise. Finally, the width of this track should be precisely calculated to have an effective impedance as close as possible to $50\ \Omega$. This value is chosen to, ideally, match the impedance of the antenna. To compute the desired track width, a formula is given in [15]. According to the PCB manufacturer, the track thickness will be $70\ \mu m$, the isolation between the top layer and the ground plane will be $75\ \mu m$, and the material will be FR4 which has a dielectric constant of 4.3. This means that the antenna needs to be routed using a track which is $120\ \mu m$ wide. In the final design the length of this route came out to be $9.4\ mm$, which is decently short and should adequately protect the GNSS chip.

Moving on, a common good practice is to first connect all *power* pins, meaning those that carry significant current either in or out of an IC. Following that, all lines carrying high frequency signals should be connected, minimizing the length and corners on their tracks. The completed design is illustrated in figures 3.7 and 3.8, each showing the front and back layers of the PCB respectively.

KiCad is also capable of rendering beautiful 3D models of the PCB. These are also included here in figures 3.9 and 3.10, from the perspective of the front and back sides respectively. Note that the 3D models of the components are not indicative of the real parts, and are generic models included only for illustration. Another thing to note regarding the bottom side of the 3D render, is that the bottom pins of the SMA antenna connector appear hidden. That is because the 3D renderer used currently does not allow for the adjustment of the PCB thickness, which results to the bottom pins of the connector clipping inside the PCB's body.

The final PCB has dimensions $30\ mm \times 37\ mm$.

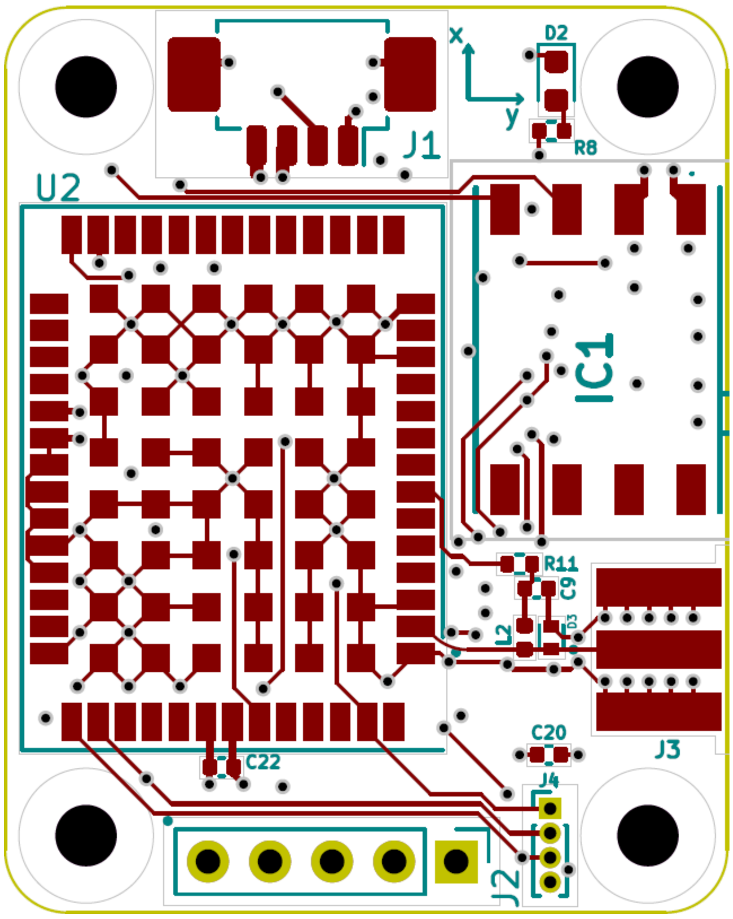


Figure 3.7: Layout of the front layer of the PCB.

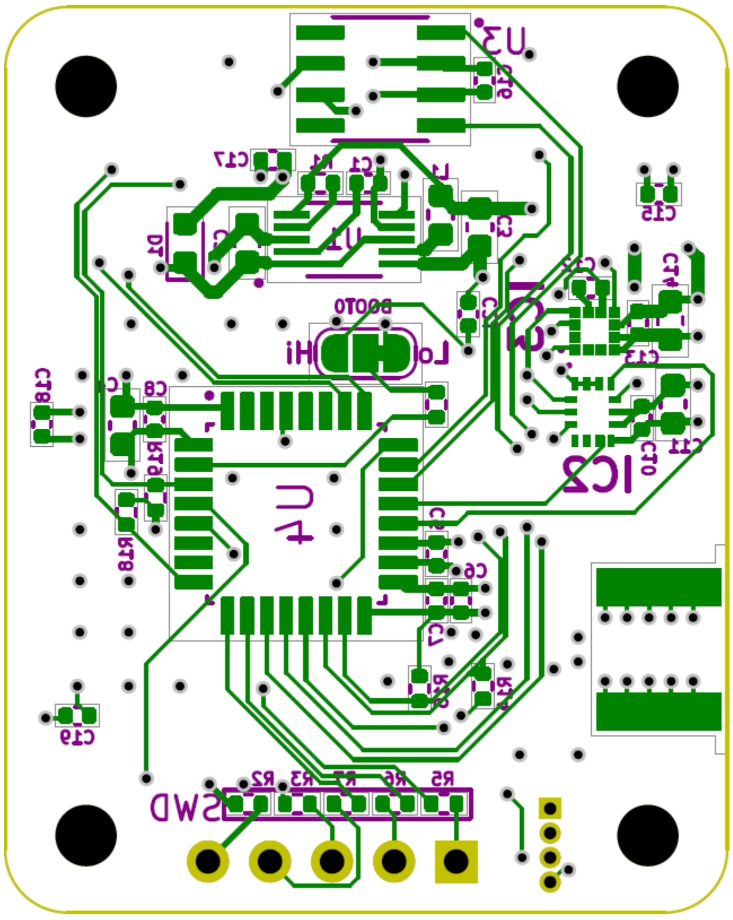


Figure 3.8: Layout of the back layer of the PCB.

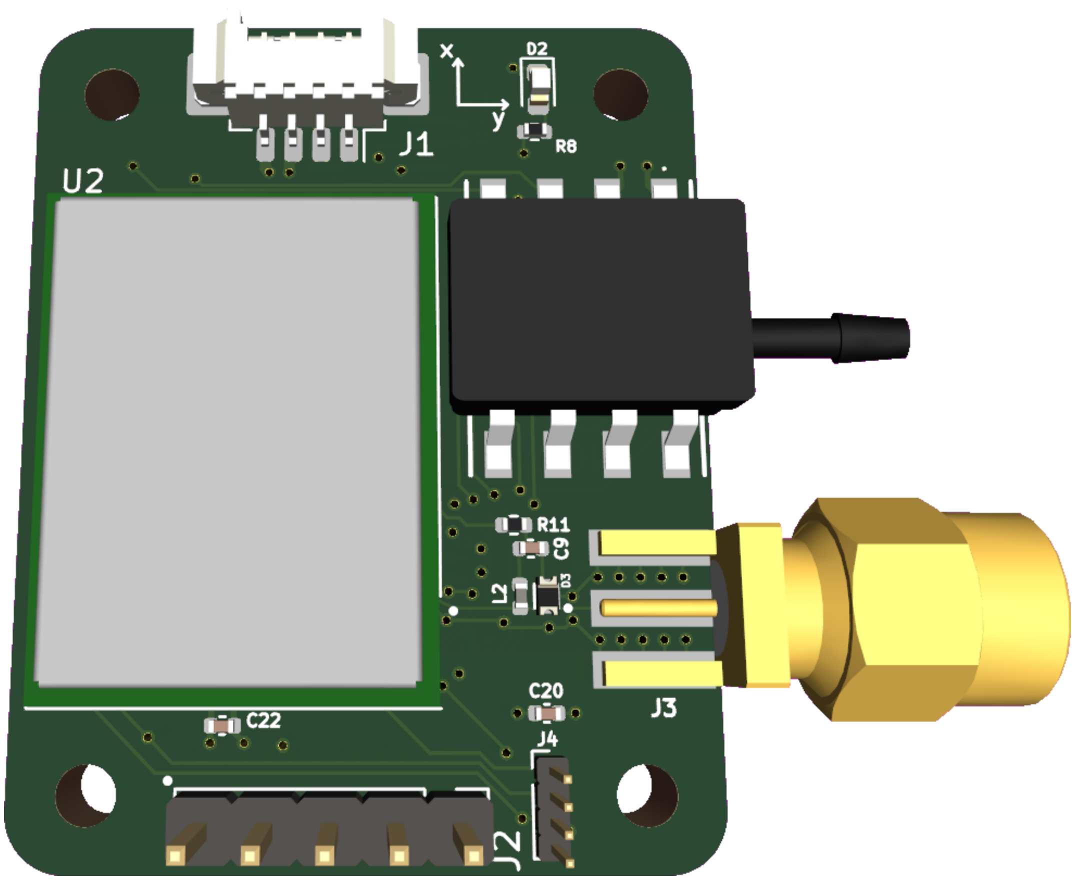


Figure 3.9: 3D rendering of the front side of the PCB.

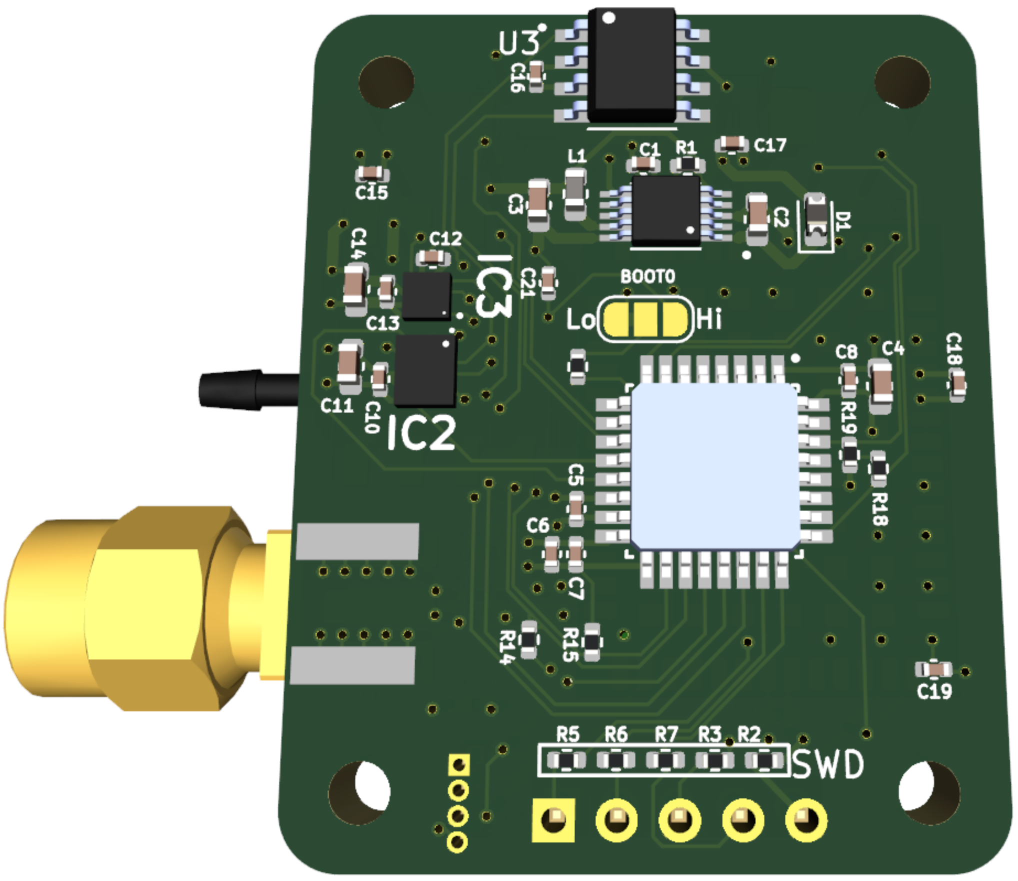


Figure 3.10: 3D rendering of the back side of the PCB.

