

PROJECT REPORT ON

Study and Implementation of MAVLink Communication Protocol for Drone Control Using Python Libraries and Network Analysis with Wireshark

Submitted by

AADIT NANDAN

Vellore Institute of Technology

Chennai

Under The Guidance of

Shri AMIT TIKARIA (Head WCSES, SECTD)

Co-Guide

Shri SANDEEP KUMAR YADAV (WCSES, SECTD)

Bhabha Atomic Research Centre,

Mumbai 400085, INDIA

September 2023

Certificate

This is to certify that **AADIT NANDAN, Vellore Institute of Technology, Chennai**, has completed his Project Work on **Study and Implementation of MAVLink Communication Protocol for Drone Control Using Python Libraries and Network Analysis with Wireshark** under my guidance.

Signature:_____

Name & Designation: AMIT TIKARIA, SO/H

Section/Division: WCSES/SECTD

Acknowledgement

I am pleased to present the project report on “**Study and Implementation of MAVLink Communication Protocol for Drone Control Using Python Libraries and Network Analysis with Wireshark**”.

I am extremely thankful to my project guide **Shri Amit Tikaria, SO/H, Head WCSES, SECTD** for his excellent guidance and constant encouragement throughout the project work. It was a pleasant and challenging experience to work under his supervision. His constant monitoring and the valuable advices ensured the completion of this report. I thank him for setting high standards and motivating me to do the best.

I also take this opportunity to express my gratitude and deep regards to my co-guide **Shri Sandeep Kumar Yadav SO/C, WCSES, SECTD** for his exemplary guidance, monitoring and constant encouragement throughout the project.

Last but not the least I thank almighty, my family and friends for their constant encouragement and support without which the project would not be possible.

Aadit Nandan

Table of Contents

1	Introduction	9
1.1	Objective of Work	9
1.2	Scope of Work	10
1.3	Deliverables.....	11
1.4	Organization of Report.....	11
2	Literature Review and Theory.....	13
2.1	Drone Fundamentals and Communication Protocols.....	13
2.1.1	Basics of Drones	13
2.1.2	Drone communication	14
2.1.3	Links in Drone Communication	15
2.1.4	Drone Companies and their protocols.....	17
2.2	MAVLink.....	17
2.2.1	Introduction	17
2.2.2	More on MAVLink	17
2.2.3	Conclusion.....	20
2.3	Wireless Local Area Networks	20
2.3.1	Introduction	20
2.3.2	Why Wireless communication	21
2.3.3	Radio Spectrum.....	21
2.3.4	ISM Bands.....	22
2.3.5	Architecture of IEEE 802.11	23
2.3.6	Limitations of Wireless Networking.....	27
2.4	Software Used.....	27
2.4.1	Dronekit-python.....	27
2.4.2	Mavproxy	28
2.4.3	Dronekit-sitl	29
2.4.4	Pymavlink	29
2.4.5	Wireshark.....	30
3	Implementation and Results.....	32
3.1.1	Setting up a simulated drone	32
3.1.2	Setting up a command line GCS.....	33
3.1.3	Setting up the python script	34
3.1.4	Implementing all the three	37
3.1.5	Verification of MAVLink Frames	38
4	Conclusion & Future Scope	47
4.1	Conclusion.....	47

4.2	Future Scope	48
5	References	49
6	Annexure	50
6.1	MAVLink Message ID and their corresponding messages	50
6.2	Python Script Containing more functions to perform	55

List of Tables

Table 1:Drone Companies and their communication protocols	17
Table 2:MAVLink Frame fields and their functions	19
Table 3: Dronekit-sitl arguments and their functions	32
Table 4: Breakdown of the Heartbeat frame	40
Table 5: Breakdown of the request data stream frame	41
Table 6: Breakdown of a command frame	42
Table 7: Breakdown of frame carrying GPS information	43
Table 8: Breakdown of the frame carrying pressure data.....	44
Table 9: Breakdown of Frame carrying IMU Data	45
Table 10: Breakdown of frame carrying Command ACK	46

List of Figures

Figure 1: Forces acting on a drone [1]	14
Figure 2: Communication Links between Drone and GCS	16
Figure 3: MAVLink v1 Frame Structure [3]	19
Figure 4: MAVLink v2 Frame Structure [3]	19
Figure 5: ISM Frequency Bands [4]	23
Figure 6: BSS Types [5]	24
Figure 7: Architecture of IEEE 802.11 [5]	26
Figure 8: Comparison of 802.11 Standards [5]	26
Figure 9: Simulating a copter using dronekit-sitl	33
Figure 10: Setting up mavproxy GCS	34
Figure 11: Implementating all the three together	37
Figure 12: Sequential sending and receiving of frames	38
Figure 13: Connection Establishment Frames	39
Figure 14: Heartbeat frame sent to python script	40
Figure 15: Frame to request data stream	41
Figure 16: Frame Carrying a command	42
Figure 17: Frame carrying GPS data	43
Figure 18: Frame carrying pressure data of the drone	44
Figure 19: Frame carrying IMU Data	45
Figure 20: Frame carrying Command ACK	46

Abstract

This report details the project work done in Bhabha Atomic Research Centre focuses on the comprehensive study, implementation, and validation of the MAVLink communication protocol for efficient drone control. The objective is to develop a Python-based implementation for simulating communication between a ground control station (GCS) and a virtual drone while employing network analysis tools like Wireshark for protocol validation.

The methodology includes a detailed examination of MAVLink, emphasizing its message structure and payload components. A Python-based MAVLink communication system will be created to mimic real-world interactions between a GCS and a drone. A simulated drone environment will be established to facilitate message exchange between the GCS and the virtual drone.

The project's critical aspect involves network traffic analysis using Wireshark. This tool will capture and dissect MAVLink packets, providing insights into the protocol's efficiency and reliability. Rigorous testing and validation within the simulated environment will ensure seamless protocol functionality under diverse conditions and message types.

This project's significance lies in its contribution to the field of drone communication, offering a practical framework for reliable drone control. Leveraging Python libraries and open-source tools ensures accessibility and adaptability to various applications. Ultimately, this study seeks to enhance our understanding of drone communication, advancing UAV technology's capabilities and applications across industries.

1 Introduction

The rapid advancements in technology have paved the way for innovative solutions in various sectors, including aerospace and communication systems. Unmanned Aerial Vehicles (UAVs), commonly known as drones, have emerged as a transformative technology with a wide range of applications, from surveillance and reconnaissance to agriculture and delivery services. This project, "Study and Development of Drone's Communication Protocol," aims to study and development of drone's communication protocol for robust and reliable communication between drone and it remote controller.

1.1 Objective of Work

The primary objective of this project is to conceptualize, develop and validate drone communication protocol with resilient communication system for practical deployment. Specifically, our objectives include:

- **Study of Drones communication protocols:** This involves an extensive study and analysis of existing communication standards and frameworks to garner valuable insights into their functionality and applicability.
- **Communication System Integration:** The subsequent step revolves around the development and seamless integration of a communication system protocol, primarily relying on the utilization of MAVLink. This system will facilitate robust and efficient data transfer, command transmission, and real-time feedback.
- **Testing and Validation with simulated drone:** Rigorous testing and validation procedures will be conducted to assess drone's communication protocol using network protocol analyzer like Wireshark.

1.2 Scope of Work

The scope of this project includes the following activities:

- **Study of Drone Communication Protocols across Various Layers:** Our project initiates with an extensive examination of drone communication protocols across different layers of the network stack. This in-depth analysis will provide a comprehensive understanding of how these protocols operate at various levels and how they interconnect within the communication framework.
- **Software Development for Application Layer Communication using MAVLink:** Building upon the insights gained from the protocol study, we will proceed to develop software tailored for facilitating drone communication at the application layer. MAVLink will serve as the foundational protocol for this purpose. Our software development efforts will focus on ensuring seamless data transfer, command propagation, and real-time interaction between the simulated drone and its control center.
- **Testing and Evaluation with Simulated Drone:** To validate the effectiveness and reliability of the communication system, we will subject it to rigorous testing and evaluation procedures. The simulated drone will serve as the testbed for these assessments. Using network analysis tools like Wireshark, we will closely monitor and analyze the communication exchanges between the simulated drone and the control center. This meticulous scrutiny will allow us to identify any anomalies, ensure adherence to industry standards, and fine-tune the communication system for optimal performance.

1.3 Deliverables

The deliverables for this project consist of the following:

- **MAVLink Implementation for Drone Communication Protocol in Python:** The primary deliverable is the MAVLink implementation for the drone communication protocol, developed using Python. This implementation will serve as a crucial component of the communication system, enabling seamless data exchange, command transmission, and real-time interaction between the drone and its control center.
- **Comprehensive Documentation of Drone Protocols, Libraries, and Software:** In addition to the MAVLink implementation, we will provide comprehensive documentation that encompasses a detailed explanation of various drone communication protocols, libraries, and software utilized throughout the project. This documentation will serve as a valuable resource, offering insights into the protocols' functionalities, libraries' capabilities, and software components' usage, facilitating a deeper understanding of the project's technological foundations.

1.4 Organization of Report

This report is organized to provide a clear and structured overview of the project. The following sections will be included:

Introduction: This section provides an overview of the project, its objectives, scope, deliverables, and organization.

Literature Review and Theory: A review of relevant literature and existing drone and communication systems will be presented to establish the project's context.

Implementation and results: All the work done on the topic will be presented along with screenshots.

Conclusion and future scope: A summary of the project's achievements, along with insights into future developments and applications.

References: A list of all sources and references used throughout the report.

Annexure.

2 Literature Review and Theory

2.1 Drone Fundamentals and Communication Protocols

In the realm of modern technology, the integration of unmanned aerial vehicles (UAVs) has burgeoned, demonstrating remarkable potential in various sectors, from surveillance and agriculture to emergency response. The Bhabha Atomic Research Centre (BARC) has consistently been at the forefront of innovation, and the project undertaken within its hallowed halls stands as a testament to this tradition. This report delves into the culmination of diligent efforts to implement the MAVLink Drone Communication Protocol, a pivotal component in the operation of UAVs, utilizing a simulated drone (copter) as the primary vehicle for development.

The project's core objectives encompassed not only the establishment of MAVLink communication but also the development of essential functions to facilitate control of the simulated drone. Leveraging the power of Python, alongside the specialized libraries pymavlink and dronekit, this endeavor achieved seamless integration and communication with the drone. Crucially, the simulation aspect was achieved using Dronekit-sitl, an integral tool for replicating real-world drone behavior within a controlled environment.

This report will outline the methodologies, challenges, accomplishments, and future prospects of this project, offering an in-depth insight into the innovative work carried out at BARC in the realm of UAV communication and control.

2.1.1 Basics of Drones

Any aircraft or flying machine operated without a human pilot such machines is called an unmanned aerial vehicle(UAV). It can be guided autonomously or remotely by a human operator

using onboard computers and robots. During surveillance or military operation, UAVs can be a part of an unmanned aircraft system (UAS), Drones are separately for air and water. Drones have become increasingly popular in recent years [1]. They are used for a variety of purposes, including photography, videography, surveying, inspection, and even delivery. The basic components of a drone are the following:

- Frame
- Battery
- Flight controllers
- Sensors
- Motors and Propellers.

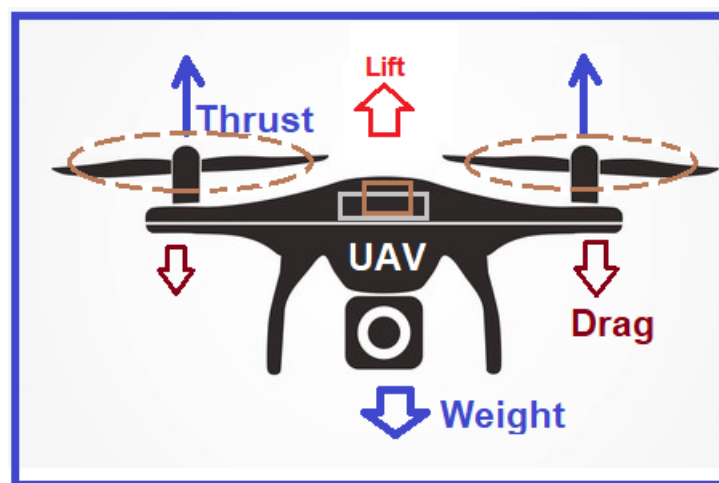


Figure 1: Forces acting on a drone [1]

2.1.2 Drone communication

Drone and UAVs (unmanned aerial vehicles) utilize a variety of communication methods in order to create a data link between the vehicle and ground control station (GCS), and sometimes between multiple aircraft (swarm technology). UAV communication systems may be used for

command and control (C2), to broadcast telemetry data, and to send video and data from sensors and payloads back to the control station. The most common method of drone communications uses radio— RF (radio frequency) signals in bands such as HF (high frequency) and UHF (ultra high frequency). RF datalinks may be analogue or digital, and provide greater range than Wi-Fi but are still limited to line-of-sight (LOS). Range will depend on the size of the antenna and the power of the transmitter in the UAV communications system, as well as frequency, with lower frequencies providing longer ranges but lower data rates. SATCOM (satellite communications) provides coverage almost anywhere on the planet, as well as excellent uptime. However, the equipment is relatively bulky, and thus the method is not yet suitable for SWaP (size, weight and power)-limited platforms. SATCOM also requires a subscription service, which may be expensive. Cellular communications using 4G and 5G are also used for UAV comms. While coverage may vary, particularly outside of urban areas, cellular data links can enable BVLOS (beyond visual line of sight) operations, and 5G in particular provides a number of advantages such as ultra-low latency and enhanced data rates. 5G is highly suited to bandwidth-intensive applications such as streaming high-definition video [2].

