TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRONIC & COMPUTER ENGINEERING

# Unification of peripheral sensors with an autopilot in an Odroid platform

**Christos Melas**

Chania, June 2016

# Abstract

Undoubtedly, the field of UAVs is a fast-growing field, with drones being widely sold for professional as well as for recreational reasons. New models are constantly developed, with more complicated functions, providing a better flying experience and more features for the user. Combining the field of UAVs, with another thriving field, the one of the smartphones, lead to the design and the implementation of an application, where the Flight Controller, an Odroid platform, is unified with an Android device. The purpose of the Android device is to replace the GPS receiver and the Inertial Measurement Unit of the UAV. In that way, the Android device is responsible for providing the sensor measurements, and the location information from the GPS receiver to the autopilot software.

The autopilot software that was selected is PenguPilot, an open source GNU/Linux based Multi-Rotor UAV Autopilot. The main advantage of using a Linux-based autopilot is that high level programming languages can be used in order to develop software that could perform a variety of functions, and take advantage of the processing power of the octa-core Odroid board.

An Android application was developed, which is capable of providing the sensor measurements as well as location information via the USB serial port of the Android device to the Flight Controller. The appropriate software for handling the received data from the serial port was developed, and finally, a new library was developed and added to PenguPilot, for the seamless integration of the Android device with the autopilot software. The functionality of the system was verified, as it is of utmost importance to guarantee that both the Android application and the software on the Flight Controller will continue to operate successfully without any unexpected malfunctions.

# Acknowledgements

I would like to thank my thesis supervisor, Professor Apostolos Dollas, for his guidance, patience and for giving me the opportunity to expand my knowledge in the field of Embedded Systems and Android Development. I would also like to thank Professor Panagiotis Partsinevelos for introducing me to the SenseLab team, as I was able to gain experience in the field of Unmanned Aerial Vehicles. Moreover, I would like to thank all members of the SenseLab team, especially Sarantis Kyritsis and Nikos Prokas, who spent hours guiding and advising me throughout the development phases of my work, as well my friend Varvara Gioti for her valuable advice.

Last but not least, I owe many thanks to my parents for the support that they provide me all of these years, as if it weren’t for them I would not be the person that I am today.

Dedicated to my parents

# Table of Contents

# List of Figures

# List of Tables

# List of Abbreviations

| **Abbreviation** | **Definition** |
|---|---|
| API | Application Programming Interface |
| ASCII | American Standard Code for Information Interchange |
| CPU | Central Processing Unit |
| ESC | Electronic Speed Control |
| GPS | Global Positioning System |
| HAL | Hardware Abstraction Layer |
| IMU | Inertial Measurement Unit |
| NMEA | National Marine Electronics Association |
| PPM | Pulse Position Modulation |
| PWM | Pulse Width Modulation |
| RSS | Resident Set Size |
| SCL | Signaling and Communication Link |
| SSH | Secure Shell |
| UAV | Unmanned Aerial Vehicle |
| USB | Universal Serial Bus |

# Chapter 1

# Introduction

## 1.1 Introduction to Unmanned Aerial Vehicles

Over the past few years, the widespread availability of Unmanned Aerial Vehicles (UAVs) has enabled consumers to approach this emerging technology. Consumer grade UAVs, commonly known as drones, have flooded the market, as their mass production resulted in lower prices and they are becoming increasingly popular for recreational and commercial uses. UAVs are an efficient cost alternative to helicopters and fixed wing aircraft for a wide variety of aerial support applications, ranging from aerial photography to advanced real time data collection [1], with rotary-wing drones dominating the field, as their maneuverability and their ability to hover renders them preferable. The most common type of commercial UAVs are the quadcopters, a special category of rotary-wing drones that are lifted and propelled by four rotors. The control systems for UAVs typically consist of radio controlled commands and can include an autopilot software for autonomous operation. All autonomous aerial vehicles depend on sensors that measure and report the acceleration rate of the body, angular velocity changes in three dimensions, known as pitch, roll, and yaw, and usually the magnetic field around it as well. Accelerometers, gyroscopes, and magnetometers are needed for each of these measurements respectively, and the group of these sensors comprises an Inertial Measurement Unit (IMU). Some of the typical components of a quadcopter and their interconnections are presented in Figure 1.1. The main component, the flight controller, that contains a microcontroller and an IMU, is connected via Pulse Width Modulation (PWM) to each one of the four Electronic Speed Control (ESC) circuits that vary the speed of the motors. Additionally, a Global Positioning System (GPS) receiver is connected to the Flight Controller via a serial port, and the Radio Control receiver via PWM.

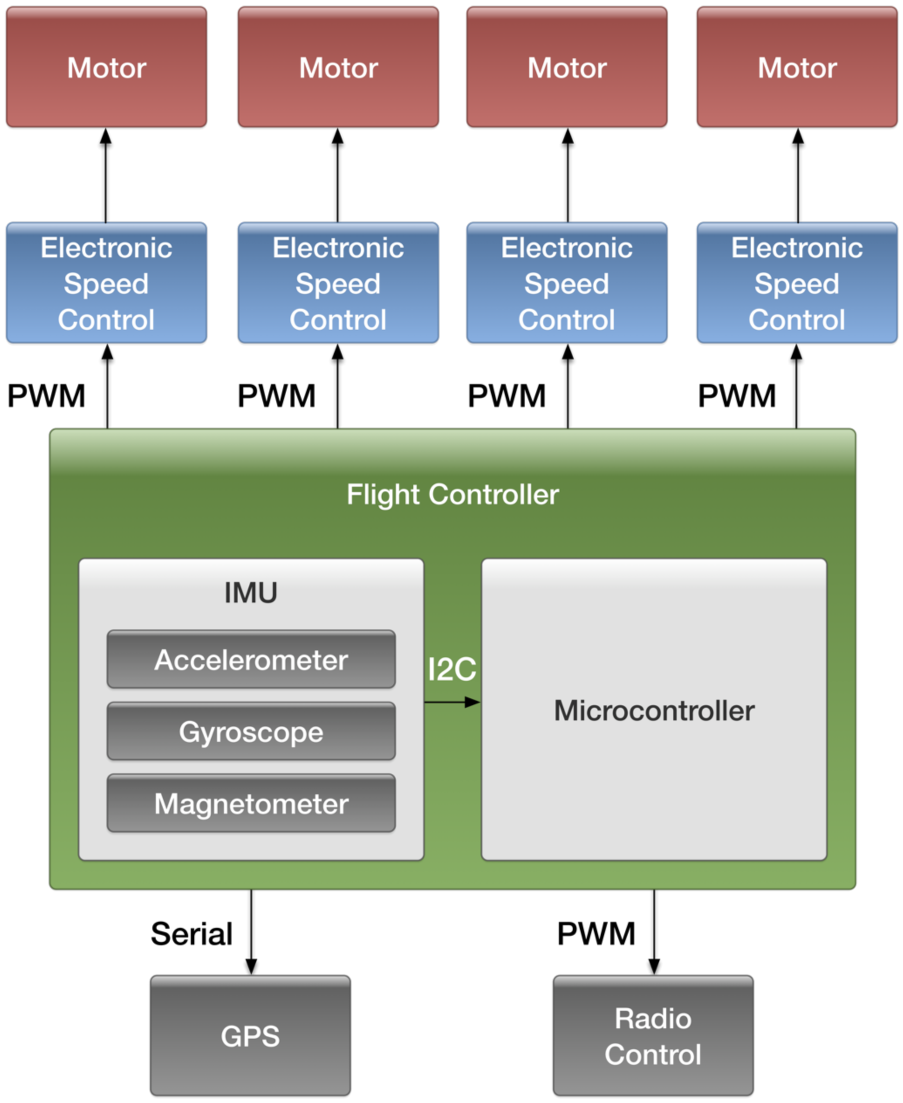


Figure 1.1: Typical components of an Unmanned Aerial Vehicle

Nowadays, all of these kinds of sensors required for inertial navigation, as well as a GPS receiver can be found inside a smartphone. This fact formed the idea of combining two of the most widespread and thriving technologies of this time, UAVs and Android smartphones that provide the necessary sensor measurements for a flight.

## 1.2 The Purpose and Contribution of this Thesis

The main purpose of this thesis is to present a reliable communication channel between the Android device and the computational platform where the autopilot software is installed, and to provide the required sensor measurements at the highest possible rate from the smartphone. The autopilot software that was selected is PenguPilot [2], an open source GNU/Linux based Multi-Rotor UAV Autopilot, due to the fact that it is a Linux based autopilot software. PenguPilot uses Linux preemption for tasks such as sensor data acquisition, data fusion, data filtering, and motor control, unlike other platforms that use a microcontroller for these functions. This fact provides to the developers the opportunity to create additional software that could perform a variety of functions during the flight. Combined with the existence of an Android device and its resources, the possibilities for UAV application development are countless. However, a major problem that came across in the developing phase of this work, is the lack of documentation of PenguPilot, for the reason that PenguPilot is open source, as mentioned above, and it is not developed for commercial reasons. PenguPilot was installed on an ODROID-XU3 Lite [3] board, a powerful heterogeneous octa-core computing device, able to run various distributions of Linux, of which, a customized image of Ubuntu 15.10 specifically designed for this board was selected. Furthermore, significant attention was given to the resource utilization, such as the utilization of the Central Processing Unit (CPU) and memory of the board computer, as it was of vital importance to guarantee that a resource overload would never occur under all possible circumstances.

The contribution of this work is twofold. Firstly, to provide an efficient and solid way of communication between two individual devices, which would form the base of even further future work on developing software designed for UAVs. Secondly, to cut down the need for external sensors, such as an IMU, a GPS receiver, and a barometer, by using the integrated sensors of an Android smartphone. With the integration of an Android smartphone into an existing autopilot software, new opportunities arise for developing unified solutions that could exploit both the hardware of the Android smartphone and the hardware of the UAV. Possible unified applications could include flight planning, video recording and even rescue from emergency situations, like project SaveME of the SenseLab research team of the Technical

University of Crete. Team SenseLab reached the third place of the UAE Drones for Good international competition in 2016 with project SaveME, among more than 1000 participants from 165 countries.

## 1.3 Thesis Outline

In this thesis, the stages of developing the Android application and all other required functionality for the autopilot software will be analyzed, and more details about the used techniques for the development will be provided. After this short introduction five more chapters will follow, each of which will refer to the different stages of the development and the implementation of the Android application and the software that coupled with it from the side of the autopilot software. More specifically, the second Chapter contains a brief review of related work, fields that they cover and a comparison of them to the work done in this thesis. In the third Chapter, a more detailed description of the system, as well as the requirements, the restrictions and the modelling of the solution that lead to the development of the Android application and the interfacing software for the side of the flight controller is presented. The fourth Chapter describes the stage of designing, the development methodology that was used and the complete system architecture, explaining the method of communication between the two devices. The results of this work are presented in the fifth Chapter, along with their evaluation. Eventually, the sixth Chapter concludes the thesis and contains some suggestions for possible future work.

# Chapter 2

# Related Work

## 2.1 Outline

High precision sensors, a great amount of processing power and powerful enough motors are required for any quadcopter, components that were made available widely for commercial use in the last three years. As the technology required to build a UAV became cheaper, a lot of work has been done in developing UAV software and hardware, so much for commercial reasons as well as projects from individuals. In this chapter, the focus is given to applications that utilize Android devices for their sensors in some part of the UAV platform, and eventually, every work is reviewed and compared to the work done in this thesis.

## 2.2 Other Unmanned Aerial Vehicles Using Android Devices

### 2.2.1 AndroCopter

AndroCopter is an experimental project by Romain Baud, which aims to use an Android smartphone as the on-board flight computer, along with an Arduino board that links it with other electronics, to achieve an autonomous quadcopter. The idea behind this project is to take advantage of the existing sensors of the smartphone, such as the accelerometer, the gyroscope, the magnetometer, the GPS receiver and the barometer, as well as the 3G and Wi-Fi network for the communication with the ground station, and the two cameras of the device for capturing pictures of the flight. AndroCopter also depends heavily on the existence of a significant amount of processing power and storage space for media, such as pictures, videos and flight data. Currently, there are not any autonomous flight modes, but only manual flying, using a controller, and the capability of capturing aerial pictures. The next goals of this project are to implement a mode of waypoint navigation that utilizes the GPS receiver and the barometer, indoor horizontal stabilization using the optic flow of the rear camera of the smartphone, and a first person view of the flight using the front camera and a mirror. [4]

### 2.2.2 PhoneDrone Ethos

PhoneDrone Ethos is a commercial product currently under development by xCraft, which allows users to deploy their smartphones as an autonomous aerial camera, using the sensors, processor, and wireless capability of the device. There are many similarities between PhoneDrone Ethos and AndroCopter that was mentioned in section 2.2, as they share the same principle of operation, however the first is compatible with both Android and iOS devices, and has many more features that are missing from the latter, such as live video streaming and control from another mobile device, a camera mirror that can provide three different views and phone protection. The company claims a flight time of 15-20 minutes, speeds of 0-35 miles per hour and that PhoneDrone Ethos is compatible with many low cost devices. PhoneDrone Ethos is available for pre-order from the official website of xCraft at the price of $299.00 and it is scheduled to ship to the first customers in September of 2016. Also, xCraft is going to release the PhoneDrone Controller Application Programming Interface (API) to the market, allowing developers to integrate UAV capabilities into their applications or to create new applications based on this technology. [5]

### 2.2.3 Smartphones Power Flying Robots

GRASP Laboratory of the University of Pennsylvania, in partnership with Qualcomm Research presented a smartphone-controlled quadcopter, in which all of the sensing, sensor fusion, control, and planning are done on an Android device. The components of the drone include only an Android device, four motors, and a motor controller. This research project features a quadcopter capable of autonomous flight, which uses information from the sensor measurements and the camera of the smartphone to estimate its pose without the use of a GPS. All software for sensor fusion, control, and planning for an autonomous flight was developed at the University of Pennsylvania, while the specialized Qualcomm vision-processing software that was used on the smartphone for computer vision, was developed by Qualcomm Research. A companion mobile application for the control of the drone was also used. [6]

### 2.2.4 SmartCopter

SmartCopter is a project of the Vienna University of Technology in the area of Virtual and Augmented Reality, which combines several research activities in order to design and develop a low weight and low cost UAV for autonomous flight and navigation in GPS-denied environments using a smartphone as its core on-board processing unit. This approach is based on open source software and hardware, is independent of any additional ground hardware and the Android device can be easily replaced if more powerful hardware is needed. The smartphone uses sensor fusion along with computer vision in order to map, locate and navigate in an unknown indoor environment, and is connected to an open hardware microcontroller that is responsible for the motor control. The software developed for mapping and localization does not require any GPS coverage of the area and this is how autonomous indoor navigation is possible. [7]

### 2.2.5 Flone 3.0

After the first release of the award winning Flone 1.0 in 2013 by aeracoop, a new release was recently presented. Flone 3.0 is an Open Source Drone Ecosystem, according to aeracoop, and they describe it as a powerful, compact and smart design of a low cost quadcopter, which is able to carry a smartphone or a camera. Unlike the former three works that were presented, Flone 3.0 does not utilize an Android device for inertial navigation but it uses it for controlling the quadcopter using the sensor measurements of the smartphone. In contrast to other Android applications that function as virtual joysticks using the screen of the smartphone, Flone app uses the inclination of the smartphone for pitch, roll, and yaw control, with only the throttle being controlled by a slider on the screen. There is also a detailed guide with instructions on how to build a Flone 3.0 quadcopter, as the project is open source and available to anyone who is interested in it. [8]

## 2.3 Comparison to this Work

While enough approaches exist for the cooperation of an Android device and a UAV, only a few of them have the capability of autonomous flight and some of them do not utilize the Android device for the inertial navigation of the UAV.

AndroCopter is definitely a very ambitious project with great potential, from the aspect of the compactness of the platform, as it uses only an Android smartphone and an Arduino board, in contrast to the platform that will be presented in this thesis, that uses an Android phone, an additional flight computer containing the autopilot software, and an Arduino board that is necessary for the four motors and the radio control. On the other hand, this approach is limited from the aspect of expandability, as there is no room for additional devices, such as cameras and telemetry, due to the lack of an extension port. Adequate attention is also required to the resource utilization of the smartphone used each time, as every device has different hardware specifications, and there is always the possibility of insufficient processing power and storage space.

PhoneDrone Ethos is a ready to fly product that works with low cost phones too and has no need for any assembly before use, a big advantage if the intended use is just for recreation, and although it is a complete solution that combines a smartphone and a flight controller, no technical information about the hardware used is available to public from xCraft, and the software is not open source. From these facts, it is obvious that it is not possible for developers to customise PhoneDrone Ethos, by adding any additional devices such as sensors or cameras, but they are able only to create applications for the smartphones that cooperate with it. Thus, this product is not really designed for developers, comparatively to the work done in this thesis and to other UAV packages that are currently out on the market and are open source.

The project of the GRASP Laboratory of the University of Pennsylvania uses a different approach to autonomous flying, which focuses on computer vision and inertial navigation, which is a very sophisticated solution. Beside the previous information, very little information is available to the public by the University of Pennsylvania about this work. On the other hand, the complete lack of using a GPS receiver and by using only the IMU and the camera of the smartphone makes it very difficult and computationally complex to determine the current location of the UAV. In the work done in this thesis all sensors of the smartphone are utilized, including the GPS receiver, however, there is no usage of the camera for computer vision applications, in contrast to the project of the University of Pennsylvania.

Unlike the work done in this thesis, SmartCopter utilizes an external IMU rather than the sensors of the Android device, with only the camera of the smartphone used for the localization and the mapping of the area. Additionally, all of the processing is conducted on the Android device, in contrast to this work, which all of the processing is done in the flight controller board. Another important difference is that SmartCopter is intended for autonomous indoor use while the work in this thesis is designed for autonomous outdoor use. Therefore, this approach appears opposite to the approach presented in this thesis.

Obviously, Flone 3.0 is utilizing an Android device for a completely different reason than using it as an IMU, and also, Flone 3.0 has no autonomous flight modes yet, in contrast to the software that is used in this work, PenguPilot, which offers many autopilot functions. There are plans for an autonomous flight mode in the future by aeracoop, a fact that makes Flone 3.0 a very promising project for the UAV community.

Firstly, it is of great importance that the work in this thesis is based on an open source software, PenguPilot, from the aspect that all of the source code can be modified to the desired needs and requirements, and that the software is distributed free of charge. However, the main reason that PenguPilot was selected is that it is Linux based software, which implements functions for autonomous flight, in contrast to all other approaches which use a microcontroller as a flight controller. Using a Linux platform as a flight controller rise endless possibilities for developing software that would perform a variety of tasks on the UAV during the flight. Taking this idea one step further, in this thesis an Android device is added to the UAV platform, creating a communication channel and utilizing its sensors, and leaving the possibility open that in the future other resources of the device, such as the cameras, the processing power, and other sensors, will be used too.

# Chapter 3

# Modelling

## 3.1 Outline

This chapter contains a brief introduction to embedded systems and a more detailed description of the Odroid platform, the autopilot software, and the Android device. The modelling of the system is presented, along with its requirements and limitations that lead to the development of this work.

## 3.2 Embedded Systems

### 3.2.1 Introduction to Embedded Systems

According to P. Marwedel [9], embedded systems are information processing systems embedded into enclosing products. Embedded systems can be found commonly in cars, industry, telecommunication equipment, aviation systems, consumer electronics, medical devices, and military applications, and as in this work, in UAVs as well. For example, automobiles use embedded systems to maximize their safety and their efficiency, as well as to reduce pollution. Telecommunication systems utilize embedded systems in cell phones, telephone switches, routers and network bridges. Consumer electronics include a wide range of devices that use embedded systems, such as printers, cameras, videogame consoles, washing machines, microwave ovens and air conditioners. Aviation systems contain inertial navigation systems, GPS receivers, electric motors and motor controllers. Medical devices employ embedded systems in medical imaging systems, electronic stethoscopes, and vital signs monitoring.

Most modern embedded systems are based on microcontrollers and in some cases on microprocessors in more complex systems. The processor used may be types ranging from general purpose to more specialised hardware, such as digital signal processors.

Design engineers can optimize embedded systems in order to reduce the size and the production cost and increase the systems reliability and performance for specific tasks. For instance, some embedded systems may have low performance requirements, allowing the system hardware to be simplified in order to reduce costs.

### 3.2.2 Characteristics of Embedded Systems

In contrast to general purpose computers, embedded systems are usually small sized, low cost and they have a low power consumption and the ability to operate in a wide range of environmental conditions. Furthermore, embedded systems are designed to perform only a specific task, in contrast to a general purpose computer that can perform a variety of tasks. However, embedded systems have limited processing resources, a fact that renders them more difficult to program and to interact with.

Embedded systems are based either on microprocessors that use separate integrated circuits for memory and peripherals, or on microcontrollers that have all of their peripherals on chip, and have a smaller size, power consumption and cost. Many embedded systems consist of smaller components within a larger device that has a more general purpose. For example, an embedded system in a car performs a specific function, such as measuring the engine temperature, but the overall purpose of the car is to transport people. Also, many embedded systems do not use common input and output peripherals, such as keyboards and monitors, instead, they use push buttons, sensors, steering wheels etc. Modern embedded systems avoid the use of components with moving parts, such as hard disks, that are less reliable than solid state components such as a flash memory. The software of an embedded system is referred to as firmware, it is stored in the flash memory of the system.

A common characteristic of embedded systems is that they have to meet their real-time constraints, as failing to complete computations within a given time frame can result in either loss of the quality of the system, or even in causing harm to the user. According to H. Kopetz, a time constraint is called hard if not meeting that constraint could result in a catastrophe, while all other time constraints are called soft time constraints. [10]

All embedded systems have to be dependable, from the aspect of reliability, maintainability, availability, safety and security, as these systems are directly connected to the physical environment and have an immediate impact on it through sensors and actuators. Embedded systems also have to be efficient, from the aspect of energy consumption, runtime efficiency, code size, weight, and cost. Embedded systems are typically in continual interaction with their environment and execute at a pace determined by that environment. Such systems are called reactive systems.

Furthermore, embedded systems may not be physically accessible, as regularly they can be found in machines that are expected to run continuously for long times, even for years, without errors. In some cases, these systems have to be able to recover themselves in the case of corruption or data loss. For this reason, the software is developed and tested more thoroughly than the software for personal computers and as mentioned before, moving parts such as hard disk drives, buttons or switches are avoided. Some examples of these systems are:

- Systems that cannot be shut down due to a significant loss of money, such as factory controls, automated sales and service, telephone switches, funds transfer and market making.
- Systems that are inaccessible to repair, or cannot safely be shut down for repair, such as buoys, remote sensing systems and space systems.
- Systems that have to be kept running for safety reasons, such as aviation systems and nuclear reactor control systems.

In order to recover from errors such as software bugs and soft errors in the hardware, a variety of techniques are used. The most common technique is a watchdog timer, a component that is responsible for resetting the system in specific time intervals measured by an internal timer, unless the software notifies the watchdog. Other techniques include subsystems that can temporary undertake the function of the main system if an error occurs, special software modes that provide partial function and following specific design principles when designing and programming the system, ensuring a highly reliable and secure system.

### 3.2.3 ODROID-XU3 Lite

Odroid is a series of single-board computers, manufactured by Hardkernel, an open source hardware company in South Korea. Its name derives from “open” and “Android”, and is a development platform for hardware as well as software. However, the hardware is not fully open source, due to the fact that some parts of the design are retained by Hardkernel.

ODROID-XU3 Lite is a heterogeneous multicore platform, with a powerful Samsung Exynos 5422 quad core Cortex-A15 at 1.8 GHz for heavier tasks, and a quad core Cortex-A7 at 1.3 GHz for lighter tasks. This provides improved processing capabilities, while maintaining the most efficient power consumption possible. This heterogeneous multiprocessing solution can utilize a maximum of all eight cores to manage computationally intensive tasks. It also features a 2 GByte RAM, one USB 3.0 port, one micro USB 3.0, four USB 2.0 ports, an HDMI display port, Ethernet with RJ-45 jack and audio with a 3.5mm jack or via the HDMI digital output. The storage of the system can be either an eMMC flash storage or a micro SD card, as used in this case. Odroid offers open source support and Linux images customised specifically for this board. A variety of Linux distributions, including Ubuntu 15.10 that was used in this case, and Android can be installed as well. The availability of many serial ports allows the possibility for developing software that utilize external devices, such as cameras, sensors and even other computers, such as the Android smartphone that is used in this work.