3.3 Firmware

This section contains a high-level overview of the software that was developed for the Mini-INS. Specifically, this refers to the firmware for the STM32 microcontroller. Since this is an embedded ARM device, the software was written in the C programming language. This document does not thoroughly analyze the source code in its entirety, since the project is quite large. Instead, the focal point of this section is to familiarize the reader with the structure of the software, as well as the basic design principles that were employed during development. Additionally, it will recount how the functional requirements of the project were met, which includes the communication with sensors and implementation of navigation equations. As such, a good chunk of this section will consist of diagrams in the *Unified Modeling Language* (UML), which is standard practice for software development.

For reference, the following list defines how variables and data structures are handled in the firmware.

- Struct fields used to hold sensor measurements are typed to match the data transmitted by the sensor. Typically these are `int16_t`, holding the value in two's complement to the sensor's full scale.
- Variables holding *latitude* or *longitude* are stored as 64-bit `double` types, in order to store global positions with precision of five decimal seconds.
- All other floating-point variables are stored as 32-bit `float` types.
- Timestamps are stored as 64-bit `uint64_t`, in order to have millisecond precision to UTC.
- Some integers used as indices in short for-loops are stored as either `int8_t` or `uint8_t`.
- All other integer types are stored as 32-bit `uint32_t`.

3.3.1 Development and build environment

This subsection provides a brief look into the tools that were utilized for the realization of the firmware. For the editing of the source code itself, *Visual Studio Code* was used, which is *Free and Open Source Software* (FOSS). The tool itself is not considered an *Integrated Development Environment* (IDE), since at its core it is merely a text editor. However, through the installation of various extensions, it becomes quite a powerful tool for the development of software. Some of the key extensions used are *C/C++ Extension Pack*, *stm32-for-vscode*, and *Cortex-Debug*. With these installed, it becomes possible to obtain accurate auto-completion, type inference, and even debug the target device directly from the editor.

The project structure is the typical one for a Makefile C project. The top-level `inc/` directory contains all of the `.h` header files, while the `src/` directory contains all of the `.c` source files. Certain staple files were also taken from the manufacturer's *Board*

Support Package (BSP), these contain header files with the register definitions of the `stm32g431` processor, the header files for the Cortex-M4 peripherals, the header files for the ARMv7 architecture, the assembly startup file `startup.s`, and finally the linker script file `LinkedScript.ld`. The whole project is then compiled with the help of a Makefile, which invokes the `arm-none-eabi-gcc` compiler. This compiler is a part of the *GNU Arm Embedded Toolchain* and is available online by *Arm Ltd*. Specifically, version `10.3-2021.10` of the toolchain is used.

By default, the Makefile is configured to compile a build for debugging. This, apart from including debug symbols in the output, sets the optimization level to `-Og`, which enables all compiler optimizations that do not interfere with debugging. To produce a release build, the `DEBUG` variable should be set to `0`, and the compiler optimization to `-O3`.

```
# makes a debug build with -Og
make
```

```
# makes a release build with -O3
make DEBUG=0 OPT=-O3
```

3.3.2 Data handling and synchronization

The management and synchronization of the data from the various sensors, as well as on the CAN bus output, are the main challenges of this project. The objective of the Mini-INS board is to provide PVA data on the CAN bus at a relatively high rate (e.g $100\ Hz$). This output data shall be in the form of NMEA sentences, whose UTC timestamps must accurately define the exact time at which the data in the message were valid at.

The sensors are configured for high performance and high rate inner sampling, which will typically be about $1000\ Hz$. In this small microcontroller, there isn't sufficient computational budget to compute the mechanization of the system at such a high rate. Even if there was, it would be unnecessary to do so, since the measurements themselves would likely be quite noisy at that rate. Instead, a simple decimation filter will be used which will, for instance, average over every 10 or so samples that are received. This brings the rate of mechanization down to about $100\ Hz$, which coincides with the output rate.

Finally, asynchronously to the inertial sensor sampling and mechanization, the error correction step with the Kalman filter shall be executed whenever new data are available from the GNSS receiver. A diagram which summarizes the data flow through the application is given in figure 3.11.

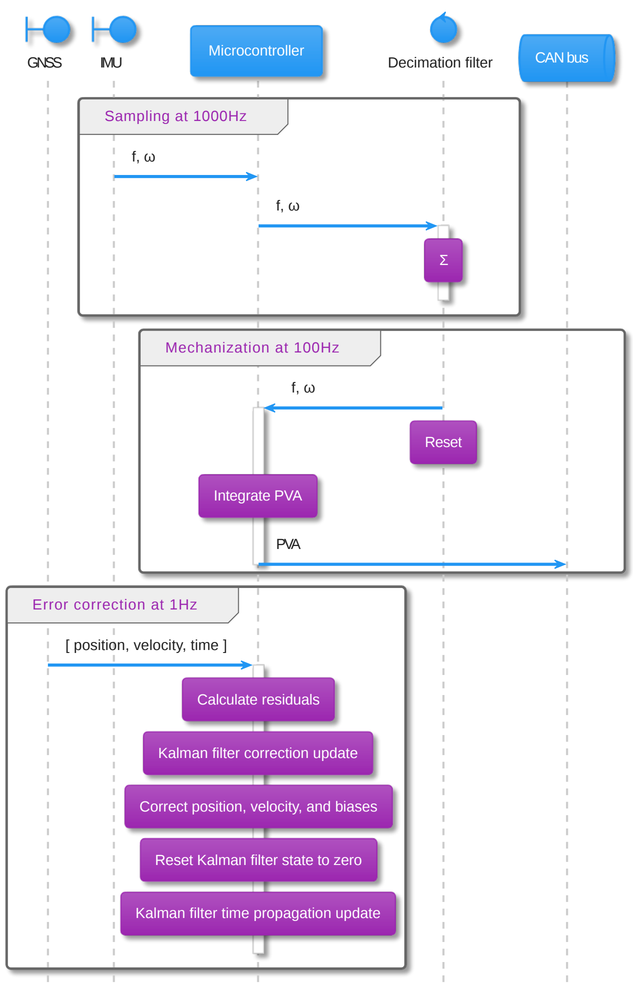

GNSS
IMU
Microcontroller
Decimation filter
CAN bus
Sampling at 1000Hz
f, ω
f, ω
Σ
Mechanization at 100Hz
f, ω
Reset
Integrate PVA
PVA
Error correction at 1Hz
[position, velocity, time]
Calculate residuals
Kalman filter correction update
Correct position, velocity, and biases
Reset Kalman filter state to zero
Kalman filter time propagation update

Figure 3.11: Sequence diagram of data handling and computations.

At a rate of $1\ Hz$, as data is received from the GNSS receiver, the state of the whole navigation model is to be updated using a linear Kalman filter. However, the intention is to output data on the CAN bus at a much higher rate, therefore the more frequent readings from the IMU sensor will be used to update the state at the output rate. This creates the following two considerations.

- The data read from the GNSS receiver come at a delay in relation to their validity due to the serial transmission to the microcontroller or other processing done by the GNSS chip. For the synchronization of this, a timepulse is given out by the GNSS receiver that indicates the point in time at which the next data to be received will be valid for. Therefore, the processing done at the Kalman filter should take into account not the IMU data at the time of reception, but the data at the time of the timepulse. For this purpose, the timepulse output of the receiver shall be wired to trigger a hardware interrupt at its rising edge on the microcontroller, which will begin to store IMU data at the right time, until they are needed by the Kalman filter.
- Between the timepulse and the complete reception of the GNSS data, the system should continue updating the navigation state and transmitting it on the CAN bus. However, when the Kalman filter is evaluated, an older mechanization state will be

used. This makes the model output invalid for the current time and not suitable for transmission to the output, since the preceding transmission will have had a *newer* state. To compensate for this, all the IMU data sampled between the timepulse and the Kalman filter update needs to be stored in a buffer, so that the mechanization state can be extrapolated to the current time.

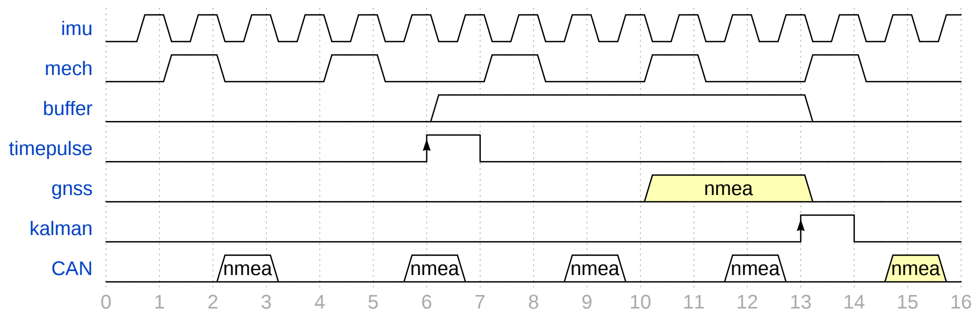


Figure 3.12: Timing diagram illustrating the data synchronization.

The timing of the navigation system can be seen on the diagram in figure 3.12. Below is a brief description of specific time instances in the diagram. Note that these labels do not represent real time units, and the duration of pulses is arbitrary, since the diagram is only for illustration.

1. The `imu` data is sampled from the sensor at an arbitrary high rate.
2. After decimating to a lower rate, the mechanization state `mech` is updated.
3. After every mechanization update, the new PVA is transmitted on the `CAN` bus.
4. At time `6` a hardware interrupt is triggered from the timepulse.
   - The extrapolation buffer is initialized and enabled.
   - The latest valid PVA state is stored as a starting point in the buffer
   - While the buffer is enabled, IMU data will be stored there at every mechanization update (times `8` and `11`).
5. At time `13`, the complete GNSS data is received.
   - Kalman filter is updated using the new GNSS data, and mechanization state from time `6`.
   - The output mechanization state from the Kalman filter is extrapolated to time `13` using the data stored in the buffer (which represent the mechanization state at times `8` and `11`).

3.3.3 Synchronization of timestamps

The position, velocity, and attitude information that is written to the system's output will typically include a timestamp, denoting the exact time at which the data in the message were valid. The *Mini-INS* board will not feature its own RTC functionality, since time-keeping is not its primary purpose. Instead, timestamps will be synchronized using the time obtained from the GNSS NMEA sentences.

However, there needs to be some time extrapolation, since the GNSS receiver will typically only output messages at $1\ Hz$ and the *Mini-INS* should output data at a rate of at least $100\ Hz$. At this rate, a timestamp precision in the order of magnitude of milliseconds shall be sufficient. The idea is to use one the microcontroller's embedded timers to keep track of the time elapsed since the last GNSS timepulse. Then, when constructing an output NMEA sentence for transmission, the timer's counter value (in milliseconds) can be added to the last received timestamp, which creates the timestamp of the sentence.

Finally, an additional $offset$ variable is needed. That is because there is a delay between the timepulse, and when the microcontroller finishes receiving the NMEA sentence. During this delay, it is still necessary to know the last received timestamp, since it is still needed to construct output messages. Therefore, since the timer has to be reset precisely at the rising edge of every timepulse, the microcontroller needs to store the counter's value before it resets it, in order to maintain the ability to calculate the time elapsed since the last known timestamp.

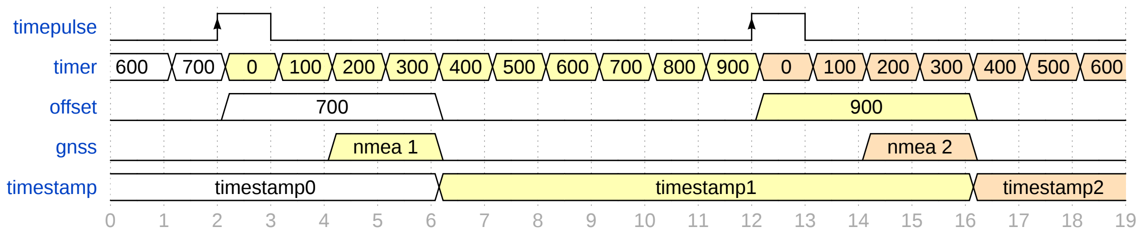


Figure 3.13: Timing diagram illustrating the computation of timestamps.