2.1.3 Links in Drone Communication

RC : Remote Control signals generally have 2MHz Bandwidth and employ Frequency hopping spread spectrum (FHSS) while operating.

Telemetry: This link is used to provide drone status information like location, altitude, battery etc to the Ground Control Station (GCS). The telemetry frequencies include 433 MHz, 915 MHz, and the newer 2.4 GHz and 5.8 GHz.

Video Stream: Usually operates on a bandwidth of 20MHz which is the standard when it comes to 802.11 Wireless Local Area Networks (WLANS). It enables the operator or GCS to see what

the drone's camera sees in real-time.

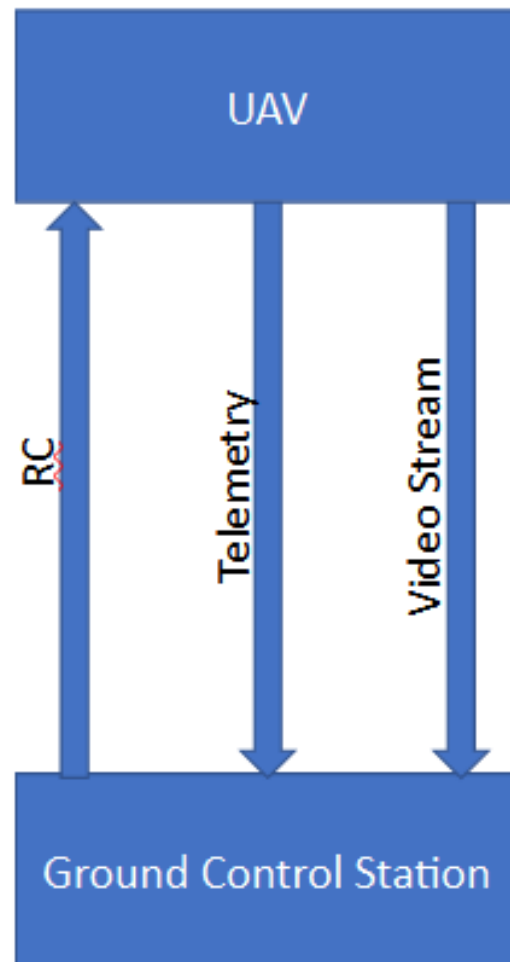


Figure 2: Communication Links between Drone and GCS

2.1.4 Drone Companies and their protocols

Table 1:Drone Companies and their communication protocols

Company Name	Protocol Used (Application Layer)	Protocol Used (MAC Layer)
DJI	Ocusync (Proprietary)	WiFi
Skydio	MAVLink (Open Source)	WiFi
Parrot	MAVLink (Open Source)	WiFi
Altavian	MAVLink (Open Source)	WiFi
Bitall Technologies	JREAP (Proprietary)	Bluetooth
General Atomics	STANAG 4586 (Proprietary)	WiFi
Vantage Robotics	MAVLink (Open Source)	Bluetooth

2.2 MAVLink

2.2.1 Introduction

MAVLink, short for Micro Air Vehicle Link, is a lightweight and efficient communication protocol designed for unmanned systems, particularly unmanned aerial vehicles (UAVs) and ground control stations (GCS). It has two versions 1.0/2.0 or v1/v2. It serves as the backbone of data exchange, enabling seamless and standardized communication between these systems. MAVLink facilitates the transmission of vital information, including telemetry data, control commands, and status updates, in a format that is both highly efficient and adaptable. This protocol plays a pivotal role in the world of autonomous vehicles, empowering developers and operators to interact with and manage UAVs, and it continues to be a fundamental component of the ever-expanding ecosystem of unmanned systems and robotics [3].

2.2.2 More on MAVLink

MAVLink, the Micro Air Vehicle Link, is a versatile communication protocol specifically designed for unmanned systems, including drones, ground control stations (GCS), and various

robotic platforms. It has become a standard in the field of autonomous vehicles and robotics due to its efficiency, reliability, and adaptability. Here are some key aspects to understand:

Efficiency and Lightweight: MAVLink is renowned for its efficiency. It is a minimalistic protocol, utilizing a binary message format that minimizes data size. This lightweight nature is critical for real-time communication, especially in resource-constrained environments common in UAVs.

Compatibility: MAVLink is compatible with both MAVLink 1.0 and MAVLink 2.0, offering flexibility for developers and users to choose the appropriate version for their applications. This ensures backward compatibility while providing advanced features in MAVLink 2.0.

Message Types: MAVLink defines a wide range of message types, covering aspects like telemetry, command and control, mission planning, status updates, and more. These messages enable comprehensive communication between UAVs and GCS, making it an invaluable tool for mission planning and monitoring.

Open Source and Extensible: MAVLink is open source, fostering a vibrant community of developers who continuously contribute to its development. It is also extensible, allowing users to define custom messages to suit their specific needs.

Platform Agnostic: MAVLink is platform-agnostic and can be used with various programming languages, including C, C++, Python, and more. This versatility makes it accessible to a wide range of developers.

Applications: MAVLink finds applications not only in drones but also in ground robots, surface vehicles, and even underwater vehicles. Its ability to facilitate real-time communication is crucial for mission-critical tasks in these domains.

Safety and Reliability: In safety-critical applications like UAVs, MAVLink's reliability is

paramount. Its message integrity checks and error detection mechanisms ensure data accuracy during transmission.

Community and Documentation: The MAVLink community provides extensive documentation, tools, and libraries to support developers in implementing and integrating MAVLink into their projects.

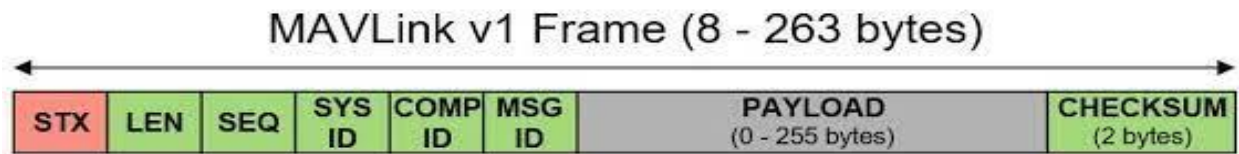


Figure 3: MAVLink v1 Frame Structure [3]

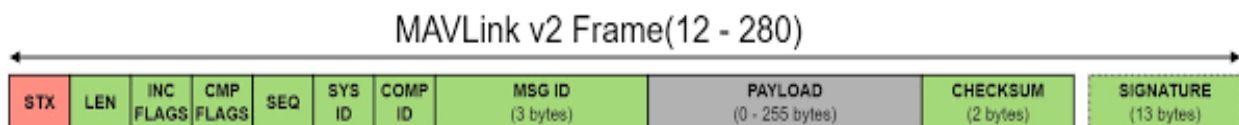


Figure 4: MAVLink v2 Frame Structure [3]

Table 2:MAVLink Frame fields and their functions

COMPONENT	FUNCTION
STX	Signifies Start of Frame
LEN	Length of Payload
SEQ	Sequence number for reordering of Frames
SYS ID	System id to identify source UAV
COMP ID	Component id to identify source component
MSG ID	Message id to identify type of message
PAYLOAD	Actual data being carried
CHECKSUM	Error Checking bits
INC FLAGS	Incompatibility flags to signify change in the way of handling the packet
CMP FLAGS	Compatibility flags to signify resources required
SIGNATURE	Used to ensure link is untampered

2.2.3 Conclusion

In summary, MAVLink plays a pivotal role in enabling the seamless exchange of data and commands between unmanned systems and ground control stations. Its lightweight design, compatibility, and extensibility make it a go-to choice for developers and researchers in the rapidly evolving fields of autonomous vehicles and robotics.

2.3 Wireless Local Area Networks

2.3.1 Introduction

Wireless Local Area Networks (WLANs) have emerged as a cornerstone of modern networking, revolutionizing the way we connect, communicate, and access information. WLANs, often referred to as Wi-Fi networks, offer wireless connectivity within a limited geographical area, such as homes, offices, educational institutions, and public spaces. This section explores the key concepts, technologies, and significance of WLANs in the contemporary digital landscape.

WLANs employ radio frequency (RF) technology to transmit data wirelessly, liberating users from the constraints of physical connections. This freedom fosters mobility, enabling seamless access to the internet, data sharing, and real-time communication across a spectrum of devices, from smartphones and laptops to IoT devices.

The proliferation of WLANs has driven advancements in wireless standards, security protocols, and network management strategies. This report delves into the essential components of WLANs, including access points, wireless routers, client devices, and the intricate protocols governing their operation. It also examines the evolving landscape of Wi-Fi technologies, from the advent of Wi-Fi 6 (802.11ax) to the promising prospects of Wi-Fi 7 (802.11be).

Moreover, this section explores the pivotal role of WLANs in enabling the Internet of Things (IoT), Industry 4.0, and the digital transformation of various sectors. As WLANs continue to evolve and shape the connected world, it becomes imperative to grasp their underlying principles and their multifaceted impact on our daily lives and the broader technological landscape.

2.3.2 Why Wireless communication

The most obvious advantage of wireless networking is mobility. Wireless network users can connect to existing networks and are then allowed to roam freely. A mobile telephone user can drive miles in the course of a single conversation because the phone connects the user through cell towers. Wireless networks typically have a great deal of flexibility, which can translate into rapid deployment. Wireless networks use a number of base stations to connect users to an existing network. The infrastructure side of a wireless network, however, is qualitatively the same whether you are connecting one user or a million users. To offer service in a given area, you need base stations and antennas in place. Once that infrastructure is built, however, adding a user to a wireless network is mostly a matter of authorization.

2.3.3 Radio Spectrum

Wireless devices are constrained to operate in a certain frequency band. Each band has an associated bandwidth, which is simply the amount of frequency space in the band. Bandwidth has acquired a connotation of being a measure of the data capacity of a link. A great deal of mathematics, information theory, and signal processing can be used to show that higher-bandwidth slices can be used to transmit more information. As an example, an analog mobile telephony channel requires a 20-kHz bandwidth. TV signals are vastly more complex and have a correspondingly larger bandwidth of 6 MHz.

2.3.4 ISM Bands

ISM bands (industrial, scientific, and medical) are parts of the RF (radio-frequency) spectrum reserved for general use by, as the name suggests, scientific, medical, and industrial devices. Their creation was in response to specific demands by these industries for access to the RF spectrum to perform functions without having to compete with the traditionally intensive users of RF, the communications, and defense sectors.

The critical thing to note with ISM bands, which can be deduced from the name, is that they were conceived to exclude any device that uses RF for audio communications explicitly. This application has its own frequency bands set aside and typically requires a frequency license to operate. This ensures that the devices operating in the ISM bands can't interfere with communications equipment through frequency segregation, rather than relying on testing and certification to ensure such interference cannot occur. This simplifies the approval process for devices that operate in the ISM bands and hence why ISM bands are sometimes referred to as unlicensed bands. It must be made clear that unlicensed means the device does not need an individual license from a telecommunication regulatory authority. This is not the same as unregulated; the device will still need to meet strict regulations and be certified by an appropriate regulatory authority.

The creation of ISM bands came about due to the radio-frequency spectrum being ever-increasing, filled with a whole range of communications, radio, television, radar, and other applications, including microwave ovens that crowded out other potential users. Another issue was the different RF bands were used for different applications worldwide, so the frequencies available for a particular application varied depending on which country the device was intended to operate in. There has been an enormous growth in such devices; everything from radio-

controlled toys, remote garage door openers, baby monitors, cordless phones, WiFi, and Bluetooth all make use of ISM bands. This is a trend for increasing the utilization of RF by domestic appliances that shows no sign of diminishing [4].

ISM frequency	Band (tolerance)	
6.78 MHz	±15.0 KHz	
13.56 MHz	±7.0 KHz	
27.12 MHz	±163.0 KHz	
40.68 MHz	±20.0 KHz	RC remotes
915 MHz	±13.0 MHz	
2.45 GHz	±50.0 MHz	WLANs
5.8 GHz	±75.0 MHz	
24.125 GHz	±125.0 MHz	Wi-Fi, M/A, X
61.25 GHz	±250.0 MHz	
122.5 GHz	±500.0 MHz	
245 GHz	±1.0 GHz	

Figure 5: ISM Frequency Bands [4]

2.3.5 Architecture of IEEE 802.11

Distribution System: When several access points are connected to form a large coverage area, they must communicate with each other to track the movements of mobile stations. The distribution system is the logical component of 802.11 used to forward frames to their destination. 802.11 does not specify any particular technology for the distribution system. In most commercial products, the distribution system is implemented as a combination of a bridging engine and a distribution system medium, which is the backbone network used to relay frames between access points; it is often called simply the backbone network. In nearly all

commercially successful products, Ethernet is used as the backbone network technology.

Access Points: Frames on an 802.11 network must be converted to another type of frame for delivery to the rest of the world. Devices called access points perform the wireless-to-wired bridging function. Access points perform a number of other functions, but bridging is by far the most important.

Basic Service Set (BSS): The basic building block of an 802.11 network is the basic service set (BSS), which is simply a group of stations that communicate with each other. Communications take place within a somewhat fuzzy area, called the basic service area, defined by the propagation characteristics of the wireless medium. When a station is in the basic service area, it can communicate with the other members of the BSS.

