

# GNSS interference detection system

Internship at S[&]T

Intern: Thomas Joesoef Djamil -

Company: Stefan van der Linden -

University: Paul Witte -

*Version 1.0*

*30-05-2022*

## Samenvatting

Voor een stage bij S[&]T moest een embedded systeem worden ontworpen dat GNSS-storingssignalen kan detecteren. Het project dat voor de stage werd gedaan duurde 5 maanden. Gedurende deze 5 maanden werd een embedded systeem ontworpen.

Dit ontworpen systeem werd uiteindelijk Mobile GNSS Interference Detection of kortweg MGID. Voor het systeem is een software ontwerp gemaakt en geschreven. Hiervoor is een elektrisch ontwerp gemaakt inclusief een PCB-ontwerp.

Tests werden gedaan met behulp van ontwikkelaarsborden. Met deze tests werd aangetoond dat het systeem in staat was om interferentie-informatie van de GNSS-ontvanger te krijgen. Het aparte GNSS storings-detectie circuit dat ook was ontworpen, kon niet goed worden getest. Dit zal worden gedaan door toekomstige ontwikkelaars van het project.

# Table of Contents

## Contents

Samenvatting.....	2
Table of Contents.....	3
Dictionary.....	6
1 Introduction .....	7
2 Organization.....	8
3 Order requirements.....	9
3.1 Project borders .....	9
4 Mobile GNSS Interference Detection .....	10
4.1 System goal.....	10
4.2 System concept.....	10
4.3 Appendices.....	10
5 MGID architecture .....	11
5.1 ST Microcontroller .....	11
5.2 ZED-F9P GNSS receiver.....	11
5.3 The host .....	12
5.4 Jamming detection circuit .....	12
5.5 GNSS antenna .....	12
6 Prototyping.....	13
7 Software architecture .....	14
7.1 Core principles.....	14
7.2 Basic readability .....	15
7.3 Main initialization .....	15
7.4 FreeRTOS .....	15
7.5 Interrupts .....	15
7.6 MGID classes .....	16
7.6.1 MGIDCORE .....	16
7.6.2 MGIDTIME .....	16
7.6.3 MGIDUART.....	16
7.6.4 MGIDUARTGNSS.....	16
7.6.5 MGIDTRANSMITTER .....	17
7.6.6 MGIDADC .....	17
7.6.7 MGIDHALPROXY .....	17
7.6.8MGIDUSRLED .....	17
7.7 Direct memory access .....	18

7.8 Connection with host .....	18
7.9 Connection with ZED-F9P .....	18
7.10 Booting the system .....	18
8 MGID electrical design .....	20
8.1 Power supply .....	20
8.2 MCP2221 .....	21
8.3 JTAG Programmer .....	21
8.4 GNSS Jamming Detection .....	21
8.5 GNSS RF input .....	21
8.6 GPIO headers .....	22
8.7 LED feedback lights .....	22
9 PCB Design .....	23
9.1 PCB design principles .....	23
9.2 PCB design rules .....	24
9.3 STM32 .....	25
9.4 Dual pin headers .....	25
9.5 ZED-F9P .....	25
9.6 Jamming circuit detection .....	26
9.7 Choice of components .....	26
10 Communication interface .....	27
10.1 Interfacing with host .....	27
10.2 Command list .....	29
10.2.1 List .....	29
10.2.2 Extended explanation .....	29
11 Processing data with Python scripts .....	30
11.1 Python graphs .....	31
12 Design validation .....	34
12.1 Electrical design .....	34
12.2 PCB design .....	34
12.3 Performed design validations .....	34
13 Future developments .....	36
13.1 Testing of jamming detection circuit .....	36
13.2 Choose housing for the system .....	36
13.3 Creating an order to a PCB manufacturer .....	36
13.4 Testing and validating the PCB prototype in practice .....	36
13.5 Mobile communication possibilities .....	36
13.6 Find ways to integrate the system in other projects .....	37

13.7 Ideal future use of the system.....	37
14 Problems and solutions .....	38
14.1 Long component delivery time .....	38
14.2 Not able to use basic components for detecting RF signals.....	38
14.2.1 Measuring the active antenna power.....	38
14.2.2 Attempting to rectify the AC RF signal.....	38
14.2.3 Using premade IC's.....	39
14.3 Bad initial trace management .....	39
14.4 No possibility of testing with jamming signals .....	40
14.5 Running out of microcontroller memory .....	40
14.6 The float to string problem .....	40
14.6.1 GNSS location .....	40
14.6.2 Satellite pseudoranges.....	41
15 Competencies .....	42
15.1 Analysis.....	42
15.2 Design .....	42
15.3 Realization.....	43
15.4 Managing.....	43
15.5 Research .....	44
16 Conclusion .....	45
References .....	46
Appendix A1: Electrical design.....	47
Appendix A2: PCB design.....	48
Appendix A3: Software main .....	49
Appendix A4: MGIDCORE class .....	50
Appendix A5: Processing NMEA messages.....	51
Appendix A6: MGIDCORE tasks.....	52

## Dictionary

MGID	Mobile GNSS Interference Detector. This is the name of the system designed for the project.
GNSS	Global Navigation Satellite system
UART	Universal asynchronous receiver-transmitter.
PCB	Printed circuit board
trace	PCB term that indicates a copper 'wire' that goes from component to components
via	PCB term that indicates a bridge between the copper layers
CNO	Carrier noise ratio
STM32	The 32-bit microcontroller series from ST microelectronics. This is the microcontroller that is used for the project.
ZED-F9P	GNSS receiver made by u-blox
IDE	Integrated Developer Environment
UBX	Communication protocol unique to u-blox products

*Table 1 dictionary*

# 1 Introduction

The company S[&]T<sup>[13]</sup> does a lot of research in the area of GNSS reliability. There currently is the need for a device that can detect and map GNSS interference sources and monitor the health of GNSS signals. To start the development of this, an intern is hired to start designing an embedded system. The main goal is to be able to measure sources of possible GNSS interference.

At the start of the project the main goal was to create a basic prototype using off the shelf development tools. Another goal is to create a PCB design for this system. Creating a PCB prototype falls outside of the project due to current component shortages.

This document will explain the basic concept of the embedded system, the software design, the electrical design and the PCB design. And an explanation of the concepts, problems, solutions and further developments of the system.

The internship run from February 9<sup>th</sup> to June 3<sup>rd</sup>, 2022.

## 2 Organization

This project has three parties involved. The intern represented by Thomas Joesoef Djamil. The company S[&]T, and the university, The Hague University of Applied Sciences.

Within S[&]T there is a team that handles GNSS projects. This is the sensing and control group. This project is part of this group. The group consists of 8 members give or take. The number of members changes often as interns enter and leave the group. It is a multinational group of people with various types of engineering degrees. The supervisor of the group is Stefan van der Linden.

Stefan will act as a mentor on behalf of the company for the intern. From the university Ad van den Bergh is assigned as mentor.

The intern is hired for 5 months to work on the project. The project is managed by the intern himself and works independently. For technical assistance the intern can get help from the sensing and control group team. For assistance related to the process van den Bergh or van der Linden can be contacted.

During the project the intern can work in the office or from home. At the office the intern has access to resources. This includes a desk spot to work on. Office desks are not assigned but due to human nature most people end up working at the same desk anyway.

Furthermore, there is access to the basic equipment necessary for electronics developments. Such as basic electronic parts, cables and wires, voltage generators and oscillators. Electric measurement tools and an oscilloscope and spectrometer.

Lastly there is access to online file storage from S[&]T to save the work on. A GitLab account is used to maintain the software via Git<sup>[14]</sup>.



### 3 Order requirements

The main required goal of the embedded system is to be able to detect GNSS interference sources. This embedded system will during the project be a prototype made from off the shelf components. This means using components that can be used to create a system without the need of third-party fabrication. Such as PCB manufacturing or other fab facilities. Components that can be either connected with jumper wires or that can be soldered by a human need to be used.

The embedded system needs to be able to do the following. It needs to be able to collect data from a GNSS receiver, most importantly the GNSS fix location and the carrier to noise ratio. Furthermore, the system is required to measure the power level of the GNSS carrier frequency using a custom designed circuit.

The system needs to then be able to send collected data to a host that can then process this data. This would be a PC, or a server connected to the embedded system.

Besides prototyping the system with developer components. A PCB design of the system will be designed.

The final deliverables for the project are: a system design, an electrical design, a PCB design, Embedded system software, documentation and a developer prototype.

#### 3.1 Project borders

The project will have multiple borders to section of what the projects goals exactly are.

This project consists of:

- Creating an embedded system using off the shelf components that can be used without the need of designing custom components. Such as a developer board.
- Writing software
- Designing an electrical circuit
- Designing a PCB
- Creating documentation for the system.

This project does not contain

- Ordering and prototype the PCB.

## 4 Mobile GNSS Interference Detection

### 4.1 System goal

The goal of the project is to create a mobile embedded device that is capable of detecting GNSS interference. The main purpose is to be able to map sources of GNSS interference. The main type of interference that needs to be detected is jamming, however, the system is also capable of detecting spoofing using build in functionality of the ZED-F9P<sup>[3]</sup>. This is the GNSS receiver that is used in the project.

To jam a GNSS signal a RF signal is emitted by the jammer in the same frequency as the carrier frequency of the GNSS signal. This jamming signal is made by a high frequency oscillator that is connected to an antenna, sending these signals into the ether. And because of the weak power level of GNSS signals they are very easy to jam<sup>[15]</sup>.

GNSS signal spoofing is when a false GNSS signal is created which causes GNSS receivers to calculate the wrong position. This is more difficult to detect than jamming. Spoofing is usually detected by getting a genuine position fix first. If the position of a specific satellite suddenly changes unrealistically then it is almost certainly being spoofed. But it is also possible that a satellite gives a wrong location due to signal reflection.

For GNSS jamming it is also possible for the cause to be unintentional, such as from random electromagnetic interference from devices, whereas spoofing is always intentional (not counting signal reflection).

### 4.2 System concept

The system would get the name Mobile GNSS Interference Detection or MGID in short. This system would consist of a microcontroller, a GNSS receiver and a GNSS jamming detection circuit. This system is in connection with a PC or server to transfer its data.

The concept of this system is that it is small and easily transported. So that it can be used outdoors in field trials. The system should be relatively low cost and easy to understand and use.

The system is designed in such a way that it basically can also act as a developer board with build in GNSS availability. This will make the system not only useful for the task that it was designed for, detecting and monitoring GNSS signals and interference. But could also be used for future projects of S&T, related to mobile GNSS solutions.

### 4.3 Appendices

This document includes a couple of appendices that show the design of the system.

- A1 shows the electrical design
- A2 shows the PCB design
- A3 is the main function of the software
- A4 shows the most important C++ class of the system
- A5 is the processing of NMEA messages from the ZED-F9P receiver
- A6 are the FreeRTOS tasks

The software appendices are not meant to give a full overview. But they may provide better understanding and preview on how the software works. The three code snippets that were chosen to show the most important functionality and give a basic understanding of the rest of the code.

## 5 MGID architecture

MGID consists of three distinct components. A STM32 microcontroller<sup>[2]</sup>, a ZED-F9P GNSS receiver and a GNSS jamming detection circuit. These three components process the incoming signals coming from the antenna. The host is where the processed data is sent. It can be thought of as an output. But it is of course a two-way communication. The abstract base design is shown in Figure 1.

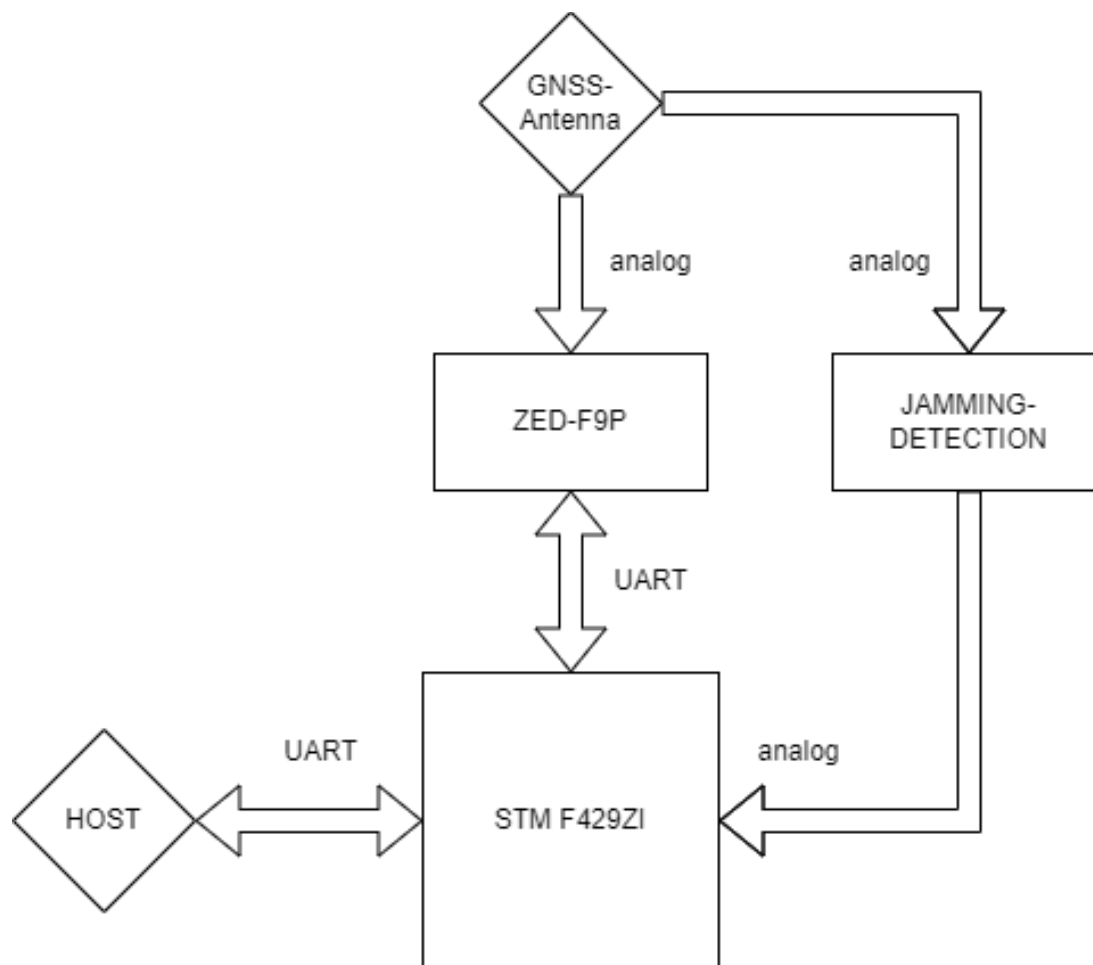


Figure 1 Base design MGID

### 5.1 ST Microcontroller

The MGID system is based around the STM F429ZI microcontroller. This microcontroller is chosen because of its fast performance, many possibilities of serial communication and the relatively large flash and RAM sizes. This makes it the perfect fit for the purposes of the system. There was the desire to get a dual core version of the ST microcontroller, but none were in stock so the F429ZI was the second-best choice.

### 5.2 ZED-F9P GNSS receiver

In order to get a GNSS position a GNSS receiver is necessary. The receiver that was chosen is the ZED-F9P receiver. For its relatively cheap price and available. The ZED-F9P has many different functions that can be used to map GNSS interference. For example, a spoofing detection is built into the ZED-F9P. A basic jamming detection is also available. The receiver

processes the analog data and can that be retrieved via an UART connection by the ST microcontroller. Also, the ZED-F9P has an easily accessible developer board in the form of the C099<sup>[16]</sup>.

### 5.3 The host

The host represents the entity that is communicating with the MGID. This is by default expected to be a PC connected with a USB cable (The protocol from the STM32 is UART, but a UART to USB is used). The host can collect data from the microcontroller. This is the data that is collected from the ZED-F9P receiver and from the analog jamming detection circuit.

### 5.4 Jamming detection circuit

This is an analog semiconductor circuit that is capable of detecting GNSS RF signals. If the incoming power is above the GNSS noise floor than it can be safely assumed that a GNSS jamming signal is nearby. With GNSS the signals are weaker than the ambient noise. So it should not be possible for a GNSS signal to be received that is stronger than the noise floor unless it is a very high grade fixed antenna.

### 5.5 GNSS antenna

The system needs to have a GNSS antenna to be able to receive GNSS signals for processing. The signals for the antenna are split and go to both the ZED-F9P receiver and the jamming detection circuit. By default, it is expected that an active antenna is used. A passive antenna with a high enough output can also work. The MGID should not rely on any specific type of antenna for maximum flexibility.

## 6 Prototyping

For the prototyping of the MGID system a STM32 Nucleo developer board is used. The board that is used for this project is equipped with a F429ZI microcontroller.