The timestamp synchronization scheme can be seen on the diagram in figure 3.13. The values shown in the $timer$ and $offset$ variables are in milliseconds for simplicity. With this implemented, the current timestamp in milliseconds can be calculated at any time using the following simple formula.

$$timestamp + timer + offset\ [ms]$$

3.3.4 Software architecture

Working with software for embedded devices, there tend to be significant limitations in terms of program memory and execution time. As such, the software architecture should be considered carefully, to ensure that the source code is small, efficient, and maintainable. While with conventional development, garbage collection and pointer sharing might be an afterthought, in embedded development irresponsible design can lead to a slew of heisenbugs [17] and undefined behaviors that are backbreaking to solve. Therefore, the software design here employs a simple, hierarchical approach. Every module of the application shall be statically allocated at boot, and maintain ownership of variables and other data structures.

An abridged version of the final software architecture is given in figure 3.14, in the form of a UML class diagram. Note that, for brevity, it does not include every single variable and method, but only enough to give the reader a good overview of the structure of the application.

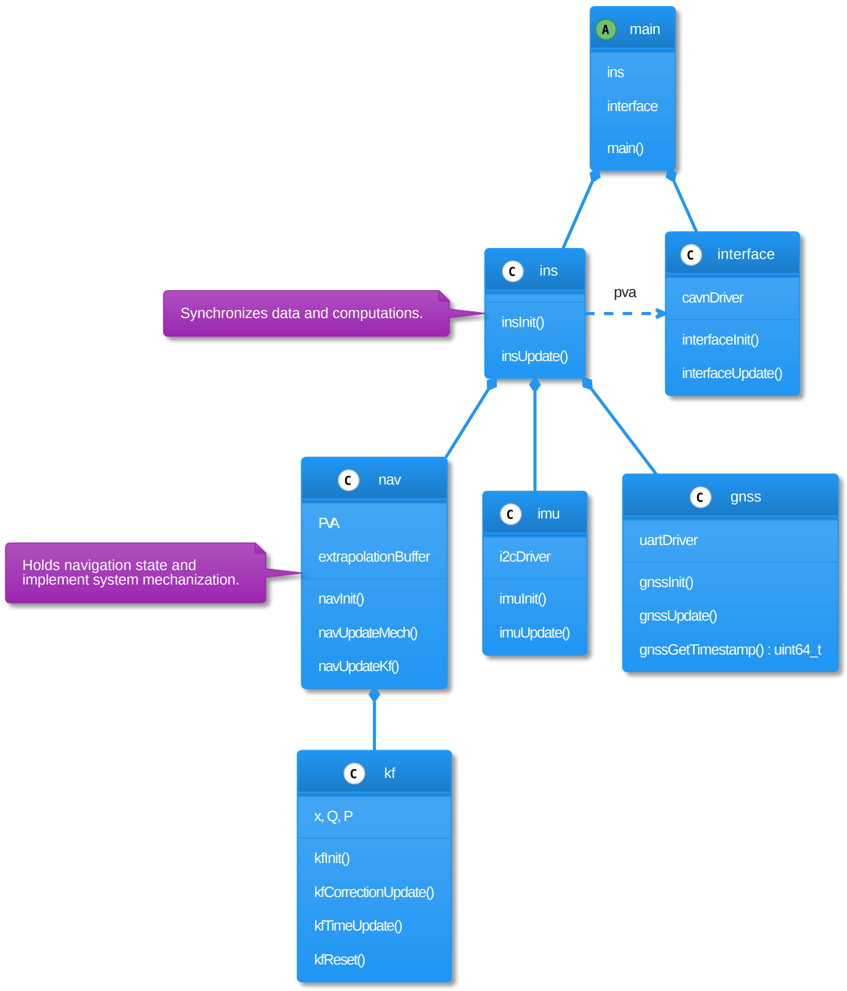


Figure 3.14: Abridged class diagram, illustrating the software architecture.

3.3.5 Software behavior

Much like the architecture, when developing embedded software it is critical to define a robust behavior pattern. In this case especially, it is necessary for the software to interact with the hardware. The latter usually carries out its tasks independently of the former, which gives rise to synchronization issues during the interoperability of the two. This is usually in the form of hardware timers interrupting the software, *Direct Memory Access* (DMA) peripherals, a signal from an external device causing an interrupt, and so on.

Therefore, it is necessary to plan the software's execution with the necessary hardware synchronization in mind. For robustness and predictability, the whole software shall be designed for synchronous execution. This means that in the final design, there will exist only one linear firmware cycle, which shall not be preempted by hardware interrupts. A simple diagram illustrating this principle is given in figure 3.15, which shows the basic execution flow in the *Initialization* and *Run* states. In the same figure, the execution of the `insUpdate()` method is also shown. Designing the software this way also makes it robust against race conditions, as well as easier to read and maintain. That is because the reader may rest assured that all lines of code will be executed sequentially, and without other parts of the codebase altering the data that is used.

One caveat is that, since everything is handled sequentially in the main firmware cycle, its execution time should be as short as possible. This is to ensure that the software responds timely to signals raised by the hardware or other events. For instance, since the method `imuUpdate()` is only called once per cycle, regardless of the status of the sensor, new measurements may only be obtained once per cycle. Naturally, if the cycle is then too long, measurements could be lost.

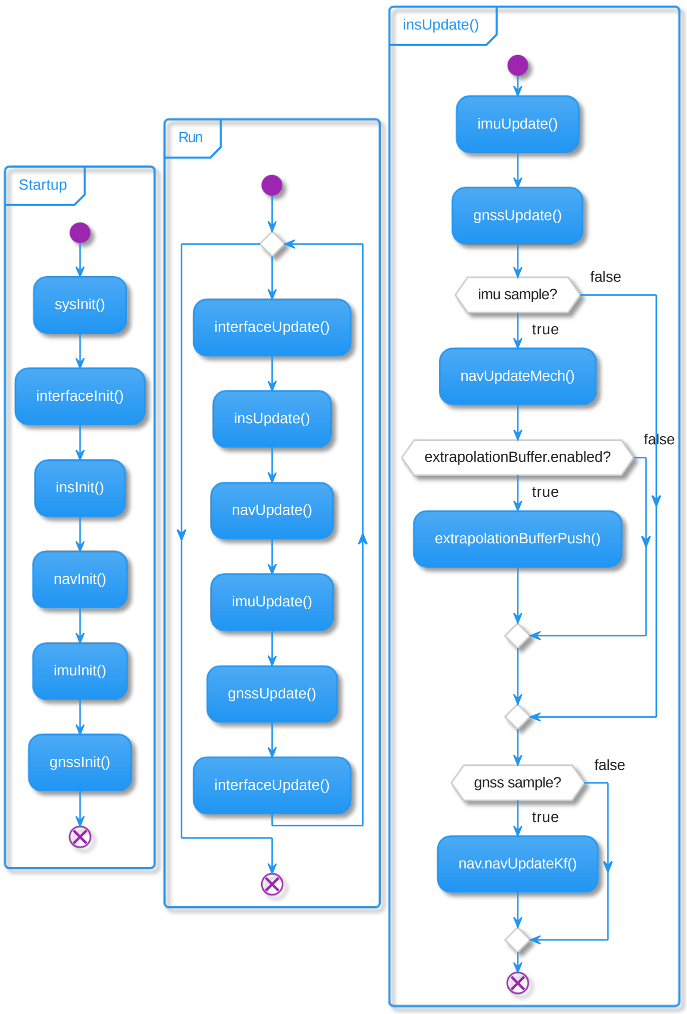


Figure 3.15: Activity diagram of module-level software execution flow.

The next important part is the `imuUpdate()` method, whose execution is shown in figure 3.16. This method is responsible for communicating with the IMU and magnetometer, as well as the application of a decimation filter over the received data. Both of these components, as is common practice with digital sensors, define registers in their data sheets on which configurations and raw data can be accessed. On the I2C bus on which these sensors are connected, the microcontroller is the master and the two sensors are slave nodes. To access any particular register on a sensor, first the sensor's slave address must be transmitted on the bus, followed by the address of the register and a one-bit flag specifying whether the register should be read from or written to.

It is of critical importance to the application that not a single sampled measurement is lost. This can easily occur if the microcontroller does not read out the sensor's data registers before a new measurement is made on the device, since the new measurement will override the older one. To get around this, the manufacturer of the IMU offers two options: one is to send a hardware interrupt to the microcontroller whenever a new measurement is ready, and other is to internally store measurements in a *First In First Out* (FIFO) buffer. Since the development is following a synchronous design scheme, the latter option is chosen. Therefore, in each firmware cycle, the microcontroller will poll the IMU for its FIFO status, which should return the amount of data in the queue, and then proceed to request items from it one at a time. Reading out the entire buffer on every cycle is problematic for two reasons: the first is that an inappropriate number of measurements may be added to the decimation filter, and the second is that just spending time to receive the data over I2C may cause the application to not meet its computation deadlines. Therefore, a parameter is defined called `FIFO_MAX_BATCH_SIZE`, which limits how many items from the FIFO queue should be processed in a single firmware cycle. This parameter should satisfy two criteria:

- The batch size should be the maximum possible, while ensuring that temporal constraints are not violated. In other words, that the firmware cycle does not become too long.
- The batch size should be the minimum possible, while ensuring that the FIFO queue never overflows. For reference, the buffer on the *ISM330DHCX* sensor is about $3\ KB$ in size.

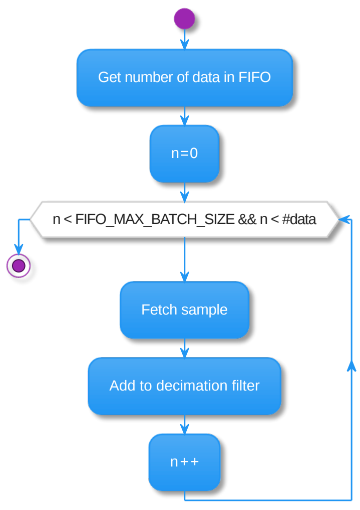

Get number of data in FIFO
n=0
n < FIFO_MAX_BATCH_SIZE && n < #data
Fetch sample
Add to decimation filter
n++

Figure 3.16: Activity diagram of imuUpdate() method.

The last module to go over, is the driver responsible for communicating with the GNSS receiver. While important, its implementation is, in practice, much simpler than the one for the IMU. The microcontroller and the GNSS receiver are wired to communicate via two

UART serial lines. Transmissions over UART are asynchronous and do not require any arrangements between nodes, which makes *Direct Memory Access* (DMA) a great way for handle them. DMA refers to a hardware peripheral offered on STM32 devices, which can autonomously transfer data between the memory and communication peripherals, without the need for intervention by the software. In this case, the DMA will be configured to automatically copy any data that are received on the UART RX line, and store them at a specific buffer in memory. This serial transmission and direct copy to memory will take place in the *background*, freeing up precious execution time for the software.

An important thing to handle here, as was described at a previous section, is the hardware timepulse interrupt that will periodically be sent from the GNSS receiver. The interrupt handler, along the lines of generally accepted good practices, should only perform a minimal number of operations and exit. One action that needs to happen, precisely at the time of the interrupt, is the resetting of the timer which counts the time elapsed since the last timepulse. Another thing is the starting of the DMA transfer to ensure that nothing of what is transmitted after the timepulse is lost by the microcontroller. The scheme of what was described here, is shown in an activity diagram in figure 3.17.