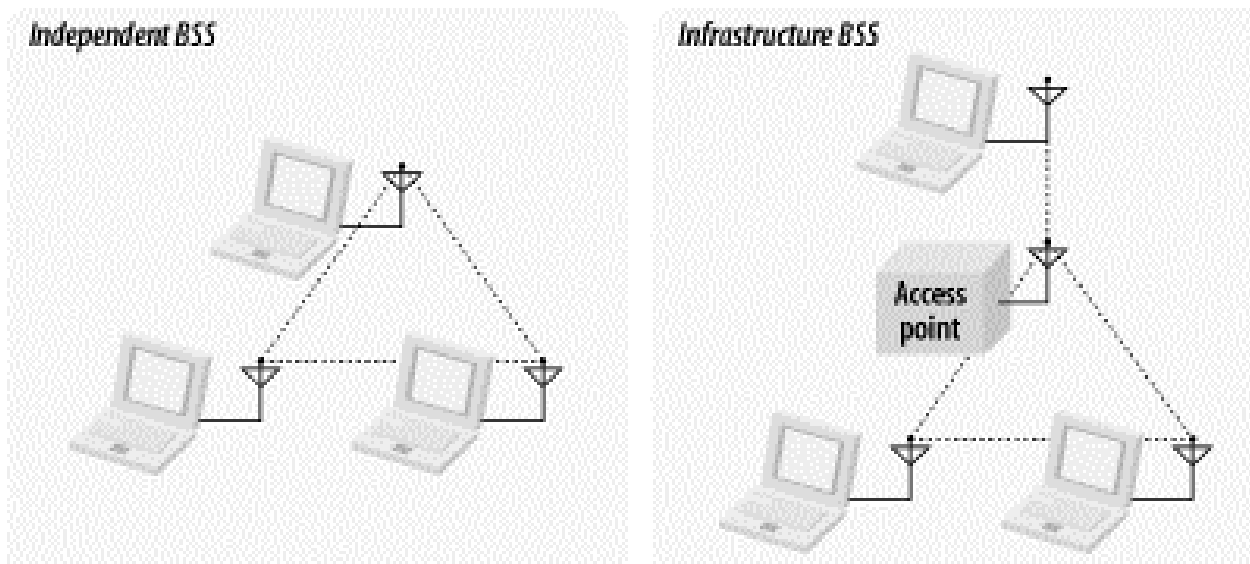


Figure 6: BSS Types [5]

On the left is an independent BSS (IBSS). Stations in an IBSS communicate directly with each other and thus must be within direct communication range. The smallest possible 802.11 network is an IBSS with two stations. Typically, IBSSs are composed of a small number of stations set up for a specific purpose and for a short period of time. One common use is to create a short-lived

network to support a single meeting in a conference room. As the meeting begins, the participants create an IBSS to share data. When the meeting ends, the IBSS is dissolved. Due to their short duration, small size, and focused purpose, IBSSs are sometimes referred to as ad hoc BSSs or ad hoc networks.

On the right side is an infrastructure BSS. Infrastructure networks are distinguished by the use of an access point. Access points are used for all communications in infrastructure networks, including communication between mobile nodes in the same service area. If one mobile station in an infrastructure BSS needs to communicate with a second mobile station, the communication must take two hops. First, the originating mobile station transfers the frame to the access point. Second, the access point transfers the frame to the destination station. With all communications relayed through an access point, the basic service area corresponding to an infrastructure BSS is defined by the points in which transmissions from the access point can be received.

Extended Service Set (ESS): BSSs can create coverage in small offices and homes, but they cannot provide network coverage to larger areas. 802.11 allows wireless networks of arbitrarily large size to be created by linking BSSs into an extended service set (ESS). An ESS is created by chaining BSSs together with a backbone network. 802.11 does not specify a particular backbone technology; it requires only that the backbone provide a specified set of services. Stations within the same ESS may communicate with each other, even though these stations may be in different basic service areas and may even be moving between basic service areas. For stations in an ESS to communicate with each other, the wireless medium must act like a single layer 2 connection. Access points act as bridges, so direct communication between stations in an ESS requires that the backbone network also be a layer 2 connection. Any link-layer connection will suffice. Several access points in a single area may be connected to a single hub or switch, or they can use

virtual LANs if the link-layer connection must span a large area [5].

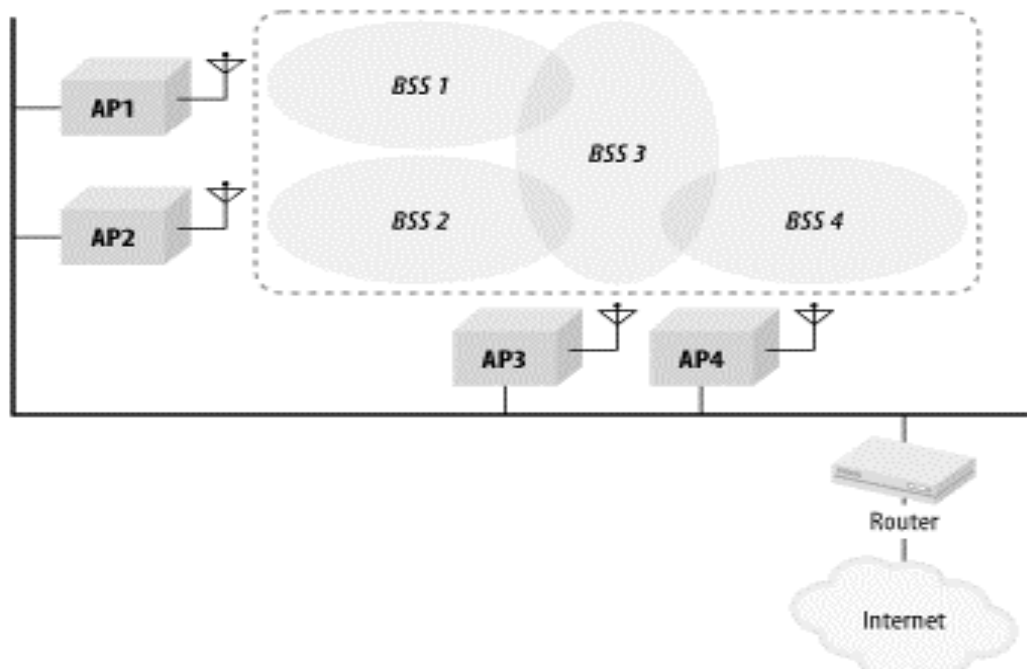


Figure 7: Architecture of IEEE 802.11 [5]

IEEE standard	Speed	Frequency band	Notes
802.11	1 Mbps 2 Mbps	2.4 GHz	First standard (1997). Featured both frequency-hopping and direct-sequence modulation techniques.
802.11a	up to 54 Mbps	5 GHz	Second standard (1999), but products not released until late 2000.
802.11b	5.5 Mbps 11 Mbps	2.4 GHz	Third standard, but second wave of products. The most common 802.11 equipment as this book was written.
802.11g	up to 54 Mbps	2.4 GHz	Not yet standardized.

Figure 8: Comparison of 802.11 Standards [5]

2.3.6 Limitations of Wireless Networking

The speed of wireless networks is constrained by the available bandwidth. Information theory can be used to deduce the upper limit on the speed of a network. Unless the regulatory authorities are willing to make the unlicensed spectrum bands bigger, there is an upper limit on the speed of wireless networks. Using radio waves as the network medium poses several challenges. Specifications for wired networks are designed so that a network will work as long as it respects the specifications. Radio waves can suffer from a number of propagation problems that may interrupt the radio link, such as multipath interference and shadows.

2.4 Software Used

This chapter of the report is dedicated to understanding the software used in this project in detail. We will be looking at the software one by one and look at their features and the problems they solve.

2.4.1 Dronekit-python

DroneKit-Python allows developers to create apps that run on an onboard companion computer and communicate with the ArduPilot flight controller using a low-latency link. Onboard apps can significantly enhance the autopilot, adding greater intelligence to vehicle behavior, and performing tasks that are computationally intensive or time-sensitive (for example, computer vision, path planning, or 3D modelling). DroneKit-Python can also be used for ground station apps, communicating with vehicles over a higher latency RF-link. The API communicates with vehicles over MAVLink. It provides programmatic access to a connected vehicle's telemetry, state and parameter information, and enables both mission management and direct control over

vehicle movement and operations. DroneKit-Python is an open source and community-driven project. DroneKit-Python is compatible with vehicles that communicate using the MAVLink protocol (including most vehicles made by 3DR and other members of the DroneCode foundation). It runs on Linux, Mac OS X, or Windows [6].

The API provides classes and methods to:

- Connect to a vehicle (or multiple vehicles) from a script
- Get and set vehicle state/telemetry and parameter information.
- Receive asynchronous notification of state changes.
- Guide a UAV to specified position (GUIDED mode).
- Send arbitrary custom messages to control UAV movement and other hardware (GUIDED mode).
- Create and manage waypoint missions (AUTO mode).
- Override RC channel settings.

2.4.2 Mavproxy

MAVProxy is a powerful command-line based “developer” ground station software that complements your favorite GUI ground station, such as Mission Planner, APM Planner 2 etc. It has a number of key features, including the ability to forward the messages from your UAV over the network via UDP to multiple other ground station software on other devices. MAVProxy can be used by developers (especially with SITL) for testing new builds. This ground station was developed as part of the Canberra UAV OBC team entry to enable the use of companion computing and multiple datalinks with ArduPilot. It has grown to be one of the most versatile tools in the ArduPilot ecosystem, and many of the features users now see in other GCS tools can

trace their origins to MAVProxy.

Lead Developers: Andrew Tridgell and Peter Barker

Windows Maintainer: Stephen Dade

MacOS Maintainer: Rhys Mainwaring [7]

2.4.3 Dronekit-sitl

The SITL (Software In The Loop) simulator allows you to create and test DroneKit-Python apps without a real vehicle (and from the comfort of your own developer desktop!). SITL can run natively on Linux (x86 architecture only), Mac and Windows, or within a virtual machine. It can be installed on the same computer as DroneKit, or on another computer on the same network. DroneKit-SITL is the simplest, fastest and easiest way to run SITL on Windows, Linux (x86 architecture only), or Mac OS X. It is installed from Python's pip tool on all platforms, and works by downloading and running pre-built vehicle binaries that are appropriate for the host operating system [8].

2.4.4 Pymavlink

Pymavlink is a powerful and versatile Python library that plays a crucial role in the field of robotics and unmanned systems, particularly in the realm of MAVLink communication. Short for Micro Air Vehicle Link, MAVLink is a lightweight communication protocol designed to facilitate the exchange of data between unmanned aerial vehicles (UAVs) and ground control stations (GCS), making it a cornerstone technology in the world of drones.

Pymavlink serves as an interface for developers, enabling them to encode, decode, and manipulate MAVLink messages effortlessly using Python, a language celebrated for its readability and ease of use. With Pymavlink, users can craft custom scripts, ground control

software, and onboard applications that communicate seamlessly with MAVLink-enabled devices.

This library provides comprehensive support for MAVLink versions 1.0 and 2.0, ensuring compatibility with a wide range of drones and GCS platforms. It grants access to the full spectrum of MAVLink messages, allowing developers to send commands, receive telemetry data, and monitor the status of UAVs in real-time.

Pymavlink's flexibility, extensive documentation, and active open-source community make it an indispensable tool for drone enthusiasts, researchers, and professionals. Whether it's for prototyping a new flight control algorithm, building a ground control station, or simply exploring the world of unmanned systems, Pymavlink empowers developers to harness the potential of MAVLink and shape the future of autonomous aerial vehicles [9].

2.4.5 Wireshark

Wireshark is a free and open-source packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education. Originally named Ethereal, the project was renamed Wireshark in May 2006 due to trademark issues.

Wireshark is cross-platform, using the Qt widget toolkit in current releases to implement its user interface, and using pcap to capture packets; it runs on Linux, macOS, BSD, Solaris, some other Unix-like operating systems, and Microsoft Windows. There is also a terminal-based (non-GUI) version called TShark. Wireshark, and the other programs distributed with it such as TShark, are free software, released under the terms of the GNU General Public License version 2 or any later version. Wireshark lets the user put network interface controllers into promiscuous mode (if supported by the network interface controller), so they can see all the traffic visible on that interface including unicast traffic not sent to that network interface controller's MAC address.

However, when capturing with a packet analyzer in promiscuous mode on a port on a network switch, not all traffic through the switch is necessarily sent to the port where the capture is done, so capturing in promiscuous mode is not necessarily sufficient to see all network traffic. Port mirroring or various network taps extend capture to any point on the network. Simple passive taps are extremely resistant to tampering.

In this project we are going to use Wireshark to capture packets on localhost and verify the structure of the theoretical MAVLink frame which is discussed later [10].

3 Implementation and Results

3.1.1 Setting up a simulated drone

The tool is installed (or updated) on all platforms using the command: *pip install dronekit-sitl* or a WHL file can be installed from the official package site <https://pypi.org/project/dronekit-sitl/#files>.

To run the latest version of Copter for which we have binaries (downloading the binaries if needed), you can simply call: *dronekit-sitl copter*

SITL will then start and wait for TCP connections on 127.0.0.1:5760. [8]

You can specify a particular vehicle and version, and also parameters like the home location, the vehicle model type (e.g. “quad”), etc. For example: *dronekit-sitl plane-3.3.0 --home=-35.363261, 149.165230,584,353*

There are a number of other useful arguments:

Table 3: Dronekit-sitl arguments and their functions

Arguments	Function
<i>dronekit-sitl -h</i>	List all parameters to dronekit-sitl
<i>dronekit-sitl copter -h</i>	List additional parameters for the specified vehicle (in this case "copter")
<i>dronekit-sitl --list</i>	List all available vehicles
<i>dronekit-sitl --reset</i>	Delete all downloaded vehicle binaries
<i>dronekit-sitl ./path [args...]</i>	Start SITL instance at target file location

Following is the execution of the command which simply simulates a copter.