For the testing the GNSS connectivity a C099 test board is used. This is the test board for a ZED-F9P GNSS receiver. The test board comes with an active GNSS antenna for the L1 band. This test board also has a WIFI module build in. But since this module is not used, it is deactivated to save power and lower interference.

The F429ZI is connected to the PC. The ZED-F9P is then connected with jumping wires to the F429ZI. The four wires that connect them are a 5 Volt wire for power, a GND wire, and the other two wires are used for the UART.

The ZED-F9P has an active antenna connected. This active antenna is capable high precision geolocation and has a signal gain of roughly 20 dB. Besides the active antenna there is also a GNSS antenna on the roof of the S[&]T building. The roof antenna can get a gain of 40 dB.

For prototyping there also is a prototype board made with an RF input and RF output. This board was used to test possible solutions to the problem of detecting GNSS jamming signals.

The prototype setup is shown in Figure 2.

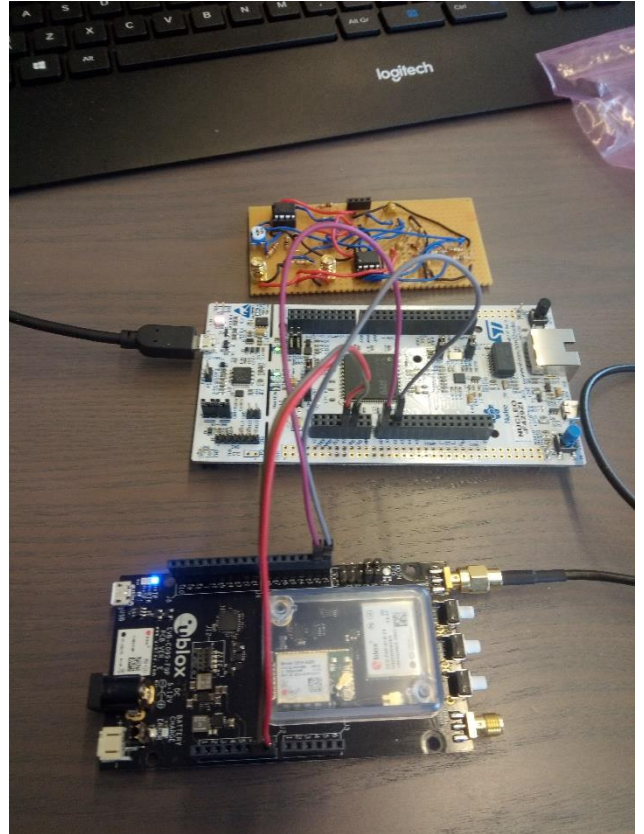


Figure 2 Developer setup

## 7 Software architecture

The STM32 microcontroller is an ARM based system that primarily runs on C but is also capable of running C++. For this project the decision was made to use C++ over C. The reason for this decision is because it became clear very quickly that this system was going to use many different elements. These elements could be easily subdivided into classes.

From the very start the design, of the software, started with the idea of having a central controller that can control multiple components. Making it easy to add functionality to the system. Every function of the MGID is therefore represented by a component. The graph below shows the general architecture and dependencies of all classes<sup>1</sup>. (Figure 3)

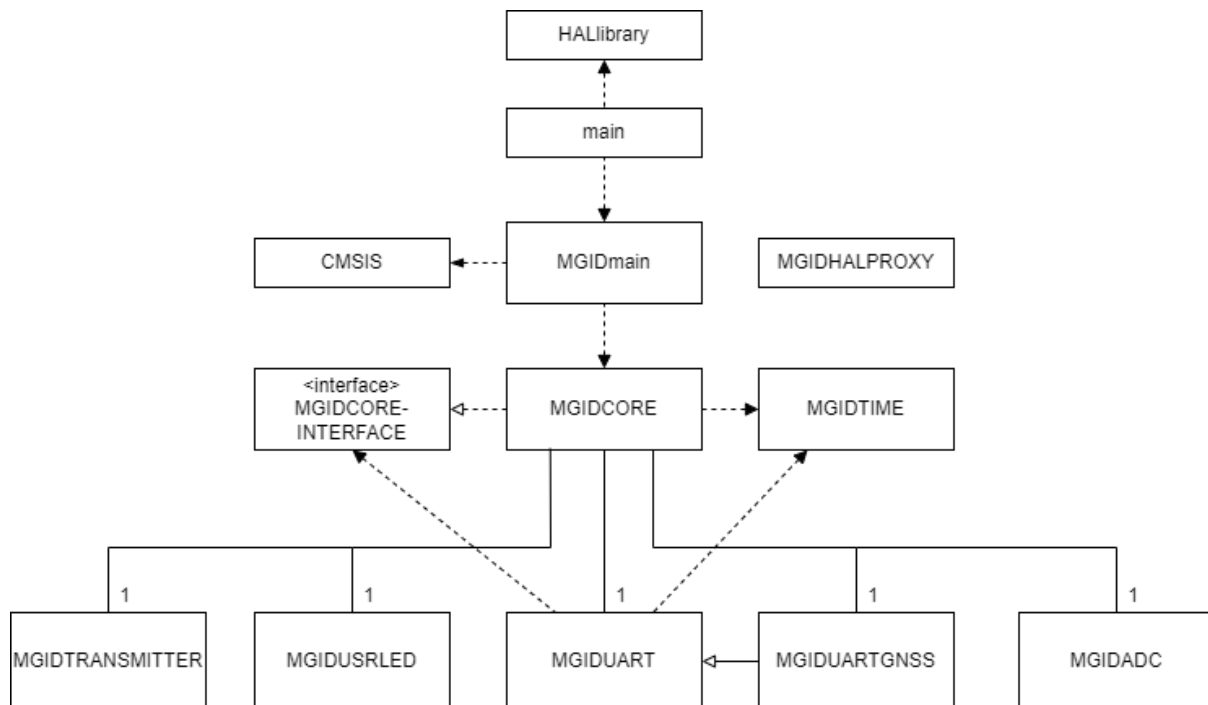


Figure 3 Software architecture

### 7.1 Core principles

It is important to understand how programming for the STM32 microcontroller works. Assuming using the standard STM IDE<sup>[9]</sup>. Whenever a project is made the user can choose in a menu which features, they want to use. When changes are made the system then generates a new main and a file which handles the interrupts. While the system does provide “safe” workspaces where user code can be placed inside of these files without being at risk of being overwritten during code generation. It is not ideal.

For a large project like this it is decided that it is safer to use the concept of a protected main shown in appendix A3. Instead of writing the main code in the standard main an additional main file is written, **MGIDmain**. The generated main is only used to initialize STM components. Such as serial communication and GPIO ports. Once all the initialization is done. In the main a single call to the **MGIDmain** can be made with references to necessary STM32 components such as the UART handlers. From this point the true main is practically exited and the **MGIDmain** takes over.

<sup>1</sup> Connections to **MGIDHALPROXY** are not shown for visibility reasons in Figure 3. All MGID classes are dependent on it.

An additional benefit about this concept of using a protected main like this, is that it makes it very easy to switch ST microcontrollers. It also makes it possible to use the software as a library since it is not attached to anything.

All of the code that is written for the MGID system is segregated into a separate folder. This does not have any functional reason other than making file management easier. (Figure 4)

## 7.2 Basic readability

For readability of the code some basic rules are laid out. All classes related to the MGID system will start with the prefix MGID.

## 7.3 Main initialization

The MGIDmain is exclusively used to initialize objects and variables. This main will initialize uint8\_t buffers that are to be used by the UART and it will initialize all the classes shown in the graph. MGIDmain is shown partially in appendix A3.

Once all the necessary variables and objects are initialized and loaded it will start FreeRTOS. The MGIDmain will then “end” and FreeRTOS takes over thread management.

## 7.4 FreeRTOS

Because the MGID system uses many different time critical functionality it was chosen to use FreeRTOS. Important to note is that in STM32, FreeRTOS is slightly different and is called CMSIS\_RTOS<sup>[12]</sup>. For the system two different threads are used. A thread that handles the measurements and continuous communication with a connected host. And a thread that handles the LED blinking.

The system originally had more threads such as a split between the communication with the host and communication with the ZED-F9P receiver. These threads ended up being combined and so the number of threads ended up being reduced in favor of using interrupt routines. Which is explained in the next chapter.

## 7.5 Interrupts

Because MGID is a time critical system it makes use of interrupts whenever possible instead of using sleep routines that block the entire processor. The only sleep routines used are the ones from CMSIS\_OS that can jump threads when a sleep function is called. These interrupts are for the UART and ADC. By default, STM IDE generates interrupts handlers in its own file. These interrupts handlers have to be deleted from this file and then copied into the MGIDmain. This makes it much easier to keep the code clean, because again all the code should be within the MGID software package. It is important to remember that whenever code is generated by the STM IDE it remakes these handlers even if they already exist elsewhere. They have to then be deleted again otherwise the code will use the wrong handler.

STM IDE will compile when two functions exist with the same name and not give a warning. If a developer doesn't delete the generated function, it will not run the code written in MGIDmain. Any future developer needs to keep this in mind.

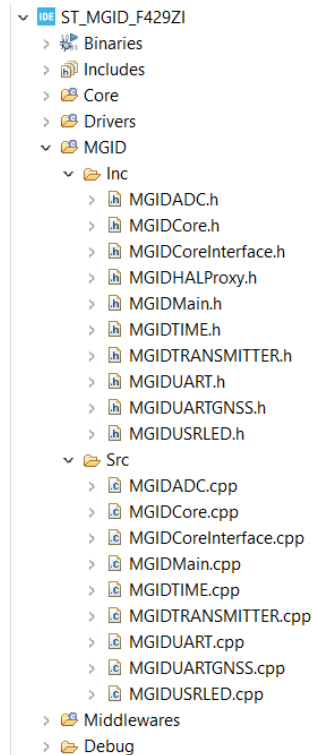


Figure 4 MGID file structure



## 7.6 MGID classes

### 7.6.1 MGIDCORE

The MGIDCORE class is the central control system of the software. A single object is made by the main and controls the system. This core class object has multiple attached components that are used for the functionality of the system. The MGIDCORE is where the functionality of the FreeRTOS tasks are being executed. It is basically the central nervous system and controls everything and is the only class that has full visibility of the entire system (with the exception of the main of course). The class is shown in appendix A4.

This class is fully dependent on its attached components in a software architecture sense. While this is not ideal, and it would be better to use interfaces for the components. There are two reasons why it is better to use a more basic approach to coding:

- The first is that it is not expected for the system to get much more complex than it is right now. The system will not be big enough to warrant the full use of interfaces. It would make the system unnecessarily complex.
- Not every employee at S[&]T who works on the STM microcontrollers is skilled with C++. Widespread use of interfaces could be difficult to work with for future developers who work on the project.

The only interfaces that are used are interfaces to prevent dependencies from the components towards the MGIDCORE class. Having two-way dependencies like this is bad practice and the benefits of interfaces here are big enough to use them. Interface can be seen in Figure 3.

### 7.6.2 MGIDTIME

The MGIDTIME class is designed to keep track of time. It is not a component for MGIDCORE, but it is referenced. It contains functions that can translate time and date data from the NMEA<sup>[5]</sup> messages coming from the ZED-F9P receiver. It also has a time struct that can keep track of the time.

```
struct custom_time_t{
public:
    void SetTimeFromString(string* fromString);
    void SetTimeFromTimeStringNoDays(string* fromString);
    string TimeToString();
    void TickTime();
    void SetYear(uint16_t newYear) { tYear = newYear; };
    void SetMonth(uint8_t newMonth) { tMonth = newMonth; };
    void SetDay(uint8_t newDay) { tDay = newDay; };
    void GetDate(uint16_t* ptrYear, uint8_t* ptrMonth, uint8_t* ptrDay)
    { *ptrYear = tYear; *ptrMonth = tMonth; *ptrDay = tDay; }

private:
    uint16_t tYear = 1970;
    uint8_t tMonth = 1;
    uint8_t tDay = 1;
    uint8_t tHour = 0;
    uint8_t tMinute = 0;
    uint16_t tMilisecond = 0;
};
```

Figure 5 custom time struct

MGID time has a struct called custom\_time\_t (Figure 5). This holds time information in such a way that it can easily convert time from the NMEA messages into this custom struct. This struct has functions attached to it that make updating easy and can easily be bound to the tick function of the operating system. Time gets automatically updated as the system clock keeps ticking.

### 7.6.3 MGIDUART

MGIDUART class has all functionality related to the use of the UART protocol by the system. It handles both outgoing and incoming communications.

### 7.6.4 MGIDUARTGNSS

The MGIDUARTGNSS class is a child class of MGIDUART. This class is modified with overridden functions that instead of talking with the host it will talk with the ZED-F9P over UART. The main function of this class is that it processes the NMEA messages coming from the ZED-F9P receiver, shown in appendix 5A. It takes all the text based messages and converts them to integers for compact storage.



Besides NMEA messages this UART class also handles U-blox's proprietary communication protocol UBX<sup>[4]</sup>. UBX is used both for inbound and outbound communications. U-Blox uses UBX to transmit information that are not send with NMEA. The most important one is the message that contains the CNO and other signal health variables.

The UBX messages that are send to the ZED-F9P are meant for configuration purposes.

### 7.6.5 MGIDTRANSMITTER

This class is not used. It was originally intended to handle the functionality of GSM or other type of mobile data transceiver. However, this ended up not being a requirement of the system. It is still part of the code, but it has no functionality.

### 7.6.6 MGIDADC

For the analog part of the system an ADC measurement is required. The MGIDADC is the part of the system that handles that functionality. It takes the input of the system and converts it to millivolts.

The ADC takes its input from the GNSS JAMMING circuit. By design the higher the GNSS power level is the higher the DC input on the ADC is. The ADC can be calibrated on command show in chapter 10 Communication interface. During calibrating the systems checks what the nominal GNSS power level is. Once calibrating is stopped the average value is saved. If a GNSS power level is detected that is a certain percentage above the threshold (can be configured by the user). Figure 6.

Calibrating can only be done if it is known that there is no GNSS jamming signal nearby. Because otherwise it would save the amplitude of that signal and the jamming detection would only trigger is there would be an even stronger jamming signal nearby.

The ADC is checked in a constant loop. At the start of the program an ADC measurement is requested. When the ADC is finished an interrupt is fired. During the interrupt routine a new measurement will be called. Resulting in a very fast constant measuring of the ADC.

### 7.6.7 MGIDHALPROXY

MGIDHALPROXY consists of only a header file. This file originally only contained a reference to the HAL library. The reason for this proxy was to make it easy to switch microcontrollers. Every ST Microcontroller has a different library. So, when switching microcontroller all the developer must do is change the library in this file and all the components and classes will have the correct dependencies.

This file is now also being used to store configuration variables using define. (Figure 6)

```
//config vars should explain themselves
#define MGID_CFG_LEDLINKSPEED 1000
#define MGID_CFG_CMDCHECKSPEED 100//no longer
#define MGID_CFG_UARTPINGSPEED 100000 //uart :

#define RX_USER_BUF_SIZE 128
#define RX_GNSS_BUF_SIZE 4096 //if this is no

#define ADC_INTER_THRESHOLD 3000 // this is in
#define ADC_INTER_PERC 20 //how much percent

//Replace this include when using a different
#include "stm32f4xx_hal.h"
#include "stm32f4xx_hal_adc.h"
#include "stm32f4xx_it.h"

#endif /* INC_MGIDHALPROXY_H_ */
```

Figure 6 Example of configuration

### 7.6.8MGIDUSRLED

The system uses LED lights to communicate with the developer without the need for a serial communication. It is a quick way to tell if the system is working correctly. The MGIDUSRLED is the class that handles the control of the LED lights. If the LED light stops blinking it means the system has crashed.

If a successful command is sent to the MGID the user LED lights will blink. Making it clear that the command was received and accepted.

## 7.7 Direct memory access

The fact the system is going to run at least two UART connections with constant connection. It became very clear that using only software interrupts was not going to work. There was too much interference between the UART controllers. Also, FreeRTOS does not function well with constantly firing UART interrupts for every character that is registered by the UART controller.

Therefore, it was decided to use DMA for UART controllers. Using DMA, no individual interrupts are fired when receiving characters over UAR. When the line goes idle (so UART has finished) a single interrupt is given by the controller. This then allows the software to process the data that is in the DMA buffer. The single DMA interrupt is handled as quickly as possible and then exits so the FreeRTOS can take over again with thread management. This makes it so an entire message only requires a single interrupt, instead of firing an interrupt every character that comes in.

## 7.8 Connection with host

The user can communicate with the ST microcontroller over UART. In order to be able for this communication to be useful a proprietary protocol is used. The user can send command through character strings. The character strings always start with MGID as an identifier. Hyphens are then used to separate words. For example, if you want to ping the MGID to see if the connection works, MGID-PING can be sent. The MGID will then send a message back.