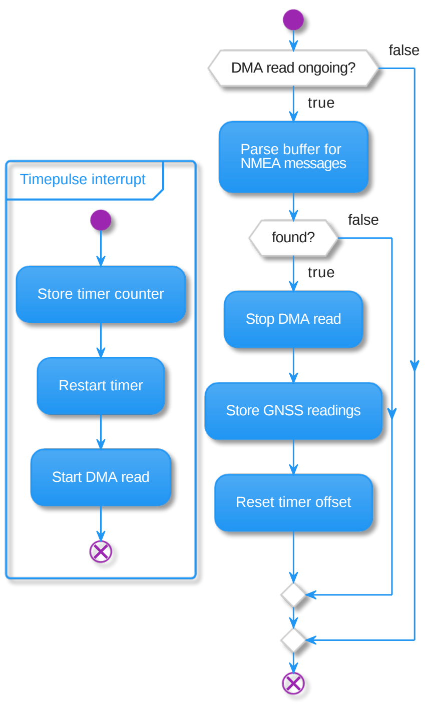


Figure 3.17: Activity diagram of gnssUpdate() method.

3.3.6 System initialization

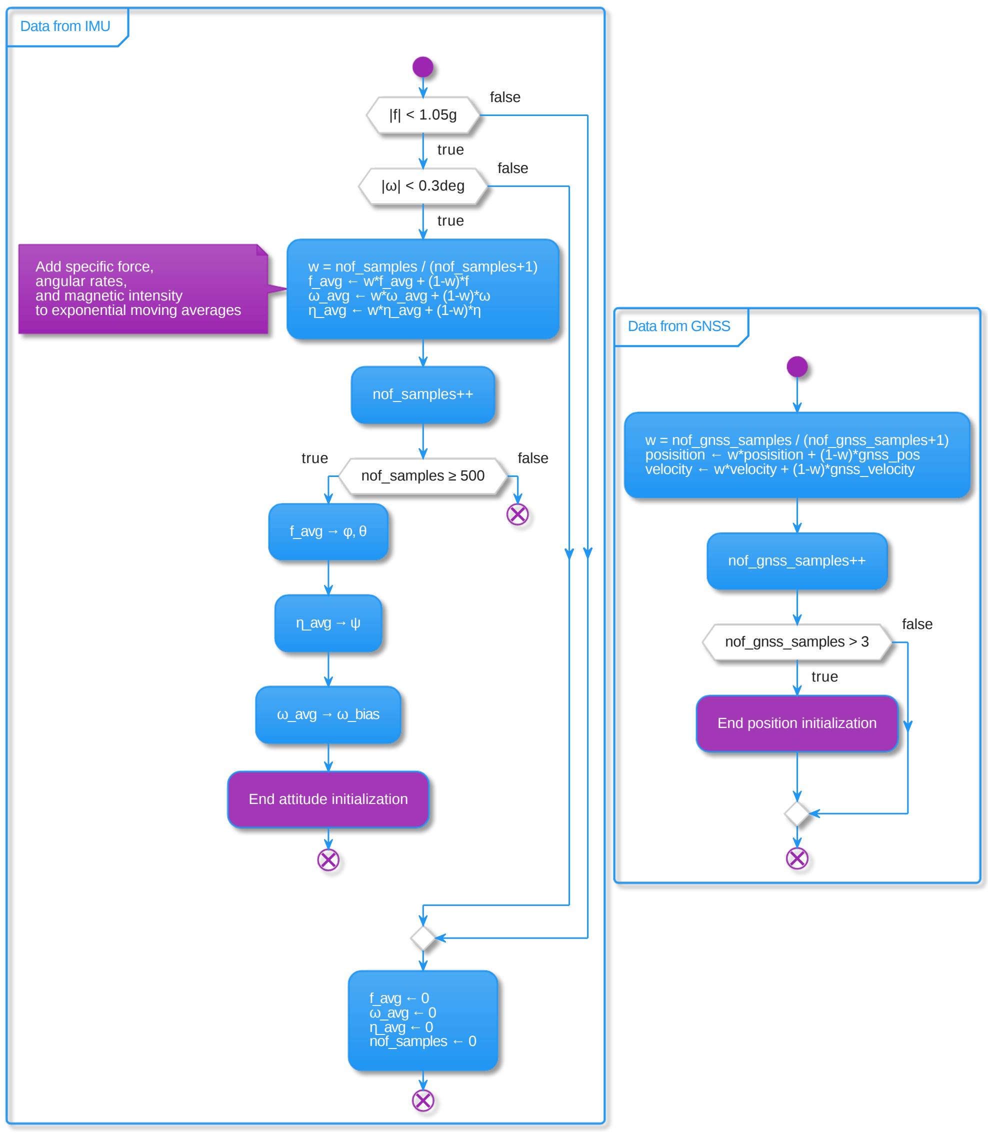
This subsection is technically an extension to the previous one, since it also describes the software behavior. However, it has been separated in order to provide a self-contained segment that explains the initialization of the system. This refers to the time right after the startup of the firmware, when the very first measurements are obtained from the sensors. As was briefly mentioned along within the theoretical background, in this phase the firmware is tasked with determining the initial attitude of the system in terms of the *roll* ϕ, *pitch* θ, and *yaw* ψ Euler angles. The *roll* and *pitch* can be computed from the accelerometer's specific force vector using equations (4) and (5). The *yaw* is then computed from the magnetic field intensity using equation (6), which also incorporates tilt compensation from the known ϕ and θ.

Specifically, the process of initialization will compute the initial attitude after passing the measurements through an *Exponential Moving Average* (EMA). This is effectively a low-pass filter, which is useful in this step, since the initial attitude determination requires ideally a constant measurement. At first, new values will simply be added to the EMA. Then, once a predefined number of values has been accumulated, the averaged measurements are used to compute the initial attitude angles. The weight used for the EMA will be adaptive and equal to $\frac{\#\text{samples}}{\#\text{samples}+1}$. This is convenient, since the fewer samples there are, the more weight it automatically gives to the new measurements.

The same EMA approach will be used to initialize the position and velocity vectors from the first GNSS data. Note that, the number of averaged samples for this cannot be too large, since only one sample will be received every second. It is considered beneficial to devote these extra few seconds to initialization for the benefit of potentially filtering out one or two early erroneous measurements from the GNSS receiver.

To compute the attitude this way, it is necessary for the system to be in standstill conditions. That is because the attitude angles are computed by sensing the effect of the gravitational acceleration on all three sensitive specific force axes. This can only be done if the force of gravity is the only force currently acting on the body. In the firmware, the state of the vehicle will be decided by computing the norms of both the specific force and the angular rate vectors, and comparing them against predefined thresholds. Furthermore, by knowing that the system is in standstill condition, there is one more piece of information that may be extracted from the first measurements: the initial bias of the gyroscope. That is because if to assume that the vehicle is at rest, then the angular rates should measure zero around all three axes. Therefore, the measured average angular rates can be directly copied over to initialize the gyroscope bias vector.

The complete initialization process that is described here is demonstrated in an activity diagram in figure 3.18. Note that the values shown for thresholds and number of samples are just examples used for illustration.



Data from IMU
|f| < 1.05g
false
true
|ω| < 0.3deg
false
true
Add specific force, angular rates, and magnetic intensity to exponential moving averages
w = nof_samples / (nof_samples+1)
f_avg ← w*f_avg + (1-w)*f
ω_avg ← w*ω_avg + (1-w)*ω
η_avg ← w*η_avg + (1-w)*η
nof_samples++
true
nof_samples ≥ 500
false
f_avg → φ, θ
η_avg → ψ
ω_avg → ω_bias
End attitude initialization
f_avg ← 0
ω_avg ← 0
η_avg ← 0
nof_samples ← 0
Data from GNSS
w = nof_gnss_samples / (nof_gnss_samples+1)
posisition ← w*posisition + (1-w)*gnss_pos
velocity ← w*velocity + (1-w)*gnss_velocity
nof_gnss_samples++
nof_gnss_samples > 3
false
true
End position initialization

Figure 3.18: Activity diagram of system initialization from first measurements.

3.3.7 Computation strategy

Previous sections establish the functional requirements for the project, according to which the system must be able to compute the mechanization at about $100\ Hz$. They also outline the software design and behavioral guidelines, according to which the firmware will be governed by a synchronous firmware cycle where all operations will take place. From these, one can conclude that the firmware cycle must not, under any circumstances, exceed $10\ ms$ in execution time. In fact, when considering the batch processing of the FIFO queue from the IMU, it becomes evident that the firmware cycle has to be, on average, significantly shorter than that. Therefore, it is necessary to plan out the time budget of the cycle in order to ensure that all computations are possible, at least theoretically.

In this subsection, for brevity, the risk of losing IMU samples is presented as the key motivator behind the requirement for a strict firmware cycle, however it is not the only one.

While the IMU sampling issue could in theory be solved with hardware interrupts, other requirements like the $100\ Hz$ mechanization update are far less trivial to deal with in a computation-intensive environment.

The first step is to obtain an estimate of how much time is required for the execution of key parts in the firmware cycle. This refers to code sections which are either computationally intensive, or interact with the hardware in a blocking way. The synchronous design of the firmware makes things simple here, since the interrupt execution time is made negligible. Therefore, the execution time of each code segment is practically constant, and does not need to be thoroughly analyzed for edge cases.

These execution time measurements were made with the `-Og` optimization flag, to avoid making optimistic estimates. For reference, the microcontroller's clock has been parameterized in a *Phase-Locked Loop* (PLL) configuration with $64\ MHz$ frequency. The measurements made are given in table 3.2.

Operation	Execution time
Receive the maximum FIFO batch from the IMU	$2\ ms$
Compute mechanization of the system	$1\ ms$
Parse GNSS reception buffer for NMEA messages	$0.7\ ms$
Compute Kalman filter correction update	$7\ ms$
Compute Kalman filter time propagation	$50\ ms$

Table 3.2: Worst-case execution times for key parts of the firmware cycle.

It is immediately obvious that, in this synchronous design, the Kalman filter computations cannot satisfy the temporal constraints of the firmware cycle if they are made sequentially. Either IMU measurements will be missed due to a FIFO overrun, or the system will fail its directive of computing its state and transmitting it to the output every $10\ ms$. An illustration of this problem is shown in figure 3.19, where the reception of new GNSS data at time `7` leads to the firmware computing the Kalman filter updates at time `8`, which causes that specific firmware cycle `fw_cycle` to become significantly longer than the rest. This leads to a hypothetical FIFO queue overrun on the IMU at time `13`.

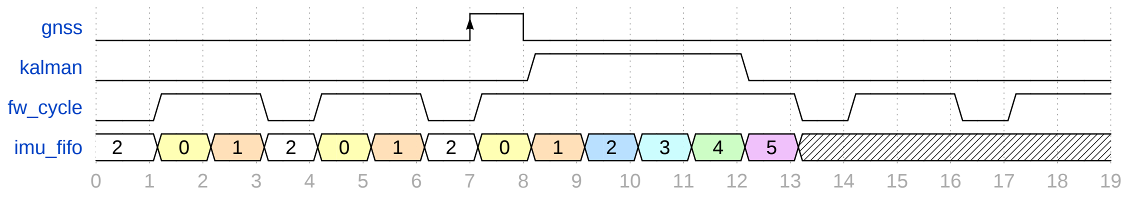


Figure 3.19: Timing diagram showing potential time violation due to Kalman filter calculations.

The only solution here is to spread out these computations over multiple firmware cycles where there is time to spare. This is possible in this case because the Kalman filter updates only need to be computed at approximately $1\ Hz$, when there is new data from the GNSS receiver. This idea, which will subsequently be referred to as *distributed computation*, is illustrated with a simple example in figure 3.20. Here, a function which computes the sum of three numbers is written in two different ways. The first sequentially adds the numbers together and returns the result, performing three operations in the process. The second implements some sort of *Finite State Machine* (FSM), wherein every time the function is called, it only performs one operation before returning. In this example, with the second version of the function, the caller needs to first call `sumStart()` to initiate the computation, and then call `sumStep()` three times to finish the computation. The caller of the function is aware that the result is ready when `sumStep()` returns `0`. In figure 3.21, this principle is also illustrated in the form of a simple state diagram.

```
// computes the sum of three numbers sequentially
int sum(a, b, c) {
    int out = a;
    out += b;
    out += c;
    return out;
}

// computes the sum of three numbers with distributed computation
void sumStart(*out, a, b, c) {
    state_ = 1;
    out_ = out; a_ = a; b_ = b; c_ = c; // store copies of parameters
}
int sumStep() {
    switch (state_)
    {
        case 1:
            *out_ = a_;

            state_++;
            break;
        case 2:
            *out_ += b_;

            state_++;
            break;
        case 3:
            *out_ += c_;

            state_ = 0;
            break;
    }
    return state_;
}
```

Figure 3.20: Illustration of distributed computation principle.

