WORKING SCHEDULE

Date	Day	Work Done	Page No
12/08/2024	Monday	1.1 General Structure of the MAVLink Protocol	1
13/08/2024	Tuesday	1.2 Structure of MAVLink Messages	2
14/08/2024	Wednesday	1.3 MAVLink Message Types	3
15/08/2024	Thursday	2.1 Installation of Required Software and Programs	4-5
16/08/2024	Friday	2.2 Receiving Simulated Telemetry Data and Determining Connection Methods	6
19/08/2024	Monday	2.3 Implementation of Data Bridge Port Between Programs	7-9
20/08/2024	Tuesday	2.4 Creation of an Application for Decoding MAVLink Messages	10-11
21/08/2024	Wednesday	3.1 Developing and Testing Software to Analyze the Acquired Data	12-17
22/08/2024	Thursday	3.1 Developing and Testing Software to Analyze the Acquired Data	12-17
23/08/2024	Friday	3.1 Developing and Testing Software to Analyze the Acquired Data	12-17
26/08/2024	Monday	3.2 Developing and Testing Software to Re-encode the Decoded Messages	18-22
27/08/2024	Tuesday	3.2 Developing and Testing Software to Re-encode the Decoded Messages	18-22
28/08/2024	Wednesday	3.2 Developing and Testing Software to Re-encode the Decoded Messages	18-22
29/08/2024	Thursday	3.3 Application Testing and Optimization	23-25
2/09/2024	Monday	3.3 Application Testing and Optimization	23-25

3/09/2024	**Tuesday**	**4 .Simulating Telemetry Data with Randomised MAVLink Messages**	26-34
4/09/2024	**Wednesday**	**4 .Simulating Telemetry Data with Randomised MAVLink Messages**	26-34
5/09/2024	**Thursday**	**4 .Simulating Telemetry Data with Randomised MAVLink Messages**	26-34
6/09/2024	**Friday**	**4 .Simulating Telemetry Data with Randomised MAVLink Messages**	26-34
9/09/2024	**Monday**	**5.1 Reporting Project Plan and Process, Challenges and Solutions, Conclusion and Evaluation**	35-36

COMPANY AND INTERNSHIP INFORMATION

This internship was conducted at TİTRA TEKNOLOJİ A.Ş., a technology company which aims to develop solutions and products in various technology fields, both nationally and internationally. The company is actively engaged in Unmanned Systems, Smart Health, Smart Cities, Artificial Intelligence, and Big Data, striving to create innovative solutions that shape the future of these sectors.

TİTRA TEKNOLOJİ A.Ş. is located in the Akıncı Industrial Zone in Kahramankazan, which is a key industrial area within Ankara, Turkey. This location provides strategic advantages, allowing the company to collaborate closely with other leading firms in the defense and technology sectors. In the factory engineers, technicians work together to create expand the boundaries of technology.

During the internship, I was placed within the Communication Team under the R&D Department. The primary objective of the internship at TİTRA TEKNOLOJİ A.Ş. was to gain practical experience in working with the MAVLink protocol and related software development tasks. The internship involved researching and understanding the MAVLink protocol, which is crucial for communication in unmanned systems. Key activities included installing and configuring necessary software, developing and testing code for encoding and decoding MAVLink messages, and creating random telemetry data for simulation purposes. The internship aimed to enhance skills in software development, data processing, and practical application of communication protocols in the defense technology sector.

WORK DONE: 1.1 General Structure of the MAVLink Protocol		Page No 1
Start Date: 12./08/2024.	End Date: 12./08/2024.	

1. Research on MAVLink Protocol

What is MAVLink? MAVLink can be considered a "language" that facilitates communication between unmanned aerial vehicles (UAVs) and ground control stations (GCS). Similar to how people use letters to communicate, MAVLink provides specific rules and structures for UAVs to communicate securely and accurately with ground control stations. MAVLink stands for "Micro Air Vehicle Link" and is a communication protocol developed to enable data exchange between UAVs and ground control stations. This protocol is used to inform about the UAV's location, actions, and the commands it needs to receive.

Why Use MAVLink? MAVLink is used by many UAVs and ground control stations worldwide because it offers reliable, fast, and efficient communication with low bandwidth usage. This allows UAVs to perform their missions safely and ground control stations to manage these missions effectively.

1.1 General Structure of the MAVLink Protocol

The MAVLink protocol is based on a system of rules that allows multiple devices to communicate simultaneously. It defines how messages are created, sent, and received. The MAVLink protocol is based on the following core components:

1. **Messages:** MAVLink works with small-sized messages. These messages are used to perform specific tasks or transmit particular information. For instance, one message might report the UAV's current location, while another might report the status of the engine.

2. **Version:** MAVLink has different versions (e.g., MAVLink 1.0, MAVLink 2.0). New versions make the protocol more secure, flexible, and functional. For example, MAVLink 2.0 offers more advanced error control mechanisms compared to MAVLink 1.0, providing more reliable communication. These additional features result in differences in message sizes between versions, with MAVLink 1.0 ranging from 8-263 bytes and MAVLink 2.0 ranging from 12-280 bytes. Due to its growing popularity and the advantages mentioned, MAVLink 2.0 will be used in this Project.

3. **Packaging and Parsing:** Each MAVLink message is sent as a package. These packages contain the message content and control information. Packaging ensures messages are sent in a specific order and format, while parsing refers to dividing the message into smaller parts if the data is too large to be sent in one go. This allows messages to be transmitted more quickly and securely.

4. **Error Checking:** MAVLink includes mechanisms to check if messages are transmitted correctly. If an error occurs during transmission, it can be detected, and the message can be resent, ensuring secure communication.

WORK DONE: 1.2 Structure of MAVLink Messages		Page No 2
Start Date: 15./08/2024.	End Date: 13./08/2024	

1.2 Structure of MAVLink Messages

MAVLink protocol has a specific message structure for reliable data exchange between UAVs and ground control stations. MAVLink messages are processed byte by byte in a defined order. The message starts with a Packet Start byte, which indicates where the data packet begins, while sections such as checksum and signature ensure data integrity and security. Additionally, there is a sequence number to maintain the order of messages and detect any potential data loss. These structures ensure messages are transmitted quickly and reliably. Below is a table illustrating how these messages are constructed:

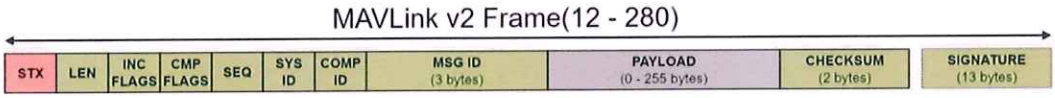

Table 1: Structure of MAVLink v2 messages

Byte Index	Content	Explanation
0	Packet Start Marker	Special marker indicating the start of the message. This byte signifies the beginning of a new packet.
1	Message Length	Indicates the length of the following payload section.
2	Incompatibility Flags	Flags that must be understood for MAVLink compliance. If flags are not understood, the message is discarded.
3	Compatibility Flags	Flags that can be ignored if not understood.
4	Packet Sequence Number	Number indicating the sequence of the message. Used for detecting message loss.
5	System ID (Sender)	Identity number of the system sending the message (e.g., an UAV).
6	Component ID (Sender)	Identity number of the component sending the message. Used to distinguish between different components in a system (e.g., autopilot and camera).
7-9	Message ID	Identifier determining the type of the message. This ID is used for decoding the message.
10-265	Main Message (Payload)	The main data of the message. Varies depending on the type and content of the message.
266-269	Checksum	Value used to verify the correct transmission of the message.
13 bytes	Signature (Optional)	Signature used for securing the message.

Table 2: Structure of MAVLink v2 messages in byte index

WORK DONE: 1.3 Types of Messages		Page No 3
Start Date: 14../.08./2024.	End Date: 14./08./2024	

1.3 Types of Messages

MAVLink protocol uses different types of messages for data transmission between flight control systems. These messages are classified based on data types and communication requirements.

Key Message Types:

- **Heartbeat:** This message confirms that the system is active and the connection is healthy. The loss of this message may indicate a problem.
- **Status:** Provides information about the current status of the flight system. This message offers insight into the system's performance and health.
- **Mission:** Used to transmit mission data. This message type is essential for mission planning and management.
- **Telemetry:** Contains flight data and sensor information. Telemetry messages provide information about the system's status during flight and sensor data.

The MAVLink protocol includes various message definitions, enumerations, and command codes to standardize data exchange between flight control systems and ground stations. Enumeration (enum) represents a data type in programming languages that defines a set of constant values. This standard message set includes general definitions managed by the MAVLink project and is expected to be implemented across all flight vehicles and ground stations. These definitions are outlined in the standard.xml file, which specifies the structure and content of messages in all MAVLink versions.

BK

2. Preparing Required Software and Programs

For this project, the following programs will be used: QGroundControl, PX4, and jMavSim.

2.1 Installation of WSL

In this project, both QGroundControl and PX4 are installed and configured on the Windows Subsystem for Linux (WSL). This is necessary because these tools are primarily designed to run on Linux environments. Windows Subsystem for Linux provides a compatibility layer for running Linux binary executables natively on Windows 10 and later versions, enabling us to leverage Linux-based software tools while operating within a Windows environment.

2.1.1 WSL Installation and Configuration

To use QGroundControl and PX4 on a Windows machine, WSL must first be installed. This can be achieved by enabling the WSL feature through the Windows Features dialog or by using PowerShell commands. Once WSL is enabled, a Linux distribution such as Ubuntu can be installed from the Microsoft Store or WSL bash commands .After installing the desired Linux distribution, initial setup involves updating package lists and installing essential packages.