Messages you get back from the MGID are sent in a way that make it easy to log. The message structure of the MGID is as follows:

`MGID|[TICKS]||[MGIDCOMPONENT]|:[MESSAGE]`

An example of a message from the MGID is: "MGID|2823443|CORE|:ping". The message ends with a "\n\r" and always starts with MGID for synchronization purposes.

It is important to note that messages send to the MGID need to be sent as one continues UART message. Otherwise the MGID will try and process a message after every single character. Most serial terminals for PC's send individual characters when typing in characters. It is important to make sure that there are no pauses between the characters.

The communication is shown in chapter 10 Communication interface

## 7.9 Connection with ZED-F9P

For the communication with the ZED-F9P two different protocols are used. The UBX protocol that is proprietary to U-Blox IC's and the NMEA protocol for GNSS data.

The UBX protocol sends raw data over the UART in the form of bytes. So not as characters. Because it is actual bytes. "/n", "/r" and "/0" cannot be relied on for synchronization. Byte 4 and 5 of the UBX message contain the total size of the message and starts with two synchronization bits at the start of the message. The UBX protocol has no characters and is therefore not human readable.

The NMEA protocol is a character-based protocol. Which means that it sends text that can easily be read and understood by humans.

## 7.10 Booting the system

The system will be booted with the expectation that the ZED-F9P is using default operating parameters. It is important to know that by default the ZED-F9P is not correctly setup for the STM32 to communicate with it.

When the system is booted, and the program starts, the first thing that happens is the STM32 sending the ZED-F9P a message to increase the baudrate to the maximum value. This is done to increase the amount of time the system has to process the message. All data from the ZED-F9P is send every second. The STM32 needs to be done with processing the data before the next round comes in. Otherwise, the system would run into problems.

After the ZED-F9P has its baud rate updated, STM32 also updates its baudrate to the fastest possible. After this the system will configure the ZED-F9P (Figure 7) so that it will send UBX messages over UART that contain information over the signal health. This includes CNO, the signal gain. And if the ZED-F9P thinks there is a spoofing or jamming signal.

Once the ZED-F9P is configured it the system can start processing the incoming data. The serial interface with the STM32 can be checked to see if the ZED-F9P is correctly configured. When a successful UBX message is send to the ZED-F9P and acknowledgement message is sent back. With the way the system works. Three acknowledgements should be received. From this point forward it will get in its operating loop until the system is shutoff. The baud rate change will not receive an acknowledgment since the change in baudrate prevents the message coming through.

```
void MGID_UART_GNSS::InitGNSSModule(){  
    this->TransmitGNSSMSG(MGID_COMMAND_TO_GNSS::GNSS_BAUD);  
    this->UpdateBaudRate(460800);  
    osDelay(2000);  
    this->TransmitGNSSMSG(MGID_COMMAND_TO_GNSS::GNSS_ENABLE_DATE);  
    osDelay(1000);  
    this->TransmitGNSSMSG(MGID_COMMAND_TO_GNSS::GNSS_ENABLE_INTER_MON);  
    osDelay(1000);  
    this->TransmitGNSSMSG(MGID_COMMAND_TO_GNSS::GNSS_ENABLE_RAW);  
}
```

*Figure 7 Function that is used during boot to configure ZED-F9P*

## 8 MGID electrical design

Before a PCB design can be made a clear electrical design is designed. The abstract of the electrical design is show in Figure 8. The connection between the power supply and other components is not shown.

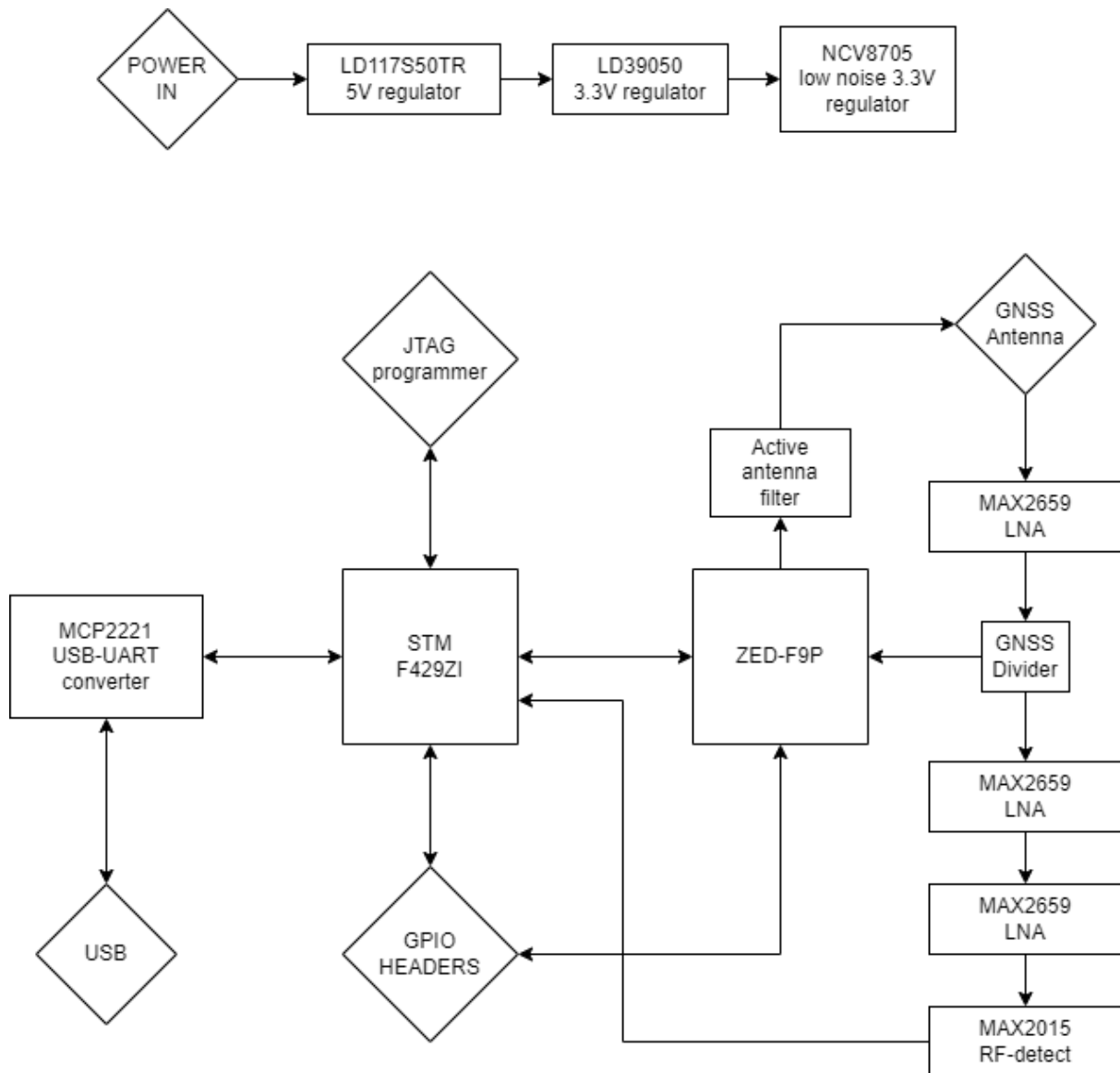


Figure 8 Simplified Electrical design

### 8.1 Power supply

The POWER IN node shows from where the system is getting power. This power can come from three different sources. From the USB power, a barrel jack or through a battery connector. The voltage source can be selected via a jumper header. This incoming power will first go to a 5 volt linear regulator. From here the voltage is lowered again to 3.3 volt. The ZED-F9P requires a low noise power supply. The NCV8705 low noise regulator is delivering power to the ZED-F9P.

## 8.2 MCP2221

Because the ST microcontroller is programmed with UART in mind it cannot directly be connected to the USB. In order for the MCU to talk with the host a MCP2221 USB to UART converter is necessary.

## 8.3 JTAG Programmer

A JTAG connection is required to program the F429ZI. This connection is connected to a ST-Link programmer<sup>[10]</sup>. Unlike STM32 developer boards the MGID system does not use an onboard chip for programming. As a matter of fact, ST-LINK chips are not even available for consumers to put on their PCB designs. It is expected to use a JTAG connection to program the designed device.

The ST-Link programmer is a device that can be connected with a PC. The ST-Link will then work as a bridge between the STM32 and the PC.

## 8.4 GNSS Jamming Detection

GNSS signals have a very low power level at earth's surface. The average gain of GNSS signals is roughly -127.5 dBm. 0 dBm would mean the power equivalent of 1 milliwatt. So -127.5 dBm is equal to 0,18 femtowatt.

The fragility of GNSS signals makes it very easy to jam these signals. Any GNSS signal jammer will easily exceed this power level. This does make it very easy to detect GNSS jammer signals. If a power spike is detected that is well above the -127.5 dBm GNSS it can be safely assumed that a GNSS jamming signal is active nearby.

For the detection of GNSS signals a MAX2015 high frequency detection chip is used. This integrated circuit is capable of detecting 0.1 GHz to 3 GHz signals. The way the detections works is that it takes the incoming RF signals and converts them to a DC voltage. It can detect RF signals with a power level from -65 dBm to 10 dBm. The output voltage increases linearly based on the logarithmic input. For every dB in input there is a flat increase in voltage.

Because the MAX2015<sup>[6]</sup> has such a wide bandwidth in which it can detect RF signals it could give problems with detecting the wrong RF frequency. This won't, however, be a problem for detecting only the GNSS signals. GNSS antenna's and LNA's<sup>[7]</sup> already filter out all other frequencies. No additional filter circuit would need to be added.

## 8.5 GNSS RF input

The system is designed to be used with an antenna that has a gain of at least 20 dB. The noise floor of GNSS signals is in worst case around the -127 dBm mark. The MAX2015 RF detector can only detect RF signals with a power level of at least -65 dBm.

The RF input needs to be split into two separate RF lines (Figure 9). With RF signals it is not possible to simply spit the lines. The reason for this is that the power level of the RF signal is measured by driving a load. In the case of GNSS this is a 50 ohm load. So if two different GNSS integrated circuits are loaded in parallel it would lower the resistance load to 25 ohm. This would cause the integrated circuits to not work correctly. A signal splitter is used to maintain the proper 50 ohm load.

The downside of this is that this signal splitter works as a voltage divider. It cuts the available power in half. Resulting in a 3 dB loss.

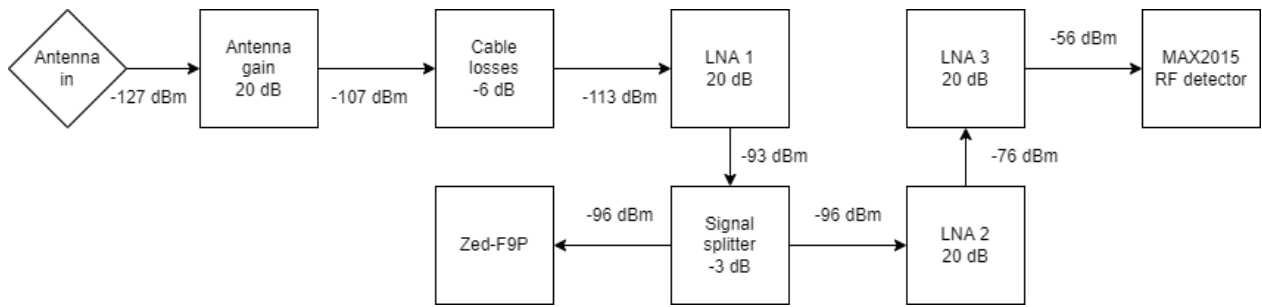


Figure 9 RF signal chain

With an active antenna gain of 20 dB and further cable losses of an estimated -6 dB. The end gain for the ZED-F9P receiver and the RF detector would be -109 dBm. In order to compensate for the loss in gain. A low noise amplifier (LNA) is added to boost the gain level.

It is however not possible to endlessly keep boosting the gain of GNSS signals. Every time a LNA is used in the signal chain. Additional noise is added to the signal. Eventually the noise becomes too great, and it will drown out the usable GNSS signals. This is important for the ZED-F9P since it needs to process GNSS signals. However, the MAX2015 detector only measures the amplitude of the signal. It would not be important to maintain the health of the signals that go to the RF detect circuit.

For the LNA amplification a MAX2659<sup>[7]</sup> is used. This LNA is specifically designed to boost GNSS signals of the L1 band. Perfect fit for the needs of this project.

## 8.6 GPIO headers

While the MGID design does not require any GPIO output to external components or devices. Since many GPIO pins of the ST microcontroller are not used it would be a good idea to make some of them available. The MGID system will most likely be used for future projects related to GNSS interference. By making GPIO ports of the STM available the system is made more future proof. Allowing future projects to use the system as a type of developer board with integrated GNSS functionality. These GPIO headers can be seen near the STM32 in Figure 12.

## 8.7 LED feedback lights

To make it easy for the user to see if the components are working correctly, it is useful to place feedback LED lights. Three different components on the board support LEDs to give feedback to the user. The STM32, the ZED-F9P and the MCP2221. With these LEDs the user can instantly see if the system is powered and based on the blinking if it is working, without the need to make a communications connection with the board.

For the MGID there are three LEDs made available to the user to program. These can be seen on the top left of the board in Figure 10.



## 9 PCB Design

For the MGID a PCB design is designed. This compact PCB of 100 mm by 105 mm (Figure 10).<sup>2</sup> The PCB design is made in Kicad<sup>[8]</sup>.

The PCB is using a standard two copper layer design. The lower copper layer is a copper fill that connects all ground collections. The upper layer contains the traces that connects the components.

The layout of the board can be divided into four quadrants.

The top left quadrant is for the STM32 microcontroller and the GPIO pin headers.

On the top right the ZED-F9P is located with the GNSS RF input placed in the corner.

The lower left is where the power and usb inputs are located. Near these power input the voltage regulators are located. Both the 5 volt and 3.3 volt regulator are placed here. The low noise 3.3 volt regulator is placed near the ZED-F9P on the right side of the board.

The lower right is reserved for the GNSS jamming circuit detection.

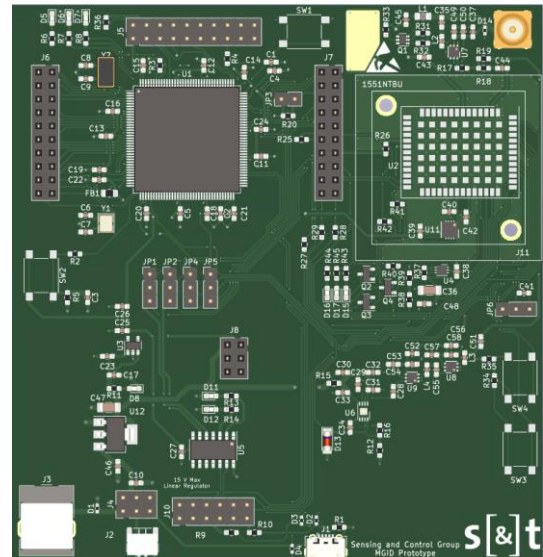


Figure 10 Render of PCB

### 9.1 PCB design principles

The requirements of the system are quite high in terms of components. The STM32 and ZED-F9P together result in a lot of necessary traces to be drawn on the PCB. In order to keep the design manageable traces are laid out parallel to each other as much as is possible. This design principle allows to easily manage many different traces. The main benefit is that collections of traces can easily go under in a single location (Figure 11). If traces were not bundled together like this, it would result in many different trace passes in different areas. Resulting the lower copper layer looking similar to Swiss cheese.

Components are laid out in such a way to minimize the amount of distance traces have to travel and to minimize the amount that traces have to pass over each other.

Because the system works with low power RF signals it is very important to keep the amount of noise generated by the PCB at a minimum. Lowering the amount of interference is done through a couple of means.

First it is important to keep the ground plane as intact as possible. The ground plane should fill the entire board and always have multiple points of contact. If there only is one point of contract that keeps a strip of ground connected to the ground plan. This strip will act like an antenna and create noise on the ground plane.

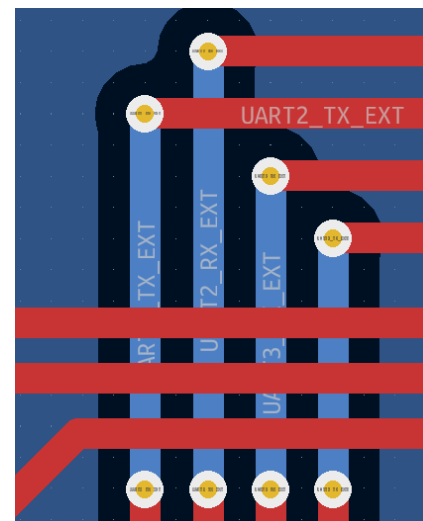


Figure 11 Example of proper trace bus overpass

<sup>2</sup> Not all components on the PCB render have 3D components. Most notably the ZED-F9P on the right side does not have a model.

