Track and Follow

With Cube Orange+

Victor Trytek Lemonnier
Anders Munch Risgaard
Nikolaj Juel Andersen

Bachelor of Engineering in Electronics, Aalborg University,
2024

Electronics and IT
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Track and Follow - Cube Orange+

Theme:
A 7th-Semester Project

Project Period:
Fall Semester 2023

Project Group:
711

Participant(s):
Victor Trytek Lemonnier
Anders Munch Risgaard
Nikolaj Juel Andersen

Supervisor(s):
Anders la Cour-Harbo

Copies: 1

Page Numbers: 99

Date of Completion:
January 4, 2023

Abstract:

The general purpose of this report is to document the design and implementation of a track-and-follow system on a drone with a Cube Orange+ flight controller. The core functionality involves obtaining GNSS signals from an external module and ESP32 transmitting them to the flight controller guiding the drone toward the obtained location. The drone is equipped with an onboard ESP32, handling incoming messages from the external ESP32 and parsing the data via MAVLink protocol to the flight controller.
The report delves into topics such as NMEA communication protocols, ArduPilot open-source coding, and implementation of test results in simulation, before evaluating the system's performance.

Abbreviations

Abbreviations	Original word or meaning
UAS	Unmanned Aerial System
GNSS	Global Navigation Satellite System
NMEA	National Marine Electronics Association
MCU	Microcontroller Unit
IoT	Internet of Things
VLOS	Visual Line of Sight
FER	Frame Error Rate
CEP	Circular Error Probability
GCS	Ground Control Station

Contents

Preface

This Project was developed in the period between the 1st of September 2023 and the 4th of January 2024, by 7th-semester group 711, Bachelor of Engineering in Electronics, at Aalborg University which provided a guidance counselor.

Project group 711 would like to extend a big thank you to our guidance counselor Anders la Cour-Harbo for his guidance throughout this project.

The report was written with the anticipation that the reader has a basic understanding of electronics. For units and symbols the standard ISO 80 000 is used. The bibliography and citation follow the IEEE style.

When describing code throughout the report, class names are highlighted with **bold font** while methods and variables are highlighted with `typewriter font`.

Aalborg University, 4th of January 2024

Anders Munch Riisgaard
ariisg20@student.aau.dk

Victor Trytek Lemonnier
vlemon20@student.aau.dk

Nikolaj Juel Andersen
njan20@student.aau.dk

1 Introduction to Track-and-follow

1.1 Concept

The general purpose of track-and-follow is, as the name implies, to select an object or person and follow its movements or position. An example of this is a follow-spot on a stage where the light is operated manually to highlight the main event. Generally what defines the expression track-and-follow is that something replicates the movement or path of a particular entity. This does not necessarily have to be done automatically. However, doing so, makes it more convenient in a lot of cases.

Track-and-follow technologies have revolutionized various industries, particularly the film industry and search and rescue operations. These technologies embrace a range of tools and techniques that enable the precise tracking and monitoring of individuals, objects, or locations. In the film industry, aerial photography utilizing Unmanned Aerial Systems (UASs) equipped with advanced tracking systems is very common. UASs can follow subjects smoothly, capturing dynamic and visually striking shots from various angles, enhancing the overall cinematic experience [1].

Furthermore, thermal sensors play a role in search and rescue operations. Equipped with thermal imaging cameras, these UASs can detect body heat and thermal signatures, aiding in locating missing persons or individuals in challenging environments [2, 3].

1.2 UAS - Drones

The first pilotless vehicles were developed at the start of the 20th century in Britain and the USA for military use. A small radio-controlled aircraft was tested

in 1917 in Britain and the USA in 1918. The term 'drone' was introduced around 1935, when the British developed UASs as targets for training [4]. In recent years the development of drones has exploded as their applications now range from recreational use to warfare.

Drones are aircraft that operate without a human pilot onboard. They can be remotely controlled or can fly autonomously through software-controlled flight plans. Drones come in many different sizes, ranging from small to large, and are equipped with different equipment, such as cameras, sensors, or other payloads. Drone technology has evolved rapidly over the years, becoming more versatile and accessible. Their applications can be found in fields such as photography, videography, agriculture, search and rescue, surveillance, and more [5].

The advancement of UAS technology necessitates corresponding regulations. In Denmark, drone legislation is governed by both common European rules and national regulations. The EU Drone Regulation has been in effect in Denmark since December 31, 2020. Notably, professional drone pilots are allowed to fly in populated areas, as long as they possess a drone pilot license. Drone owners with drones weighing more than 250 grams or flying faster than 50 km/h need to register with the Danish Transport, Construction, and Housing Authority [6]. Furthermore, pilots may not fly their drones too close to military areas, helipads, police stations, and public airports [7].

1.2.1 Autopilot

UASs can be piloted by remote control via radio or they can be controlled autonomously. The traditional way of controlling the movement of UASs (radio) gives the pilot full responsibility for the UAS. This comes with a risk of the pilot crashing or damaging the UAS. Further, the pilot needs to always be controlling the UAS manually and can thereby never take his attention away from the flight. Autopilot makes it possible to plan a route for the UAS to fly autonomously, which in turn makes it much easier for the pilot and expands on the capabilities and tasks the UAS can take on.

Autopilots in UASs can either be fully or semi-automatic. Semi-auto-piloted UASs have a variety of safety configurations that ensure that the UAS does not crash or fly into unauthorized air spaces as well as helping to stabilize the flight [8]. Fully-auto-piloted UASs can be preplanned to fly routes independently of a pilot. This also makes it possible for a single pilot to control many UASs si-

multaneously [8]. For fully-auto-piloted UASs, several open-source commercial autopilot systems exist, like PX4, and ArduPilot.

1.3 Project scope

For this project, a Cube Orange+ flight controller was made available. Therefore, the goal of this project is to gather knowledge about the Cube Orange+ and its environment. It is chosen to center a problem-based project around the Cube Orange+. For this project, it is decided to mount the Cube Orange+ to a UAS, specifically a quadcopter drone, and make it track, and follow another moving object. By doing so, an external system can interface with the Cube Orange+ and integrate the system data into the firmware, thus obtaining knowledge about the firmware and the Cube Orange+ itself.

1.3.1 Cube Orange+ as flight controller

The Cube Orange+ is an advanced autopilot utilized for UASs. It is a versatile, open-source, and customizable flight controller that provides a platform for controlling different aspects of a drone's flight. This includes navigation, stabilization, communication, and more [9].

Being open-source means that the software's source code is freely available and editable. This allows developers and enthusiasts to modify and tailor the autopilot to specific needs and preferences. This ensures a robust community and support ecosystem. All users can provide assistance, share insights, and contribute to the improvement of the flight controller.

The Cube Orange+ uses GNSS data and other sensors to guide the drone, ensuring it follows the desired route accurately. As well as using gyros and accelerometers to maintain stability. Furthermore, it transmits important flight data, such as altitude, speed, and battery status, allowing real-time monitoring of the UAS's performance. All these features are advantageous for tasks such as; mapping, surveying, monitoring large areas, and capturing steady and clear photos and videos [9].

2 Track-and-follow analysis

In this chapter, track-and-follow modes are further investigated, as well as rules and regulations about UASs as they are necessary to obey. To achieve the desired functionality of a track-and-follow solution using the Cube Orange+, it is necessary to have a module that receives data containing the whereabouts of the objects being tracked relative to the drone. This is followed by an analysis of the firm- and hardware used with the Cube Orange+.

2.1 Autonomous track-and-follow

Traditionally drones have been controlled manually by radio, but in recent years track-and-follow or FollowMe mode has advanced [8]. Automatic track-and-follow technologies are systems that enable real-time monitoring and tracking of objects. These technologies function through hardware and software components that collaborate to provide location data for tracked objects. Manufacturers have made technologies that can handle track-and-follow, on the devices they sell. Track-and-follow modes are referred to, using various names such as; "Motion tracking", "FollowMe" and "ActiveTrack" [10]. What differentiates followMe from fully-auto-piloted mode is that the route is generated at runtime rather than being pre-planned.

To do this, different specialized tags or sensors can be used on objects to emit signals. These can be in the form of QR codes, barcodes, RFID tags, GNSS modules, and IoT-enabled devices. Once tagged, these objects continuously transmit data to a centralized system or a cloud-based platform. The data may include information like location, temperature, humidity, motion, and other relevant parameters, depending on the purpose of tracking.

2.1.1 Existing solutions

DJI is a widely known Chinese technology company that specializes in manufacturing various drones. DJI has developed both "Follow-me" and "ActiveTrack" for some of the drones they offer. The following will revolve mainly around DJIs follow-me and ActiveTrack features.

ActiveTrack on a DJI drone enables the user to have the drone act in accordance with the movement of the user. This can be handy if one wants photography while performing activities like skiing, running, or biking [10]. ActiveTrack utilizes the mounted camera to lock onto an object and thereby, track it. The interface for ActiveTrack allows the pilot to mark objects that it must follow. As a built-in functionality for ActiveTrack, it utilizes sensors and actuators to avoid any obstacles when flying. Besides ActiveTrack, DJI also provides its Follow-me feature. The Follow-me feature relies solely on the GNSS signal from the controller of the drone. Following an object's coordinates and keeping a set distance from that, implies that the object does not have to be in the frame at all times. However, when relying on communication between a device and the drone itself the drone can lose signal. In the event of this, the drone will hover in the air until a stable connection can be re-established [11]. The distance from the drone to the pilot in both ActiveTrack and Follow-me mode is customizable but in follow-me mode, a 5-10 meter distance together with at least 3 meters altitude seems to give the best results [12].

2.2 Rules and regulations

When flying with UASs, there are strict regulations to follow to be able to fly legally. These rules and regulations depend on multiple parameters, such as size and weight. Furthermore, commercially produced drones are required to be marked with a C-marking classification from 2024 [13]. This C-marking secures that that drone satisfies necessary requirements, and divides the drones into weight categories. According to which C-marking the drone has, the operator is obliged to have a certain license/certificate to fly the type of drone. The table of C-markings/certificated based on weight can be seen in Table 2.1.

C-marking	Weight	Certificate
C0	0 - 249 g	No certificate required
C1	250 g - 899 g	A1 theoretical
C2	900 g - 4 kg	A2 competence
C3 & C4	4 kg - 25 kg	A3 theoretical

Table 2.1: C-Marking indicates the weight of a commercial drone [13].

Drones that are not marked with a C-certification are called "legacy" drones. When piloting a legacy drone, the operator still needs a certificate, but the weight-to-certificate is differently distributed. This is shown in Table 2.2.

Legacy drone	
Weight	Certificate
0 - 249 g	No certificate required
250 g - 499 g	A1 theoretical
500 g - 2 kg	A2 competence
2 kg - 25 kg	A3 theoretical

Table 2.2: Legacy drone table [13].

These different categories of certificates point to a slightly different set of rules. The difference in these rules mainly focuses on the overflight of people, horizontal safety distances, and the pilot following VLOS (Visual Line of Sight) operation [14]. Some of the important takes from the "EU UAS regulation" are listed below:

- The UAS must not exceed an altitude of 120 meters above ground level [14].
- The UAS needs to maintain at least a 50-meter safety radius to non-involved people, but the horizontal distance to the non-involved people may never be less than the vertical distance off the ground. However, this can vary depending on the certificate/C-marking of the UAS but is still used as a guideline [14].
- In track-and-follow mode, a maximum follow distance of 50 meters is allowed. This means that the UAS may never be more than 50 meters away from the pilot/object that it is following. Although there is no minimum distance to the person involved by regulation, it is still practiced to not fly closer horizontally than the vertical altitude of the UAS [14].

- If the UAS exceeds 50 meters from the pilot/object that it is following, it must stop its actions and hover in place. If this is the case resuming flight is strictly prohibited, and the UAS must be restarted in able to reengage the track-and-follow system. Lastly, the pilot must always be able to retake control over the UAS at any point during the track-and-follow operation [15].

2.3 Wireless object-tracking

Examining the track-and-follow solutions in the products developed by DJI for their drones, they have a solution that utilizes advanced image technology, while their other solution consists solely of communication and exchange of GNSS coordinates.

GNSS modules are individual modules that can receive GNSS signals directly from satellites. A GNSS signal consists of longitude, latitude, and altitude data among other things [16]. This allows for monitoring the position of the object which can then be compared with the drone's position.

Image recognition is another method that could be used for a track-and-follow system. This method utilizes computers to track objects in photos or videos. It uses AI and machine learning to process the visual data. Although this method is ubiquitous it is very complex as it uses deep learning algorithms that take inspiration from our biological nervous system to form an intricate network of data and learning capacities [17].

In addition to determining the object by using image recognition, are thermal cameras. Using thermography is advantageous since it is independent of lighting. This means that object tracking can be done at any time of the day given that the object has a different temperature than the environment. For this specific reason, it is more aimed at tracking humans and animals as their body temperature would stand out significantly [3].

2.4 Methods for transceiving location data

Relying solely on GNSS positioning, a connection between a GNSS module and the drone must be established. The drone needs a receiver to capture the GNSS data transmitted wirelessly.

One approach involves an onboard microcontroller unit (MCU) with an antenna mounted on the drone. The MCU would act as an intermediary, receiving the data and relaying it through an interface to the drone's Cube Orange+ for the data to be handled. The Cube Orange+ has certain limitations when it comes to communication methods. As a standard, the Cube Orange+ supports UART, CAN, and I^2C [9].

Another approach is making a connection between the GNSS module and the drone through the Cube Orange+ directly. The Cube Orange+ can connect to a PC through USB cables, Telemetry Radios, Bluetooth, and IP connections [18]. Therefore, a GNSS module connected to an MCU should be able to connect to the Cube Orange+ with, for instance, Bluetooth.

2.4.1 Additional module on the drone frame

Whichever system is chosen as the track-and-follow solution, the dimensions, weight, and placement of the system have to be taken into consideration. Adding weight to the drone might change the center of balance and add complexity to flying, which then requires a correction for possible balance offsets. Luckily the Cube Orange+ acts as a flight controller and acts as a feedback loop controller to make sure the drone can stably hover in the air even with an off-center balance [19].

If such a system is mounted on the drone it will have to draw its power from the drone batteries as well. This will require some sort of interface between the batteries and the system to ensure the correct voltage is met for the system to run operationally.

3 Problem definition

From the problem analysis above, a problem definition is developed to form the frame and direction of this project. The problem definition states:

> *How can a track-and-follow system using GNSS data be developed and integrated into an already existing Cube Orange+ autopilot drone system, such that knowledge and familiarity of the Cube Orange+ are obtained?*

4 Proposed solution

This chapter's objective is to describe a solution for the track-and-follow system, which can then serve as a basis for establishing its requirements. The following is a proposed solution to solve the problem definition 3.

The functional diagram of the proposed solution is shown in Figure 4.1.

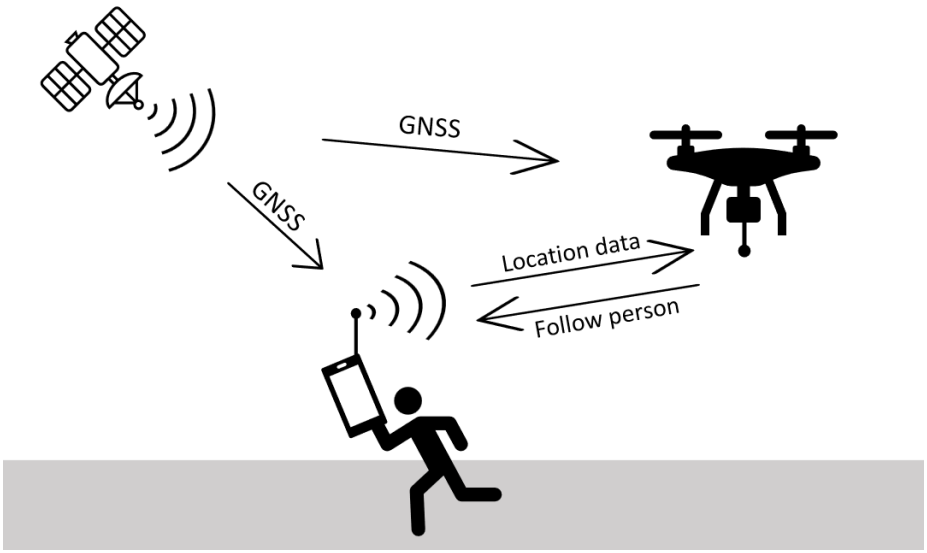


Figure 4.1: Functional diagram of the proposed solution.

In this proposed solution, a person is equipped with a GNSS (Global Navigation Satellite System) module, which receives information about its location from satellites and transmits it to another module mounted on a drone. The module then transmits the data further to the drone. The drone, in response, uses this data to follow the person, maintaining a fixed distance.

GNSS technology for tracking is chosen on the basis that image recognition is out of the scope of this project. Thermography is limited to humans, animals, and

other heated objects and is therefore neglected. Further, GNSS does not require the drone to have a camera mounted and enables any objects that have a GNSS module mounted to be followed. This is a significant advantage, as the drone's performance remains unaffected by its surroundings and environment.

Ideally, the communication between Cube Orange+ and the user-held GNSS module should be direct and wireless. However, the option of changing the module placed on the drone is favorable in case the communication range is not sufficient. Also having an extra link enables easier debugging and troubleshooting, especially since the project's time frame is limited.

This approach facilitates a real-time, reliable track-and-follow system where a drone autonomously follows a person, allowing for various applications, such as aerial photography, surveillance, or search and rescue operations.

5 System frame

This chapter states the physical limits of the system frame with the received hardware for this project. Due to the scope of the project, the system is to be framed around the Cube Orange+, which is well suited for the specific open-source firmware called ArduPilot. Furthermore, necessary hardware has been made available by Aalborg University to explore the project definition.

5.1 Available hardware

In this section, the given hardware is described as it may limit system requirements. The hardware made available from the University is the following:

- Cube Orange+
- Drone kit
- Battery pack
- GNSS module
- Telemetry radio set
- RC controller
- RC receiver
- Spare parts for the drone kit

Cube Orange+

The Cube Orange+, introduced in section 1.3.1, operates within an open-source environment. ArduPilot, the open-source firmware, provides a flexible and customizable codebase that can be specified to the specific objective of this project. The firmware provides compatibility with many types of vehicles including rovers, planes, submarines, and drones, which makes it a universal autopilot module.

In terms of hardware, The Cube Orange+ is equipped with a H7 processor and has a Cortex M7 with double-precision (DP) FPU, 400MHz CPU, a Cortex M4 running at 200 MHz, 2 MB of Flash, and 1 MB RAM [20]. It has Advanced Auto avoidance features with the help of a 1090 MHz customized ADS-B receiver from uAvionix, that receives attitude and location data of commercial manned aircraft [20]. Moreover, it has customized carrier boards that can be optimized for specific applications [20].

The carrier board facilitates the organization of all the relevant I/O components, connecting them to their respective ports and sockets. This design ensures a simple process for assembling, disassembling, and servicing the drone without the need for soldering [21].

As mentioned in section 1.3.1 the Cube utilizes multiple communication methods. The carrier board facilitates the utilization of these communication methods. Table 5.1 provides an overview of the UART ports on the carrier board, outlining the default systems they control. These can be modified to be used for different systems through a ground control station. This feature enables the customization of UART ports; for instance, the TELEM1 port, as well as many other ports, offers versatility by accommodating up to 32 different protocols.

Serial Port Mapping

UART	Device	Port
USART2	/dev/ttyS0	TELEM1 (flow control)
USART3	/dev/ttyS1	TELEM2 (flow control)
UART4	/dev/ttyS2	GPS1
USART6	/dev/ttyS3	PX4IO
UART7	/dev/ttyS4	CONSOLE/ADSB-IN
UART8	/dev/ttyS5	GPS2

TELEM1, TELEM2 ports

Pin	Signal	Volt
1 (red)	VCC	+5V
2 (blk)	TX (OUT)	+3.3V
3 (blk)	RX (IN)	+3.3V
4 (blk)	CTS (IN)	+3.3V
5 (blk)	RTS (OUT)	+3.3V
6 (blk)	GND	GND

Table 5.1: Serial- and Telem ports on the Cube Orange+ carrier board.