More details about the ODROID-XU3 Lite, including its schematics, can be found on the official website of Odroid. However, ODROID-XU3 Lite has been replaced by the ODROID XU4, which is cheaper and has even better performance than its predecessor. [3]

## 3.3 Autopilot Software

The autopilot software that was selected for the implementation of this thesis, PenguPilot, was briefly described in the previous chapters. Currently, there are two versions of PenguPilot available, a “master” and a “new generation – work in progress” release, which has features that are not included in the former version, therefore, the latter was selected. However, as its name indicates, this version is still in development and is not considered as stable as the “master” release. In this section, a more detailed description will be provided about its features, components and way of operation.

### 3.3.1 Overview of PenguPilot

As also mentioned in the previous chapters, PenguPilot is an open source autopilot software, which is Linux based and the whole flight infrastructure is based on Linux preemptive user-space tasks. This gives it the ability to perform tasks in real-time that usually needed a microcontroller, such as sensor readings, sensor fusion, data filtering and motor control.

A significant advantage of PenguPilot, compared to other approaches, is the architecture of the software that is component based. This allows code execution on demand, and in that way power is saved in standby mode. Also, PenguPilot provides memory protection among critical and non-critical components using Mutex, message-passing using ZeroMQ, MessagePack and Protobuf, process and priority management, dependency tracking, online parameter configuration that is usable for inflight parameter updates, a black box service that logs sensor measurements at each control step, sensor data replay for offline optimization and quick integration of new hardware through Linux Hardware Abstraction Layer (HAL). Additionally, PenguPilot offers platform-neutral code, which eliminates the need of low-level interrupt and timer programming. Until now, PenguPilot supports only a few platforms, as the Raspberry Pi, Gumstix Overo, and ODROID-U3, but the goal of the developers is to create a true platform-independent, Linux based autopilot.

A very useful feature of PenguPilot is the Secure Shell (SSH) based user interface for configuration and calibration, due to the fact that it was installed on an ODROID-XU3 Lite board, an also compatible board similar to ODROID-U3, and all the work was done headless, exclusively via SSH. The fact that PenguPilot is designed to use symmetric multiprocessing allows a better utilization of the multicore ODROID-XU3 Lite. Finally, there is also support for additional Universal Serial Bus (USB) devices, such as cameras and Wi-Fi sticks, in contrast to other approaches of autopilot platforms.

The system architecture is similar to a typical architecture of a UAV, which was presented in Figure 1.1, with the difference that in this case there is an additional input - output shield on the top of the flight controller. This shield only parses the radio control Pulse Position Modulation (PPM) signals and sends them to the flight controller via a serial port and reads from the same port the output signals generated from the flight controller for the motors and send the PWM signal to the four ESCs. The flight controller is performing all the sensor readings from the IMU, data fusion, and data filtering and calculates the signals for the motors. The IMU is connected via an Inter-Integrated Circuit ($I^2C$) bus to the flight controller, the GPS receiver is connected via a USB serial port and other devices such as Wi-Fi and cameras can be connected via a USB serial port as well. The architecture of the system is shown in Figure 3.1.

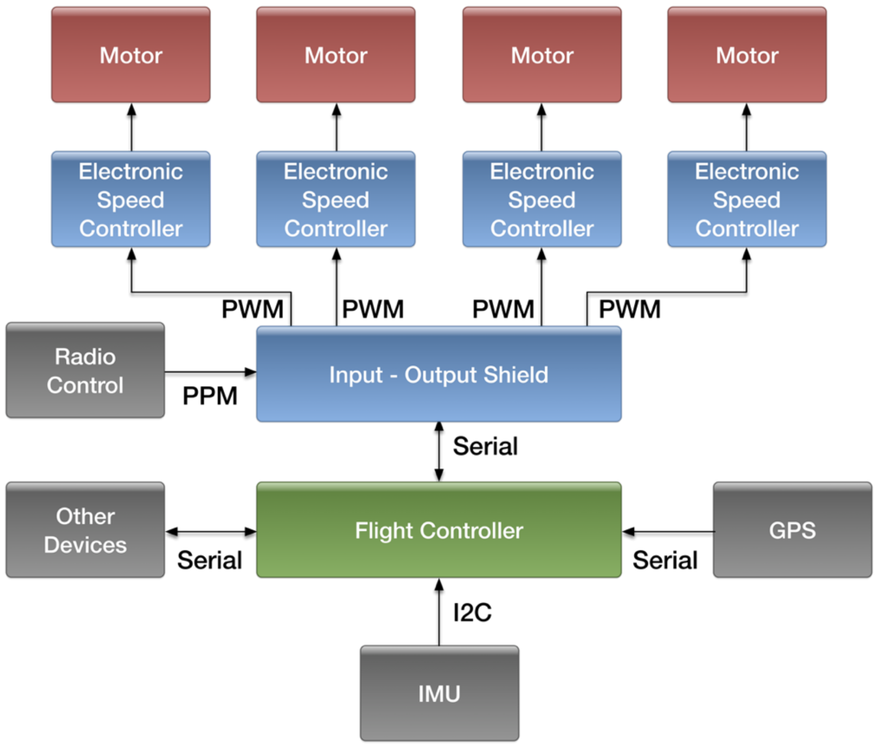


Figure 3.1: Architecture of a UAV with PenguPilot on an Odroid platform

In order to utilize an Android device instead of an IMU and a GPS receiver, the architecture of the system had to be changed. The Android device should provide reliable sensor measurements, at a high enough rate, in order for meet the requirements of PenguPilot, therefore, the first step of the development was to locate the corresponding components of PenguPilot associated with the sensor and GPS data readings. After that, the requirements of PenguPilot had to be identified, such as the required types of sensors, the format of the data and the update rate. PenguPilot is comprised of services with dependencies between them that are going to be presented in the following section of this chapter. As mentioned in the introduction of this thesis, PenguPilot lacks of documentation, thus working on it was a serious challenge. Eventually, the architecture of the system that utilizes an Android device along with the autopilot software is shown in Figure 3.2. [2]

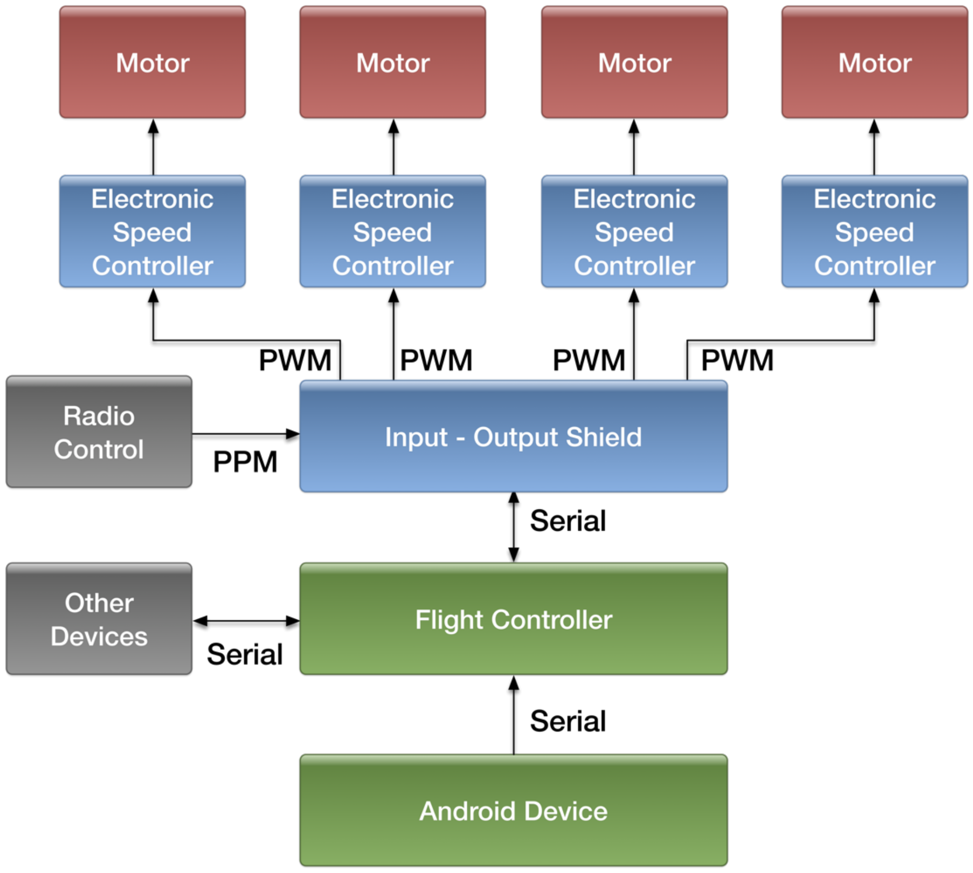


Figure 3.2: The architecture of the new system that uses an Android device

### 3.3.2 PenguPilot Services

The services of PenguPilot are structured in a hierarchical way. Higher-order services require other lower-order services to be running, for example, the service for retrieving and handling the GPS data, called “gpsp” for GPS Publisher, require the “opcd” and “log_proxy” services to be running in order to start, while “opcd” requires only the “log_proxy” service. All services and dependencies of PenguPilot are shown in Appendix A, in order to briefly present the structure of the software.

The services that were directly related to the work that had to be done in this thesis is the “gpsp” service and the “i2c_sensors”. The “gpsp” service, as mentioned before, is responsible for the management of the received GPS data, while the “i2c_sensors” service is responsible for the sensor readings. Therefore, the work in this thesis was exclusively conducted in these two services, whose individual functionality was confirmed after the development.

More specifically, the GPS publisher service originally is configured to receive data from a serial port of the platform and publish the data in order to make it available to other services, such as the autopilot. This service uses a third party library for the GPS data, the National Marine Electronics Association (NMEA) library [11], which reads the data in the American Standard Code for Information Interchange (ASCII) format, character by character, and parses them in order to extract the required information. The NMEA [12] standard allows the communication between marine electronics, such as a GPS receiver and an autopilot, via a serial port using sentences comprised of ASCII characters.

The “i2c_sensors” service implements the real-time low-level flight controller, which is responsible for the sensor readings. In the main loop of the service, depending on the hardware platform that PenguPilot is installed on, the corresponding code is executed to initialize the sensors. Then, a periodic thread that performs the reading of the sensors every 0.005 seconds, is created. This service supports a variety of $I^2C$ sensors, and a specific number of hardware platforms, such as the exynos_quad that corresponds to the ODROID-XU3 Lite. [2]

### 3.3.3 Signaling and Communication Link and Sensor Calibration

The Signaling and Communication Link (SCL) subsystem of PenguPilot is a message-based Inter-Process Communication framework that is using sockets for the distribution of the data across the components of PenguPilot.

The sockets that “gpsp” service is using are the “gps” and “sats” socket. Location information such as the longitude and the latitude of the vehicle can be found at the “gps” socket, while satellite information such as the signal strength of the available satellites can be found in the “sats” socket.

Unlike the GPS service, which has only two sockets for publishing its data, the “i2c_sensors” service uses multiple sockets, one for the raw data of every sensor and one for the calibrated values. The only exception is the barometer, which provides its measurements without any calibration. The sensor sockets that PenguPilot is using, are the following:

| | |
|---|---|
| • acc_raw: | Raw accelerometer values |
| • acc: | Accelerometer values after calibration |
| • gyro_raw: | Raw gyroscope values |
| • gyro: | Gyroscope values after calibration |
| • mag_raw: | Raw magnetic Field values |
| • mag_adc_cal: | Magnetic field values after calibration |
| • baro_raw: | Barometer values |

PenguPilot has calibration tools implemented, which are able to perform calibration to the values of the sensor measurements. The calibration of the accelerometer is performed by executing the “pp_acc_cal” command, which collects data from the accelerometer and calculates the bias and the scale of the three axes of it. After the calibration is complete, the “acc_cal” service can be started, and the calibrated accelerometer values will be published to the “acc” socket. Similarly, the magnetic field sensor can be calibrated by executing the “pp_mag_adc_cal” command. The calibrated magnetic field values will be published to the “mag_adc_cal” socket after starting the “mag_adc_cal” service. Finally, the gyroscope does not need a calibration procedure in order to calculate the calibrated values of it. After the “gyro_cal” service is started, the calibrated values of the gyroscope will be published at the “gyro” socket.

## 3.4 Android Device

### 3.4.1 Sensors, Specifications and Limitations

The Android smartphone that would pair with the flight controller had to be able to provide reliable sensor measurements the via the micro USB port of the device and accurate GPS information. In addition, the sensor measurements had to be available at a rate fast enough for the requirements of autopilot software. The device that was used during the development of the application was a Samsung Galaxy S6, with an accelerometer, a gyroscope, a magnetic field sensor, a barometer and a GPS receiver available. Fortunately, Android API [13] provides support for the NMEA data format, however with very little documentation. The coordinate system of the Android device is shown in Figure 3.3.

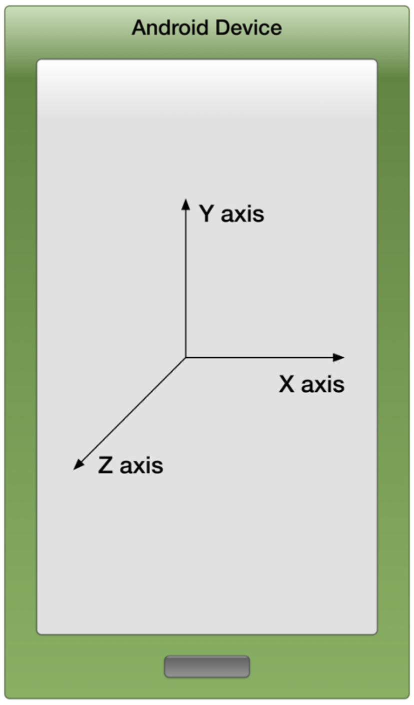


Figure 3.3: The coordinate system of the Android device

More specifically, it was found that the accelerometer and the gyroscope can provide measurements at the rate of more than 200 Hz, the magnetic field sensor at 100 Hz and the barometer at 5.5 Hz. The results that are shown in Table 3.1 occurred from calculations that were conducted in order to define the sensor rates and from the Android API functions getResolution() and getMaximumRange() in a test application, which was developed for the Android device that was used.

| | Update Rate | Resolution | Maximum Range |
|---|---|---|---|
| Accelerometer [14] | 225 Hz | 0.001197 m/s$^2$ | 39.2266 m/s$^2$ |
| Gyroscope [14] | 225 Hz | 0.001065 rad/s | 34.9066 rad/s |
| Magnetometer [15] | 102 Hz | 0.1 μT | 1200 μT |
| Barometer [16] | 5.5 Hz | 1 Pa | 1260 hPa |

Table 3.1: Samsung Galaxy S6 Sensor Specifications

However, there is a conflict about the resolution and the range of the magnetic field sensor. In the information provided by Yamaha Corporation for their product YAS537, the magnetic field sensor of Samsung Galaxy S6, it is noted that its resolution is 0.3 μT, in contrast to the result of 0.1 μT from the call of the function getResolution() in the Android test application. As well, the maximum range of the sensor is claimed to be 2000 μT by Yamaha Corporation, while the result from the call of the function getMaximumRange() in the Android test application was 1200 μT. The data for the magnetic sensor in Table 3.1 are based on the results from the Android test application that was developed, due to the fact that the information about the resolution and the maximum range of the sensor by Yamaha Corporation are marked as "preliminary" and "subject to change".

### 3.4.2 GPS Receiver

Although a GPS receiver is pointless for indoor UAVs, it is one of the most crucial components of an autonomous outdoor UAV. The most USB GPS receivers can provide an update rate of 1 Hz, while modern GPS receivers designed for UAVs can provide update rates from 5 Hz up to 20 Hz. The GPS receiver of the Android device that is used in this work is capable of providing information at 1 Hz, a rather low update rate, and an accuracy of 3 meters with a clear view of the sky. It also has to be taken into consideration the fact that the GPS receiver of the smartphone is accurate enough, but not as accurate as an expensive external GPS receiver, as there always exists trade-offs between the accuracy and the cost of the receiver. This fact also implies to the cost of an Android device, as expensive smartphones usually have a better GPS receiver and sensors, compared to other cheaper devices, from the aspect of their accuracy, update rate and measurement range. If an even higher accuracy is required, then there are two solutions, either using an external USB GPS receiver that would be connected to the flight controller via a serial port, as in Figure 3.1, or an external Bluetooth GPS receiver, connected to the Android device, that would be used instead of the integrated GPS receiver. However, the performance of the integrated GPS receiver of the Android device used is considered acceptable for an autonomous outdoor flight, compared to other GPS receivers that have similar specifications.

## 3.5 Modelling of the System

### 3.5.1 Real-time Constraints and Process Scheduling

A deadline in real-time systems is a timing milestone. As it was mentioned in the previous sections of this chapter, if a deadline is missed, the system may fail catastrophically. In the work of this thesis, the time constraints that are related to the sensor measurements are of high importance, as if these data were lost, the UAV would be uncontrollable, and unable to maintain its stability.

More specifically, as PenguPilot is software that is written in C and Python, running on a Linux operating system, there is no low-level interrupt and timer programming implementation. Instead, the processes of the autopilot software that are responsible for reading the accelerometer and the gyroscope are scheduled with a real-time period of 5 milliseconds (200 Hz) by default, while the processes for reading the magnetic field sensor and the barometer have a real-time period of 10 milliseconds (100 Hz). The service that is responsible for the sensor readings is scheduled with the highest priority among PenguPilot processes, in order to avoid missing deadlines by other non-critical processes.

According to section 3.4.1, it appears that the accelerometer, the gyroscope, and the magnetic field sensor of the Android device have a high resolution and a very wide range of measurements, in acceptable levels for a UAV flight and for the autopilot software to operate. The first two can provide a more than 200 measurements every second, while the magnetic field sensor is capable of a 100 Hz rate, which is sufficient for a UAV flight and PenguPilot.

There is a limitation on the barometer of the Android device, which lacks a higher update rate. However, even its 5.5 Hz update rate appears to be sufficient under certain circumstances. For example, supposing a maximum vertical speed of 6 m/s, according to the maximum ascending speed of a DJI Phantom 4 the fastest UAV of the top consumer UAV manufacturer currently [17], and that the pressure decreases by about 12 Pa for every meter [18], the following results occur:

$$\Delta \mathrm{P} = \frac{\text{maximum vertical speed} * \text{pressure per meter}}{\text{measurements per second}}$$

$$\Delta \mathrm{P} = \frac{6\frac{\mathrm{m}}{\mathrm{sec}} * 12\frac{\mathrm{Pa}}{\mathrm{m}}}{5.5\frac{\text{measurements}}{\mathrm{sec}}}$$

$$\Delta \mathrm{P} = 13.09\frac{\mathrm{Pa}}{\text{measurement}}$$

$$\Delta \mathrm{m} = \frac{\Delta \mathrm{P}\frac{\mathrm{Pa}}{\text{measurement}}}{12\frac{\mathrm{Pa}}{\mathrm{m}}}$$

$$\Delta \mathrm{m} = 1.091\frac{\mathrm{m}}{\text{measurement}}$$

It follows that when moving vertically with a speed of 6 m/s, every measurement will have a difference from the last measurement of 13.09 Pa, or else 1.091 m, which is a sensible resolution for an outdoor flight, considering the fact of moving with a fast vertical velocity. This deviation of almost ±0.55 m, is practically negligible outdoors at such speeds, rendering this accuracy acceptable for commercial products. Additionally, when higher accuracy is required, the UAV can always limit its vertical speed accordingly, and monitor its altitude more efficiently.

The following illustration in Figure 3.4 depicts the structure of the components where the sensor measurements have to pass through. In the next chapter of this thesis, a detailed description of the components that were developed is available, as well as a detailed illustration of them in Figure 4.12.

The Android device transmits the sensor measurements at a rate of 200 Hz. By polling at a higher rate of 1000 Hz, the software that is responsible for receiving the sensor measurements and forwarding them to PenguPilot guarantees that transmissions will not be lost under any circumstances. The new library that had to be developed in order to utilize the sensor measurements from an Android device instead of $I^2C$ sensors, is running in a loop with a period of 2.5 milliseconds (400 Hz), in order to be able to update the values of the struct as soon as possible after new sensor measurements are available. Finally, as described above, the processes of the “i2c_sensors” that read the sensor measurements are running with the default periods set by PenguPilot, at 200 Hz for the accelerometer and gyroscope, and 100 Hz for the magnetic field sensor and the barometer. The priority of the last two components, which are processes related to the task of reading the sensors measurements, is set at the highest among all other non-critical processes of PenguPilot.

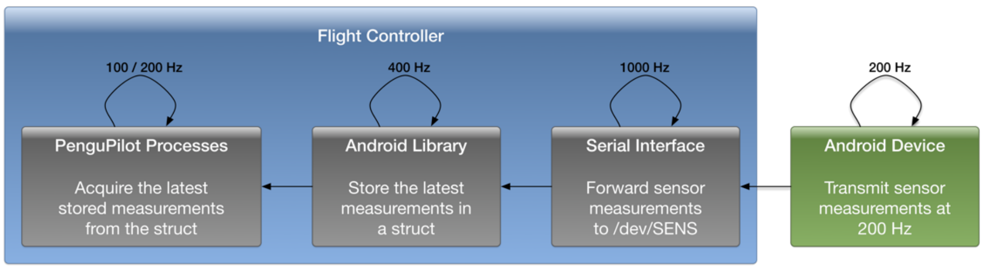

Flight Controller
100 / 200 Hz
400 Hz
1000 Hz
200 Hz
PenguPilot Processes
Acquire the latest stored measurements from the struct
Android Library
Store the latest measurements in a struct
Serial Interface
Forward sensor measurements to /dev/SENS
Android Device
Transmit sensor measurements at 200 Hz

Figure 3.4: The sensor measurements through the various components of the system

The scheduling polices of the processes that read the values of the accelerometer, of the gyroscope, of the magnetic field sensor and of the barometer, and distribute them to other services of PenguPilot that need them, have been chosen carefully by the developers of PenguPilot. Therefore, they did not require any tests in order to confirm their functionality, as well the testing of the autopilot software, which was developed by others, is not a part of this thesis. On the other hand, the scheduling of the newly created processes had to be based on the needs, as well as the performance capabilities of the system, and their appropriate functionality had to be verified.

In order to avoid of excess CPU usage, and therefore any possible misses of deadlines, when the processes of the serial interface and of the newly implemented library are not performing any task, are being put to sleep. The processes of receiving the data from the serial port and forwards them to the appropriate components of PenguPilot is called with a frequency of 1000 Hz, therefore it has to be able to complete all of its computations within a time window of 1 ms. Similarly, the process that is responsible for reading the sensor measurements and storing them into a defined struct is called with a frequency of 400 Hz. Due to the multiple memory accesses, one iteration takes more time than an iteration in the process that receives and forwards the data, therefore a bigger time window had to be available in order to guarantee that no deadlines will be missed.

It was found that the process of receiving data from the Android device and forwarding them to the appropriate components of PenguPilot need an amount of time ranging from 41 μs up to 255 μs under extreme circumstances, with an average time of 94 μs, which is almost 10 times less than the 1 ms time window of the deadline of the process. However, even in the worst case scenario, the deadline of the 1 ms time window is always met. The process that is responsible for reading the sensor measurements and store them in a struct need an average of 429 μs for one iteration, with a minimum time of 89 μs, and a maximum time of 1.718 ms under extreme circumstances. Again, the deadline of 2.5 ms is met, and no measurements are lost in any case.

In Figure 3.5, the time scheduling for the processes that were developed for the integration of an Android device is shown. It has to be noted that the time scheduling of the processes that were developed by others and are a part of PenguPilot, were left intact, without any modifications, in order to avoid any issues that would degrade the performance of the autopilot software.