Another source of possible noise is placing high value resistors close to data lines or other types of switching power sources. Finally, traces should never have 90 degree turns. Only exception when they come from via's.

When traces must pass over each other they should do this as fast as possible. So, they should pass perpendicular to each other whenever possible.

Capacitors should always be placed as close as possible to the pin they are trying to voltage stabilize. For some components multiple capacitors are recommended. In this case the smallest capacitor is placed closest to the component.

Traces that are designed for power delivery are made as wide as possible. Wider traces have less resistance resulting in less noise over the power lines. It also makes it less likely that a trace is burned if too much power flows through it. For this embedded system that is not likely to happen but is still something to keep in mind.

## 9.2 PCB design rules

The design rules of the PCB tell exactly what the minimal dimensions are for traces and via's. And the distances between via's. These minimal design rules are there to make sure that the PCB can be fabricated by a PCB manufacturer. It also limits the cost of a PCB. Most manufacturers have breakpoints on how small a dimension can be. If a design goes over such

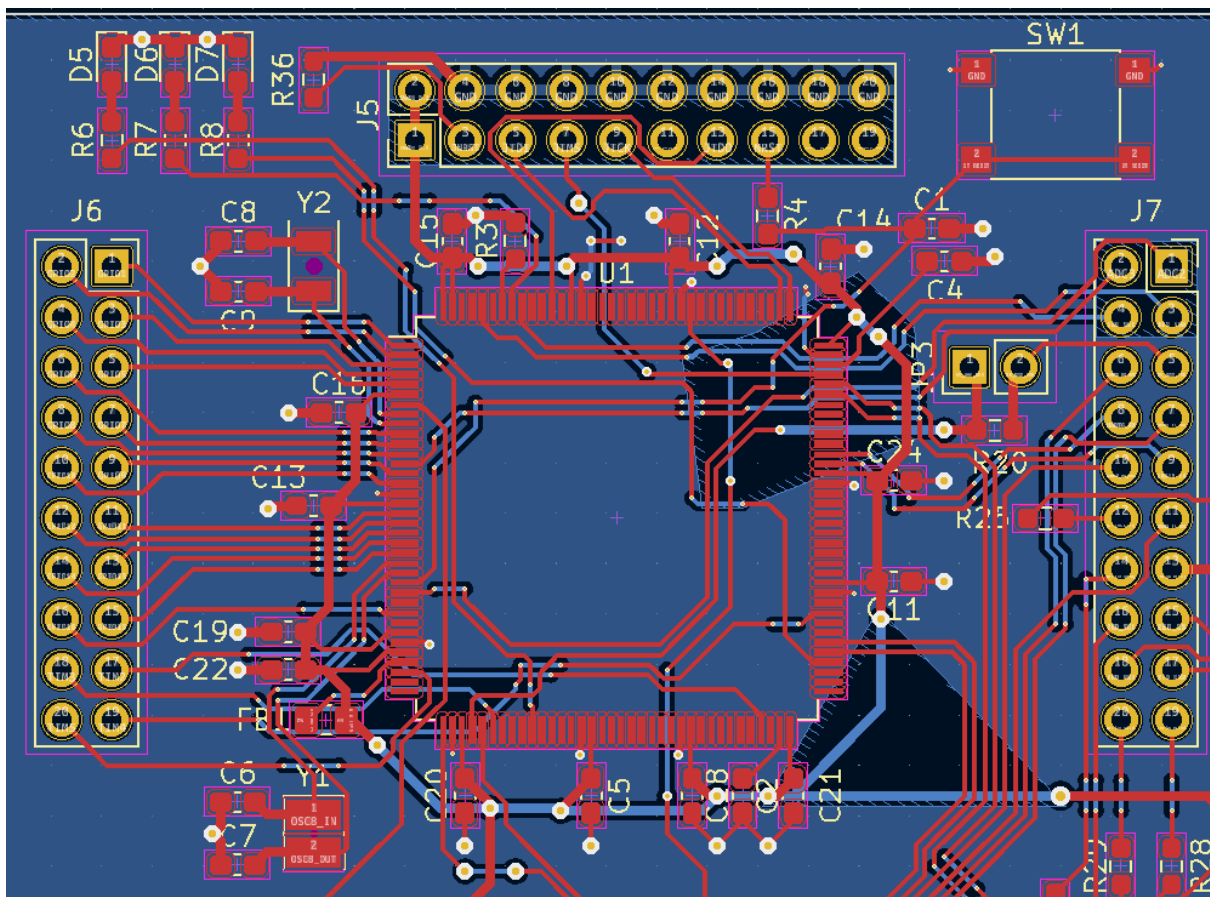


Figure 12 STM32 footprint and pin headers

a breakthrough the price can rise quite a lot. The design rules that were chosen for the design are based on the smallest components chosen. In this case that is the STM32 microcontroller. Which has very little space between the pins (Figure 12).



### 9.3 STM32

The STM32 microcontroller is placed on the top left of the board (Figure 12). The GPIO ports and GPIO pin headers are placed in such a way, so it is easy to create parallel trace busses that can easily be managed in the design.

The STM32 has multiple pins all around the IC's that take the main 3.3 voltage as VCC input. At every one of the pins a 100 nF capacitor is placed to maintain stability on the voltage.

There is a JTAG connector above the STM32. This JTAG is for the ST-Link programmer. The JTAG connector header is placed close to the edge of the board. This makes it easy to place and remove the connector if necessary. When placing the connector, the user can use the edge of the PCB as anchor point. Preventing the user from possibly touching board components and thus preventing damage.

### 9.4 Dual pin headers

On both sides of the STM32 the GPIO pin headers are located. They are placed close to the board so the large number of traces that go between them can be kept as short as possible.

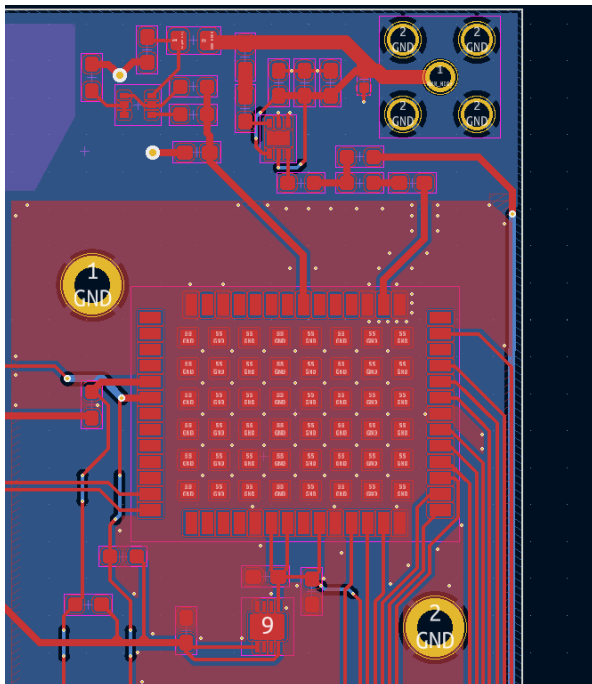


Figure 13 ZED-F9P with RF input

### 9.5 ZED-F9P

The ZED-F9P is positioned on the top right side of the board (Figure 13). The chip is enclosed in a plastic housing. This plastic housing prevents airflow that could negatively affect the temperature balance of the chip<sup>[1]</sup>. This disruption of the temperature balance causes the sensitive chip to decrease in performance. For this reason, there is the plastic housing. This plastic housing can be seen in Figure 2 on the developer board.

The ZED-F9P also has an additional ground fill on the upper copper layer to increase the grounding benefit. There is a low noise voltage regulator near the ZED-F9P, as close as is possible.

The RF input is located close above the ZED-F9P. The input trace is kept as short as possible as to minimize the loss of the signal.

The data traces coming from the right side of the chip is not ideal. Because of this reason the traces need to take a long way round the board to reach the GPIO ports and connected LEDs. But because the RF input trace needs to point to the top corner of the board it was necessary to make this decision.

Left of the RF input there is a spot where the solder mask is removed exposing the ground fill. This open space is there so there is the possibility to clamp something to it improving the ground connection by introducing more material.

## 9.6 Jamming circuit detection

The jamming circuit detection consists of two LNA's going to the input of the MAX2015 RF Detector (Figure 14). The jamming circuit is quite far away from the RF input. This is not ideal especially since the RF line comes close to the data lines. But since data integrity is not important for the jamming circuit it is not really of a concern. And it is unlikely that there would be enough interference that the amplitude of the signal would change.

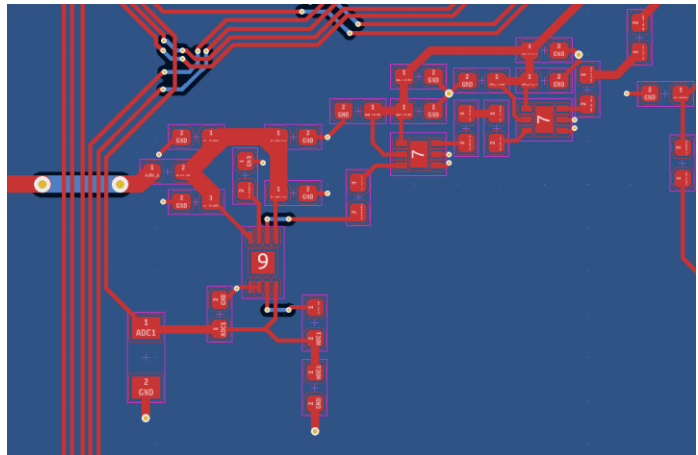


Figure 14 GNSS Jamming detection circuit

The system has a lot of open space available on the board. This is by design to allow easy changes and additions to the board. The jamming circuit detection is the part of the electrical design that is most likely to change in the future.

## 9.7 Choice of components

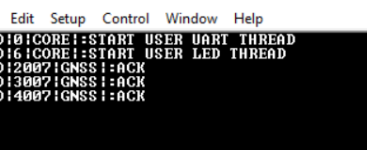
At first for the choice of the components the schematics of the dev-boards (Figure 2) of the STM32 MCU and ZED-F9P were analyzed. The components that are used for these boards have proven to work so they were a good choice as a starting point in the design of the system. Both the STM32 manual<sup>[2]</sup> and the ZED-F9P manual<sup>[1]</sup> have examples of circuit design that are given to show what is best practice.

However, for fabricating the PCB. The fabrication services of EuroCircuits<sup>[11]</sup> would have been used. EuroCircuits has a list of components that are readily available to be used. Using these components is significantly cheaper than having components that need to be ordered on demand.

EuroCircuits has four tiers of availability from most ideal to least ideal. Generic parts, electronic stock, on demand or customer delivered. Generic parts are only basic passive components such as resistors and capacitors. Electronic stock contains many of the most used IC components. On demand has most of the electronic component library but not everything. An extra fee needs to be paid for on demand ordering. Customer delivered is when the customer who made the order has to deliver the components themselves. Which adds a lot of complications. Such as longer delivery time or having to set up the process of delivering components to EuroCircuits.

The electrical and PCB design was slightly revised in order to adjust for the EuroCircuits fabrication process. The voltage regulators were switched out for components that are in their electronic stock. The STM32 microcontroller is part of the electronic stock. Unfortunately, the Maxim Integrated components can only be delivered by the customer. This makes the manufacturing process more expensive. No good alternatives are available.

## 10.1 Interfacing with host



COM4 - Tera Term VT

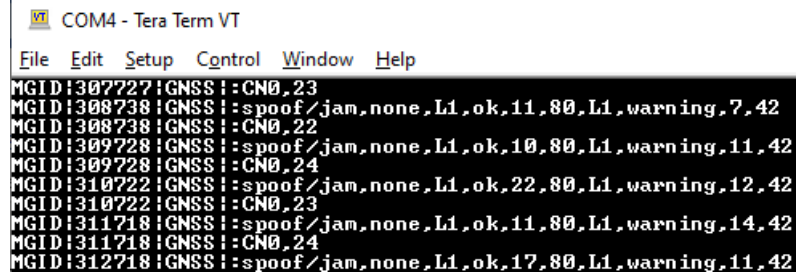
File Edit Setup Control Window Help

```
MGID:0:CORE::START USER UART THREAD
MGID:6:CORE::START USER LED THREAD
MGID:2007:GNSS:=ACK
MGID:3007:GNSS:=ACK
MGID:4007:GNSS:=ACK
```

In Figure 16 it is shown what happens when the user requests GNSS feedback using command “MGID-GNSS-FBK-CMPCT”.

Figure 16 Basic compact feedback from MGID

In Figure 17 it is shown what happens when the user requests GNSS feedback using command “MGID-GNSS-FBK-INTER”. This command only makes the MGID return the CNO data and possible spoofing and jamming results from the ZED-F9P. This is the minimal data transfer option for the MGID if only signal health is important.



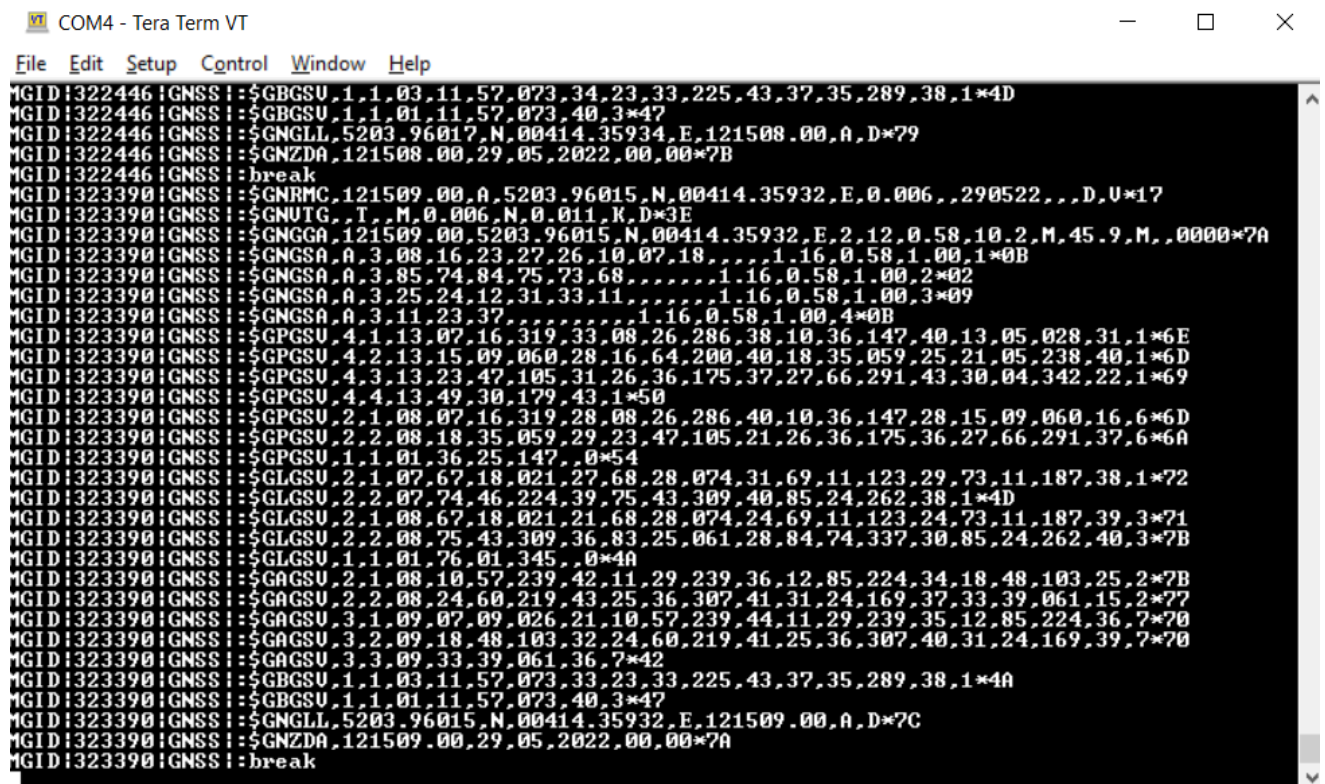
```

COM4 - Tera Term VT
File Edit Setup Control Window Help
MGID:307727!GNSS!:=CNO,23
MGID:308738!GNSS!:=spoof/jam,none,L1,ok,11,80,L1,warning,7,42
MGID:308738!GNSS!:=CNO,22
MGID:309728!GNSS!:=spoof/jam,none,L1,ok,10,80,L1,warning,11,42
MGID:309728!GNSS!:=CNO,24
MGID:310722!GNSS!:=spoof/jam,none,L1,ok,22,80,L1,warning,12,42
MGID:310722!GNSS!:=CNO,23
MGID:311718!GNSS!:=spoof/jam,none,L1,ok,11,80,L1,warning,14,42
MGID:311718!GNSS!:=CNO,24
MGID:312718!GNSS!:=spoof/jam,none,L1,ok,17,80,L1,warning,11,42