Drone kit

The available drone kit is a **Holybro X500 V2 ARF** quadcopter kit, with a lightweight carbon frame and four 2216-920KW brushless DC motors [22]. In addition to the main top and bottom plate of the drone structure, a platform board, with the dimensions 93 mm x 65 mm, on the back of the drone is added for the GNSS module to be mounted. The assembly instructions of the drone kit, suggest that a microcontroller can be mounted on the bottom of this platform board. The weight of the drone kit is 610 g. The drone structure with some added components is illustrated in Figure 5.1.

Figure 5.1: Holy Bro X500 V2 ARF kit assembled where the platform board is on the right side of the body.

Battery pack

The drone is powered by a 14.8 V 4-cell high discharge Li-Po battery pack which is mounted underneath the body of the drone. This battery has a capacity of 3300 mAh and weighs 359 g including wire, plug, and case.

GNSS module

The GNSS module **Holybro M9N GPS 6Pin - 2nd GPS**, is a second-generation GNSS module that supports multiple satellite connections such as GPS, GLONASS, Galileo, and BeiDou to ensure better connectivity due to more available satellites. It features a 6-pin interface, with serial and I^2C connections, that allows for easy implementation and compatibility with most flight controllers including the Cube Orange+ [23].

Telemetry radio set

The Telemetry set '**TELEMETRY RADIO SET V3 433MHZ**', including two radio modules and some USB cables, can be used to connect the flight controller on the drone to a Ground Control Station (GCS) through radio communication. This allows for GCS software to control the drone with a predetermined mission [24]. The radio modules are equipped with both USB-A and JST-GH plugs which makes it easy to connect to either a PC or a flight controller as The Cube Orange+. Furthermore, it has automatic detection of firmware like ArduPilot, which makes for a seamless startup [24].

RC controller

The **RadioMaster TX16S Mark II** is a transmitter designed with a 4-in-1 multiprotocol or ELRS FR system and open source firmware as OpenTX and EdgeTX [25]. Connecting a RadioMaster receiver to the flight controller allows for manual control of the drone when the TX16S is calibrated. It has support for telemetry data transmission and Hall sensors for the joysticks, which ensure contactless position sensing of cross controllers [25].

Figure 5.2: RadioMaster TX16S Mark II [25].

Figure 5.3: RadioMaster RP1 ExpressLRS Nano Receiver V2 (2.4GHz) [26].

RC receiver

The receiver given is a **RadioMaster RP1 ExpressLRS Nano Receiver V2 (2.4GHz)** running ExpressLRS (ELRS) as the controller mentioned above [26]. This makes binding them an easy process and the high-refresh-rate RF module is also suitable for long-range.

5.2 ArduPilot firmware

As ArduPilot is an open-source firmware compatible with all the Pixhawk/Cube series, it can be used to run any vehicle compatible with the Cube series. In this case, the Orange Cube+ is mounted on a drone, and the ArduCopter directory is the vehicle-specific software used in this project.

Apart from the vehicle-specific software, ArduPilot also has shared libraries with functions compatible with multiple vehicle types. This is convenient for the vehicle-specific software code to remain as short and structured as possible.

Lastly, ArduPilot has a Hardware Abstraction Layer (HAL) that acts as an interface between the many compatible hardware solutions and the ArduPilot software. This architecture is shown in Figure 5.4.

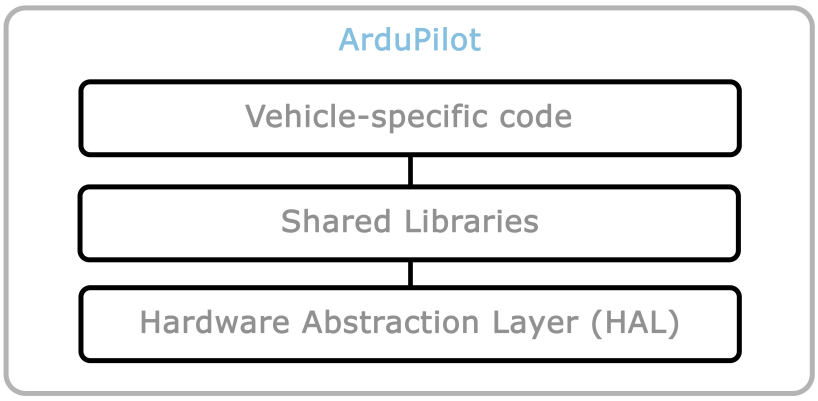


Figure 5.4: The simplified architecture of the ArduPilot Firmware.

Since most of the relevant software for this project is in the vehicle-specific ArduCopter code, only ArduPilot functionality relevant to this project will be investigated further.

Since the entire ArduPilot is open-source, a sense of better structure is sometimes wanted. Dealing with only the ArduCopter code there are a lot of files, and not all files are important for everything. One of the most important ones is the `Copter.cpp` file which acts as a `main.cpp` file. It consists of the main **Copter** class and uses the `AP_Scheduler` library to schedule all the different tasks running the Copter flight controller.

`Copter.cpp`

The first thing in `Copter.cpp` is the scheduler table, which utilizes two different functions to schedule various tasks. An example of the scheduler table is shown in Listing 1.

```
1 const AP_Scheduler::Task Copter::scheduler_tasks[] = {
2     FAST_TASK(run_rate_controller),
3     ...
4     SCHED_TASK(rc_loop,              250,    130,  3),
5     SCHED_TASK(throttle_loop,         50,     75,  6),
6     ...
7     }
```

Listing 1: A few selected tasks listed in the scheduler table of `Copter.cpp` to demonstrate the scheduler list [27].

As seen in Listing 1 there are two different functions called in the scheduler table. The first is `FAST_TASK()` shown in line 3. The other task displayed in lines 5-6 is the `SCHED_TASK()`. This function takes more inputs than the `FAST_TASK()` function, as seen in Listing 2.

```
1 #define SCHED_TASK(func, _interval_ticks, _max_time_micros, _prio)
```

Listing 2: Definition of **SCHED_TASK**.

Here `_interval_ticks` sets how often the task should run and `_max_time_micros` defines how much time should be allocated at maximum to the task in microseconds and `_prio` defines what priority the task should have. It is important to note that all priorities must be unique. The functions `SCHED_TASK()` and `FAST_TASK()` are defined from macros in the `AP_Scheduler` library.

After the scheduler table, the functions that are members of the **Copter** class are defined, one of them being potentially relevant is the `set_target_location` demonstrated in Listing 3. Where an instance of the **Location** class is used to set the destination, when in guided mode.

```
1 bool Copter::set_target_location(const Location& target_loc){
2     // exit if vehicle is not in Guided mode or Auto-Guided mode
3     if (!flightmode->in_guided_mode()) {
4         return false;
5     }
6     return mode_guided.set_destination(target_loc);
7 }
```

Listing 3: One of the functions defined in the `Copter.cpp` file that inherit the Copter class [27].

After all the functions are declared the last thing happening in `Copter.cpp` is the HAL callback, which tells the HAL what vehicle the flight controller has to control. This is shown in Listing 4.

```
1     Copter copter;
2     AP_Vehicle& vehicle = copter;
3     AP_HAL_MAIN_CALLBACKS(&copter);
```

Listing 4: One of the functions defined in the `Copter.cpp` file that inherit the Copter class [27].

AP_Scheduler

As mentioned above, the two different scheduler tasks in `Copter.cpp` are defined using the macros from the `AP_Scheduler` library, where the class name and the address of the class are inputs too. The two functions are defined as macros and one is shown in Listing 5.

```
#define SCHED_TASK_CLASS(classname, classptr, func, _rate_hz,
    _max_time_micros, _priority)
```

Listing 5: Definition of `SCHED_TASK_CLASS`.

The difference between the two macros that define the scheduler tasks, is that the `FAST_TASK_CLASS` sets the parameters `.rate_hz` and `.max_time_micros` to 0, and also tells the scheduler to execute them as fast as possible with the highest priority.

Location.cpp

Since location data is used throughout the system, and the **Location** class is used to set the destination in Listing 3, it makes sense to look into the class. An instance of this class represents a geographical location with latitude, longitude, and altitude, as well as methods for various location-related calculations and operations. The latitude, longitude, and altitude are represented as 32-bit integers. This is seen in Listing 6.

```
class Location{
  public:
    ...
    int32_t alt;
    int32_t lat;
    int32_t lng;
    ...
}
```

Listing 6: Coordinate definitions in the **Location** class.

Even though the variables are saved as 32-bit integers, the values are supposed to be decimal degrees. To justify the decimal degrees as integers the values are scaled with a factor of 10^7 or 10^{-7}. This gives a maximum resolution of 7 decimals of the decimal degree, which translates to a resolution of approximately 11,1 mm

at equator [28]. This scaling is documented in the `Location` header file as private variables and is shown in Listing 7.

```
private:
    ...
    // scaling factor from 1e-7 degrees to meters at the equator
    // == 1.0e-7 * DEG_TO_RAD * RADIUS_OF_EARTH
    static constexpr float LOCATION_SCALING_FACTOR = LATLON_TO_M;
    // inverse of LOCATION_SCALING_FACTOR
    static constexpr float LOCATION_SCALING_FACTOR_INV = LATLON_TO_M_INV
   ;
};
```

Listing 7: Scaling factors in the **Location** class.

The class also offers versatile functions for distance and bearing calculations, altitude frame conversions, and vector manipulations. Some functions can sanity-check the location data to ensure the right format is used. This class seems to be useful for storing and implementing Location data.

6 Requirements

This chapter describes the essential requirements for the track-and-follow system, aiming to provide a comprehensive understanding of its functionality and establish a structured framework for product development. The requirements are crucial in defining the system's expected capabilities. The requirements are divided into functional and technical requirements respectively.

To define the modules illustrated in the proposed solution in Figure 4.1, the person with the device transceiving GNSS signal is now defined as the Position module (**Pos**). The drone's receiving end is defined as the Link module (**Link**), linking the signal from the Position module to the drone. The flight controller is defined as the Cube module (**Cube**). The requirements for the communication between the Position and Link module are defined as **Com**.

The requirements are divided into tables with their ID, which module they regard, a description of the requirement, and their traceability in the report.

6.1 Functional requirements

Functional requirements encompass all aspects of a system's functionality, and they are non-technical and not directly testable.

Functional requirements			
ID	Module	**Description**	**Traceability**
1.1	Link	Needs to be able to receive and save data.	Section 2.
1.2	Link	Needs to be able to send data through the interface to the Cube Orange+.	Section 2.4
1.3	Link	Needs to be able to decode the received packages.	Chapter 4
1.4	Link	Needs to be able to format the data appropriately for the Cube Orange+ to handle it.	Chapter 4
1.5	Cube	Needs to be able to send the decoded data to the appropriate place in the ArduCopter code.	Chapter 4
1.6	Cube	Needs to be able to calculate the desired destination of the drone.	Chapter 4
1.7	Cube/ Link	Must be powered by the drone on which it is mounted.	Section 2.4.1
1.8	Pos	Needs to be able to transceive GNSS data wirelessly.	Section 2.4
1.9	Pos	Needs to be appropriately sized to be carried in hand.	Chapter 4

Table 6.1: Functional requirements for the system.

6.2 Technical requirements

The technical requirements encompass all specifications related to the system's underlying technology and infrastructure. These are specific and can be tested for compliance.

Technical requirements			
ID	Module	**Description**	**Traceability**
2.1	Link	The microcontroller needs to output a signal of the type; Serial, CAN or I^2C.	Section 2.4.
2.2	Link	Needs to be able to receive data wirelessly with a minimum distance of 50 m.	Section 2.2
2.3	Link	Needs to weigh a maximum of 1031 g.	Table 2.2
2.4	Link	The maximum allowed dimensions is 93 mm x 65 mm x 1500 mm.	Section 5.1
2.5	Pos	Needs to be able to transmit data wirelessly with a minimum distance of 50 m.	Section 2.2
2.6	Com	Data transmission between Pos and Link with a maximum of 1-second intervals.	Section 9.2
2.7	Cube	The drone needs to maintain a distance of 7.5 meters to the object it is following with a margin of $\pm$ 2.5 m.	Section 2.1.1
2.8	Pos	Receive data with a horizontal position accuracy of maximum 2.5 m.	Requirement 2.7
2.9	Com	The data transmission between the Pos and the drone needs to have a maximum FER of 1%.	Section 9

Table 6.2: Technical requirements for the system.

6.3 Test specifications

To validate both the requirements and the proposed solution, a variety of tests are specified. There are three different ways of testing the solution. First, we have the Unit test, where one specific unit of the system is tested independently from the system. Secondly, a verification test is where the full system, or a part of the system is tested to see if it fulfills the requirements. Lastly, there are validation tests, which validate if the solution solves the problem stated in the problem definition regardless if it passes the requirements or not. Here at least one of each test is specified.

6.3.1 Unit test

Position module unit test

Requirement	Receive data with a horizontal position accuracy of maximum 2.5 m.	Test: A.1
Purpose	The purpose of this test is to validate the horizontal position accuracy for the specific GNSS submodule used in the Position module.	
Success criteria	The success criteria of this test is that 50% of the coordinates are within 2.5 meters of each other.	

6.3.2 Verification test

Transmission range verification test

Requirements	The data transmission between the Pos and the drone needs to have a maximum FER of 1%.	Test: A.2
	Needs to be able to receive data wirelessly with a minimum distance of 50 m.	Test: A.2
Purpose	The purpose of this test is to measure the Frame Error Rate (FER) of the communication between the Position module and the Link module. This is done at 50 meters to also test the maximum data transmission range of the technical requirement 2.9.	
Success criteria	The success criteria of this test is that the calculated FER does not exceed 1%.	

Speed of data transmission acceptance test

Requirement	Data transmission between Pos and Link with a maximum of 1-second intervals.	Ch: 10
Purpose	The purpose of this test is to control and observe the data flow of the position data between the Position module to the Link module. This is done on the full system to see if the Cube module receives the data at the given sending rate.	
Success criteria	The success criteria of this test is to send and receive readable data continuously with no longer intervals of 1-seconds.	

Follow distance acceptance test

Requirement	The drone needs to maintain a distance of 7.5 meters to the object it is following with a margin of ± 2.5 m.	Ch: 10
Purpose	The purpose of this test is to see if the full drone system is able to maintain a 5-10 m distance to the position module. To simplify the test it is converted to be a simulation of the cube firmware, so no flight is needed.	
Success criteria	The success criteria of this test is that the simulated position of the drone can follow a person walking with the position module with a distance to the module between 5-10 meters.	

6.3.3 Validation test

In the validation test, the evaluation will focus solely on validating how the system addresses the problem outlined in the Problem Definition, see Chapter 3, excluding the specific requirements.

Problem definition test

Hypothesis	*How can a track-and-follow system using GNSS data be developed and integrated into an already existing Cube Orange+ autopilot drone system, such that knowledge and familiarity of the Cube Orange+ are obtained?*	Test: A.3&A.4
Purpose	The purpose of this test is to validate if the entire system lives up to the problem definition. Can it solve the problem?	
Success criteria	The success criteria of this test is that the system answers the problem definition by following an object i.e. the drone follows the Position module without failure.	

7 Preparation for system design

This chapter will elaborate on what measures must be taken before the system is ready to be manipulated further. As well as documentation on information about the system based on experience.

7.1 Mission Planner - Ground Control Station

First, it is important to get accustomed to the ground control software - Mission Planner - as this is where most configurations of the Cube Orange+ take place. Mission Planner is specifically developed by the same contributors as ArduPilot, making it the perfect use for Cube Orange+. From within Mission Planner, it is possible to flash the latest firmware from the ArduPilot directory to whatever UAS platform is selected and plugged in.

The purpose of Mission Planner is as its name suggests, to easily plan missions for any autopilot flight controllers, such as the Cube Orange+.

7.2 Preparation of the Cube Orange+

Multiple steps are needed to prepare the Mission Planner setup before the drone is fully operational. This section delves into which initial steps are needed for the drone to be armable and different hacks to implement new code into the already existing ArduCopter code base.

When in Mission Planner, parameters can be changed and written to the drone. This has to be done with some specific parameters to communicate and control the drone. This is done under the "Config/Full Parameter List" tab. Firstly, "LOG_DISARMED" is set to 1 as this allows for all log messages to be readable when the drone is disarmed. This helps in debugging the system. Next, the

"RSSI_TYPE" parameter is set to "ReceiverProtocol" to allow the controller to connect with the Cube Orange+ [29].

Once these steps are done, the compass needs to be calibrated. This is done by having the Cube Orange+ connected to the PC, then in Mission Planner under "Setup", "Mandatory Hardware", and "Compass" start the calibration and follow the instructions. It is important to disable excess compasses as they interfere with each other and prevent arming. After this is done the drone should be able to arm. Arming is done in the "Data" window, under the "Actions" tab where an "Arm/Disarm" button is located.

For communication with the Cube Orange+, the TELEM1 port is used. Therefore, the protocol is changed to MAVLink2 in Mission Planner. The pinout of the TELEM1 port can be seen in table 5.1.

7.2.1 Manipulation of ArduPilot code

After successfully enabling the drone for arming, the next step for obtaining additional functionality involves modifying the ArduPilot code. Firstly, make a new source and header file in the ArduCopter folder, and call these `AP_custom_code.cpp` and .h respectively to adhere to the structure of the rest of the code. The term "custom_code" functions as a stand-in or placeholder representing the example at hand. This class should hold all the functions and variables needed to execute the wanted functionality.

Next, make a source file called `custom_code.cpp` with all lowercase. This source file should use a function that acts as a task from the Copter header file, where the function needs to be defined. For example, the public function `custom_code_init()` is created in `Copter.h`. Further, an object of the **AP_custom_code** class needs to be initialized in the `Copter.h` so that functions from that class can be utilized in `custom_code.cpp`. To do this `AP_custom_code.h` needs to be included in the `Copter.h` file as well as including `Copter.h` in `custom_code.cpp`. Lastly, fill the `custom_code_init()` with the functionality of the **AP_custom_code** class.

After this is done the `custom_code_init()` function can be incorporated in the scheduler and thereby run with the rest of the program.

A simple illustration of this structure can be seen in figure 7.1.

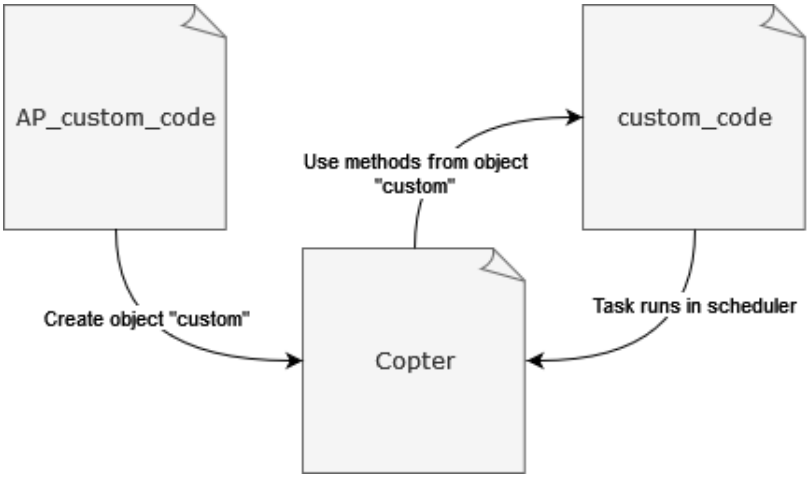


Figure 7.1: Simple illustration of the structure used in ArduCopter.

It is important to adhere to the already existing structure of any open source code as many different developers are manipulating the same code. The normal class naming convention for C and C++ is called PascalCase, this is a convention where names start with an uppercase letter and capitalize the first letter of each subsequent joined word. In the context of ArduPilot, the main folders are named using the PascalCase convention [30]. Whereas the classes within these folders are named by the snake_case convention, where words are written in lowercase and separated by underscores [30].

7.2.2 Incorporating custom code in ArduCopter's scheduler

To incorporate custom code in the scheduler, explained in Section 5.2, a function needs to be called in either `SCHED_TASK()` or `FAST_TASK()`. The first input should be the name of the task used in the `custom_code.cpp` file. From the example before it would be `custom_code_init()`. Afterward, it is important to choose the parameters `_interval_ticks`, `_max_time_micros`, and `_prio` carefully.