A terminal window titled 'aadit@Aadit: ~' with standard window controls. The terminal shows the command 'dronekit-sitl copter' being executed. The output includes system information (os: linux, apm: copter, release: stable), a confirmation that SITL is downloaded and extracted, and a 'Ready to boot.' message. It then shows the execution of a specific dronekit command with various parameters, followed by status updates: 'Started model quad at -35.363261,149.165230,584,353 at speed 1.0', 'bind port 5760 for 0', 'Starting sketch 'ArduCopter'', 'Serial port 0 on TCP port 5760', 'Starting SITL input', and 'Waiting for connection'. A cursor is visible at the end of the last line.

```
aadit@Aadit:~$ dronekit-sitl copter
os: linux, apm: copter, release: stable
SITL already Downloaded and Extracted.
Ready to boot.
Execute: /home/aadit/.dronekit/sitl/copter-3.3/apm --home=-35.363261,149.165230,
584,353 --model=quad -I 0
SITL-0> Started model quad at -35.363261,149.165230,584,353 at speed 1.0
SITL-0.stderr> bind port 5760 for 0
Starting sketch 'ArduCopter'
Serial port 0 on TCP port 5760
Starting SITL input
Waiting for connection ....
█
```

Figure 9: Simulating a copter using dronekit-sitl

3.1.2 Setting up a command line GCS

Following are the commands to be executed to install mavproxy for Debian based systems (including Ubuntu, WSL, Raspian) [11]:

```
sudo apt-get install python3-dev python3-opencv python3-wxgtk4.0 python3-pip python3-matplotlib
python3-lxml python3-pygame

pip3 install PyYAMLmavproxy --user

echo 'export PATH="$PATH:$HOME/.local/bin"'>> ~/.bashrc
```

If you get a “permission denied” error message when connecting to serial devices, the user permissions may need to be changed with:

```
sudo usermod -a -G dialout<username>
```

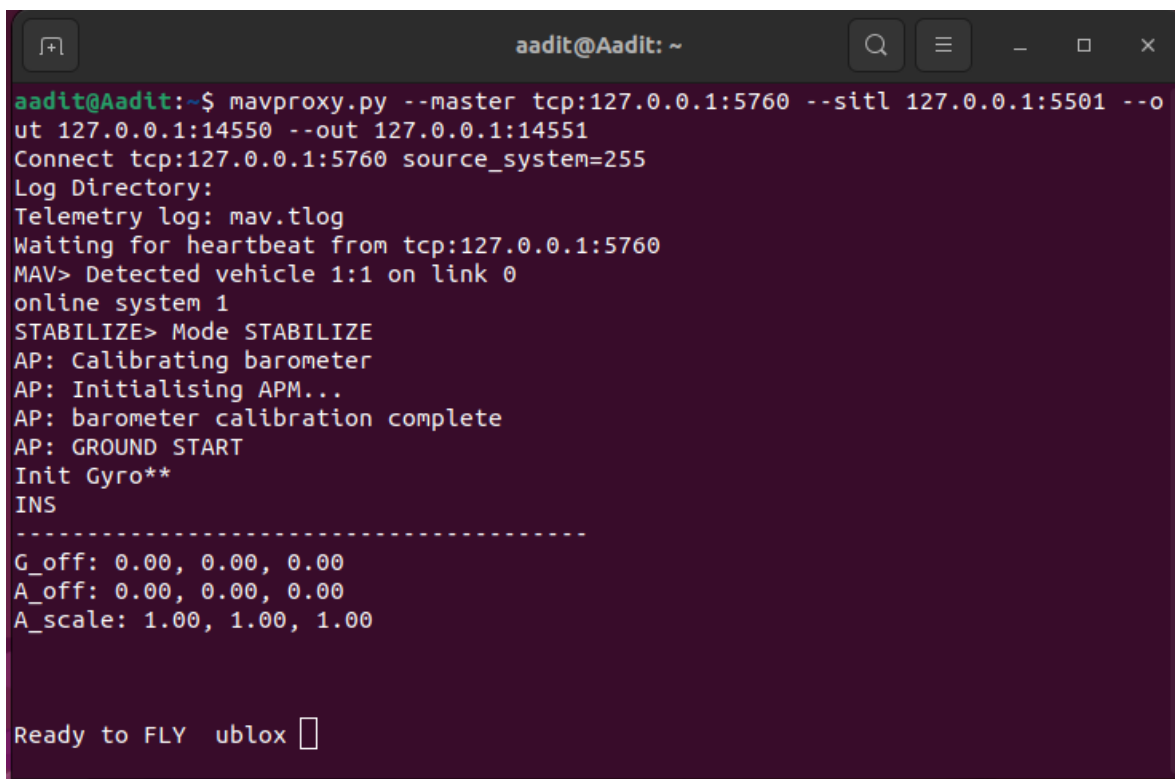
To update an existing installation with the current release:

```
pip3 install mavproxypymavlink --user --upgrade
```

To update an existing installation with the current development version (ie, from its master branch):

```
pip3 install mavproxy --user git+https://github.com/ArduPilot/mavproxy.git@master
```

Following is how to run the mavproxy GCS in the terminal:

A terminal window titled 'aadit@Aadit: ~' with search, menu, and window control icons. The terminal shows the execution of 'mavproxy.py' with various command-line arguments. The output includes connection details, log directory, telemetry log path, heartbeat waiting, vehicle detection, system status, mode setting, and sensor calibration progress. It ends with a 'Ready to FLY' message and a battery icon.

```
aadit@Aadit:~$ mavproxy.py --master tcp:127.0.0.1:5760 --sctl 127.0.0.1:5501 --o
ut 127.0.0.1:14550 --out 127.0.0.1:14551
Connect tcp:127.0.0.1:5760 source_system=255
Log Directory:
Telemetry log: mav.tlog
Waiting for heartbeat from tcp:127.0.0.1:5760
MAV> Detected vehicle 1:1 on link 0
online system 1
STABILIZE> Mode STABILIZE
AP: Calibrating barometer
AP: Initialising APM...
AP: barometer calibration complete
AP: GROUND START
Init Gyro**
INS
-----
G_off: 0.00, 0.00, 0.00
A_off: 0.00, 0.00, 0.00
A_scale: 1.00, 1.00, 1.00


Ready to FLY  ublox 
```

Figure 10: Setting up mavproxy GCS

3.1.3 Setting up the python script

The python script is user written code to perform several functions with the simulated copter.

Let's write a script containing a function for arming and takeoff.

```

from dronekit import connect, VehicleMode, LocationGlobalRelative

from pymavlink import mavutil

import time

import argparse

parser = argparse.ArgumentParser()

parser.add_argument('--connect', default='127.0.0.1:14550')

args = parser.parse_args()

# Connect to the Vehicle

print(f"Connecting to vehicle on: {args.connect}")

vehicle = connect(args.connect, baud=921600, wait_ready=True)

#921600 is the baudrate

print(f"Getting some vehicle attribute values:")

print(f" GPS: {vehicle.gps_0}")

print(f" Battery: {vehicle.battery}")

print(f" Last Heartbeat: {vehicle.last_heartbeat}")

print(f" Is Armable?: {vehicle.is_armable}")

print(f" System status: {vehicle.system_status.state}")

print(f" Mode: {vehicle.mode.name}")


# Function to arm and then takeoff to a user specified altitude

defarm_and_takeoff(aTargetAltitude):

    print ("Basic pre-arm checks")

    while not vehicle.is_armable:

        print (" Waiting for vehicle to initialise...")

        time.sleep(1)

```

```

        print ("Arming motors")

vehicle.mode = VehicleMode("GUIDED")

vehicle.armed = True

while not vehicle.armed:

    print (" Waiting for arming...")

    time.sleep(1)

    print ("Taking off!")

    vehicle.simple_takeoff(aTargetAltitude) # Take off to target altitude


# Check that vehicle has reached takeoff altitude

while True:

    print (f" Altitude: , {vehicle.location.global_relative_frame.alt}")

    #Break and return from function just below target altitude.

    if vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:

        print ("Reached target altitude")

        break

    time.sleep(1)


# Initialize the takeoff sequence to 15m

arm_and_takeoff(15)


print("Take off complete")


# Hover for 10 seconds

time.sleep(10)

```

```
print("Now let's land")
```

```
vehicle.mode = VehicleMode("LAND")
```

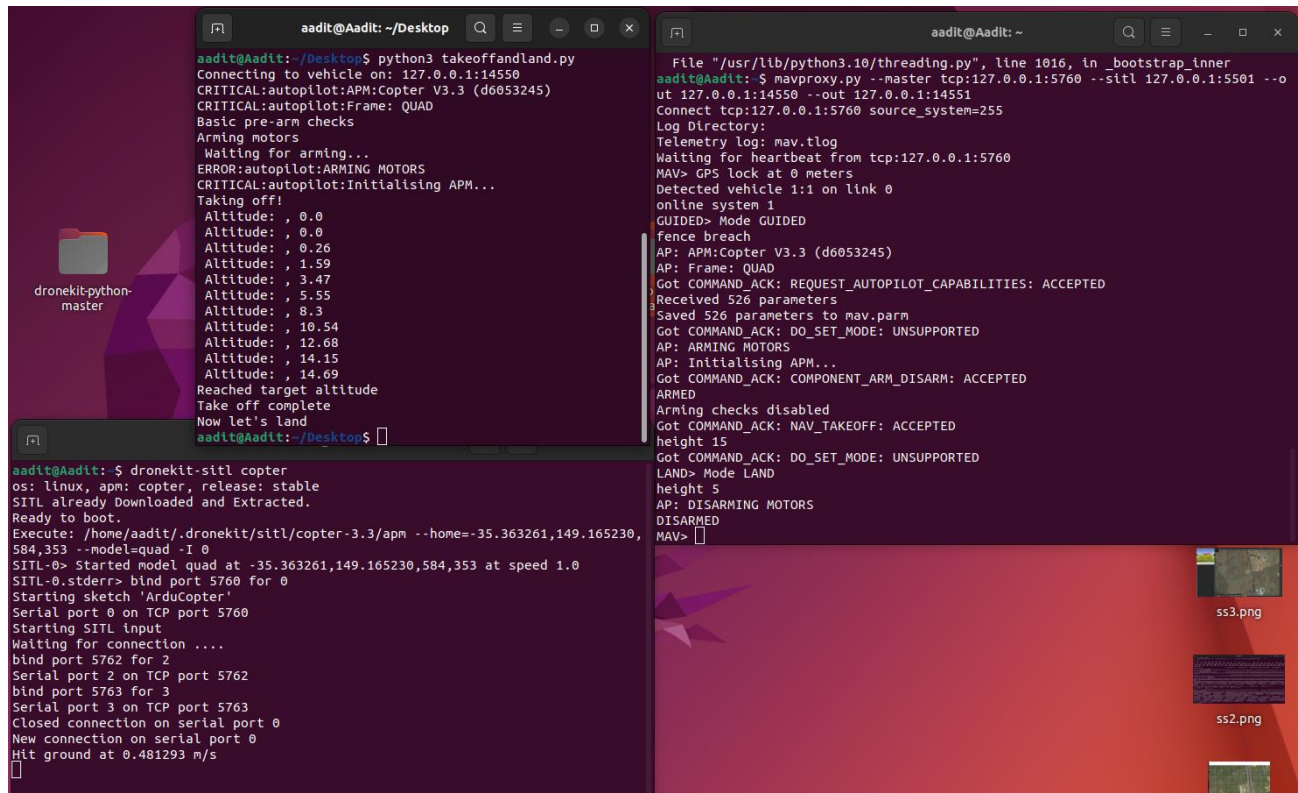
```
# Close vehicle object
```

```
vehicle.close()
```

The above script conducts several pre arm checks. If the vehicle connected is ready to arm, it is armed. Then the vehicle takes off to a user defined altitude, which in this case is 15m. Once target altitude is reached, the vehicle will hover for 10 seconds and then land. The script will be executed as any other python script in the terminal. It should be executed after the simulated copter is ready and mavproxy is already running.

3.1.4 Implementing all the three

Let's execute all that we have seen till now and see what happens.



```
aadit@Aadit: ~/Desktop
aadit@Aadit:~/Desktop$ python3 takeoffandland.py
Connecting to vehicle on: 127.0.0.1:14550
CRITICAL:autopilot:APM:Copter V3.3 (d6053245)
CRITICAL:autopilot:Frame: QUAD
Basic pre-arm checks
Arming motors
Waiting for arming...
ERROR:autopilot:ARMING MOTORS
CRITICAL:autopilot:Initialising APM...
Taking off!
Altitude: , 0.0
Altitude: , 0.0
Altitude: , 0.26
Altitude: , 1.59
Altitude: , 3.47
Altitude: , 5.55
Altitude: , 8.3
Altitude: , 10.54
Altitude: , 12.68
Altitude: , 14.15
Altitude: , 14.09
Reached target altitude
Take off complete
Now let's land
aadit@Aadit:~/Desktop$

aadit@Aadit: ~
File "/usr/lib/python3.10/threading.py", line 1016, in _bootstrap_inner
aadit@Aadit: $ mavproxy.py --master tcp:127.0.0.1:5760 --sitl 127.0.0.1:5501 --o
ut 127.0.0.1:14550 --out 127.0.0.1:14551
Connect tcp:127.0.0.1:5760 source_system=255
Log Directory:
Telemetry log: mav.tlog
Waiting for heartbeat from tcp:127.0.0.1:5760
MAV> GPS lock at 0 meters
Detected vehicle 1:1 on link 0
online system 1
GUIDED> Mode GUIDED
Fence breach
AP: APM:Copter V3.3 (d6053245)
AP: Frame: QUAD
Got COMMAND_ACK: REQUEST_AUTOPILOT_CAPABILITIES: ACCEPTED
Received 526 parameters
Saved 526 parameters to mav.parm
Got COMMAND_ACK: DO_SET_MODE: UNSUPPORTED
AP: ARMING MOTORS
AP: Initialising APM...
Got COMMAND_ACK: COMPONENT_ARM_DISARM: ACCEPTED
ARMED
Arming checks disabled
Got COMMAND_ACK: NAV_TAKEOFF: ACCEPTED
height 15
Got COMMAND_ACK: DO_SET_MODE: UNSUPPORTED
LAND> Mode LAND
height 5
AP: DISARMING MOTORS
DISARMED
MAV>

aadit@Aadit: $ dronekit-sitl copter
os: linux, apm: copter, release: stable
SITL already Downloaded and Extracted.
Ready to boot.
Execute: /home/aadit/.dronekit/sitl/copter-3.3/apm --home=-35.363261,149.165230,
584,353 --model=quad -I 0
SITL-0> Started model quad at -35.363261,149.165230,584,353 at speed 1.0
SITL-0.stderr> bind port 5760 for 0
Starting sketch 'Arducopter'
Serial port 0 on TCP port 5760
Starting SITL input
Waiting for connection ....
bind port 5762 for 2
Serial port 2 on TCP port 5762
bind port 5763 for 3
Serial port 3 on TCP port 5763
Closed connection on serial port 0
New connection on serial port 0
Hit ground at 0.481293 m/s
```

Figure 11:Implementing all the three together.