```

Figure 17 Compact signal health feedback

In Figure 18 it is shown what happens when the user requests GNSS feedback using command “MGID-GNSS-FBK-ON”. With this command the MGID directly forwards the NMEA messages from the ZED-F9P to the host PC. This is a useful option if the host PC wants to process the NMEA messages themselves.



```

COM4 - Tera Term VT
File Edit Setup Control Window Help
MGID:322446!GNSS!:=GBGSU,1,1,03,11,57,073,34,23,33,225,43,37,35,289,38,1*4D
MGID:322446!GNSS!:=GBGSU,1,1,01,11,57,073,40,3*47
MGID:322446!GNSS!:=GNGLL,5203.96017,N,00414.35934,E,121508.00,A,D*79
MGID:322446!GNSS!:=GNZDA,121508.00,29,05,2022,00,00*7B
MGID:322446!GNSS!:=break
MGID:323390!GNSS!:=GNRMC,121509.00,A,5203.96015,N,00414.35932,E,0.006,,290522,,D,U*17
MGID:323390!GNSS!:=GNUTG,,T,,M,0.006,N,0.011,K,D*3E
MGID:323390!GNSS!:=GNGGA,121509.00,5203.96015,N,00414.35932,E,2,12,0.58,10.2,M,45.9,M,,0000*7A
MGID:323390!GNSS!:=GNGSA,A,3,08,16,23,27,26,10,07,18,,,,,1.16,0.58,1.00,1*0B
MGID:323390!GNSS!:=GNGSA,A,3,85,74,84,75,73,68,,,,,1.16,0.58,1.00,2*02
MGID:323390!GNSS!:=GNGSA,A,3,25,24,12,31,33,11,,,,,1.16,0.58,1.00,3*09
MGID:323390!GNSS!:=GNGSA,A,3,11,23,37,,,,,1.16,0.58,1.00,4*0B
MGID:323390!GNSS!:=GPGSV,4,1,13,07,16,319,33,08,26,286,38,10,36,147,40,13,05,028,31,1*6E
MGID:323390!GNSS!:=GPGSV,4,2,13,15,09,060,28,16,64,200,40,18,35,059,25,21,05,238,40,1*6D
MGID:323390!GNSS!:=GPGSV,4,3,13,23,47,105,31,26,36,175,37,27,66,291,43,30,04,342,22,1*69
MGID:323390!GNSS!:=GPGSV,4,4,13,49,30,179,43,1*50
MGID:323390!GNSS!:=GPGSV,2,1,08,07,16,319,28,08,26,286,40,10,36,147,28,15,09,060,16,6*6D
MGID:323390!GNSS!:=GPGSV,2,2,08,18,35,059,29,23,47,105,21,26,36,175,36,27,66,291,37,6*6A
MGID:323390!GNSS!:=GPGSV,1,1,01,36,25,147,,0*54
MGID:323390!GNSS!:=GLGSU,2,1,07,67,18,021,27,68,28,074,31,69,11,123,29,73,11,187,38,1*72
MGID:323390!GNSS!:=GLGSU,2,2,07,74,46,224,39,75,43,309,40,85,24,262,38,1*4D
MGID:323390!GNSS!:=GLGSU,2,1,08,67,18,021,21,68,28,074,24,69,11,123,24,73,11,187,39,3*71
MGID:323390!GNSS!:=GLGSU,2,2,08,75,43,309,36,83,25,061,28,84,74,337,30,85,24,262,40,3*7B
MGID:323390!GNSS!:=GLGSU,1,1,01,76,01,345,,0*4A
MGID:323390!GNSS!:=GAGSU,2,1,08,10,57,239,42,11,29,239,36,12,85,224,34,18,48,103,25,2*7B
MGID:323390!GNSS!:=GAGSU,2,2,08,24,60,219,43,25,36,307,41,31,24,169,37,33,39,061,15,2*77
MGID:323390!GNSS!:=GAGSU,3,1,09,07,09,026,21,10,57,239,44,11,29,239,35,12,85,224,36,7*70
MGID:323390!GNSS!:=GAGSU,3,2,09,18,48,103,32,24,60,219,41,25,36,307,40,31,24,169,39,7*70
MGID:323390!GNSS!:=GAGSU,3,3,09,33,39,061,36,7*42
MGID:323390!GNSS!:=GBGSU,1,1,03,11,57,073,33,23,33,225,43,37,35,289,38,1*4A
MGID:323390!GNSS!:=GBGSU,1,1,01,11,57,073,40,3*47
MGID:323390!GNSS!:=GNGLL,5203.96015,N,00414.35932,E,121509.00,A,D*7C
MGID:323390!GNSS!:=GNZDA,121509.00,29,05,2022,00,00*7A
MGID:323390!GNSS!:=break

```

Figure 18 Forwarding of the NMEA messages from the ZED-F9P

## 10.2 Command list

### 10.2.1List

Command	Description
MGID-PING	Ping to the microcontroller. Get PONG back. Use to check connectivity
MGID-LED-OFF	Turn off the user feedback leds
MGID-LED-ON	Turn on the user feedback leds
MGID-TIM-UPDATE-[YEAR]-[MONTH]-[DAY]	Update the datetime. Only works when not using the ZED-F9P. Because NMEA overwrites it
MGID-GNSS-FBK-ON	Turns on direct feedback data from the ZED-F9P
MGID-GNSS-FBK-CMPCT	Turn on feedback but in a processed format from MGID
MGID-GNSS-FBK-INTER	Turn on feedback but only show interference data
MGID-ADC-RAW	Get current ADC measurement in raw 12-bit data.
MGID-ADC-MVOLT	Get current ADC measurement in millivolt
MGID-GNSS-POS	Get current langlong GNSS position
MGID-GNSS-SIGNAL	Get current GNSS signal state in dB
MGID-GNSS-HEALTH	Get the health of signals
MGID-GNSS-SATS	Get current connected satellites
MGID-GNSS-TIME	Get the time from the GNSS satellites
MGID-GNSS-SATS	Get current visible satellites
MGID-GNSS-Inter	Get current interference status
MGID-ADC-CALI-ON	Start calibrating the ADC (must be no GNSS interference while calibrating)
MGID-ADC-CALI-OFF	Complete calibrating

Table 2 MGID commands

### 10.2.2Extended explanation

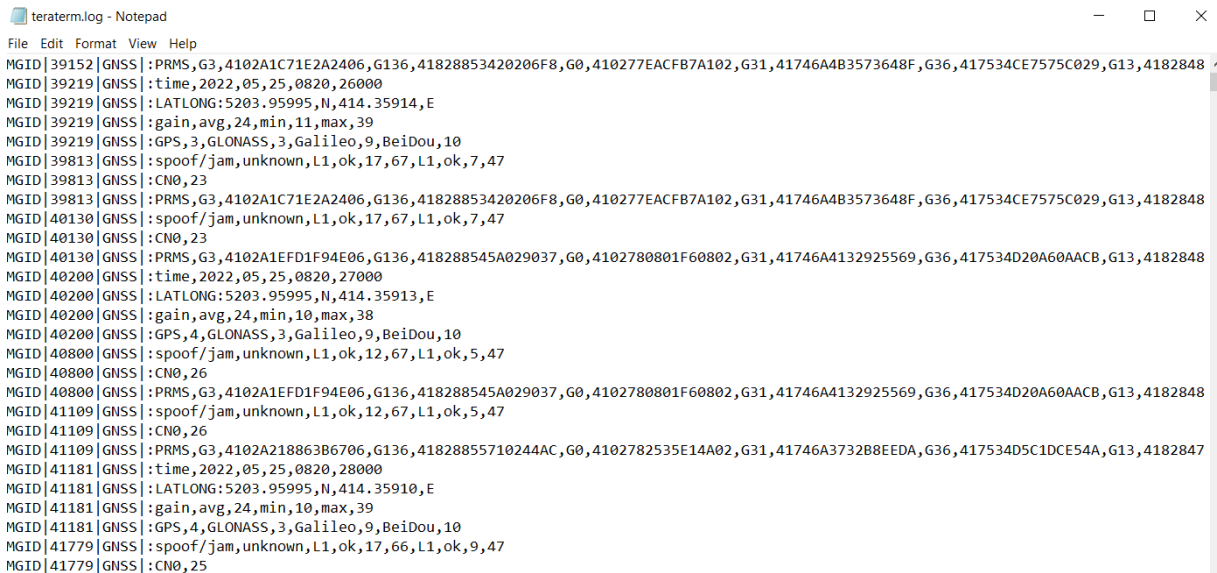
Some of the commands require more explanation on what exactly is happening. The calibration commands are explained in the MGIDADC section of the software architecture.

The MGID-GNSS-HEALTH command will return values related to spoofing and jamming. This return message starts with spoof/jam in the message field. It is then followed by whether there is a suspected spoofing. If there is no spoofing a none is returned. Then an array of all communications band is given with a warning if they are being jammed or not. This is given by the string values of “none, warning, critical”. The difference between warning and critical is that with a warning information can still be accessed but there is a lot of noise on the line. Critical means that the noise level is too great for the signal to be usable.



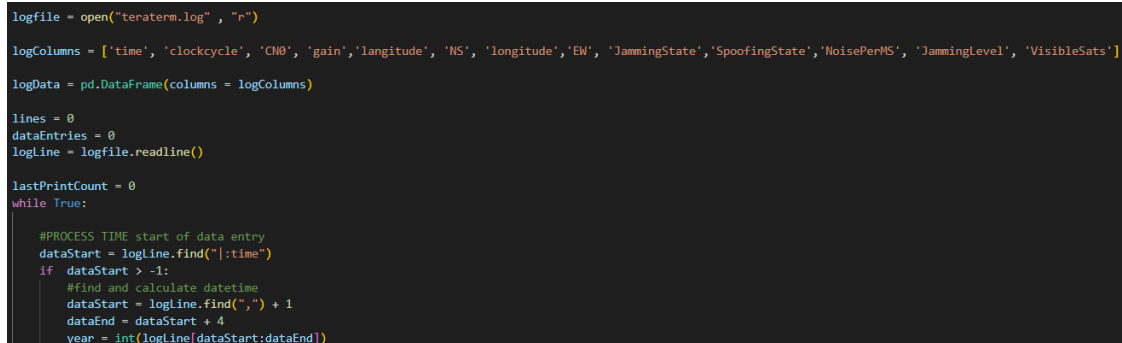
## 11 Processing data with Python scripts

In order to process the logging messages (Figure 19) to useful data. Two python scripts are written. One python script takes the logging messages and converts them to a csv data file (Figure 20 and Figure 21). The other python script can take the made csv file and turns the data into graphs that a human can understand. Chapter 11.1 Python graphs.



```
MGID|39152|GNSS|:PRMS,G3,4102A1C71E2A2406,G136,41828853420206F8,G0,410277EACFB7A102,G31,41746A4B3573648F,G36,417534CE7575C029,G13,4182848
MGID|39219|GNSS|:time,2022,05,25,0820,26000
MGID|39219|GNSS|:LATLONG:5203.95995,N,414.35914,E
MGID|39219|GNSS|:gain,avg,24,min,11,max,39
MGID|39219|GNSS|:GPS,3,GLONASS,3,Galileo,9,BeiDou,10
MGID|39813|GNSS|:spoof/jam,unknown,L1,ok,17,67,L1,ok,7,47
MGID|39813|GNSS|:CN0,23
MGID|39813|GNSS|:PRMS,G3,4102A1C71E2A2406,G136,41828853420206F8,G0,410277EACFB7A102,G31,41746A4B3573648F,G36,417534CE7575C029,G13,4182848
MGID|40130|GNSS|:spoof/jam,unknown,L1,ok,17,67,L1,ok,7,47
MGID|40130|GNSS|:CN0,23
MGID|40130|GNSS|:PRMS,G3,4102A1EFD1F94E06,G136,418288545A029037,G0,4102780801F60802,G31,41746A4132925569,G36,417534D20A60AACB,G13,4182848
MGID|40200|GNSS|:time,2022,05,25,0820,27000
MGID|40200|GNSS|:LATLONG:5203.95995,N,414.35913,E
MGID|40200|GNSS|:gain,avg,24,min,10,max,38
MGID|40200|GNSS|:GPS,4,GLONASS,3,Galileo,9,BeiDou,10
MGID|40800|GNSS|:spoof/jam,unknown,L1,ok,12,67,L1,ok,5,47
MGID|40800|GNSS|:CN0,26
MGID|40800|GNSS|:PRMS,G3,4102A1EFD1F94E06,G136,418288545A029037,G0,4102780801F60802,G31,41746A4132925569,G36,417534D20A60AACB,G13,4182848
MGID|41109|GNSS|:spoof/jam,unknown,L1,ok,12,67,L1,ok,5,47
MGID|41109|GNSS|:CN0,26
MGID|41109|GNSS|:PRMS,G3,4102A218863B6706,G136,41828855710244AC,G0,4102782535E14A02,G31,41746A3732B8EEDA,G36,417534D5C1DCE54A,G13,4182847
MGID|41181|GNSS|:time,2022,05,25,0820,28000
MGID|41181|GNSS|:LATLONG:5203.95995,N,414.35910,E
MGID|41181|GNSS|:gain,avg,24,min,10,max,39
MGID|41181|GNSS|:GPS,4,GLONASS,3,Galileo,9,BeiDou,10
MGID|41779|GNSS|:spoof/jam,unknown,L1,ok,17,66,L1,ok,9,47
MGID|41779|GNSS|:CN0,25
```

Figure 19 example of logged data



```
logfile = open("teraterm.log", "r")

logColumns = ['time', 'clockcycle', 'CN0', 'gain', 'latitude', 'NS', 'longitude', 'EW', 'JammingState', 'SpoofingState', 'NoisePerWS', 'JammingLevel', 'VisibleSats']

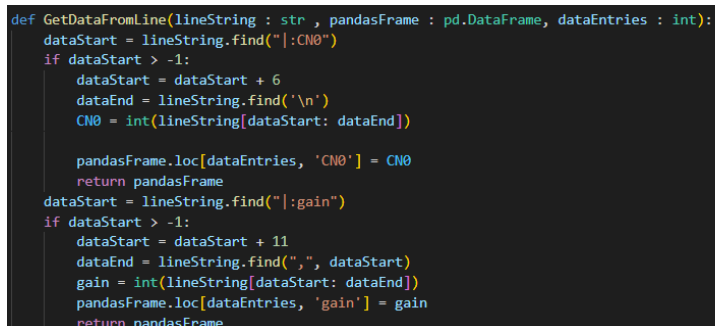
logData = pd.DataFrame(columns = logColumns)

lines = 0
dataEntries = 0
logLine = logfile.readline()

lastPrintCount = 0
while True:

    #PROCESS TIME start of data entry
    dataStart = logLine.find("time")
    if dataStart > -1:
        #find and calculate datetime
        dataStart = logLine.find(",") + 1
        dataEnd = dataStart + 4
        year = int(logLine[dataStart:dataEnd])
```

Figure 20 Starting data tables



```
def GetDataFromLine(lineString : str , pandasFrame : pd.DataFrame, dataEntries : int):
    dataStart = lineString.find("CN0")
    if dataStart > -1:
        dataStart = dataStart + 6
        dataEnd = lineString.find("\n")
        CN0 = int(lineString[dataStart: dataEnd])

        pandasFrame.loc[dataEntries, 'CN0'] = CN0
        return pandasFrame
    dataStart = lineString.find("gain")
    if dataStart > -1:
        dataStart = dataStart + 11
        dataEnd = lineString.find(",", dataStart)
        gain = int(lineString[dataStart: dataEnd])
        pandasFrame.loc[dataEntries, 'gain'] = gain
        return pandasFrame
```

Figure 21 Example of reading data from log

## 11.1 Python graphs

In this chapter multiple graphs are shown that showcase the data that has been collected over the duration of roughly 12 hours. The graphs show the capability of the system.