To know how much extra time is available when the scheduler table is running, the `_max_time_micros` and `_interval_ticks` are multiplied for each task and added together, for context see the `Copter.cpp` file in repository [27]. This will result in the amount of time available in each second in the worst-case scenario. These calculations are shown in Equation 7.2.2.

$$
\begin{aligned}
\sum(\mathrm{Hz}\cdot\mu\mathrm{s}) = \quad & (250\cdot 130)+(50\cdot 75)+(50\cdot 200)+(10\cdot 120)\\
& +((10\cdot 50)\cdot 3)+(10\cdot 75)+(10\cdot 100)+(50\cdot 100)\\
& +(100\cdot 90)+(3\cdot 75)+(90\cdot 50)+100+(10\cdot 75)\\
& +((10\cdot 50)\cdot 2)+(50\cdot 50)+(100\cdot 75)+(10\cdot 50)\\
& +(400\cdot 180)+(400\cdot 550)+(400\cdot 50)+(0.1\cdot 75)\\
& +(100\cdot 75)\\
& = 401\,282.5\,\mu\mathrm{s}^{1}
\end{aligned}
$$

This leaves a margin of just below 600 000 microseconds of air each second, which makes implementing custom tasks completely doable.

[1] The added maximum allocated time of the base SCHED_TASKs excluding the FAST_TASKs and the ones depending on parameters.

8 System design

In this chapter, the overall design of the track-and-follow system is shaped. From choosing the hardware to designing the software and firmware. It sheds light on the thought process and decision-making involved in shaping the track-and-follow system. The system is split up into different sub-systems. Hence, the structure of this chapter is separated into modules. Firstly, the Position module and the Link module are described under Module design. Then the firmware, that is to be implemented on the Cube Orange+, is documented.

8.1 Module design

Before designing the modules, an overview of the system is shown in 8.1. The figure is based on the modules defined in Chapter 6.

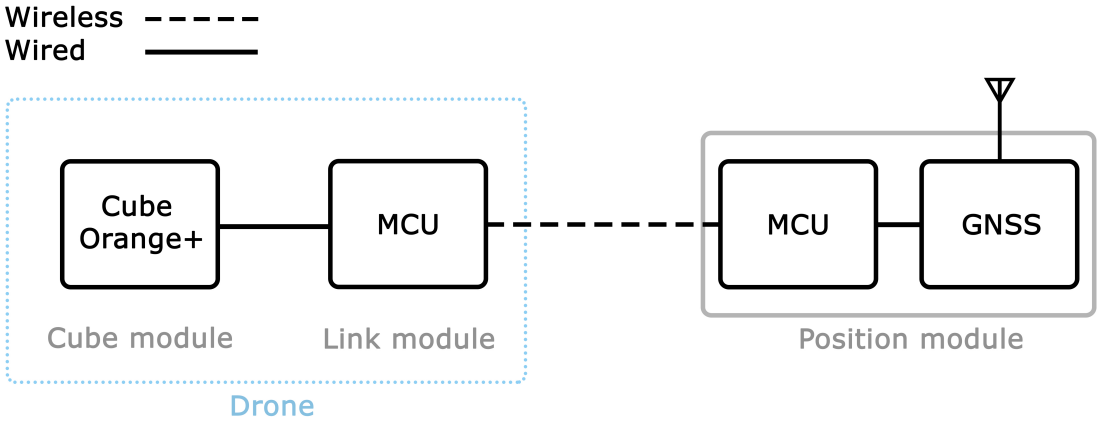


Figure 8.1: A block diagram illustrating different modules and submodules of the track-and-follow system, and how they are connected.

The figure shows the Position module where the GNSS submodule gets the location data and sends it through a wired connection to the MCU. The MCU transmits this data wirelessly to the other MCU mounted on the drone. Lastly, the MCU on the drone sends the location data to the Cube module where the flight controller can use this data to follow the Position module.

8.1.1 Micro-controller

To establish a connection between the two modules, it is important to consider which microcontroller to use. Some microcontroller modules have many ways of communicating wirelessly while others have none.

For this project, the microcontroller ESP32 was made available for testing and use. The MCU ESP32-WROOM-32 is a 32-bit dual-core microcontroller that allows for both WiFi and Bluetooth communication as well as two sets of hardware serial ports [31].

8.1.2 GNSS sub-module

The GNSS module provided for transmitting location data from the position module to the drone is the GY-GPSV3-NEO-7M. It is equipped with the Ublox NEO-7M chipset, offering precise GNSS positioning capabilities [32]. Below are a few of its technical details:

- Horizontal position accuracy : 2.5 $mCEP^{50\%}$ 8.1.2
- Update rate: Maximum 10 Hz
- Communication Protocol: NMEA (default) / UBX Binary
- Serial communication baud rate: 9600
- Working temperature : -40 °C to 85 °C
- Operating voltage: 2,7 V - 5,0 V
- Working Current: 35 mA

One of the important specifications of this GNSS submodule is the horizontal position accuracy that determines the accuracy of the data the drone has to follow. If the incoming data from the submodule varies too much, there is potential for the drone to wander around, and not follow the position module as planned.

Even though the accuracy is high the precision cannot be too low either.

In terms of accuracy and precision, the difference can be described by mean and variance. High accuracy is simply that the mean of all the samples is close to the actual position, whereas high precision is the samples not varying too much from the mean [33]. This is illustrated with Figure 8.2.

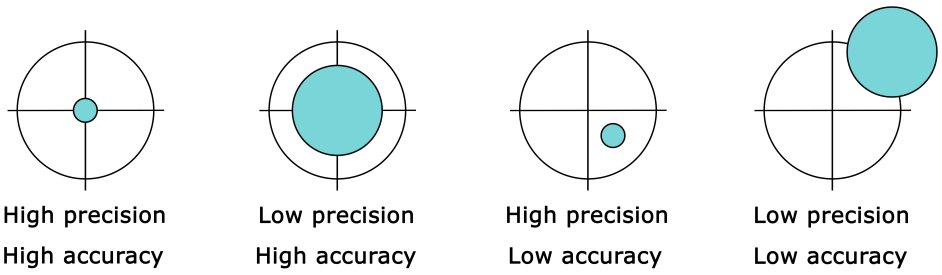


Figure 8.2: Illustration of the difference between precision and accuracy.

Circular error probability

The horizontal position accuracy of the GNSS submodule is specified in meters but with the condition of Circular Error Probability (CEP). It refers to the radius of the circle in which a percentage of the values occur [33]. In this case, the CEP of the NEO-7M is written as $CEP^{50\%}$ which means that 50 % of the GNSS measurements are certainly within a radius of 2.5 m of the actual position. On the other hand, the rest of the data could potentially be much further away. The concept of CEP with different probability percentages is illustrated in Figure 8.3.

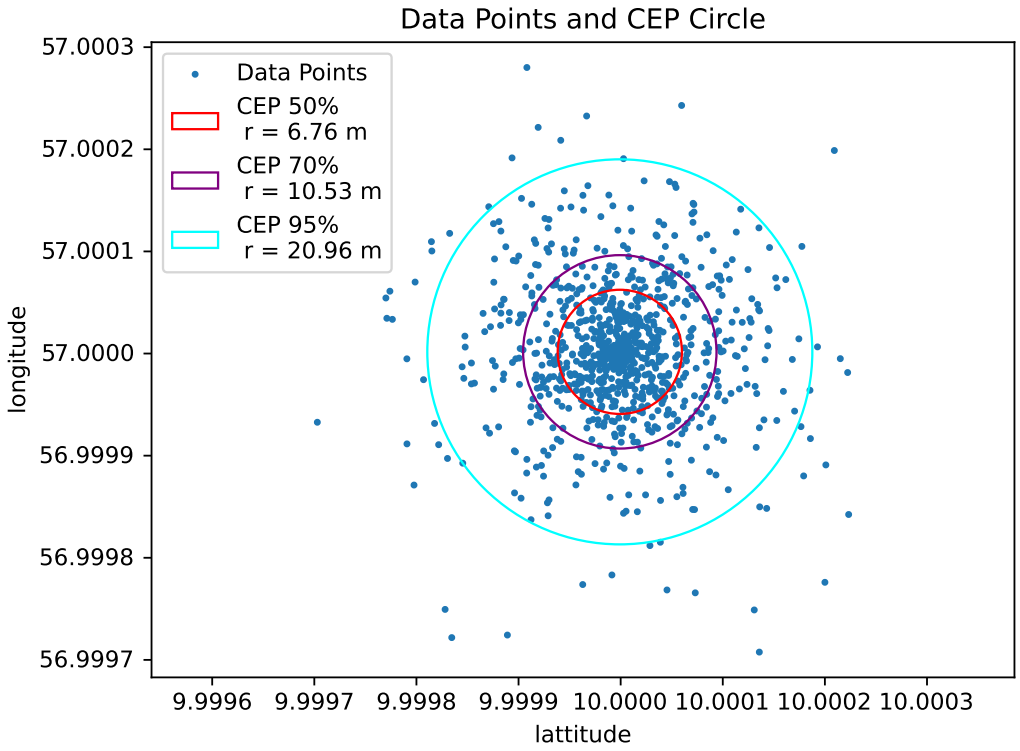


Figure 8.3: A concept figure of CEP with 1000 data points generated with a spread of 0.001 following the normal distribution around the mean coordinate [57.00 , 10.00].

Chipset capabilities:
The Ublox NEO-7M chipset integrated into the module expands its functionality. It supports GLONASS and GPS/QZSS satellite constellations, broadening the scope of satellite signals and enhancing positioning accuracy. The chipset further offers compatibility with various communication interfaces, including UART, USB, SPI, and DDC, providing flexibility for integration into diverse systems [34].

Additionally, the chipset is equipped with an RTC crystal, enabling the transmission of real-time data [34].

Communication Protocol:
The GY-GPSV3-NEO-7M module supports multiple communication protocols, primarily NMEA as default. This protocol plays a crucial role in determining how the module communicates and shares data with other devices or systems.

NMEA, or the National Marine Electronics Association, has defined a standard

for the communication of marine and navigation data between devices. The most commonly used version in the context of GNSS and navigation is NMEA 0183.

NMEA 0183 data is transmitted in the form of sentences, each beginning with a dollar sign character ($), and is terminated by a carriage return and line feed ($\backslash r \backslash n$). Each sentence is a string of ASCII characters containing information such as latitude, longitude, time, and more [35].

Different types of sentences are defined for different data types. The types are defined as $GPGGA, $GPRMC, $GPGLL, $GPGSA, and $GPGSV [35]. Figure 8.4 is an example of what a $GPGGA sentence contains.

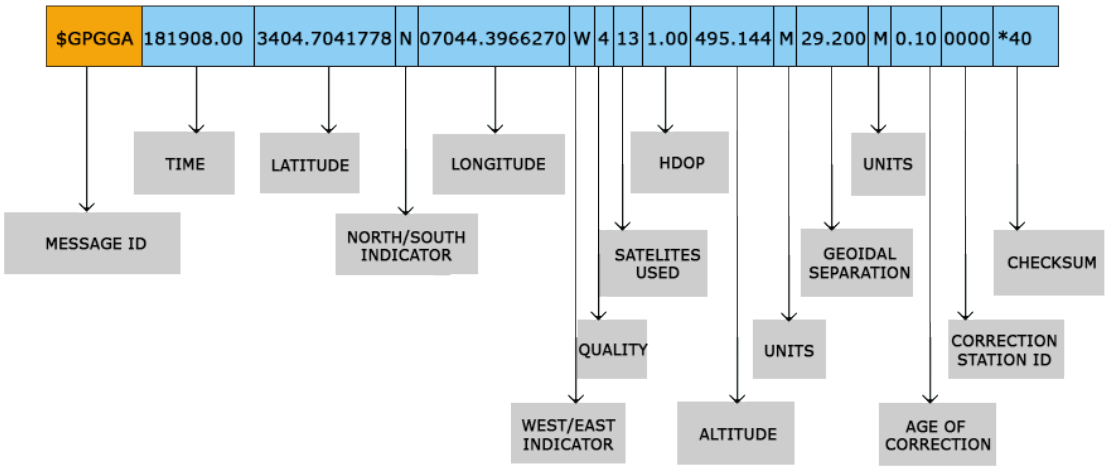


Figure 8.4: The full GPGGA message containing time, location data, amount of satellites it is connected to, and more. [35].

Software for reading location data

The code designed to read the GNSS location data on an ESP32 can be seen in the GitHub repository [36] in the "Position Module" folder. The code is designed to read and process location data from the NEO-7M module using the TinyGPS++ library, and then transmit the latitude, longitude, and altitude information to the link module [37]. The flowchart can be seen in Figure 8.5.

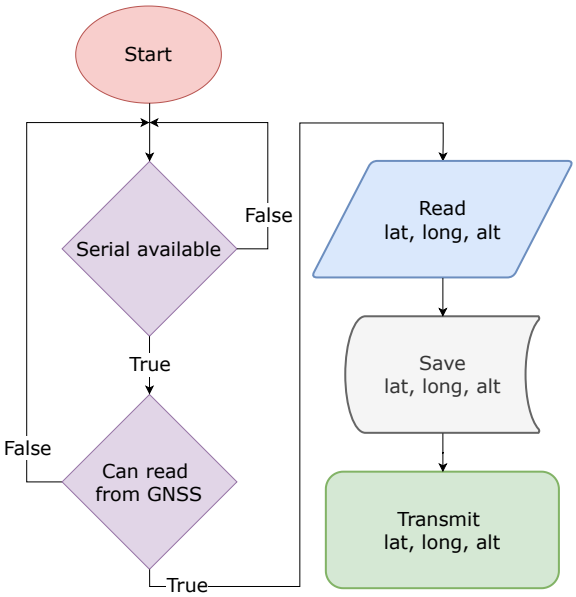


Figure 8.5: Flowchart that describes the structure and functionality of the code running on the ESP32 for Position module. The functionality is to read coordinates and transmit them.

TinyGPS++ is designed to work with GNSS modules that communicate using the NMEA standard sentences. It parses incoming NMEA sentences from the GNSS module and provides functions to extract specific pieces of information such as latitude, longitude, altitude, speed, course, and more [38]. On the ESP32 the library is solely used to parse the latitude, longitude, and altitude. As seen in Listing 8, these are saved into a string that can be transmitted to the Link module.

```
1 sprintf(gnssData, "%.8f,%.8f,%.2f", Loc.Lat, Loc.Long, Loc.Alt);
```

Listing 8: Using the `sprintf()` to save the variables with the given decimals in the char array `gnssData`.

8.1.3 Communication between modules

The ESP32-WROOM-32 microcontroller has two different built-in technologies for radio communication, WiFi and Bluetooth Low Energy (BLE). These technologies are widely used in many IoT solutions. Some of the pros and cons of the two are seen in Table 8.1.

	WiFi	BLE
Max Data Rate	1 - 600 Mbit/s [1]	700 Kbit/s
Max Range	50 - 200 m [39]	80 - 120 m [2]
Power consumption	95 - 240 mA [3]	95 - 130 mA [40]

Table 8.1: Comparison of the wireless communication technologies WiFi and BLE, where Maximum data range, maximum transceiving distance, and power consumption are compared.

In terms of the requirement of the following range of 50 meters in Table 6.2, both technologies support that. In terms of power consumption, the BLE is preferred in a power-scarce system as drones. Additionally, it is worth noting that the ESP32 does not consume more power when it receives a WiFi signal than when it receives a BLE signal. It is only when transmitting using WiFi that the power consumption doubles. This will not have an impact on the Link module but on the position module.

The motivation for using WiFi over BLE is significant because of familiarity with the technology and protocols. Therefore it is chosen to communicate between the modules with WiFi using IEEE 802.11b/g/n protocols.

Point-to-Point

When communicating in a network, a topology is used to arrange the systems on the network. One of the most common ones when connecting two modules is a Point-to-Point topology. In Point-to-Point topology the systems are connected directly via a link, which can be wired or wireless. This direct connection only has the purpose of sending packages back and forth between the systems. The closed communication channel also provides privacy since it is not shared with other systems [43]. The topology is shown in Figure 8.6.

[1] The Maximum data rate of WiFi depends on which IEEE standard is used (802.11b/g/n) [41].
[2] Calculated from The Bluetooth Range Estimator with the ESP32 BLE characteristics [42] [31].
[3] Receiving BLE and WiFi is the same (95 - 100 mA), whereas transmitting WiFi depends on which protocol IEEE 802.11b/g/n (180 - 240 mA) [40].

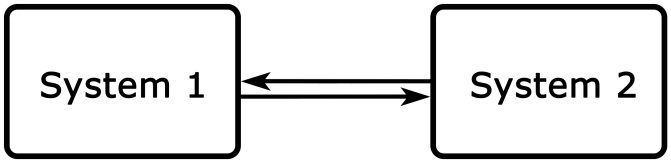


Figure 8.6: Point-to-Point topology in a communication network.

TCP

Because a secure and reliable connection is preferred, the Transmission Control Protocol (TCP) is chosen for the WiFi connection. TCP is a connection-based protocol that ensures reliable data delivery between systems on the network. It directs data flow, error detection, and correction, thereby providing a guaranteed and stable method for transmitting data over WiFi [44]. When sending data via TCP, a three-way handshake must be established for both systems to synchronize and acknowledge the connection. As seen in Figure 8.7, the sender starts by sending an initial synchronization request `SYN`, where the receiving system responds with a synchronization and acknowledge request `SYN ACK`. The sender then acknowledges the `SYN` request from the receiver [45].

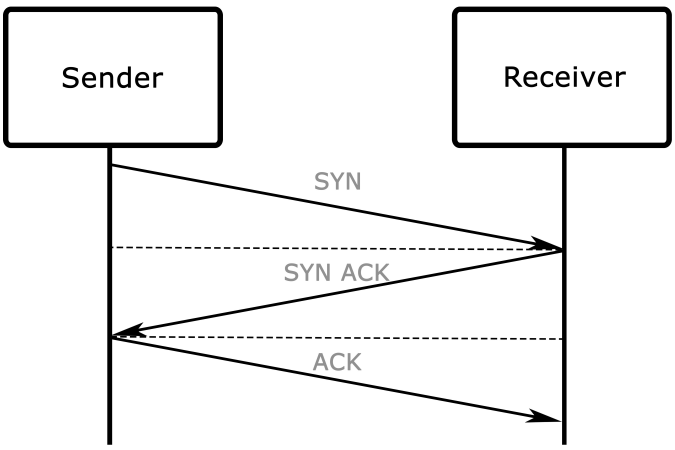


Figure 8.7: The three-way handshake used in TCP to establish a reliable connection.

To connect the two ESP32 via TCP the `WiFi.h` library is used to establish a client connection using the server IP and the server port addresses. Where server IP is a unique address for the device and the server port is the specified port it sends

and receives data from/to. This is seen in the `connectToServer()` function in Listing 9.

```
const char* serverIP = "192.168.1.1";
const int serverPort = 80;

void connectToServer() {
  client.connect(serverIP, serverPort);
  if (client.connected()) {
    Serial.println("Connected to the server.");
  } else {
    Serial.println("Connection to the server failed.");
  }
}
```

Listing 9: The function of connecting the server to the client and vice versa.

The full software of the position module can be found in the GitHub repository [36] in the folder called "Position Module".

Frame intervals

The NEO-7M GNSS submodule has different update rates when receiving signals from different satellite systems. The maximum navigation update rate is 10 Hz for GPS and only 1 Hz for GLONASS [34]. As the maximum update rate for the navigation data is defined by the GNSS submodule, the minimum frame interval can be set to $\frac{1}{10}$ s, as it is ideal to only send information as fast as the GNSS data updates.

Additionally, it is worth noting that the frame interval should match the frequency of the rest of the system. This frequency is to be decided in the scheduler of the ArduPilot code. This will be decided in Section 9.2.

Frame error rate

In an ideal world, all the frames are received by the Link module in their entirety. But in reality, some frames may be lost in the process. To define the maximum allowed Frame Error Rate (FER) the system frequency needs to be taken into account. The faster frames are sent, the less impact one lost frame has, and the slower the system frequency the more critical it is to lose frames. It is expected

that the WiFi connection between the two ESP32s is reliable enough to meet a Maximum FER of 1 %.