Figure 1: QGroundControl

• **QGroundControl:** This is a ground control software used to monitor and configure flight control systems. It displays flight data and sends commands through a user interface. QGroundControl is used to adjust flight parameters in PX4 simulations and real flights.

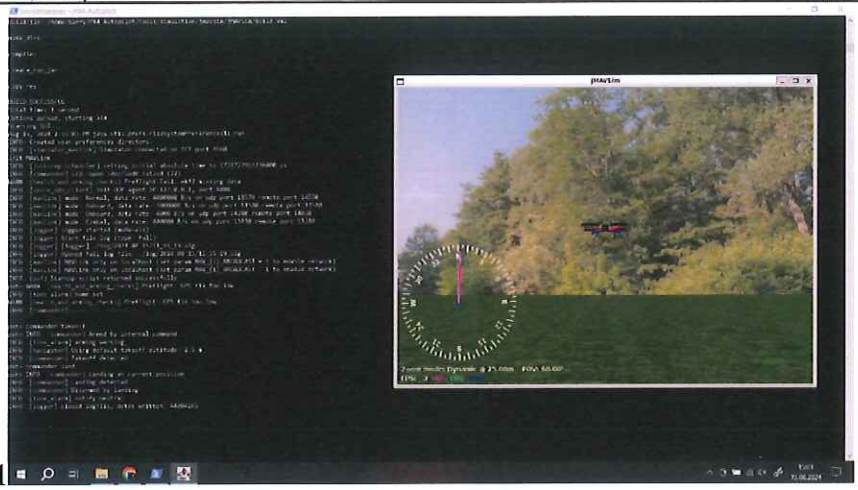

Figure 2: PX4 and jMAVSim

• **PX4:** PX4 is an open-source flight control software used for managing autonomous aerial vehicles (drones). PX4 can be tested in a simulated environment with jMAVSim and provides algorithms and controls for real flights.

• **jMAVSim:** jMAVSim is a 3D simulator that acts as a simulation tool for PX4. It performs flight simulations to mimic real flight conditions and tests system performance by exchanging data with QGroundControl.

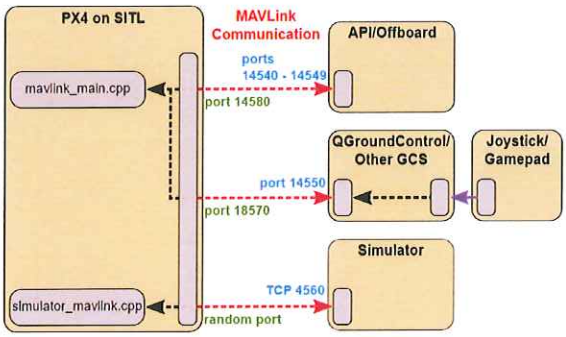

Figure 3: Diagram explaining PX4 communcation

PX4 and jMAVSim work together to generate and test real flight data in a simulation environment. QGroundControl receives, visualizes, and sends control commands based on data from PX4 and jMAVSim. This allows users to monitor and adjust the system's status during simulations and real flights.

WORK DONE: 2.2 Acquisition of Simulation Telemetry Data and Determining Connection Methods		Page No 6
Start Date: 16./08./2024	End Date: 16../02/2024	

2.2 Acquisition of Simulation Telemetry Data and Determining Connection Methods

As seen in Table 5, PX4 and QGC communicate through ports. In computer networks, ports are numerical representations of specific network connections or services. Each port is associated with a specific application or service and directs data transmission. Ports ensure the proper routing of incoming and outgoing data over the network. UDP (User Datagram Protocol) is a protocol used for data transmission over the network. UDP provides connectionless communication; that is, it does not check if the recipient has received the data after it is sent.

Why Use UDP?

- **Low Latency:** UDP is a connectionless protocol, so data is delivered to the recipient faster. This is important for applications requiring real-time data flow.
- **High Efficiency:** UDP does not include extensive control and error correction mechanisms, making data transmission more efficient and quicker.
- **Real-Time Communication:** Suitable for the quick and uninterrupted transmission of real-time information such as flight data and telemetry.

PX4 sends telemetry data through a specific UDP port. This port enables data transmission to QGroundControl during simulation or real flights. QGroundControl receives data from PX4 through this port and also sends flight data and commands back to PX4.

2.3 Implementation of Data Bridge Port Between Programs

In the project, PX4 and QGroundControl (QGC) are configured to communicate over specific UDP ports. The configuration for these ports is handled in the mavlink_main.h file located in the /PX4-Autopilot/src/modules/mavlink/ directory as can be seen in Figure 3. By default, PX4 is set to transmit data to QGC through port 5056. However, to pull data from PX4 to another application or system, it is necessary to modify this port configuration.

To change the port, the DEFAULT_REMOTE_PORT_UDP value in mavlink.h should be updated from 5056 to 5057. Before making this change, it is crucial to verify that the new port (5057) is not already in use. This verification can be performed using the netstat or ss commands in the Linux command line. The IP address for the port can be identified using the ifconfig command.
Since the project is being conducted on a single computer, the local host address '0.0.0.0' is used to facilitate communication between PX4 and QGC. This setup ensures that data transmission between the systems is routed correctly, allowing for effective testing and validation of the MAVLink protocol.

This configuration facilitates the routing and bridging of telemetry data between the PX4 flight controller and QGroundControl. It ensures that the data flow between these two systems is handled accurately and that MAVLink messages are processed correctly during data exchanges. Proper port configuration and verification are essential to avoid communication errors and ensure reliable data transmission.

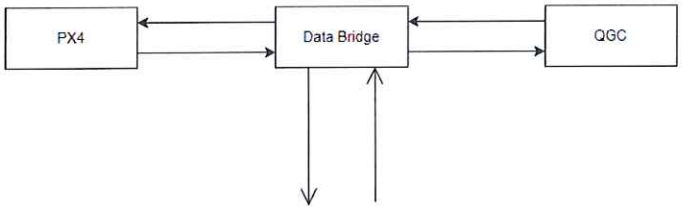

Figure 4: Diagram showing relations of data bridge to other applications

2.3.1 General Operation of the Application(data_bridge.py):

1. **Managing Data Flow:**
    - **Data Acquisition:** The application receives data from the manually modified UDP port (5057) in PX4. This port acts as the main port for listening to incoming data.
    - **Data Routing:** Routes incoming data based on its source:
        - If data comes from the QGroundControl (QGC) system, it is forwarded to the PX4 system.
        - If data comes from the PX4 system, it is routed to both the QGroundControl (QGC) system and an application that decodes MAVLink data.

- **Transmission to Target Systems:**
  - Data is routed to the correct ports (18570 for PX4 and 14550 for QGC).

Additionally, data is sent to an application processing MAVLink data (port 5058).

Benefits of the Application:

- **Data Bridging:** Provides seamless data transmission between PX4 and QGroundControl.
- **Data Monitoring:** Allows testing and verification of data transmission by monitoring the data flow.
- **Real-Time Communication:** Ensures fast routing of real-time data and telemetry information.

Key Components

1. **Socket Creation and Binding**

A socket is a software structure that enables communication between different processes over a network. It provides a standard interface for sending and receiving data between devices. In networking, a socket represents an endpoint in a two-way communication channel, typically characterized by an IP address and a port number. Sockets can be used for various types of communication, including connection-oriented (TCP) and connectionless (UDP) protocols.The application begins by creating a UDP socket using the socket library:

```
telemetry_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
telemetry_socket.bind(("0.0.0.0", bridge_port))
```

Here, socket.AF_INET specifies the use of IPv4, and socket.SOCK_DGRAM indicates that the socket is for UDP. The bind method associates the socket with a specific port (bridge_port, set to 5057) on all network interfaces ("0.0.0.0").

2. **Data Reception and Routing**

The recvfrom method is used to receive incoming data:

```
data, addr = telemetry_socket.recvfrom(1024)
```

This method reads up to 1024 bytes of data from the socket and also retrieves the address (addr) of the sender

3. **Source-Based Data Forwarding**

The critical aspect of the data routing lies in checking the source of the data:

```
if addr[1] == local_qgc_port:
    telemetry_socket.sendto(data, ("0.0.0.0", px4_local_port))
elif addr[1] == px4_local_port:
    telemetry_socket.sendto(data, ("0.0.0.0", local_qgc_port))
```

WORK DONE: 2.3 Implementation of Data Bridge Port Between Programs		Page No 9
Start Date: 19../08/2024.	End Date: 19../08/2024.	

The addr[1] component represents the source port of the received data. Depending on whether the data originates from QGC (local_qgc_port) or PX4 (px4_local_port), it is routed to the corresponding destination port.

This approach ensures that: Data from QGC is also forwarded to PX4, enabling commands and other data to be communicated correctly.

By routing data based on the source port, the system ensures reliable communication between the two components. Initially, without this setup, there were issues in sending commands from QGC to the PX4, which were resolved once the routing was correctly implemented.

WORK DONE: 2.4 Creation of an Application for Decoding MAVLink Messages		Page No 10
Start Date: 20/08/2024	End Date: 20/08/2024	

2.4 Creation of an Application for Decoding MAVLink Messages

This application is used to decode MAVLink data from PX4 systems, analyze flight data, and verify the accurate reception of this data. This application can also be used for debugging other applications in which MAVLink messages are sent.

General Operation of the Application:

1. **Establishing Data Connection:** The application establishes a connection to listen for data from the PX4 system. This connection is provided through a set UDP port (5058), used for listening to MAVLink messages.
2. **Reading Messages:** The application continuously reads incoming MAVLink messages. As messages are received, they are displayed on the screen. This is useful for checking whether the data is received and processed correctly.
3. **Asynchronous Operation:** The application uses an asynchronous structure, allowing it to perform multiple tasks simultaneously. Specifically, data reading is conducted without blocking other operations.