As we can see, we successfully conduct the pre-arm checks, arm the vehicle and takeoff. We reach a height of 14.69m and then hover. Then we begin to land and hit the ground at 0.481293 m/s. The simulated copter was connected to the mavproxy GCS on port TCP: 5760 and mavproxy was connected to the python script at UDP: 14550.

3.1.5 Verification of MAVLink Frames

First, let's see the sequence of sending and receiving frames that is being followed by the system.

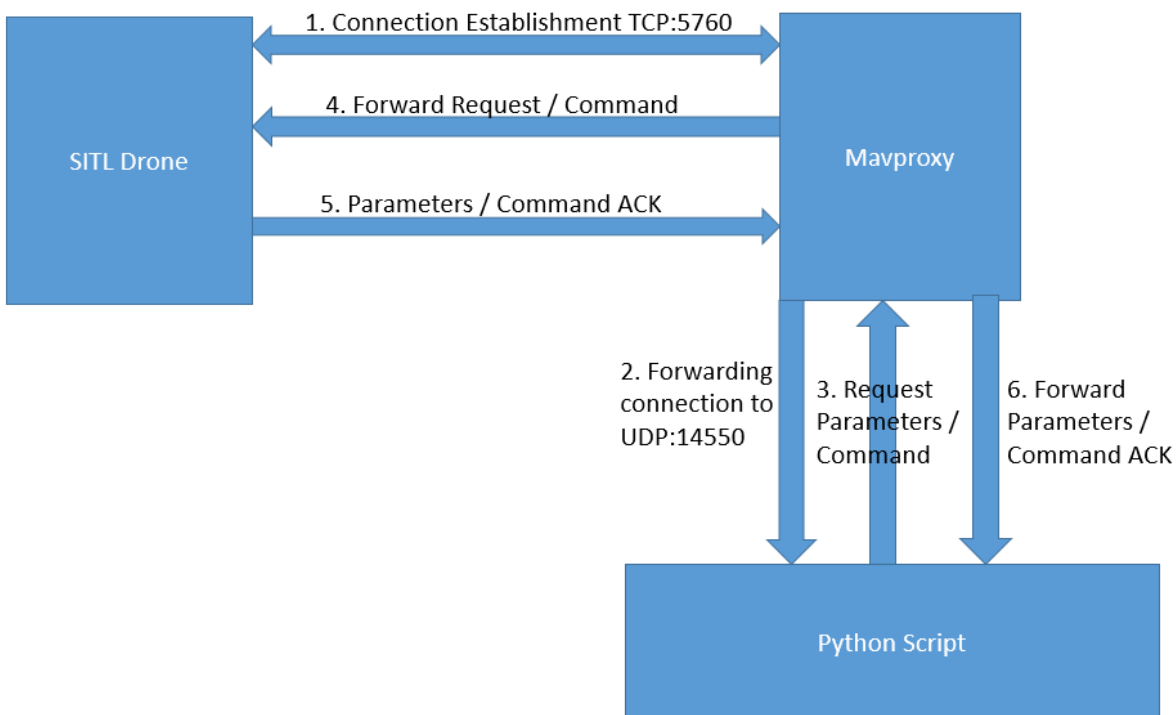


Figure 12: Sequential sending and receiving of frames

Let's try to see the binary data sent between these software's and the python script and compare it to the theoretical MAVLink frame that we saw earlier. We will be using Wireshark, which is a software designed for many functions one of which is capturing packets. We will use it to

Following are two of the frames sent from Drone to Mavproxy and Mavproxy to Drone for establishing connection on TCP:5760.



Following is a heartbeat frame. Heartbeat frame is used in the MAVLink protocol to advertise existence of a system. Heartbeat frames have a message id of 00.

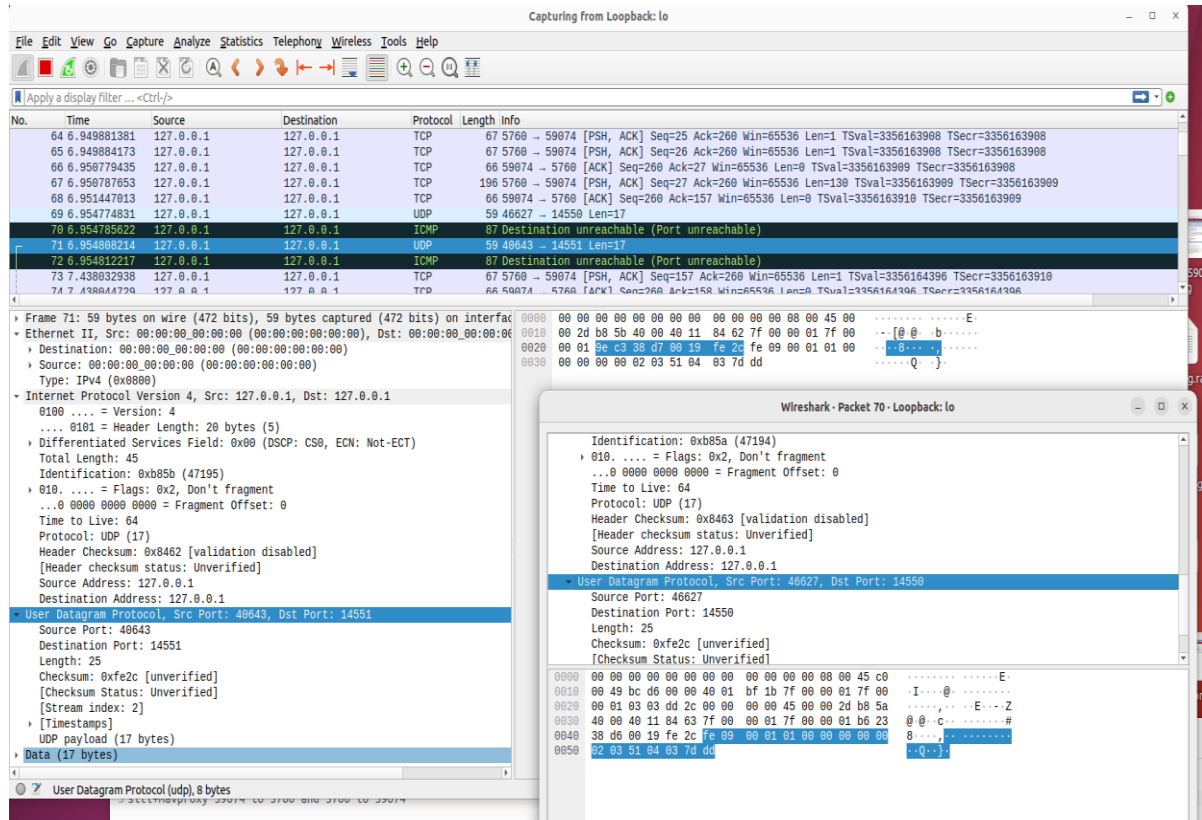
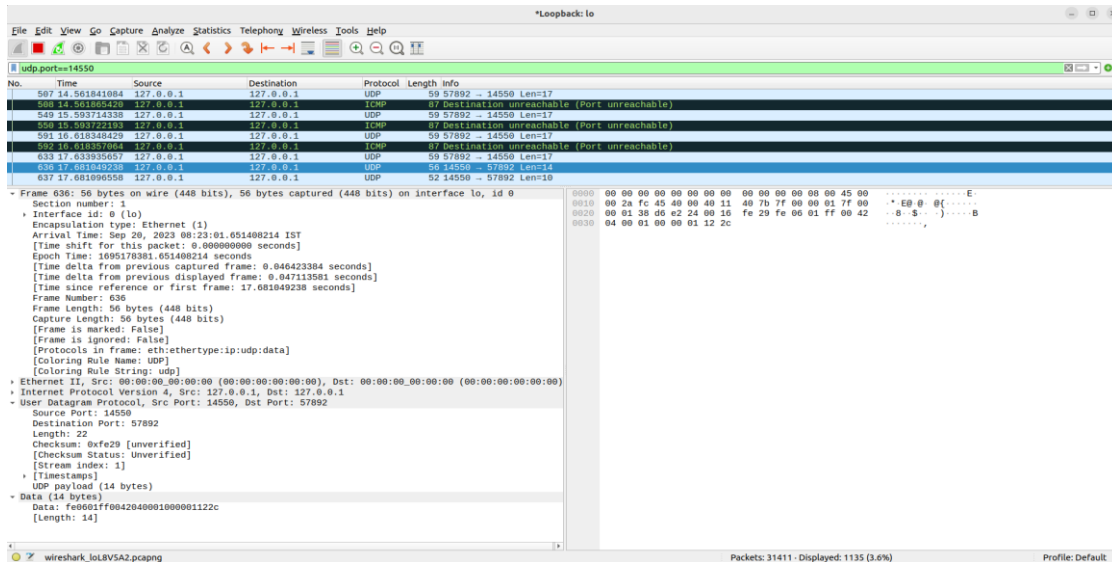


Figure 14: Heartbeat frame sent to python script

Table 4: Breakdown of the Heartbeat frame

FIELD	FUNCTION
STX	FE
LEN	09
SEQ	00
SYS ID	01
COMP ID	01
MSG ID	0x00
PAYLOAD	000000000203510403
CHECKSUM	7DDD

Let's look at a frame sent to request parameters.



We can figure out that MAVLink v1 frame structure has been used. The frame transferred is 0xFE0601FF0042040001000001122c. Breaking down the frame according to the MAVLink v1 structure we can deduce the following information.

Table 5: Breakdown of the request data stream frame

FIELD	FUNCTION
STX	FE
LEN	06
SEQ	01
SYS ID	FF
COMP ID	00
MSG ID	0x42
PAYLOAD	040001000001
CHECKSUM	122C

A message id of 42 in hexadecimal i.e. 66 in decimal refers to a frame sent to request data stream. All the message id and their corresponding meaning can be seen in Annexure.

Now let's look at a frame carrying a command from the script. In pymavlink, `COMMAND_LONG` is the frame sent when a command is being given to the drone. It contains

a message id of 76.

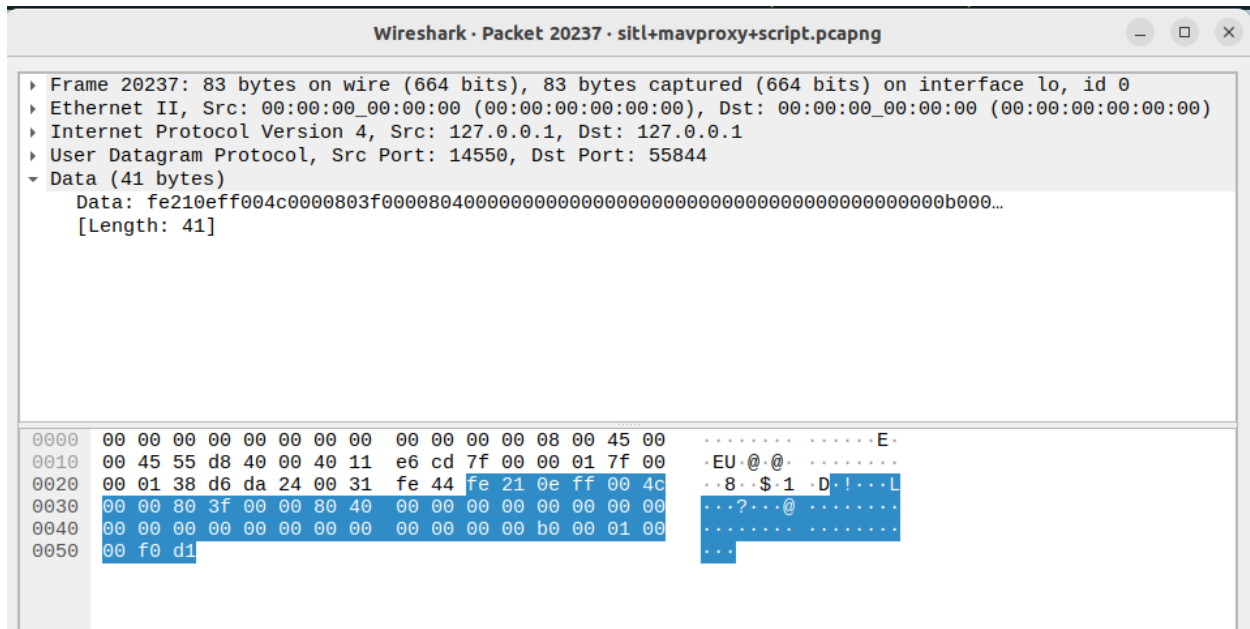


Figure 16: Frame Carrying a command

Table 6: Breakdown of a command frame

FIELD	FUNCTION
STX	FE
LEN	21
SEQ	0E
SYS ID	FF
COMP ID	00
MSG ID	0x4C
PAYLOAD	0000...010000
CHECKSUM	F0D1

Now we have seen how frames sent to request parameters or give commands look like. When one of these frames is sent to the Drone, it responds by giving the parameters or Command ACK frame. Let's have a look at how those frames look like.