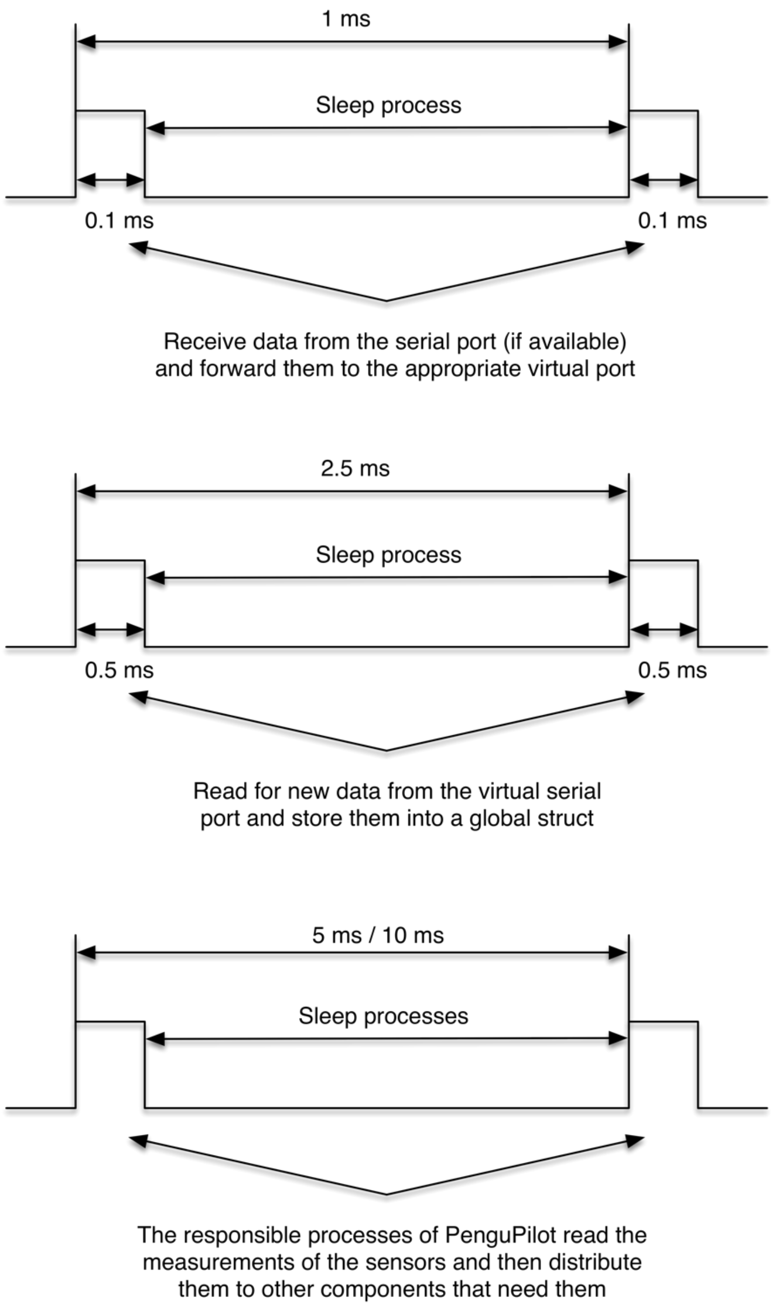

1 ms
Sleep process
0.1 ms
0.1 ms
Receive data from the serial port (if available) and forward them to the appropriate virtual port
2.5 ms
Sleep process
0.5 ms
0.5 ms
Read for new data from the virtual serial port and store them into a global struct
5 ms / 10 ms
Sleep processes
The responsible processes of PenguPilot read the measurements of the sensors and then distribute them to other components that need them

Figure 3.5: The time scheduling of the processes that handle sensor data

However, since the reading of the accelerometer and gyroscope by PenguPilot is conducted at a rate of 200 Hz, even missing a single measurement would not result into a disaster, as the values of the next measurements are going to be close to the ones in the lost transmission. An example of continuous accelerometer and gyroscope measurements during an upward movement of the device is shown in Table 3.2.

| Measurements | X Axis | Y Axis | Z Axis |
|---|---|---|---|
| k | -1.936909 | -1.980005 | 19.942503 |
| k + 1 | -1.975216 | -1.802834 | 20.390219 |
| k + 2 | -1.866280 | -1.646014 | 20.627245 |
| k + 3 | -1.491588 | -1.546654 | 20.730196 |
| k + 4 | -1.089362 | -1.484405 | 20.616470 |
| k + 5 | -0.871489 | -1.440112 | 20.235792 |
| k + 6 | -0.818817 | -1.466449 | 19.675550 |

Table 3.2: Example of continuous accelerometer measurements

As it is apparent, if one measurement is lost, the next measurement can compensate it, due to the fact that measurements are acquired at a rate of 200 Hz, therefore the loss of information from a single missed measurement would be insignificant. The same applies to the gyroscope and magnetic field sensor, as both of these sensors have a high update rate, and a possible loss of one measurement would not affect the performance of the system. The barometer has a significantly lower update rate, which means that each measurement is going to be read repeatedly for an average of 18 times, as it is read 100 times per second but its update rate is only 5.5 times per second.

Furthermore, since the system is an octa-core platform, with four cores clocked at 1.8 GHz and another four cores clocked at 1.3 GHz, the CPU usage by the running processes is being split among the eight cores of the flight controller, and no deadlines are missed under any circumstances. The detailed resource utilization of the system is described in the fifth chapter of this thesis.

# Chapter 4

# Design and System Architecture

## 4.1 Outline

In this chapter, the development methodology that was used is described, as well as the complete system architecture and the steps in the designing phase. The stages of the Android application development are presented in section 4.3. In the last section of the chapter, the development of the appropriate software for the Odroid platform is presented, along with the utilities that were used, and the modifications that were performed at PenguPilot in order to utilize the received sensor measurements and the GPS information from a serial port.

## 4.2 Agile Software Development

### 4.2.1 Introduction to Agile Software Development

The development of the Android application, as well as the interfacing software that was integrated with the autopilot, was done based on Agile Software Development [19] principles. Agile software development is preferred to the Waterfall model, due to the fact that in the latter the system has to be fully specified since the beginning of the development. In contrast to the waterfall model, agile is based on evolutionary development, adaptive planning, early delivery and continuous improvement, and also encourages rapid and flexible response to changes.

In the waterfall model the specifications and the requirements of the system are defined in the stage of evaluation which is first, and they are constant in a great degree, as the phases of analysis, designing, development and testing in the waterfall model are discrete. This approach is considered risky, due to the fact that the developer cannot test the design or the architecture until the late phase of the project, and cannot be sure if the functionality is the desired until it is too late to make any changes, as the waterfall model cannot handle any changes. Furthermore, in this approach, there is always a chance of delivering a low quality product, for the reason that when the time is running out, the phase of testing that is the last phase, will be limited.

As shown in Figure 4.1, agile treats analysis, designing, development and testing as continuous activities. In this way, the risk is reduced because the customers can provide feedback early, and changes can be made without excessive costs. This approach guarantees improved quality as well, due to the fact that the phase of testing starts from the beginning.

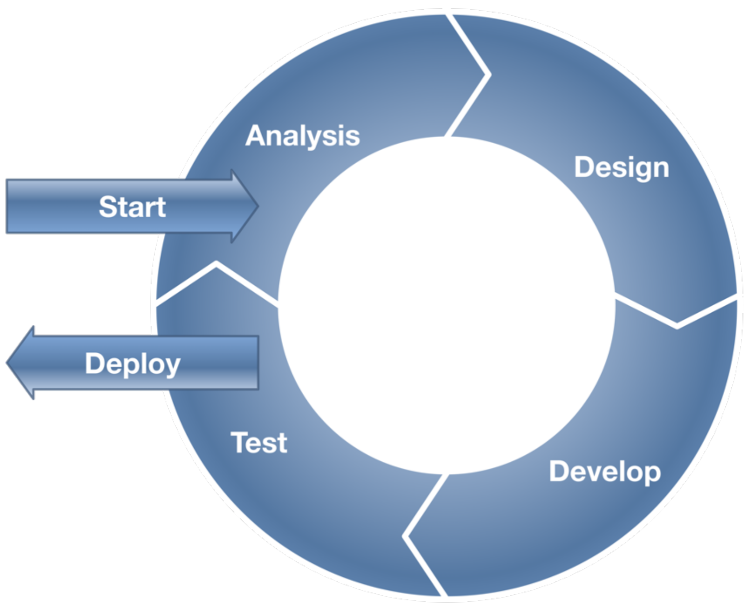


Figure 4.1: Agile Software Development

### 4.2.2 Adaptive Software Development

There are many different methodologies for agile software development, such as the Scrum, which is the most commonly used methodology, the Extreme

Programming, the Adaptive Software Development and the Feature Driven Development. Every agile methodology has its individual approach, however, all of them share the same principles that are defined in the Agile Manifesto [20]. The methodology that was used in this work was a form of the Adaptive Software Development, which is based on the principle that continuous adaption of the process to the work at hand is the normal state of affairs [21]. As all of the agile methodologies, the adaptive software development is iterative, mission focused, feature based and change tolerant.

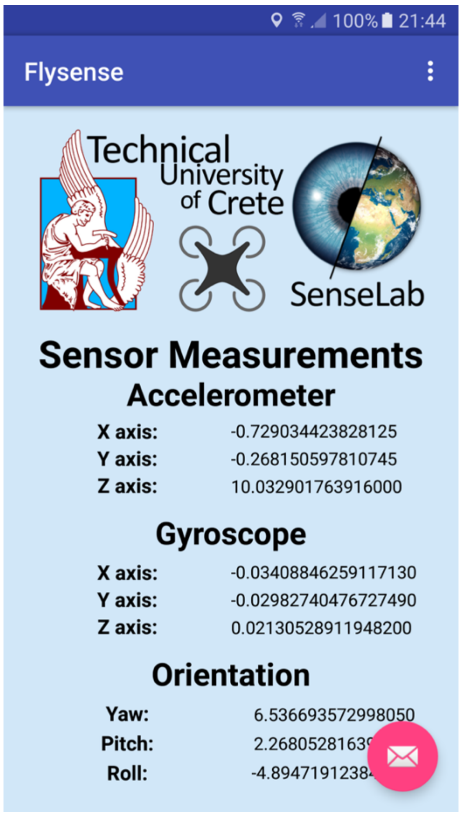
At first, the requirements of the final product had to be specified. After some meetings with the thesis supervisor and the members of the SenseLab team, the necessary functionality of the work that had to be done was defined. As mentioned in the previous chapters, the aim was to create a reliable communication between an Android device and the Odroid platform, and utilize the sensors of the smartphone for the inertial navigation of the UAV. Weekly meetings with the supervisor and the SenseLab team members made the completion of this work possible. In every meeting, the progress of the work, problems, and possible solutions were discussed. In that way, the Android application and the interfacing software for the autopilot were developed based on the needs of the autopilot software as well as the available hardware of the Android device and the Odroid platform.

As a basic principle of the Agile Software Development, at first, a small section of the complete functionality was created, and later more features were added. The Android application was created step-by-step, beginning only with some basic functionality of the sensors. Initially, a simple application was developed that displayed the measurements of the accelerometer, the gyroscope, and the orientation sensor on the screen of the smartphone, as shown in Figure 4.2. The next features that were added in every iteration of the agile development cycle are:

- Data retrieval from the GPS receiver
- Measurement of the maximum update rates of the sensors
- NMEA sentences retrieval
- Development of the serial interface
- Replacement of the orientation sensor with a geomagnetic field sensor
- Addition of barometer and a text input field for entering temperature

- Improvement of the application and the serial communication

In this way, in every iteration of the agile development cycle, the Android application was tested and the added functionality was confirmed before proceeding to the addition of another feature. More details about the features and the development of the Android application will be provided in the following section of this chapter.



Sensor Measurements

Accelerometer

| | |
|---|---|
| X axis: | -0.729034423828125 |
| Y axis: | -0.268150597810745 |
| Z axis: | 10.032901763916000 |

Gyroscope

| | |
|---|---|
| X axis: | -0.03408846259117130 |
| Y axis: | -0.02982740476727490 |
| Z axis: | 0.02130528911948200 |

Orientation

| | |
|---|---|
| Yaw: | 6.536693572998050 |
| Pitch: | 2.2680528163 |
| Roll: | -4.894719123 |

Figure 4.2: The first stage of the development of the Android application

Similarly, the interfacing software that was created for the Odroid platform was developed step-by-step, beginning with the establishment of the serial communication, and improvements, as well as functionality were added in the next iterations of the agile development cycles. Improvements were made on the performance of the software in order to consume fewer resources of the Odroid platform, such as CPU and memory, and functionality was added, such as a data handler that is responsible for the distribution of the sensor measurements to the appropriate component of the autopilot software. More details about this software will follow in section 4.4 of this chapter.

The result from working with an adaptive software development methodology was the creation of a high quality application, as well as a reliable serial communication software that was used as a linked library into the existing PenguPilot software.

## 4.3 Android application

### 4.3.1 The Purpose of the Android Application and the Activity Lifecycle

Before the development of the application, the fundamentals and the components of the Android application development had to be understood. The book that contributed to the understanding of these, is the "Head First Android Development", by Dawn & David Griffiths [22]. Furthermore, for more complex issues, such as the serial communication and the NMEA data, knowledge was extracted by the Android developers [13] and other independent sources.

In the first stages of the Application development, before the implementation of the serial communication, the graphical user interface contained many text fields in order to verify the functionality of the sensors. However, all of these unnecessary fields were removed in the last stage of the application development in order to make the application as lightweight as possible.

The purpose of this Android application is to transmit the sensor measurements and the received GPS data to the Odroid platform. It is crucial that the application is using the minimum resources required in order to perform optimally. For this reason, the final graphical user interface is minimal, containing only the necessary buttons. Therefore, the application consists of a single activity with only the required functionality. The lifecycle of an Android activity can be seen in Figure 4.3.

The onCreate() and onDestroy() methods are called only once in the lifetime of an activity, when the activity starts its life and when the activity finishes its life cycle respectively. All setup of the global state of the activity is conducted in the onCreate() method, while the release of all remaining resources is conducted in the onDestroy() method.

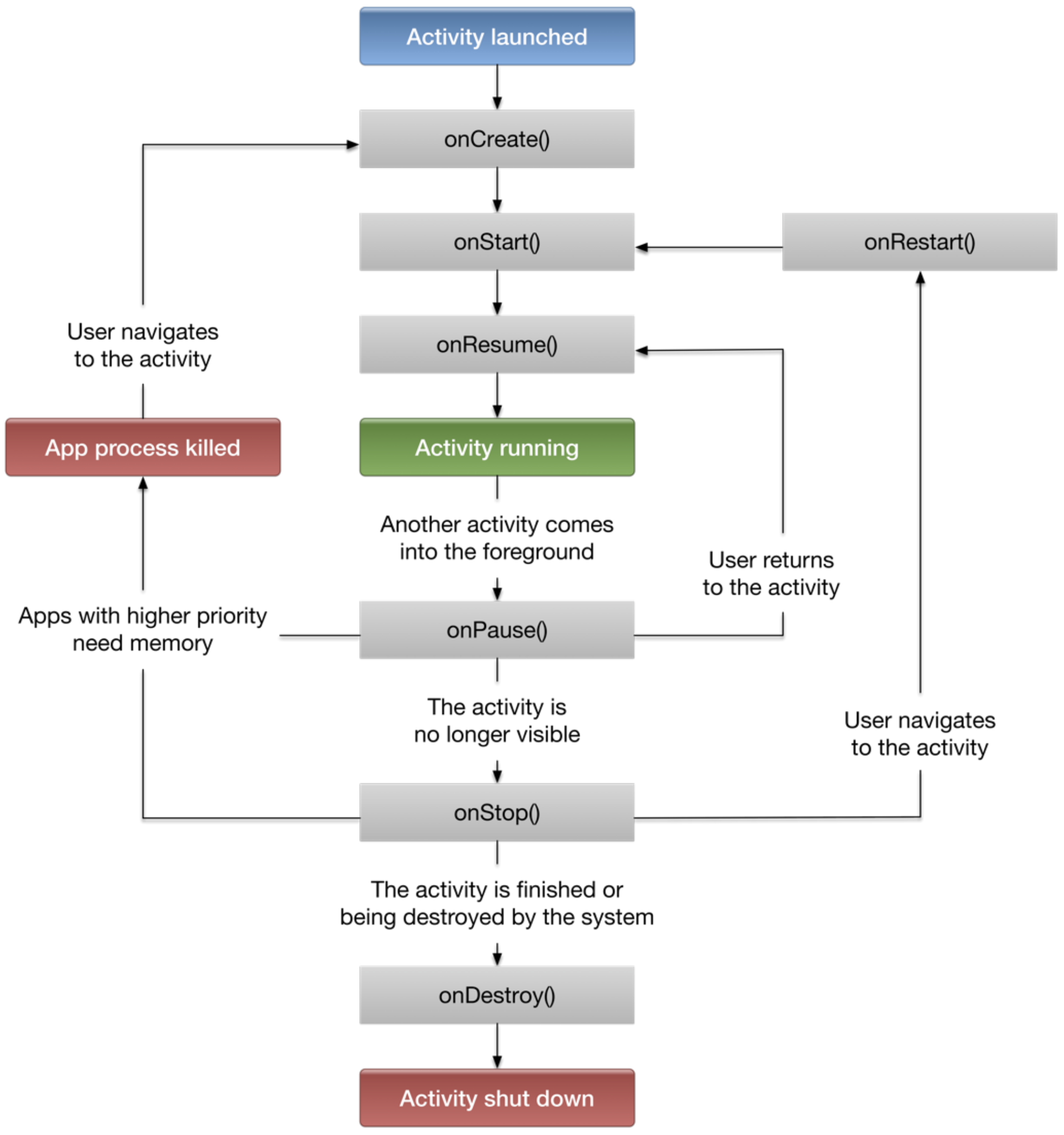

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finished or being destroyed by the system
onDestroy()
Activity shut down

Figure 4.3: The Android activity life cycle

The application is designed in a way that is able to run continuously, even when the screen is locked or when the application is in the background, for example if the phone is ringing from an incoming call. To achieve this, a wake lock mechanism is used in order to indicate that the application needs to have the device stay on. [23] Using a partial wake lock, it is ensured that the CPU will continue to be running even if the screen and keyboard backlight go off. The wake lock mechanism is implemented by acquiring the wake lock in the onCreate() method and by releasing it in the onDestroy() method. This guarantees that in the whole lifetime of the application the CPU will stay on and will run the application.

The application can be terminated only by the user, or by the Android device in the case of the system running low on memory. Unless the Android device is rooted, there is no way of avoiding the termination of a process by the system when it is running low on memory, however, this scenario is unlikely to happen, due to the low CPU and memory usage of the Android application. More details about the reliability of the application will be provided in the fifth chapter of this thesis.

### 4.3.2 Acquiring Data from the Sensors

As it was mentioned in the previous section of this chapter, the Android application was developed gradually, along with the development of the interfacing software that was on Odroid platform. References about sensor programming were found at the Sensors Overview website of the Android Developers [24]. Furthermore, the Motion Sensors website [25] was referenced for the accelerometer and gyroscope, and the Position Sensors website [26] for the orientation sensor. However, the orientation sensor that was used initially was replaced later by the geomagnetic field sensor, due to the fact that the orientation sensor was deprecated in Android 2.2, and that PenguPilot is designed to use a geomagnetic field sensor instead of an orientation sensor. Both the orientation sensor and the geomagnetic field sensor utilize the same hardware resources, the magnetic field sensor. Their only difference is the way that they calculate their output.

In the last stages of the development of the Android application, when it was found that PenguPilot can utilize a barometer and a temperature sensor, the functionality of the barometer of the Android device was added to the application. However, on the Android device that was used, a temperature sensor is not available unfortunately. For that reason, in order to provide at least some indicative data to PenguPilot, a simple text field was added at the application that allows the user to manually input the environment temperature. References for the barometer were found at the Environment Sensors website of the Android Developers [27].

In the first stage, of the development, as shown in Figure 4.2, only the accelerometer, the gyroscope, and the orientation sensor were used, and their measurements were displayed on the screen of the Android device. Before creating the sensor instances in the onCreate() method, a check is conducted in order to identify the available sensors of the device. If a device lacks the required sensors, a message appears on the screen of the device, and the application does not proceed in its execution. Otherwise, the application proceeds in creating the necessary sensors and registering the appropriate sensor listeners. The classes that are required are the following:

- android.hardware.Sensor;
- android.hardware.SensorEvent;
- android.hardware.SensorEventListener;
- android.hardware.SensorManager;

After registering the sensor listeners, the onSensorChanged(SensorEvent) method is implemented. This method is responsible for acquiring the new data of the sensors measurements, by “listening” to SensorEvents. When a SensorEvent occurs, the onSensorChanged() is called, with that event as an argument. Inside the onSensorChanged() there are some control flow statements in order to identify the source of that Event.

The Event contains an array named Values, which contains the measurements of the sensor. Its length and its contents vary for every sensor. For example, the Values array of the accelerometer, contain the data of the measurements in the following form:

- values[0]: Acceleration minus $G_x$ on the x-axis in $m/s^2$
- values[1]: Acceleration minus $G_y$ on the y-axis in $m/s^2$
- values[2]: Acceleration minus $G_z$ on the z-axis in $m/s^2$

Respectively, the same applies to the gyroscope, where the Values array has the following form:

- values[0]: Angular speed around the x-axis in rads/s
- values[1]: Angular speed around the y-axis in rads/s
- values[2]: Angular speed around the z-axis in rads/s

The orientation sensor as well has a similar Values array containing the measurements of the orientation of the device in degrees.

- values[0]: Azimuth, angle between the magnetic north direction and the y-axis, around the z-axis (0 to 359). 0=North, 90=East, 180=South, 270=West

- values[1]: Pitch, rotation around x-axis (-180 to 180), with positive values when the z-axis moves toward the y-axis.
- values[2]: Roll, rotation around the y-axis (-90 to 90) increasing as the device moves clockwise.

In this stage, when a sensor event occurs, the onSensorChanged() is called and the appropriate text field on the screen of the device is updated accordingly.

### 4.3.3 Acquiring Data from the GPS Receiver

The next step was to manage to receive GPS data from the GPS receiver of the smartphone. After studying the Location Services API on the Location Strategies website of the Android Developers [28], it was made possible to implement the code that allowed the application to receive GPS data, such as the latitude, longitude, and altitude, every one second, provided that the smartphone has a clear view of the sky.

A Location Listener is needed in order to receive the position of the device. For the implementation of the Location Listener, the following classes were required, and were imported to the application:

- android.location.Location;
- android.location.LocationListener;
- android.location.LocationManager;

The application is designed in a way that if the Location Services of the device are disabled, an Intent will prompt the user to enable them, and until they are enabled, the user will not be able to proceed to the main screen of the application. If the Location Services are enabled, the application will start, the Location Manager will request for Location Updates, and whenever a Location Update happens, the onLocationChanged() method will be called. The onLocationChaned() has an input argument of the Location class, which contains the methods of getLongitude(), getLatiture, getAltitude(), and many others. These methods return a floating point number that represents the longitude, latitude, and altitude respectively. In this stage, when the onLocationChaned is called, all of these three numbers are acquired, and they are displayed on the screen. This addition to the application can be seen in Figure 4.4.

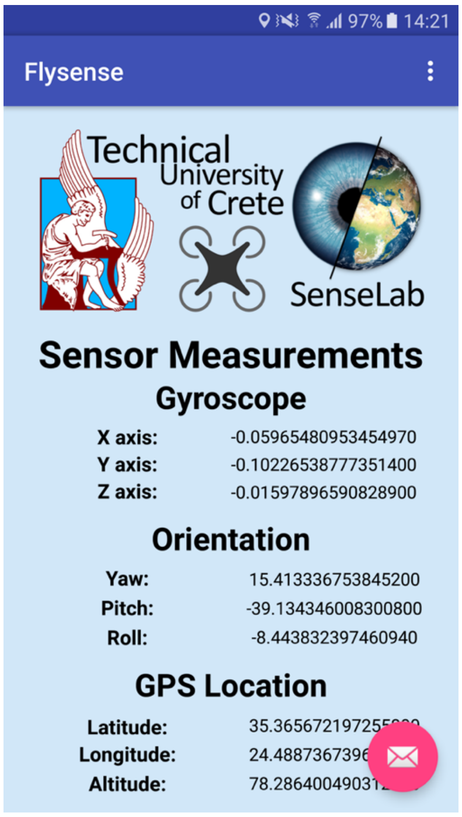
### 4.3.4 Measuring the Update Rate of the Sensors and the GPS Receiver

In the next stage of the development of the Android application, it had to be confirmed that the sensors can provide measurements in a sufficiently high update rate, in order for them to be utilized by the autopilot software. As mentioned in chapter 3, PenguPilot is capable of reading new sensor data from the IMU via $I^2C$ every 0.005 seconds. In the early stages of this work, when the requirements were set, the goal was to achieve at least an update rate of 100 Hz from the sensors of the Android device in order to be considered sufficient for a UAV flight. According to this fact, the Android device had to be capable of providing measurements at an update rate of at least 100 Hz. In order to successfully measure the update rate of the sensors, a simple timer, and some counters had to be implemented.