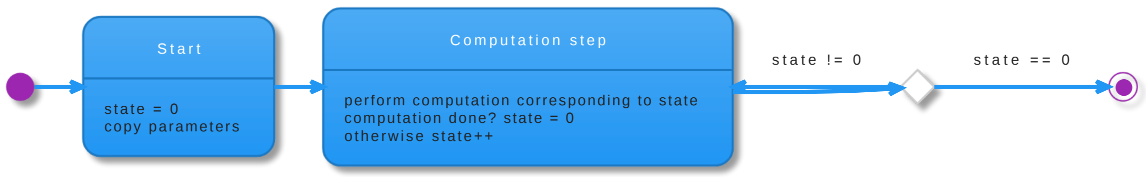


Figure 3.21: State diagram showing the basic FSM structure of distributed computation.

This computation method does of course add some additional overhead. Neither the copying of the function's parameters in the starting step nor the repeated entering and exiting of the computation step function are negligible in terms of processing time.

However, distributed computation will, in principle, be used in cases where the computations performed in each step completely overshadow this overhead. This assumption holds in the case of the Kalman filter updates, where individual computational steps usually involve the multiplication of 15×15 matrices.

What this computation strategy offers is not speed, but flexibility. It provides the ability to split long computations into state machines, which perform only a single computation at each state transition. To realize the benefit of this strategy, recall the example in figure 3.19, where a long computation inside the firmware cycle caused the overrun of the IMUs internal memory, which leads to a loss of measurements. Then, compare that timing diagram with the one shown in figure 3.22, where the Kalman filter updates have been implemented using a distributed computation strategy. As is illustrated in the improved example, the firmware cycle still becomes slightly longer while a Kalman filter computation is ongoing, and in this longer time the FIFO on the IMU does manage to accumulate more measurements, but with proper tuning of the `FIFO_MAX_BATCH_SIZE` parameter and the amount of operations delegated to each computation step, the buffer overrun is prevented in the end.

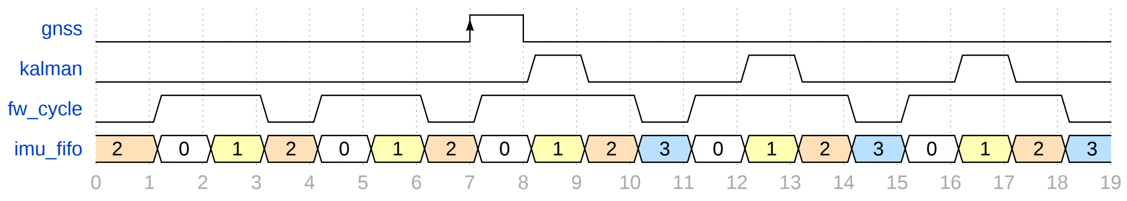


Figure 3.22: Timing diagram showing distribution of computations over multiple cycles.

There are some caveats to consider when implementing this computation strategy:

- Since the Kalman filter computations are effectively preempted by the rest of the firmware between each step, they will take longer to be completed than if they were computed sequentially. This may significantly increase the time-span between the complete reception of the GNSS data, and the correction of the state and biases in the model. This is considered acceptable in this case, since the rest of the firmware continues unimpeded and can compute the system's mechanization at the regular rate.
- Additional care must be exhibited in the processing of the extrapolation buffer, which was described in previous subsections. During the Kalman filter update in the final implementation, the extrapolation itself will also be distributed across multiple computation steps. That is because this buffer can contain up to 100 IMU samples, and each extrapolation step involves a complete computation of the mechanization from a single IMU sample. The consideration here, is that the firmware will continue appending samples to the extrapolation buffer between these computation steps. This is also assumed to not be a risk in this case, since race conditions are prevented by design, and only when the last sample is consumed the extrapolation can conclude and the buffer can be disabled.

- If the distribution of computation operations is done too aggressively, there exists the possibility that the Kalman filter update will not have concluded by the time the next timepulse is received from the GNSS receiver. As has been explained in previous subsections, upon reception of a timepulse interrupt, the firmware needs to reset the extrapolation buffer and to start filling it with IMU samples. If an ongoing Kalman filter computation is still in the process of using this buffer at that time, the system shall enter an error state. Therefore, the distribution of the Kalman filter computations has to be done diligently. The granularity with which operations are spread into multiple cycles shall be coarse enough, so that the maximum firmware cycle length is not exceeded, and at the same time fine enough, so that the error correction phase is not delayed excessively.

3.3.8 Control Area Network bus output

One of the objectives of the diploma thesis, in the context of implementation of the INS, is to output the state of the system on a CAN bus. The INS will continuously and independently output data on the bus at a rate of $100\ Hz$. It was decided that this data shall be in the form of NMEA messages, which is a proprietary protocol issued by the *National Marine Electronics Association* [16]. This protocol was chosen since it is a well-defined industry standard. The messages of this protocol start with the `$` character, followed by a five-character message identification code, then a set of comma-separated data, and finally a checksum, the carriage-return `<CR>` and line-feed `<LF>` characters. The typical message structure is shown below.

$$\$XXCCC, D_1, D_2, .., D_N - -^*hh < CR >< LF >$$

$$\begin{aligned} XX &- \text{talker ID, used to identify the source of the message} \\ CCC &- \text{protocol header, used to identify the message type} \\ D_i &- \text{message payload} \\ ^*hh &- \text{checksum} \end{aligned}$$

For the Mini-INS, it is decided that the output messages should bear the IN talker ID, which stands for *Inertial Navigation*. Then, in order to communicate the entire PVA state, the messages outlined further will be used. Unfortunately, there are no standard messages in the NMEA specification for providing 3D velocity or attitude. Therefore, to convey those two pieces of information, two vendor-proprietary messages will be used: *RMV* [18] and *SHR* [19].

- **GGA - Global Positioning System Fix Data**

  ```
  $--GGA,hhmmss.ss,llll.lll,a,yyyyy.yyy,a,x,uu,v.v,w.w,M,x.x,M,,zzzz*hh<CR><LF
  ```

  - `hhmmss.ss` UTC Time UTC of position in hhmmss.sss format, (000000.000 ~ 235959.999)
  - `llll.lll` Latitude Latitude in ddmm.mmmm format. Leading zeros are inserted.

- `A` N/S Indicator ‘N’ = North, ‘S’ = South
- `yyyyy.yyy` Longitude Longitude in dddmm.mmmm format. Leading zeros are inserted.
- `A` E/W Indicator ‘E’ = East, ‘W’ = West
- `x` GPS quality indicator
- `uu` Number of satellites in use, (00 ~ 24)
- `v.v` HDOP Horizontal dilution of precision, (00.0 ~ 99.9)
- `w.w` Altitude Mean sea level altitude (-9999.9 ~ 17999.9) in meters
- `x.x` Geoidal Separation in meters
- `zzzz` DGPS Station ID Differential reference station ID, 0000 ~ 1023
- `hh` Checksum

- **RMV - 3D Velocity Information**

  `$--RMV,+xxx.x,+yyy.y,+zzz.z*hh<CR><LF>`

  - `+xxx.x` True east velocity (-514.4 to 514.4 meters/second)
  - `+yyy.y` True north velocity (-514.4 to 514.4 meters/second)
  - `+zzz.z` Up velocity (-999.9 to 999.9 meters/second)
  - `hh` Checksum

- **SHR - Inertial Attitude Data**

  `$--SHR,hhmmss.sss,hhh.hh,T,rrr.rr,ppp.pp,xxx.xx,a.aaa,b.bbb,c.ccc,d,e*hh<CR>`

  - `hhmmss.ss` UTC Time UTC of position in hhmmss.sss format, (000000.000 ~ 235959.999)
  - `hhh.hh` Heading in degrees
  - `T` Flag to indicate that the Heading is True Heading (i.e. to True North)
  - `rrr.rr` Roll Angle in degrees
  - `ppp.pp` Pitch Angle in degrees
  - `xxx.xx` Heave
  - `a.aaa` Roll Angle Accuracy Estimate (Stdev) in degrees
  - `b.bbb` Pitch Angle Accuracy Estimate (Stdev) in degrees
  - `c.ccc` Heading Angle Accuracy Estimate (Stdev) in degrees
  - `d` Aiding Status
  - `e` IMU Status
  - `hh` Checksum

Note that values such as HDOP, number of satellites in use, and GPS quality indicator are not anyhow computed in the firmware. These will simply be stored from the input NMEA sentences from the GNSS receiver and written directly to these output messages.

3.3.9 Auxiliary serial output

This section briefly recounts the implementation of an auxiliary serial interface that is available on the microcontroller. Already since the subsection about prototyping it was hinted that there exists a serial interface on which raw data were sent to a *Raspberry Pi* or a computer running *Simulink*. In fact, this is a simple UART driver, on which data are continuously being written during the board's operation. In the final PCB, this is done on the ST_UART1_TX pin which is lead out to the J4 connector. This way, the interface may be utilized even in the future during further development.

The existence of this interface was omitted from the software behavior diagrams in the previous subsections. This was done for brevity, since this is an optional part which should be possible to be even completely disabled for normal operation. Apart from that, it plugs into the main application much like any other module, as it is illustrated in figure 3.15. Specifically, with methods that are called `rawInterfaceInit()` and `rawInterfaceUpdate()`.

The operation of this driver is quite simple. New data are all written on a buffer, and when data in the buffer exceed some threshold, a DMA transfer starts, which transmits the entire buffer out on the UART TX line. In practice, two buffers are required to do this properly: one read buffer and one write buffer. New data are all always added to the write buffer. Then, before a transfer is initiated, the two buffers are swapped, so that the write buffer becomes the new read buffer and vice versa. The DMA transfer then transfers the contents of the read buffer. This scheme is necessary for the firmware to continue being able to write data even when a DMA transfer is ongoing. This functionality of the `rawInterfaceUpdate()` method is shown in figure 3.23.

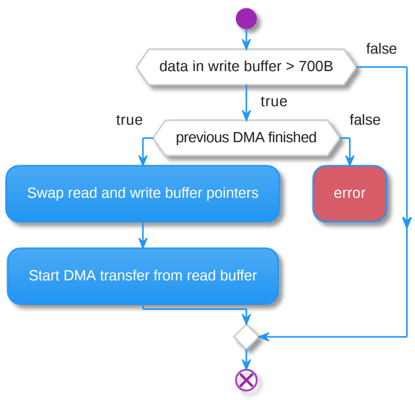

data in write buffer > 700B
false
true
true
previous DMA finished
false
Swap read and write buffer pointers
error
Start DMA transfer from read buffer

Figure 3.23: Activity diagram of rawInterfaceUpdate() method.

Data is transmitted on this serial output in the form of messages. Each message starts with the `$` character, and is terminated by `\n`. It is made up of fixed-length data in a specific sequence. The structure of these messages is given for reference in tables 3.3, 3.4, and 3.5.

Byte position	Length	Type	Value
0	4B	`string`	`$imu`
4	8B	`uint64`	timestamp in ms
8	4B	`float`	temperature in $^{\circ}C$
12	12B	3x `float`	angular rates in $\frac{rad}{s}$

Byte position	Length	Type	Value
24	12B	3x `float`	specific force in g
36	12B	3x `float`	magnetic field intensity in T
48	4B	`uint32`	sample time in us
52	12B	3x `float`	accelerometer bias in g
54	12B	3x `float`	gyroscope bias in $\frac{rad}{s}$

Table 3.3: Structure of IMU raw data message data on the serial output.