Following frame has a msg id of 24 in decimal which corresponds to GPS data in MAVLink.

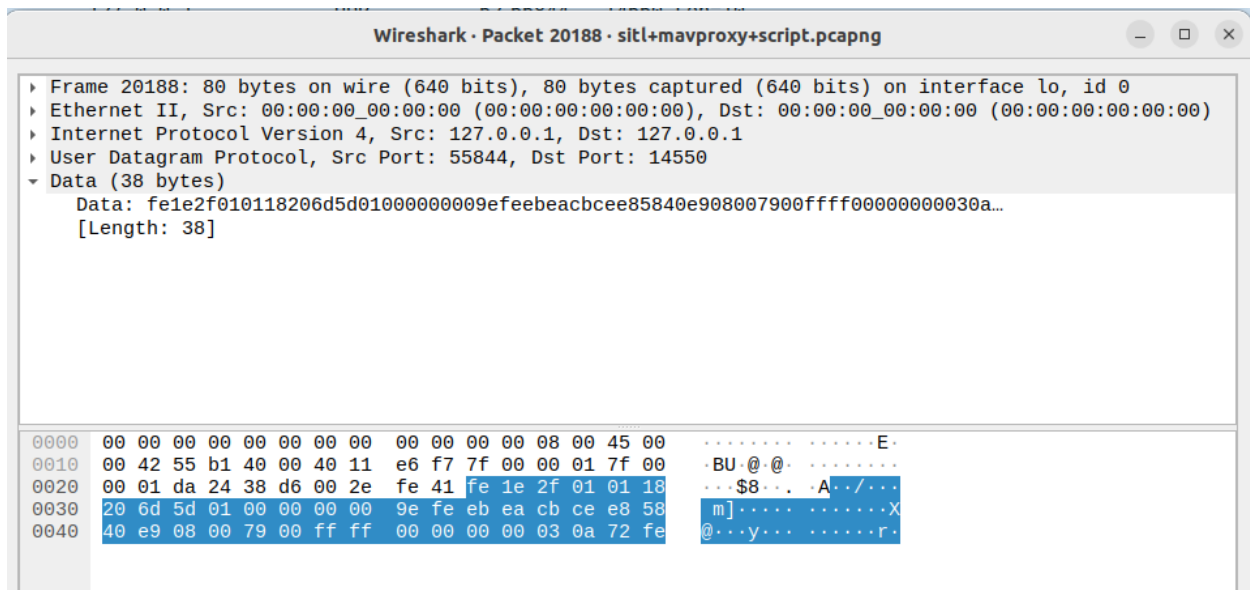


Figure 17: Frame carrying GPS data

Table 7: Breakdown of frame carrying GPS information

FIELD	FUNCTION
STX	FE
LEN	1E
SEQ	2F
SYS ID	01
COMP ID	01
MSG ID	0x18
PAYLOAD	206D...0A
CHECKSUM	72FE

Let's have a look at another frame carrying data of parameters of the Drone. The following Frame carries a msg id of 29 in decimal i.e. it is carrying data about pressure.

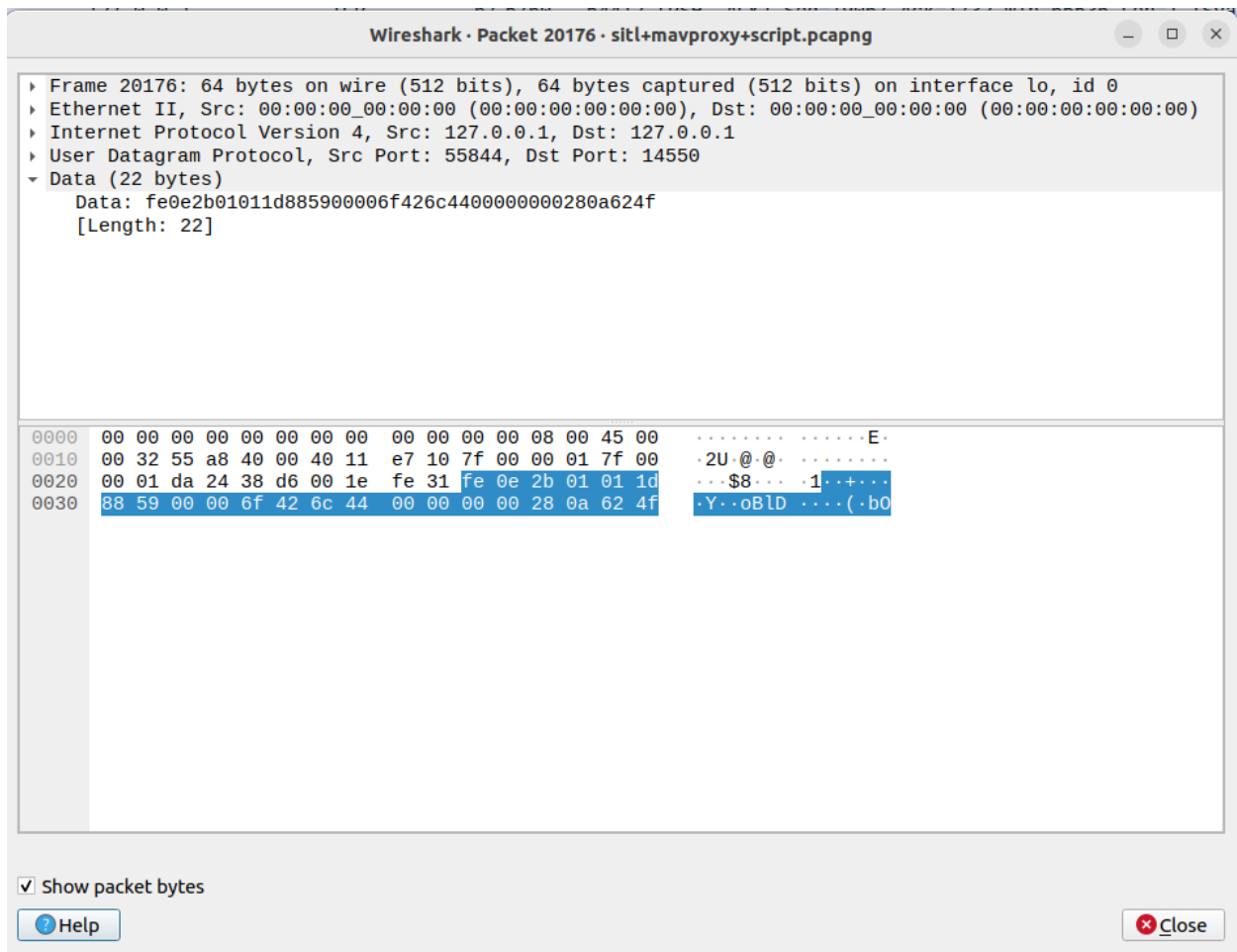


Figure 18: Frame carrying pressure data of the drone

Table 8: Breakdown of the frame carrying pressure data

FIELD	FUNCTION
STX	FE
LEN	0E
SEQ	2B
SYS ID	01
COMP ID	01
MSG ID	0x1D
PAYLOAD	8859...0A
CHECKSUM	624F

The following frame is an example of a frame carrying Inertial Measurement Unit data since it has a msg id of 27 in decimal.

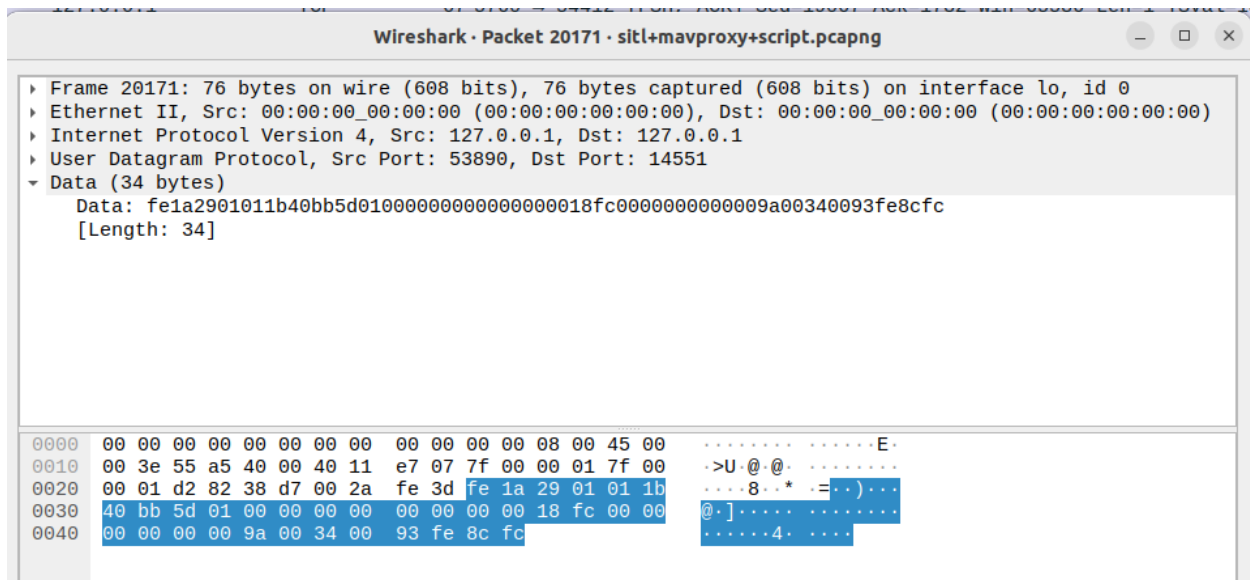


Figure 19: Frame carrying IMU Data

Table 9: Breakdown of Frame carrying IMU Data

FIELD	FUNCTION
STX	FE
LEN	1A
SEQ	29
SYS ID	01
COMP ID	01
MSG ID	0x1B
PAYLOAD	40BB...FE
CHECKSUM	8CFC

We have already seen how frames carrying data look like. Let's see how a frame carrying ACK for a command looks like.

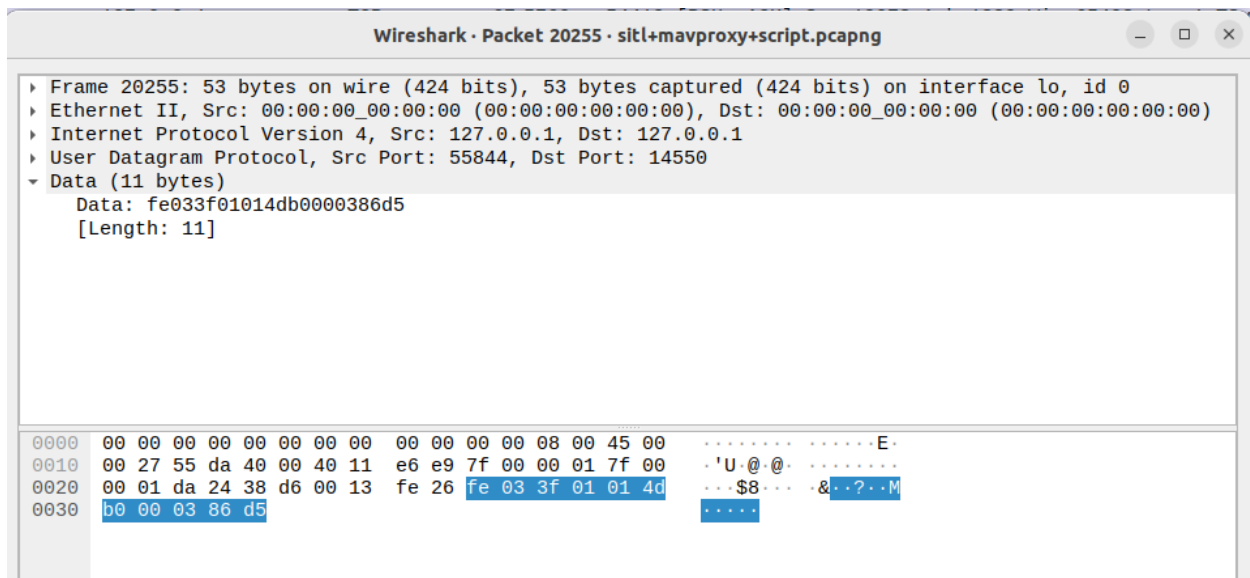


Figure 20: Frame carrying Command ACK

Table 10: Breakdown of frame carrying Command ACK

FIELD	FUNCTION
STX	FE
LEN	03
SEQ	3F
SYS ID	01
COMP ID	01
MSG ID	0x4D
PAYLOAD	B00003
CHECKSUM	86D5

Notice that the command and request originate from the same location as the command ACK and parameter data are headed i.e. 14550. That is because 14550 is the port where the python script is connected.

4 Conclusion & Future Scope

4.1 Conclusion

In conclusion, the project successfully implemented the MAVLink communication protocol for Unmanned Aerial Vehicles (UAVs). Through diligent efforts and a systematic approach, we achieved several key objectives.