Interface with Cube Orange+

After establishing a connection between the position module ESP32 and the link module ESP32 mounted on the drone, it is essential to transmit the gathered information. The final destination is the Cube Orange+ which should receive the location data. As the receiving ESP32, is mounted on the drone chassis, setting up a serial connection between the receiving ESP32 and the Cube Orange+ is a reliable solution. The Cube Orange+ has five general-purpose serial ports as mentioned in Section 5.1. However, accessing the related RX (receiver) and TX (transmit) pins requires some modifications. The interface with the carrier board is shown in figure 8.8.

Figure 8.8: Picture of accessible ports on the Cube Orange+ carrier board. RX and TX are connected through the TELEM 1 port

In the Cube Orange+ package, connectors and cables that fit each port are included. Pin 2 and 3 in the TELEM 1 port are the TX and RX pins respectively [46]. By soldering the wires reversed onto the ESP32's RX and TX pins it is possible to write and read between the two. The TELEM 1 port is tied to the serial 1 port [46]. The regular use for this port is connecting telemetry radio receivers, allowing communication with the GCS (Mission Planner 7.1). Hence the port must

be reconfigured before it can transfer data through UART. In Mission Planner it is therefore necessary to set the protocol on serial 1 to MAVlink2.

MAVLink2 protocol

MAVLink2 is the default protocol used for communication within Cube-based systems. MAVlink2, or Micro Air Vehicle Link is commonly used in different UASs. However, it is strongly associated with the ArduPilot Software base [47]. The purpose of MAVLink2 is to convert information and data into a standardized format that flight controllers like the Cube Orange+ can read. MAVLink2 uses a process known as Marshalling to do this. Marshalling is a computer technology that involves gathering information and serializing it. The term serialization refers to converting complex data structures into a format that can be easily transmitted. In this context, the data is restructured in a specific format compatible with the communication requirements of the Cube Orange+ platform. The full MAVlink2 frame is shown in Figure 8.9.

Figure 8.9: Image of a full MAVLink 2.0 frame, where the payload contains the desired data .

The first byte in a MAVLink2 packet is the start-of-text (STX). This is specific to the MAVLink2 protocol. It ensures that any packets that do not start with the STX byte are automatically rejected [48]. The next byte in the MAVLink2 header describes the length of the message. The following two bytes are flags that signal "incompatibility and compatibility" respectively. The reason for having them in the frame header is to obtain backward compatibility with the former MAVLink protocol (MAVLink 1.0).

The Packet Sequence Number is a counter that increments with each new packet sent. It helps in detecting packet loss, allowing the receiving system to identify missing or duplicated packets. These fields identify the source of the packet. The next byte of the header is the System ID. It represents the unique identifier for the sending system, while the Component ID specifies the particular component on the sending system responsible for generating the message e.g. sensors or cameras. Message ID describes the message type, and occupies 3 bytes in the header [48].

In a MAVLink2 header, the payload represents the actual message or data being transmitted. This can vary in size from anything between 0 and 255 bytes. Lastly,

the checksum is a value computed over the entire packet (including the header and payload). It is used to verify the integrity of the data and detect transmission errors [47]. The receiving system recalculates the checksum and checks it against the received checksum to ensure the data's integrity.

Software for link module

The software on the link module enables multiple purposes. It acts as an access point and establishes a WiFi connection for communication with the position module. It receives pre-processed data from the position module. It encodes the data into an MAVLink2 message and transmits it to the Cube Orange+. To obtain this functionality, code on the ESP32 is made. The code that is running on the ESP32 is to be seen in the GitHub repository [36] in the folder "Link Module".

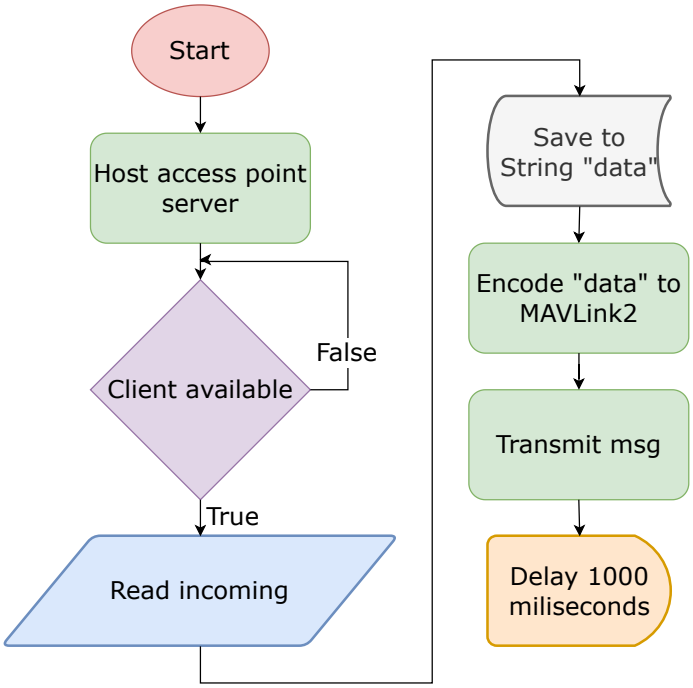


Figure 8.10: Functional flowchart describing the program flow through the code on the Link module. The code hosts a WiFi connection so that data can be transferred. It then receives the data, encodes and forwards it to the Cube Orange+.

The ESP32 initializes the connection by hosting an access point. Once a device connects, it starts reading the incoming data stream from the connected device. The incoming data is saved into a string called `data`.

Before transferring the data to the Cube Orange+, it must be encoded into the MAVLink2 structure. This is done using a MAVLink library for Arduino. The library includes functionalities that take care of the data and insert it into the payload section of the MAVLink2 as described in Section 9. The code that is responsible for doing this is shown in Listing 10.

```
if (client.connected()) {
  if (client.available()) {
    String incoming = client.readStringUntil('\n');
    String data = "$" + incoming + "#" + "\r\n";
    Serial.println("Received : " + data);
    mavlink_message_t msg;
    uint8_t buf[MAVLINK_MAX_PACKET_LEN];

    mavlink_msg_statustext_pack(2, 200, &msg, MAV_SEVERITY_INFO, data.
  c_str());

    uint16_t len = mavlink_msg_to_send_buffer(buf, &msg);

    cubeOrange.write(buf, len);
    delay(5);
  }
}
```

Listing 10: Code that receives a location, encodes it to MAVLink2 and transmits through UART/Serial.

The code running on the Link module which can be seen in Listing 10 utilizes the `mavlink_msg_statustext_pack()` method to pack the received String `data` into a MAVLink2 message. The input parameters are used to determine items in the MAVLink2 frame `SYS_ID` is set to 2, LEN is set to 200, `MSG_ID` is determined to be `MAV_SEVERITY_INFO`, and the final input determines what the `PAYLOAD` contains. The impact of these are described in Section 9. This message gets added to the send buffer by using the `mavlink_msg_to_send_buffer(buf, &msg)`. It takes the buffer and a pointer to the packed message as input parameters. The message is now put into the buffer, char by char, and can be sent through UART/Serial using the `write(buf, len)` function.

8.2 Firmware design

To use the GNSS data from the microcontrollers in the ArduPilot code, additional firmware is written and added to the ArduCopter directory. One class needs to parse the GNSS data from the serial port and save it into usable location variables. The other class must use these location variables to calculate a vector for the drone to follow. The class diagram of these classes is shown in Figure 8.11.

The upper section of the classes shows the variables, shown with a '-' or a '+' in front, symbolizing private ('-') or public ('+') variables. The lower section shows the functions in the class, these are in the same way illustrated as private or public. It is worth noting that the **Location** class is an already existing part of the ArduCopter code, where a few changes are made, hence the inclusion in the class diagram.

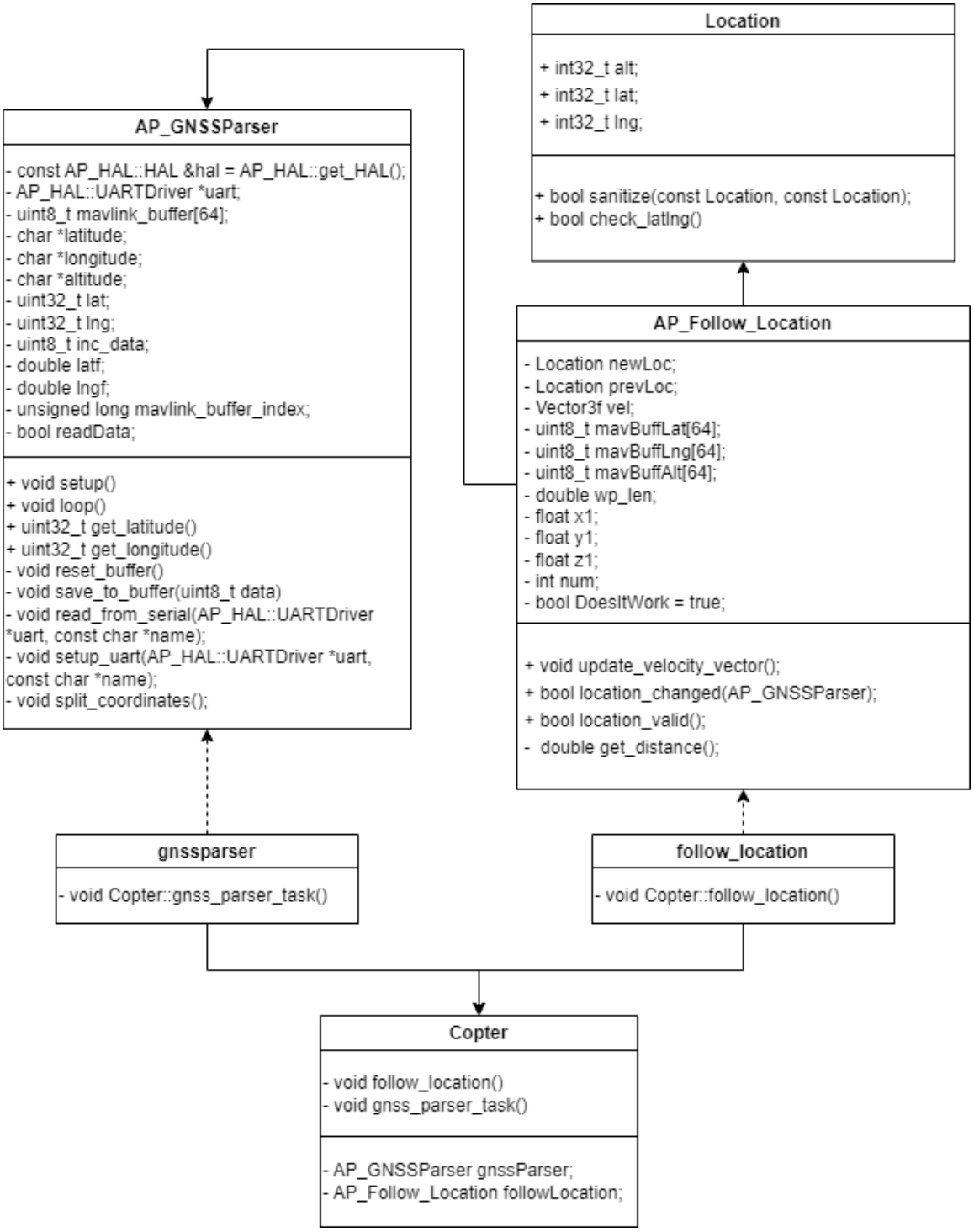


Figure 8.11: Class diagram of the classes GNSSParser and Follow_Location and how they interact with the Location and Copter class.

As highlighted in Section 7.2.1, a class requires an AP source file, an AP header file, and if necessary an extra source file to serve as a task. This section reflects the results obtained by sticking to this approach.

8.2.1 Firmware for handling location data

The **GNSSParser** class contains the functionality to read from the serial port, parse the incoming data to the right format, and save it in further usable variables. This functionality is shown as the flowchart illustrated in Figure 8.12.

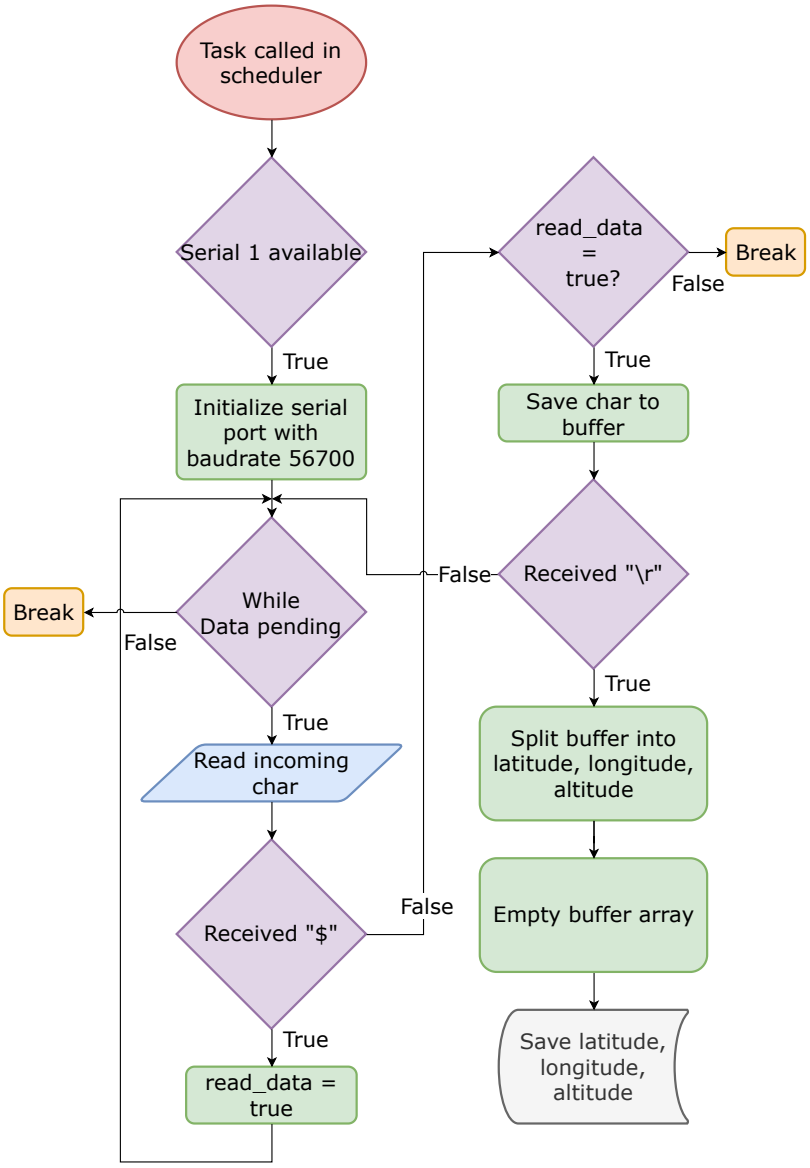


Figure 8.12: Flowchart of the procedure for receiving and parsing MAVLink2 data.

The **AP_GNSSParser** class files are among the files introduced to the ArduPilot code base. The main responsibility of this class is to establish a serial connection between the Cube Orange+ and the link module cf 8.1. As described in Figure 8.10, the link module receives the current location of the position module, encodes it into MAVLink2 protocol, and passes it onto the Cube Orange+. The **AP_GNSSParser** class contains code that reads, decodes, and saves the incoming message so it can later be used in the software.

The process begins with the initialization of serial port 1, configured at a baud rate of 56 700 to ensure accurate data reception by the Cube Orange+. Next, the system checks for incoming bytes on the serial connection. If a byte is detected as '$', the process loops back to read from the port again; otherwise, it terminates.

Upon confirming the presence of a numeric byte, the code checks whether `read_data` is true. If so, the character is stored in a buffer; otherwise, the process breaks. The system then examines for a carriage return ('\r'), signifying the completion of a frame. If detected, the frame is parsed into three distinct values: latitude, longitude, and altitude. Following this, the values are saved as doubles and cast as 32-bit integers for further use. Finally, the buffer array is cleared to accommodate a new frame.

In the following sections, each method used for this code will be described. Firstly, the header file will be demonstrated to give an overview of all the functions and variables used in the class.

AP_GNSSParser header file

This file contains the public and private variables and functions used in the `AP_GNSSParser` source file and is illustrated in Listing 11.

```
#ifndef AP_GNSSParser_H
#define AP_GNSSParser_H

class AP_GNSSParser {
public:
    AP_GNSSParser();
    void setup();
    void loop();
    uint32_t get_latitude();
    uint32_t get_longitude();
private:
    void reset_buffer();
    void save_to_buffer(uint8_t data);
    void read_from_serial(AP_HAL::UARTDriver *uart, const char *name);
    void setup_uart(AP_HAL::UARTDriver *uart, const char *name);
    void split_coordinates();
    const AP_HAL::HAL &hal = AP_HAL::get_HAL();

    AP_HAL::UARTDriver *uart;
    uint8_t mavlink_buffer[64];
    char *latitude;
```

```
22      char *longitude;
23      char *altitude;
24      uint32_t lat;
25      uint32_t lng;
26      uint8_t inc_data;
27      double latf;
28      double lngf;
29      unsigned long mavlink_buffer_index;
30      bool readData;
31  };
32
33  #endif // AP_GNSSParser_H
```

Listing 11: The AP_GNSSParser header file.

All methods and variables in the header file are used in the code to obtain the desired functionality described in Flowchart 8.12.

`setup_uart()`

This function initializes serial port 1, using the Hardware Abstraction Layer (HAL) driver. It has two parameter inputs **uart_param* and **name*. These are the port from the HAL driver and the name of the port respectively.

The code then checks if the assigned UART pointer is `nullptr` (null), indicating a failure to find a valid serial connection. In such a case, it prints an error message using the HAL's console and exits the method. If a valid UART pointer is found, the method proceeds to initialize the UART communication with a baud rate of 57600 using the `begin` method of the UART driver.

This method, illustrated in Listing 12, is called in function `setup()`, used in the `GNSSParser` task to initialize the UART within the scheduler. This is illustrated in Figure 8.11.

```
#include <AP_HAL/AP_HAL.h>

void AP_GNSSParser::setup_uart(AP_HAL::UARTDriver *uart_param, cons+-t
    char *name){
     this->uart = uart_param;
     if (uart == nullptr)
     {
         hal.console->println("Failed to find serial");
         return;
     }
     uart->begin(57600);
}
```

Listing 12: The setup_uart function in AP_GNSSParser class.

`read_from_serial()`

This function takes a pointer to an `AP_HAL::UARTDriver` (*uart_param*) and a name as parameters. The first condition in this method is identical to the one in `setup_uart`. This ensures that the code will not enter the following while loop if there is no connection. The condition of the while loop becomes true for as long as available data are waiting in the UART/Serial buffer.

The purpose of this method is to extract the location data from the UART/Serial buffer as described in the Flowchart 8.12. This method is represented as the while condition in the flowchart.

```
void AP_GNSSParser::read_from_serial(AP_HAL::UARTDriver *uart_param,
    const char *name){
    if (uart_param == nullptr){
        hal.console->println("Failed to find serial");
        return;
    }else{
        while (uart_param->available()){
            inc_data = uart_param->read();
            if (inc_data == '$'){
                readData = true;
            }else if (readData){
                save_to_buffer(inc_data);
                if (inc_data == '\r'){
                    split_coordinates();
                    readData = false;
}   }   }   }   }
```

Listing 13: The read_from_serial function in AP_GNSSParser class.

`save_to_buffer()`

The **AP_GNSSParser** class contains a method named `save_to_buffer`, designed to store received characters in a buffer array. The function takes an 8-bit unsigned integer as its parameter. The first condition ensures that the buffer does not overflow. If true, the function assigns the incoming data to the buffer at the current index, increments the index, and adds a null terminator ('\ 0') at the new index.