Using a Handler [29], which is a part of the framework of the Android system for managing threads, this was made possible. A Handler allows the scheduling of messages and Runnables to be executed at some point in the future. The scheduling is accomplished with various methods, such as the postDelayed(Runnable, long) that is used in this application. This method causes the Runnable object to be added to the message queue and to be run on the thread to which the Handler is attached, after the specified amount of time elapses.

In this case, the time delay was set at 1000 milliseconds, in order to measure the update rate per one second of every sensor. The idea behind this implementation is to use a counter for every sensor which will be increased by one when the onSensorChanged() method is called for that particular sensor. Respectively, the same applies for the GPS receiver in order to measure its update rate. In contrast to the onSensorChanged(), for the GPS receiver the counter is located in the onLocationChanged() method, and is increased in every call of the function.

When 1000 milliseconds have passed, the Handler will execute the code that displays to the screen the last counter values, as shown in Figure 4.5, and it will set the counters back to zero. This method can provide a realistic result about the true update rates of the sensors.


97% 14:21

Flysense

Technical University of Crete

SenseLab

**Sensor Measurements**

**Gyroscope**

| | |
|---|---|
| X axis: | -0.05965480953454970 |
| Y axis: | -0.10226538777351400 |
| Z axis: | -0.01597896590828900 |

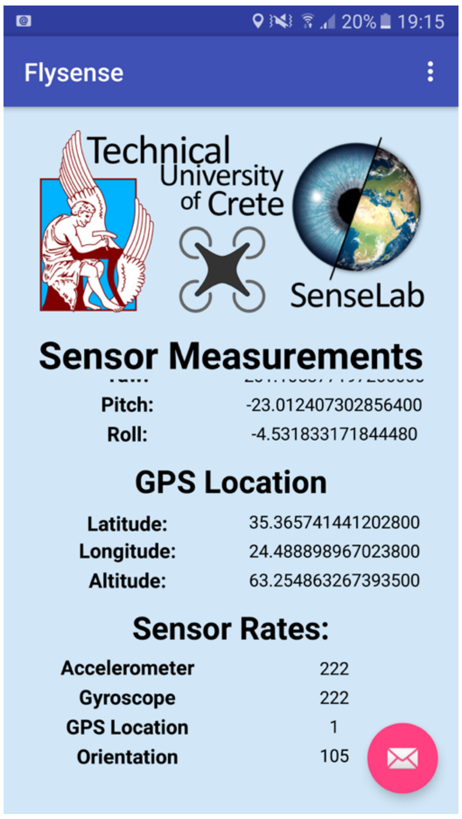
**Orientation**

| | |
|---|---|
| Yaw: | 15.413336753845200 |
| Pitch: | -39.134346008300800 |
| Roll: | -8.443832397460940 |

**GPS Location**

| | |
|---|---|
| Latitude: | 35.36567219725 |
| Longitude: | 24.4887367396 |
| Altitude: | 78.2864004903 |

Figure 4.4: The received GPS data displayed on the screen of the smartphone


20% 19:15

Flysense

Technical University of Crete

SenseLab

**Sensor Measurements**

| | |
|---|---|
| Pitch: | -23.012407302856400 |
| Roll: | -4.531833171844480 |

**GPS Location**

| | |
|---|---|
| Latitude: | 35.365741441202800 |
| Longitude: | 24.488898967023800 |
| Altitude: | 63.254863267393500 |

**Sensor Rates:**

| | |
|---|---|
| Accelerometer | 222 |
| Gyroscope | 222 |
| GPS Location | 1 |
| Orientation | 105 |

Figure 4.5: Measuring the update rates of the sensors and the GPS receiver

The accelerometer and the gyroscope can provide rates of more than 200Hz, the orientation sensor can reach 100Hz, and the GPS receiver is updated only once per second. Although the update rate of the orientation sensor is the half of the update rate of the accelerometer and the gyroscope, it is still considered sufficient for this implementation. This is because when the requirements for this work were set, the goal was to achieve at least an update rate of 100 Hz from the sensors of the Android device in order to be considered sufficient for a UAV flight. These results were also described with more details in the third chapter of this thesis.

The ability of the smartphone to provide sensor measurements at an even higher update rate is an advantage, which is utilized by PenguPilot, as PenguPilot can read sensor measurements up to 200 Hz. However, this is a fact that applies only to the Android device that was used in this work. It is obvious that other smartphones have different specifications, such as different sensors, different update rates of sensors and a different number of sensors, as they may include a temperature sensor, but may not include a barometer and magnetic field sensor for example. Since specifications may vary from device to device, some devices might be incompatible with the application.

### 4.3.5 Receiving NMEA Sentences

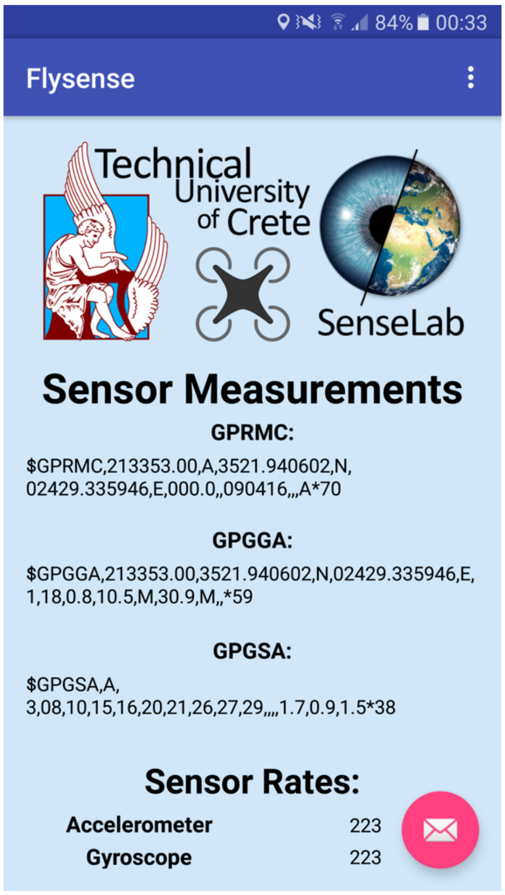
The GPS service of PenguPilot uses NMEA sentences, as it was mentioned in chapter 3. After studying the NMEA data format, the appropriate functionality had to be added to the Android application, in order to provide the required data to the autopilot software. In order to acquire the NMEA sentences from the GPS receiver, an NMEA Listener had to be implemented. The class that contains the NMEA Listener is the following:

- android.location.GpsStatus;

The NMEA interface is implemented by calling the addNmeaListener() method in order to receive NMEA data from the GPS receiver. When an NMEA sentence is received, the onNmeaReceived() method is called. However, there are many different NMEA sentences in the NMEA standard, each one providing different information. The NMEA library that is used in PenguPilot can extract information only from specific NMEA sentences. These are the following:

- $GPGGA - GPS Fix Information
- $GPGSA - GPS Dilution of Precision and Active Satellites
- $GPGSV - GPS Detailed Satellite Information
- $GPRMC - Recommended Minimum Data for GPS
- $GPVTG - Vector Track and Speed over the Ground

Each Android device can output a different list of NMEA sentences. For example, the Samsung Galaxy S6 can provide $GPGGA, $GPGSV, $GPGSA, $GPRMC, $GNGSA and many more sentences, but it cannot provide $GPVTG sentences. Nevertheless, all Android devices can provide the most important NMEA sentences that contain the vital location information, such as the $GPGGA, $GPGSA and the $GPRMC sentences. Due to the fact that Android devices can output many different sentences, a kind of filtering had to be applied to the output of the device, in order to send only the necessary NMEA sentences to PenguPilot.



Sensor Measurements

GPRMC:

```
$GPRMC,213353.00,A,3521.940602,N,
02429.335946,E,000.0,,090416,,,A*70
```

GPGGA:

```
$GPGGA,213353.00,3521.940602,N,02429.335946,E,
1,18,0.8,10.5,M,30.9,M,,*59
```

GPGSA:

```
$GPGSA,A,
3,08,10,15,16,20,21,26,27,29,,,,,1.7,0.9,1.5*38
```

Sensor Rates:

| | |
|---|---|
| Accelerometer | 223 |
| Gyroscope | 223 |

Figure 4.6: Receiving NMEA data

This kind of filtering was implemented by some control flow statements, which check the type of the NMEA sentences. Depending on the NMEA prefix, each NMEA sentence is displayed on the screen of the device in the appropriate text field. The feature of receiving NMEA data in the Android application, as well as the format of some NMEA sentences, can be seen in Figure 4.6.

### 4.3.6 Implementation of the Serial Communication

After the development of the basic functionality, the serial communication had to be implemented. The aim of this stage was to create a serial communication between the Android device and the Odroid platform, as well as to successfully build a data block that contains all of the sensor last measurements in order to send it to the Odroid platform via the serial port.

As it was mentioned before, the purpose of the application is to transmit the data of the sensor measurements and the NMEA sentences. Therefore, in order to keep the CPU and memory usage of the Android device as low as possible, all of the unnecessary graphical user interface components were removed. Instead of displaying the sensor measurements on the screen of the device, when the method onSensorChanged() is called for a Sensor Event, the measurements contained in the Values array of the Event are stored in a global floating point array. There are three global floating point arrays, each one containing the last sensor measurements for each one of the three sensors respectively.

For the implementation of the serial communication, the USB Host API from the Android Developers [30] was used, in combination with an external library that contains some serial port drivers [31]. The serial communication is achieved by using the following classes:

- android.hardware.usb.UsbDevice;
- android.hardware.usb.UsbDeviceConnection;
- android.hardware.usb.UsbManager;

At first, a Broadcast Receiver is started and then, the application waits for the user to begin the serial communication. A filtering mechanism is used, in order for the smartphone to be able to establish a serial connection only with the Odroid platform, and not with any USB device that it may be connected. The mechanism is implemented with a device filter that checks the vendor ID of the device connected to the smartphone. If the vendor ID of the connected device is that of the Odroid platform, the user is requested to grant access to the device. If the permission is granted, the device is opened, the serial connection is created, and the parameters of the connection are set.

### 4.3.7 Transmission of Data

While having a serial connection established, the transmission of the data is ready to begin. In order to have a continuous transmission of the sensor measurements to the Odroid platform, a Handler is used. The scheduling of the Handler is achieved again with the postDelayed() method, the same method that was used in the earlier stages of the application development in order to measure the update rates of the sensors.

When the runnable of this Handler is executed, the last measurements of the sensors are acquired from the three global arrays that always contain the last measurements of each sensor. A string of characters that contains the information of the sensor measurements is created, and then it is written to the serial port. The fact that the size of the written string is nearly 150 bytes, is prohibitive for a serial communication with transmission rates of 200 Hz. It follows that 150 bytes, or equally 1200 bits, multiplied by 200 times per second are equal to 240,000 bits per second. The maximum baud rate of the serial port is 115,200 bits per second, way less than 240,000 bits per second. However, the purpose of this implementation at this stage was only to verify the basic functionality of the serial communication. Therefore, the time delay in the postDelayed() method of the Handler was set by a seek bar, ranging from 1000 milliseconds to 50 milliseconds, not fast enough to exceed 115,200 bits per second, but also not fast enough to achieve a transmission rate of 200 Hz.

In contrast to the transmission method of the sensor measurements, NMEA sentences are written to the serial port immediately when they arrive. NMEA sentences are received from the GPS receiver as ASCII character strings, with a variable length, and their update rate is only once per second.

If the received NMEA sentence begins with a prefix such as $GPGGA, $GPGSA, $GPGSV, $GPRMC, or $GPVTG, then the sentence is concatenated to a temporary character string variable. The NMEA sentences are received in a specific order every time. First, the $GPGGA sentence is received, the $GPGSV sentences are received next, then the $GPGSA sentence is received, and at last the $GPRMC sentence is received. In order not to send each NMEA sentence separately, all sentences are concatenated to the first sentence, creating a string of ASCII characters that contains all of the received NMEA sentences. Therefore, the first sentence, the $GPGGA sentence, is concatenated to the string variable, which initially is empty, then the other sentences follow, and finally, when the $GPRMC sentence is concatenated to the string variable, the whole string variable is transmitted via the serial port to the Odroid platform. After that, the string variable is cleared, and the process repeats.

On the Odroid platform, the GPS service of PenguPilot is designed to extract information from the $GPGGA, $GPGSA and $GPRMC sentences, only when new information about the location is available. Therefore, if the new $GPGGA sentence is the same as the previous $GPGGA sentence, which means that the position is unchanged, PenguPilot will ignore the new block of sentences and it will wait for the next $GPGGA sentence.

A transmitted block of multiple NMEA sentences can be nearly 400 bytes in size, or even more, thus sending all of this information along with the sensor measurements at a high transmission rate would cause a bottleneck in the serial communication, beyond the fact that identical transmitted NMEA information would be useless to PenguPilot. Therefore, the approach for transmitting NMEA data is to immediately send the received sentence to the Odroid platform via the serial port.

The user interface of the application in this stage is as simple as possible, in order to reduce the need of CPU and memory resources. As shown in Figure 4.7, the user interface in this stage is comprised of only three buttons and a seek bar. Displaying the sensor measurements and other information on the screen of the smartphone, as in the previous stages of the development, could cause an unnecessary load to the CPU, leading to degraded performance, such as lower sensor update rates.

The start button initializes the serial communication, the stop button terminates the serial communication and the clear button simply clears the screen from the information messages that are created each time when a connection is established or terminated. The seek bar can is responsible for the amount of the time delay in the postDelayed() method of the Handler. The flow chart of the Android application is shown in Figure 4.8.

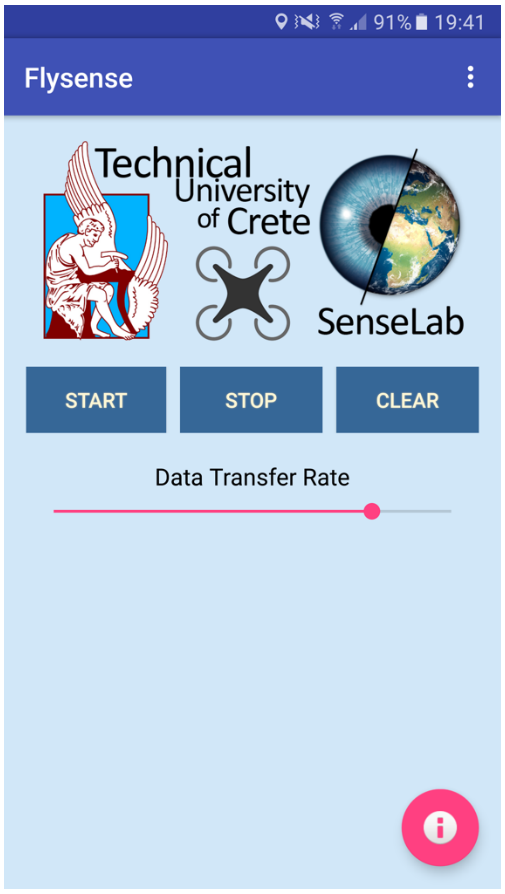


Figure 4.7 The application after the implementation of the serial communication

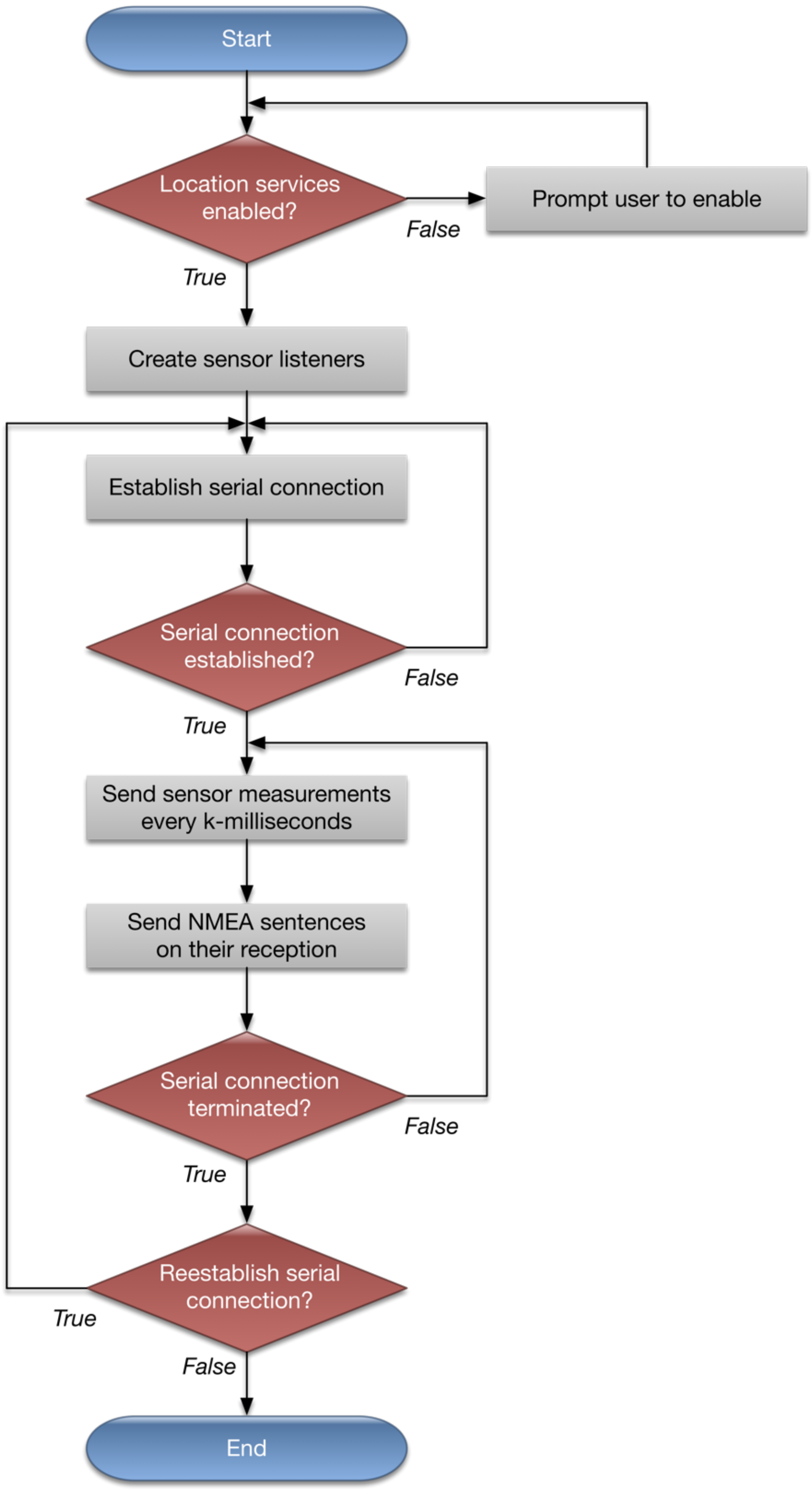

Start
Location services enabled?
False
Prompt user to enable
True
Create sensor listeners
Establish serial connection
Serial connection established?
False
True
Send sensor measurements every k-milliseconds
Send NMEA sentences on their reception
Serial connection terminated?
False
True
Reestablish serial connection?
True
False
End

Figure 4.8: Android application flow chart

### 4.3.8 Changes and Improvements

While studying further the autopilot software, it was found that it does not utilize an orientation sensor, but a geomagnetic field sensor instead. In fact, the orientation sensor and the magnetic field sensor use the same hardware in the Android device, the Yamaha YAS537, with the difference that the function of the orientation sensor outputs the calculated the azimuth, pitch, and roll of the device, in contrast to the function of the geomagnetic field sensor that outputs the geomagnetic field strength in μT for each of the three coordinate axes of the device.

Furthermore, it was also found that PenguPilot can optionally utilize a barometer and a temperature sensor, as well as an ultrasonic sensor when available. For this reason, a barometer sensor was added to the Android application, as well as a text input field for entering manually the environment temperature. However, other Android devices may have a temperature sensor available that can be utilized instead of entering manually the environment temperature.

The barometer was implemented in the same way that the other sensors were implemented too. Before registering a Listener for the pressure sensor in the onCreate() method, a check is conducted in order to verify whether there is a pressure sensor available in the device. If a pressure sensor is not available, the application does not proceed its execution, and a message appears on the screen of the smartphone that informs the user that the device is incompatible, otherwise, a Listener for the pressure sensor is registered.

When new data from the pressure sensor are available, the onSensorChanged() method is called, with a SensorEvent as an argument. A pressure sensor event contains a Values array, similarly to the other sensor events, with only one difference. The Values array of the Sensor Event of the pressure sensor contains only one element, values[0], which describes the atmospheric pressure in hPa. When a new measurement is available, it is stored in a global floating point variable, in a similar way to the other sensor measurements. When the Handler is executing the Runnable code that transmits the sensor measurements to the Odroid device, the pressure sensor measurement is added to the transmitted block of the sensor measurements as well.

The fact that the barometer has a low update rate raised some concerns, as it was mentioned in chapter 3. However, in the detailed study that was conducted in the same chapter, it was found out that even its low update rate is considered sufficient for outdoor autonomous flights.

In order to provide data for the ambient temperature to the autopilot software, a simple input text field was added to the graphical user interface. The user is allowed to input a temperature in Celsius degrees with a precision of two decimal digits, and then press the update button in order to apply the change. The idea of the temperature input was made possible due to the fact that the temperature is generally constant in an area, and to the fact that PenguPilot is not heavily dependent on temperature measurements. In fact, currently PenguPilot is not utilizing the temperature measurements, however, temperature measurements may be utilized in a future version.

Finally, in the last stage of the Android application development, some improvements were made, in order to achieve the transmission of the NMEA sentences and the sensor measurements blocks within the 115,200 baud rate that is used. The data provided from the sensors are in the format of floating point numbers, and they are the following:

- 3 axes accelerometer: 3 floating point numbers
- 3 axes Gyroscope: 3 floating point numbers
- 3 axes magnetic field sensor: 3 floating point numbers
- Barometer: 1 floating point number
- Temperature: 1 floating point number

Instead of writing the sensor measurements in the serial port in the format of ASCII characters, the sensor measurements are written in the serial port in the form of raw bytes. As shown in Table 2, every floating point number is represented by four bytes, adding all of the output of the sensors if follows that 44 bytes are needed for encoding the information from the sensors. Three bytes are added as a header of the block, in order to identify and distinguish the sensor measurements from the NMEA sentences, and three more bytes are added at the end of the block as a footer. The footer contains a single byte for a simple checksum (CS) verification, and the carriage return and line feed characters. The function of the simple checksum verification will be described in detail in the fifth chapter of this thesis.

A total of 400 bits, equal to 50 bytes, comprise a full block of sensor measurements. Multiplying this number by 200, which is the times per second a sensor measurements block is sent, it occurs that it is equal to 80,000 bits pes second. Adding the typical NMEA sentences that are received by the GPS receiver every second, this number roughly approaches the 83,200 bits per second. This is a reasonable number, compared to the available bandwidth of 115,200 bits per second of the serial interface.

| **Byte:** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Data:** | & | S | E | Accelerometer: X axis | | | | Accelerometer: Y | | |
| **Byte** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** |
| **Data:** | axis | Accelerometer: Z axis | | | | Gyroscope: X axis | | | | Gyro- |
| **Byte:** | **21** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** |
| **Data:** | scope: Y axis | | | Gyroscope: Z axis | | | | Magnetic Field: X | | |
| **Byte:** | **31** | **32** | **33** | **34** | **35** | **36** | **37** | **38** | **39** | **40** |
| **Data:** | axis | Magnetic Field: Y axis | | | | Magnetic Field: Z axis | | | | Baro- |
| **Byte:** | **41** | **42** | **43** | **44** | **45** | **46** | **47** | **48** | **49** | **50** |
| **Data:** | meter | | | Temperature | | | | CS | \r | \n |

Table 4.1: Sensor measurements block format

In this stage of the application development, the seek bar can adjust the time of the postDelayed(Runnable, long) method with values ranging from 1024 to 4 milliseconds, giving the ability to send data via the serial port from almost every second to more than 200 times per second. Furthermore, in this stage, some changes were made in the user interface, such as adding a spinner and some switches. The spinner allows the user to select between an accelerometer, a gravity sensor, or a linear accelerometer, two switches are responsible for the output of the gyroscope and the magnetic field sensor, and using the last switch, the user can enable or disable the transmission of NMEA sentences to the Odroid platform.