Benefits of the Application:

- **Real-Time Data Analysis:** The application decodes and displays MAVLink messages in real time, facilitating rapid data analysis.
- **Data Monitoring:** An effective tool for monitoring and validating incoming data. Used to check whether systems are operating correctly.
- **Asynchronous Performance:** Asynchronous operation allows for independent data reading, leading to a faster and more efficient data processing process.

When asynchronous operation is not utilized, it has been observed that not all MAVLink messages sent by PX4 can be observed. This issue is confirmed by inspecting the MAVLink Inspector tab in QGroundControl (QGC), which provides a detailed view of the MAVLink messages being transmitted and received, helping to identify any discrepancies or issues in the data flow. The issue is due to the synchronous processing model, where data reading and processing must wait for other operations to complete, which can lead to data reading being blocked and messages being lost or unobserved. In systems with high-speed data flow, there is a risk of missing or incorrect message reception. To address the issue of missed MAVLink messages when asynchronous operations are not utilized, the asyncio library in Python is employed. The asyncio library allows for efficient handling of asynchronous operations, which is crucial in scenarios where high-speed data flow is involved. The provided code snippet demonstrates the use of asyncio for reading MAVLink messages:

```
async def read_messages(connection):
    while True:
        msg = connection.recv_match(blocking=False)
        if msg:
            print(msg)
        await asyncio.sleep(0.001)
```

In this code:

The read_messages coroutine reads MAVLink messages from the connection asynchronously. The use of await asyncio.sleep(0.001) allows the program to periodically yield control, preventing it from blocking and enabling efficient handling of other tasks by not using excessive CPU. Otherwise when certain functions are not periodically slept computer tries to run it as fast as it can leading to unnecessary CPU usage.

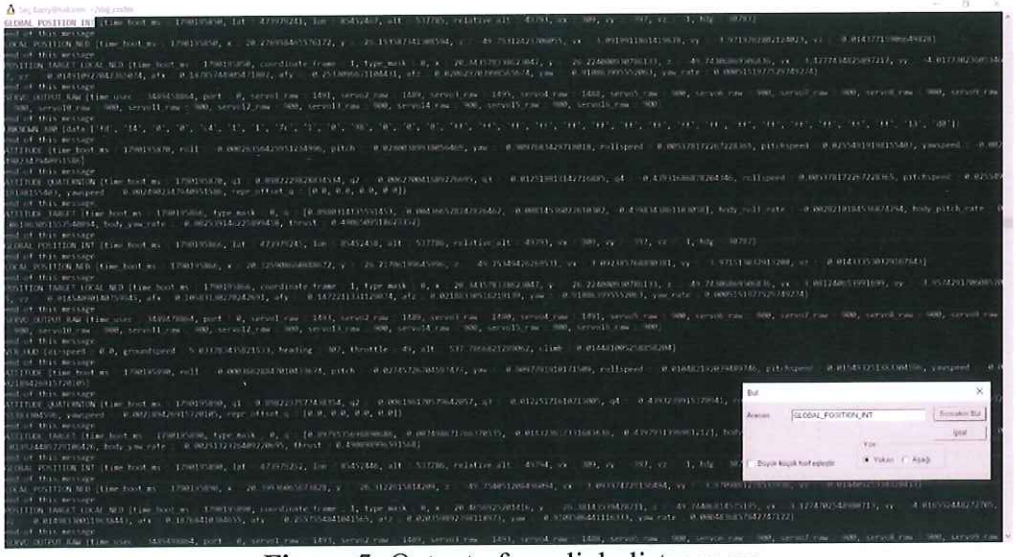

Figure 5: Output of mavlink_listener.py

The decoded MAVLink messages are in text format. These decoded messages will later be re-encoded using the MAVLink protocol and sent to QGC to test whether the encoding process is performed correctly. Thus, the capabilities for desired decoding and encoding with the MAVLink protocol will be achieved.

3. Encoding and Decoding Data Using the MAVLink Protocol

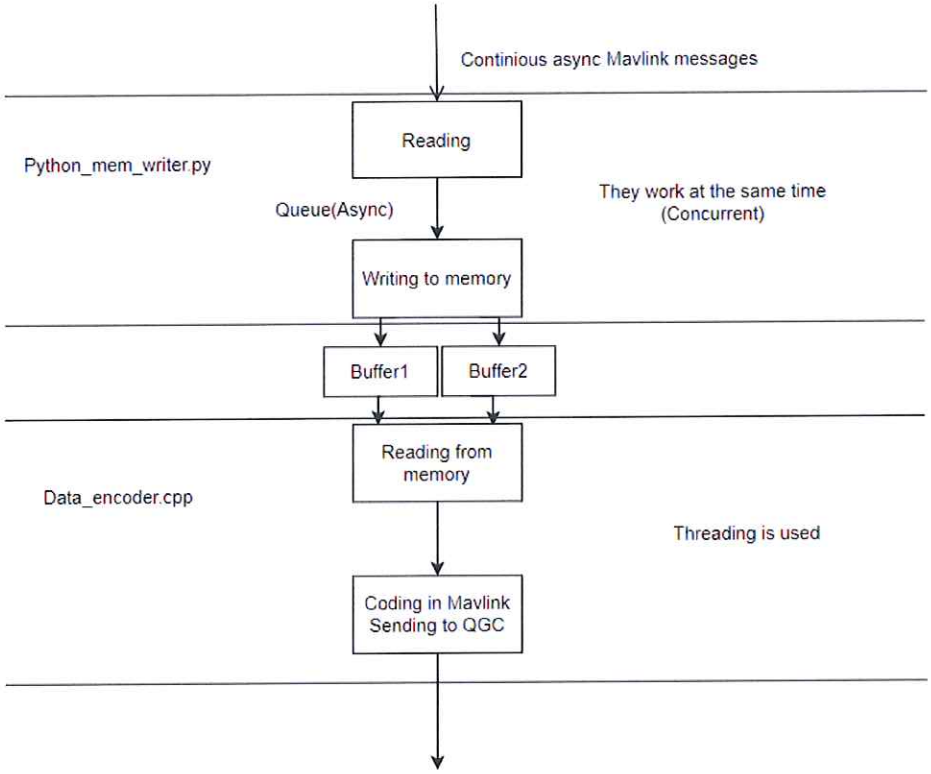

Figure 6: Diagram explaining the application

As mentioned earlier, the necessity of decoding incoming MAVLink data and then re-encoding it to send to QGC (QGroundControl) has been highlighted. MAVLink messages are fast, asynchronous, and transmitted at various frequencies. The application required to decode all incoming messages must be efficient, fast, and compatible with the asynchronous nature of the messages. Additionally, the part of the application that reads messages is written in Python, while the part that encodes messages is written in C++. This choice is due to Python's ease of development, the need to experience the interaction between two different programming languages, and the suitability of C++ code for running on devices with memory and CPU constraints, such as drones. This modular approach is anticipated to be useful for other purposes in the future.

The two scripts python_mem_writer.py and data_reader_encoder.cpp is modular and could be used in different scenarios or they can be used seperately. Firstly, they will be used in conjuction with data_bridge.py. This will provide an ideal testing ground because it will enable to send data to MAVLink data decoding,encoding application while the communication way from QGC to PX4 is unchanged.

WORK DONE: 3.1 Development and Testing of Software for Analyzing the Acquired Data		Page No 13
Start Date: 21./08./2024	End Date: 23./08./2024	

3.1 Development and Testing of Software for Analyzing the Acquired Data

In this section, we delve into the development and testing of the software responsible for analyzing the telemetry data acquired from the PX4 flight controller and processed by the python_mem_writer.py component. The python_mem_writer serves a critical role in the overall data analysis framework by efficiently managing the flow of MAVLink messages.

When developing systems where different components are implemented in distinct programming languages, it is crucial to ensure seamless compatibility and communication between these components. In this context, our system integrates applications written in Python and C++ to achieve efficient data processing and transmission. To facilitate this, several key strategies were employed:

1. Standardized Data Formats: To ensure that data exchanged between Python and C++ components is correctly interpreted, a standardized data format was utilized. While Python and C++ have different functions for memory handling,common understanding of sizes are needed. This is established by referencing the variable:

SIZE_OF_INT = struct.calcsize('i') and
BUFFER_SIZE = 140

instead of using proper Python libraries. This approach ensures that both systems can understand and process the data without discrepancies. To ensure seamless communication between Python and C++, data must first be converted into a byte format that both languages can interpret and process. This byte-level conversion standardizes the data, allowing for efficient exchange and consistent handling between the two programming environments, thereby facilitating smooth interoperability in the system.

2. Interfacing Through Shared Memory: Shared memory was chosen as the primary mechanism for inter-process communication between Python and C++ components. Reasons are:

- **Real-Time Data Processing:** The need for real-time communication between different system components necessitates a method that minimizes data transfer delays. Shared memory provides the required speed and efficiency by allowing processes to access and update data instantly.
- **High Data Volume:** Telemetry data can be voluminous and may need to be processed continuously. Shared memory allows for efficient handling of large data volumes without the overhead of traditional IPC mechanisms.
- **Inter-Process Coordination:** The application involves multiple processes, such as data acquisition and processing components, that need to work together seamlessly. Shared memory facilitates this coordination by providing a common data space where processes can exchange information without the need for intermediate data formats or conversions.