In short, this function appends a byte to the buffer if there is sufficient space, preventing buffer overflow by ensuring the index stays within the buffer's bounds. The code can be seen in Listing 14.

```
void AP_GNSSParser::save_to_buffer(uint8_t data){
    if (mavlink_buffer_index < sizeof(mavlink_buffer) - 1){
        mavlink_buffer[mavlink_buffer_index] = data;
        mavlink_buffer_index++;
        mavlink_buffer[mavlink_buffer_index] = '\0';
    }
}
```

Listing 14: The save_to_buffer function in **AP_GNSSParser** class.

reset_buffer and split_coordinates

As earlier mentioned, when transmitting the data, the last byte of the message is the return carriage $\backslash r$. Once a return carriage is detected inside the `read_from_serial()` method, the `split_coordinates()` method is called. The purpose here is to split the saved buffer from Listing 14 into latitude, longitude, and altitude. For this job, a pre-defined cpp function `strtok` is used. The cpp function separates the array of chars depending on what delimiter is specified. In the `split_coordinates()` method the delimiter is a comma. What it does in Listing 15 is that it saves all chars until the first comma into the variable `latitiude`, in the following lines it saves whatever is remaining until a new comma appears into the variable `latitude`, this repeats until all three variables have been saved.

```
1  void AP_GNSSParser::split_coordinates(){
2      latitude = strtok((char *)mavlink_buffer, ",");
3      longitude = strtok(NULL, ",");
4      altitude = strtok(NULL, ",");
5      reset_buffer();
6  }
7
8  void AP_GNSSParser::reset_buffer(){
9      mavlink_buffer[0] = '\0';
10     mavlink_buffer_index = 0;
11 }
```

Listing 15: The `split_coordinates` and `reset_buffer` functions in **AP_GNSSParser** class.

get_latitude and get_longitude

The latitude and longitude have separate getters. In the function shown in Listing 16 the variable `"latitude"` is, with the use of the `"atof"` method, converted to a floating-point number. It is also multiplied by 10^7 to enlarge the number, ensuring that when converted to a 32-bit integer, no information is lost. This is done so the variables can be used in the **Location** class, where the latitude, longitude, and altitude all are stored as 32-bit integers, cf Subsection 5.

The `get_longitude` function is identical, only with the longitude variable instead of the latitude.

```
uint32_t AP_GNSSParser::get_latitude(){
    latf = atof(latitude) * 10000000.0;
    lat = static_cast<int32_t>(latf);
    return lat;
}
```

Listing 16: The `get_latitude` function in **AP_GNSSParser** class.

8.2.2 Firmware for using the location data

The **Follow_Location** class takes the desired location from the **GNSSParser** class and flies towards that location, behaving as a follow mode. The functionality of this class is shown with a flowchart in Figure 8.13.

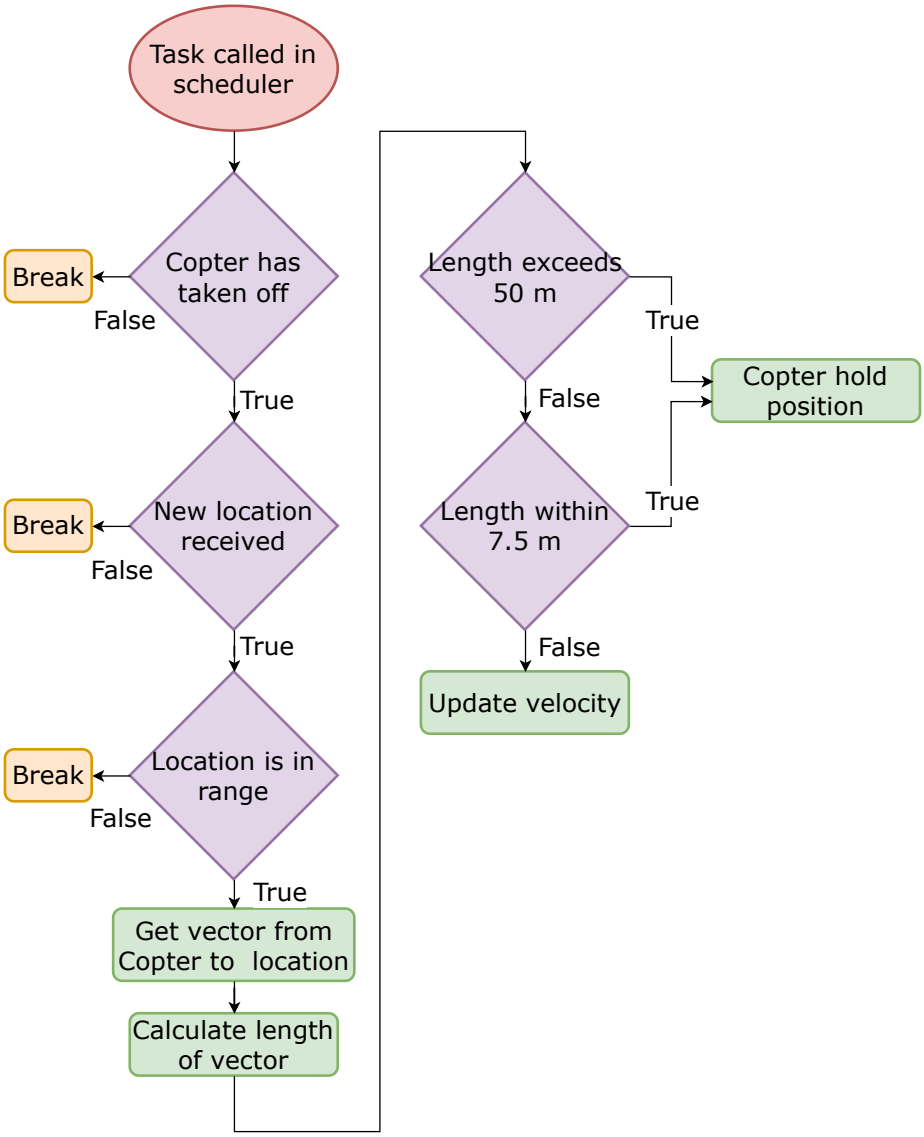


Figure 8.13: Flowchart showing the functionality of the **Follow_Location** class.

Once the code has been called in the scheduler it checks if the drone has taken off, received a location, and if the location is in range. This ensures that a velocity reference is not supplied until the drone is in the air and the location has been verified.

When the drone is in the air and ready to follow, it receives the new desired location from the object of the **GNSSParser** class, gets the distance from the current location to the new one, and saves it in a distance vector. The length of this dis-

tance vector is then calculated. If the length of this vector exceeds 50 m the drone is set to hover in its position due to regulations in Section 2.2. If the length of the vector is somewhere between 7.5 m and 50 m, it updates the velocity reference with a vector, based on the distance vector. If the drone finds its position closer than 7.5 m to the target it hovers in place.

AP_Follow_Location header file

The header file of the **AP_Follow_Location** class contains the definition of all the functions and variables used in the class, and if they are private to the class only or public for other classes to use. The file is shown in Listing 17.

```
#ifndef AP_Follow_Location_H
#define AP_Follow_Location_H

class AP_Follow_Location{
    friend class Copter;
public:
    AP_Follow_Location();
    void update_velocity_vector();
    bool location_changed(AP_GNSSParser gnssParser);
    bool location_valid();
private:
    double get_distance();
    Location newLoc;
    Location prevLoc;
    Vector3f vel;
    uint8_t mavBuffLat[64];
    uint8_t mavBuffLng[64];
    uint8_t mavBuffAlt[64];
    double wp_len;
    float x1;
    float y1;
    float z1;
    int num;
    bool DoesItWork = true;
};
#endif // AP_Follow_Location_H
```

Listing 17: The `AP_Follow_Location` header file.

The header file describes all methods and variables that are used in the `AP_Follow_Location.cpp` the following sections describe how they are all used

to obtain the exact functionality that is described in Flowchart 8.13. Causing the drone to follow the GNSS coordinate that is parsed down from the **GNSSParser** class.

`location_changed()`

The `location_changed()` is a method returning a boolean that takes one input of the `AP_GNSSParser` class. The function returns `True` if the latitude and longitude have been obtained from the **GNSSParser** class and stored in the **Location** class instance `newLoc`. This function is the second decision in the Flowchart in Figure 8.13, and is seen in Listing 18.

```
bool AP_Follow_Location::location_changed(AP_GNSSParser _gnssParser){
      prevLoc.lat = newLoc.lat;
      prevLoc.lng = newLoc.lng;
      newLoc.lat = _gnssParser.get_latitude();
      newLoc.lng = _gnssParser.get_longitude();
      return true;
}
```

Listing 18: The `location_changed` function in the **AP_Follow_Location** class.

`location_valid()`

The other boolean function `location_valid()` uses functions defined in the **Location** class, cf subsection 5, to verify and check if the values stored in `newLoc` are usable. One of the methods utilized is `sanitize()`. It makes sure to change the latitude and longitude in case the values become 0 or are out of scope.

When testing the GNSS submodule, it was observed that once in a while a location was logged, which was more than 1000 kilometers away. To sort out these obvious errors, a modification to the `sanitize()` function was added. It checks if the coordinates are within a certain range from the last coordinate. This function was added in the **Location** class and is shown in Listing 19.

```
bool Location::is_out_of_range(int32_t curr_coord, int32_t prev_coord,
   int dist){
    int32_t delta = abs(curr_coord - prev_coord);
    return (delta > dist);}
```

Listing 19: A function added in the `Location.cpp` library for the `sanitize()` function to validate if the location point is in range.

This function is the third decision in the flowchart Figure 8.13, and returns either `True` or `False`.

`get_distance()`

The double function `get_distance()` uses an instance of the **Copter** class to get the drone's current location and measures the distance to the location `newLoc` with the function `get_distance_NE(newLoc)`. This is done separately for the x and y axis of the distance vector. The difference in altitude can be found using the function `get_alt_cm(newLoc.get_alt_frame(), newLoc.alt);`
The length of this distance vector is then calculated and returned as a double. This functionality is shown in the Flowchart in figure 8.13 as the green boxes at the bottom and is shown in Listing 21.

```
1 double AP_Follow_Location::get_distance(){
2    x1 = copter.current_loc.get_distance_NE(newLoc).x;
3    y1 = copter.current_loc.get_distance_NE(newLoc).y;
4    z1 = copter.current_loc.get_alt_cm(newLoc.get_alt_frame(), newLoc.alt
     );
5    return sqrt(x1 * x1 + y1 * y1);
6 }
```

Listing 20: The `get_distance` function in the **AP_Follow_Location** class.

`update_velocity_vector()`

The update velocity vector functionality is everything to the right in the Flowchart in Figure 8.13. If the drone is more than 50 meters from the desired location it goes into `POSHOLD` mode which holds the position. Else the velocity vector `vel` is set to equal the distance vector `(x1, y1)` divided by the length of the vector `dist_len`. This is shown in Listing 21.

```
void AP_Follow_Location::update_velocity_vector(){
    ...
    else if (dist_len > PERIPHERY){
        vel.x = (x1 * 100 / dist_len) * KP;
        vel.y = (y1 * 100 / dist_len) * KP;
        vel.z = 0;
        }
    ...
  copter.mode_guided.set_velocity(vel);
}
```

Listing 21: The `update_velocity_vector` function in the **AP_Follow_Location** class.

The coordinates in the velocity are scaled by 100 to convert the units from meters to centimeters. The z-coordinate is set to 0 to maintain the fixed take-off altitude. If the drone reaches the desired periphery of 7.5 meters from the target, the velocity vector is set to the null vector, by setting all coordinates equal to zero. This makes the drone hover in place. Lastly, the velocity is set using the velocity reference function from mode guided.

9 Full system integration

This section offers an exploration of the integration process for key system components. It outlines the incorporation of essential elements necessary for the operation of the entire system.

9.1 Firmware integration

As mentioned in Section 7.2.1, both classes **AP_GNSSParser** and **AP_Follow_Location** need a task to run in the scheduler. They are created in respective files to keep them separate from the `Copter.cpp` file. These tasks use an object, which is an instance of the respective class created in `Copter.h`, used to call the functions defined in the AP class files. The `gnss_parser` task file is shown in Listing 22.

```
#include "Copter.h"

void Copter::gnss_parser_task(){
    gnssParser.setup();
    gnssParser.loop();
}
```

Listing 22: The `gnssparser.cpp` task file that runs the setup and loop functions from the **AP_GNSSParser** class.

A similar method is applied to the `follow_location` task file, which is shown in the Listing 23.

```
#include "Copter.h"

void Copter::follow_location(){
    static bool setupDone = false;

    if (flightmode->is_taking_off()){
        if (!setupDone){
            setupDone = true;
            return;
    }
}
    if (setupDone){
        if ((!flightmode->is_taking_off()) && followLocation.
   location_changed(gpsParser)){
            if (followLocation.location_valid()){
                followLocation.update_velocity_vector();
                return;
            } return;
        } return;
}   }
```

Listing 23: The `follow_location.cpp` task file that runs the `location_changed`, `location_valid`, and `update_velocity_vector` functions from **AP_Follow_Location** class.

Both of these tasks are then to be called in the scheduler, but before calling them, the other parameters of the `SCHED_TASK()` functions have to be set.

9.2 Allocated task time

To know how much time is needed for each task, a small test is executed where the allocated time is set higher than needed, and the current time is logged at the beginning and the end of both tasks. To track the current time accurately, a function is defined in both classes that use the C++ library `chrono` to track the time. This function, defined in the **AP_GNSSParser** class, is shown in Listing 24.

```
#include <chrono>

void AP_GNSSParser::print_time(char *str){
    using std::chrono::high_resolution_clock;
    auto now = high_resolution_clock::now().time_since_epoch();
    auto now_in_us = std::chrono::duration_cast<std::chrono::microseconds
    >(now).count();
    hal.console->printf("%s: %ld\n", str, now_in_us);
}
```

Listing 24: The `print_time` function created as a member of the **AP_GNSSParser** class.

Subtracting the start and the end times will result in the amount of time used for each task. This is seen in Equation 9.1.

$$\begin{aligned} &\texttt{gnss\_parser\_task()} \\ \text{Start\_time} &= 9\,347\,760\,\mu s \\ \text{End\_time} &= 9\,347\,801\,\mu s \\ \text{Time\_used} &= 9\,347\,801 - 9\,347\,760 = 41\,\mu s \end{aligned} \tag{9.1}$$

$$\begin{aligned} &\texttt{follow\_location()} \\ \text{Start\_time} &= 10\,863\,331\,\mu s \\ \text{End\_time} &= 10\,863\,336\,\mu s \\ \text{Time\_used} &= 10\,863\,336 - 10\,863\,331 = 5\,\mu s \end{aligned} \tag{9.2}$$

It is seen from the calculations above that both tasks use less than 50 µs. To make sure that the task is not limited by accident the `gnssParser` task is set to use a maximum of 1000 µs, and the `follow_location` is to 500 µs. It is known from the calculations in Section 7.2.1 that the scheduler has plenty of time for new tasks to be implemented. This makes the priority of the task less relevant and is therefore set to the lowest in the scheduler. This is also done to not mess with the flow of the already working ArduCopter system.

9.3 System frequency

One can argue that the drone has some inertia that makes changing speed and direction take time. Updating the desired location as sparingly as once every 10 seconds would probably give a visual result of a slow or even lagging follow

mode, that would either under- or overshoot. But as the update rate for the desired location rises there will be a point where the inertia of the drone is not settled before trying to update the location again. It is believed that the following mode will act smoother in this state. If, for example, the frequency is 10 Hz it is hard to imagine the drone can change its direction 10 times a second. Increasing the frequency vastly would not make a difference in terms of performance since the drone has physical limitations in terms of direction changes, acceleration, and deceleration. So there is a balance between using as little excess time in the scheduler as possible and updating fast enough to fly smoothly. If a test is conducted where the system frequency is adjusted from low to high, it would not be difficult to spot the difference and decide the optimal frequency for a smooth flight.

Without executing this test, it is decided to run the tasks with a frequency of 5 Hz, aiming for a smooth state for the follow mode. This is then implemented in the scheduler seen in Listing 25.

```
1 SCHED_TASK(gnss_parser_task,      5,     1000,    174),
2 SCHED_TASK(follow_location,       5,     500,     179),
```

Listing 25: The two tasks implemented in the `Copter.cpp` scheduler.

9.4 Hardware integration

To mount the Cube Orange+, the Link module, and the different telemetry antennas, the hardware has to be securely mounted to the drone for safe use. In the top and bottom plates of the drone, there are existing mounting holes and extra screws for mounting a flight controller and external hardware. This, unfortunately, does not line up with the mounting holes of the Cube Orange+ carrier board. Therefore a 3D print is printed to fit the mounting holes of the drone. The 3D printed mount for the Cube Orange+ is shown in Figure 9.1.

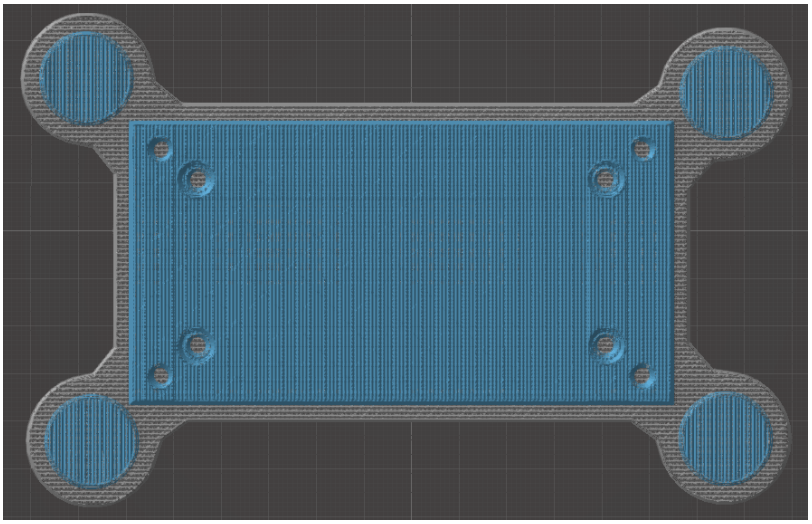


Figure 9.1: Top view of the Cube mount 3D print file.

As suggested in Section 5.1 mounting the Link module to the bottom of the platform board, another 3D print is created to encase and protect the ESP32. This 3D-printed case is then stuck onto the platform with sticky foam that arrived in the Cube Orange+ box. The case for the ESP32 has the dimensions 36.2 mm x 28 mm x 51.5 mm and is shown in Figure 9.2.

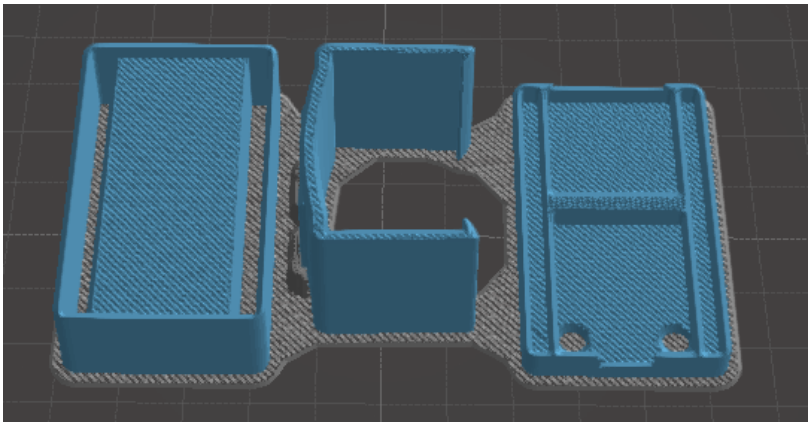


Figure 9.2: Print preview of the ESP32 case for mounting and protecting the Link module.

Sticky foam is used to mount the HolyBro USB telemetry to the drone. It is stuck onto the left side of the Cube Orange+ on the top plate of the drone, with the adjustable antenna pointing up in the air.

The RC receiver is not as robust as the HolyBro telemetry module, so yet another

3D print is created to mount and secure it in place. The print is designed so the antenna is pointed up in the air at a 45-degree angle from the top plate of the drone. The 3D print is then bolted to the bottom of the top plate. The print is seen in Figure 9.3.