1. **Selection of Simulation Tools:** The project began with the careful selection of simulation tools. We utilized Dronekit-SITL to simulate a copter, providing a realistic environment for testing and development. Additionally, MAVProxy served as a reliable Command Line Interface (CLI) ground control station which enhanced our ability to monitor UAV operations.
2. **Seamless Communication:** The implementation of the MAVLink communication protocol ensured seamless and reliable data exchange between the ground control station and the simulated UAV. This robust communication foundation is essential for the successful execution of UAV missions.
3. **Functional Script Development:** One of the key achievements of this project was the development of the "takeoffandland.py" Python script. This script allowed users to interact with the UAV. Users could input the desired altitude and initiate the arm and takeoff sequence with the execution of the script. The successful execution of this script demonstrated the practicality and user-friendliness of the system.
4. **Demonstrated Success:** The successful execution of the "arm and takeoff" function is a testament to the project's effectiveness and functionality. It showcases the ability to control a UAV in a simulated environment and paves the way for further developments in autonomous flight and mission planning.

4.2 Future Scope

While the project has achieved its primary goals, there are numerous opportunities for future enhancements. These may include:

- Developing programs for complex functions to be performed.
- Implementing a protocol on the Data-link Layer.
- Implementing a protocol on the Physical Layer.
- Developing a GUI (Graphical User Interface) to combine with the program and hence command the drone through buttons.
- Developing a mesh network of Drones.

5 References

- [1] "CFD Flow Engineering," [Online]. Available: <https://cfdflowengineering.com/working-principle-and-components-of-drone/>.
- [2] "The most common method of sight," [Online]. Available: [https://www.unmannedsystemstechnology.com/expo/drone-communications/#:~:text=Radio&text=The%20most%20common%20method%20of,of%20D%20sight%20\(LOS\)](https://www.unmannedsystemstechnology.com/expo/drone-communications/#:~:text=Radio&text=The%20most%20common%20method%20of,of%20D%20sight%20(LOS).).
- [3] "MAVLink," [Online]. Available: <https://mavlink.io/en/>.
- [4] "ISM Bands around world," [Online]. Available: <https://resources.altium.com/p/ism-bands-around-world>.
- [5] S. Haykins, Digital Communication Systems, Wiley.
- [6] "Overview," [Online]. Available: <https://dronekit-python.readthedocs.io/en/latest/about/overview.html>.
- [7] "Mavproxy Developer GCS," [Online]. Available: <https://ardupilot.org/dev/docs/mavproxy-developer-gcs.html>.
- [8] "SITL Setup," [Online]. Available: https://dronekit-python.readthedocs.io/en/latest/develop/sitl_setup.html.
- [9] "Mavgen Python," [Online]. Available: https://mavlink.io/en/mavgen_python/.
- [10] "Wireshark," [Online]. Available: <https://en.wikipedia.org/wiki/Wireshark>.
- [11] "Mavproxy Download and Installation," [Online]. Available: https://ardupilot.org/mavproxy/docs/getting_started/download_and_installation.html.

6 Annexure

6.1 MAVLink Message ID and their corresponding messages

MAVLink Message	Message id
MAVLINK_MSG_ID_HEARTBEAT	0
MAVLINK_MSG_ID_SYS_STATUS	1
MAVLINK_MSG_ID_SYSTEM_TIME	2
MAVLINK_MSG_ID_PING	4
MAVLINK_MSG_ID_CHANGE_OPERATOR_CONTROL	5
MAVLINK_MSG_ID_CHANGE_OPERATOR_CONTROL_ACK	6
MAVLINK_MSG_ID_AUTH_KEY	7
MAVLINK_MSG_ID_SET_MODE	11
MAVLINK_MSG_ID_PARAM_REQUEST_READ	20
MAVLINK_MSG_ID_PARAM_REQUEST_LIST	21
MAVLINK_MSG_ID_PARAM_VALUE	22
MAVLINK_MSG_ID_PARAM_SET	23
MAVLINK_MSG_ID_GPS_RAW_INT	24
MAVLINK_MSG_ID_GPS_STATUS	25
MAVLINK_MSG_ID_SCALED_IMU	26
MAVLINK_MSG_ID_RAW_IMU	27
MAVLINK_MSG_ID_RAW_PRESSURE	28
MAVLINK_MSG_ID_SCALED_PRESSURE	29
MAVLINK_MSG_ID_ATTITUDE	30
MAVLINK_MSG_ID_ATTITUDE_QUATERNION	31
MAVLINK_MSG_ID_LOCAL_POSITION_NED	32
MAVLINK_MSG_ID_GLOBAL_POSITION_INT	33
MAVLINK_MSG_ID_RC_CHANNELS_SCALED	34
MAVLINK_MSG_ID_RC_CHANNELS_RAW	35
MAVLINK_MSG_ID_SERVO_OUTPUT_RAW	36
MAVLINK_MSG_ID_MISSION_REQUEST_PARTIAL_LIST	37

MAVLINK_MSG_ID_MISSION_WRITE_PARTIAL_LIST	38
MAVLINK_MSG_ID_MISSION_ITEM	39
MAVLINK_MSG_ID_MISSION_REQUEST	40
MAVLINK_MSG_ID_MISSION_SET_CURRENT	41
MAVLINK_MSG_ID_MISSION_CURRENT	42
MAVLINK_MSG_ID_MISSION_REQUEST_LIST	43
MAVLINK_MSG_ID_MISSION_CLEAR_ALL	45
MAVLINK_MSG_ID_MISSION_ITEM_REACHED	46
MAVLINK_MSG_ID_MISSION_ACK	47
MAVLINK_MSG_ID_SET_GPS_GLOBAL_ORIGIN	48
MAVLINK_MSG_ID_GPS_GLOBAL_ORIGIN	49
MAVLINK_MSG_ID_PARAM_MAP_RC	50
MAVLINK_MSG_ID_SAFETY_SET_ALLOWED_AREA	54
MAVLINK_MSG_ID_SAFETY_ALLOWED_AREA	55
MAVLINK_MSG_ID_ATTITUDE_QUATERNION_COV	61
MAVLINK_MSG_ID_NAV_CONTROLLER_OUTPUT	62
MAVLINK_MSG_ID_GLOBAL_POSITION_INT_COV	63
MAVLINK_MSG_ID_LOCAL_POSITION_NED_COV	64
MAVLINK_MSG_ID_RC_CHANNELS	65
MAVLINK_MSG_ID_REQUEST_DATA_STREAM	66
MAVLINK_MSG_ID_DATA_STREAM	67
MAVLINK_MSG_ID_MANUAL_CONTROL	69
MAVLINK_MSG_ID_RC_CHANNELS_OVERRIDE	70
MAVLINK_MSG_ID_MISSION_ITEM_INT	73
MAVLINK_MSG_ID_VFR_HUD	74
MAVLINK_MSG_ID_COMMAND_INT	75
MAVLINK_MSG_ID_COMMAND_LONG	76
MAVLINK_MSG_ID_COMMAND_ACK	77
MAVLINK_MSG_ID_MANUAL_SETPOINT	81
MAVLINK_MSG_ID_SET_ATTITUDE_TARGET	82
MAVLINK_MSG_ID_SET_POSITION_TARGET_LOCAL_NED	84
MAVLINK_MSG_ID_POSITION_TARGET_LOCAL_NED	85
MAVLINK_MSG_ID_SET_POSITION_TARGET_GLOBAL_INT	86

MAVLINK_MSG_ID_POSITION_TARGET_GLOBAL_INT	87
MAVLINK_MSG_ID_LOCAL_POSITION_NED_SYSTEM_GLOBAL_OFFSET	89
MAVLINK_MSG_ID_HIL_STATE	90
MAVLINK_MSG_ID_HIL_CONTROLS	91
MAVLINK_MSG_ID_HIL_RC_INPUTS_RAW	92
MAVLINK_MSG_ID_OPTICAL_FLOW	100
MAVLINK_MSG_ID_GLOBAL_VISION_POSITION_ESTIMATE	101
MAVLINK_MSG_ID_VISION_POSITION_ESTIMATE	102
MAVLINK_MSG_ID_VISION_SPEED_ESTIMATE	103
MAVLINK_MSG_ID_VICON_POSITION_ESTIMATE	104
MAVLINK_MSG_ID_HIGHRES_IMU	105
MAVLINK_MSG_ID_OPTICAL_FLOW_RAD	106
MAVLINK_MSG_ID_HIL_SENSOR	107
MAVLINK_MSG_ID_SIM_STATE	108
MAVLINK_MSG_ID_RADIO_STATUS	109
MAVLINK_MSG_ID_FILE_TRANSFER_PROTOCOL	110
MAVLINK_MSG_ID_TIMESYNC	111
MAVLINK_MSG_ID_CAMERA_TRIGGER	112
MAVLINK_MSG_ID_HIL_GPS	113
MAVLINK_MSG_ID_HIL_OPTICAL_FLOW	114
MAVLINK_MSG_ID_HIL_STATE_QUATERNION	115
MAVLINK_MSG_ID_SCALED_IMU2	116
MAVLINK_MSG_ID_LOG_REQUEST_LIST	117
MAVLINK_MSG_ID_LOG_ENTRY	118
MAVLINK_MSG_ID_LOG_REQUEST_DATA	119
MAVLINK_MSG_ID_LOG_DATA	120
MAVLINK_MSG_ID_LOG_ERASE	121
MAVLINK_MSG_ID_LOG_REQUEST_END	122
MAVLINK_MSG_ID_GPS_INJECT_DATA	123
MAVLINK_MSG_ID_GPS2_RAW	124
MAVLINK_MSG_ID_POWER_STATUS	125
MAVLINK_MSG_ID_SERIAL_CONTROL	126
MAVLINK_MSG_ID_GPS_RTK	127

MAVLINK_MSG_ID_GPS2_RTK	128
MAVLINK_MSG_ID_SCALED_IMU3	129
MAVLINK_MSG_ID_DATA_TRANSMISSION_HANDSHAKE	130
MAVLINK_MSG_ID_ENCAPSULATED_DATA	131
MAVLINK_MSG_ID_DISTANCE_SENSOR	132
MAVLINK_MSG_ID_TERRAIN_REQUEST	133
MAVLINK_MSG_ID_TERRAIN_DATA	134
MAVLINK_MSG_ID_TERRAIN_CHECK	135
MAVLINK_MSG_ID_TERRAIN_REPORT	136
MAVLINK_MSG_ID_SCALED_PRESSURE2	137
MAVLINK_MSG_ID_ATT_POS_MOCAP	138
MAVLINK_MSG_ID_SET_ACTUATOR_CONTROL_TARGET	139
MAVLINK_MSG_ID_ACTUATOR_CONTROL_TARGET	140
MAVLINK_MSG_ID_ALTITUDE	141
MAVLINK_MSG_ID_RESOURCE_REQUEST	142
MAVLINK_MSG_ID_SCALED_PRESSURE3	143
MAVLINK_MSG_ID_CONTROL_SYSTEM_STATE	146
MAVLINK_MSG_ID_BATTERY_STATUS	147
MAVLINK_MSG_ID_AUTOPILOT_VERSION	148
MAVLINK_MSG_ID_LANDING_TARGET	149
MAVLINK_MSG_ID_SENSOR_OFFSETS	150
MAVLINK_MSG_ID_SET_MAG_OFFSETS	151
MAVLINK_MSG_ID_MEMINFO	152
MAVLINK_MSG_ID_AP_ADC	153
MAVLINK_MSG_ID_DIGICAM_CONFIGURE	154
MAVLINK_MSG_ID_DIGICAM_CONTROL	155
MAVLINK_MSG_ID_MOUNT_CONFIGURE	156
MAVLINK_MSG_ID_MOUNT_CONTROL	157
MAVLINK_MSG_ID_MOUNT_STATUS	158
MAVLINK_MSG_ID_FENCE_POINT	160
MAVLINK_MSG_ID_FENCE_FETCH_POINT	161
MAVLINK_MSG_ID_FENCE_STATUS	162
MAVLINK_MSG_ID_AHRS	163

MAVLINK_MSG_ID_SIMSTATE	164
MAVLINK_MSG_ID_HWSTATUS	165
MAVLINK_MSG_ID_RADIO	166
MAVLINK_MSG_ID_LIMITS_STATUS	167
MAVLINK_MSG_ID_WIND	168
MAVLINK_MSG_ID_DATA16	169
MAVLINK_MSG_ID_DATA32	170
MAVLINK_MSG_ID_DATA64	171
MAVLINK_MSG_ID_DATA96	172
MAVLINK_MSG_ID_RANGEFINDER	173
MAVLINK_MSG_ID_AIRSPEED_AUTOCAL	174
MAVLINK_MSG_ID_RALLY_POINT	175
MAVLINK_MSG_ID_RALLY_FETCH_POINT	176
MAVLINK_MSG_ID_COMPASSMOT_STATUS	177
MAVLINK_MSG_ID_AHRS2	178
MAVLINK_MSG_ID_CAMERA_STATUS	179
MAVLINK_MSG_ID_CAMERA_FEEDBACK	180
MAVLINK_MSG_ID_BATTERY2	181
MAVLINK_MSG_ID_AHRS3	182
MAVLINK_MSG_ID_AUTOPILOT_VERSION_REQUEST	183
MAVLINK_MSG_ID_LED_CONTROL	186
MAVLINK_MSG_ID_MAG_CAL_PROGRESS	191
MAVLINK_MSG_ID_MAG_CAL_REPORT	192
MAVLINK_MSG_ID_EKF_STATUS_REPORT	193
MAVLINK_MSG_ID_PID_TUNING	194
MAVLINK_MSG_ID_GIMBAL_REPORT	200
MAVLINK_MSG_ID_GIMBAL_CONTROL	201
MAVLINK_MSG_ID_GIMBAL_RESET	202
MAVLINK_MSG_ID_GIMBAL_AXIS_CALIBRATION_PROGRESS	203
MAVLINK_MSG_ID_GIMBAL_SET_HOME_OFFSETS	204
MAVLINK_MSG_ID_GIMBAL_HOME_OFFSET_CALIBRATION_RESULT	205
MAVLINK_MSG_ID_GIMBAL_SET_FACTORY_PARAMETERS	206
MAVLINK_MSG_ID_GIMBAL_FACTORY_PARAMETERS_LOADED	207