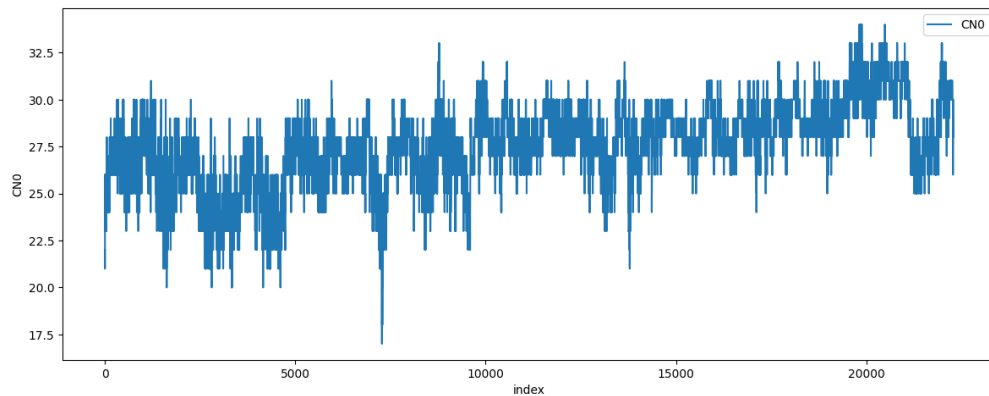


Figure 22 Carrier to noise measurements

Figure 22 shows the Carrier to Noise ratio. The graph shows that there is a slow oscillation at play. This makes sense since as time goes on. Satellites enter and exit the field of view of the antenna. Increasing and decreasing the signal health based on how well each satellites information is being received.

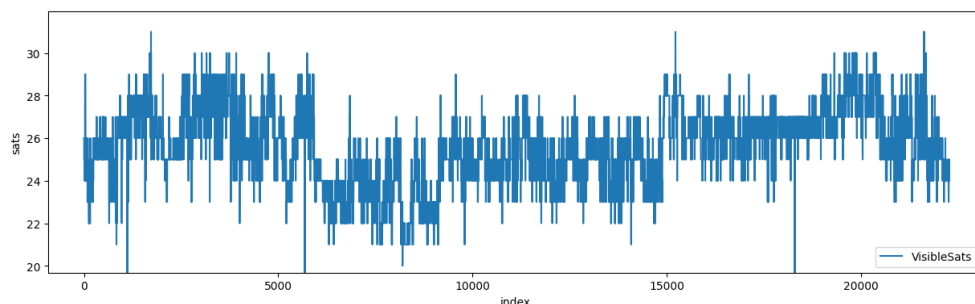


Figure 23 Number of visible satellites

Figure 23 show the total amount of visible satellites by the ZED-F9P receiver. This is the total number and not all of these satellites will be used for position calculation. This will be because the signal of these satellites is too far degraded to make an accurate calculation. But the satellite is still visible to the receiver.

In Figure 24, Figure 25, Figure 26 and Figure 27 the pseudoranges of randomly chosen satellites are graphed (GPS 13, GPS 30, GPS 24 and GPS 16). Multiple things can be derived from these. GPS 30 has a very stable pseudorange calculation with only starting to lose signal when the satellite is getting far away from the receiver.

Both GPS 13 and GPS 24 are suffering from the same problem. The correct distance is shown but is often offset by a fixed amount up to an unrealistic distance. This could be because they both have a weak signal and the receiver struggles and gets the wrong distance. Or because there is some kind of reflection that causes a distance offset.

GPS 16 has no jumps in distance, but clearly the receiver does not get a good signal.

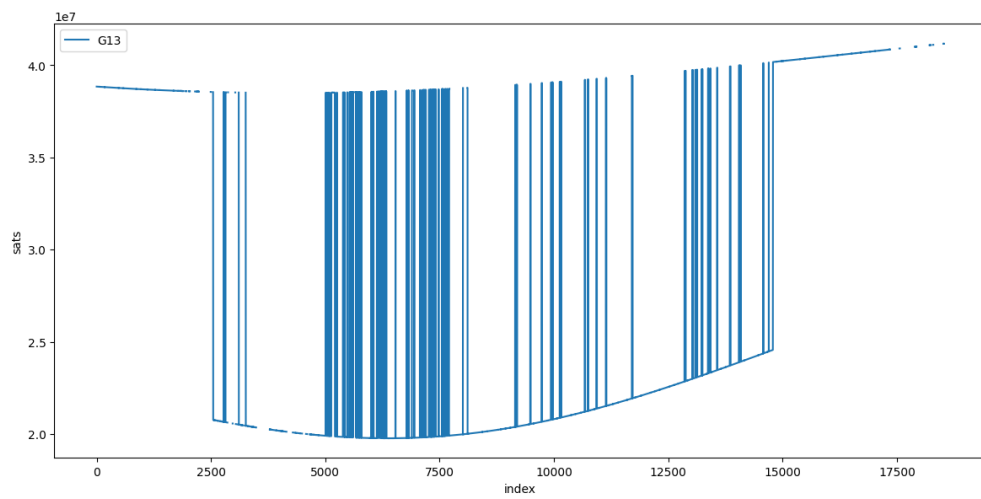


Figure 24 Pseudorange of GPS satellite 13

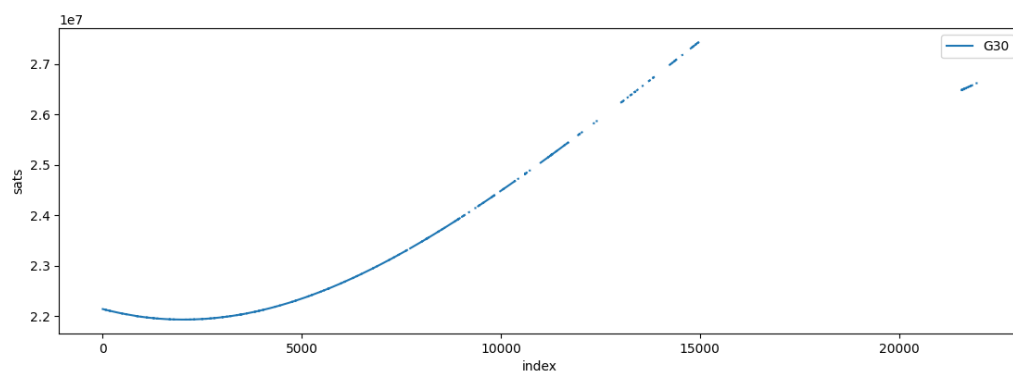


Figure 25 Pseudorange of GPS satellite 30

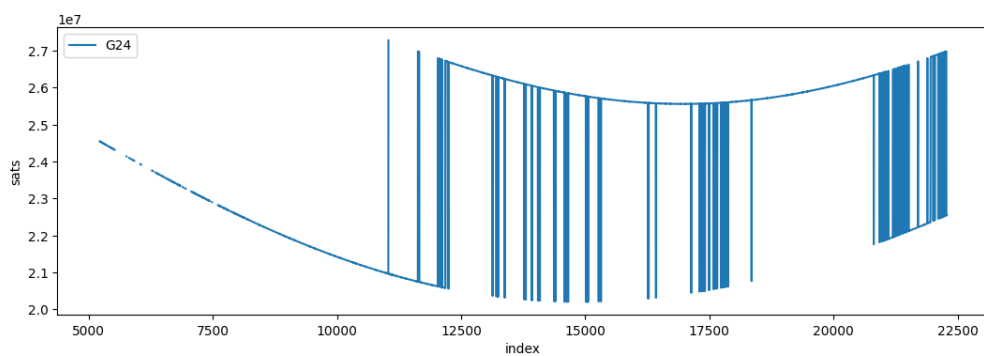


Figure 26 Pseudorange of GPS satellite 24



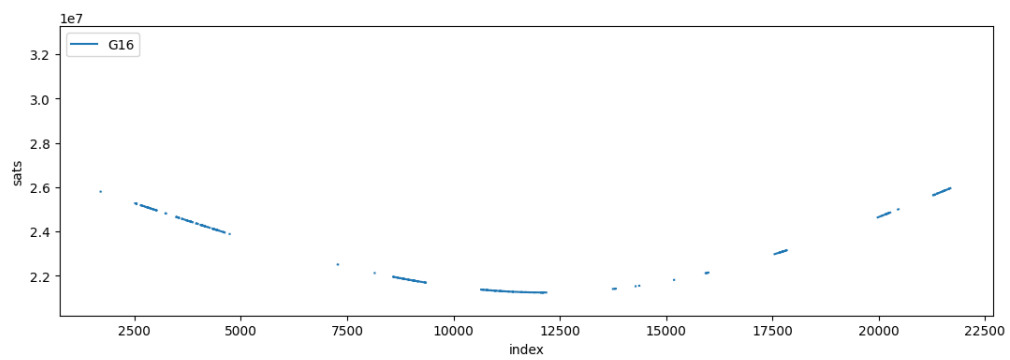


Figure 27 Pseudorange of GPS satellite 16

## 12 Design validation

Checking if the design is correct and valid is very important for this project. The ZED-F9P is an expensive part and so is making a custom PCB with included component placing. There are many things that need to be made sure will function correctly. A single flaw could turn the entire system into a dud. If there is, it would not realistically be possible to do a small fix after the PCB has been delivered. An entire new PCB would have to be ordered, which would obviously cost a lot.

The two ways that the system could have a serious design flaw is if there is a flaw in the electrical design itself or if there is a flaw in the design of the PCB.

Possible bugs and glitches in the software are not really of great concern since these can be fixed after the ordering of the PCB. A problem with the software can be fixed by simply reprogramming the STM32.

### 12.1 Electrical design

The electrical design of the PCB is quite simple. In the end it uses simple linear voltage converters to feed of the shelf integrated chips. The design is also based of the developer board of the STM32. The only danger is that the capacitors are not enough to keep the voltages stable. But this is unlikely and even if this is the case. Replacing capacitors is one of the things that could realistically be fixed after ordering the PCB.

The biggest danger in the design is that the RF circuit does not work. There are two major points of possible failure in the design that need to be checked. The first is if the first LNA in the sequence might add too much noise and causing the ZED-F9P to not be able to make a position fix reliable. While it is calculated that the noise level should remain in acceptable limits it is still something that should be kept in mind. A physical test on this specific question is advised.

Another possible problem is that the RF detector circuit does not function like expected and is not possible to detect GNSS interference sources. Because it was not possible to physically test this circuit it cannot be made certain that it will work. This is the main thing that still needs to be validated and tested by future development.

### 12.2 PCB design

For the PCB design there are three possible ways that there could be a flaw in the design. The first is that a component footprint is wrong making it not possible to place the component on the board.

Another possible flaw is an incorrectly laid trace that makes a wrong connection. This is most likely to happen with the large IC's that have many traces, such as either the STM32 or ZED-F9P.

The final flaw that is possible, is electrical interference because of the way the traces, components, and copper fills are laid out.

### 12.3 Performed design validations

In order to control if the system has no mistakes in the design either the software, electrical or PCB design a system design validation is required. This is to improve the quality of the work and can prevent future errors. Below a table can be found with all of the validation checks that have been done in order to validate the system.

Design type	Validation	Result
Electrical	All electrical connections are correct	Valid
Electrical	Power regulators feed sufficient power to the system	Valid, the power regulator used has the same power output as the STM devboard. Which is already proven to work in practice
Electrical	Too much noise on the RF input	Valid, calculation of the expected noise shows that the noise figure stays within limits
PCB	Pins are correct	Valid
PCB	Footprints are correct	Invalid, some of the footprints on the board did not fit the components. This has been updated accordingly
PCB	No violated design rules	Valid, all design measurements were within the design rules
Software	Software stability	The system has proven to run for at least 12 hours with no problems.

*Table 3 Performed validation checks*

## 13 Future developments

Because of the limitations during the internship, lack of physical testing due to component shortages and the time limit for such a complex system, there are future developments required to make a functional prototype of the system. To make it clear. It was never expected at the start of the internship that a PCB would be ordered. But it would have been nice if it was possible if there was a lot of time left over.

For future developers of this project a wiki was written on the GitLab page of the project. This GitLab wiki contains all of the information necessary to continue this project.

### 13.1 Testing of jamming detection circuit

The most important development that still needs to happen is to be able to physically test the jamming detection circuit to see what the test results are. A test would need to be done with the components from the design and then using a jamming signal to validate if the circuit is capable of detecting jamming signals.

Not only would there need to be determined if it is capable of detecting jamming signal but also measuring at which interference level the circuit can detect them.

### 13.2 Choose housing for the system

Currently the system does not have a case or housing in which it is going to be build. It would be a good idea to have a solid case for it, making it more rugged to be used in the field. The PCB design right now does not have any mounting holes. And this is on purpose because no case or housing has been selected yet. Future developers would have to add mounting holes so that the PCB would fit in their chosen housing/case.

### 13.3 Creating an order to a PCB manufacturer

If a design is finalized that has proven to work in practice. A PCB can be ordered from a PCB manufacturer. But it is important to understand that this manufacturer also needs to install the components on the PCB. The components for this system are not capable of being installed using human soldering techniques.

The most important thing that future developers need to keep in mind is that the ZED-F9P is a very sensitive piece of circuitry. It requires a very specific soldering heat diagram in order to be correctly installed and not be damaged.

### 13.4 Testing and validating the PCB prototype in practice

If a PCB prototype is made, then it needs to be tested in practice using actual GNSS jamming signals. It should be looked at if it is possible somewhere to perform such a test and document the measured results.

### 13.5 Mobile communication possibilities

When the system has been validated, it might be useful to add some kind of mobile communication. Such as GSM or satellite communication. This would allow the system to operate without the need of a human operator with a PC. The system is designed to be able to communicate with other devices. There are multiple ways that other devices could be connected. Such as USB, UART and I2C. USB would be the safest option since this connection has retention that prevents the cable from getting loose (USB-C).

The UART and I2C connection are easier to work with but would have to be done via pin headers which don't have the most rigid connections. Still since the system is not expected to move when placed it is not the biggest worry.

### 13.6 Find ways to integrate the system in other projects

Because of the modular nature of the PCB it would be very easy to use the system in other projects. The system is not just designed to perform a single task. The combination of having a programmable microcontroller, GNSS receiver and GNSS jamming detection on a single board can be very useful in the future. S[&]T has many GNSS related projects so looking for ways to use the MGID system could prove beneficial.

### 13.7 Ideal future use of the system

If this system can successfully and reliably detect the power level of GNSS interference. Multiple MGID systems could theoretically work together to possibly triangulate the GNSS interference jamming location using their last known positions. By comparing the different power levels that the MGID systems are monitoring a rough estimation can be made in which direction the jamming signal is coming from.

## 14 Problems and solutions

During the project there were multiple times that it ran into problems. This could either be because of problems with the planning or because of problems with the technical side.

### 14.1 Long component delivery time

During the first attempt at making a RF jamming detection circuit a couple of components had to be ordered. This resulted in a very long delivery time in which there was no more work to be done by the planning. To fix this the planning was changed so instead of doing the PCB design last, the RF jamming detection would be done last.

So the PCB design started without having a RF jamming detection circuit. This would have been added later during the internship.

### 14.2 Not able to use basic components for detecting RF signals

At the start of the project there were ideas on how to be able to detect GNSS jamming. The first idea was to simply measure the power output of the active antenna. There were two designs made that simply used basic components in order to attempt to measure the RF amplitude.

#### 14.2.1 Measuring the active antenna power

The design of the first attempt was to use a current measurement circuit on the ground line of the antenna. A differential circuit was used using single supply high accuracy operational amplifiers and a low ohm resistor in series with the active antenna. This system during testing could read out a voltage. However, the voltage that was measured was only DC voltage. The circuit was tested at multiple times a day and also multiple places. It always read out the same voltage. While there was no possibility to test the circuit with a jamming signal. It can be safely assumed that the circuit is not capable of detecting increasing power usage of the active antenna. This is most likely because the drawn power is in AC. Which a DC measurement circuit is not capable of detecting.

#### 14.2.2 Attempting to rectify the AC RF signal

Because measuring the DC voltage was not an option. Another design was made to attempt to rectify the RF signal so it could be measure by a DC differential circuit. Using high frequency Schottky diodes capable of switching at speeds even higher than GNSS signals. The circuit was almost identical to the DC measurement circuit except with a rectified AC signal instead.

This circuit also did not work. During testing there was no DC output measured. The reason for this is quite simple. GNSS signals do not have enough power to come even close to being able to switch Schottky diodes.

It was obvious that it would not work based on the low output of GNSS signals and the high forward voltage requirements of diodes. But since most of the circuit was reused from the previous circuit, and thus did not take long to make, it was worth a shot.