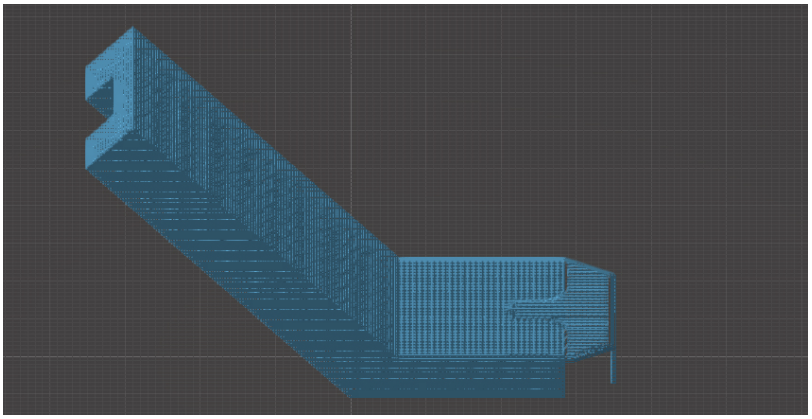


Figure 9.3: Print preview of the RC receiver case for mounting and protecting the antenna.

All external hardware is mounted and secured to the drone kit which can be seen in Figure 9.4. The full product includes the drone with the Link module attached and weighs 1564.5 g. The Position module is by itself.

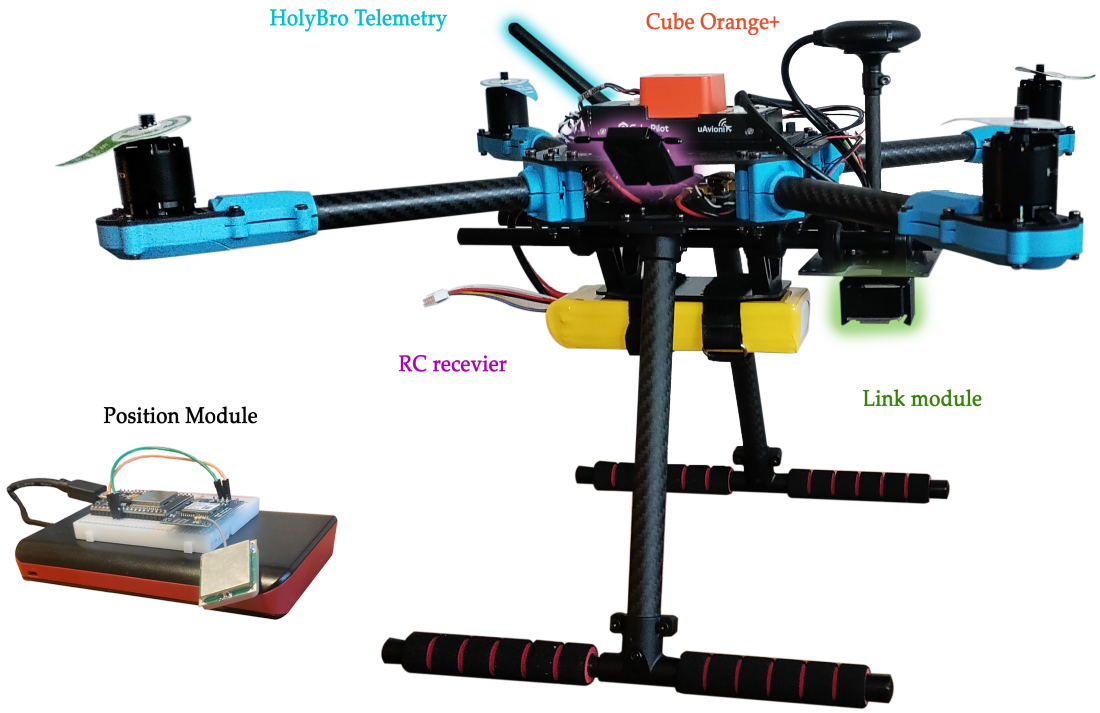


Figure 9.4: A photo of the full system including the Link Module (green glow), the telemetry antenna (cyan glow), the RC receiver (purple glow), and the Position Module.

Link module

The Link module is attached to the drone under its GNSS module as seen in Figure 9.4. It is mounted with a custom 3D-printed part. The ESP32 from this module is connected via its RX and TX pins to the Cube's carrier board on the TELEM 1 port. The TELEM 1 port provides, other than RX and TX pins, a 5 V outlet that the ESP32 is powered from. The connection between the ESP32 and the carrier board can be seen in Figure 9.5.

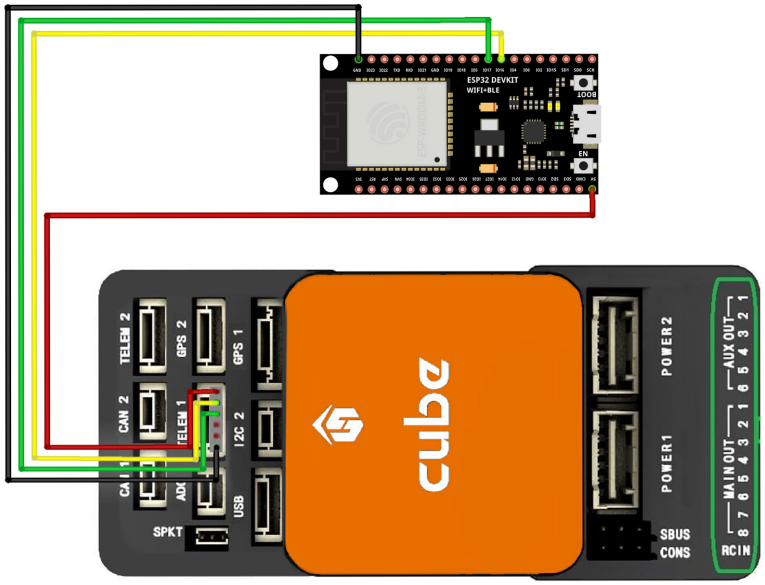


Figure 9.5: The hardware integration of the ESP32 DEVKIT connected to the TELEM 1 connector on the Cube Orange+ carrier board.

Position module

The Position module is a stand-alone module (not connected via wires to the drone) and can be seen in Figure 9.4 on the right. This ESP32 is connected to the NEO-7M GNSS module and is responsible for transmitting GNSS data to the Link module on the drone. To power the Position module the ESP32 is attached to a power bank.

The GNSS module is connected to the RX and TX pins on the ESP32, as shown in Figure 9.6.

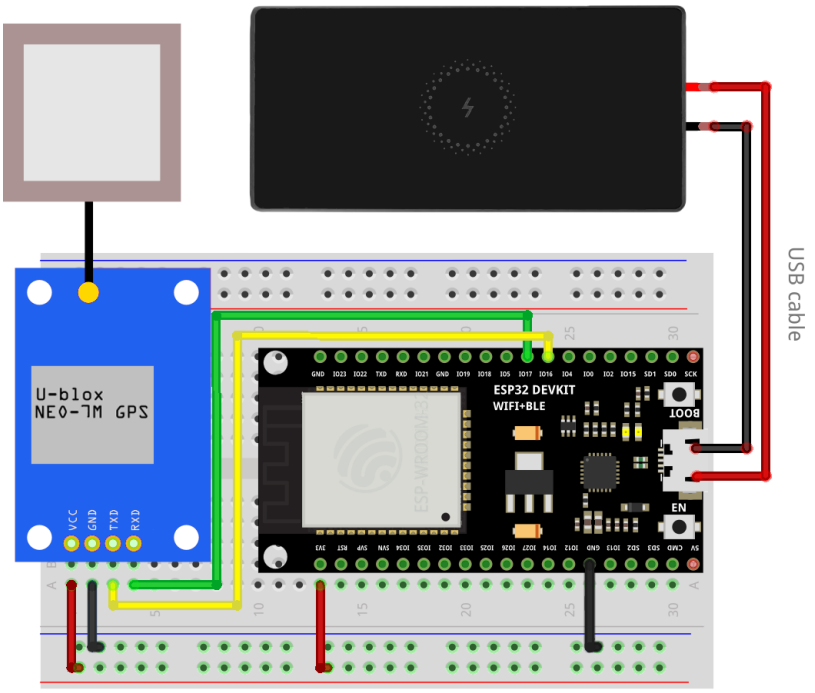


Figure 9.6: The hardware integration of the NEO-7M connected to an ESP32 DEVKIT with a power bank as a power supply.

10 System tests

In this chapter, the different system test is executed. It is decided only to do some of the tests, which are deemed to give a sufficient understanding of the process of testing with different purposes. The tests chosen are listed below:

- Unit test - Position module 6.3.1
- Verification test - Transmission range and FER 6.3.2
- Validation test - Drone following location data 6.3.3
- Validation test - Storing data in the Cube 10.4

10.1 Unit test - Position module

To demonstrate the use of a unit test, one is constructed to measure the horizontal position accuracy of the GNSS submodule of the Position module. The test specification is mentioned in Section 6.3.1.

The purpose of this test is to measure the horizontal position accuracy of the NEO-7M GNSS module and calculate the CEP. According to the requirement, it should be no greater than 2.5 meters, as stated in the Ublox NEO 7M datasheet[34]. A unit test is a small, isolated test that checks if a specific part of a program (usually a function or method) behaves as expected for a given input. But in this case, the reliability of the datasheet is tested. The full test journal can be seen in Appendix A.1.

10.1.1 Test setup

To test the horizontal position accuracy of the NEO-7M GNSS submodule, it is wired up to an ESP32 microcontroller which is connected to a laptop, where

the latitude and longitude are logged. To ensure a stable position, the test is conducted for a minimum of 15 minutes, because this is believed to be sufficient to get an adequate result. The test is conducted outdoors for more similar results to the use case. The setup is shown in figure 10.1.

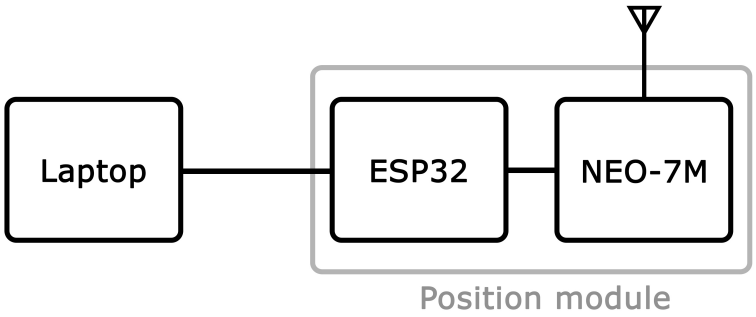


Figure 10.1: Test setup for the unit test of the GNSS submodule where it is connected to an MCU which is connected to a laptop for storing data.

10.1.2 Success criteria

The success criteria of this test is that the measured CEP 50% is 2.5 meters or less.

10.1.3 Results

The 4566 data points, found in GitHub repository [36] folder "Python scripts and data\ CEP test" file "`gnss_data_test.csv`", were collected over 15 minutes, in an open outdoor space. The data points are fed into the Python script, found in the same folder with the file name "`unit-test_cep.py`". The script plots the data and uses the Haversine equation to calculate the radius of the different CEPs. The Haversine equation accurately computes the distance between two points on the surface of a sphere [49]. The result of the Python script is shown in figure 10.2.

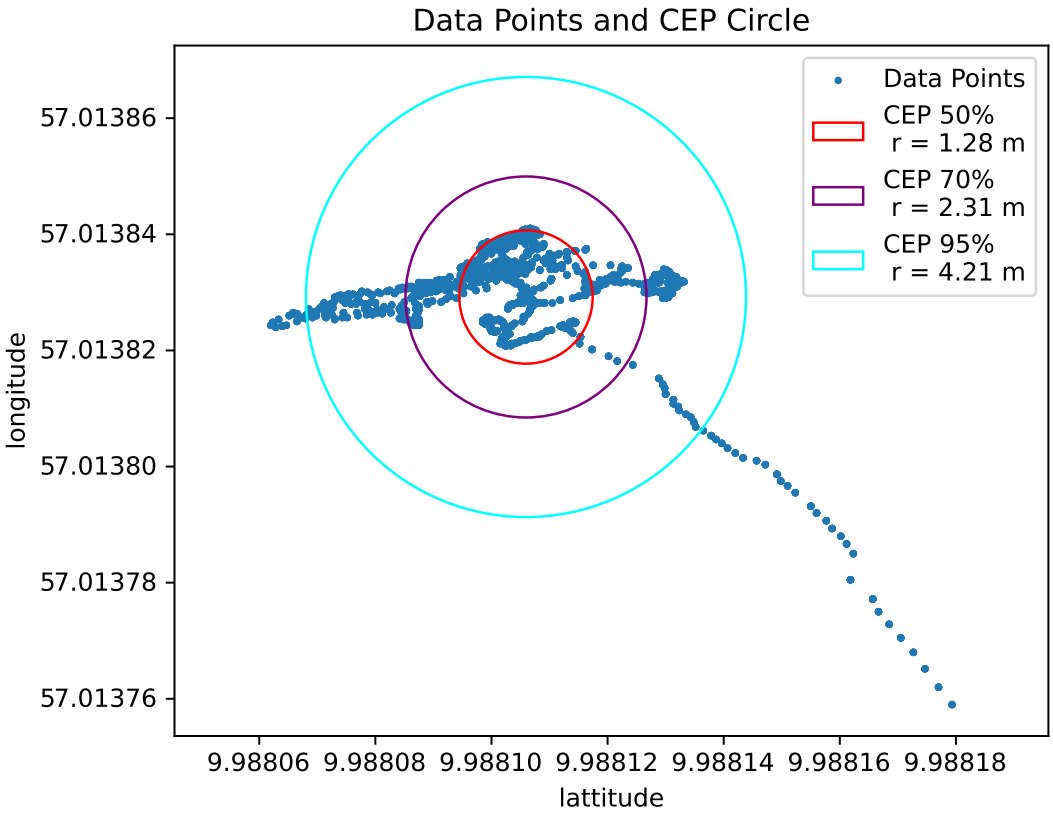


Figure 10.2: 4566 GNSS samples were collected over 15 minutes and plotted together with three different CEPs. Lattitude on the x-axis, Longitude on the y-axis.

10.1.4 Test conclusion

It can be observed in Figure 10.2, that the CEP 50% is 1.28 meters, and even the CEP 70% is 2.31 meters. Thereby the success criteria of the test is obtained.

10.2 Verification test - Transmission range and FER

A verification test reveals if a part of the system fulfills the requirements. Such a test is constructed to ensure that the communication between the Position and Link modules fulfills the requirements. The test specification is mentioned in Section 6.3.2.

The purpose of this test is to measure the Frame Error Rate (FER) of the communication between the Position module and the Link module. This is done at 50 meters to also test the minimum required data transmission range of the technical requirement 2.9.

10.2.1 Test setup

Each ESP32 is connected to its respective laptop and placed 50 meters apart without obstacles. 100 000 frames are sent from one ESP32 to the other. All sent and received are saved and reviewed. The test setup is shown in figure 10.3

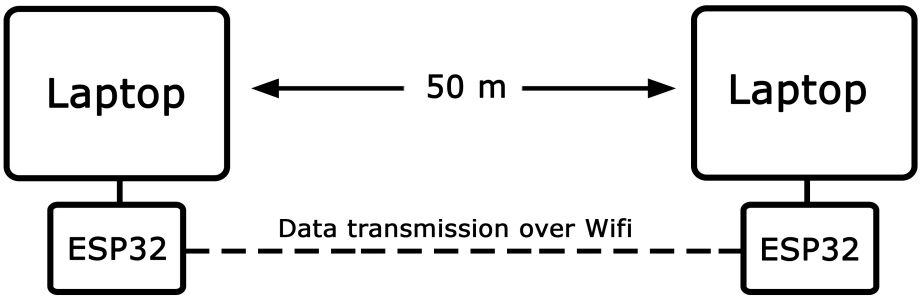


Figure 10.3: Test setup for the FER test, where two ESP32 are placed 50 meters apart, both connected to a laptop to log the data sent and data received.

10.2.2 Success criteria

The success criteria of this test is that the calculated FER does not exceed 1%.

10.2.3 Results

The result of the test can be seen in listing 26.

```
Running file: FER_Script.py
Total Frames: 100227
Number of frames that do not match in the received file: 144
Frame Error Rate (FER): 0.14%

Process finished with exit code 0
```

Listing 26: Result of Python script.

10.2.4 Test conclusion

The test yielded a FER of 0.14 %, comfortably meeting the success criteria.

10.3 Validation test - Drone following location data

The validation test is split up into two tests. One test focuses on the transmission of the GNSS data, to validate that the data arrives successfully from the Position module and is parsed and stored correctly in the Cube Orange+. The other test is a simulation of the **Follow_Location** class to validate if the drone would follow the GNSS data points sent by the Position module.
This section describes a test setup to assess the drone's adherence to specific coordinates through simulation. The process involves generating a route, simulating the movement, and visually inspecting the results. The test journal for this test can be found in appendix A.3.

10.3.1 Test setup

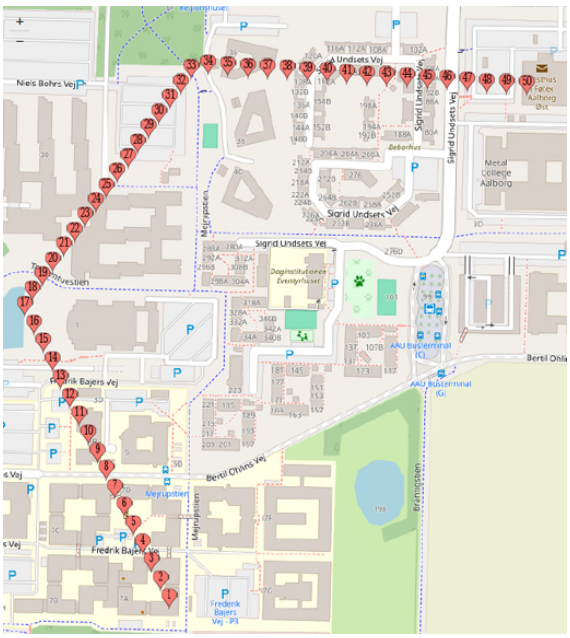
To validate the drone's ability to adhere to specific coordinates, a simulation is conducted using Python and XLaunch. The test involves generating a route with 50 coordinates and integrating it with the Cube Orange+ through a loop. The code used to perform this test can be found in the `AP_Follow_Location.cpp` file within the "CodeForSimulation" branch of the GitHub repository [50].

A C++ code is crafted to generate 50 coordinates, including four predefined via points, thereby marking a particular route. This code can be found in the GitHub repository [36] folder "Python scripts and data\Validation test follow\route_generator" and is run separately. These coordinates are then employed in the simulation with the Cube Orange+, utilizing an if loop to replicate the position module moving. The test code can be found in the GitHub repository [50] branch "CodeForSimulation".

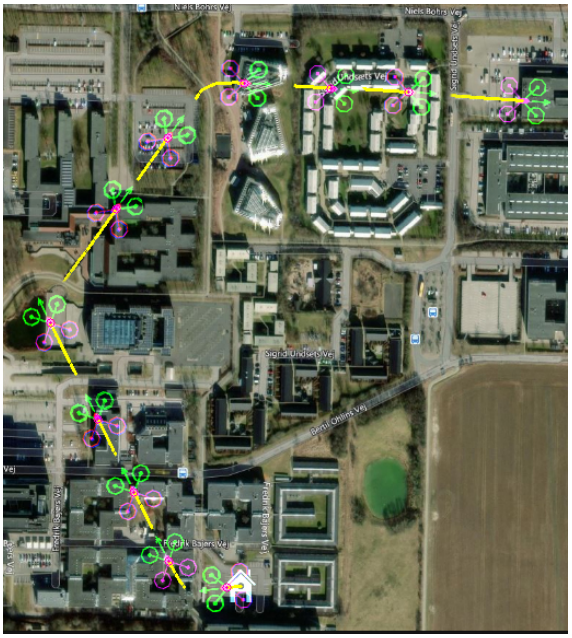
Following the simulation, the generated coordinates are plotted on a map for a visual inspection, allowing for a direct comparison with the drone's simulated route.

10.3.2 Results

The results of this test are shown below. In Figure 10.4a the map with 50 coordinates is illustrated and in Figure 10.4b the route the simulated drone flew can be seen.