The application is creating instances of every sensor type, such as the accelerometer, linear accelerometer, gravity sensor, gyroscope, uncalibrated gyroscope, magnetic field sensor, uncalibrated magnetic field sensor and pressure sensor in the onCreate() method. The onSensorChanged() method checks only for the selected type of sensors, according to the selections of the user. For example, if the user has selected the linear accelerometer, the uncalibrated gyroscope, and the magnetic field sensor, the onSensorChanged() method will acquire the measurements only from these sensors as well as the measurements of the barometer. The measurements from barometer are always acquired, as there is no alternative sensor to the pressure sensor.

More specifically, the gravity sensor and the linear accelerometer utilize the same hardware of the accelerometer sensor. The gravity sensor is able to provide a three dimensional vector indicating the magnitude and direction of gravity, while the linear accelerometer is able to provide data without the influence of the gravity. In fact, the gravity sensor acts as a normal accelerometer with a low-pass filter applied, in contrast to the linear accelerometer, which acts as a normal accelerometer with a high-pass filter applied. Similarly, it is also possible to select between a calibrated gyroscope and an uncalibrated gyroscope, as well as a geomagnetic field sensor and an uncalibrated magnetic field sensor. The uncalibrated gyroscope is similar to the calibrated gyroscope, except that no gyro-drift compensation is applied to the rate of rotation. The difference between the geomagnetic field sensor and the uncalibrated magnetic field sensor is that no hard iron calibration is applied to the magnetic field output data of the uncalibrated magnetic field sensor.

It has to be noted that, uncalibrated sensors are able to provide more raw results and may include some bias, however, their measurements contain fewer jumps from corrections applied through calibration. These uncalibrated results may be preferred sometimes as they are smoother and more reliable. For example, if an application is designed to conduct sensor fusion on its own, introducing calibrations can actually distort the results. Providing the ability to select between different types of sensors, as well as between calibrated and uncalibrated sensors, the user is allowed to configure the output of the application in order to acquire the desired data.

The final graphical user interface of the Android application can be seen in Figure 4.9.

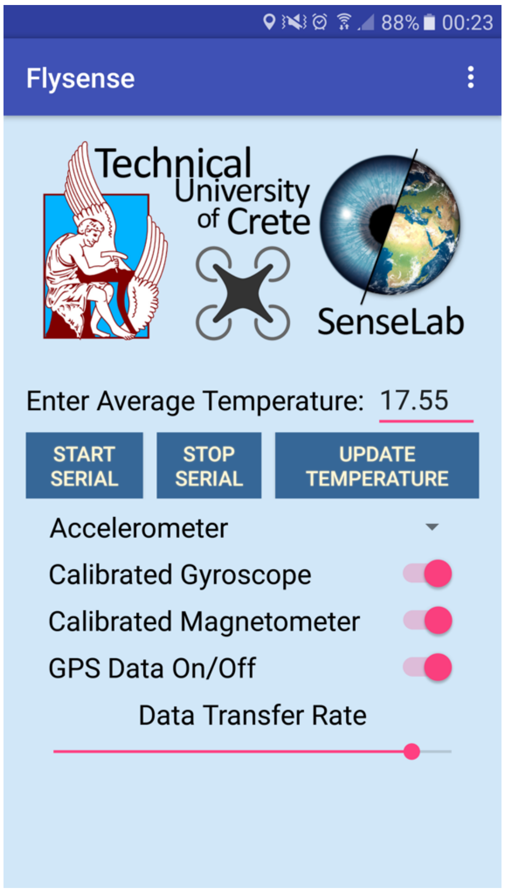


Figure 4.9: The final Android application

## 4.4 Software Development on the Odroid Platform

### 4.4.1 Serial Communication Infrastructure

The first stage of the development of the interfacing software was to establish a reliable serial communication between the Odroid platform and the Android device. The first approach was to develop a Python script that would simply receive the data from the Android device and print them on a terminal. Initially, all of the sensor measurements were transmitted in the format of ASCII characters. Of course, this form of communication could not allow high transmission rates. Therefore, in the last stage of the Android application development, the sensor measurements are transmitted in the format described in the previous section and is presented in Table 2.

The serial connection between the Android device and the Odroid platform is implemented between the micro USB ports of the two devices. The Android application is designed to work by having the Android device as the host of the connection. For that reason, a special Micro USB Host cable was needed that had to be connected to the Android device. The Host side of the cable is connected to the micro USB port of the Android device, and the other side, a USB type A female port is connected to a type A male port of another USB cable of which the other end is connected to the micro USB port of the Odroid platform. The diagram of the connection is shown in Figure 4.10.

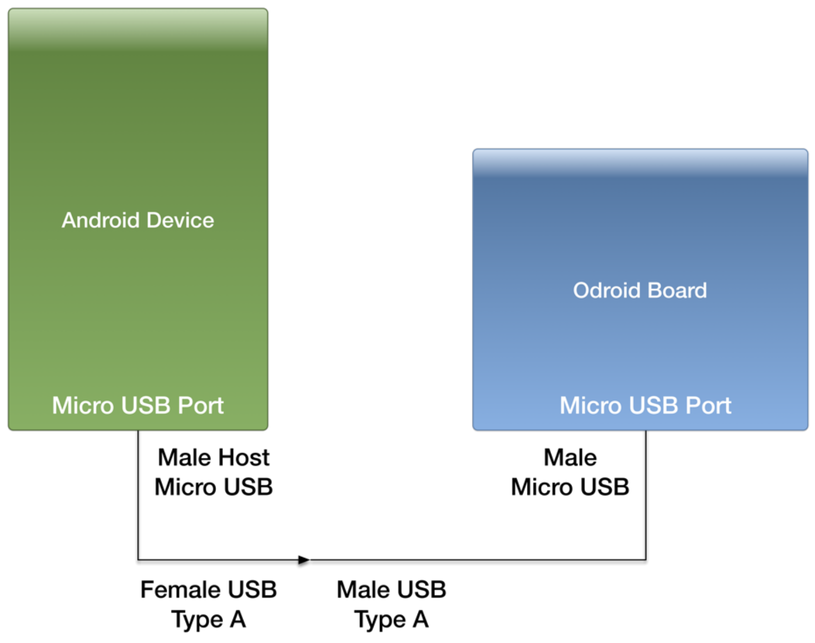


Figure 4.10: The USB serial connection

In order to use the micro USB port of the Odroid platform, the appropriate kernel module had to be loaded. Loadable kernel modules are commonly used to add support for new hardware, such as device drivers and filesystems, or for adding system calls. A loadable kernel module is an object file, which contains the necessary code to extend the base kernel of an operating system. In Linux, loadable kernel modules are loaded, and unloaded, by using the modprobe [32] utility. When executing the modprobe g_serial [33] command on the Odroid board, an emulated serial port is created, and the Odroid platform can be used as a serial device via its micro USB port. Then, the device connected to the Odroid platform is going to be the host of the serial communication.

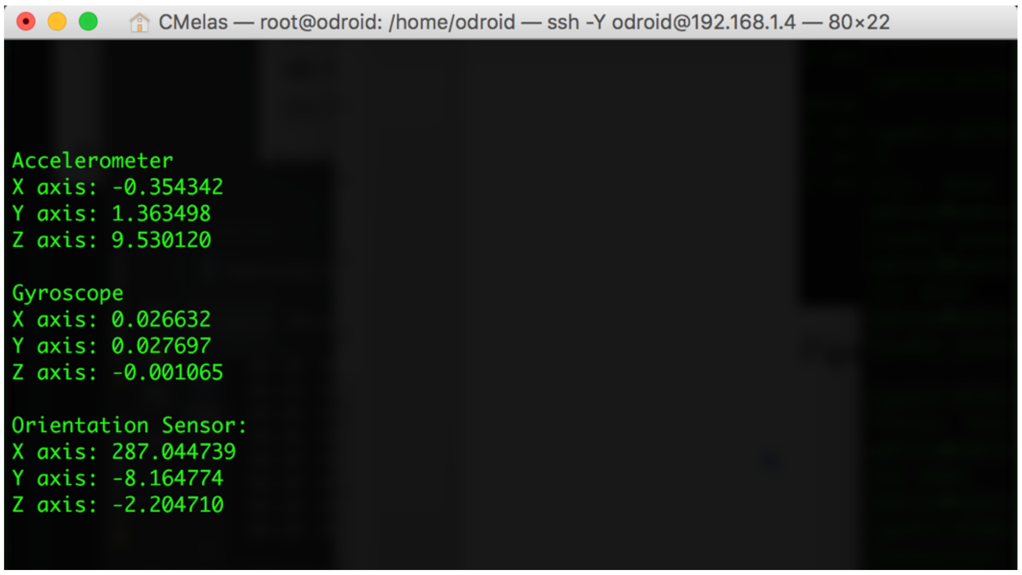
The command modprobe g_serial is creating the /dev/ttyGS0 serial port on the Odroid platform, and this port is used for receiving the data from the Android device. In order to avoid executing the modprobe command every time the Odroid platform boots, the g_serial module was added to the /etc/modules file of the system. In this way, the modprobe g_serial command is executed on the boot of the device, and the serial port is created automatically.

### 4.4.2 Receiving Data

The first approach was to simply receive the transmitted data and print them on a terminal in order to verify the serial communication between the two devices. For that reason, a python script was developed on the Odroid platform that could open the /dev/ttyGS0 serial port and read the incoming data from the other device. The transmitted data were in the format of ASCII characters, and when new data were available, they were received and printed on a terminal, as shown in Figure 4.11. This simple script made the verification of the serial communication possible. More details about the verification of the software, as well as of the Android application, will follow in the next chapter.

After the verification of the serial communication, improvements and changes were made to the Android application, as they were described in the previous section of this chapter. The most important issue was to achieve a transmission rate of 200 Hz, and to be able to receive the data successfully. The implementation of the sensor measurements blocks, as described in section 4.3.8 of this chapter, made possible to transmit all of the sensor measurements at rates of more than 200 Hz.

A checksum function allowed to verify the checksum of the received sensor measurements with the received checksum. For example, if the checksum of a sensor measurements block is not verified with the received checksum, then the measurements are considered corrupt and they are ignored. Along with the measurements, the NMEA sentences had to be successfully received and verified as well. However, the NMEA library uses a function that can verify the data of the received NMEA sentences on its own, by checking the checksum of the data of the received sentence with a received checksum that is appended to the end of the sentence.


CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.4 — 80×22

```
Accelerometer
X axis: -0.354342
Y axis: 1.363498
Z axis: 9.530120

Gyroscope
X axis: 0.026632
Y axis: 0.027697
Z axis: -0.001065

Orientation Sensor:
X axis: 287.044739
Y axis: -8.164774
Z axis: -2.204710
```

Figure 4.11: The output of the first python script on the Odroid platform

In fact, the implementation of the verification function of the sensor measurements is similar to the verification of the NMEA sentences. The NMEA checksum is represented by two hexadecimal characters of the XOR of all characters in the sentence, between the “$” and the “*” character. Similarly, the checksum of the sensor measurements is represented by a single byte, which is the result of the XOR of all bytes in the sensor measurements block, between the “E” character and the checksum byte (CS). More details about the checksum function and verification process will be presented in the next chapter.

Furthermore, in order to verify the data reception, some python scripts were developed that were able to perform specific actions. For example, dataRate.py is responsible for the measurement of the transmissions per second. More information about the verification process that confirmed the successful operation of the data transmission and reception will be provided in detail in the following chapter as well.

The biggest issue that had to be addressed was the distribution of the sensor measurements and of the NMEA sentences to the appropriate PenguPilot services. As it was mentioned in chapter 3, PenguPilot consists of services and dependencies between them. The service responsible for the sensor measurement readings is the "i2c_sensors" service, while the service that is responsible for the GPS information is the "gpsp" service.

The "gpsp" service is designed to extract information from the NMEA sentences received from a GPS receiver connected to a serial port on the Odroid platform. The output of these receivers is NMEA sentences, as described previously, in the ASCII character format. Therefore, the NMEA sentences had only to be forwarded to the appropriate serial port, which the "gpsp" service is using.

The "i2c_sensors" service is designed to read sensor measurements from $I^2C$ sensors connected to the Odroid platform. In this case, all of the sensor measurements are provided by the Android device instead of $I^2C$ sensors. Therefore, modifications had to be done in order to read the sensor measurements from the serial connection with the Android device, instead of reading them via $I^2C$ sensors.

The received data from the Android device had to be distributed to the appropriate service. The NMEA sentences have to be forwarded to the "gpsp" service, and the sensor measurements to the "i2c_sensors" service. As only one process can acquire the serial port at a time, is impossible to have both "gpsp" and "i2c_sensors" services accessing the /dev/ttyGS0 port simultaneously. For that reason, the socat [34] utility was used, in order to create two virtual serial port pairs. The first virtual serial port pair, /dev/NMEA and /dev/NMEAin are going to be utilized for the distribution of the NMEA sentences to the "gpsp" service, and the second virtual pair port, /dev/SENS and /dev/SENSin, are going to be utilized for the distribution of the sensor measurements to the "i2c_sensors" service.

The socat utility can be used for many different purposes. It is a command line based utility, which is able to create two bidirectional byte streams and to transfer data between them. The two commands that were used are the following:

- socat -d -d pty, raw, echo=0, link=/dev/SENSin pty, raw, echo=0, link=/dev/SENS
- socat -d -d pty, raw, echo=0, link=/dev/NMEAin pty, raw, echo=0, link=/dev/NMEA

Each of these two commands generates a pair of pseudo terminals (pty), and by using link=<filename> a symbolic link that points to the actual pseudo terminal is generated. This is a very useful option, as pseudo terminals are usually generated with random names, rendering the automatic direct access to them impossible. As it is obvious, the /dev/NMEA is the pseudo terminal that the “gpsp” service will use for receiving the NMEA sentences, while /dev/NMEAin is the pseudo terminal that the NMEA sentences will be forwarded to from /dev/ttyGS0. Similarly, the /dev/SENS pseudo terminal will be used from the “i2c_sensors” service in order to receive the sensor measurements, while the sensor measurements will be forwarded to the /dev/SENSin pseudo terminal from the /dev/ttyGS0.

### 4.4.3 Handling Data

The software that is responsible for the separation of the data and the forwarding to the appropriate pseudo terminals was initially developed in python. The first python script was designed to open the /dev/ttyGS0 port, the one to where the sensor measurements and the NMEA sentences are transmitted by the Android phone, and also open the /dev/NMEAin and /dev/SENSin virtual serial ports to which it would forward the appropriate data. The sensor measurements are going to be forwarded to /dev/SENSin, and the NMEA sentences are going to be forwarded to /dev/NMEAin.

There is a simple loop, where the reading of the data from the /dev/ttyGS0 serial port is conducted, then some control flow statements are responsible for the identification of the received data. For example, all NMEA sentences begin with the characters “$”, “G”, “P”, and all of the sensor measurements blocks begin with the characters “&”, “S”, “E”. If the program identifies the received data as NMEA sentences then they are immediately written to the /dev/NMEAin virtual serial port, otherwise, if the received data are identified as sensor measurements, they will be written to the /dev/SENSin virtual serial port. This loop is executed once every one millisecond, or 1000 Hz, a much higher rate than the 200 Hz of the transmission of the data from the Android device. This approach guarantees that the transmitted data from the Android device will be read immediately as soon as they are available to the /dev/ttyGS0 serial port. Of course, the execution of the loop can be performed in lower rates, however, it is obvious that checking the serial port for new data less frequently can result in degraded performance and malfunctions.

The time.sleep() method is used in the python script, in order to suspend the execution of the current program thread. The time is set in seconds, therefore in the python script the argument in the time.sleep() method is 0.001seconds. With this approach, the Python program uses the CPU way less than continuously polling for new data in the serial port, while checking once per one millisecond is more than enough for this application, which transmits new data once every five milliseconds. The verification of the functionality of this program will be presented in the next chapter, along with the statistics of its resource utilization.

However, it was found out that the python script was consuming a considerable amount of resources, and for this reason the software responsible for the separation and the forwarding of the received data was implemented in C. The C implementation of this program resulted in a reduction of the CPU load up to 90% in comparison to the implementation in Python.

The equivalent implementation in C follows the same execution pattern. Initially, the /dev/ttyGS0, /dev/NMEAin and /dev/SENSin ports are opened. Then, inside a loop the reception of the data from /dev/ttyGS0 is conducted. As in the python script, some control flow statements separate the received data and transmit the sensor measurements to /dev/SENSin, and the NMEA sentences to /dev/NMEAin. Similarly to the time.sleep() of the python script, here the nanosleep() function is used. The nanosleep() function is called in order to suspend the execution of the calling thread until the time interval specified in an argument of the function has elapsed. The time interval is also set at 1 millisecond in order to make sure that the transmitted data from the Android device are received as soon as possible, as in the previously mentioned python script. Using C for the implementation of this program resulted in a more efficient resource utilization of the Odroid platform. However, more details about the performance of this program will be presented in the next chapter. The route of the data through the components of the system can be seen in Figure 4.12, beginning from the sensors of the Android device and ending to the two services of PenguPilot.

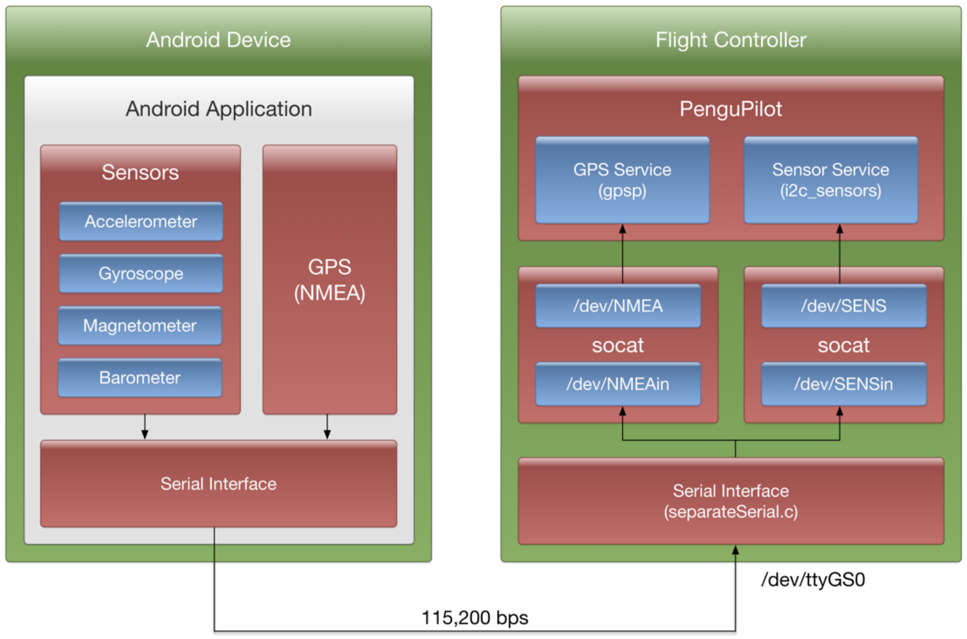


Figure 4.12: The route of the data through the components of the system

### 4.4.4 Modifying PenguPilot

Both “gpsp” and “i2c_sensors” services of PenguPilot had to be modified in order to receive the data from the virtual serial ports. The “gpsp” service did not need any code modifications, as it is already designed to receive NMEA sentences from a GPS receiver that is connected to a serial port. However, the “i2c_sensors” service had to be modified as it is designed to read the sensor measurements via I$^2$C, in contrast to this approach, where all of the sensor measurements are transmitted from the Android device via a serial port.

Each service of PenguPilot is launched by executing its main file. In the main file of the “gpsp” service, there is a control flow statement, which decides whether the data from a GPS receiver are going to be received by a serial port or via I$^2$C, according to the platform of the system. The “gpsp” service has already a serial communication implementation available, which is able to open the serial port where the GPS receiver is connected, and then receive the NMEA sentences from the GPS receiver. In this case, the platform of the system is an Odroid XU3 Lite, therefore, the serial implementation is selected.

The main_serial file of the “gpsp” service is responsible for the reception of the NMEA sentences, the parsing of the data, and the generation of the information that PenguPilot will use. Initially, the serial port that the NMEA sentences are going to be received from is opened. Then, the incoming data are parsed using the NMEA Library [11], and the required location information is generated. Finally, using MessagePack [35] and the SCL subsystem of PenguPilot, the information is passed to all of the components of PenguPilot that rely on GPS information.

Since the serial communication and the generation of the location information from the NMEA sentences is already implemented in the “gpsp” service, the only change that had to be made, was to edit the System Parameters Configuration File of PenguPilot, params.yaml, in order to define a different path of the serial port that was to be used by the “gpsp” service. Specifically, the serial port path was set to /dev/NMEA, the virtual serial port that was created by socat, where the NMEA sentences are forwarded to. The “gpsp” service was tested and its functionality was verified when using the virtual serial port. The results are presented in the next chapter.

As the “i2c_sensors” is designed to read all of the sensor measurements via $I^2C$, a new implementation had to be developed as all of the sensor measurements, in this case, are forwarded to the /dev/SENS virtual serial port. A new library for the “i2c_sensors” service had to be developed, in order to receive the sensor measurements via the serial port. Initially, the functionality of the “i2c_sensors” service had to be understood.

In the main file of the “i2c_sensors” service, there is a control flow statement that decides which library will be used for the reading of the sensor measurements, according to the platform of the system. The initialization function of the selected library is called in order to initialize the vectors where the sensor measurement will be held, and to register the read functions of the available sensor. For example, an ultrasonic sensor will be used only if it is available, however, an ultrasonic sensor is not a critical sensor component for PenguPilot to operate. Only the accelerometer and the gyroscope are the two compulsory sensors. However, when more types of sensors are available, PenguPilot is capable of providing more functionality.

The reading functions of the accelerometer and gyroscope are called inside the periodic thread of the main file of the service, which is running on a period of 0.005

seconds. All other sensor reading functions, such as for the Magnetic Field sensor and the Barometer, are called by other threads that are created individually for each additional sensor by PenguPilot. The data of the measurements are distributed by using MessagePack and the Signaling and Communication Link subsystem of PenguPilot to all of the components that rely on them.

The new library is structured as the one that is used in PenguPilot for reading the sensor measurements via the $I^2C$ protocol, and has equivalent implementations of the functions that the latter library has. The functions were designed to provide the same output format as of the previous library that uses the $I^2C$ protocol. However, the major difference is that in the new library utilizes a serial port for receiving the sensor measurements from the Android device. The sensor measurements are received in a data block and they are stored in a global struct of the program. Whenever PenguPilot is calling a sensor read function, the latest stored data are retrieved. In "i2c_sensors" all of the sensor measurements are received from the /dev/SENS virtual serial port.

Initially, the equivalent initialization function of the new library is called by the main file. This function registers the read functions of the available sensors and initializes the vectors that will hold the sensor measurements. Furthermore, a new function is called inside the initialization function, which is responsible for opening the appropriate serial port, /dev/SENS, and for creating a new thread that will perform the reading of the received sensor measurements from the serial port.

The new thread is created by using the pthread_create() function. The parameters and the scheduling policy of the thread are set by using the pthread_setschedparam() function. The scheduling policy is set to First In - First Out (SCHED_FIFO), a real-time policy, which allows defining a priority for the thread. The lowest priority level is represented by the value 1, and the highest priority level is represented by the value 99. The priority of the thread is set accordingly to the highest priority that PenguPilot is using for its threads. The highest priority of PenguPilot is 49 by default while other lower priorities have smaller values. It is very important that the vital services of PenguPilot will have higher priority to other services. For example, the thread that is responsible for reading the accelerometer and the gyroscope has to have a higher priority than the "log_proxy" service. It does not mean that the "log_proxy" service is unnecessary, but in a situation when the CPU load is high, priority will be given to the most important services of PenguPilot in order to maintain the safe control of the UAV. The following list describes the priorities of PenguPilot and where they are applied to.

- PenguPilot Priority 1: 49 (Rotation speed control)
- PenguPilot Priority 2: 48 (Rotation position control)
- PenguPilot Priority 3: 47 (N/E/U speed control)
- PenguPilot Priority 4: 46 (N/E/U position control)
- PenguPilot Priority 5: 45 (Missions/manual control)
- PenguPilot Priority 6: 44 (Logging/config/time functionality)

The argument of the routine that is going to be executed is a pointer to a global struct of variables, where the received sensor measurements are going to be stored. The new thread executing a loop, which performs the reading from the virtual serial port, similarly to the program that is used to separate and forward the data that are received from the Android device to the two virtual serial ports.