Byte position	Length	Type	Value
0	4B	`string`	`$gps`
4	8B	`uint64`	timestamp in ms
12	8B	`double`	latitude in deg
20	8B	`double`	longitude in deg
28	8B	`double`	altitude in m
36	12B	3x `float`	velocity in NED $\frac{m}{s}$
48	8B	`uint64`	NMEA timestamp in ms

Table 3.4: Structure of GNSS raw data message on the serial output.

Byte position	Length	Type	Value
0	4B	`string`	`$pva`
4	8B	`uint64`	timestamp in ms
12	24B	3x `double`	position in LLA (rad, rad, m)
36	12B	3x `float`	velocity in NED $\frac{m}{s}$
48	36B	9x `float`	attitude as a 3x3 DCM
84	1B	`bool`	valid gnss fix

Table 3.5: Structure of PVA message on the serial output.

3.3.10 Configuration parameters

This last subsection provides an overview of the available parameters for the configuration of the INS. The reader is referred to the file `config.h` among the source code, where these parameters are defined as macros. Table 3.6 serves as a reference of available options, along with their description and default values. Note that specific hardware configurations, such as GPIO ports and pin numbers, are also defined in that file, but are not listed here since they are fixed to the PCB and not of operational significance. Also, during the testing and evaluation of the board, only the default values listed were used. How the system operates under different configurations has not been explored.

Parameter	Description	Default value
`IMU_ODR`	Output data rate of the IMU sensors (in Hz according to the datasheet)	`1660`
`MAG_ODR`	Output data rate of the magnetometer (in Hz according to the datasheet)	`100`
`IMU_ACCEL_RANGE_G`	Full scale range of the accelerometer (in g according to the datasheet)	`4`
`IMU_GYRO_RANGE_DPS`	Full scale range of the gyroscope (in $\frac{\circ}{s}$ according to the datasheet)	`500`
`IMU_INITIALIZATION_SAMPLES`	Number of IMU samples to be used in the initialization phase	`500`
`IMU_FIFO_MAX_BATCH_SIZE`	Maximum number of IMU samples to be read out of the FIFO in a single firmware cycle	`16`
`IMU_MIN_NOF_SAMPLES`	Minimum number of IMU samples to accumulate in the decimation filter, before using them to compute the system's mechanization	`16`
`SF_EMA_WEIGHT`	A weight in the range `[0, 1]`, to be given to new specific force measurements in the EMA filtering	`0.05`
`GNSS_BUFFER_LEN`	Size of the buffer that stores data read on the UART RX line from the GNSS receiver (in bytes)	`512`
`EXTRAPOLATION_BUFFER_LEN`	Size of the extrapolation buffer (in number of IMU samples)	`100`

Parameter	Description	Default value
`ACC_DYN_THRESHOLD`	Threshold for determining system dynamics from accelerometer measurement norm (in g)	`1.05`
`GYRO_DYN_THRESHOLD`	Threshold for determining system dynamics from gyroscope measurement norm (in $\frac{rad}{s}$)	`deg2rad(0.3)`
`ACC_BIAS_DELTA_LIMIT`	Maximum absolute change to be made in a single Kalman update, to the bias of a single axis of the accelerometer	`0.001`
`GYRO_BIAS_DELTA_LIMIT`	Maximum absolute change to be made in a single Kalman update, to the bias of a single axis of the gyroscope	`deg2rad(0.001)`
`GNSS_FIX_VALIDITY_MS`	The timespan after a GNSS timepulse, where it is considered that there is an active GNSS fix. This GNSS status flags at the system's outputs (in ms).	`2000`
`GNSS_FIX_RESET_MS`	Time of GNSS outage, after which the Kalman filter is reset (in ms)	`5000`

Table 3.6: Parameters available for compile-time configuration.

3.4 Verification of results

This section outlines the verification of the completed INS. It includes a brief overview of how experimental data were obtained and processed. In addition, it showcases the results that were obtained, as well as the author's assessment of the system's performance.

As a disclaimer, it should be noted that the verification was not performed on a completed PCB. At the time of writing, the manufacturing process has begun, but has not yet been completed. Therefore, any verification data included here were obtained using the breadboard prototype shown in an earlier section. Unfortunately, it is not possible to assess the final accuracy of the designed system this way. That is because some components that are used on the prototype offer (theoretically) inferior performance, compared to the components that were selected for the PCB manufacturing. The most significant of them being the GNSS receiver *u-blox NEO-6P* that was used with the prototype, which advertises a lower accuracy than the selected *u-blox ZED-F9T*, and also lacks its features of compensation to ionospheric errors. Finally, the overall makeshift assembly of MEMS inertial sensors connected with jumper wires is not robust, and potentially leads to vibration on the IMU or noise on the communication signals.

Therefore, this section focuses on a qualitative assessment of the system's performance. This means that the system is expected to perform its function correctly throughout the experiment. For reference, this includes the following:

- Correctly perform the Kalman filter error compensations, while maintaining an output rate of $100\ Hz$.
- Maintain accurate attitude, acknowledging that the platform may exhibit vibrations in excess to those of the vehicle.
- Maintain position with a horizontal accuracy, at least that of the standalone GNSS receiver, which is $2.5\ m$.
- Maintain velocity which upholds the positional accuracy requirement between GNSS updates.
- Timestamp all output data with a $10\ ms$ precision to the UTC time synchronized at the GNSS receiver.

As described in the subsection about the prototype, the auxiliary serial interface is connected to a *Raspberry Pi Zero* which is able to record all raw data to a USB drive. The data is then parsed and plotted using a Python script. To avoid overloading this document, mostly an overview will be given with key plots. For detailed results of the experiments, the reader is instead referred to the data which is provided among the attachments.

For the experiment, the prototype was taken on public transport vehicles in Prague and secured on a seat as much as possible. The results shown here are from a two-part experiment: firstly tram number 9 was ridden from stop *Vozovna Motol* to stop *Sídliště Řepy*, followed by a ride on bus number 180 to the stop *Zličín*.

The *latitude* and *longitude* of positions recorded is shown in figure 3.24. A red star denotes the position where the experiment was moved from the tram to the bus. Zooming in, it is possible to observe the effect of the aided navigation using the Kalman filter. As can be seen in the enhanced part of the image, the output positions can be visually distinguished into groups of ten or so points, where the velocity and position are computed at a high rate from IMU measurements. Between these groups of points, it can be seen how $1\ Hz$ update from the GNSS is used with the Kalman filter to correct the vehicle's trajectory. The reported altitude is also given for completeness in figure 3.25. Note that the gap near the middle of the following plots is due to the move from the tram to the bus, when recording was temporarily stopped.

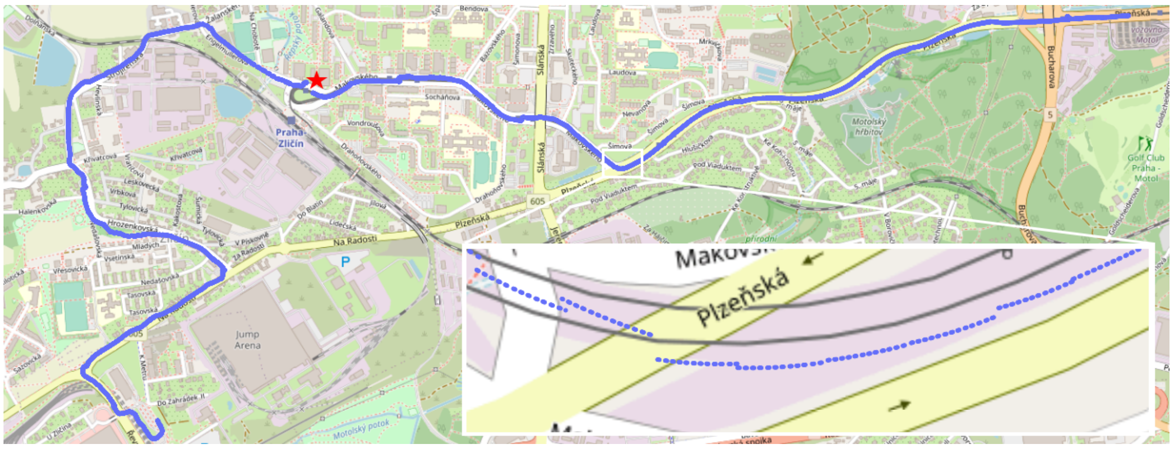


Figure 3.24: Map overview of experiment route.

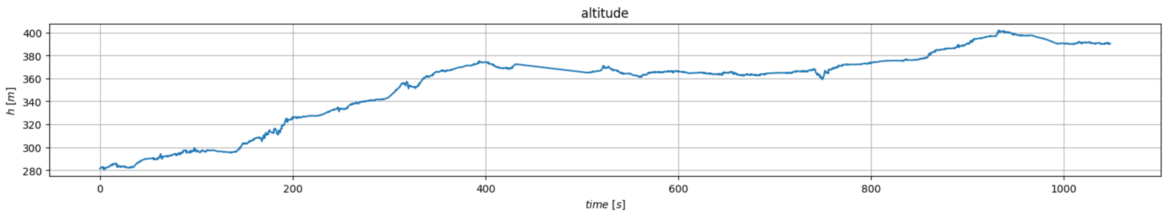


Figure 3.25: Altitude from experiment data.

Next up, the attitude-tracking precision of the INS should be ascertained. Firstly, a good way to evaluate the computed *yaw*, is to overlay arrow lines on the map at various points on the route, which are angled according to the heading at that position. This can be seen in figure 3.26. There are areas with some error in the heading, but these errors are corrected in each case after a few seconds.

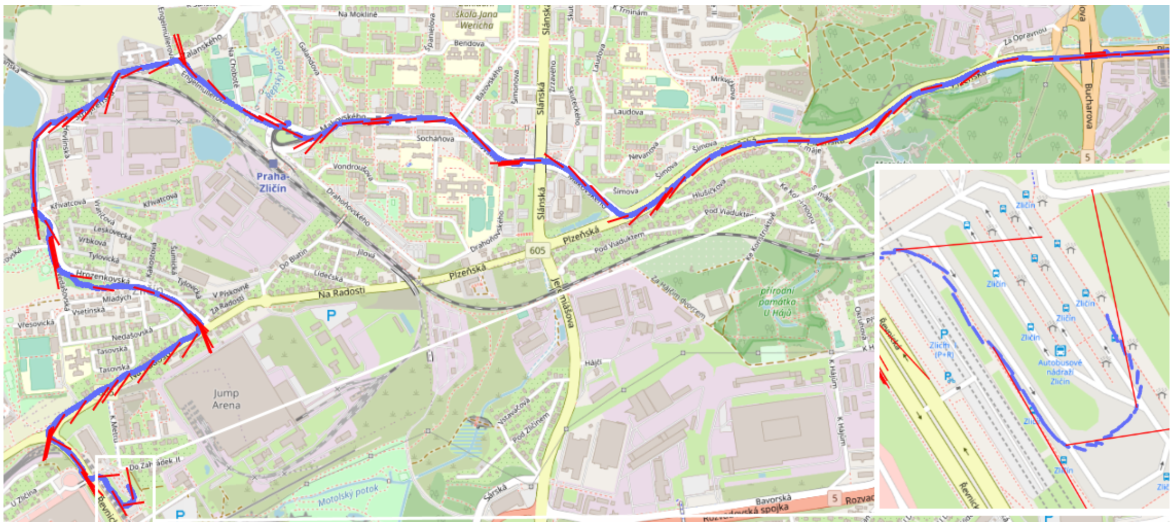


Figure 3.26: Map overview of experiment route, with superimposed heading pointers.

Following that, the computed speed shall be evaluated. Since the experiment was performed on ground vehicles, it is useful to inspect the velocity in terms of ground speed in $\frac{km}{h}$, and vertical speed. The results are shown in the figure 3.27. Note that for convenience, the vertical speed depicted in the plot is inverted, since the vertical velocity reported by the INS points downwards. The ground speed results look quite good; it is clear how between stops the vehicle accelerates up to roughly $50\ \frac{km}{h}$, and then decelerates to a halt. In the second part of the plot, which represents data collected from the bus, it can even be seen how the bus decelerates to yield priority or to make a turn.