3. Signal Handling for Graceful Termination and Resource Cleanup: In the application, managing the proper shutdown and cleanup of resources(shared memory needs to freed) is crucial, especially when dealing with concurrent processes and shared memory. This is achieved through the handling of CTRL+C signals and the use of a stop flag mechanism to ensure that resources are correctly freed. This signal handling mechanism is implemented in Python because it is responsible for writing data to shared memory. If it closes first and then C++ frees the memory, future memory corruption will be prevented and graceful exit will be provided.

```
stop_flag_file = '/tmp/stop_flag'
def signal_handler(signum, frame):
    # Create a flag file to indicate termination
    with open(stop_flag_file, 'w') as f:
        f.write('stop')
        print(f"works stop file")
    sys.exit(0)
```

It creates a file in temprorary directory making it detectable to C++.

4. Two-Buffer Memory: In order to process high data volumes with speed a two-buffer system is implemented enabling pyton_mem_writer.py to write to one buffer while allowing data_reader_encoder.cpp to read other buffer thus gainig speed, eliminating the time needed for waiting each process.

5. Synchronization Mechanisms: It is important to only read or write to buffer to prevent conflicts and ensure data integrity. This includes using flags to coordinate access to shared memory, ensuring that both the Python and C++ components can safely read from and write to the shared memory without causing data corruption or inconsistency also making the code simpler by not choosing a more complex solution like locking systems. Each flag of the buffer is added to end of that buffer with the size of int. These are the resulting sizes for buffers:

```
SHM_SIZE = BUFFER_SIZE * 2 + 2 * SIZE_OF_INT  # Dual buffers + two flags
BUFFER_1_START = 0
BUFFER_2_START = BUFFER_SIZE
FLAG_1_OFFSET = BUFFER_SIZE * 2
FLAG_2_OFFSET = FLAG_1_OFFSET + SIZE_OF_INT
```

Flags are equal to 0 or 1. 0 means it is suitable for Python to write, 1 means it is suitable for C++ to read the filled buffer.

6. Concurrency and Coroutines: Concurrency refers to the ability of a system to handle multiple tasks or processes simultaneously. In the context of software development, concurrency allows a program to perform several operations at once, improving efficiency and responsiveness. In the developed application, concurrent

routines are employed to manage the simultaneous execution of data processing tasks. This is achieved through two primary routines: one for decoding MAVLink messages and another for writing data to shared memory. This approach leverages asynchronous processing to handle multiple tasks efficiently, ensuring that data is processed and transferred without significant delays or bottlenecks.

```
queue = asyncio.Queue()
await asyncio.gather(read_messages(pymav_analysis_port, queue), write_to_shared_memory(queue))
```

The queue serves as an intermediary buffer, holding data between the read_messages function (which reads MAVLink messages) and the write_to_shared_memory function (which writes data to shared memory).

Explaining the code:

```
async def main():
    try:
        signal.signal(signal.SIGINT, signal_handler)
        connection_string = 'udpin:0.0.0.0:%d' % px4_pymav_port
        pymav_analysis_port = mavutil.mavlink_connection(connection_string)
        print(f"it works until here")
        queue = asyncio.Queue()
                await asyncio.gather(read_messages(pymav_analysis_port, queue), write_to_shared_memory(queue))
        print(f"it should have never reached here")
    except Exception as e:
        print(f"Error in main function: {e}")
        sys.exit(1)
```

The main function serves as the central hub for managing the application's operations. It begins by setting up a signal handler for SIGINT (interrupt signal) to ensure graceful termination when the user sends an interrupt signal (e.g., pressing Ctrl+C). It then establishes a MAVLink connection using a UDP port specified in the connection_string, allowing the application to receive MAVLink messages from PX4. An asyncio.Queue is initialized to facilitate communication between asynchronous tasks, ensuring smooth data flow between reading MAVLink messages and writing to shared memory.

The core of the function involves concurrently executing two asynchronous tasks: read_messages and write_to_shared_memory. read_messages captures MAVLink messages from the established connection and places them into the queue, while write_to_shared_memory retrieves these messages from the queue and writes them to shared memory. The function includes error handling to catch and report exceptions, ensuring that the application can exit gracefully in case of any issues. The print statements within the function serve as debugging aids, confirming the execution flow and highlighting potential issues if unexpected behavior occurs. Now the two methods read_messages and write_to_shared_memory function will be explained.

read_messages:

The read_messages function is responsible for continuously receiving and processing MAVLink messages from the given connection. It utilizes a bytearray buffer to temporarily store incoming messages. The function operates in a loop that runs indefinitely until a termination signal is detected via the stop_flag_file. Inside the loop, it reads MAVLink messages from the connection using connection.recv_match(blocking=False). If no message is received, it briefly pauses with await asyncio.sleep(0.01) to avoid excessive CPU usage.

Upon receiving a message, the function extracts the message as bytes and performs several checks: it verifies that the message is not empty and does not exceed the predefined BUFFER_SIZE. Valid messages are added to the buffer. Once the buffer reaches the specified size, a chunk of data is placed into the queue for further processing, and the buffer is trimmed. After the loop exits, any remaining data in the buffer is also added to the queue. This process ensures that MAVLink messages are efficiently managed and transferred for subsequent operations.

write_to_shared_memory:

The write_to_shared_memory function is designed to manage writing MAVLink messages to a shared memory segment, which involves several crucial steps for handling and switching between two buffers.

The function begins by initializing the shared memory segment through the initialize_shared_memory function.

```
shm[FLAG_1_OFFSET:FLAG_1_OFFSET + SIZE_OF_INT] =
(0).to_bytes(SIZE_OF_INT, byteorder='little')
shm[FLAG_2_OFFSET:FLAG_2_OFFSET + SIZE_OF_INT] =
(0).to_bytes(SIZE_OF_INT, byteorder='little')
```

The lines above set the initial values of these flags to zero. This zero value signifies that the buffers are empty and available for writing.
The initialize_shared_memory function returns the shm object, which is a memory-mapped file object representing the shared memory space. This object is subsequently used in the write_to_shared_memory function to perform operations such as writing MAVLink data to the shared memory buffers and updating the buffer status flags. The shm object allows direct access to the memory space, facilitating efficient inter-process communication and data handling.

It then enters an infinite loop where it continuously processes data from the queue.
In each iteration of the loop, if the queue is empty, the function pauses briefly with await asyncio.sleep(0.01) to reduce CPU usage and avoid busy-waiting. When data is available in the queue, the function retrieves it and decides which buffer to use based on the buffer_switch flag. This flag determines the starting position

(BUFFER_1_START or BUFFER_2_START) and the corresponding flag offset (FLAG_1_OFFSET or FLAG_2_OFFSET) in shared memory.

Before writing, the function checks if the designated buffer is ready by examining the flag at flag_offset. It waits until this flag indicates that the buffer is available for new data. Once the buffer is ready, the function writes the MAVLink message in chunks to the current buffer. If the message size exceeds the current buffer's capacity, only the portion that fits is written, and the remaining data is directed to the next buffer. This approach ensures seamless data handling between buffers without overwriting existing data or causing data loss. Relevant code is below:

```
while bytes_written < len(mavlink_bytes):
        remaining_space = BUFFER_SIZE - (offset % BUFFER_SIZE)
        chunk_size = min(remaining_space, len(mavlink_bytes) - bytes_written)
        shm[offset:offset + chunk_size] =
mavlink_bytes[bytes_written:bytes_written + chunk_size]
        bytes_written += chunk_size
        offset = (offset + chunk_size) % BUFFER_SIZE
        if (offset % BUFFER_SIZE) == 0:
          break
```

After completing the write operation, the function sets the flag at flag_offset to indicate that the buffer is filled and ready for reading. It then toggles the buffer_switch flag to use the alternate buffer for subsequent data writes.
This approach allows for continuous data writing and reading from shared memory, with seamless switching between buffers to ensure data integrity and prevent conflicts.

3.2 Development and Testing of Software for Re-Encoding the De-Coded Messages

In this section, we explore the development and testing of the data_reader_encoder.cpp component, which is essential for handling the encoded MAVLink messages. This component is responsible for reading the decoded MAVLink data from shared memory, re-encoding it, and ensuring its proper routing for further use. By focusing on both functionality and performance, this component ensures that data integrity is maintained throughout the encoding process and that the data flow is seamless and efficient. For this, asynchronous processing and multi-threading is chosen, driven by the need to handle high volumes of telemetry data and maintain real-time performance.

ThreadSafeQueue Implementation

Our system utilizes ThreadSafeQueue, a thread-safe data structure that ensures safe concurrent access to shared data. This is crucial in a multi-threaded environment where multiple threads (or processes) need to read from or write to a common resource without causing data corruption or race conditions. By employing std::mutex and std::condition_variable, we provide synchronization mechanisms that prevent simultaneous access issues, thus maintaining the integrity of our data queues.

```
template<typename T>
void ThreadSafeQueue<T>::push(const T& value) {
   std::unique_lock<std::mutex> lock(mtx);
   cv_not_full.wait(lock, [this]() { return queue.size() < max_size; });
   queue.push(value);
   cv_not_empty.notify_one();
}
```

The push method above adds an element to the queue while ensuring that the queue does not exceed its maximum size. This method uses a std::unique_lock to manage access to the queue and a std::condition_variable to wait until there is space available. When the queue size is below the maximum, the method inserts the new element and notifies any waiting threads that space is now available.

```
template<typename T>
void ThreadSafeQueue<T>::wait_and_pop(T& value) {
   std::unique_lock<std::mutex> lock(mtx);
   cv_not_empty.wait(lock, [this]() { return !queue.empty(); });
   value = queue.front();
   queue.pop();
   cv_not_full.notify_one();
}
```

Similarly, the wait_and_pop method above removes and retrieves an element from the queue. It uses a std::unique_lock and std::condition_variable to wait until there is at least one element in the queue. Once an element is available, it is removed from the front of the queue, and the queue is notified that there is now space for new elements.

template<typename T>

In the context of the ThreadSafeQueue class, the use of templates provides the flexibility to create a thread-safe queue that can hold any type of data. By defining the class as template<typename T>, the ThreadSafeQueue becomes a generic container that can be instantiated with different types (e.g., integers, strings, or custom objects). This eliminates the need for redundant code and makes the class adaptable to various data types, enhancing code maintainability and reducing duplication.