Using a global struct, the stored data are becoming available to read from every sensor read function inside the library. For example, when the read_acc() function is called, it will always return the last values of the accelerometer that were stored in the appropriate variables of the global struct, as shown in Figure 3.4.

From this point and beyond, the functionality of the service remains the same, as the only part that had to be modified was the functions that read the sensor measurements. By using the approach that was presented, it was made possible to receive the sensor measurements from the virtual serial port, instead of using the $I^2C$ protocol. The data from the sensors are distributed to the components of PenguPilot that require them, similarly to the way that was described previously, by using MessagePack and the Signaling and Communication Link subsystem of PenguPilot.

After the implementation of the modifications, tests had to be conducted in order to verify and confirm the proper functionality of the two services of PenguPilot. The process of the verification along with the results of the tests will be described in detail in the next chapter of this thesis.

# Chapter 5

# Results and Evaluation

## 5.1 Outline

In this chapter the verification of the system is presented, along with the analysis of its performance. The tests that were conducted are described in detail both for the Android application and the software that was created to run in the Odroid platform. Furthermore, the procedure of the calibration of the sensors will be presented, and finally, the resource utilization of the whole system will be discussed.

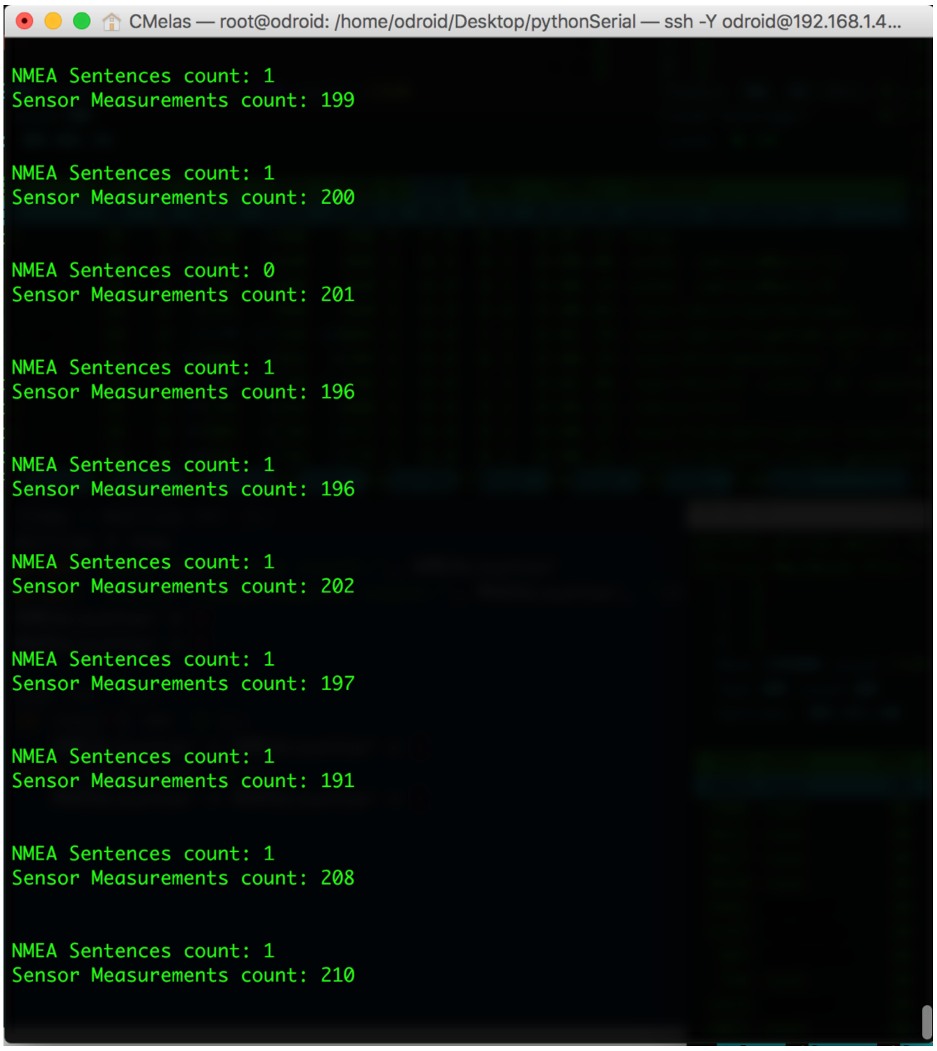
## 5.2 Android Application

### 5.2.1 Verification of the Serial Communication

It is vital that the Android application is able to run continuously, and that it provides the sensor measurements and the GPS data to the flight controller reliably during a flight. The application had to be able to run when it is not in the foreground of the screen, or even when the screen is locked. For example, it would be a disaster if a possible incoming phone call during a flight could cause the application to pause or stop its execution. This would result in the loss of the sensor measurements, and the crash of the UAV. Furthermore, the load of the CPU had to be measured, in order to guarantee the proper execution of the application under all circumstances.

In order to verify the proper functionality of the Android application, some test programs were developed on the Odroid platform. The main functionality that had to be confirmed was the transmission rate, as it had to be able to reach 200 Hz, and the integrity of the received data. The received data could be verified by using a simple checksum function, and the true transmission rate was measured in order to confirm that it can reach rates at 200 Hz.

For the measurement of the data rate a python script was used. The only purpose of this program is to open the /dev/ttyGS0 port and read the transmitted data. There is a simple timer that is responsible for measuring 1 second. From the beginning of the execution, for every transmission, a counter is increased. When 1 second has elapsed, the value of the counter is printed, and then it is reset to zero. As shown in Figure 5.1, the transmission rate is able to reach 200 Hz reliably. A fluctuation of around 5% was observed, with values ranging from 190 Hz to 210 Hz. This is an expected result of the system, as the Handler of the Android application is not accurate enough to provide a fixed transmission rate of 200 Hz.


CMelas — root@odroid: /home/odroid/Desktop/pythonSerial — ssh -Y odroid@192.168.1.4...

```
NMEA Sentences count: 1
Sensor Measurements count: 199

NMEA Sentences count: 1
Sensor Measurements count: 200

NMEA Sentences count: 0
Sensor Measurements count: 201

NMEA Sentences count: 1
Sensor Measurements count: 196

NMEA Sentences count: 1
Sensor Measurements count: 196

NMEA Sentences count: 1
Sensor Measurements count: 202

NMEA Sentences count: 1
Sensor Measurements count: 197

NMEA Sentences count: 1
Sensor Measurements count: 191

NMEA Sentences count: 1
Sensor Measurements count: 208

NMEA Sentences count: 1
Sensor Measurements count: 210
```

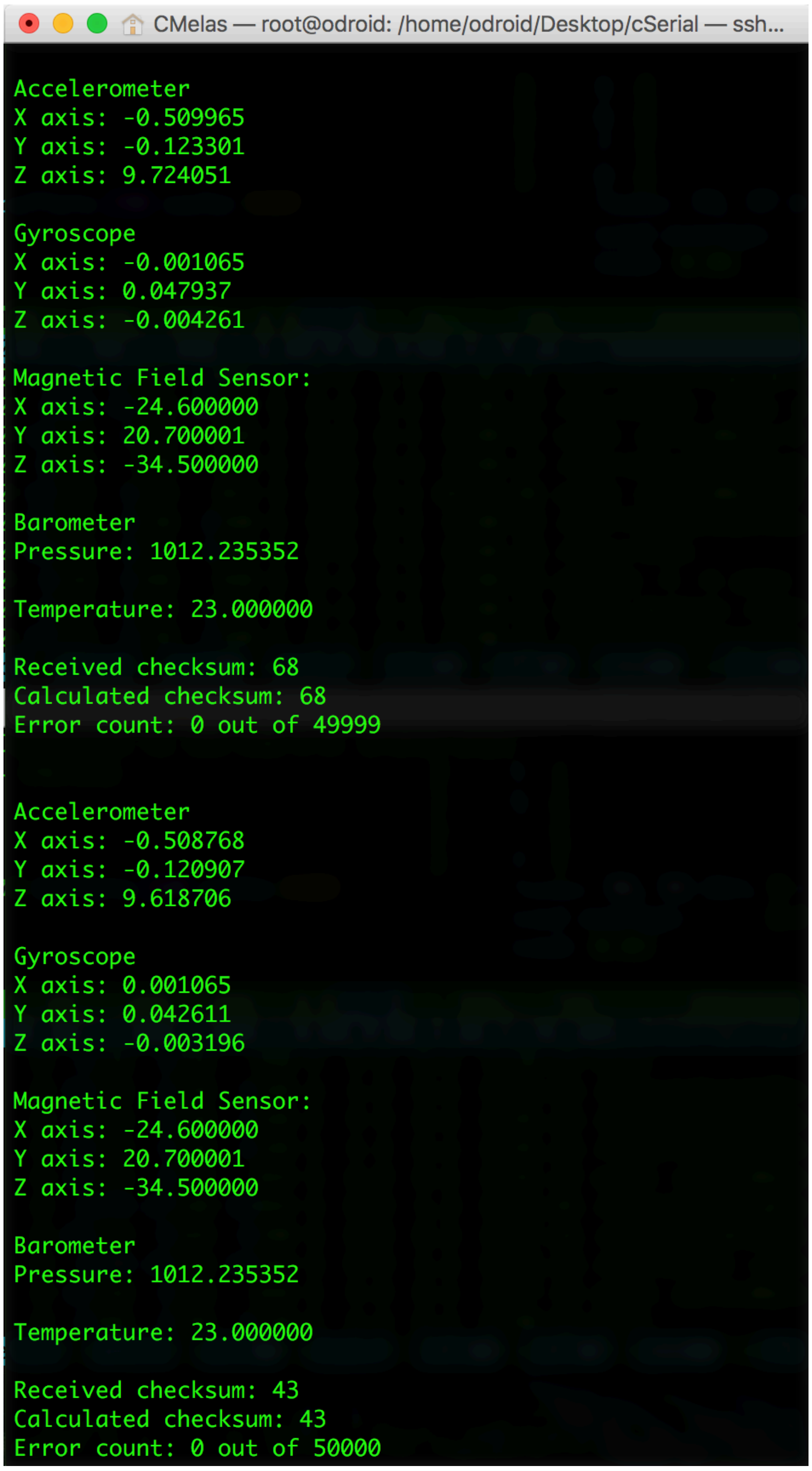
Figure 5.1: Testing the transmission rate

A wake lock mechanism was used in order to keep the device stay on during the execution of the application. Tests were conducted when the Android device was in various states operation. For example, the application was initially in the foreground of the screen with the screen on and the device unlocked. Later, while the application was left running, the screen went off and the device was locked. After some time, a phone call was made to the smartphone, in order to verify that the data rate would not be affected. Finally, the device was unlocked and other applications were launched, leaving in the background the application that is responsible for the transmission of the sensor measurements to the Odroid platform. The transmission rate was not reduced under any of these circumstances under the typical rate fluctuation of about 5%. However, these tests apply only to the smartphone that was used, a Samsung Galaxy S6, and may be different to other Android devices.

Another important issue apart from the transmission rate is the integrity of the data. Receiving false measurements could lead the autopilot software to miscalculations, which could result even in a disastrous event. The received data had to be verified, in order to make sure that they match the sensor measurements that the Android device had produced.

The solution derived from the structure of the NMEA sentences. In the end of each sentence, there are two characters which represent a simple checksum of the NMEA data in a hexadecimal format. More specifically, each byte of the sentence between the "$" and the "*" is XORed, and the 8-bit result is represented by two hexadecimal numbers. Similarly, each sensor measurements block contains a single byte, byte 48, that is the result of the XOR of all the bytes of the sensor measurements, bytes 4 to 47. This is a fast and lightweight implementation of a simple checksum function, as it has to be performed every time a transmission is made. The result had to be small in size in order to append it to the sensor measurements block and maintain a relatively small block size. From the side of the Odroid platform, when a new transmission is received all bytes from byte 4 to byte 47 are XORed, and the result is compared with byte 48, which is the result of the checksum function that was performed on the Android device. If the two numbers are the same, then the data are forwarded to the appropriate virtual serial ports, otherwise, the received data are ignored.

After several tests, it was found that the serial communication is reliable, without occurring corruptions in the transmissions. In Figure 5.2, a sample test is shown where 50000 transmissions were performed from the Android device to the Odroid platform. The serial communication is functioning as it was expected to be.



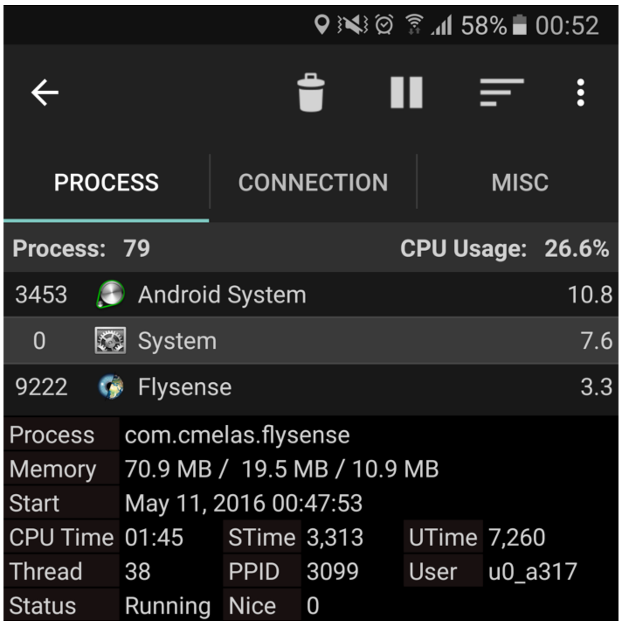
```
Accelerometer
X axis: -0.509965
Y axis: -0.123301
Z axis: 9.724051

Gyroscope
X axis: -0.001065
Y axis: 0.047937
Z axis: -0.004261

Magnetic Field Sensor:
X axis: -24.600000
Y axis: 20.700001
Z axis: -34.500000

Barometer
Pressure: 1012.235352

Temperature: 23.000000

Received checksum: 68
Calculated checksum: 68
Error count: 0 out of 49999


Accelerometer
X axis: -0.508768
Y axis: -0.120907
Z axis: 9.618706

Gyroscope
X axis: 0.001065
Y axis: 0.042611
Z axis: -0.003196

Magnetic Field Sensor:
X axis: -24.600000
Y axis: 20.700001
Z axis: -34.500000

Barometer
Pressure: 1012.235352

Temperature: 23.000000

Received checksum: 43
Calculated checksum: 43
Error count: 0 out of 50000
```

Figure 5.2: Verifying the integrity of the data

## 5.2.2 Resource Utilization of the Application

The next important issue of the Android application that had to be monitored was the resource utilization of the smartphone. The CPU usage, as well as the used memory by the application, was measured using the OS Monitor application. OS Monitor is an open source application that is able to monitor all processes of an Android device. [36]

It was found that the application is capable of using up to 6% of the CPU, and up to 130 Mbyte of memory, however, these figures appear rarely. In a typical execution of the application, with a rate of 200 Hz, the CPU usage ranges from 2.5% up to 5%, with an average memory usage of 115 Mbyte. The application uses an insignificant amount of memory, considering the total of 2.8Gbytes of available memory of the device, and a reasonable amount of CPU. The CPU usage quite low, therefore it is not considered as a factor that can downgrade the performance of the device. In Figure 5.3 a screenshot of OS Monitor is displayed, showing an instance of the application running. The process ID of the application is 9222, the CPU usage is at 3.3% and the Resident Set Size (RSS) memory is at 70.9 Mbyte. The RSS memory is the portion of memory occupied by a process that is held in the main memory.



Process: 79 CPU Usage: 26.6%

| | | |
|---|---|---|
| 3453 | Android System | 10.8 |
| 0 | System | 7.6 |
| 9222 | Flysense | 3.3 |

| | | | | | |
|---|---|---|---|---|---|
| Process | com.cmelas.flysense | | | | |
| Memory | 70.9 MB / 19.5 MB / 10.9 MB | | | | |
| Start | May 11, 2016 00:47:53 | | | | |
| CPU Time | 01:45 | STime | 3,313 | UTime | 7,260 |
| Thread | 38 | PPID | 3099 | User | u0_a317 |
| Status | Running | Nice | 0 | | |

Figure 5.3: Monitoring the performance of the application

Finally, from the results of the evaluation of the Android application, it appears that the application is functioning properly, and the resource utilization is at acceptable levels as well.

## 5.3 Autopilot Software

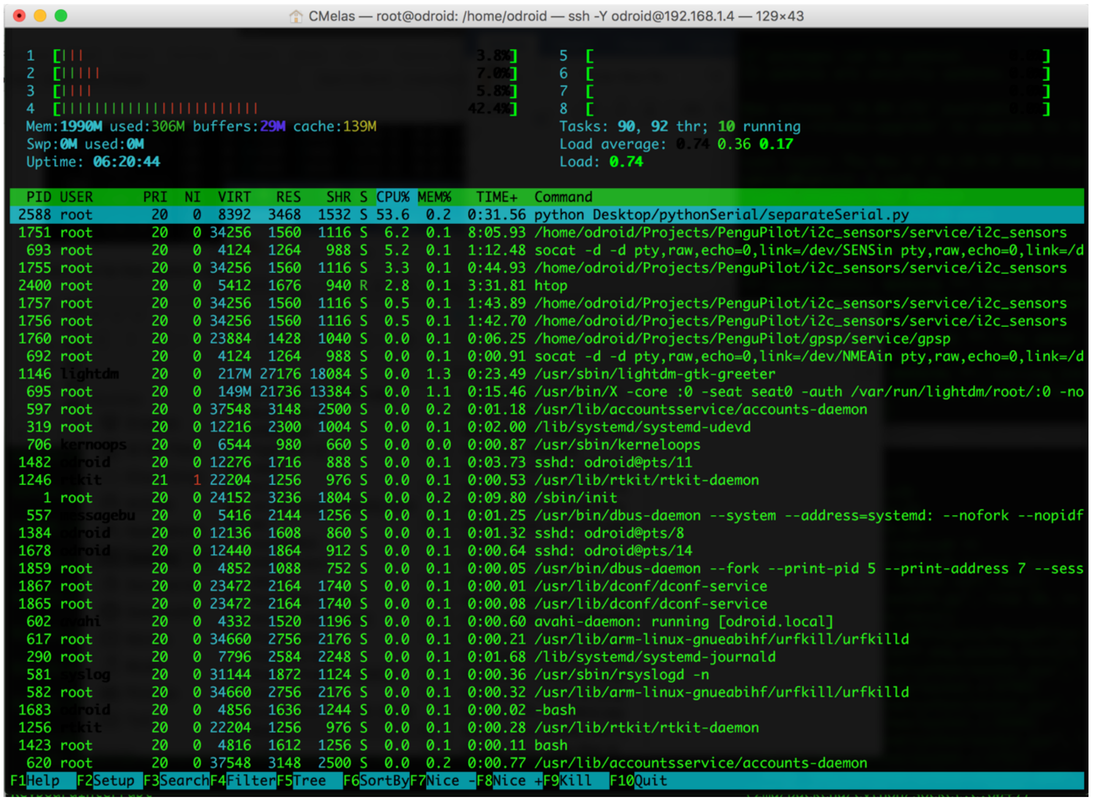
### 5.3.1 Data Handling Software and its Resource Utilization

It was crucial to verify that PenguPilot is able to utilize the data from the Android device. Also, the data had to be in the appropriate format for the autopilot software to operate.

Initially, the appropriate services, the “gpsp” and “i2c_sensors” services that are responsible for the GPS and the sensor measurements, are started. The controlling of the PenguPilot services is conducted by using the “pp_svctrl” utility. With the services running, the program that handles the data is executed. From this point, the services of PenguPilot are receiving the data input they need in order to operate.

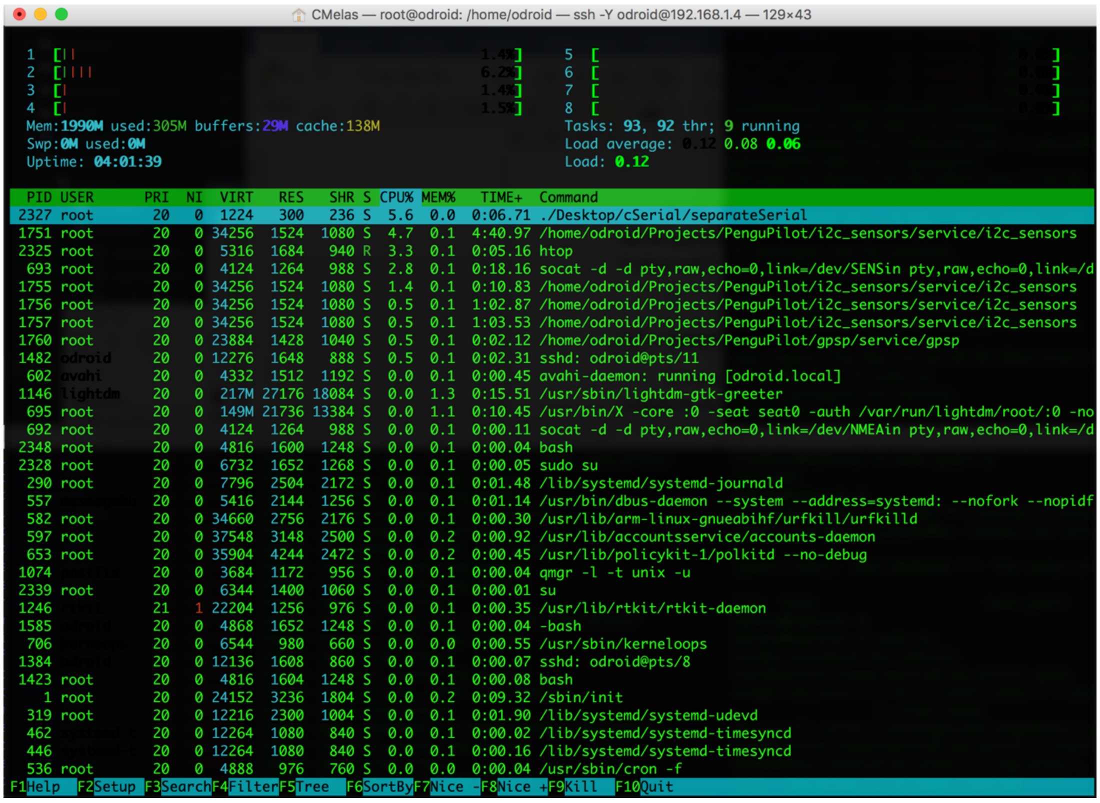
In this step, the resource utilization of the system had to be monitored, in order to verify that no resources are being overused. At first, the program that is responsible for receiving the data from the Android device and forwards them to the appropriate serial ports was developed in Python, due to the simplicity of a high-level programming language. However, it was found that the program used a significant amount of CPU, as well as memory, and for this reason, the program was implemented again in C.