(a) Map with the 50 generated coordinates.



(b) Result of simulating the drone following the coordinates.

10.3.3 Test conclusion

In summary, the test successfully evaluated the drone´s ability to adhere to specified coordinates through simulation. By inputting 50 coordinates into the Cube Orange+ during simulation, and then comparing the plotted path with the expected route, it is clear that the drone successfully followed the designated path. This outcome demonstrates the drone's reliable performance in adhering to coordinates fed to the Cube Orange+.

10.4 Validation test - Storing data in the Cube

This section describes a test setup to assess the drone's ability to receive GNSS data from the Position module and process the coordinates. The process involves choosing a route, navigating it with the equipment, and visually inspecting the results. The test journal for this test can be found in appendix A.4.

10.4.1 Test setup

In a practical test, the Cube Orange+ is connected to a PC for logging while being carried around to simulate real-world movement. The test validates the Cube

Orange+'s ability to receive and use GNSS data from the Position module in practical scenarios, such as when the drone is in motion. The logging captures data on its performance, helping identify and address any issues for reliable GNSS utilization in real-world applications.

10.4.2 Results

In Figure 10.5, the comparison between the walked route and the data points received on the Cube, is shown. The GNSS data points received on the Cube had 5 error frames out of the total 724 frames, which results in a FER of 0.69 %.

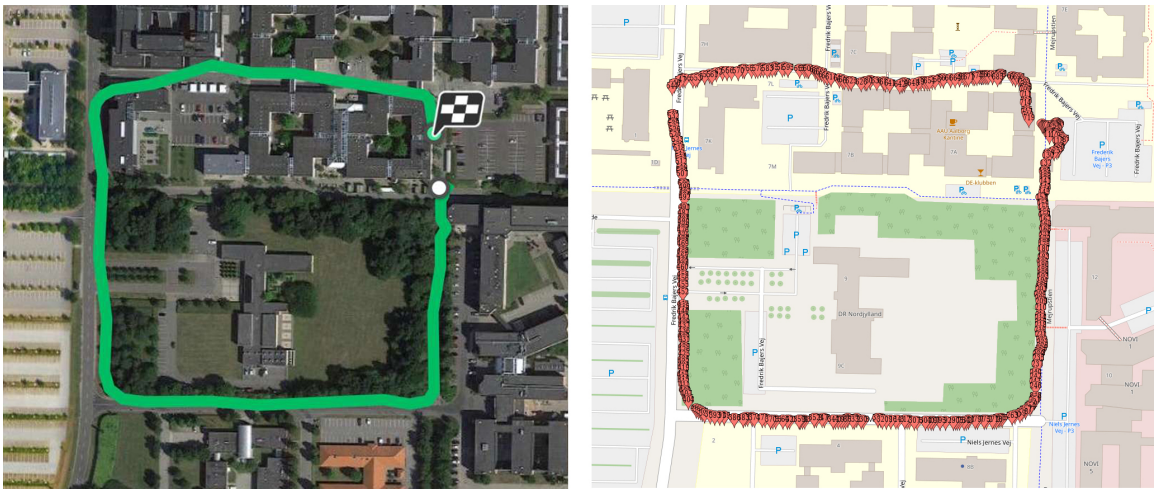


Figure 10.5: Comparison of the marked-out route and actual route received in the Cube Orange+. The marked-out route is on the left and the route from the Cube Orange+ is on the right.

10.4.3 Test conclusion

The results of the test, state that the latitude and longitude after being received and processed match the format of the simulated points in Validation test A.3. This suggests that the full system performs as desired, and it would follow the Position module if it were to be tested with propellers. Additionally, by observation, the data points saved in the Cube Orange+ are sufficiently identical to the data points sent from the Position module.

11 Evaluation

This chapter delves into an evaluation of the project, offering reflections on areas where alternative approaches or enhancements could have been implemented. Firstly, an overview of how the system complied with the requirements is illustrated. This gives an idea of what areas of the system need more work. The discussion revolves around opportunities for future improvements. Furthermore, the project is concluded with the problem definition in mind.

11.1 System compliance

The verification tests above are to be directly compared to their respective requirements. As only one verification test is conducted, the full technical requirement list is still evaluated to know if the system fulfills the requirements. The technical requirements from Table 6.2 are evaluated in Table 11.1, where they are referred to with their respective ID.

Requirement evaluation		
ID	**Description**	**Passed?**
2.1	The Link module is communicating with the Cube Orange+ through a serial line.	✓
2.2	According to Test A.2 the receiving range requirement of minimum 50 m is met.	✓
2.3	Since the Link module weighs 24.2 g it passes the requirements.	✓
2.4	The 3D printed case for the link module has the dimensions 36.2 mm x 28 mm x 51.5 mm the requirement is met.	✓
2.5	According to Test A.2 the transmission range requirement of minimum 50 m is met.	✓
2.6	The system runs at 5 Hz as well as the data transmission.	✓
2.7	Since the full system is not tested through practical tests, it cannot be concluded that the drone would not deviate from the target position with more than 2.5 m.	✗
2.8	According to the Test A.1 the Horizontal Position Accuracy requirement of maximum 2.5 m is met.	✓
2.9	According to the Test A.2, where the FER was found to be 0.14%, the requirement is met.	✓

Table 11.1: The evaluation table of the technical requirements stating if the system fulfills them or not.

11.2 Discussion

Through the project, minor challenges were encountered and the correction of these ultimately led to an improved prototype. This includes communication between the Link and Position module, connecting the Link and Cube module, and the custom ArduPilot code. These thoughts of improvements will be discussed in this section.

11.2.1 Pipeline tests

As a part of the ArduPilot open-source software, there are multiple requirements for the code before it can be approved in a pull request and merged to the main line. This ensures that different contributors can add functionality or maintain things in the code and merge it for everyone to use. One of the measures ArduPi-

lot does to avoid major bugs and errors in the code is pipeline tests.

Pipeline tests are automated tests that in the case of ArduPilot, run each time somebody does a pull request. It is a way of identifying bugs and issues before the end user is affected. In this project, we became aware of ArduPilots pipeline tests when the branch was wrongfully merged onto the main line of the ArduPilot repository. Here it was evident that there were errors in the pipeline related to our code. This does not mean that the code's features were nonfunctional, just that it did not adhere to ArduPilots guidelines.

If the goal was for our code to be added to the ArduPilot repository, we would have to ensure that any errors were taken care of. As a merging rule in the ArduPilot community, all tests must succeed. This is not the case for our code, which has been developed with the intent of making a prototype. This means there are safety features necessary for our code to be implemented in the main repository.

A part of the pipeline tests include integration tests. These are testing whether communication protocols and hardware interfaces are behaving properly.

When developing the UART communication the pipeline tests came out being a great help as they revealed how the UART is being set up. The main use of the pipeline test is the Hardware In Loop (HIL) test. HIL tests are a way to simulate what would happen if the code was deployed on specific hardware. This test is specifically useful in the ArduPilot directory as the purpose of the code is to be volatile and able to run on different setups whether it is a Cube Orange+, Pixhawk, or many other flight controllers [27].

This is a problem in the case of our code since we solely developed it to run on a copter setup using the Cube Orange+. Some features in our code, especially how it is activated, must be changed. Features such as deactivating and activating the track and follow mode safely and controlled are missing.

11.2.2 Why not use altitude?

In the current state of the track and follow mode, that has been developed throughout this project, the position of the drone is entirely dependent on the latitude and longitude of the position module. The reason for not including the altitude from the GNSS module is that we noticed how unreliable the altitude measurements were. This was clear when performing tests on the altitude read-

ings. When the test was performed outside the altitude fluctuated with at least 18.1 m, which is extremely unreliable, cf Test A.5. The altitude measurements are too uncertain to use in this project and would need further work to be implemented. This could potentially be done by finding a rolling average of the altitude values.
Furthermore, the product has been developed to follow a GNSS signal with a fixed altitude, as the altitude is determined at takeoff from the GCS. This makes real-time updating altitude data unnecessary but would be a vital update in further development.

11.2.3 Communication directly to the Cube Orange+

If the communication of this project were to be restructured, the link module could potentially be omitted. Transmitting the position directly to the Cube Orange+ through Bluetooth or IP connections would eliminate the need for the link module. This structure allows for another benefit as well. Omitting the Link module frees up the TELEM 1 port on the cube 8.8 which enables us to connect the telemetry radio and the RC transceiver and still receive GNSS data through the UART. This is a negative limitation that the current system has as either the telemetry radio or the RC transceiver must be disconnected for track-and-follow mode to work since there are only two available ports. The ports referred to are TELEM 1 and TELEM 2, which can be seen in Figure 8.8.

11.2.4 Feedback controller

As of now the `update_velocity_vector` function mentioned in Section 20 uses a proportional controller, where a fixed speed is set from the parameter `KP`. Realistically speaking, for the follow mode to keep up with the Position module's potentially changing velocities, some kind of feedback controller could be implemented. It could vary the length of the velocity vector according to the distance from the drone to the desired location. A proportional controller was implemented in the development phase, resulting in the drone's speed becoming almost 50 times faster when at the maximum allowed distance, compared to when it was close to the desired location. One could argue that this problem could be solved with a nonlinear model, where the velocity of the drone increases slightly if the distance to the desired location exceeds the 7.5 m. So if the Position module is moving, the drone will eventually catch up to the desired 7.5 m from the Position module, no matter the speed of the Position module. Another way to implement feedback is to measure the difference in the GNSS data and add some

kind of parameter to the length of the velocity vector proportional to this difference, as a feed-forward loop.

11.2.5 Future enhancement

In the future, if the project were to be developed further, the most crucial next step would be to test its capabilities in the real world. However, a practical test requires that the user has a drone certificate of type A2 as mentioned in Section 2.2. Such a test would provide a realistic evaluation of performance, identify challenges, and offer insights for improvement.

As mentioned in Section 9.3 the frequency of updating the location sent to the Cube Orange+ has a great impact on the drone's maneuvering performance. One of the critical attributes to pay attention to when performing a practical test is the smoothness of the flight. Trying different scenarios while altering the update rate would lead to finding the best frequency for updating the location data. Until such a test has been performed and passes the requirements, the readiness of the product can not be confirmed.

11.3 Conclusion

This project is focused on implementing track-and-follow functionality into the already autonomous Cube Orange+ flight controller system. This is done by utilizing an ESP32 connected to a NEO-7M GNSS submodule supplying location data wirelessly to the Cube Orange+, using another ESP32 as an intermediary.

Throughout the project, an analysis of track-and-follow in general, led to a problem definition defining the direction of the project. From this, a proposed solution was made where track-and-follow is implemented with the use of GNSS data, transmitted to the drone. Transmitting this data through WiFi using TCP after which the data is parsed into the Cube Orange+ through one of the serial ports on the carrier board. Communication protocol MAVLink2 was adapted for the Cube Orange+ to receive the data. For handling the GNSS data in ArduPilot, two C++ classes were created; **AP_GNSSParser** for parsing and storing the data from the serial port. **AP_Follow_Location** for the drone to follow the GNSS data using velocity references of the already existing guided mode.

Even though the full system was not tested in its entirety, the communication and parsing of the GNSS data were validated through a test. On top of that, simulating the flight of the drone when given GNSS data, not only validated the feasibility of the solution but together with the rest of the system, contributed valuable insights into integrating external hardware with ArduPilot running on the Cube Orange+. Comparing these tests to the technical requirements shows that eight out of nine are passed, and the last is partially met through simulation.

In conclusion, this project achieved its primary goal of obtaining knowledge about Cube Orange+ and ArduPilot, while successfully interacting with the flight controller using external hardware. This led to the showcasing of a theoretical but viable solution for a track-and-follow mode.

Bibliography

[1] TIME STEPHANIE ZACHAREK. *How Drones Are Revolutionizing the Way Film and Television Is Made*. Available at https://time.com/5295594/drones-hollywood-artists/.

[2] Law Enforcement Case Studies Drones. *Drone Lights And Thermal Imaging Camera Used For Search [Case Study]*. Available at https://www.foxfury.com/drone-lights-and-thermal-imaging-camera-used-for-search-case-study/.

[3] Daegu University Seokwon Yeom School of ICT Eng. *Moving People Tracking and False Track Removing with Infrared Thermal Imaging by a Multirotor*. Available at https://www.mdpi.com/2504-446X/5/3/65.

[4] IWM. *A Brief History of Drones*. Available at https://www.iwm.org.uk/history/a-brief-history-of-drones.

[5] Builtin. *What Is a Drone?* Available at https://builtin.com/drones.

[6] UAVCoach. *DENMARK DRONE REGULATIONS*. Available at https://uavcoach.com/drone-laws-in-denmark/.

[7] Trafikstyrelsen. *Flying drones in Denmark*. Available at https://www.droneregler.dk/english.

[8] EMBENTION. *Common and Advanced Control Modes with UAVs*. Available at https://www.embention.com/news/control-modes-with-uavs/.

[9] ArduPilot Dev Team. *The Cube Orange/+ With ADSB-In Overview¶*. Available at https://ardupilot.org/copter/docs/common-thecubeorange-overview.html.

[10] Peter Karanja. Drone Blog. *Follow Me and ActiveTrack on DJI Drones*. Available at https://www.droneblog.com/dji-follow-me-and-activetrack/.

[11] Sem Medhane. Drone Fixer. *DJI Follow Me vs Active Track- which one should you use*. Available at https://drone-fixer.com/follow-me-vs-active-track/.

[12] forum.dji.com. Available at https://forum.dji.com/thread-246113-1-1.html.

[13] luftfartstilsynet.no. Available at https://luftfartstilsynet.no/en/drones/veiledning/c-marking-of-drones/.

[14] eur lex.europa. Available at https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:02019R0947-20210805&from=EN.

[15] eudroneport.com. Available at https://eudroneport.com/blog/follow-me-mode-drone/.

[16] dewesoft. *GPS measurement and recording - GNSS*. Available at https://training.dewesoft.com/online/course/gps-measurement-and-recording-gnss.

[17] MINDTITAN Umama Rahman. *What is object tracking and how does it work?* Available at https://mindtitan.com/resources/blog/object-tracking-with-computer-vision/.

[18] Cubepilot. *The Cube User manual*. Available at https://docs.cubepilot.org/user-guides/autopilot/the-cube-user-manual#telemetry-connection.

[19] PX4.io. *CubePilot Cube Orange Flight Controller*. Available at https://docs.px4.io/main/en/flight_controller/cubepilot_cube_orange.html.

[20] CubePilot. *Most advanced Cube yet*. Available at https://www.cubepilot.com.

[21] ComTek Group 353. *Cube Orange+ Blackbox Analysis*. Technical report. visited: 2023-11-30. Aalborg University, 2023.

[22] www.mybotshop.de. *HOLYBRO X500 V2 ARF KIT*. Available at https://www.mybotshop.de/Holybro-X500-V2-ARF-Kit_1.

[23] mybotshop.de. *HOLYBRO M9N GPS 6PIN - 2ND GPS*. Available at https://www.mybotshop.de/Holybro-M9N-GPS-6Pin-2nd-GPS_1.

[24] MYBOTSHOP. *TELEMETRY RADIO SET V3 433MHZ*. Available at https://www.mybotshop.de/Telemetry-Radio-Set-V3-433MHz.

[25] radiomasterrc.eu. Available at https://www.radiomasterrc.eu/shop/transmitters/tx16s/radiomaster-tx16s-mark-ii/.

[26] radiomasterrc.eu. Available at https://www.radiomasterrc.eu/shop/receivers/expresslrs/rp1-expresslrs-nano-receiver/.

[27] ArduPilot development team. Available at https://github.com/ArduPilot/ardupilot.

[28] Nosferattus. Available at https://en.wikipedia.org/wiki/Decimal_degrees.

[29] ArduPilot.org. Available at https://ardupilot.org/copter/docs/common-rssi-received-signal-strength-indication.html.

[30] Java T Point. Available at https://www.javatpoint.com/naming-conventions-in-cpp.

[31] espressif.com. Available at https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf.

[32] Ardustore.dk. Available at https://ardustore.dk/produkt/ublox-neo-6-7m-gps-module.

[33] NovAtel. Available at https://www.gnss.ca/app_notes/APN-029_GPS_Position_Accuracy_Measures_Application_Note.html.

[34] www.u blox.com. Available at https://content.u-blox.com/sites/default/files/products/documents/NEO-7_DataSheet_(UBX-13003830).pdf.

[35] Arduino.cc. Available at https://docs.arduino.cc/learn/communication/gps-nmea-data-101.

[36] Available at https://github.com/niko5462/P7_MCU-_com.

[37] Github mikalhart. Available at https://github.com/mikalhart/TinyGPSPlus/blob/master/src.

[38] Waveshare. Available at https://www.waveshare.com/w/upload/8/8c/UART-GPS-NEO-7M-C-UserManual.pdf.

[39] ardupilot.org. Available at https://ardupilot.org/blimp/docs/common-esp32-telemetry.html.

[40] espressif.com. Available at https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[41] Gavin Phillips. Available at https://www.makeuseof.com/tag/understanding-common-wifi-standards-technology-explained/.

[42] bluetooth.com. Available at https://www.bluetooth.com/learn-about-bluetooth/key-attributes/range/.

[43] tutorialspoint.com. Available at https://www.tutorialspoint.com/data_communication_computer_network/computer_network_topologies.htm.

[44] Ben Gorman. Available at https://www.avast.com/c-tcp-vs-udp-difference.

[45] Eric Conrad and Joshua Feldman. Available at https://www.sciencedirect.com/topics/computer-science/three-way-handshake.

[46] ArduPilot. Available at https://ardupilot.org/copter/docs/common-thecubeorange-overview.html#system-features.

[47] Koubâa Anis. Allouch Azza. Alajlan Maram. Javed Yasir. Belghith Abdelfettah. Khalgui Mohamed. Available at https://arxiv.org/pdf/1906.10641.pdf.

[48] MAVLink. Available at https://mavlink.io/en/guide/serialization.html.

[49] Simon Kettle. Available at https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-p/902128.

[50] Available at https://github.com/niko5462/ardupilot_TracknFollow.

A Test journals

A.1 Test journal - Position module unit test

A.1.1 Purpose

The purpose of this test is to test the horizontal position accuracy for the specific GNSS submodule used in the Position module. According to the requirement, it should not deviate by more than 2.5 meters.

A.1.2 Test setup

To test the horizontal position accuracy of the NEO-7M GNSS submodule, it is wired up to an ESP32 microcontroller which is connected to a laptop, where the latitude and longitude are read through the serial monitor. To ensure a stable position, the test is conducted over a time period of a minimum of 15 minutes. The test is conducted outside for more similar results to the use case. The setup is shown in Figure A.1.

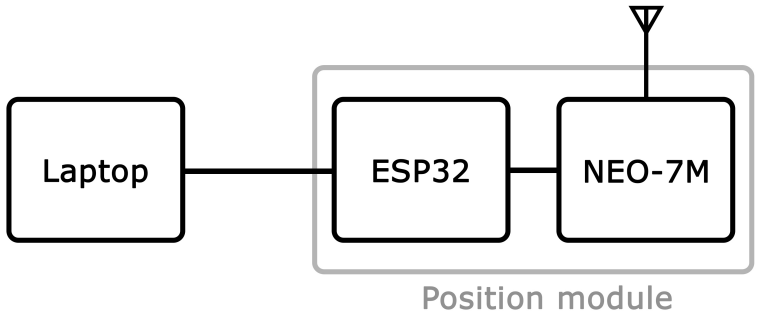


Figure A.1: Test setup of the Unit test to test the GNSS submodule. ESP32 is connected to a laptop for logging.

A.1.3 Succes criteria

The success criteria of this test is that the measured CEP 50% is less than 2.5 meters.

A.1.4 Equipment

The list of equipment used for this test:

- NEO-7M GNSS module
- ESP32-WROOM-32
- Breadboard
- Wires
- µ-USB to USB-A cable
- Laptop w. Arduino, PuTTY, and Python

A.1.5 Procedure

The list of procedures to recreate the test:

- Wire the GNSS module up with the ESP32 on the breadboard.
- Connect the ESP32 to the laptop and open the serial monitor.
- Run the code shown in the GitHub repository [36] folder "ESP Test code\Unit_Test".
- Wait until a minimum of 10 minutes have passed to reach an appropriate sample size.
- Save the samples in a .txt file using a serial monitor (e.g. PuTTY).
- Run the code shown in the GitHub repository [36] folder "Python scripts and data\CEP test" file "unit-test_cep.py".
- Observe the plotted data and the calculated CEPs.