Synchronization with python_mem_writer.py

To ensure that two programs operate synchronously, it is essential to implement synchronization mechanisms suitable with python_mem_writer.py . This is a critical step to maintain data consistency and system performance. There are two considerations for full synchronisation:

1. Shared Memory Mapping: In the context of the application, the shared memory is initially created and managed by the Python python_mem_writer component. Therefore, the primary role of the C++ component is to map this pre-existing shared memory for its own use. In Python, memory is also mapped after creation. Once the C++ program maps the shared memory using the custom-made get_shared_memory method and retrieves the pointer to the mapped memory region with void* shm_ptr = get_shared_memory();, it can then access and interact with the shared memory as needed. This approach ensures that the C++ component does not need to create or manage the shared memory itself, but rather relies on the Python component's setup.

2. Graceful Exit Mechanism: In the application, a graceful exit mechanism is implemented to ensure that all resources are properly released and that the program terminates cleanly. This is achieved through the use of a stop flag file created by python_mem_writer.py after ctrl+c is pressed on the terminal. When stop flag file is detected, while (access("/tmp/stop_flag", F_OK) != 0) loops in all threads stops working and exiting from the thread resulting in later threads joining. Then after threads joining, it checks for the file again with while loop, ensuring the validity of graceful exit. Once the stop flag is detected, the program proceeds to clean up resources:

```
std::remove("/tmp/stop_flag");
munmap(shm_ptr, SHM_SIZE);
std::cout << "Unmapping complete, exiting gracefully." << std::endl;
```

Shared memory is freed and stop flag file is destroyed for reusability of the application.

WORK DONE: 3.2 Development and Testing of Software for Re-Encoding the De-Coded Messages		Page No 20
Start Date: 26./08/2024.	End Date: 28/08/2024.	

Explanation of the main code

```
int main() {
   try {
      int sockfd = setup_socket();
      sockaddr_in qgc_addr = setup_qgc_address();
      int sndbuf_size = 8388608; // 8 MB, adjust as needed
      if (setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &sndbuf_size,
sizeof(sndbuf_size)) < 0) {   //for socket size
         perror("Setsockopt SO_SNDBUF failed");
      }
      fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

The first part of the main function focuses on setting up the communication channel between the application and QGroundControl (QGC) via a socket. This process begins with creating a UDP socket by calling the setup_socket() function. The socket serves as a network endpoint that facilitates data transmission, and the file descriptor (sockfd) returned by this function is used in subsequent network operations. Proper error handling is in place to terminate the program if socket creation fails, ensuring that any issues are immediately identified and addressed.

Following the socket creation, the program sets up the address configuration for QGC using the setup_qgc_address() function. This function populates a sockaddr_in structure with the IP address and port number of QGC, ensuring that the MAVLink messages are directed to the correct destination.
To optimize data transmission, the program adjusts the socket's send buffer size to 8 MB using the setsockopt() function. This increased buffer size accommodates larger bursts of outgoing data, which is particularly beneficial given the high-frequency nature of telemetry data transmission.
Lastly, the program configures the socket to operate in non-blocking mode using the fcntl() function. By setting the socket to non-blocking, the program allows send and receive operations to return immediately, even if they would typically block due to buffer constraints. This setup is beneficial for maintaining the overall responsiveness and efficiency of the application.

```
      void* shm_ptr = get_shared_memory();
      ThreadSafeQueue<std::vector<uint8_t>> data_queue;
      ThreadSafeQueue<std::vector<mavlink_message_t>> message_queue;
      std::thread reader(reader_thread, std::ref(data_queue), shm_ptr);
      std::thread converter(converter_thread, std::ref(data_queue),
std::ref(message_queue));
      std::thread sender(sender_thread, std::ref(message_queue), sockfd,
std::ref(qgc_addr));
      reader.join();
      converter.join();
      sender.join();
```

The next segment of the main function involves setting up shared memory access and initializing the multi-threading components that drive the data processing pipeline. The shared memory is mapped using the get_shared_memory() function, which returns a pointer (shm_ptr) to the mapped region.

Following the shared memory setup, two thread-safe queues are instantiated: data_queue and message_queue. The data_queue is designed to hold raw byte data read from the shared memory, while the message_queue stores MAVLink messages that have been parsed and are ready to be sent to QGroundControl. These queues use mutexes and condition variables to ensure that access is synchronized among threads, preventing race conditions and maintaining data integrity.To handle the flow of data through the system, three threads are created, each executing a specific task in the data processing pipeline.

```
void reader_thread(ThreadSafeQueue<std::vector<uint8_t>>& data_queue, void*
shm_ptr) {
   while (access("/tmp/stop_flag", F_OK) != 0) {
      read_and_push_buffer(shm_ptr, BUFFER_1_START, data_queue);
      //std::this_thread::sleep_for(std::chrono::milliseconds(10));
      read_and_push_buffer(shm_ptr, BUFFER_2_START, data_queue);
      //std::this_thread::sleep_for(std::chrono::milliseconds(10));
      std::cout << "it comes to here reader" << std::endl;
   }
}
```

The reader_thread as shown above is responsible for the initial stage of the data processing pipeline—acquiring raw telemetry data from the shared memory and pushing it into the data_queue. This thread continuously monitors two alternating memory buffers within the shared memory region, checking specific flags to determine when new data is available. When data is ready, the thread reads the contents of the buffer, converts the data into a std::vector<uint8_t>, and pushes it into the data_queue for further processing. After reading the data, the thread resets the buffer to zero and clears the flag, signaling that the buffer is ready for new data.

```
void converter_thread(ThreadSafeQueue<std::vector<uint8_t>>& data_queue,
ThreadSafeQueue<std::vector<mavlink_message_t>>& message_queue) {
   while (access("/tmp/stop_flag", F_OK) != 0) {
      std::vector<uint8_t> bytes;
      data_queue.wait_and_pop(bytes);
      std::vector<mavlink_message_t> messages;
      convert_bytes_to_mavlink(bytes, messages);
      message_queue.push(messages);
      std::cout << "it comes to here converter" << std::endl;

   }
}
```

The converter_thread shown above functions as the intermediate stage of the pipeline, focusing on converting raw byte data into structured MAVLink messages. It retrieves data from the data_queue using the wait_and_pop method, ensuring it waits if the queue is empty, thus synchronizing the flow of data through the pipeline. Once data is obtained, the thread processes each byte using the convert_bytes_to_mavlink function. This function parses the raw bytes, extracting complete MAVLink messages which are then batched and stored in a std::vector<mavlink_message_t>. These parsed messages are subsequently pushed into the message_queue, where they are held until ready for transmission. This conversion process is critical, as it transforms the raw telemetry data into a format that is understood by QGroundControl, ensuring that the messages are correctly formatted and transmitted.

```
void sender_thread(ThreadSafeQueue<std::vector<mavlink_message_t>>&
message_queue, int sockfd, const sockaddr_in& server_addr) {
   while (access("/tmp/stop_flag", F_OK) != 0) {
      std::vector<mavlink_message_t> messages;
      message_queue.wait_and_pop(messages);
      send_mavlink_messages(messages, sockfd, server_addr);
      std::cout << "it comes to here sender" << std::endl;