The CPU usage of the C program is ranging from 4% to 6% when the transmission rates reach 200 Hz, while the equivalent program in Python had a CPU usage of up to 54%. The physical memory consumed by the C program is 300 kbytes, in contrast to the 3.468 Mbytes that uses the equivalent Python program. It is clear that the Python implementation uses about 10 times more resources than the implementation in C. In the following figures, Figure 5.4 and Figure 5.5, the resource utilization of the system is presented when the Python program was used and when the C program was used respectively. Other important processes can be seen as well, such as the processes of the “i2c_sensors” and the “gpsp” services, and the two socat processes.



```
CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.4 — 129×43

  1  [|||                                           3.8%]   5  [                                        0.0%]
  2  [|||||                                         7.0%]   6  [                                        0.0%]
  3  [||||                                          5.8%]   7  [                                        0.0%]
  4  [||||||||||||||||||||||||                     42.4%]   8  [                                        0.0%]
  Mem:1990M used:306M buffers:29M cache:139M                Tasks: 90, 92 thr; 10 running
  Swp:0M used:0M                                            Load average: 0.74 0.36 0.17
  Uptime: 06:20:44                                          Load: 0.74

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2588 root       20   0  8392  3468  1532 S 53.6  0.2  0:31.56 python Desktop/pythonSerial/separateSerial.py
 1751 root       20   0 34256  1560  1116 S  6.2  0.1  8:05.93 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
  693 root       20   0  4124  1264   988 S  5.2  0.1  1:12.48 socat -d -d pty,raw,echo=0,link=/dev/SENSin pty,raw,echo=0,link=/d
 1755 root       20   0 34256  1560  1116 S  3.3  0.1  0:44.93 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 2400 root       20   0  5412  1676   940 R  2.8  0.1  3:31.81 htop
 1757 root       20   0 34256  1560  1116 S  0.5  0.1  1:43.89 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 1756 root       20   0 34256  1560  1116 S  0.5  0.1  1:42.70 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 1760 root       20   0 23884  1428  1040 S  0.0  0.1  0:06.25 /home/odroid/Projects/PenguPilot/gpsp/service/gpsp
  692 root       20   0  4124  1264   988 S  0.0  0.1  0:00.91 socat -d -d pty,raw,echo=0,link=/dev/NMEAin pty,raw,echo=0,link=/d
 1146 lightdm    20   0  217M 27176 18084 S  0.0  1.3  0:23.49 /usr/sbin/lightdm-gtk-greeter
  695 root       20   0  149M 21736 13384 S  0.0  1.1  0:15.46 /usr/bin/X -core :0 -seat seat0 -auth /var/run/lightdm/root/:0 -no
  597 root       20   0 37548  3148  2500 S  0.0  0.2  0:01.18 /usr/lib/accountsservice/accounts-daemon
  319 root       20   0 12216  2300  1004 S  0.0  0.1  0:02.00 /lib/systemd/systemd-udevd
  706 kernoops   20   0  6544   980   660 S  0.0  0.0  0:00.87 /usr/sbin/kerneloops
 1482 odroid     20   0 12276  1716   888 S  0.0  0.1  0:03.73 sshd: odroid@pts/11
 1246 rtkit      21   1 22204  1256   976 S  0.0  0.1  0:00.53 /usr/lib/rtkit/rtkit-daemon
    1 root       20   0 24152  3236  1804 S  0.0  0.2  0:09.80 /sbin/init
  557 messagebu  20   0  5416  2144  1256 S  0.0  0.1  0:01.25 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidf
 1384 odroid     20   0 12136  1608   860 S  0.0  0.1  0:01.32 sshd: odroid@pts/8
 1678 odroid     20   0 12440  1864   912 S  0.0  0.1  0:00.64 sshd: odroid@pts/14
 1859 root       20   0  4852  1088   752 S  0.0  0.1  0:00.05 /usr/bin/dbus-daemon --fork --print-pid 5 --print-address 7 --sess
 1867 root       20   0 23472  2164  1740 S  0.0  0.1  0:00.01 /usr/lib/dconf/dconf-service
 1865 root       20   0 23472  2164  1740 S  0.0  0.1  0:00.08 /usr/lib/dconf/dconf-service
  602 avahi      20   0  4332  1520  1196 S  0.0  0.1  0:00.60 avahi-daemon: running [odroid.local]
  617 root       20   0 34660  2756  2176 S  0.0  0.1  0:00.21 /usr/lib/arm-linux-gnueabihf/urfkill/urfkilld
  290 root       20   0  7796  2584  2248 S  0.0  0.1  0:01.68 /lib/systemd/systemd-journald
  581 syslog     20   0 31144  1872  1124 S  0.0  0.1  0:00.36 /usr/sbin/rsyslogd -n
  582 root       20   0 34660  2756  2176 S  0.0  0.1  0:00.32 /usr/lib/arm-linux-gnueabihf/urfkill/urfkilld
 1683 odroid     20   0  4856  1636  1244 S  0.0  0.1  0:00.02 -bash
 1256 rtkit      20   0 22204  1256   976 S  0.0  0.1  0:00.28 /usr/lib/rtkit/rtkit-daemon
 1423 root       20   0  4816  1612  1256 S  0.0  0.1  0:00.11 bash
  620 root       20   0 37548  3148  2500 S  0.0  0.2  0:00.77 /usr/lib/accountsservice/accounts-daemon
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 5.4: Resource utilization when using a Python implementation



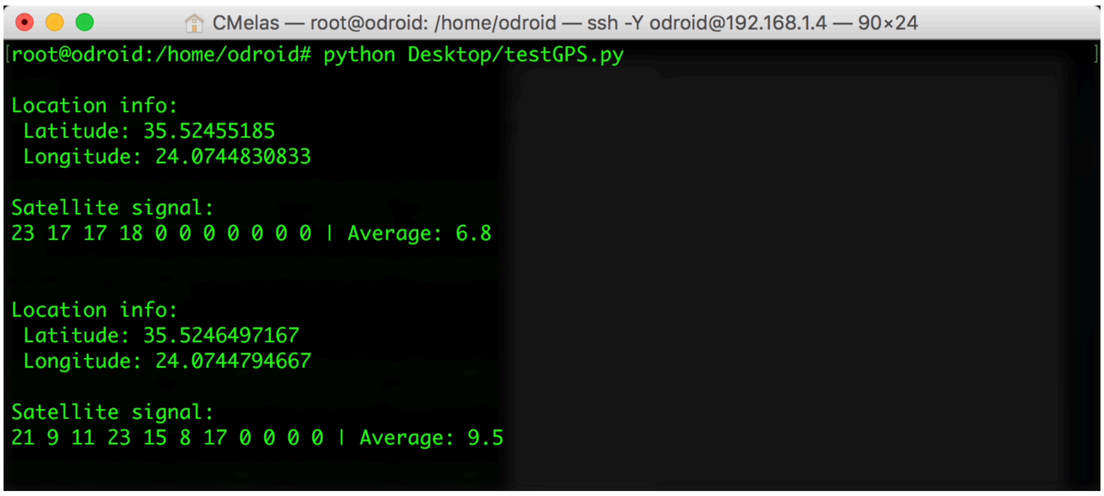
```
CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.4 — 129×43

  1  [||                                            1.4%]   5  [                                        0.0%]
  2  [||||                                          6.2%]   6  [                                        0.0%]
  3  [|                                             1.4%]   7  [                                        0.0%]
  4  [|                                             1.5%]   8  [                                        0.0%]
  Mem:1990M used:305M buffers:29M cache:138M                Tasks: 93, 92 thr; 9 running
  Swp:0M used:0M                                            Load average: 0.12 0.08 0.06
  Uptime: 04:01:39                                          Load: 0.12

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2327 root       20   0  1224   300   236 S  5.6  0.0  0:06.71 ./Desktop/cSerial/separateSerial
 1751 root       20   0 34256  1524  1080 S  4.7  0.1  4:40.97 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 2325 root       20   0  5316  1684   940 R  3.3  0.1  0:05.16 htop
  693 root       20   0  4124  1264   988 S  2.8  0.1  0:18.16 socat -d -d pty,raw,echo=0,link=/dev/SENSin pty,raw,echo=0,link=/d
 1755 root       20   0 34256  1524  1080 S  1.4  0.1  0:10.83 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 1756 root       20   0 34256  1524  1080 S  0.5  0.1  1:02.87 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 1757 root       20   0 34256  1524  1080 S  0.5  0.1  1:03.53 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 1760 root       20   0 23884  1428  1040 S  0.5  0.1  0:02.12 /home/odroid/Projects/PenguPilot/gpsp/service/gpsp
 1482 odroid     20   0 12276  1648   888 S  0.5  0.1  0:02.31 sshd: odroid@pts/11
  602 avahi      20   0  4332  1512  1192 S  0.0  0.1  0:00.45 avahi-daemon: running [odroid.local]
 1146 lightdm    20   0  217M 27176 18084 S  0.0  1.3  0:15.51 /usr/sbin/lightdm-gtk-greeter
  695 root       20   0  149M 21736 13384 S  0.0  1.1  0:10.45 /usr/bin/X -core :0 -seat seat0 -auth /var/run/lightdm/root/:0 -no
  692 root       20   0  4124  1264   988 S  0.0  0.1  0:00.11 socat -d -d pty,raw,echo=0,link=/dev/NMEAin pty,raw,echo=0,link=/d
 2348 root       20   0  4816  1600  1248 S  0.0  0.1  0:00.04 bash
 2328 root       20   0  6732  1652  1268 S  0.0  0.1  0:00.05 sudo su
  290 root       20   0  7796  2504  2172 S  0.0  0.1  0:01.48 /lib/systemd/systemd-journald
  557 messagebu  20   0  5416  2144  1256 S  0.0  0.1  0:01.14 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidf
  582 root       20   0 34660  2756  2176 S  0.0  0.1  0:00.30 /usr/lib/arm-linux-gnueabihf/urfkill/urfkilld
  597 root       20   0 37548  3148  2500 S  0.0  0.2  0:00.92 /usr/lib/accountsservice/accounts-daemon
  653 root       20   0 35904  4244  2472 S  0.0  0.2  0:00.45 /usr/lib/policykit-1/polkitd --no-debug
 1074 postfix    20   0  3684  1172   956 S  0.0  0.1  0:00.04 qmgr -l -t unix -u
 2339 root       20   0  6344  1400  1060 S  0.0  0.1  0:00.01 su
 1246 rtkit      21   1 22204  1256   976 S  0.0  0.1  0:00.35 /usr/lib/rtkit/rtkit-daemon
 1585 odroid     20   0  4868  1652  1248 S  0.0  0.1  0:00.04 -bash
  706 kernoops   20   0  6544   980   660 S  0.0  0.0  0:00.55 /usr/sbin/kerneloops
 1384 odroid     20   0 12136  1608   860 S  0.0  0.1  0:00.07 sshd: odroid@pts/8
 1423 root       20   0  4816  1604  1248 S  0.0  0.1  0:00.08 bash
    1 root       20   0 24152  3236  1804 S  0.0  0.2  0:09.32 /sbin/init
  319 root       20   0 12216  2300  1004 S  0.0  0.1  0:01.90 /lib/systemd/systemd-udevd
  462 systemd-t  20   0 12264  1080   840 S  0.0  0.1  0:00.02 /lib/systemd/systemd-timesyncd
  446 systemd-t  20   0 12264  1080   840 S  0.0  0.1  0:00.16 /lib/systemd/systemd-timesyncd
  536 root       20   0  4888   976   760 S  0.0  0.0  0:00.04 /usr/sbin/cron -f
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 5.5: Resource utilization when using a C implementation

### 5.3.2 Verification of the “gpsp” and “i2c_sensors” Services

As there were no performance issues with the C implementation of that program, the next step was to verify that the “gpsp” and the “i2c_sensors” services can provide successfully the appropriate data to other services of PenguPilot. In order to verify the functionality of the “gpsp” and “i2c_sensors” services, some test scripts were created.

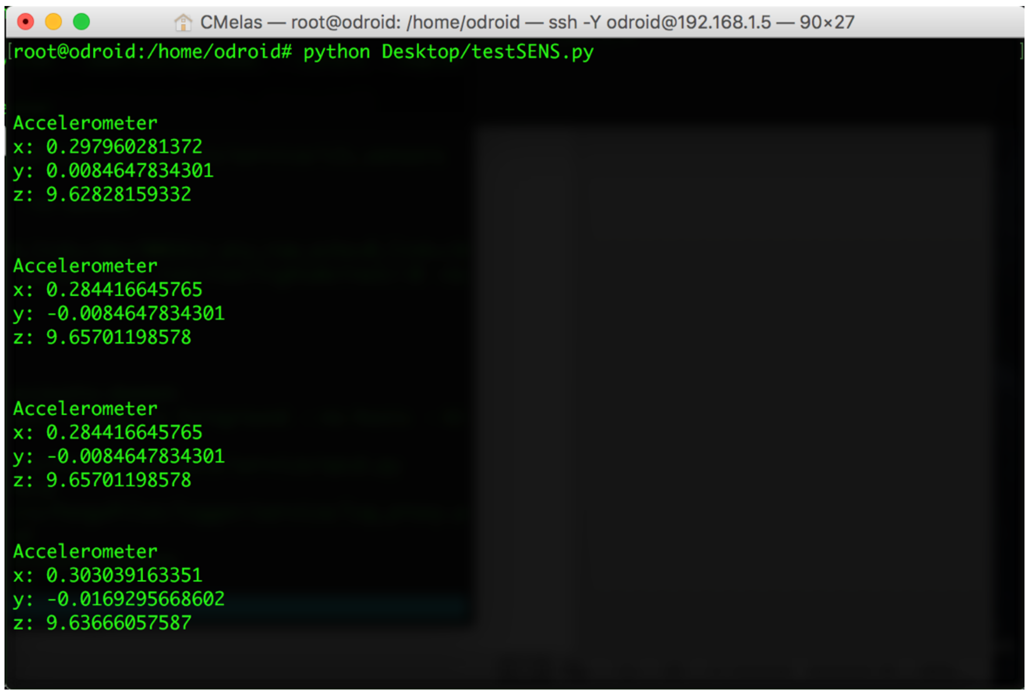
For the verification of the “gpsp” service, a Python script was developed, with similar functionality to the “gps_print.py” and to “print_sat_signals.py” scripts of PenguPilot. The program uses the SCL subsystem of PenguPilot and opens the “gps” and “sats” sockets. When new location information is available, the latitude, the longitude, and the information about the signal of the satellites will be printed on the screen. In the best case, new location information is available once every second, however, if no new location information is available, or if it is unchanged, nothing will be printed on the screen. An example of the script that verifies the successful functionality of the “gpsp” service can be seen in Figure 5.6.


CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.4 — 90×24

```
root@odroid:/home/odroid# python Desktop/testGPS.py

Location info:
 Latitude: 35.52455185
 Longitude: 24.0744830833

Satellite signal:
23 17 17 18 0 0 0 0 0 0 0 | Average: 6.8


Location info:
 Latitude: 35.5246497167
 Longitude: 24.0744794667

Satellite signal:
21 9 11 23 15 8 17 0 0 0 0 | Average: 9.5
```

Figure 5.6: Verifying the functionality of the "gpsp" service

Similarly, the functionality of the “i2c_sensors” service was tested and verified with a Python program. The test program uses the Signaling and Communication Link subsystem of PenguPilot and opens a sensor socket from the ones that were described in section 3.3.3 in order to receive the sensor measurements. Since no calibration has been applied yet, one of the raw sockets, such as the “acc_raw”, “gyro_raw”, mag_raw” and “baro_raw” was selected in this step. In Figure 5.7 the output of the test program is shown, in which the accelerometer measurements are obtained by the “acc_raw” socket.


CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.5 — 90×27

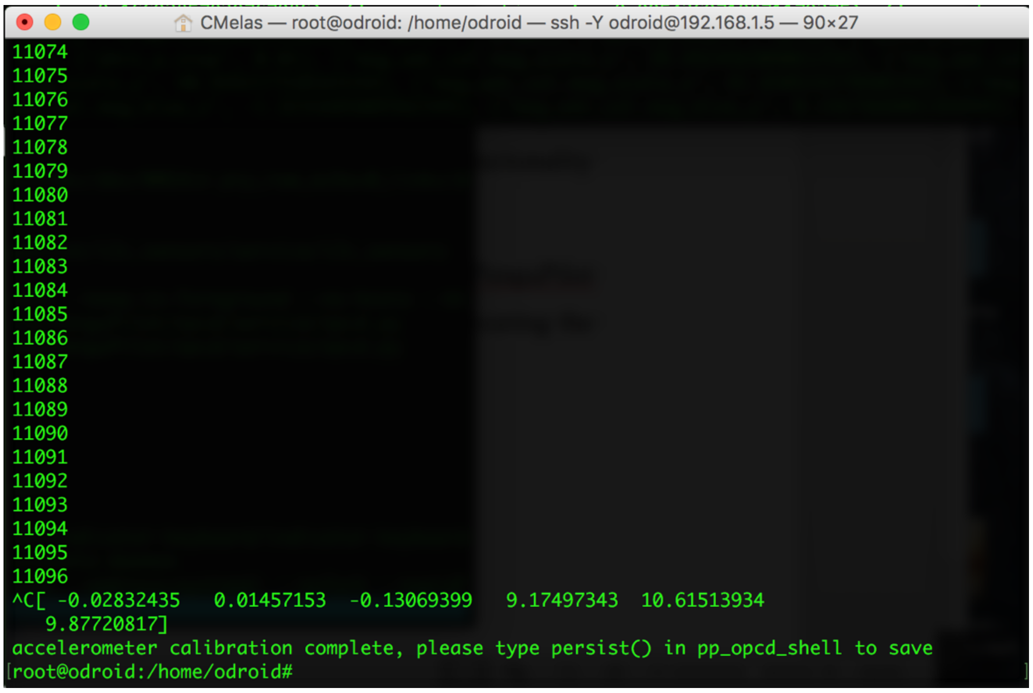
```
root@odroid:/home/odroid# python Desktop/testSENS.py


Accelerometer
x: 0.297960281372
y: 0.0084647834301
z: 9.62828159332

Accelerometer
x: 0.284416645765
y: -0.0084647834301
z: 9.65701198578

Accelerometer
x: 0.284416645765
y: -0.0084647834301
z: 9.65701198578

Accelerometer
x: 0.303039163351
y: -0.0169295668602
z: 9.63666057587
```

Figure 5.7: Verifying the functionality of the "i2c_sensors" service

### 5.3.3 Calibrating the Sensors

After the verification of the “i2c_sensors” service, the calibration tools of PenguPilot were used in order to calibrate the accelerometer and the magnetic field sensor. Using these calibration tools successfully was a further way to verify the functionality of the “i2c_sensors” service.

The calibration of the sensors is possible if the appropriate tools of PenguPilot are used. For example, the calibration of the accelerometer is made by executing the “pp_acc_cal” command. Every side of the device has to be placed towards the ground for a few seconds, in order the calibration tool to acquire enough samples of the sensor measurements. Then, the scale and the bias of each axis of the sensor is calculated, and PenguPilot can use these values in order to calculate the calibrated accelerometer measurements. The calibration of the accelerometer is a fine procedure, which has to be done carefully and with patience in order to acquire the most accurate calibration as possible. The results of a sample calibration that was conducted are shown in Figure 5.8. The first three numbers refer to the bias of the X, Y, and Z axis, and the last three numbers refer to the scale of the X, Y, and Z axis respectively.

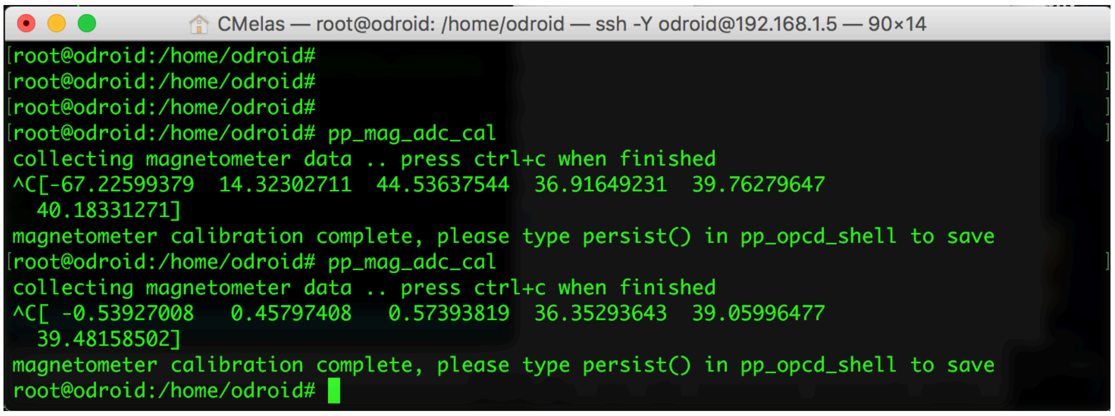

CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.5 — 90×27

```
11074
11075
11076
11077
11078
11079
11080
11081
11082
11083
11084
11085
11086
11087
11088
11089
11090
11091
11092
11093
11094
11095
11096
^C[ -0.02832435   0.01457153  -0.13069399   9.17497343  10.61513934
    9.87720817]
accelerometer calibration complete, please type persist() in pp_opcd_shell to save
[root@odroid:/home/odroid#
```

Figure 5.8: Calibrating the accelerometer

Similarly, the calibration of the magnetic field sensor was conducted by using the “pp_mag_adc_cal” command. The calibration tool for the magnetic field sensor is acquiring samples from the measurements of the sensor and then it calculates the bias and the scale of the three coordinate axes of the Android device. Unlike the calibration of the accelerometer, the Android device has to be moved in a figure-8 pattern in order to acquire samples from the intensity of the magnetic field in every direction. The result of two calibration procedures for the magnetic field sensor is shown in Figure 5.9. In the first execution of the calibration tool, the option of the “Calibrated Magnetometer” was disabled from the Android application. The first three numbers are the bias of the X, Y and Z axis, while the last three numbers are the scale of the three coordinate axes respectively. When the option of the “Calibrated Magnetometer” is enabled, the bias of the axes is nearly zero, as the Android device is capable providing the magnetic field measurements without any bias.

The results of the calibration procedures, both for the accelerometer and the magnetic field sensor, can be stored in the “params.yaml” file by calling the “persist()” function in the “pp_opcd_shell”. Next, the “acc_cal” and the “mag_adc_cal” services can be started, which will provide the calibrated measurements in the “acc” and “mag_adc_cal” sockets of SCL respectively.


CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.5 — 90×14

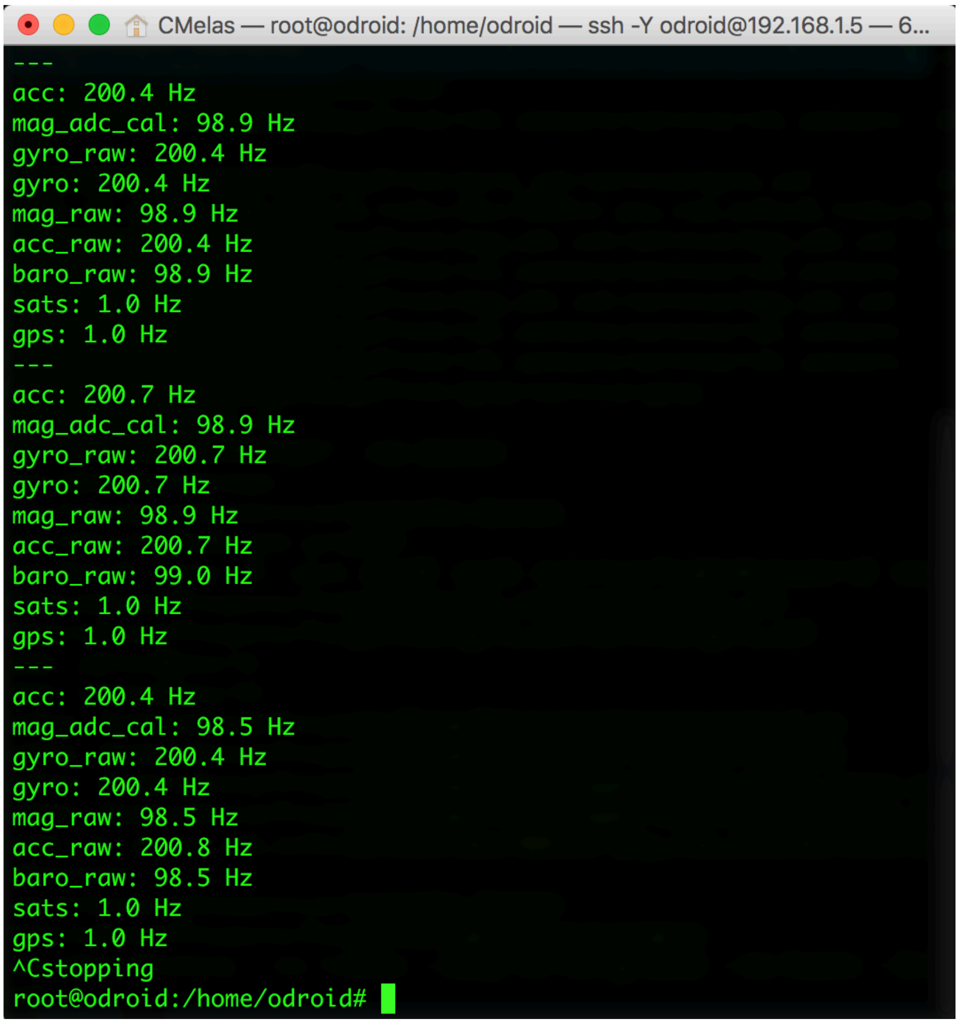
```
root@odroid:/home/odroid#
root@odroid:/home/odroid#
root@odroid:/home/odroid#
root@odroid:/home/odroid# pp_mag_adc_cal
collecting magnetometer data .. press ctrl+c when finished
^C[-67.22599379  14.32302711  44.53637544  36.91649231  39.76279647
  40.18331271]
magnetometer calibration complete, please type persist() in pp_opcd_shell to save
root@odroid:/home/odroid# pp_mag_adc_cal
collecting magnetometer data .. press ctrl+c when finished
^C[ -0.53927008   0.45797408   0.57393819  36.35293643  39.05996477
  39.48158502]
magnetometer calibration complete, please type persist() in pp_opcd_shell to save
root@odroid:/home/odroid#
```

Figure 5.9: Calibrating the magnetic field sensor

In order to acquire calibrated measurements from the gyroscope, the “gyro_cal” service has to be started. There is no calibration procedure that has to be applied to the gyroscope before starting the “gyro_cal” service. When the “gyro_cal” service is running the calibrated measurements of the gyroscope will be published at the “gyro” socket of the SCL subsystem of PenguPilot.