Here are the ESP code used for the position module to get the GNSS data:
Here is the Python script used to plot the GNSS data and calculate multiple CEPs depending on the probability:

A.1.6 Results

The data points were collected over a time period of 15 min.

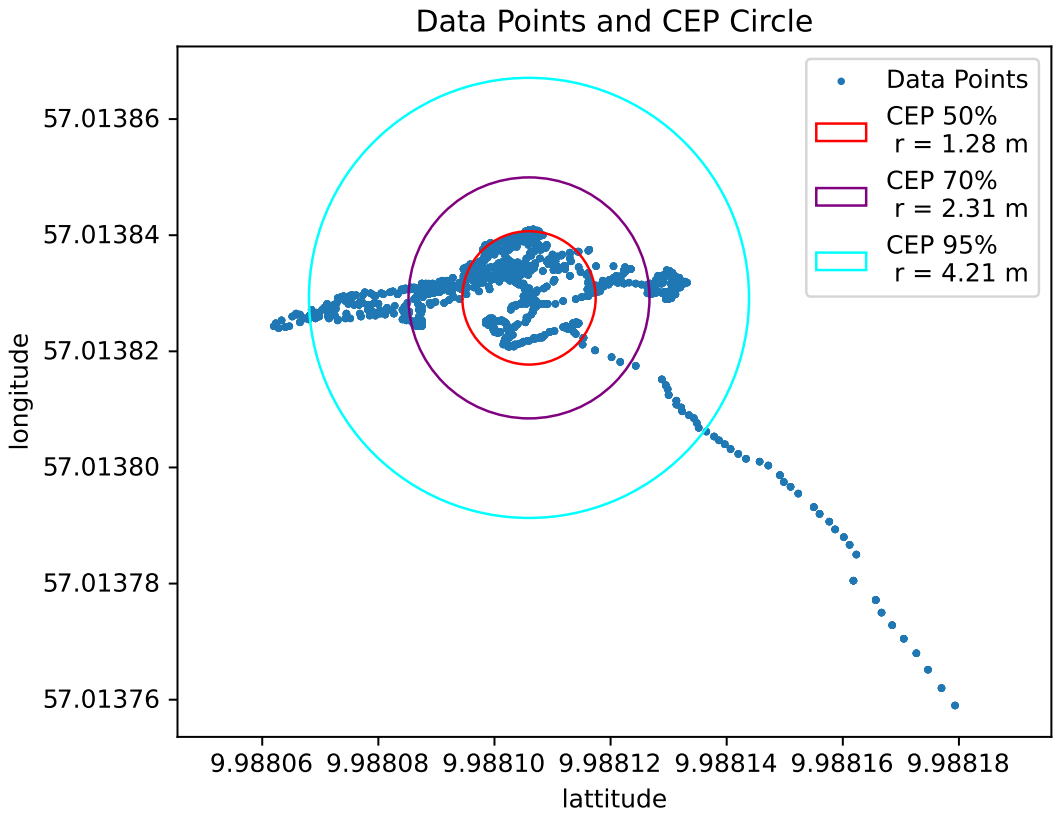


Figure A.2: 4566 GNSS samples collected over 15 min and plotted together with three different CEPs. Latitude on the x-axis, Longitude on the y-axis.

A.1.7 Conclusion

It can be observed in Figure A.2, that the CEP 50% is 1.28 meters and even the CEP 70% is 2.31 meters, and thereby the success criteria of the test is obtained.

A.2 Test journal - Connection range verification test

A.2.1 Purpose

The purpose of this test is to measure the Frame Error Rate (FER) of the communication between the Position module and the Link module. This is done at 50 meters to also test the maximum data transmission range of the technical requirement 2.9.

A.2.2 Test setup

Each microcontroller is connected to its respective laptop and placed 50 meters from each other with no obstacles between them. Then 100 000 frames are sent from one microcontroller to the other, where all the frames are saved and reviewed.

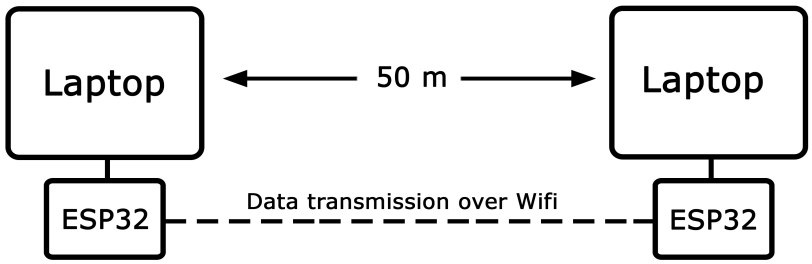


Figure A.3: Test setup of FER test. Both ESP32s are connected to individual laptops for logging.

A.2.3 Equipment

- 2x ESP32-WROOM-32 (with Wifi).
- 2x µ-USB to USB-A cable.
- 2x laptop with a serial monitor.
- Python script to calculate FER found in repository [36] folder "Python scripts and data\FER Tests" file "FER_script4.py".

A.2.4 Procedure

- Connect the micro USB cable from one ESP32 to one laptop.
- Upload the sender code to the microcontroller found in repository [36] folder "ESP Test code\FER_Test_code_position_".
- Connect the other micro USB cable to the other laptop.
- Upload the receiver code to the other microcontroller found in repository [36] folder "ESP Test code\FER_Test_code(link)".
- Open the serial monitor on the receiver laptop and verify that the ESPs are connected.
- Distance the sender and receiver ESP 50 meters from each other in a straight line with no visible interruptions.
- When ready to execute the test, restart the sender ESP, en log the data incoming on the receiver ESP.
- After the 100 000 frames have been sent, upload all data to a file where it can be processed (excel).
- Count the incomplete frames divide the number by 100 000 and multiply with 100 to get the FER in percentage

The Position module, with the code seen in GitHub repository [36] folder "Position Module", is sending a data frame close to the expected data simulating latitude and longitude:

```
Data Sent: 57.pck_no, 9.pck_no, timestamp
```

Where pck_no. is the number of the frame sent and timestamp is in milliseconds. The Link module with the code seen in GitHub repository [36] folder "Link Module" is receiving the data frames with a similar format:

```
57.pck_no, 9.pck_no, timestamp
```

and acting as an access point.

A.2.5 Success criteria

The success criteria of this test is that the calculated FER does not exceed 1%.

A.2.6 Results

The Python script's outcome, or the test result, is presented in the following listing 27. Where the frame error rate and the number of frames with errors can be seen.

```
Running file: FER_Script.py
Total Frames: 100227
Number of frames that do not match in the received file: 144
Frame Error Rate (FER): 0.14%

Process finished with exit code 0
```

Listing 27: Result of Python script.

It is observed that the time it takes to send and receive the 100 000 frames is 347 776 milliseconds or ≈ 348 seconds. This results in a transmission rate (TR) of:

$$TR = \frac{100\,000}{347.776} \approx 266.83 \text{ packages/s} \tag{A.1}$$

This is the amount of packages sent without any additional data flow control.

A.2.7 Test conclusion

The test yielded a FER of 0.14 %, comfortably meeting the success criteria.

A.3 Test journal - Validation test follow

A.3.1 Purpose

The purpose of this test is to test the full system and to validate if the drone would follow the GNSS data points sent by the Position module.

A.3.2 Test setup

Specific test code is used for this test setup, which can be seen in the GitHub repository [50] as the branch "CodeForSimulation", and is simulated on a PC with Python.

A.3.3 Equipment

- PC with Ubuntu and Python.

A.3.4 Procedure

- Generate or choose 50 coordinates.
- Open Ubuntu (optionally in WSL).
- Open the code shown in the GitHub repository [50] as the branch "CodeForSimulation".
- Simulate with the prompt;

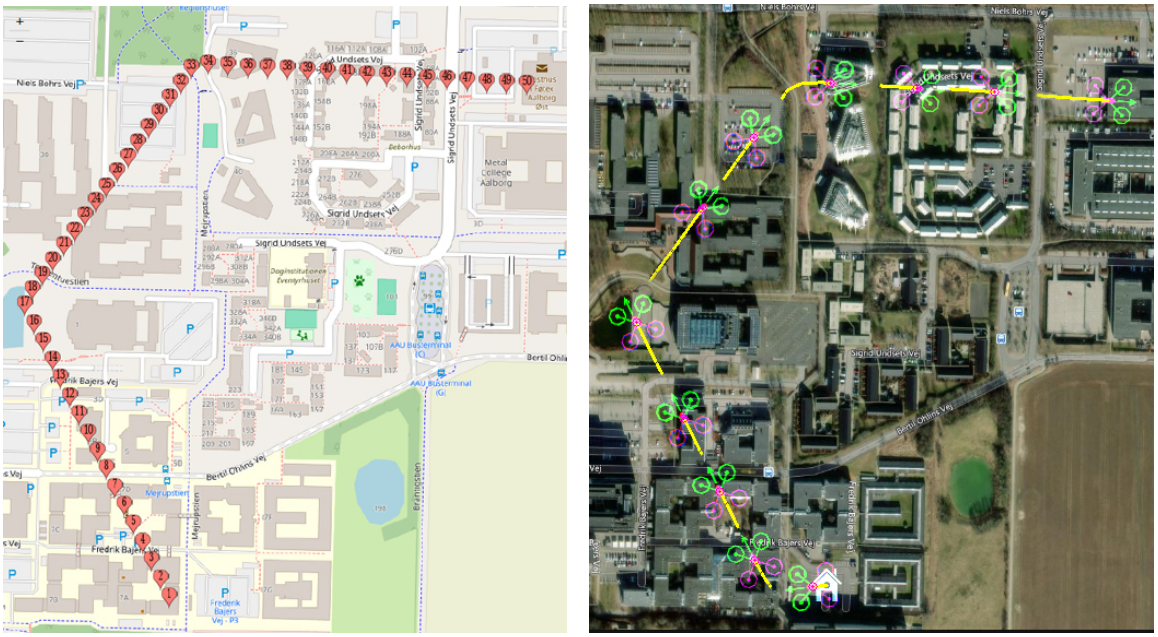
```
python sim\_vehicle.py -v ArduCopter -L AAU\_grupperum --console
    --map
```

- Enter mode guided.
- Arm throttle.
- Takeoff at whatever altitude (has no impact on the test).
- Observe that the drone follows the coordinates.

A.3.5 Success criteria

The success criteria of this test is that by visual inspection the drone adheres to the specified coordinates.

A.3.6 Results



(a) Map with the 50 generated coordinates.

(b) Simulation of the drone following the coordinates.

A.3.7 Conclusion

Upon visual examination, it is evident that the drone adheres to the specified coordinates.

A.4 Test journal - Validation test data

A.4.1 Purpose

The purpose of this test is to focus on the data transmission of the GNSS data, to validate that the data arrives successfully from the Position module and is stored in the Cube Orange+.

A.4.2 Test setup

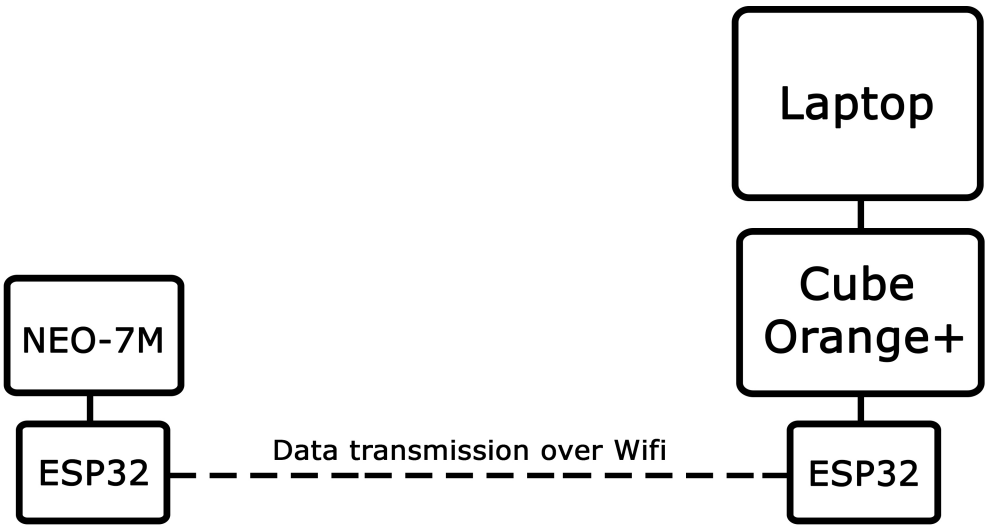


Figure A.5: Test setup of the validation test. The ESP32 is connected to the Cube Orange+ that is connected to the laptop where values are logged. The other ESP32 is coupled with the GNSS module.

A.4.3 Equipment

- Cube Orange+
- 2x ESP32
- NEO-7M
- Power bank
- 2x µ-USB to USB-A cable
- Laptop

A.4.4 Procedure

- Connect the laptop to the Link module via the micro USB cable.
- Upload the code to the Link module ESP32, found in GitHub repository [36] folder "Link Module".
- Disconnect the laptop from the Link module and connect the Link module to the Cube Orange+ via the method described in Figure 9.5.
- Connect the laptop to the Position module via the micro USB cable.
- Upload the code to the Position module ESP32, found in GitHub repository [36] folder "Position Module".
- Disconnect the laptop from the Position module, connect it to the power bank, and let it connect to the Link module via WiFi.
- Upload the code found in GitHub repository [50] branch "master" to the Cube Orange+.
- Mark out a route to walk.
- Walk the route with the Position module and the drone with the Link module attached.
- Log all data on serial port 1 on the Cube Orange+.
- Observe if the two routes are nearly identical.

A.4.5 Success criteria

The success criteria of this test is to observe that the marked-out route is nearly identical to the actual data received on the Cube Orange+. As this proves, that the Cube can receive location data from the position module.

A.4.6 Results

In Figure A.6, the comparison between the walked route and the data points received on the Cube, is shown. The GNSS data points received on the Cube had 5 error frames out of the total 724 frames, which results in a FER of 0.69 % i.e. 0.69 % of the frames were received without a latitude reading.

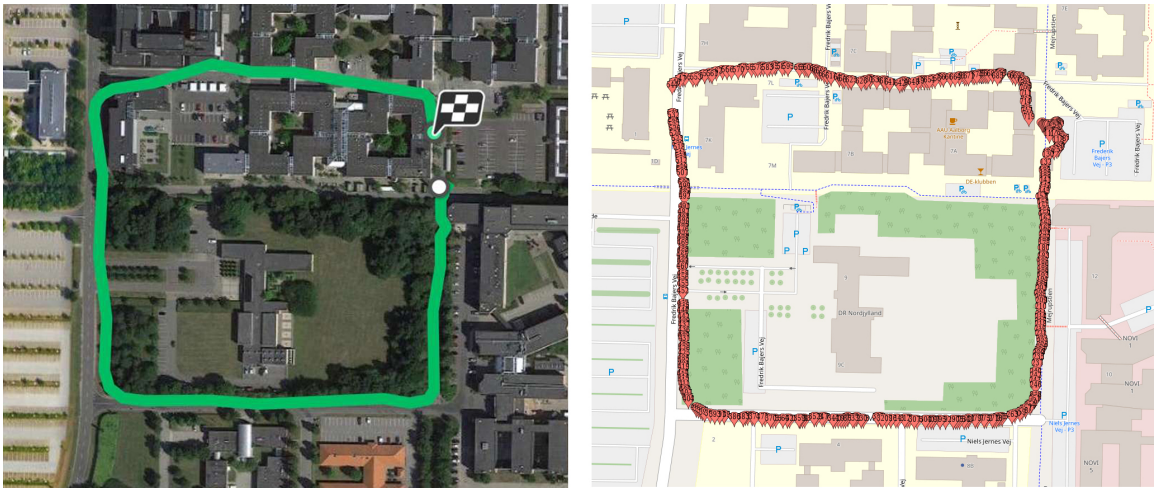


Figure A.6: Comparison of marked-out route (left) and actual route (right) from Cube Orange+.

A.4.7 Conclusion

By observation, the data points saved in the Cube Orange+ are sufficiently identical to the data points sent from the Position module. This is the case even though some frames of data were received with errors.

A.5 Test journal - Altitude test

A.5.1 Purpose

The purpose of this test is to validate the precision of the GNSS module's altitude readings.

A.5.2 Test setup

To test the precision of the altitude readings the ESP32 with the NEO-7m connected is set to read location values and print them on serial. On the laptop, they are being logged to a .txt file with PuTTY.

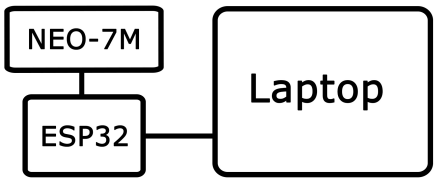


Figure A.7: Test setup for the altitude test. The ESP32 coupled with the GNSS module and connected to a laptop for logging.

A.5.3 Equipment

- ESP32
- NEO-7M GNSS module
- Laptop
- µ-USB to USB-A cable

A.5.4 Procedure

To perform the same test that was carried out, follow the procedure below.

- Connect the ESP32 to the Laptop via the micro USB cable.
- Upload the position module code found in GitHub repository [36] folder "Position Module".

- Make sure that the ESP32 is receiving GNSS locations.
- Place the module outside in a fixed position.
- Start logging the prints through PuTTY.
- Log values for 10 minutes.
- Save the file.

A.5.5 Results

A diagram with results from the test done outside can be seen in Figure A.8. It shows that the altitude fluctuates between the highest value of 116.3 m and the lowest at 51.3 m. However, it is easy to tell that around 2500 measurements an error occurs, where the altitude suddenly spikes. The cause of this issue is uncertain and the unreliable measurements are therefore neglected. With these incorrect data points removed the delta is much smaller.

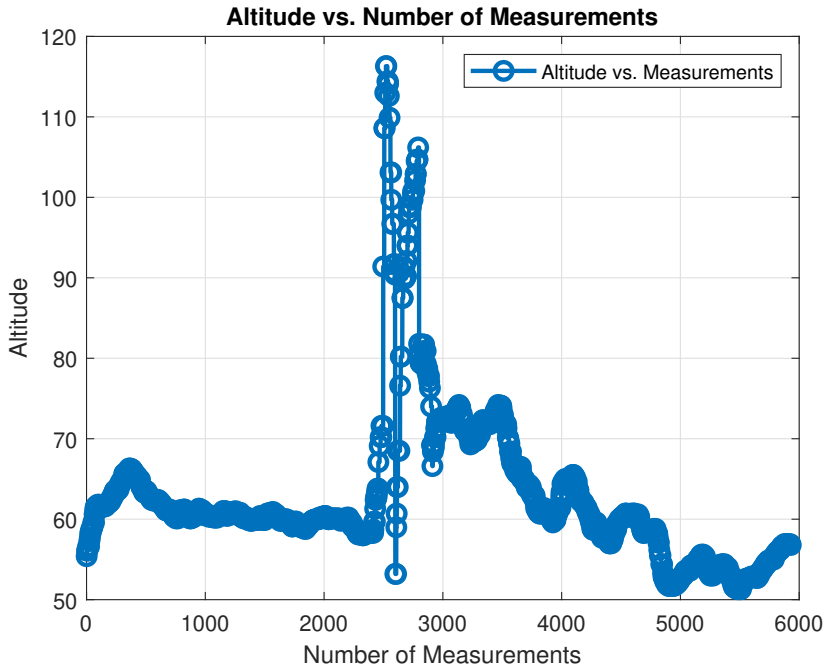


Figure A.8: Diagram of outdoors altitude test. Latitude on the y-axis and number of measurements on the x-axis.

A.5.6 Conclusion

The altitude fluctuates with 65 m when tested outside with the error frames and 18.1 m without the error frames. Hence the test concludes that in an outdoor environment altitude fluctuation becomes