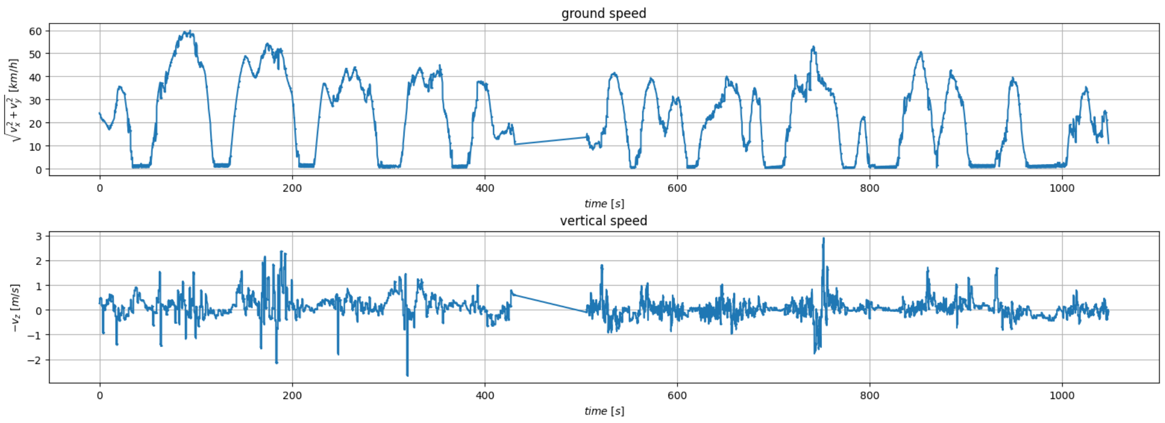


Figure 3.27: Ground speed and vertical speed from experiment data.

Next, it is important to evaluate the output attitude angles. The results of the experiment are shown in figure 3.28. As expected, the *yaw* angle exhibits little variance and only changes when the vehicle makes a turn. The *roll* and *pitch* angles on the other hand, are quite noisy here. This is likely attributed to the less-than-ideal fastening of the prototype on the bus seat. However, it is still possible to observe that there is increased pitch when the vehicle is climbing to a higher altitude, and some increased roll when the vehicle is turning.

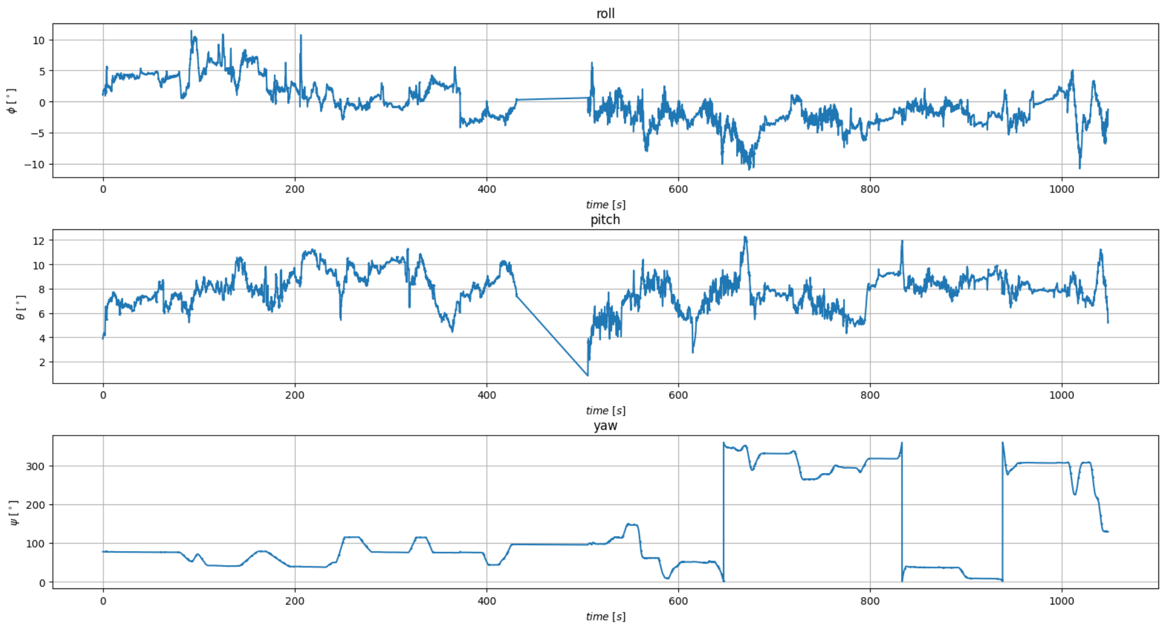


Figure 3.28: Attitude Euler angles from experiment data.

One more output that may be useful to include, is the progression of the sensor biases throughout the experiment. They are shown in figure 3.29. These values are difficult to evaluate by themselves, but they do exhibit a progression in one direction, which is what is generally expected from these types of sensors.

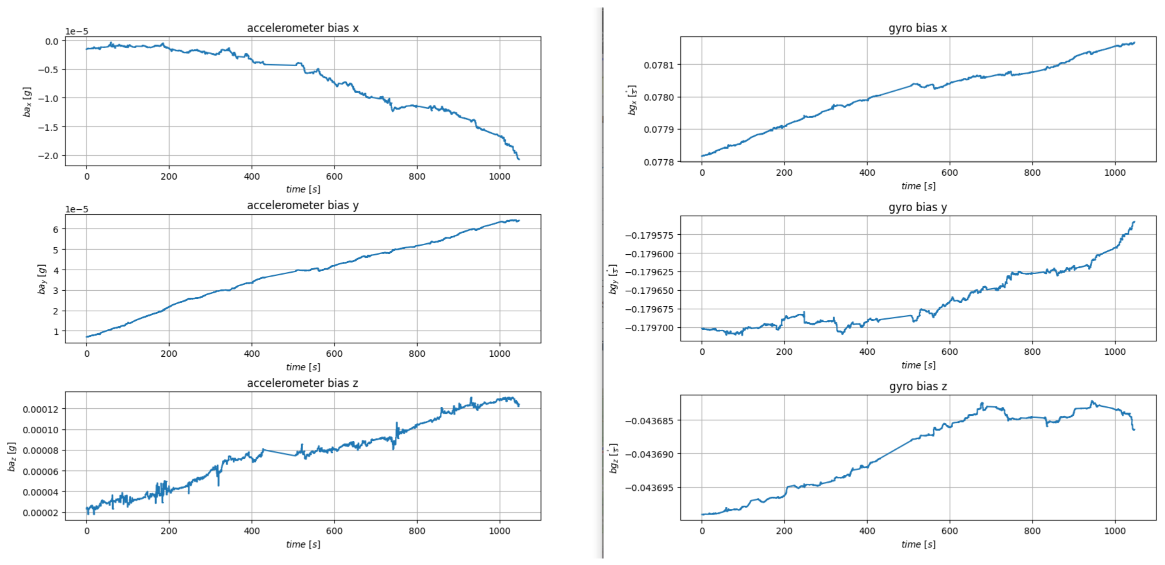


Figure 3.29: Accelerometer and gyroscope biases from experiment data.

Finally, the last thing to verify is the correct timestamping of the reported data. As mentioned previously in the document, all output data should include a timestamp of its validity with $10\ ms$ precision. These timestamps should be synchronized to the UTC time through the GNSS receiver and the timepulses. A short chunk of timestamps from the output data of the experiment is shown in figure 3.30. In the plot, timestamps from the GNSS NMEA messages which are received approximately every $1\ sec$, are shown as a dashed red line. The blue line shows the timestamps as they annotate the output PVA data that are read out. This is a $Y = X$ plot for illustration.

Note that some steps of the dashed red line are longer than others. In fact, the shorter steps represent $1\ sec$ time increases. There are two factors that may have contributed to the longer gaps between GNSS timestamp updates. The first is that there was some temporary GNSS outage, and the current firmware discards GNSS data when there is no 3D fix. The second is that some data may have been periodically discarded at the serial interface of the *Raspberry Pi Zero* due to it being a single-core processor running Linux, which may not have kept up with the transmission rate in real-time.

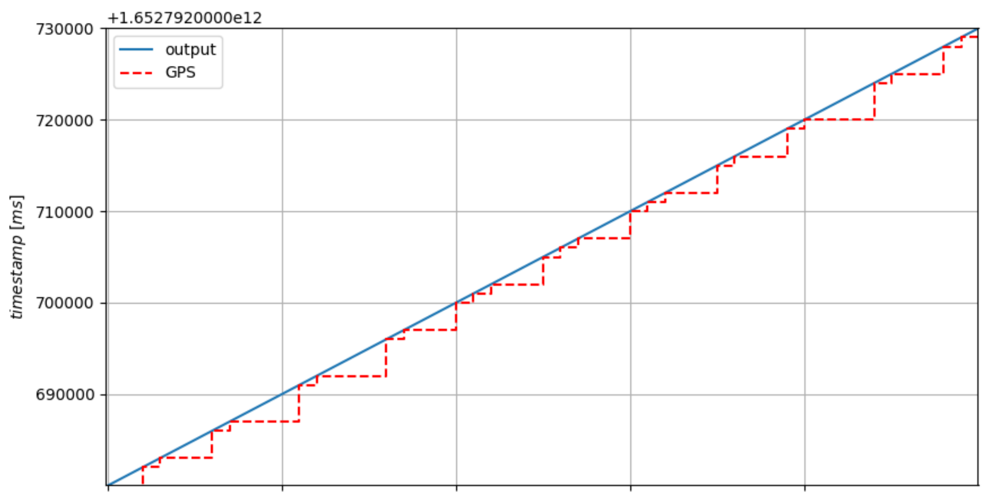


Figure 3.30: Time-stamping of outputs in experiment data.

3.4.1 Comparison with reference MATLAB model

The results presented in this section, appear to be those of a functional INS. However, the focus of this diploma thesis is the adjustment of a given MATLAB model for use by an embedded microcontroller. Looking at the results alone, it is difficult to ascertain whether or not that conversion is successful. Therefore, an additional step is necessary: the comparison between the output computed by the micronctroller on the INS and by the reference MATLAB model. In the interest of this, raw measurements from the IMU and GNSS receiver are also being sent out from the microntroller via the auxiliary serial interface. These raw measurements were also stored during the experiment, and thus can be given as input to the MATLAB model.

The main outputs to compare are those for position, velocity, and attitude. Side-by-side comparisons are given for each in figures 3.31, 3.32, and 3.33 respectively.

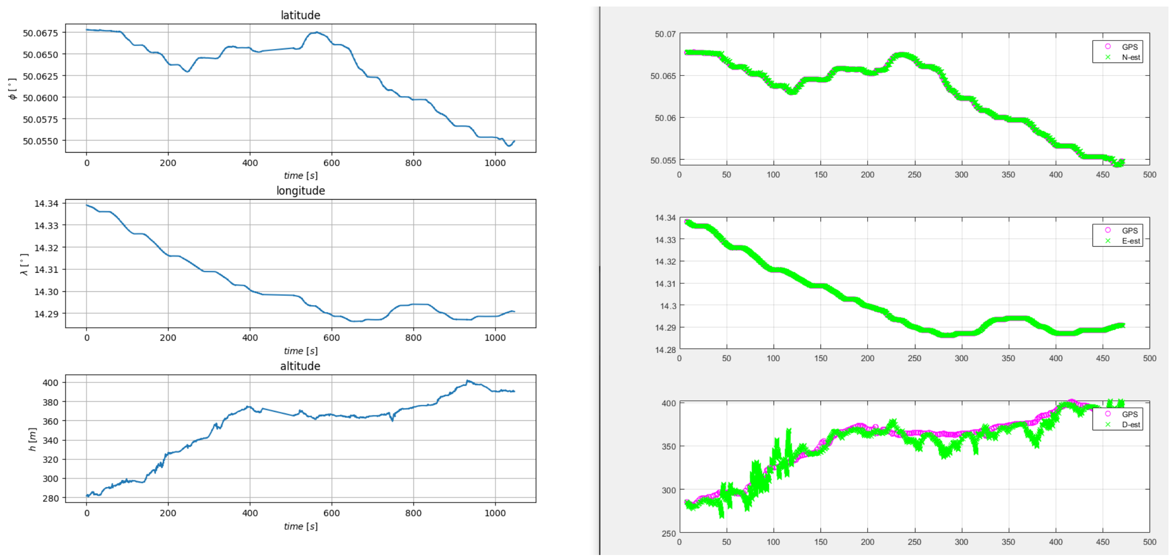


Figure 3.31: Comparison of position output, between own implementation (left) and reference MATLAB model (right).