There is no calibration for the barometer, however, instead of publishing the raw atmospheric pressure, PenguPilot calculates the altitude by using the hypsometric formula with the standard sea level conditions of pressure and temperature. Finally, the data derived from the measurements of the barometer are available at the “baro_raw” socket of the SCL subsystem.

The update rate of the sockets of the SCL subsystem can be seen by using the “pp_scl_show_stats” command. It has to be noted that these rates are not the transmission rates of the Android device, but the update rates of the processes responsible for the sensor measurements. For example, in Figure 5.10 the data for the barometer socket indicate an update rate of almost 100 Hz, however, the true update rate is 5.5 Hz. Therefore, most of the published measurements will be duplicate as the Android device is not capable of providing such a high update rate for the barometer.


CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.5 — 6...

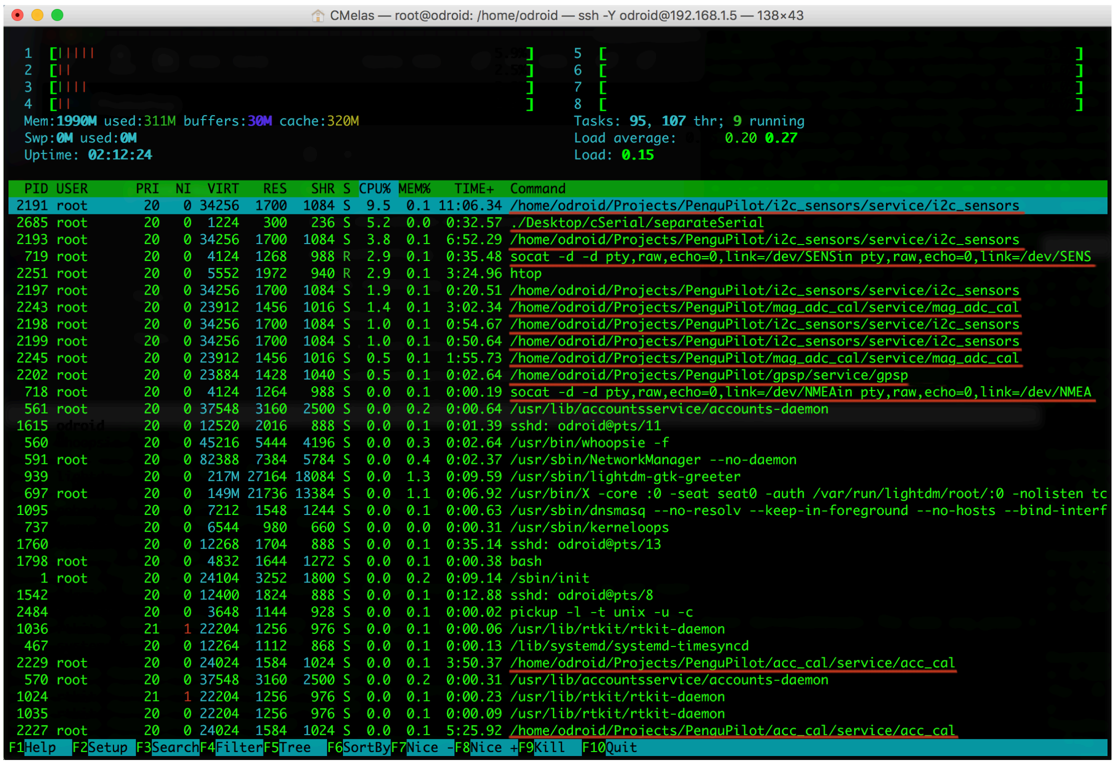
```
---
acc: 200.4 Hz
mag_adc_cal: 98.9 Hz
gyro_raw: 200.4 Hz
gyro: 200.4 Hz
mag_raw: 98.9 Hz
acc_raw: 200.4 Hz
baro_raw: 98.9 Hz
sats: 1.0 Hz
gps: 1.0 Hz
---
acc: 200.7 Hz
mag_adc_cal: 98.9 Hz
gyro_raw: 200.7 Hz
gyro: 200.7 Hz
mag_raw: 98.9 Hz
acc_raw: 200.7 Hz
baro_raw: 99.0 Hz
sats: 1.0 Hz
gps: 1.0 Hz
---
acc: 200.4 Hz
mag_adc_cal: 98.5 Hz
gyro_raw: 200.4 Hz
gyro: 200.4 Hz
mag_raw: 98.5 Hz
acc_raw: 200.8 Hz
baro_raw: 98.5 Hz
sats: 1.0 Hz
gps: 1.0 Hz
^Cstopping
root@odroid:/home/odroid#
```

Figure 5.10: Statistics of the Signaling and Communication Link

### 5.3.4 Overall Resource Utilization on the Odroid Platform

The final part of the verification and evaluation process was to examine the overall resource utilization of the system on the Odroid platform. Therefore, all necessary services software and services that provide calibrated sensor data to the components of PenguPilot had to be running. This includes the C program which receives the data from the Android device via the serial port and forwards them to the appropriate virtual serial ports, the two socat processes that create the two virtual serial port pairs, and the “gpsp”, “i2c_sensors”, “acc_cal”, “gyro_cal”, and “mag_adc_cal” services of PenguPilot.

It was found that the services and the software running on the Odroid platform require only a fraction of the total CPU and memory resources. More specifically, the CPU utilization of the whole system was always under 0.35, out of a maximum total load of 8.00, as the system is octa-core. As for the memory usage, the average memory utilization was always under 700 Mbytes, leaving about 1.3 Gbytes free of the total of 1990 Mbytes of memory. These figures are referred to the total resource utilization of the system, including the system processes. A sample of the resource utilization of the system is shown in Figure 5.11, where all processes related to PenguPilot and the data handling are underlined.


CMelas — root@odroid: /home/odroid — ssh -Y odroid@192.168.1.5 — 138×43

```
  1  [|||||                                                 ]     5  [                                                       ]
  2  [||                                                    ]     6  [                                                       ]
  3  [||||                                                  ]     7  [                                                       ]
  4  [||                                                    ]     8  [                                                       ]
  Mem:1990M used:311M buffers:30M cache:320M                     Tasks: 95, 107 thr; 9 running
  Swp:0M used:0M                                                 Load average:      0.20 0.27
  Uptime: 02:12:24                                               Load: 0.15

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 2191 root       20   0 34256  1700  1084 S  9.5  0.1 11:06.34 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 2685 root       20   0  1224   300   236 S  5.2  0.0  0:32.57 ./Desktop/cSerial/separateSerial
 2193 root       20   0 34256  1700  1084 S  3.8  0.1  6:52.29 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
  719 root       20   0  4124  1268   988 R  2.9  0.1  0:35.48 socat -d -d pty,raw,echo=0,link=/dev/SENSin pty,raw,echo=0,link=/dev/SENS
 2251 root       20   0  5552  1972   940 R  2.9  0.1  3:24.96 htop
 2197 root       20   0 34256  1700  1084 S  1.9  0.1  0:20.51 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 2243 root       20   0 23912  1456  1016 S  1.4  0.1  3:02.34 /home/odroid/Projects/PenguPilot/mag_adc_cal/service/mag_adc_cal
 2198 root       20   0 34256  1700  1084 S  1.0  0.1  0:54.67 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 2199 root       20   0 34256  1700  1084 S  1.0  0.1  0:50.64 /home/odroid/Projects/PenguPilot/i2c_sensors/service/i2c_sensors
 2245 root       20   0 23912  1456  1016 S  0.5  0.1  1:55.73 /home/odroid/Projects/PenguPilot/mag_adc_cal/service/mag_adc_cal
 2202 root       20   0 23884  1428  1040 S  0.5  0.1  0:02.64 /home/odroid/Projects/PenguPilot/gpsp/service/gpsp
  718 root       20   0  4124  1264   988 S  0.0  0.1  0:00.19 socat -d -d pty,raw,echo=0,link=/dev/NMEAin pty,raw,echo=0,link=/dev/NMEA
  561 root       20   0 37548  3160  2500 S  0.0  0.2  0:00.64 /usr/lib/accountsservice/accounts-daemon
 1615 odroid     20   0 12520  2016   888 S  0.0  0.1  0:01.39 sshd: odroid@pts/11
  560            20   0 45216  5444  4196 S  0.0  0.3  0:02.64 /usr/bin/whoopsie -f
  591 root       20   0 82388  7384  5784 S  0.0  0.4  0:02.37 /usr/sbin/NetworkManager --no-daemon
  939            20   0  217M 27164 18084 S  0.0  1.3  0:09.59 /usr/sbin/lightdm-gtk-greeter
  697 root       20   0  149M 21736 13384 S  0.0  1.1  0:06.92 /usr/bin/X -core :0 -seat seat0 -auth /var/run/lightdm/root/:0 -nolisten tc
 1095            20   0  7212  1548  1244 S  0.0  0.1  0:00.63 /usr/sbin/dnsmasq --no-resolv --keep-in-foreground --no-hosts --bind-interf
  737            20   0  6544   980   660 S  0.0  0.0  0:00.31 /usr/sbin/kerneloops
 1760            20   0 12268  1704   888 S  0.0  0.1  0:35.14 sshd: odroid@pts/13
 1798 root       20   0  4832  1644  1272 S  0.0  0.1  0:00.38 bash
    1 root       20   0 24104  3252  1800 S  0.0  0.2  0:09.14 /sbin/init
 1542            20   0 12400  1824   888 S  0.0  0.1  0:12.88 sshd: odroid@pts/8
 2484            20   0  3648  1144   928 S  0.0  0.1  0:00.02 pickup -l -t unix -u -c
 1036            21   1 22204  1256   976 S  0.0  0.1  0:00.06 /usr/lib/rtkit/rtkit-daemon
  467            20   0 12264  1112   868 S  0.0  0.1  0:00.13 /lib/systemd/systemd-timesyncd
 2229 root       20   0 24024  1584  1024 S  0.0  0.1  3:50.37 /home/odroid/Projects/PenguPilot/acc_cal/service/acc_cal
  570 root       20   0 37548  3160  2500 S  0.0  0.2  0:00.31 /usr/lib/accountsservice/accounts-daemon
 1024            21   1 22204  1256   976 S  0.0  0.1  0:00.23 /usr/lib/rtkit/rtkit-daemon
 1035            20   0 22204  1256   976 S  0.0  0.1  0:00.09 /usr/lib/rtkit/rtkit-daemon
 2227 root       20   0 24024  1584  1024 S  0.0  0.1  5:25.92 /home/odroid/Projects/PenguPilot/acc_cal/service/acc_cal
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 5.11: Total resource utilization on the Odroid platform

The proper functionality of the whole system is guaranteed, as the verification of the system was completed successfully. It is clear that under any circumstances both the Android application and the Flight Controller will be able to operate and communicate reliably, which is a very important issue for a UAV application.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions from this Work

In this thesis an Android application, as well as the appropriate software for the Flight Controller was developed, in order to unify the required sensors and the GPS receiver of a UAV into a single device. By replacing the IMU and the GPS receiver of a UAV with an Android device, not only less external components are required for a UAV, but also the possibility of taking advantage of other features of the Android device is opened. Most importantly, this is the first approach of combining an Android device with a Linux based autopilot software, as all other related works that were presented in the second chapter are microcontroller based systems.

The Android application that was developed, is able to run on all modern Android devices featuring an accelerometer, a gyroscope, a magnetic field sensor and a barometer. The application uses only a small amount of resources, therefore, the user does not have to own a high end device in order to run it. However, the device has to have a high enough sensor update rate in order to cooperate successfully with the autopilot software. The battery consumption of the Android application was not measured for the following two reasons. Firstly, each device has different hardware specifications and battery capacity, therefore measuring the battery consumption for a single device would be pointless. Secondly, the average flight time of a UAV cannot exceed 30 minutes, which is a very small amount of time, compared to the battery life of a smartphone.

Additionally, an open source Linux based autopilot software was selected, as it is a software which is still evolving, and even the work of this thesis could be considered as an extension of its functionality. PenguPilot was not modified from the aspect of removing portions of code and replacing them with others. All of the code that was required for PenguPilot to cooperate with an Android device was added as additional libraries to its appropriate components. The tests that were conducted after the development of the required software verified the operation of the new system, where the Android device was seamlessly integrated into the existing autopilot software.

It was also found that using a low level programming language for a specific task is much more efficient than using a high level programming language. This was proved in the fifth chapter, when a significantly smaller resource utilization from the execution of C program was measured, in contrast to the execution of the equivalent program in Python. Furthermore, C has also the advantage of speed over Python. Python is an interpreted language and requires a longer execution time for a specific program, compared to the execution time of the equivalent program in C. Although it is much easier to work with a high level programming language, when it is important to maintain a low CPU and memory load, preferring a low level programming language is the best option.

Furthermore, the communication channel between the Android device and the Flight Controller that was established, has the potential of transmitting and receiving additional data apart from the sensor measurements and the GPS data, allowing functionality to be added in the future. Currently, it is calculated that for the transmission of the sensor measurements and the GPS data, up to 90,000 bps are required. Therefore, from the baud rate of the 115,200 bps, more than 25,000 bps are available for the exchange of additional data between the Android device and the Flight Controller.

## 6.2 Future Work

The successful combination of two distinct, and different systems into a modern UAV system was the first and fundamental step for the development of many new applications, which will be able to combine the hardware and the processing power of both the Android device and the Flight Controller.

The possibilities are endless for developing new software that will run on the Flight Controller, as the Flight Controller is hosted on a powerful Linux based platform, in contrast to other common UAV platforms. Therefore, high level programming languages can be used in order to develop software that could perform a variety of functions, and take advantage of the processing power of the octa-core system.

Furthermore, additional hardware can be added to the system, such as external sensors and cameras. These components can be utilised for providing additional accuracy to the flight control or for monitoring the environment and providing additional functionality to the user. For example, ultrasonic sensors can provide the feature of collision avoidance to the UAV. Other external sensors, such as humidity, chemical, and radiological sensors, can provide measurements of the environmental conditions of the area where the UAV is, avoiding the presence of humans to a potentially dangerous area. Furthermore, the UAV can be programmed to fly a specific route, acquiring measurements, and even upload them online by using the cellular data network of the Android device. A UAV can become an online portable measurement station, saving the need for fixed ground stations. Finally, the camera of the Android device, can be used for taking pictures from the flight and store them into the storage of the device, or even upload them to an online server.

Moreover, there are some features that could be implemented to the system without adding extra hardware, such as an emergency landing plan when the Android device is running low on battery. A special data packet could be transmitted to the Flight Controller via the existing serial communication, which would inform the autopilot software that the battery of the Android device is low, and that it should proceed to an emergency landing. The emergency landing script would be executed from the side of the Flight Controller, and a message would be sent to the user containing the geographical coordinates of the position of the UAV. The message could be either a simple SMS message, or a message to a web server, by using the cellular data network.

# Bibliography

[1] P. Partsinevelos, I. Agadakos, V. Athanasiou, I. Papaefstathiou, S. Mertikas, S. Kyritsis, A. Tripolitsiotis, and P. Zervos. On-board computational efficiency in real time UAV embedded terrain reconstruction, 2014

[2] PenguPilot. A GNU/Linux based Multi-Rotor UAV Autopilot, [Online], https://github.com/PenguPilot/PenguPilot

[3] Hardkernel. ODROID-XU3 Lite, [Online], http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141351880955

[4] AndroCopter. The flying smartphone, [Online], https://github.com/romainbaud/andro-copter

[5] xCraft. PhoneDrone Ethos, [Online], http://store.xcraft.io/xplusone-store-xcraft-vtol-drone-sandpoint-id/phonedrone

[6] GRASP Laboratory. Smartphones Power Flying Robots, [Online], https://www.grasp.upenn.edu/projects/smartphones-power-flying-robots-vijay-kumar-lab

[7] M. Leichtfried, C. Kaltenriner, A. Mossel and H. Kaufmann. Autonomous Flight using a Smartphone as On-Board Processing Unit in GPS-Denied Environments, 2014

[8] Flone. A platform for smartphones that can fly, [Online], http://flone.cc

[9] P. Marwedel. Embedded System Design, 2011

[10] H. Kopetz. Real-Time Systems – Design Principles for Distributed Embedded Applications, 1997

[11] F. Huberts. NMEA Library. [Online], https://github.com/AHR-Project/nmealib

[12] Sirf Technology Inc.. NMEA Reference Manual, 2007, [Online], http://usglobalsat.com/store/downloads/NMEA_commands.pdf

[13] Android Developers. Reference API Class Index, 2016, [Online], http://developer.android.com/reference/classes.html

[14] InvenSense. MPU-6500 Product Specification, 2014, [Online], http://43zrtwysvxb2gf29r5o0athu.wpengine.netdna-cdn.com/wp-content/uploads/2015/02/MPU-6500-Datasheet2.pdf

[15] Yamaha Corporation. YAS537 Magnetic Field Sensor, 2014, [Online], http://www.willow.co.uk/html/components/com_virtuemart/shop_image/product/Duplicates/pdfs/MS-3T_YAS537_PBAS537A-000-01-e_Jul15.pdf

[16] STMicroelectronics. LPS25H Datasheet, 2014, [Online], http://www.st.com/web/en/resource/technical/document/datasheet/DM00066332.pdf

[17] DJI. Phantom 4 Specs, 2016, [Online], http://www.dji.com/product/phantom-4/info#specs

[18] Wikipedia. Atmospheric pressure, Altitude variation, 2016, [Online], https://en.wikipedia.org/wiki/Atmospheric_pressure

[19] Wikipedia. Agile software development, The Agile Manifesto, 2016, [Online] https://en.wikipedia.org/wiki/Agile_software_development#The_Agile_Manifesto

[20] Agile Alliance. 12 Principles Behind the Agile Manifesto, 2015, [Online], https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto

[21] G. v. d. Vyver, A. Koronios, M. Lane. Agile methodologies and the emergence of the agile organization: A software development approach waiting for its time?, 2003

[22] D. Griffiths, D. Griffiths. Oreilly Head First Android Development 1st Edition, 2015

[23] Android Developers. Wake Lock, 2016, [Online] http://developer.android.com/reference/android/os/PowerManager.WakeLock.html

[24] Android Developers. Sensors Overview, 2016, [Online], http://developer.android.com/guide/topics/sensors/sensors_overview.html

[25] Android Developers. Motion Sensors, 2016, [Online], http://developer.android.com/guide/topics/sensors/sensors_motion.html

[26] Android Developers. Position Sensors, 2016, [Online], http://developer.android.com/guide/topics/sensors/sensors_position.html

[27] Android Developers. Environment Sensors, 2016, [Online], http://developer.android.com/guide/topics/sensors/sensors_environment.html

[28] Android Developers. Location Strategies, 2016, [Online], http://developer.android.com/guide/topics/location/strategies.html

[29] Android Developers. Handler, 2016, [Online], http://developer.android.com/reference/android/os/Handler.html

[30] Android Developers. USB Host, 2016, [Online], http://developer.android.com/guide/topics/connectivity/usb/host.html

[31] F. Herranz. UsbSerial: A serial port driver library for Android v3.0, 2016, [Online], https://felhr85.net/2014/11/11/usbserial-a-serial-port-driver-library-for-android-v2-0

[32] M. Kerrisk. modprobe man page – Linux manual page, 2016, [Online], http://man7.org/linux/man-pages/man8/modprobe.8.html

[33] A. Borchers. Linux Gadget Serial Driver v2.3, 2008, [Online], https://kernel.org/doc/Documentation/usb/gadget_serial.txt

[34] G. Rieger. socat man page, 2016, [Online], http://www.dest-unreach.org/socat/doc/socat.html

[35] MessagePack. It's like JSON. But fast and small, 2016, [Online] http://msgpack.org

[36] K. Lu. OS Monitor for Android, 2015, [Online] https://github.com/eolwral/OSMonitor

# Appendix

PenguPilot Services and their Dependencies:

| | |
|---|---|
| aircomm: | (Depends: opcd, log_proxy) |
| motors: | (Depends: opcd, log_proxy) |
| ahrs: | (Depends: gyro_cal, acc_cal, cmc, geomag, opcd, log_proxy) |
| rs_ctrl_prx: | (Depends: rs_ctrl, opcd, log_proxy) |
| hp_ctrl_prx: | (Depends: hp_ctrl, opcd, log_proxy) |
| mixer_prx: | (Depends: mixer, opcd, log_proxy) |
| acc_rot_neu: | (Depends: ahrs, acc_cal, opcd, log_proxy) |
| flight_detect: | (Depends: acc_rot_neu, motors, opcd, log_proxy) |
| file_logger: | (Depends: log_proxy, opcd, log_proxy) |
| opcd: | (Depends: log_proxy) |
| gps_rel: | (Depends: gpsp, opcd, log_proxy) |
| log_proxy: | |
| pos_speed_est_neu: | (Depends: acc_hpf_neu, gps_rel, opcd, log_proxy) |
| battmon: | (Depends: arduino, opcd, log_proxy) |
| geomag: | (Depends: gpsp, opcd, log_proxy) |
| mavlink: | (Depends: opcd, log_proxy) |
| rs_ctrl: | (Depends: mixer_prx, opcd, log_proxy) |
| vs_ctrl: | (Depends: mixer_prx, pos_speed_est_neu, opcd, log_proxy) |
| vp_ctrl: | (Depends: vs_ctrl_prx, elevmap, opcd, log_proxy) |
| hp_ctrl: | (Depends: hs_ctrl_prx, opcd, log_proxy) |
| gpsp: | (Depends: opcd, log_proxy) |
| i2c_sensors: | (Depends: opcd, log_proxy) |

| | |
|---|---|
| gyro_cal: | (Depends: i2c_sensors, opcd, log_proxy) |
| rp_ctrl: | (Depends: ahrs, rs_ctrl_prx, opcd, log_proxy) |
| hs_ctrl: | (Depends: rp_ctrl_prx, pos_speed_est_neu, opcd, log_proxy) |
| elevmap: | (Depends: gpsp, opcd, log_proxy) |
| wifi_loc: | (Depends: gpsp, wifi_sensor, opcd, log_proxy) |
| wifi_sensor: | (Depends: opcd, log_proxy) |
| blackbox: | (Depends: opcd, log_proxy) |
| acc_cal: | (Depends: i2c_sensors, opcd, log_proxy) |
| autopilot: | (Depends: vp_ctrl_prx, hp_ctrl_prx, opcd, log_proxy) |
| rc_gestures: | (Depends: rc_cal, opcd, log_proxy) |
| gps_logger: | (Depends: gpstime, opcd, log_proxy) |
| autopilot_prx: | (Depends: autopilot, opcd, log_proxy) |
| gpstime: | (Depends: gpsp, opcd, log_proxy) |
| mag_adc_cal: | (Depends: i2c_sensors, opcd, log_proxy) |
| vs_ctrl_prx: | (Depends: vs_ctrl, opcd, log_proxy) |
| hs_ctrl_prx: | (Depends: hs_ctrl, opcd, log_proxy) |
| arduino: | (Depends: opcd, log_proxy) |
| cmc: | (Depends: arduino, mag_adc_cal, opcd, log_proxy) |
| mixer: | (Depends: motors, opcd, log_proxy) |
| ofs: | (Depends: opcd, log_proxy) |
| rc_cal: | (Depends: arduino, opcd, log_proxy) |
| acc_hpf_neu: | (Depends: acc_rot_neu, opcd, log_proxy) |
| emerg_kill: | (Depends: opcd, log_proxy) |
| heartbeat: | (Depends: gpsp, battmon, aircomm, opcd, log_proxy) |
| rp_ctrl_prx: | (Depends: rp_ctrl, opcd, log_proxy) |
| test_service: | (Depends: opcd, log_proxy) |
| vp_ctrl_prx: | (Depends: vp_ctrl, opcd, log_proxy) |
| teensy_taranis: | (Depends: opcd, log_proxy) |