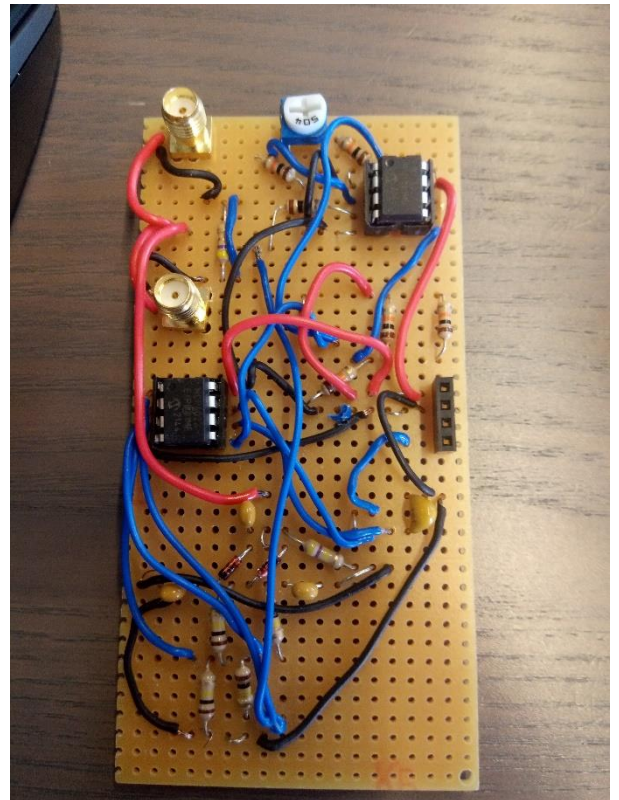


Figure 28 Failed design of a GNSS detection circuit using OPAMPS



### 14.2.3 Using premade IC's

After these two failed attempts at measuring GNSS antenna output it became clear that it is not possible to use basic electrical components to measure GNSS signals. These signals are both too fast and too low power. The best way forward is to create a circuit using premade IC's that are designed to work with high frequency RF signals.

After searching on the website of different semiconductor manufacturers, the MAX2015 chip was found, and this is now what is used in the current design. But sadly, the developer test circuits had a delivery time much longer than the internship.

### 14.3 Bad initial trace management

In the first draft of the PCB an the chosen GPIO locations for the STM32 microcontroller resulted in a non-ideal situation with way the traces ended up having to be laid out. Seen in Figure 29.

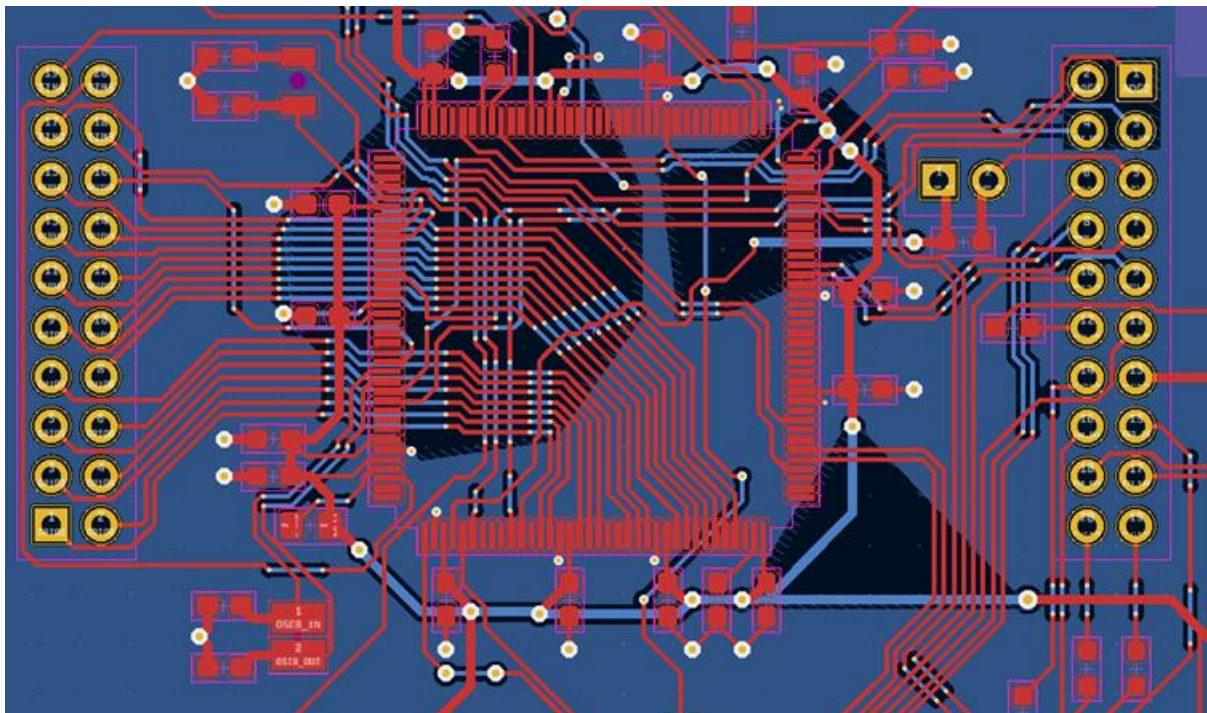


Figure 29 Initial STM32 trace layout

It was realized that this trace design was too messy to be approved for prototyping. The decision was made to go back to the choice of GPIO ports and change them around, so they result in a cleaner design. This design can be seen in the chapter for the STM32 in PCB design. In Figure 12.

## 14.4 No possibility of testing with jamming signals

While the project was about detecting GNSS jamming signals. There was no possibility to actually use a GNSS jamming signal to test the prototype. The possession and usage of jammers is illegal in the Netherlands. The science and control group did have a GNSS jamming test at a military base. But this was too early in the internship for the MGID prototype to participate in.

## 14.5 Running out of microcontroller memory

Because of the complexity and real-time nature of the system it was very easy to run out memory. Both flash and RAM memory were at risk.

The reason for the difficulties came from three separate reasons. The first reason is C++. C++ uses more memory than straight C. The possibility to use C instead was thought of but in the end, it was decided to use C++. The benefits of C++ outweigh its higher memory requirements.

FreeRTOS is the biggest user of memory in the system. However, FreeRTOS can be considered mandatory for a real-time system this size.

The final big user of memory, specifically RAM, are the unsigned integer buffers for the UART controllers. The receiver memory for the GNSS receiver UART needs to be quite large. Because the ZED-F9P microcontroller sends one long string of NMEA messages the buffer needs to be large enough to contain all the characters.

Before the F4 microcontroller another microcontroller was used. This microcontroller ended up not having enough memory by quite a lot. It had to be swapped out mid testing to the F4. The benefit of the modular software architecture design came into play here. It was very easy to convert the software to use the F4 microcontroller instead.

But even with this microcontroller it was still very important to keep the memory usage low. All variables are kept with as low of a memory use as is possible. Always using the lowest unsigned integer possible and making sure all unused variables are cleaned up, so they don't take up unnecessary RAM. Also making use of pointers to access large variables instead of cloning them in functions.

In the end only 15% of the RAM is used. And 6% of the flash memory. This might seem contradictory to the previous paragraphs. But if no thought was put into memory management the memory usage could easily be double the current values.

## 14.6 The float to string problem

During development a strange problem occurred related to using the C++ string library. The system would always crash when trying to use the function `std::to_string(float)`. This resulted in not being able to convert float values to human readable data to send over the UART. Two variables use float values. The GNSS geolocation and the pseudoranges of the satellites. Two different solutions were made to bypass this limitation.

### 14.6.1 GNSS location

For the GNSS location instead of storing the data in a float variable it is stored in a struct containing only integers see picture. Having the whole numbers and decimals separated. An extra variable is necessary to count the number of zeros before the possible decimal. Basically, the exponent 10 of the value. (Figure 30). Similar to how floats save their data. With the exception of not suffering from floating point error.

```
struct precisePos{
public:
    uint16_t whole;
    uint32_t decimal;
    uint8_t decimalZeros;

    void SetFromString(string inString);
    string GetFromString();
};

struct latlong{
    bool isNorth;
    bool isWest;
    precisePos latitude;
    precisePos longitude;
```

Figure 30 Precise pos structs



A custom function was made to both set the values of the position from a string. That would normally imply a float value. Another function is present to turn the position back into a string. So the data can be send to a host PC for a human operator to read.

#### 14.6.2 Satellite pseudoranges

Extracting the float data from the pseudoranges of the satellites was a bit more difficult to do. The GNSS location was send from the ZED-F9P using NMEA (character strings) whereas the pseudoranges are send using UBX protocol. So, bytes that directly translate to data. The pseudoranges are send in 8 bytes. These 8 bytes together form a double.

It would be possible to translate this double into multiple integers in the same way done with GNSS location. But it would require a lot of work and in the end it would not be important. Unlike the GNSS geolocation data it is not important that a human operator can read the pseudoranges. However, sending the raw data is also not an option. Because the databytes could be any number. It is also possible that the send data contains characters such as “\n”, “\r” or “\0”. Which would ruin the readability of the terminal.

The best thing to do is save the data as a 64 unsigned integer and turn it into a hexadecimal string during communication. A 64 unsigned integer has the same number of bytes as a double. The data of the double could be stored in the unsigned integer. To do this it is important to know that the incoming data is in 8 bit unsigned integers, a byte. So we could insert 8 bit integers into the 64 bit integer and then use bitwise operations to fill the 64 integer. But it would be faster to just write straight to the memory location of the 64 bit integer instead. Using a single clock cycle instead of dozens. Especially important since this code is inside a interrupt service routine.

What we do is, get the memory location of the 64 bit integer using a pointer and then cast it to a 8 bit integer pointer. Then by simply advancing the pointer in memory it is possible to fill the 64 bit variable quickly with 8 bit data. (Figure 31).

```
uint8_t* transferPtr = NULL;
transferPtr = (uint8_t*)&satteliteRanges[i].pseudoRange;
for(uint8_t k = 0; k < 8; k++){
    *transferPtr = tBuffer[UBX_PREDATA_SIZE + k + 16 + (i * 32)];
    transferPtr++;
}
```

*Figure 31 Fast data transfer using pointer*

With the double saved as a 64 bit unsigned integer it can easily be transformed into a 8 character hexadecimal string that can be send to the host (Figure 16). The host can than transform this string into a double variable. This process is shown in chapter 11 Processing data with Python scripts.

## 15 Competencies

During the internship it is expected that the intern is capable of a multitude of competencies that are expected of an electrical engineer. These competencies are analysis, design realization, managing and research.

### 15.1 Analysis

For the design of the system a thorough analysis was required. The system that was going to be designed did not have exact specifications. Therefore, there was a lot of design that had to be thought of from the ground up. It had to be clear which steps had to be taken in order to finish the project.

The analysis of the problem showed that it was going to be difficult to know exactly what exactly was going to be done over the duration of the project. There was a plan of approach made that was considered the safest and most flexible way to tackle the problem.

It was decided to start with software since this is the most malleable part. If things did not work out exactly as planned then it could easily be changed. As time went on work became more hardware related. By already having a software design it was clear what exactly was expected of the hardware. And if changes had to be made to the software then that was easy since, again, software is malleable and can thus quickly and easily change to suit the need of the system.

The most difficult part to get an analysis of was for the problem of how to detect GNSS jamming signals. There was a lot of research required to understand exactly how to do it. The initial analysis of the problem was wrong since it became clear that using basic electrical components it was not possible to detect GNSS signals. After these failures there was a better understanding of the problem. Allowing to reanalyze it and come up with a solution that works.

This analysis allowed for the design of a system that fulfills the order requirements. And best fits into the possibilities of S[&]T.

Analysis of the project did not stop after the initial gathering of order requirements. During the development the sensing and control team was working on processing data related to pseudoranges of satellites. I upgraded the system to also take in pseudorange data from the ZED-F9P. This data is valuable to the group and even though it was not in initial requirements it was added as a useful addition.

### 15.2 Design

During the internship a design was made for the MGID system. The design consists of three separate parts. A software design, an electrical design and a PCB design.

A lot of attention was put in the design of the system. The main goal of the system was to make it easy to expand for future developers. Since it was very clear from the start of the project that it would not be possible to finish it during a single internship. The design all three areas were all kept as modular as possible.

For the software an object-based approach using C++ was used. This approach using components allows for easy addition and modification. It is very clear by the naming of the class and their functions what everything does.

The electrical design is the least modular. The main thing that was kept in mind with the electrical design was the addition of GPIO pin headers for the STM32 microcontroller. Even though the MGID system doesn't use most of the GPIO ports. There are made available to future developers. Another thing that was added is the possibility to use different power

sources. During prototyping only power from a USB was used. But making it possible to use batteries or wall-brick supplies adds to the future applications of the system.

The PCB design is also kept as clean as possible. This was done in the way of dividing the board into four quadrants. Each with its own clear purpose. The lower right corner of the board is where the GNSS RF jamming detection circuit is located. Since it is expected that this part of the circuit might be changed in the future. There is open space left open so that a future developer has the room required if they need it.

### 15.3 Realization

While it was not possible to fully realize the MGID system due to a combination of time constraints and component shortage. A prototype using the Nucleo developer board and ZED-F9P was put together and used for testing.

During testing this setup was able to record the carrier to noise ratio of the signals. Which is considered the most important part of the system.

While there was no full realization of the hardware. The software that was written has been fully realized. The software works as intended and is capable of talking with the ZED-F9P GNSS receiver, can record ADC data and can send the results to a host PC for recording data.

There were also the two failed designs for the GNSS jamming detection circuit. And while these systems did not work in their function because of their design. The actual boards that were soldered together did work in their designed function it was just that the GNSS signals were not powerful enough to drive the system.

Lastly using the developer board prototype. Tests were able to be done and data could be logged and processed.

### 15.4 Managing

The project was completely managed by the intern himself. This was a difficult task because the problem was very open. There were many ways that the problem could have been tackled. This required a good planning and understanding of development processes. How to exactly go about working on certain aspects of the system. And when to move on to the next phase.

The project started off quite rapidly with quick developments on the software. Unfortunately, when it came to the hardware development multiple issues came to light. The first problem are the component shortages that are currently present in 2022 due to the ongoing worldwide crisis's such as COVID-19 and the war in Ukraine. This made designing the hardware very difficult.

Because it was no longer possible to work efficiently on the hardware side of the project, the quick decision was made to not lose time and instead go on to work on the PCB design and documentation of the project. The hardware design was going to be done last, while having to wait for components.

The resources required to make a successful hardware design were just not available. While a hardware design was made it has not been practically tested.

It was very tough but, in the end, it was possible to make the project into something that is useful and could easily be picked up and finished by future developers.

## 15.5 Research

In order to be able to complete the project it was necessary to do research in the workings of GNSS signals. GNSS signals are very difficult to work with. The combination of high carrier frequency and ultra-low power at earth's surface make it almost impossible to do anything without any specialized IC's.

At the start of the internship there was not much knowledge about GNSS signals. Most of this was basic understanding of the satellite constellations and that uses three directional positioning to geo locate a position.

The main goal of the research was to understand how GNSS signals work on a fundamental level. The research that has been done allowed a greater understanding on how GNSS receivers are capable of converting the low power signals into usable data. What GNSS signals exactly consist of and how the range calculations for satellites are done.

## 16 Conclusion

At the end of the internship a design is delivered containing a design for a mobile embedded system that is capable of measuring and monitoring GNSS signals. The design consists of an electrical design, PCB design, system design and software package. Additionally, documentation is written for future developers that include, detailed explanations of how the system is designed and works. And what steps need to be taken in order to finish the PCB prototype.