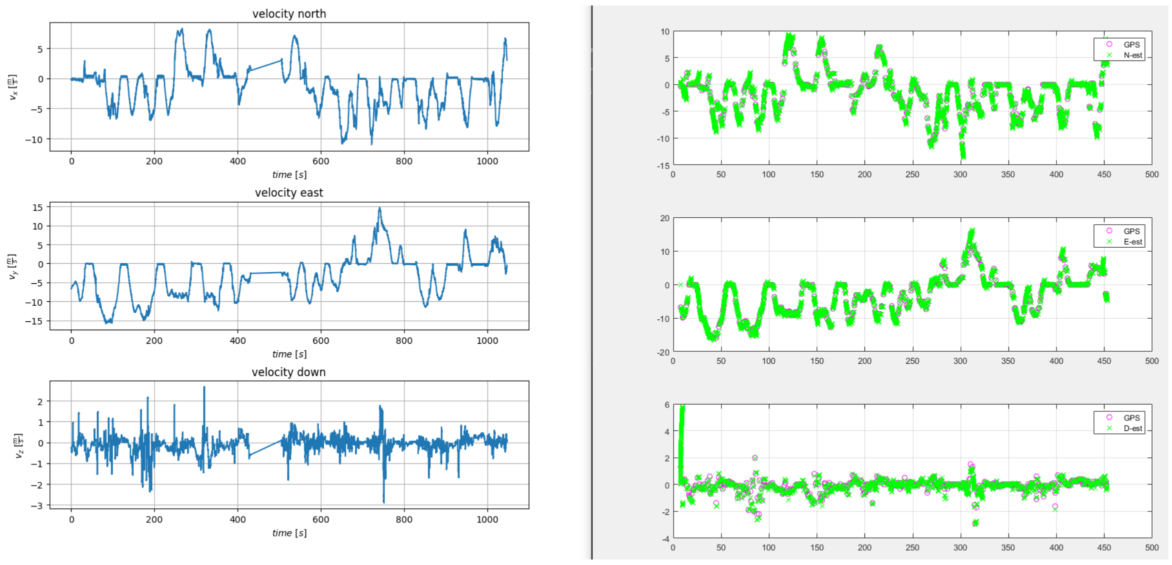


Figure 3.32: Comparison of velocity output, between own implementation (left) and reference MATLAB model (right).

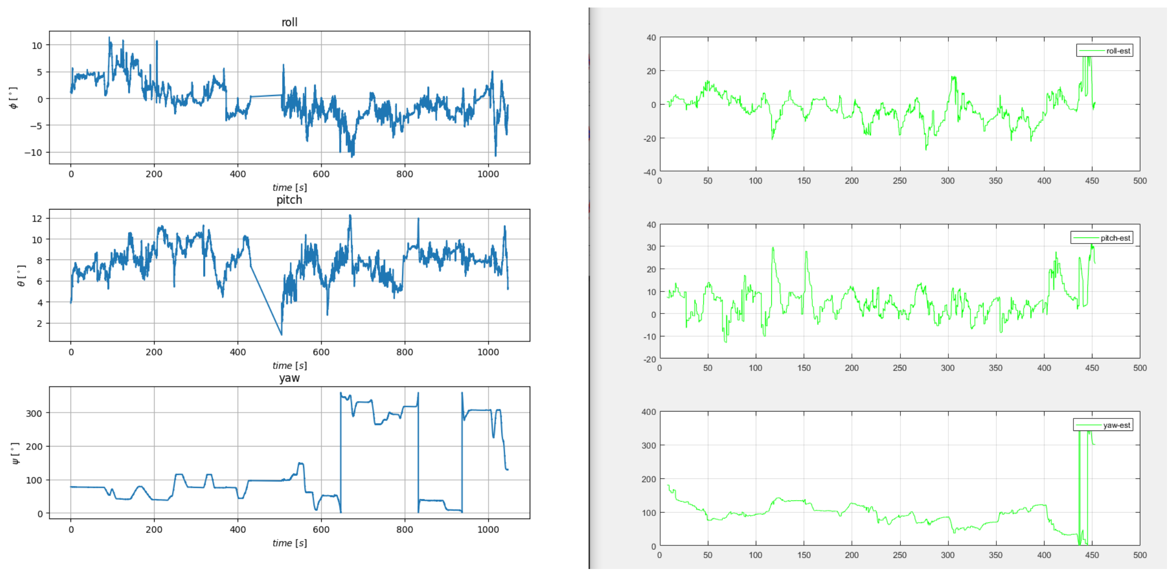


Figure 3.33: Comparison of attitudes output, between own implementation (left) and reference MATLAB model (right).

Visually, the microcontroller implementation appears to be able to track the reference model decently. Some deviations do exist, particularly with the attitude outputs. The hypothesis is that this may be due to differences in the computation flow in each case; for example filter windows, initialization, and so on.

Chapter 4
Conclusion

This diploma thesis involved the design and realization of an *Inertial Navigation System*. Specifically, the focus was to create a miniature unit, whose tight size and low power consumption would make it an attractive choice for compact *Unmanned Aerial Systems*. The primary objectives were to design a small low-power *Printed Circuit Board* and to develop the firmware for an embeeded microcontroller which would provide position, velocity, and attitude information at $100\ Hz$.

Having concluded the work on the INS, the following goals were met:

- The PCB was designed and sent for manufacturing. In terms of size, the final dimensions were $30\ mm \times 37\ mm$, which is only about twice the area of a €2 coin. The design meets the standard good practices, and minimizes electrical noise propagation around sensitive components and the entire board. At the time of writing, it is pending delivery and soldering.
- The Kalman filter model that was given to the author, was adapted for use on an embedded microcontroller. The correct functionality was verified in operation, and found to be satisfactory. The algorithm was optimized for the available computation power, and distributed in such a way that the target $100\ Hz$ output rate is not compromised.
- A complete INS firmware was developed for the selected STM32 microcontroller. It is able to communicate with all connected sensors efficiently, and can process measurements fast enough so that not a single sampled quantity is lost. It can compute the system's mechanization to produce PVA output at a rate of $100\ Hz$, as well as compute the Kalman filter update steps every $1\ sec$ without compromising its output rate.

Having concentrated on the fulfillment of the above objectives, some topics for future improvement still exist regarding this INS. These were planned for at the hardware level, and included in the PCB design, so their implementation should be possible entirely in the software.

- Incorporation of the magnetometer into the the Kalman filter estimation. This has the potential of improving system accuracy, since the quantity measured by this sensor is different than the rest, and can be theoretically used to infer other values such as heading and gyro bias.
- Incorporation of the static pressure sensor into the Kalman filter estimation. For the same reasons stated above.

Appendix A
List of Acronyms

AC	Alternating Current	INS	Inertial Navigation System
ARM	Advanced RISC Machines	LED	Light-Emitting Diode
BSP	Board Support Package	LLA	Latitude Longitude Altitude
CAN	Controller Area Network	LQE	Linear Quadratic Estimator
DC	Direct Current	LQFP	Low Profile Quad Flat Package
DCM	Direction Cosine Matrix	MCU	Microcontroller Unit
DMA	Direct Memory Access	MEMS	Micro Electromechanical Systems
DOF	Degrees Of Freedom	NED	North East Down
ECEF	Earth-Centered, Earth-Fixed	NMEA	National Marine Electronics Association
ENU	East North Up	PCB	Printed Circuit Board
ESD	Electrostatic Discharge	PLL	Phase-Locked Loop
FIFO	First In First Out	PPS	Pulse Per Second
FOSS	Free and Open Source Software	PVA	Position, Velocity, Attitude
FSM	Finite State Machine	RTC	Real Time Clock
GNSS	Global Navigation Satellite Systems	SMD	Surface-Mounted Device
GPIO	General Purpose Input Output	SWD	Serial Wire Debug
GPS	Global Positioning System	UART	Universal Asynchronous Receiver-Transmitter
HDOP	Horizontal dilution of precision	UAS	Unmanned Aerial System
I/O	Input Output	UAV	Unmanned Aerial Vehicle
I2C	Inter-Integrated Circuit	UML	Unified Modeling Language
IC	Integrated Circuit	USB	Universal Serial Bus
IDE	Integrated Development Environment	UTC	Universal Time Coordinated
IMU	Inertial Measurement Unit		

Appendix B
References

[1] DiSalle, Robert, (2002), "Space and time: Inertial frames" https://plato.stanford.edu/entries/spacetime-iframes/#InerFram20thCentSpecGeneRela

[2] Farrell, Jay, (2008), "Aided navigation: GPS with high rate sensors", McGraw-Hill, Inc.

[3] Nourmohammadi, Hossein and Keighobadi, Jafar, (2018), "Fuzzy adaptive integration scheme for low-cost SINS/GPS navigation system", Mechanical Systems and Signal Processing

[4] Ahmad, Norhafizan and Ghazilla, Raja Ariffin Raja and Khairi, Nazirah M and Kasi, Vijayabaskar, (2013), "Reviews on various inertial measurement unit (IMU) sensor applications", International Journal of Signal Processing Systems

[5] Sukkarieh, Salah and Gibbens, Peter and Grocholsky, Ben and Willis, Keith and Durrant-Whyte, Hugh F, (2000), "A low-cost, redundant inertial measurement unit for unmanned air vehicles", The International Journal of Robotics Research, SAGE Publications

[6] El-Sheimy, Naser and Hou, Haiying and Niu, Xiaoji, (2007), "Analysis and modeling of inertial sensors using Allan variance", IEEE Transactions on instrumentation and measurement, IEEE

[7] VectorNav, "What is an Inertial Measurement Unit?" https://www.vectornav.com/resources/inertial-navigation-articles/what-is-an-inertial-measurement-unit-imu

[8] Elmenreich, Wilfried, (2002), "An introduction to sensor fusion", Vienna University of Technology, Austria

[9] Kalman, Rudolph Emil, (1960), "A new approach to linear filtering and prediction problems"

[10] Petteri Aimonen https://commons.wikimedia.org/w/index.php?curid=17475883

[11] Pedley, Mark, (2013), "Tilt sensing using a three-axis accelerometer", Freescale semiconductor application note, Freescale Semiconductor Austin, TX, USA

[12] Grygorenko, Vadym and Family, Associated Part and CY8C27xxx, C, (2011), "Sensing-magnetic compass with tilt compensation", Cypress Perform

[13] Thornton, Catherine L and Bierman, Gerald J, (1980), "UDU/T/covariance factorization for Kalman filtering"

[14] Bierman, Gerald J, (1976), "Measurement updating using the UD factorization", Automatica, Elsevier

[15] Burkhardt, Andrew J and Gregg, Christopher S and Staniforth, J Alan, (2000), "Calculation of PCB track impedance", Circuit World, MCB UP Ltd

[16] SiRF Technology, Inc, (2007), "NMEA Reference Manual" https://www.sparkfun.com/datasheets/GPS/NMEA%20Reference%20Manual-Rev2.1-Dec07.pdf

[17] Hobbs, Chris, (2010), "Protecting Applications Against Heisenbugs", QNX Software Systems

[18] Garmin International, Inc., (2008), "Garmin Proprietary NMEA 0183 Sentences TECHNICAL SPECIFICATIONS" https://developer.garmin.com/downloads/legacy/uploads/2015/08/190-00684-00.pdf

[19] Eric S. Raymond, (2022), "NMEA Revealed" https://docs.novatel.com/OEM7/Content/SPAN_Logs/PASHR.htm