   }
}
```

The sender_thread is tasked with the final stage of the data processing pipeline—sending the MAVLink messages to QGroundControl via a pre-configured UDP socket. The thread continuously retrieves batches of MAVLink messages from the message_queue using wait_and_pop, ensuring a steady flow of data ready for transmission. It then packages these messages into a buffer using the send_mavlink_messages function, which serializes each MAVLink message and prepares the data for sending over the network. Once the buffer is populated, the thread sends the data to QGroundControl using the sendto system call.

WORK DONE: 3.3 Testing and Optimization of the the code		Page No 23
Start Date: 21/08/2024.	End Date: 02/09/2024.	

3.3 Testing and Optimization of the the code

In the initial stages of running the code, I noticed that the performance was significantly suboptimal, with the application running extremely slowly. To diagnose the issues, I employed various profiling methods available on WSL, which allowed me to pinpoint the bottlenecks and inefficiencies. For example, using gprof helped you observe that the function for sending MAVLink messages was already performing well, and you could visually track how quickly functions executed in real-time. This profiling insight guided your optimization efforts, showing where your code's performance was already adequate and where further improvements were needed. During these tests, it became evident that PX4 was consuming an excessive amount of CPU resources, leaving insufficient performance headroom for my application. Using tools like htop, I observed that PX4 was utilizing up to 100% of all four CPUs, which made it impossible to run my application alongside PX4 and jMAVSim.

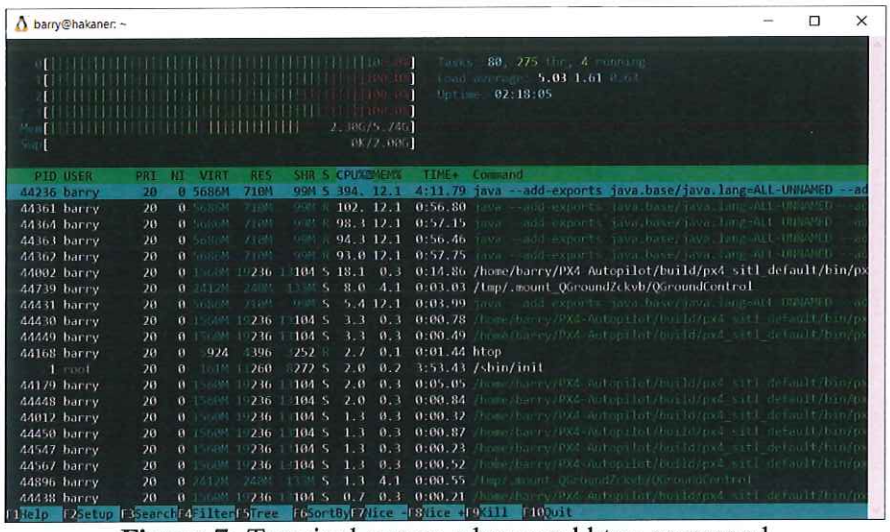

Figure 7: Terminal screen when used htop command

This insight led me to test this application with my own MAVLink message generator, which was specifically optimized to consume fewer CPU resources. With my custom MAVLink message generator in place, I was able to reduce CPU usage; however, the system still operated slowly. This prompted me to further optimize the code, beginning with the Python components. Through analysis, I realized that my application was not fully utilizing the buffer capacities during data handling processes. To address this, I modified the read_messages(connection, queue) function to read data continuously until a buffer was fully filled before pushing it into the queue. Similarly, I ensured that the write_to_shared_memory function wrote data in complete buffer-sized chunks. This approach enabled the system to handle data in bulk rather than processing messages individually. Processing data in bulk not only optimizes buffer usage but also reduces the frequency of checks by threads, which can potentially prevent race conditions and enhance overall system performance.

WORK DONE: 3.3 Testing and Optimization of the the code		Page No 24
Start Date: 21/08/2024	End Date: 22/08/2024	

On the C++ side, the initial implementation of read_and_push_buffer was already optimized to read entire buffers, but issues were identified in the convert_bytes_to_mavlink method. The main challenge was that the mavlink_parse_char function processes data byte-by-byte, which can be inefficient as it only checks one byte per loop iteration. This inefficiency caused bottlenecks in data throughput. To overcome this, I encapsulated the parsing process within a loop, iterating over all bytes in the buffer. I introduced a constant, const size_t BATCH_SIZE = 100, which allowed MAVLink messages to be batched together before being processed. This batching approach significantly improved throughput by allowing messages to be processed and sent in groups rather than individually.

Regarding the choice of the batch size, I opted for a relatively modest value of 100 to strike a balance between performance and system stability. Setting the batch size too high could introduce delays due to the time required to accumulate a large number of messages, especially given the high frequency of message generation in MAVLink communication. On the other hand, keeping the batch size too low could negate the benefits of batching by increasing the overhead associated with frequent processing. The value of 100 was selected as an optimal compromise, providing a substantial improvement in processing efficiency without overwhelming the system. The send_mavlink_messages function in the final thread then reads and sends these batched messages in bulk, further contributing to the system's overall performance improvements. These considerations resulted in the code below:

```
for (size_t offset = 0; offset < bytes.size(); ++offset) {
  // Parse each byte
  if (mavlink_parse_char(MAVLINK_COMM_0, bytes[offset], &msg, &status)) {
    // A complete MAVLink message has been successfully parsed
    batch.push_back(msg);

    // Check if the batch is full
    if (batch.size() >= BATCH_SIZE) {
      // Add the batch to the main message container
      messages.insert(messages.end(), batch.begin(), batch.end());
      batch.clear();  // Clear the batch
    }
  }
```

These optimization efforts were essential in addressing the challenges faced during the initial development and testing phases. The iterative process of profiling, analyzing, and adjusting the code was critical in achieving the desired performance enhancements. By focusing on buffer utilization and batching strategies, I was able to significantly reduce the CPU overhead and improve the overall efficiency of the system. Initially, the program's performance was so poor that it could barely send heartbeat messages, and even those were not being transmitted consistently once per second. The optimization efforts not only addressed the high CPU usage but also resolved the critical issue of the program's sluggishness. Through careful profiling and adjustments, the program's throughput improved, allowing it to handle a higher volume of messages effectively.

WORK DONE: 3.3 Testing and Optimization of the the code		Page No 25
Start Date: 22./.06/2024	End Date: 02./.09/2024	

Additionally, I discovered that the limited visibility of only heartbeat messages was due to incorrect header files being included in the project. I initially used standard.h, which was not suitable for the task at hand. Upon further investigation, I realized that common.h was the correct header to use, as it encompasses a broader set of MAVLink messages essential for the system's proper operation. This adjustment was crucial, as common.h provides a comprehensive set of message definitions, unlike standard.h, which is limited and resulted in restricted functionality. This experience underscored the importance of careful resource management, the correct selection of libraries, and the significant impact of optimization techniques in developing high-performance applications.

4. Simulating Telemetry Data with Randomised MAVLink Messages

In the development of telemetry systems, generating varied data is crucial for thorough testing and validation. This section of the report focuses on the random generation of MAVLink messages, a fundamental aspect of simulating telemetry data. By incorporating realistic randomness into the generated messages, we aim to mimic real-world conditions more accurately, ensuring that the system's performance can be evaluated under diverse scenarios. This approach not only enhances the reliability of the testing phase but also provides a comprehensive assessment of how the system handles different data inputs. The following discussion will delve into the methodologies employed for creating these random MAVLink messages and the significance of realistic enough data generation in the overall testing process.

First Insights for the Implementation of MAVLink Message Generation

Before developing the application for MAVLink message generation, several key considerations must be addressed. The application needs to handle a variety of MAVLink message types, focusing particularly on those that are crucial for communication and interpretation by QGroundControl (QGC). Additionally, it is essential to manage the frequency at which different messages are sent. Like PX4 simulation, which does not transmit all messages at the same frequency, the generated messages need to be categorized and dispatched at specific intervals. This approach ensures that each message type is transmitted at a frequency that reflects its importance and relevance in the telemetry data stream, optimizing the simulation's accuracy and effectiveness.

Explanation of Frequency Selection and Message Assignment

The details of the MAVLink messages and their corresponding frequencies are provided in the attached document. This attachment includes a comprehensive list of the messages categorized by the frequency at which they are transmitted.

The selection of message types and their corresponding frequencies is based on their relevance and the need to reflect real-world scenarios accurately. Lower frequency messages (1Hz) are chosen for critical status updates and essential telemetry data, which do not require rapid changes. These include heartbeat and system status messages that provide fundamental operational information.

Messages sent at 5Hz include data that changes more frequently, such as GPS and altitude information, which are crucial for ongoing navigation and system health monitoring. These messages provide more dynamic updates that are essential for real-time analysis but do not need to be updated as often as those sent at higher frequencies.
The 10Hz messages are related to specific control and sensor data, such as attitude and position measurements, which benefit from more frequent updates to ensure precision and responsiveness in the system's operation.

Finally, messages sent at 20Hz are for high-resolution data that directly impacts control and feedback mechanisms, such as actuator control targets and attitude quaternion data. These messages are sent at the highest frequency to provide the most up-to-date information necessary for fine-tuned adjustments and accurate performance monitoring.

By categorizing messages into different frequencies, we balance the system's performance with the need for accurate and timely telemetry data, ensuring both efficiency and effectiveness in the simulation and analysis processes. Athough message types and their frequencies might change according to different needs. Developed application should be easily configurable for this scenario.

Design and Implementation of Telemetry Message Generation System

The telemetry message generation system is designed to handle the real-time generation and transmission of telemetry data efficiently. The resulting program flow will be given at the end while core requirements and the corresponding design choices are outlined below:

1. Handling Multiple Message Frequencies
Requirement:

- The system needs to generate telemetry messages at various frequencies to simulate different types of data updates.

Design Solution:

- The system employs multiple threads, each dedicated to generating messages at a specific frequency. These threads are called "producer threads" beacuse they are producing the messages. By using separate threads for different frequencies, the system can produce messages continuously and efficiently without blocking.

```
std::vector<std::vector<uint8_t>> messages =
messageCreator.createMessagesFor1Hz();
```

In each producer thread, the line std::vector<std::vector<uint8_t>> messages = messageCreator.createMessagesFor1Hz(); is used to call the appropriate message creation function for the specified frequency. This line retrieves a vector of messages to be enqueued, ensuring that each thread generates and processes messages at its designated rate (e.g., 1Hz, 5Hz, 10Hz, or 20Hz) as part of the telemetry data simulation.

```
std::thread producer1(producerThread1Hz, std::ref(queue));
```

The line above initializes a new thread for producing messages at 1Hz. This creates a separate execution context for the producerThread1Hz function, which generates messages at the specified frequency and enqueues them into the shared queue.

2. Efficient Communication Between Threads

Requirement:

- The system requires an efficient mechanism for transferring messages between producer threads and a consumer thread. This must be done in a way that avoids bottlenecks and ensures smooth data flow.

Design Solution:

- A lock-free queue is used as the communication channel between producer and consumer threads. This data structure allows multiple threads to access and modify the queue concurrently without requiring locks, which helps avoid performance degradation due to contention.