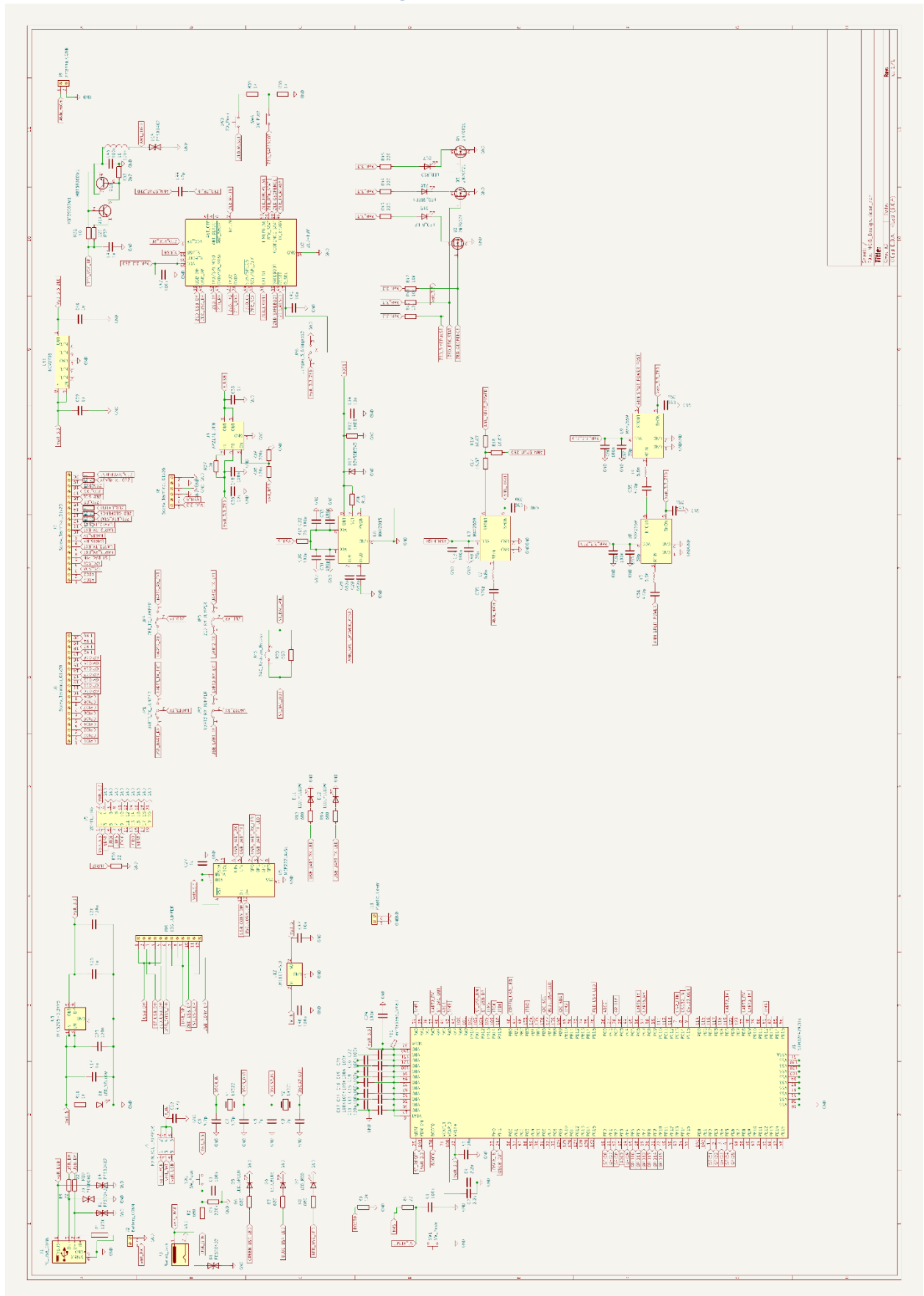
The MGID design is fully realized in theory and largely tested in practice, minus the GNSS power RF detection. Due to lack of component availability. The software is proven to work reliably and can collect data from the ZED-F9P. Both the software design and the hardware design are very modular and can easily be changed or expanded upon by future developers.

The completion of the internship and the fulfillment of the product order. Have shown that the competencies of Analysis, design, research, managing and realization are acquired.

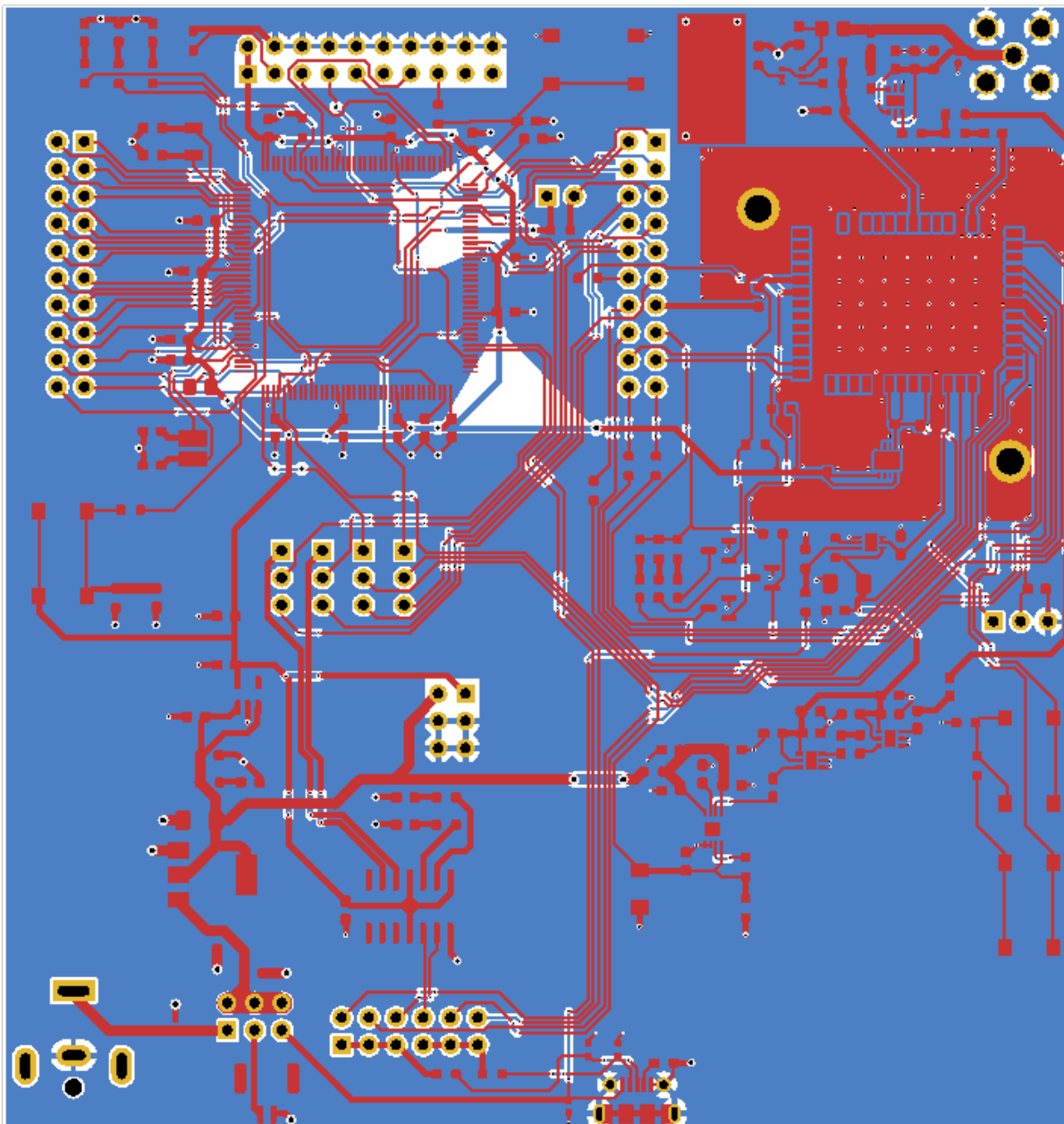
## References

- [1] [https://content.u-blox.com/sites/default/files/ZED-F9P\\_IntegrationManual\\_UBX-18010802.pdf](https://content.u-blox.com/sites/default/files/ZED-F9P_IntegrationManual_UBX-18010802.pdf)
- [2] <https://www.st.com/en/evaluation-tools/nucleo-f429zi.html>
- [3] <https://www.u-blox.com/en/product/zed-f9p-module>
- [4] [https://content.u-blox.com/sites/default/files/documents/u-blox-F9-HPG-1.32\\_InterfaceDescription\\_UBX-22008968.pdf](https://content.u-blox.com/sites/default/files/documents/u-blox-F9-HPG-1.32_InterfaceDescription_UBX-22008968.pdf)
- [5] <https://www.nmea.org>
- [6] <https://www.maximintegrated.com/en/products/comms/wireless-rf/MAX2015.html/>
- [7] [www.maximintegrated.com/en/products/comms/wireless-rf/MAX2659.html](http://www.maximintegrated.com/en/products/comms/wireless-rf/MAX2659.html)
- [8] <https://www.kicad.org>
- [9] <https://www.st.com/en/development-tools/stm32cubeide.html>
- [10] <https://www.st.com/en/development-tools/st-link-v2.html>
- [11] <https://www.eurocircuits.com>
- [12] <https://www.keil.com/pack/doc/CMSIS/RTOS/html/index.html>
- [13] <https://www.stcorp.nl>
- [14] <https://git-scm.com>
- [15] <https://www.maritimelobalsecurity.org/media/1043/2019-jamming-spoofing-of-gnss.pdf>
- [16] <https://www.u-blox.com/en/product/c099-f9p-application-board>

# Appendix A1: Electrical design



## Appendix A2: PCB design





## Appendix A3: Software main

```
// RX buffers for the uarts
uint8_t rxBuffer_user[RX_USER_BUF_SIZE] = {'0'};
uint8_t rxBuffer_Gnss[RX_GNSS_BUF_SIZE] = {'0'};

// freeRTOS thread handlers
osThreadId_t MGID_thread_Led_handle;
osThreadId_t MGID_thread_Uart_handle;

const osThreadAttr_t MGID_Led_attr = {
    .name = "MGID_task_Led",
    .stack_size = 256 * 4,
    .priority = (osPriority_t) osPriorityLow,
};

const osThreadAttr_t MGID_Uart_attr = {
    .name = "MGID_task_Uart",
    .stack_size = 1024 * 4,
    .priority = (osPriority_t) osPriorityHigh,
};

osMutexId_t MGID_mutex_uartHandle;
const osMutexAttr_t MGID_mutex_uart_attributes = {
    .name = "MGID_mutex_uart"
};

// create the core and its subcomponents
MGID_Core mgidCore;
MGID_USR_LED userLed;
MGID_UART userUARTClass;
MGID_UART_GNSS gnssUARTClass;
MGID_ADC adcClass;

void MGID_task_led_start(void *argument);
void MGID_task_Uart_start(void *argument);

//This is a protected main function to prevent the generated code from possibly interfering
void MGID_main(GPIO_TypeDef* usrLed, uint16_t usrLedPin, UART_HandleTypeDef* usrUart, UART_HandleTypeDef* gnssUart, ADC_HandleTypeDef*
adcHandle){

    osKernelInitialize();
    //init mutexes
    MGID_mutex_uartHandle = osMutexNew(&MGID_mutex_uart_attributes);

    //init MGID sub components for use
    userLed.InitUsrLed(usrLed, usrLedPin);
    userUARTClass.InitUart(usrUart, &MGID_mutex_uartHandle, rxBuffer_user, sizeof(rxBuffer_user));
    gnssUARTClass.InitUart(gnssUart, &MGID_mutex_uartHandle, rxBuffer_Gnss, sizeof(rxBuffer_Gnss));
    userUARTClass.InitParentCore(&mgidCore);
    gnssUARTClass.InitParentCore(&mgidCore);
    adcClass.InitADC(adcHandle);
    mgidCore.InitFeedbackLed(&userLed);
    mgidCore.InitUserUART(&userUARTClass);
    mgidCore.InitGnssUART(&gnssUARTClass);
    mgidCore.InitADC(&adcClass);
    mgidCore.SendUserLedState(1);
    //start threads
    MGID_thread_Uart_handle = osThreadNew(MGID_task_Uart_start, NULL, &MGID_Uart_attr);
    MGID_thread_Led_handle = osThreadNew(MGID_task_led_start, NULL, &MGID_Led_attr);

    // uint8_t buffer[] = "Pre FreeRTOS Test message\r\n";
    // HAL_UART_Transmit(usrUart, buffer, sizeof(buffer), 10);

    osKernelStart();//code in this function stops here. freeRTOS takes over

    while(1){

        //should never reach this

    }
}
```

## Appendix A4: MGIDCORE class

```
class MGID_USR_LED; //Forward declarations to prevent bad dependencies
class MGID_UART;
class MGID_UART_GNSS;
class MGID_ADC;
struct custom_time_t;

//MGID Core is the main component of MGID and has many subcomponents that handle functionality
class MGID_Core: public MGID_Core_Interface {
public:
    MGID_Core();

    //All these components need to be initialized in order for the MGID to function correctly
    virtual void InitFeedbackLed(MGID_USR_LED* newLed);
    virtual void InitUserUART(MGID_UART* newUart);
    virtual void InitGnssUART(MGID_UART_GNSS* newUart);
    virtual void InitADC(MGID_ADC* newADC);
    //start the receivers of the UARTs so they can start receiving messages
    virtual void SetUARTReceiver();
    //Send message to Host PC
    virtual void SendUserMessage(string msgOrigin, string sendString);
    //Send a new state to the feedbackLed
    virtual void SendUserLedState(uint8_t newState);
    //Check if there is an incoming user command
    virtual void CheckForUserCommands() override;
    //IRQ relay to the uart component IRQ
    virtual void HandleUARTIRQ(UART_HandleTypeDef* huartHandle);
    //IRQ relay to the ADC component IRQ
    virtual void HandleADCIRQ(ADC_HandleTypeDef* adchHandle);
    //This is the looping task that controls the LED called from the freeRTOS task
    virtual void MGIDTaskLed();
    //This is the looping task that controls the uart messages to the host called from the freeRTOS task
    virtual void MGIDTaskUart();
    //This relays the tick count to the timer. Called from the freeRTOS tickHandler
    virtual void CoreTick();
    //Basic get functions
    virtual UART_HandleTypeDef* MGIDGetUartUser();
    virtual UART_HandleTypeDef* MGIDGetUartGnss();

    virtual ~MGID_Core();

private:
    MGID_USR_LED* feedbackLed; // pointer to the LED component
    MGID_UART* userUart; //debugging UART to the PC (host)
    MGID_UART_GNSS* gnssUart; //uart that goes to the ZED GNSS Module
    MGID_ADC* adcModule; //ADC access
    custom_time_t time;
    virtual void blinkLed(); //call the blink on the feedbackLed
};

#endif /* INC_MGIDCORE_H_ */
```

## Appendix A5: Processing NMEA messages

```
void MGID_UART_GNSS::ProcessNMEA(uint8_t* tBuffer, uint16_t tBufferSize){
    if(tBuffer[0] != '$'){
        return;
    }
    uint16_t position = 1;
    string stringVar;
    stringVar = this->GetWordFromBuffer(&position);
    if(stringVar == "GNGLL"){
        stringVar = this->GetWordFromBuffer(&position);
        gpsPos.latitude.SetFromString(stringVar);

        stringVar = this->GetWordFromBuffer(&position);
        if(stringVar == "N"){
            gpsPos.isNorth = true;
        }else{
            gpsPos.isNorth = false;
        }
        stringVar = this->GetWordFromBuffer(&position);
        gpsPos.longitude.SetFromString(stringVar);
        stringVar = this->GetWordFromBuffer(&position);
        if(stringVar == "W"){
            gpsPos.isWest = true;
        }else{
            gpsPos.isWest = false;
        }
        stringVar = this->GetWordFromBuffer(&position);
        ProcessNMEATime(&stringVar);
        return;
    };
    if(stringVar == "GPGSV" || stringVar == "GLGSV" || stringVar == "GAGSV" || stringVar == "GBGSV"){
        GNSS_TYPE currentType;
        if(stringVar == "GPGSV"){
            currentType = GNSS_TYPE::GPS;
        }else if(stringVar == "GLGSV"){
            currentType = GNSS_TYPE::GLONASS;
        }else if(stringVar == "GAGSV"){
            currentType = GNSS_TYPE::Galileo;
        }else if(stringVar == "GBGSV"){
            currentType = GNSS_TYPE::BeiDou;
        }
        ProcessSattelite(currentType, position);
        return;
    }
    if(stringVar == "GNZDA"){
        stringVar = this->GetWordFromBuffer(&position);
        stringVar = this->GetWordFromBuffer(&position); //day

        timeTemp.SetDay(atoi(stringVar.c_str()));

        stringVar = this->GetWordFromBuffer(&position); //month
        timeTemp.SetMonth(atoi(stringVar.c_str()));
        stringVar = this->GetWordFromBuffer(&position); //year
        timeTemp.SetYear(atoi(stringVar.c_str()));
    }
}
```

## Appendix A6: MGIDCORE tasks

```
void MGID_Core::MGIDTaskLed(){
    /* Infinite loop */
    SendUserMessage(MGID_UART_MSG_CORE, "START USER LED THREAD");
    while(1){
        osDelay(MGID_CFG_LEDBLINKSPEED);
        blinkLed();
    }
}

void MGID_Core::MGIDTaskUart(){
    /* Infinite loop */
    SendUserMessage(MGID_UART_MSG_CORE, "START USER UART THREAD");
    SetUARTReceiver();
    gnssUart->InitGNSSModule();
    const uint16_t loopCount = MGID_CFG_UARTPINGSPED/MGID_CFG_CMDCHECKSPEED;//make sure this results in a value higher than 0
    uint16_t loopCounter = 0;
    while(1)
    {
        time = gnssUart->GetTime();
        osDelay(MGID_CFG_CMDCHECKSPEED);
        if(loopCounter >= (loopCount - 1)){//-1 because we count from 0
            SendUserMessage(MGID_UART_MSG_CORE, "PING");
            loopCounter = 0;
        }else{
            loopCounter++;
        }

        adcModule->PerformADCMeasurement();
        if(adcModule->IsInterferenceFlagSet()){
            //above nominal power detected on antenna. warn host
            SendUserMessage(MGID_UART_MSG_ADC, "INTERFERENCE POWERSURGE");
        }
    }
}
```