MAVLINK_MSG_ID_GIMBAL_ERASE_FIRMWARE_AND_CONFIG	208
MAVLINK_MSG_ID_GIMBAL_PERFORM_FACTORY_TESTS	209
MAVLINK_MSG_ID_GIMBAL_REPORT_FACTORY_TESTS_PROGRESS	210
MAVLINK_MSG_ID_GOPRO_POWER_ON	215
MAVLINK_MSG_ID_GOPRO_POWER_OFF	216
MAVLINK_MSG_ID_GOPRO_COMMAND	217
MAVLINK_MSG_ID_GOPRO_RESPONSE	218
MAVLINK_MSG_ID_RPM	226
MAVLINK_MSG_ID_VIBRATION	241
MAVLINK_MSG_ID_HOME_POSITION	242
MAVLINK_MSG_ID_SET_HOME_POSITION	243
MAVLINK_MSG_ID_MESSAGE_INTERVAL	244
MAVLINK_MSG_ID_EXTENDED_SYS_STATE	245
MAVLINK_MSG_ID_ADSB_VEHICLE	246
MAVLINK_MSG_ID_V2_EXTENSION	248
MAVLINK_MSG_ID_MEMORY_VECT	249
MAVLINK_MSG_ID_DEBUG_VECT	250
MAVLINK_MSG_ID_NAMED_VALUE_FLOAT	251
MAVLINK_MSG_ID_NAMED_VALUE_INT	252
MAVLINK_MSG_ID_STATUSTEXT	253
MAVLINK_MSG_ID_DEBUG	254
MAVLINK_MSG_ID_EXTENDED_MESSAGE	255

6.2 Python Script Containing more functions to perform

```
from dronekit import connect, VehicleMode, LocationGlobal, LocationGlobalRelative
```

```
from pymavlink import mavutil
```

```
import time
```

```
import math
```

```
import argparse
```

```
import dronekit_sitl
```

```

print("Starting simulator (SITL)")

sitl = dronekit_sitl.start_default()

connection_string = sitl.connection_string()

print(f"Connecting to vehicle on: {(connection_string,)}")

vehicle = connect(connection_string, wait_ready=True)

print(f" GPS: {vehicle.gps_0}")

print(f" Battery: {vehicle.battery}")

print(f" Last Heartbeat: {vehicle.last_heartbeat}")

print(f" Is Armable?: {vehicle.is_armable}")

print(f" System status: {vehicle.system_status.state}")

print(f" Mode: {vehicle.mode.name}")

def arm_and_takeoff(aTargetAltitude):
    """
    Arms vehicle and fly to aTargetAltitude.
    """

    print("Basic pre-arm checks")

    while not vehicle.is_armable:

        print( " Waiting for vehicle to initialise..." )

        time.sleep(1)

    print ("Arming motors" )

    vehicle.mode = VehicleMode("GUIDED")

    vehicle.armed = True

    while not vehicle.armed:

        print( " Waiting for arming..." )

        time.sleep(1)

    print("Taking off!")

```



```

vehicle.simple_takeoff(aTargetAltitude)

while True:

    print( f" Altitude: {vehicle.location.global_relative_frame.alt}")

    if vehicle.location.global_relative_frame.alt>=aTargetAltitude*0.95:

        print( "Reached target altitude" )

        break

    time.sleep(1)

"""

MAV_CMD_CONDITION_YAW - set direction of the front of the Copter

MAV_CMD_DO_SET_ROI - set direction where the camera gimbal is aimed

MAV_CMD_DO_CHANGE_SPEED - set target speed in metres/second.

"""

def condition_yaw(heading, relative=False):

    """

    This method sets an absolute heading by default, but you can set the `relative` parameter to `True` to
    set yaw relative to the current yaw heading.

    By default the yaw of the vehicle will follow the direction of travel. After setting the yaw using this
    function there is no way to return to the default yaw "follow direction of travel" behaviour

    message_factory is used to factory encode a message before using the send_mavlink function to
    transmit the message to the UAV. message_factory contains the encoded version of each message.

    Example- to encode a SET_POSITION_TARGET_LOCAL_NED message we call
    message_factory.set_position_target_local_ned_encode function.

    vehicle.message_factory.command_long_encode(target system, target component, command,
    confirmation, yaw in degrees, yaw speed deg/s, direction, relative offset 1 absolute angle 0, unused,
    unused, unused)

```

```

"""

if relative:

    is_relative = 1

else:

    is_relative = 0

msg = vehicle.message_factory.command_long_encode(0, 0,
mavutil.mavlink.MAV_CMD_CONDITION_YAW, 0, heading, 0, 1, is_relative, 0, 0, 0)

vehicle.send_mavlink(msg)

def set_roi(location):
    """

    Send MAV_CMD_DO_SET_ROI message to point camera gimbal at a specified region of interest
    (LocationGlobal).The vehicle may also turn to face the same.

    message_factory is used to factory encode a message before using the send_mavlink function to
    transmit the message to the UAV. message_factory contains the encoded version of each message.
    Example- to encode a SET_POSITION_TARGET_LOCAL_NED message we call
    message_factory.set_position_target_local_ned_encode function.

    Note that latitude, longitude and altitude are also taken as inputs for the camera gimbal to point.

    """

    msg = vehicle.message_factory.command_long_encode(0, 0,
mavutil.mavlink.MAV_CMD_DO_SET_ROI, 0, 0, 0, 0, 0, location.lat, location.lon, location.alt)

    vehicle.send_mavlink(msg)
    """

get_location_metres - Get LocationGlobal (decimal degrees) at distance (m) North & East of a given
LocationGlobal.

get_distance_metres - Get the distance between two LocationGlobal objects in metres

get_bearing - Get the bearing in degrees to a LocationGlobal

```

```

"""

def get_location_metres(original_location, dNorth, dEast):
    """
    Returns a LocationGlobal object containing the latitude/longitude `dNorth` and `dEast` metres from
    the specified `original_location`. The returned LocationGlobal has the same `alt` value
    as `original_location`.

    The function is useful when you want to move the vehicle around specifying locations relative to
    the current vehicle position.

    """
    earth_radius = 6378137.0
    dLat = dNorth/earth_radius
    dLon = dEast/(earth_radius*math.cos(math.pi*original_location.lat/180))
    newlat = original_location.lat + (dLat * 180/math.pi)
    newlon = original_location.lon + (dLon * 180/math.pi)
    if type(original_location) is LocationGlobal:
        targetlocation=LocationGlobal(newlat, newlon,original_location.alt)
    elif type(original_location) is LocationGlobalRelative:
        targetlocation=LocationGlobalRelative(newlat, newlon,original_location.alt)
    else:
        raise Exception("Invalid Location object passed")
    return targetlocation

def get_distance_metres(aLocation1, aLocation2):
    """
    Returns the ground distance in metres between two LocationGlobal objects.

    """
    dlat = aLocation2.lat - aLocation1.lat

```

```

    dlong = aLocation2.lon - aLocation1.lon

    return math.sqrt((dlat*dlat) + (dlong*dlong)) * 1.113195e5

def get_bearing(aLocation1, aLocation2):
    """
    Returns the bearing between the two LocationGlobal objects passed as parameters.

    """
    off_x = aLocation2.lon - aLocation1.lon
    off_y = aLocation2.lat - aLocation1.lat
    bearing = 90.00 + math.atan2(-off_y, off_x) * 57.2957795
    if bearing < 0:
        bearing += 360.00
    return bearing
"""

Functions to move the vehicle to a specified position (as opposed to controlling movement by setting
velocity components).

goto_position_target_global_int - Sets position using SET_POSITION_TARGET_GLOBAL_INT
command in MAV_FRAME_GLOBAL_RELATIVE_ALT_INT frame

goto_position_target_local_ned - Sets position using SET_POSITION_TARGET_LOCAL_NED command
in MAV_FRAME_BODY_NED frame

goto - A convenience function that can use Vehicle.simple_goto (default) or
goto_position_target_global_int to travel to a specific position in metres North and East from the current
location. This method reports distance to the destination.

"""

def goto_position_target_global_int(aLocation):
    """
    Send SET_POSITION_TARGET_GLOBAL_INT command to request the vehicle fly to a specified

```

LocationGlobal.

```
vehicle.message_factory.set_position_target_global_int_encode(unused, target_system,  
target_component, frame, type_mask, lat_int - X Position in WGS84 frame in 1e7 * meters , lon_int, alt, x  
velocity in NED frame, y velocity, z velocity, afx, afy, afz, yaw, yaw_rate)
```

```
"""
```

```
msg = vehicle.message_factory.set_position_target_global_int_encode(0, 0, 0,  
mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT_INT, 0b0000111111111000,  
aLocation.lat*1e7, aLocation.lon*1e7, aLocation.alt, 0, 0, 0, 0, 0, 0, 0)
```

```
vehicle.send_mavlink(msg)
```

```
def goto_position_target_local_ned(north, east, down):
```

```
"""
```

Send SET_POSITION_TARGET_LOCAL_NED command to request the vehicle fly to a specified location in the North, East, Down frame.

It is important to remember that in this frame, positive altitudes are entered as negative "Down" values. So if down is "10", this will be 10 metres below the home altitude.

type_mask (0=enable, 1=ignore).

```
set_position_target_local_ned_encode(time_boot_ms, target system, target component, frame,  
type_mask, x pos, y, z, x velocity, y, z, x acceleration, y, z, yaw, yaw_rate)
```

```
"""
```

```
msg = vehicle.message_factory.set_position_target_local_ned_encode(0, 0, 0,  
mavutil.mavlink.MAV_FRAME_LOCAL_NED, 0b0000111111111000, north, east, down, 0, 0, 0, 0, 0, 0,  
0, 0)
```

```
vehicle.send_mavlink(msg)
```

```
def goto(dNorth, dEast, gotoFunction=vehicle.simple_goto):
```

```
"""
```

Moves the vehicle to a position dNorth metres North and dEast metres East of the current position.

The method takes a function pointer argument with a single `dronekit.lib.LocationGlobal` parameter for

the target position. This allows it to be called with different position-setting commands.

By default it uses the standard method: `dronekit.lib.Vehicle.simple_goto()`.

The method reports the distance to target every two seconds.

```
"""
currentLocation = vehicle.location.global_relative_frame
targetLocation = get_location_metres(currentLocation, dNorth, dEast)
targetDistance = get_distance_metres(currentLocation, targetLocation)
gotoFunction(targetLocation)
#print "DEBUG: targetLocation: %s" % targetLocation
#print "DEBUG: targetLocation: %s" % targetDistance
while vehicle.mode.name=="GUIDED":
    remainingDistance=get_distance_metres(vehicle.location.global_relative_frame, targetLocation)
    print("Distance to target: ", remainingDistance)
    if remainingDistance<=targetDistance*0.01:
        print("Reached target")
        break
    time.sleep(2)
"""
```

Functions that move the vehicle by specifying the velocity components in each direction.

The two functions use different MAVLink commands. The main difference is

that depending on the frame used, the NED velocity can be relative to the vehicle orientation.

`send_ned_velocity` - Sets velocity components using `SET_POSITION_TARGET_LOCAL_NED` command

send_global_velocity - Sets velocity components using SET_POSITION_TARGET_GLOBAL_INT command

"""

def send_ned_velocity(velocity_x, velocity_y, velocity_z, duration):

"""

Move vehicle in direction based on specified velocity vectors and for the specified duration.

This uses the SET_POSITION_TARGET_LOCAL_NED command with a type mask enabling only velocity components.

type_mask (0=enable, 1=ignore).

set_position_target_local_ned_encode(unused, target system, target component, frame, type_mask, x pos, y pos, z pos, x vel, y vel, z vel, x acc, y acc, z acc, yaw, yaw_rate)

"""

msg = vehicle.message_factory.set_position_target_local_ned_encode(0, 0, 0, mavutil.mavlink.MAV_FRAME_LOCAL_NED, 0b0000111111000111, 0, 0, 0, velocity_x, velocity_y, velocity_z, 0, 0, 0, 0, 0, 0)

for x in range(0,duration):

vehicle.send_mavlink(msg)

time.sleep(1)

def send_global_velocity(velocity_x, velocity_y, velocity_z, duration):

"""

Move vehicle in direction based on specified velocity vectors.

This uses the SET_POSITION_TARGET_GLOBAL_INT command with type mask enabling only velocity components

*set_position_target_global_int_encode(time_boot_ms (not used), target system, target component, frame, type_mask (only speeds enabled), X Position in WGS84 frame in $1e7$ * meters, Y Position in WGS84 frame in $1e7$ * meters, Altitude in meters in AMSL altitude(not WGS84 if absolute or relative), X velocity in NED frame in m/s, Y velocity, Z velocity, afx, afy, afz acceleration, yaw, yaw_rate)*

"""

*msg = vehicle.message_factory.set_position_target_global_int_encode(0, 0, 0,
mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT_INT, 0b0000111111000111, 0,
0,0,velocity_x,velocity_y,velocity_z,0, 0, 0, 0, 0)*

for x in range(0,duration):

vehicle.send_mavlink(msg)

time.sleep(1)

vehicle.mode = VehicleMode("LAND")

print("Close vehicle object")

vehicle.close()

if sitl is not None:

sitl.stop()

print("Completed")