```
LockFreeQueue<std::vector<uint8_t>> queue(600);
```

The line initializes a lock-free queue with a buffer size of 600. This queue serves as the communication channel between the producer and consumer threads.

3. Synchronization of Message Generation

Requirement:

- As shown in later testing of this application to ensure data integrity, message generation processes must be synchronized. This is crucial to prevent issues such as sequence number confusion or data corruption when multiple threads are generating messages concurrently. Due to the simultaneous operation of multiple threads generating messages, there is a risk of sequence number confusion, which can lead to a packet loss rate of 3-7% as observed by the QGroundControl (QGC). QGC employs various methods to detect packet loss, one of which involves monitoring the sequence number field in the messages. When the sequence numbers are inconsistent or out of order, QGC identifies these anomalies as packet loss.

Design Solution:

- An atomic flag is employed to manage access to the message generation functions. This flag ensures that only one thread at a time can execute the message creation functions, thereby preventing concurrent access issues and ensuring consistent data generation. In the provided code snippet, the following segment ensures proper synchronization when generating messages at different frequencies:

```
while (access_flag.load(std::memory_order_acquire)) {
   std::this_thread::sleep_for(std::chrono::milliseconds(1));
}
access_flag.store(true, std::memory_order_release);
std::vector<std::vector<uint8_t>> messages =
messageCreator.createMessagesFor20Hz();
message_count += messages.size();
access_flag.store(false, std::memory_order_release);
```

In the code snippet, the access_flag is used to manage access to the message generation function. The thread first checks if the access_flag is set to true, indicating that another thread is currently generating messages. If so, it waits for 1 millisecond before rechecking. Once the flag is false, the thread sets it to true to gain exclusive access for generating messages at 20Hz. After message creation, the flag is reset to false, signaling that other threads can now access the function. This approach ensures orderly access and prevents conflicts during message generation

4. Accurate Timing for Message Generation

Requirement:

- The system must generate messages at precise intervals to match the required frequencies. For instance, messages need to be generated exactly at 1Hz, 5Hz, 10Hz, and 20Hz, respectively.

Design Solution:

- Each producer thread calculates the time taken for message processing and adjusts its sleep duration accordingly to meet the target frequency. This approach ensures that messages are generated at the correct intervals, maintaining the required data update rates.
- In the code, each producer thread starts by recording the current time with `auto start_time = std::chrono::high_resolution_clock::now();`. It then proceeds to generate messages at a specific frequency, such as 10Hz, using `messageCreator.createMessagesFor10Hz()`.
  After generating the messages, the time taken for processing the messages is then calculated with `auto process_end_time = std::chrono::high_resolution_clock::now();`, and the duration is measured to determine how long the processing took.
  To ensure that messages are generated at the target frequency, the thread calculates the `sleep_duration` required to match the desired interval. For instance, if the target frequency is 10Hz, the interval between message generations should be 0.1 seconds. The thread sleeps for the remaining time to ensure that the total cycle time matches the target interval, thus maintaining the required frequency.

5. Efficient Message Transmission

Requirement:

- The system needs to transmit generated messages to a remote system (QGroundControl) efficiently. The communication must be handled in a way that prevents excessive CPU usage and ensures timely delivery of data.

Design Solution:

- A consumer thread retrieves messages from the lock-free queue and sends them using a UDP socket. The consumer thread includes a sleep interval when the queue is empty to avoid high CPU usage during idle times. Because in profiling it is shown this thread loops incredibly fast. Time for sleeping should be considered carefully and configured in different message settings since there is 20Hz messages will be recieved from the buffer. Through testing and consideration 150-350 microsecond are deemed efficent. This setup ensures that

message transmission is handled efficiently and reduces the risk of resource overuse.

6. Network Configuration

Requirement:

- Proper network configuration is essential to ensure reliable communication between the system and QGroundControl. This includes setting up the correct socket parameters and managing network addresses.

Design Solution:

- The system initializes and configures a UDP socket with appropriate buffer sizes and sets it to non-blocking mode. The network address for QGroundControl is configured to ensure that messages are sent to the correct destination This network addresses and socket parameters are passed to consumer thread. This setup helps maintain a stable and reliable communication link.

```
std::thread consumerThread(consumer, std::ref(queue), sockfd, qgc_addr);
```

Program Flow

1. **Initialization**:
    - The main function initializes the lock-free queue and sets up the UDP socket for communication.
    - It then creates and starts four producer threads and one consumer thread.
2. **Message Generation**:
    - Each producer thread runs in a loop, generating messages at its designated frequency.
    - Messages are enqueued into the lock-free queue, and the producer threads manage timing to ensure messages are generated at the correct rate.
3. **Message Transmission**:
    - The consumer thread continuously retrieves messages from the queue and sends them to QGC.
    - The consumer thread manages network communication efficiently, adjusting for any idle time when the queue is empty.

This program flow ,aformentioned methods and ideas are implemented in msg_sender.cpp

Implementing Message Creation Functions

To facilitate the generation of telemetry messages at various frequencies, the system employs a set of specialized functions for creating different types of MAVLink messages. Each message creation function is designed to generate a specific type of telemetry data, which is then used by the producer threads to simulate real-time data updates. These methods are implemented in message_generator.cpp .

For example, the createMessagesFor5Hz function is responsible for assembling a collection of messages that are generated at a frequency of 5Hz. This function calls several internal methods, each tailored to create a different MAVLink message type, such as GPS data, altitude, or system status. By organizing message creation into discrete functions, the system ensures that each message type is generated accurately and efficiently, maintaining the desired update rates.

These functions are crucial for the overall operation of the system, as they enable the producer threads to produce diverse sets of messages, which are subsequently enqueued and transmitted to the consumer thread for further processing. The modular design of these message creation functions allows for flexibility and scalability, accommodating various data simulation needs.

To illustrate how message creation functions are structured and utilized, consider the following example:

```
std::vector<std::vector<uint8_t>> MessageCreator::createMessagesFor5Hz() {
    std::vector<std::vector<uint8_t>> messages;
    messages.push_back(createGpsRawIntMessage());
    messages.push_back(createVfrHudMessage());
    messages.push_back(createGlobalPositionIntMessage());
    messages.push_back(createAltitudeMessage());
    messages.push_back(createPositionTargetGlobalIntMessage());
    messages.push_back(createScaledPressureMessage());
    messages.push_back(createEstimatorStatusMessage());
    messages.push_back(createVibrationMessage());
    return messages;
}
```

In this example, the createMessagesFor5Hz function is responsible for generating a set of telemetry messages at a frequency of 5Hz. The function creates a std::vector of std::vector<uint8_t>, where each inner vector represents a different type of MAVLink message. The function includes several calls to specialized message creation methods—such as createGpsRawIntMessage and createAltitudeMessage—each of which generates a specific type of message.

This approach ensures that at each invocation, the function assembles a comprehensive set of messages to be produced and enqueued by the producer thread. By structuring message creation in this modular way, the system can effectively manage and simulate various data types and update rates, aligning with the requirements of the telemetry data simulation.

To implement message creation functions for MAVLink messages, it's essential to first include the necessary MAVLink headers. These headers define the structures and encoding functions required for crafting MAVLink messages. Each MAVLink message consists of a set of fields and enumarations for defining specific data, and to generate these messages accurately, we must use the specific MAVLink functions provided to handle the message encoding.

For example, consider the createHeartbeatMessage function. This function creates a MAVLink heartbeat message, which is fundamental for indicating the status of a system. The process begins by defining a mavlink_message_t structure and a mavlink_heartbeat_t structure to hold the heartbeat data. The fields within the heartbeat structure are set to reflect the type of system, the autopilot used, the base mode, and other relevant status indicators.

```
std::vector<uint8_t> MessageCreator::createHeartbeatMessage() {
    mavlink_message_t msg;
    mavlink_heartbeat_t heartbeat;

    heartbeat.type = MAV_TYPE_QUADROTOR;
    heartbeat.autopilot = MAV_AUTOPILOT_PX4;
    heartbeat.base_mode = (MAV_MODE_FLAG_GUIDED_ENABLED |
MAV_MODE_FLAG_SAFETY_ARMED);
    heartbeat.custom_mode = 0;
    heartbeat.system_status = MAV_STATE_ACTIVE;
    heartbeat.mavlink_version = 2;

    uint8_t buffer[MAVLINK_MAX_PACKET_LEN];
    uint16_t len = mavlink_msg_heartbeat_encode(1, 1, &msg, &heartbeat);
    len = mavlink_msg_to_send_buffer(buffer, &msg);
    return std::vector<uint8_t>(buffer, buffer + len);
}
```

The mavlink_msg_heartbeat_encode function encodes the heartbeat data into the msg structure, using the system ID and component ID (both set to 1 in this case). This encoding process prepares the message for transmission. The encoded message is then copied into a buffer using mavlink_msg_to_send_buffer, which converts the msg into a raw byte stream. Finally, the function returns this byte stream as a std::vector<uint8_t>.

Adding Randomness to Messages

To ensure that telemetry data is both realistic and dynamic, incorporating randomness and consistency is crucial. The following methods are some of the some of them which used to introduce variability and maintain coherence between different types of telemetry messages.

For messages where data frequently changes or where variability is beneficial, such as GPS coordinates or sensor readings, random noise is added to simulate real-world fluctuations. This is achieved using various random number generation techniques such as :

- **Uniform Distribution**: Used for generating random values within a specified range, adding variability to parameters like GPS coordinates or sensor noise.
- **Gaussian Distribution**: Applied for generating noise that follows a normal distribution, useful for creating more realistic variations around a mean value.
- **Exponential Decay**: Used to simulate values that gradually decrease over time, reflecting processes like battery discharge.
-

These methods are implemented through functions such as randomFloat, randomGaussian, and randomWalk, which adjust data to reflect realistic variations and prevent unrealistic uniformity in telemetry data.

Ensuring Consistency in Related Data

In addition to adding randomness, it's essential that related telemetry messages maintain coherence in different message types. For instance, position-related data (e.g., GPS coordinates) and velocity data is used in multiple message types for different representations and should align to avoid discrepancies in the simulated environment. This is managed by updating shared variables, such as gps_lat, gps_lon, and velocity_x, which are used across multiple message creation functions.
For example, the function createGpsRawIntMessage demonstrates how these shared variables are updated and used:

```
std::vector<uint8_t> MessageCreator::createGpsRawIntMessage() {
    mavlink_message_t msg;
    mavlink_gps_raw_int_t gps_raw_int;

    gps_lat = randomWalk(gps_lat, 1000, 408000000, 410000000);
    gps_lon = randomWalk(gps_lon, 1000, 289000000, 291000000);
    // gps_alt is updated in the altitude method

    gps_raw_int.lat = gps_lat + static_cast<int32_t>(randomGaussian(0.0, 1.0) * 1e7);
    gps_raw_int.lon = gps_lon + static_cast<int32_t>(randomGaussian(0.0, 1.0) * 1e7);
    gps_raw_int.alt = gps_alt + static_cast<int32_t>(randomGaussian(0.0, 1.0) * 1e3);

    gps_raw_int.time_usec = randomInt(0, 1000000000);
    gps_raw_int.eph = randomInt(50, 150);
    gps_raw_int.epv = randomInt(50, 150);
    gps_raw_int.vel = static_cast<int16_t>(sqrt(velocity_x * velocity_x + velocity_y *
velocity_y) * 100);
    gps_raw_int.cog = randomInt(0, 36000);
    gps_raw_int.satellites_visible = randomInt(5, 15);

    uint8_t buffer[MAVLINK_MAX_PACKET_LEN];
```

```
    uint16_t len = mavlink_msg_gps_raw_int_encode(1, 1, &msg, &gps_raw_int);
    len = mavlink_msg_to_send_buffer(buffer, &msg);
    return std::vector<uint8_t>(buffer, buffer + len);
}
```

In this example, the function updates the GPS coordinates (gps_lat, gps_lon) using random walk and Gaussian noise methods, ensuring realistic fluctuations. It then encodes these values into the MAVLink message format. The randomGaussian function adds a small amount of noise to the GPS data, enhancing the realism of the generated telemetry. Overall, the approach combines randomness with consistency to ensure that the telemetry data generated is both varied and coherent, simulating a realistic and functional environment.

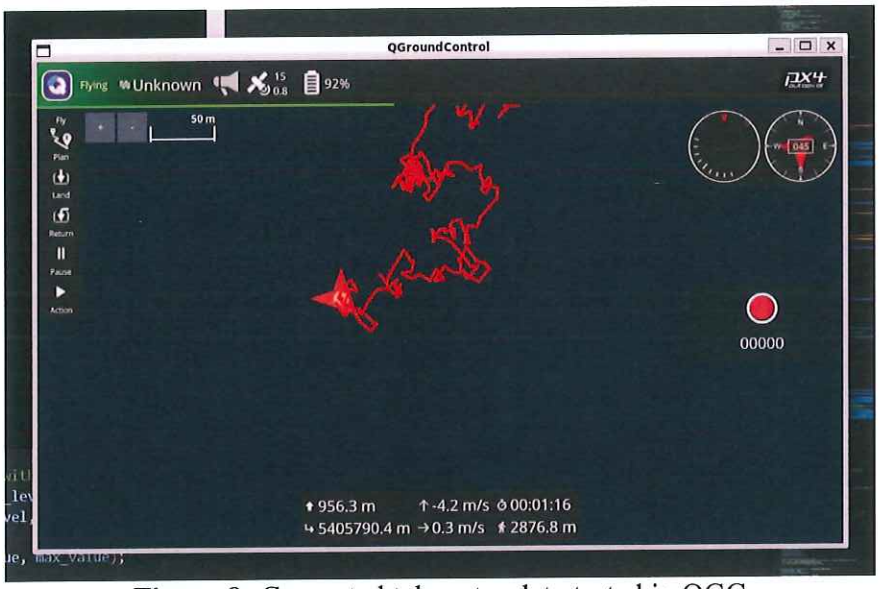

Figure 8: Generated telemetry data tested in QGC

Although the current implementation may not fully capture real-world variability, the modular implementation of the randomization functions allows for significant improvements in realism. By enhancing these functions, more accurate and realistic telemetry data can be generated, reflecting a more authentic simulation environment.

In the TABLE it is shown as data stream is not established completely resulting the bar for loading being half filled. The reason for this is QGC also needs to communicate with the source of telemetry data for exchanging commands are other user interface functionality. Nothing can be done as this application's goal is only to create and send telemetry data.

5. Reporting Project Plan and Process, Challenges and Solutions, Conclusion and Evaluation

5.1 System Overview

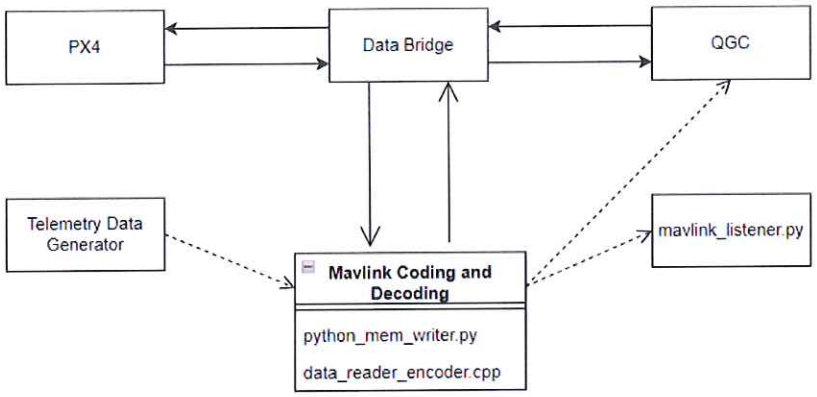

Figure 9: Diagram showing the general system overwiew

In the system depicted in the diagram above, data flows between the PX4 and QGroundControl (QGC) through a series of components and processes. The data from PX4 is received through the Data Bridge port, to MAVLink De-coding and En-coding application. This process utilizes shared memory accessible by both Python and C++ applications. The encoded MAVLink messages are then sent to the Data Bridge port, which forwards them to QGC.

Conversely, data from QGC is routed back to PX4 through the Data Bridge port, ensuring bidirectional communication. For scenarios where QGC is not used or when testing message transmission, the mavlink_listener.py script can be employed to directly decode incoming messages. This serves as a tool for verifying message integrity and communication in the absence of QGC.

Additionally, the msg_sender.cpp component can be utilized to generate custom telemetry data replacing PX4, which can be directed to QGC or the Data Bridge port or the port that python_mem_writer.py listens for further processing. This setup provides flexibility for testing and validating the system's functionality under various conditions.

Implemented scripts through the report are listed below:

data_bridge.py
mavlink_listener.py
python_mem_writer.py
data_reader_encoder.cpp
msg_sender.cpp
message_generator.cpp
LockFreeQueue.h

WORK DONE: 5.2 Areas for Improvement		Page No 36
Start Date: 09./09/2024	End Date: 09./09/2024	

5.2 Areas for Improvement

Two areas of potential improvement have been identified in the current system:

1. **Frequency Stability in msg_sender.cpp**: The system occasionally experiences minor variations in the frequency of messages generated by msg_sender.cpp especially in messages sent with higher frequency . This issue is likely attributable to small delays encountered during message retrieval from the queue. To mitigate this problem and ensure more stable message frequencies, it is suggested that the calculation of the remaining time after message creation be incorporated into the system. This would involve updating the queue with the remaining time, allowing the consumer thread to adjust its waiting period accordingly. Such a modification could enhance frequency consistency; however, it has not yet been tested.
2. **Simplification of Execution**: Given that the system's components can operate together seamlessly, providing a CMake configuration file alongside automated build and run scripts would significantly simplify user interaction. This would replace the current need for manual terminal commands to run each application, thereby improving usability and efficiency.

Conclusion

During this internship, I gained substantial knowledge and skills in developing applications for telemetry data handling and communication using Python and C++. My programming skills and debugging skills are increased .The project involved the integration of various components to facilitate the flow of data between PX4 and QGroundControl (QGC) then creating my own telemetry data generator. Key aspects included the encoding and decoding of MAVLink messages, efficient data handling through shared memory, code optimization for high bandwidth systems and the implementation of a flexible data bridge system.

Throughout the project, I worked with several scripts and modules, each contributing to the overall functionality. The data_bridge.py script played a central role in managing the data transfer between the PX4 and QGC, serving as a conduit for all communication. Through this, I learned the working principles of sockets and was introduced to network communication concepts, which are fundamental to real-time data exchange. The mavlink_listener.py provided a mechanism to decode incoming messages directly, offering an alternative means of monitoring and validating message integrity, which was especially useful when QGC was not in use.

The python_mem_writer.py and data_reader_encoder.cpp components were integral to data processing, utilizing shared memory for efficient communication between Python and C++ applications. Developing these components taught me valuable lessons in thread managing, asynchronous programming, debugging, optimizing code for performance, minimizing CPU usage, and managing data flow efficiently.

The msg_sender.cpp was developed to simulate telemetry data generation, replacing PX4 in scenarios requiring custom data inputs. This involved understanding and implementing message frequency management, synchronization, and randomness to create realistic telemetry data. The message_generator.cpp and LockFreeQueue.h were crucial in ensuring that message creation and data flow were managed efficiently, maintaining the required performance standards. It thought me use of different queue methods and managing concurrent processes.
Overall, this internship allowed me to apply programming knowledge in practical scenarios, enhancing my skills in systems integration, real-time data processing, and efficient communication protocols. I learned the importance of modularity and flexibility in system design, which facilitated the testing and extension of the project's functionality. The experience has not only deepened my technical expertise but also highlighted areas for potential improvement, such as enhancing message frequency stability and simplifying the execution process with CMake configurations. The hands-on experience with coding, debugging, and system optimization has been invaluable, providing a strong foundation for future projects in similar domains.